

# Betriebssysteme

**R. Thomas**

**(Stand : SS 2010)**



# Betriebssysteme

## Kapitel-Überblick

### Teil 1 : Betriebssysteme Allgemeines

- |                                      |          |
|--------------------------------------|----------|
| 1. Einführung                        | I-BS-100 |
| 2. Grundlegende Aufgaben (Überblick) | I-BS-200 |
| 3. Ausgewählte Dateisysteme          | I-BS-300 |

### Teil 2 : Beispiel eines PC-Betriebssystems - Ausgewählte Komponenten von LINUX

- |  |          |
|--|----------|
| 4. Einführung in LINUX                     | I-BS-400 |
| 5. Betriebssystemfunktionen (System Calls) | I-BS-500 |
| 6. Einige Kernel-Konzepte und -Mechanismen | I-BS-600 |
| 7. Prozessverwaltung                       | I-BS-700 |
| 8. Arbeitsspeicherverwaltung               | I-BS-800 |
| 9. Externe Datenverwaltung                 | I-BS-900 |
| 10. Interprozesskommunikation              | I-BS-A00 |
| 11. Socket-Schnittstelle                   | I-BS-B00 |

# **Betriebssysteme**

## **Teil 1 : Betriebssysteme Allgemeines**

### **Übersicht**

#### **1. Einführung**

- 1.1. Wesen und Aufgaben eines Betriebssystems
- 1.2. Klassifizierung von Betriebssystemen
- 1.3. Architektur von Betriebssystemen

#### **2. Grundlegende Aufgaben (Überblick)**

- 2.1. Prozessverwaltung
  - 2.1.1. Prozesse
  - 2.1.2. Prozess-Scheduling
  - 2.1.3. Prozess-Synchronisation
  - 2.1.4. Kommunikation zwischen Prozessen
  - 2.1.5. Verklemmung von Prozessen
- 2.2. Speicherverwaltung
  - 2.2.1. Allgemeines
  - 2.2.2. Verwaltung des realen Arbeitsspeichers
  - 2.2.3. Virtueller Speicher
- 2.3. Externe Datenverwaltung
- 2.4. Geräteverwaltung
- 2.5. Benutzerverwaltung

#### **3. Ausgewählte Dateisysteme**

- 3.1. Logische Strukturierung von PC-Festplatten
- 3.2. FAT12/16-Dateisystem von MS-DOS/WINDOWS
- 3.3. VFAT-Dateisystem von WINDOWS
- 3.4. FAT32-Dateisystem von WINDOWS
- 3.5. NTFS von WINDOWS NT
- 3.6. LINUX Extended 2 / Extended 3

# Betriebssysteme

## Teil 2 : Beispiel eines PC-Betriebssystems - Ausgewählte Komponenten von LINUX

### Übersicht

#### 4. Einführung in LINUX

- 4.1. Von UNIX zu LINUX
- 4.2. Wesentliche Eigenschaften
- 4.3. Architektur
- 4.4. Bootvorgang
- 4.5. Run-Level

#### 5. Betriebssystemfunktionen (System Calls)

- 5.1. Implementierung
- 5.2. Überblick über System Calls und Fehlercodes
- 5.3. Beispiele für System Calls

#### 6. Einige Kernel-Konzepte und -Mechanismen

- 6.1. Listenverwaltung
- 6.2. Warteschlangen
- 6.3. Synchronisation im Kernel
- 6.4. Interruptbearbeitung

#### 7. Prozessverwaltung

- 7.1. Allgemeines
- 7.2. Prozess-Zustände
- 7.3. Prozess-Identifikation
- 7.4. Datenstrukturen zur Prozessverwaltung
- 7.5. Erzeugung und Beendigung von Prozessen
- 7.6. Scheduling

#### 8. Arbeitsspeicherverwaltung

- 8.1. Allgemeines
- 8.2. Virtueller Arbeitsspeicher eines Prozesses
- 8.3. Verwaltung des physikalischen Speichers
- 8.4. Paging

#### 9. Externe Datenverwaltung

- 9.1. Virtuelles Dateisystem (VFS)
- 9.2. Datenstrukturen zur externen Datenverwaltung
- 9.3. Registrierung und Mounten von Dateisystemen
- 9.4. System Calls zur externen Datenverwaltung

#### 10. Interprozesskommunikation

- 10.1. Überblick
- 10.2. Signale
- 10.3. Pipes
- 10.4. Fifos (Named Pipes)
- 10.4. System V IPC

#### 11. Socket-Schnittstelle

- 11.1. Überblick über das Socket-API
- 11.2. Unix Domain Sockets
- 11.3. INET Sockets

## Betriebssysteme - Literaturhinweise

- ( 1) Rüdiger Brause  
**Betriebssysteme Grundlagen und Konzepte**  
Springer Verlag
- ( 2) A.S. Tanenbaum  
**Moderne Betriebssysteme**  
2. überarbeitete Auflage 2002  
Prentice Hall / Pearson Studium
- ( 3) William Stallings  
**Betriebssysteme – Prinzipien und Umsetzung**  
Prentice Hall/Pearson Studium
- ( 4) Volkmar Richter  
**Grundlagen der Betriebssysteme**  
Fachbuchverlag Leipzig / Carl Hanser Verlag
- ( 5) Albrecht Achilles  
**Betriebssysteme**  
Springer-Verlag
- ( 6) Erich Ehses u.a.  
**Betriebssysteme**  
Pearson Studium
- ( 7) Robert Love  
**Linux-Kernel-Handbuch**  
Addison Wesley / Pearson Education
- ( 8) Wolfgang Mauerer  
**Professional Linux Kernel Architecture**  
Wiley Publishing Inc
- ( 9) Daniel P. Bovet, Marco Cesati  
**Understanding the LINUX Kernel**  
O'Reilly
- (10) Scott Maywell  
**LINUX Core Kernel Commentary**  
Coriolis Open Press
- (11) Michael K. Johnson, Erik W. Troan  
**Linux Application Development**  
Addison Wesley Longman
- (12) Allesandro Rubini, Jonathan Corbet  
**LINUX Device Drivers**  
O'Reilly
- (13) Helmut Herold  
**Linux – Unix – Kurzreferenz**  
Addison Wesley Longman
- (14) Michael Kofler, Jürgen Plate  
**Linux für Studenten**  
Pearson Studium

# **Betriebssysteme**

## **Kapitel 1**

### **1. Einführung**

- 1.1. Wesen und Aufgaben eines Betriebssystems
- 1.2. Klassifizierung von Betriebssystemen
- 1.3. Architektur von Betriebssystemen

## Wesen und Aufgaben eines Betriebssystems

- **Betriebssystem (BS) = Operating System (OS)**

Ein **Betriebssystem** umfaßt die **Programme** eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechanlage die **Basis der möglichen Betriebsarten** des digitalen Rechensystems bilden und die insbesondere die **Abwicklung von Programmen steuern und überwachen**. (nach DIN 44300)

- **"Bottom-Up"-Sicht**

Ein **Betriebssystem** ist eine **Sammlung von Programmen und Routinen** zur geregelten **Verwaltung und Benutzung** der **Betriebsmittel** eines Datenverarbeitungssystems (nach Wettstein)

- ◇ **Betriebsmittel ("Ressourcen") :**

- **physikalische Betriebsmittel** (Funktionseinheiten des DVS, Hardware-Komponenten)  
z.B. CPU, Arbeitsspeicher, externe Speichergeräte, Ein-/Ausgabe-Geräte
- **logische Betriebsmittel** (Software-Komponenten)  
z.B. Datendateien, Programmbibliotheken, Dienstprogramme

- **"Top-Down"-Sicht**

Ein **Betriebssystem** bildet ein **Interface** zwischen dem **Benutzer** und der **Hardware** eines DVS, das die **Komplexität der Hardware** vor dem Benutzer **verbirgt**.  
Es stellt die **Dienste der Hardware** in einer **abstrakten** Form – als Dienste einer **virtuellen Maschine** – zur Verfügung  
Der Benutzer wird dadurch in die Lage versetzt, **ein DVS ohne genaue Hardwarekenntnisse sinnvoll und flexibel zu nutzen**

- **Hauptaufgaben eines Betriebssystems :**

- ◇ **Verbergen der Komplexität der DVS-Hardware vor dem Benutzer** (Abstraktion)
- ◇ **Bereitstellen einer Benutzerschnittstelle** (Shell, Kommandointerpreter, Graphical User Interface (GUI))
- ◇ **Bereitstellen einer Programmierschnittstelle** (BS-Funktionen, Application Programming Interface, API)
- ◇ **Prozessverwaltung**
- ◇ **Prozessorverwaltung**
- ◇ **Arbeitsspeicherverwaltung**
- ◇ **externe Datenverwaltung** (Verwaltung der Hintergrundspeicher, wie Platte, Band usw)
- ◇ **Geräteverwaltung** (Verwaltung der E/A-Geräte, wie Terminal, Drucker, Scanner, Modems usw)
- ◇ **Benutzerverwaltung**
- ◇ in einer erweiterten Betrachtungsweise : **Bereitstellen von Dienstprogrammen** (Editor, Übersetzer usw)

## Klassifizierung von Betriebssystemen

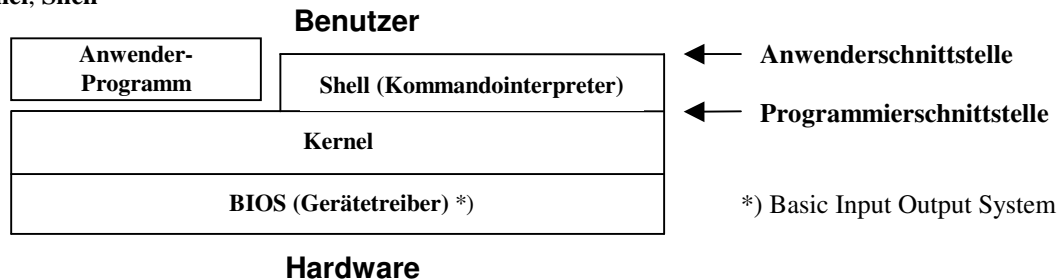
- **Klassifizierung nach dem Einsatzbereich des Betriebssystems**
  - ◇ **Mainframe-Betriebssystem** (*mainframe os*)
  - ◇ **Server-Betriebssystem** (*server os*)
  - ◇ **PC-Betriebssystem** (*personal computer os*)
  - ◇ **Prozessrechner-Betriebssystem**
  - ◇ **PDA-Betriebssystem** (*personal digital assistant os, handheld os*)
  - ◇ **Betriebssystem für eingebettete Systeme** (*embedded systems os*)
  - ◇ **Betriebssystem für Chipkarten** (*smart card os*)
- **Klassifizierung nach der Betriebsart des DVS**
  - ◇ **Stapelverarbeitungs-Betriebssystem** (*batch processing os*)
  - ◇ **Dialog-Betriebssystem** (*interactive processing os, dialog processing os*)
  - ◇ **Echtzeit-Betriebssystem** (*real time processing os*)
  - ◇ **Universelles Betriebssystem** (unterstützt mehrere Betriebsarten)
- **Klassifizierung nach der Anzahl der gleichzeitig laufenden Programme :**
  - ◇ **Einzelprogramm-Betriebssystem** (*single tasking os*)
  - ◇ **Mehrprogramm-Betriebssystem** (*multi tasking os, multi programming os*)
- **Klassifizierung nach der Anzahl gleichzeitig am Rechner arbeitender Benutzer :**
  - ◇ **Einzelbenutzer-Betriebssystem** (*single user os*)
  - ◇ **Mehrbenutzer-Betriebssystem** (*multi user os*)
- **Klassifizierung nach der Anzahl der verwalteten Prozessoren bzw Rechner :**
  - ◇ **Einprozessor-Betriebssystem** (*single processor os*)
  - ◇ **Mehrprozessor-Betriebssystem** (*multi processor os*)
  - ◇ **Mehrrechner-Betriebssystem** (*multi computer os, distributed system os*)
    - ▷ **Netzwerk-Betriebssystem** (*network os*)
    - ▷ **Verteiltes Betriebssystem** (*distributed os*)
- **Zielfunktionen von Betriebssystemen** (Strategien zur Erfüllung der Aufgaben) :
  - ◇ **Auslastung der Betriebsmittel**
    - ▷ **maximale** Auslastung **bestimmter** Betriebsmittel (z.B. CPU)
    - ▷ **gleichmäßige** Auslastung **aller** Betriebsmittel
    - ▷ ökonomisch **optimale** Auslastung **aller** Betriebsmittel
  - ◇ **Erfüllung von Nutzerforderungen**
    - ▷ **maximaler Auftragsdurchsatz** (z.B. bei Stapelverarbeitung)
    - ▷ Einhaltung **maximal zulässiger Reaktionszeiten** (Echtzeit-BS) bzw **Antwortzeiten** (Dialog-Betrieb)



## Architektur von Betriebssystemen

### • Schichtenmodell :

- ◇ Zur **logischen Strukturierung** ist ein Betriebssystem heute üblicherweise in **hierarchische Schichten** (bzw **Schalen**) unterteilt.
- ◇ Die **Anzahl der Schichten** (Schalen) hängt vom jeweiligen Betriebssystem ab. Sie beträgt aber i.a. **wenigstens drei** : BIOS, Kernel, Shell



- ◇ Jede Schicht bildet eine **abstrakte (virtuelle) Maschine**, die – bei konsequenter Realisierung des Schichtenmodells – nur mit ihren beiden direkt benachbarten Schichten über wohldefinierte **Schnittstellen** kommunizieren kann. Sie kann Funktionen der nächstniedrigeren Schicht aufrufen und stellt ihrerseits ihre Funktionen der nächsthöheren Schicht zur Verfügung.  
Die Gesamtheit der von einer Schicht an ihrer oberen Schnittstelle angebotenen Funktionen werden auch als **Dienste** dieser Schicht bezeichnet.  
Die Gesamtheit der Vorschriften (syntaktische, logische und physische Festlegungen), die bei der Nutzung der Dienste einer Schicht einzuhalten sind, nennt man das **Protokoll** der Schnittstelle.
- ◇ Die **unterste Schicht** setzt direkt auf der **Rechner-Hardware** auf. Sie beinhaltet alle **hardwareabhängigen Teile** des Betriebssystems. Häufig wird diese Schicht **BIOS** (Basic Input Output System) genannt.  
Die Funktionalität des BIOS (**Gerätetreiber**) kann aber auch auf **mehrere Schichten** aufgeteilt sein.  
Die unterste – hardwarenächste – Schicht wird häufig als **HAL** – *Hardware Abstraction Layer* bezeichnet.
- ◇ Alle **weiteren Schichten** sind **hardwareunabhängig**.
- ◇ Der **Kernel** enthält die **Betriebssystem-Funktionen "höherer Ebene"**. Diese stehen über die **Programmierschnittstelle** (**API** –Application Programming Interface) den Anwendungsprogrammen (und der Shell) zur Verfügung.  
Auch hier ist eine Aufteilung auf **mehrere Schichten** möglich.
- ◇ Die **oberste Schicht** (**Shell**, Benutzeroberfläche, Kommandointerpreter) kommuniziert mit dem Benutzer.  
Sie stellt die **Benutzerschnittstelle** (Anwenderschnittstelle) zur Verfügung.
- ◇ Bei **Singleuser-Singletasking-Betriebssystemen** kann die **Schichtenstruktur** i.a. **durchbrochen** werden.  
**Anwenderprogramme** können z.B. **direkt** die Funktionen des **BIOS** aufrufen bzw sogar **direkt** auf die **Hardware** zugreifen.  
Bei **Multisuser-** und **Multitasking-Betriebssystemen** ist dies **nicht zulässig** (und auch nicht möglich), da jeglicher unzulässiger Zugriff auf die Betriebsmittel ausgeschlossen werden muß.
- ◇ Häufig werden Kernel und BIOS – unter Beibehaltung der Schichtenstruktur – zusammenfassend als Kernel bezeichnet.  
Wenn dieser als **ein Programm** in einem **einheitlichen Adressraum** ausgeführt wird → **monolithischer Kernel**

### • Modularität :

- ◇ Betriebssysteme sind häufig **modular aufgebaut**.
- ◇ Ein **Modul** fasst eine Anzahl von **Funktionen** und die von ihnen zu manipulierenden **Daten** zu einer **Einheit** zusammen (Vorstufe von Objekten in der OOP) und läßt sich damit weitgehend **unabhängig von anderen Modulen** einsetzen.
- ◇ Modular aufgebaute Betriebssysteme lassen sich **konfigurieren**, d.h. entsprechend individuellen Anforderungen zusammenstellen (z.B. Aufnahme bestimmter Treiber in Abhängigkeit von der Hardware oder Integration bestimmter Netzwerkdienste). Gegebenenfalls können Module bei Bedarf **dynamisch hinzugefügt** und **entfernt** werden.
- ◇ **Mikrokern-Architektur** : Konsequente Nutzung der Modularität  
Der eigentliche **Betriebssystemkern** enthält nur das absolute **Minimum an Funktionalität** (Prozessverwaltung einschliesslich Interprozesskommunikation, u. U. Speicherverwaltung). Die **restliche Funktionalität** ist in **eigenständige Prozesse** ausgelagert, die mit dem Mikrokern – und den eigentlichen Benutzerprozessen – über eine **wohldefinierte Schnittstelle** (z.B. Messages) kommunizieren → **Client-Server-Modell**.

# **Betriebssysteme**

## **Kapitel 2**

### **2. Grundlegende Aufgaben (Überblick)**

#### 2.1. Prozessverwaltung

- 2.1.1 Prozesse
- 2.1.2. Prozess-Scheduling
- 2.1.3. Prozess-Synchronisation
- 2.1.4. Kommunikation zwischen Prozessen
- 2.1.5. Verklemmung von Prozessen

#### 2.2. Speicherverwaltung

- 2.2.1. Allgemeines
- 2.2.2. Verwaltung des realen Arbeitsspeichers
- 2.2.3. Virtueller Speicher

#### 2.3. Externe Datenverwaltung

#### 2.4. Geräteverwaltung

#### 2.5. Benutzerverwaltung

## Prozesse (1)

### • Definition :

Unter einem **Prozess** versteht man den **Ablauf** eines **sequentiellen Programms** bzw **Programmabschnitts** in einem Datenverarbeitungssystem.

Jeder Prozess stellt eine **Ablaufumgebung** (Prozessumgebung) zur Verfügung, in der ein Programm bzw Programmabschnitt ausgeführt wird.

Weitgehend **gleichbedeutend** wird auch der Begriff **Task** verwendet.

Es gibt Betriebssysteme, die Programme mit mehreren **parallel ausführbaren** – in sich sequentiellen – **Programmabschnitten** zulassen. Die Abarbeitung solcher Programme führt zum Ablauf mehrerer Prozesse.

Jeder derartiger Prozess wird auch als **Thread** ("Programmfaden", "leichtgewichtiger Prozess") bezeichnet.

### • Prozesseigenschaften

- ◇ Aus der Sicht des Betriebssystems sind **Prozesse** die **Objekte**, denen die **CPU-Kapazität zugeteilt** wird.  
→ Die Prozessorverwaltung ist ein Teilaspekt der Prozessverwaltung.
- ◇ Bei **Multitasking-Betriebssystemen** können sich **mehrere Prozesse gleichzeitig** im **Arbeitsspeicher** befinden.  
Von diesen kann – in einem Einprozessorsystem - jedoch immer **nur ein Prozess** jeweils **die CPU belegen**, d.h. **aktiv** (rechnend) sein. Durch einen **ständigen Wechsel** des **aktiven Prozesses** in kurzen Zeitabständen läßt sich ein **zeitlich verschachtelter quasisimultaner** (quasiparalleler) **Ablauf aller Prozesse** erreichen.
- ◇ **Parallel** (bei mehreren CPUs) bzw **quasiparallel** (bei einer CPU) ablaufende Prozesse werden auch als **nebenläufig** bezeichnet.
- ◇ Prozesse besitzen einen **Zustand**.
- ◇ Jeder Prozess besitzt seine eigene **Prozessumgebung (Kontext)** die durch den Inhalt sämtlicher **CPU-Register** (virtuelle CPU des Prozesses) und den Zustand der von ihm verwendeten **Ressourcen** (z.B. Arbeitsspeicher, offene Dateien mit ihrer jeweiligen Bearbeitungsposition usw) festgelegt ist.
- ◇ Ein Prozess kann **"Kindprozesse"** erzeugen → **Prozesshierarchie, Prozessbaum**  
Jeder Kindprozess hat genau einen Elternprozess.  
Ein Elternprozess kann mehrer Kindprozesse besitzen.
- ◇ Prozesse können miteinander kommunizieren, d.h. Informationen austauschen. → **Interprozesskommunikation**
- ◇ Prozesse können voneinander abhängen → **kooperierende Prozesse**.  
Derartige Prozesse müssen sich untereinander extern **synchronisieren**.
- ◇ Prozessen kann eine **Priorität** zugeordnet werden. Diese wirkt sich i.a. auf die Zuteilungsreihenfolge und –dauer der CPU aus. (Prozesse höherer Priorität können Prozesse niedrigerer Priorität verdrängen).

### • Repräsentation von Prozessen

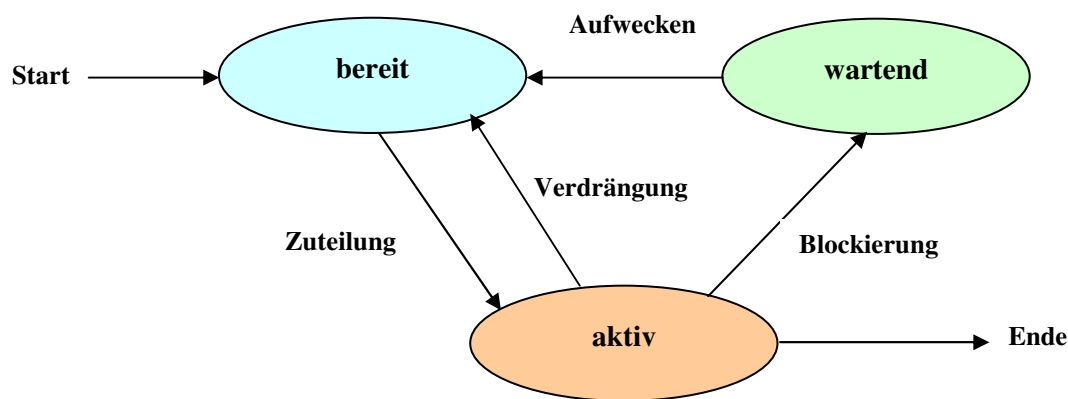
- ◇ Das Betriebssystem legt für jeden Prozess einen **Prozesssteuerblock** (Prozessleitblock, Tasksteuerblock, process control block, **PCB**) an.
- ◇ Ein Prozesssteuerblock fasst alle **für die Prozessverwaltung notwendigen Informationen** über den Prozess zusammen, u.a.
  - Prozesszustand
  - Prozesskennung (process identification, PID)
  - Prozesspriorität
  - Prozessumgebung (Kontext)
  - zugeordneter Benutzer
  - ggfls Informationen über Wartebedingungen
- ◇ Die Prozesssteuerblöcke sind in einer oder mehreren – häufig als verkettete Listen realisierte – **Prozesstabelle(n)** zusammengefaßt

## Prozesse (2)

### • Prozesszustände

- ◇ Während seiner **Abarbeitung** kann ein Prozess **unterschiedliche Zustände** einnehmen :  
Die **elementaren Zustände** sind :

- ☐ **aktiv** (rechnend, *running*) : Der Prozess belegt die CPU
- ☐ **bereit** (*ready*) : Der Prozess ist lauffähig, d.h. er verfügt über alle benötigten Betriebsmittel außer der CPU. Durch Zuteilung der CPU kann er jederzeit aktiv werden.
- ☐ **wartend** (blockiert, *waiting*) : Der Prozess wartet auf den Eintritt eines bestimmten Ereignisses



- ◇ **Zustandsübergänge** :

- ☐ **Zuteilung** (*dispatch*) : **bereit** → **aktiv**  
Die CPU wird dem Prozess zugeteilt.
- ☐ **Verdrängung** : **aktiv** → **bereit**  
Dem Prozess wird die CPU entzogen
- ☐ **Blockierung** : **aktiv** → **wartend**  
Der Prozess gibt die CPU frei, da er auf ein Ereignis wartet
- ☐ **Aufwecken** (*wakeup*) : **wartend** → **bereit**  
Das Ereignis, auf das der Prozess gewartet hat, ist eingetreten

- ◇ Für Prozesse in den Zuständen **bereit** und **wartend** wird jeweils eine eigene **Warteliste** (Warteschlange, *queue*) geführt.  
Ein **neu gestarteter Prozess** wird an das **Ende** der **Warteschlange** für den Zustand **bereit** gehängt.
- ◇ Die **Zuteilung der CPU** an einen der bereitenden Prozesse wird durch eine spezielle Betriebssystem-Komponente, den **Scheduler** realisiert.  
Es existieren mehrere **unterschiedliche Zuteilungsstrategien** bzw **-algorithmen**
- ◇ Ein **Wechsel des aktiven Prozesses** (**Prozesswechsel**, *process switch*, *task switch*) erfordert ein **Retten der Prozessumgebung** (Kontext) des **bisher aktiven Prozesses** und die **Aktivierung des Kontexts** des **neuen aktiven Prozesses**, sowie eine **Aktualisierung** der betreffenden **Prozesssteuerblöcke** und der betroffenen **Warteschlangen**.
- ◇ In Abhängigkeit von dem im Betriebssystem realisierten Prozessmodell können noch **weitere Prozesszustände** (z.B. differenzierte Wartezustände) und **Zustandsübergänge** möglich sein.

## Prozess-Scheduling (1)

### • Prozess-Scheduler (*task scheduler*)

- ◇ Eine nur in Multitasking-Systemen vorhandene Komponente der Prozessverwaltung, die aus den bereiten Prozessen den **nächsten aktiven Prozess** (→ Prozess, dem als nächstem die CPU zugeteilt wird) **auswählt**.
  - ◇ **Zeitpunkte** für eine Scheduling-Entscheidung :
    - Erzeugung eines neuen Prozesses
    - Beendigung eines Prozesses
    - Blockierung des bis dahin aktiven Prozesses
    - Unterbrechung durch ein E/A-Gerät (z.B. bei Beendigung eines angeforderten Datenverkehrs)
    - Gegebenenfalls bei Unterbrechungen durch einen Taktgeber (Timer)
  - ◇ Aufgabe eines Schedulers ist es, auf der Basis einer vorgegebenen Strategie Entscheidungen zu treffen. Seine Arbeitsweise wird durch den von ihm verwendeten **Scheduling-Algorithmus** bestimmt. Durch diesen können **unterschiedliche** – sich zum Teil widersprechende – **Ziele** verfolgt werden. Ihre Gewichtung hängt zum Teil von der Betriebsart (Stapel, Dialog, Echtzeit) ab. Einige dieser Ziele sind:
    - ▷ **Gerechtigkeit (Fairness)** : Jeder Prozess erhält einen "gerechten" CPU-Anteil
    - ▷ **Effizienz** : Optimale Auslastung der CPU und der übrigen Systemkomponenten (möglichst zu 100%) zu häufiger Prozesswechsel verbraucht zuviel "Verwaltungs"-CPU-Zeit und führt daher zur Ineffizienz
    - ▷ **Antwortzeit** : Minimale Antwortzeit in interaktiven Systemen bzw Einhalten der Antwortzeit in Echtzeitsystemen
    - ▷ **Verweilzeit** : Möglichst geringe Verweilzeit für Hintergrund-Prozesse
    - ▷ **Durchsatz** : Abarbeitung möglichst vieler Aufträge pro Zeiteinheit
    - ▷ **Terminerverfüllung** : Bereitstellung bestimmter Ergebnisse zu festgelegten Terminen.
  - ◇ Der vom Scheduler realisierte Algorithmus sollte u.a. auch das **Prozessverhalten** bezüglich der Häufigkeit von E/A-Anfragen berücksichtigen. Man unterscheidet :
    - ▷ **berechnungslastige Prozesse** (CPU-lastige Prozesse) : Sie wenden die meiste Zeit für Berechnungen auf, d.h. sie haben typischerweise lange CPU-Nutzungszeiten, die von wenigen E/A-Operationen unterbrochen werden.
    - ▷ **E/A-lastige Prozesse** : Sie führen sehr viele E/A-Operationen durch, zwischen denen immer nur relativ kurze CPU-Nutzungszeiten liegen, d.h. sie warten die meiste Zeit auf die Erledigung von E/A-Operationen.
- Da die CPUs – auch im Vergleich zur Arbeitsgeschwindigkeit von Festplatten – immer schneller werden, werden die Prozesse immer abhängiger von der E/A. Das heißt das Scheduling von E/A-lastigen Prozessen wird immer wichtiger. Einem rechenbereiten Prozess, der eine E/A durchführen möchte, sollte sehr schnell die CPU zugeteilt werden. Nach kurzer CPU-Belegung wird der Prozess die E/A-Anforderung durchführen und die CPU wieder freigeben. Diese kann dann einem anderen Prozess zugeteilt werden. Dadurch werden sowohl die CPU als auch die Platte besser ausgelastet. Problem : Wie soll der Scheduler erfahren, ob ein rechenbereiter Prozess E/A durchführen möchte ?

### • Grundsysteme für das Scheduling :

- ◇ **kooperatives Multitasking** (*non preemptive*)  
Der aktive Prozess gibt die CPU von sich aus frei (freiwillig oder weil er blockiert wird).  
Nur geringer Verwaltungsaufwand, Gefahr der Blockierung durch unkooperative Prozesse
- ◇ **verdrängendes Multitasking** (*preemptive*)  
Dem aktiven Prozess wird die CPU durch den Scheduler entzogen (z.B. bei Ablauf der Zeitscheibe, Timer-Interrupt !)  
Erhöhter Verwaltungsaufwand, keine Blockierung durch unkooperative (fehlerhafte) Prozesse.  
**Anstoß für einen Prozesswechsel durch Verdrängung :**
  - Ablauf von Zeitscheiben → **zeitgesteuerte** Strategien
  - Eintritt von Ereignissen → **ereignisgesteuerte** Strategien (Prozess höherer Priorität wird rechenbereit)

## Prozess-Scheduling (2)

- **Einige Scheduling-Strategien** (es gibt noch zahlreiche weitere Strategien)

- ◇ **Eingangsreihenfolge** (*first come first served, FCFS*)

Zuteilung der CPU in der Start-Reihenfolge der Prozesse. Alle rechenbereiten Prozesse bilden eine einzige Warteschlange. Jeder neu gestartete Prozess wird an das Ende der Warteschlange gehängt. Bei einem Prozesswechsel wegen Beendigung oder Blockierung des aktiven Prozesses wird dem am Anfang der Warteschlange stehenden Prozess die CPU zugeteilt. Ein nach Blockierung wieder rechenbereit werdender Prozess kommt ebenfalls an das Ende der Warteschlange. Der jeweils aktive Prozess kann die CPU bis zu seiner Beendigung oder bis er blockiert wird belegen. Es findet keine Verdrängung des aktiven Prozesses statt.

Gute Systemauslastung (geringe Verluste durch Prozesswechsel),

schlechtes Antwortzeitverhalten (kurz laufende Prozesse können von langlaufenden stark verzögert werden),  
einfache Implementierung

- ◇ **Kürzester Auftrag zuerst** (*shortest job first, SJF*)

Zuteilung der CPU nach der – bekannten oder geschätzten – Rechenzeit der Prozesse. Kürzer laufende Prozesse werden vor länger laufenden ausgeführt → kürzeste mittlere Verweilzeit, gut für Batch-Systeme geeignet.

Modifikation : **Shortest Remaining Time Next (SRTN)**

Auch für preemptives Multitasking geeignet.

Bei jeder Scheduling-Entscheidung wählt der Scheduler den Prozess aus, dessen **verbleibende** Rechenzeit am kürzesten ist. Hier muß die Rechenzeit eines Prozesses im voraus bekannt sein.

- ◇ **Zeitscheibenverfahren** (*round robin, RR*)

Die CPU wird zyklisch nacheinander allen bereiten Prozessen für einen bestimmten Zeitabschnitt (Zeitscheibe, *time slice*) zugeordnet (in der Reihenfolge der Warteschlange).

Nach Ablauf seiner Zeitscheibe wird der aktive Prozess verdrängt und an das Ende der Warteschlange gehängt, alle Prozesse haben die gleiche Priorität. → Gleichmäßige Aufteilung der Betriebsmittel auf alle Prozesse, kurze Antwortzeiten bei kurzen Zeitscheiben, aber erhöhte Verluste durch Prozesswechsel → Kompromiss

- ◇ **Prioritätssteuerung :**

Jedem bereiten Prozess wird eine Priorität zugeordnet. Vergabe der CPU in absteigender Priorität.

Ein Prozess niedrigerer Priorität kann die CPU erst erhalten, wenn alle Prozesse höherer Priorität abgearbeitet sind.

Ein bereit werdender Prozess höherer Priorität verdrängt einen aktiven Prozess niedrigerer Priorität.

Alle Prozesse gleicher Priorität werden häufig in jeweils einer eigenen Warteschlange geführt (→ **Prioritätsklassen**)

Realisierung **mehrerer unterschiedlicher Verfahren**, z. Tl miteinander u/o mit anderen Strategien kombiniert, z.B.:

- ▷ **Reine Prioritätssteuerung**

Prozesse gleicher Priorität werden nach dem FCFS-Prinzip abgearbeitet (z.B. in Echtzeit-BS)

- ▷ **Prioritätssteuerung mit unterlagertem Zeitscheibenverfahren**

Prozesse gleicher Priorität werden nach dem Zeitscheibenverfahren abgearbeitet

- ▷ **Dynamische Prioritätsänderung**

Allmähliche Erhöhung der Priorität der auf die CPU wartenden Prozesse

bzw Erniedrigung der Priorität des gerade ausgeführten Prozesses nach jedem Timer-Tick

- ▷ **Mehrstufiges Herabsetzen** (*multilevel feedback, MLF*)

Festlegung einer maximalen Rechenzeit für jede Prioritätsstufe. Hat ein Prozess die max. Rechenzeit seiner Priorität verbraucht, bekommt er die nächstniedrigere Priorität, bis er die niedrigste Stufe erreicht hat.

Je niedriger die Prioritätsstufe ist, desto länger ist die für sie festgelegte maximale Rechenzeit.

- ◇ **Lotterie-Scheduling :**

Bei jeder Scheduling-Entscheidung wird der als nächstes laufende Prozess **zufällig** ausgewählt.

Eine Prozess-Priorität kann dadurch berücksichtigt werden, dass höherprioritäre Prozesse eine ihrer Priorität entsprechende höhere Auswahl-Chance bekommen (durch entsprechend mehrmalige Berücksichtigung bei der Auswahl)

- **Abhängigkeit nebenläufiger Prozesse**

- ◇ **Kooperierende nebenläufige Prozesse** müssen wegen der zwischen ihnen vorhandenen **Abhängigkeiten** miteinander **synchronisiert** (koordiniert) werden.
- ◇ Prinzipiell lassen sich **zwei Klassen von Abhängigkeiten** unterscheiden :
  - ▶ **Die Prozesse konkurrieren um die Nutzung gemeinsamer** - exklusiv nutzbarer – **Betriebsmittel** (*race condition*)  
Beispiel : Zwei Prozesse greifen verändernd zu gemeinsamen Daten zu  
Der **Zugriff** zu den gemeinsamen Daten muß **koordiniert** werden, um eine Inkonsistenz der Daten zu vermeiden  
→ **Sperrsynchronisation** (gegenseitiger Ausschluss, *mutual exclusion*), **äußere Abhängigkeit**
  - ▶ **Die Prozesse sind voneinander datenabhängig** .  
Beispiel : Ein Prozess erzeugt Daten, die von einem anderen Prozess weiter bearbeitet werden sollen.  
Es muß eine **bestimmte Abarbeitungsreihenfolge** entsprechender Verarbeitungsschritte eingehalten werden  
→ **Zustands- oder Ereignissynchronisation** (z.B. Produzenten–Konsumenten–Synchronisation), **innere Abhängigkeit**

- **Semaphore**

- ◇ Zur Realisierung einer Synchronisation bei beiden o.a. Abhängigkeitsklassen werden häufig **Semaphore** eingesetzt
- ◇ Ein **Semaphor** (Sperrvariable, Steuervariable) signalisiert einen **Zustand** (Belegungszustand, Eintritt eines Ereignisses) und **gibt** in Abhängigkeit von diesem Zustand den **weiteren Prozessablauf frei** oder versetzt den betreffenden Prozess in den **Wartezustand**.  
**Semaphore für den gegenseitigen Ausschluss** sind dem jeweiligen exklusiv nutzbaren Betriebsmittel zugeordnet und verwalten eine **Warteliste** für dieses Betriebsmittel. Sie sind allen Prozessen zugänglich.  
**Semaphore für die Ereignissynchronisation** zweier voneinander datenabhängiger Prozesse sind diesen Prozessen direkt zugeordnet. Sie dienen zur **Übergabe einer Meldung** über das Ereignis zwischen den Prozessen.
- ◇ Zur **Manipulation und Abfrage von Semaphoren** existieren **zwei unteilbare** – d.h. nicht unterbrechbare – **Operationen** (Semaphor **s**) :

- **P-Operation** (*Down-Operation*) : Anfrage-Operation (P = *Passeer* (holländ.) = Passieren)

```

if (s>0)
    s=s-1; /* Prozess kann weiterlaufen , Zugriff für andere Prozesse wird gesperrt */
else
{ der die P-Operation ausführende Prozess → wartend;
  Eintrag des Prozesses in die vom Semaphor verwaltete Warteliste;
}

```

- **V-Operation** (*Up-Operation*): Freigabe Operation (V = *Verlaat* (holländ.) = Verlassen)

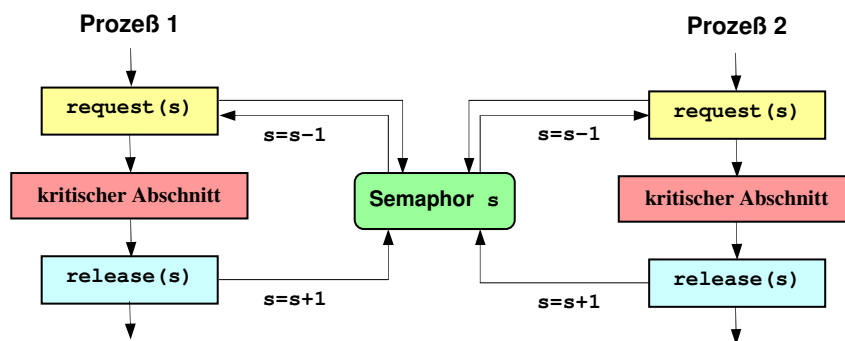
[illegible]

## Prozess-Synchronisation (2)

### • Gegenseitiger Ausschluss :

- ◇ Der Programmabschnitt, in dem ein Zugriff zu dem nur exklusiv nutzbaren Betriebsmittel (z.B. die gemeinsamen Daten) erfolgt, wird **kritischer Abschnitt** genannt.
- ◇ Es muss verhindert werden, daß sich zwei Prozesse gleichzeitig in ihren kritischen Abschnitten befinden.
- ◇ **Lösungsmöglichkeit** mittels **Semaphor** (Vorbesetzung Semaphor **s=1**) :

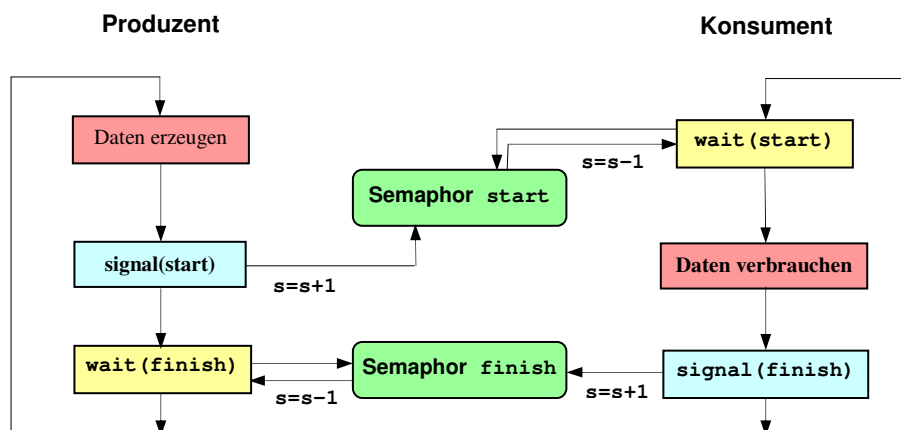
```
unkritischer Abschnitt;
request (s);      /* Anfrage-Operation (P-Operation) */
kritischer Abschnitt;
release (s);      /* Freigabe-Operation (V-Operation) */
unkritischer Abschnitt;
```



### • Ereignissynchronisation

- ◇ Annahme : Semaphor **event**  
 BS-Funktion **signal(event)** : Meldung über Eintritt des Ereignisses →  $event=1$ ; (V-Operation)  
 BS-Funktion **wait(event)** : Warten auf Meldung; nach Empfang der Meldung →  $event=0$ ; (P-Operation)
- ◇ Lösungsmöglichkeit für Produzenten-Konsumenten-Synchronisation :  
 (Vorbesetzung Semaphore  $start=0$ ;  $finish=0$ ):

<b>Produzent :</b> while (TRUE) { while (!buffer_full) write_into_buffer(); <b>signal(start)</b> ; /* V-Op. */ <b>wait(finish)</b> ; /* P-Op. */ }	<b>Konsument :</b> while (TRUE) { <b>wait(start)</b> ; /* P-Op. */ while(!buffer_empty) read_from_buffer(); <b>signal(finish)</b> ; /* V-Op. */ }
---	--





## Kommunikation zwischen Prozessen

- In Multitasking-System werden **Mechanismen für den Informationsaustausch zwischen Prozessen** bereitgestellt.  
→ Prozesse können miteinander kommunizieren → **Interprozess-Kommunikation** (*interprocess communication, IPC*)

U.a. beruht auch die Prozess-Synchronisation auf einer Kommunikation der beteiligten Prozesse (z.Tl. indirekt über das Betriebssystem)

- **Einige Möglichkeiten für die Interprozesskommunikation**

- ◇ Kommunikation über **gemeinsame Speicherbereiche**  
Prozesse können dort bestimmte Variable, Pufferbereiche usw anlegen und gemeinsam nutzen
- ◇ Kommunikation über **gemeinsame Dateien**  
Prozesse schreiben in Dateien, die von anderen Prozessen gelesen werden können
- ◇ Kommunikation über **Pipes**  
Eine Pipe (oder Pipeline) ist ein unidirektionaler Kommunikationskanal zwischen zwei Prozessen :  
Ein Prozess schreibt Daten in den Kanal (Anfügen am Ende), der andere Prozess liest die Daten in derselben Reihenfolge wieder aus (Entnahme am Anfang).  
Realisiert werden Pipes entweder im Arbeitsspeicher oder als spezielle Dateien. Von den beteiligten Prozessen werden sie i.a. wie Dateien behandelt.  
Die Lebensdauer einer Pipe ist i.a. an die Lebensdauer der beteiligten Prozesse gebunden.
- ◇ Kommunikation über **Signale**  
Signale sind asynchron auftretende Ereignisse, die eine Unterbrechung bewirken  
Typischerweise werden sie zur Kommunikation zwischen Betriebssystem und Prozess eingesetzt
- ◇ Kommunikation über **Nachrichten** (Botschaften, *message passing*)  
Nachrichten werden **vom BS verwaltet**. Dieses stellt eine den beteiligten Prozessen gemeinsam zugängliche **Transportinstanz** (z.B. **Mailbox**) zur Verfügung. Zu dieser greifen die an der Kommunikation beteiligten Prozesse mittels **Transport-Operationen** (BS-Funktionen) zu. Der Prozess A sendet z.B. eine Nachricht an den Prozeß B, indem er diese in der Transportinstanz ablegt (**send (message)**). Der Prozess B holt die für ihn bestimmte Nachricht dann von der Transportinstanz ab (**receive (message)**).  
Es sind zahlreiche Ausprägungen und Varianten dieses Mechanismus implementiert.
- ◇ Kommunikation über **Sockets**  
Sockets ermöglichen eine Kommunikation über **Rechnernetze**.
- ◇ Kommunikation über **Prozedurfernaufrufe** (*remote procedure call, RPC*)  
Ein Prozess ruft eine in einem anderen Prozess angesiedelte Prozedur – also über seine Adressgrenzen hinweg – auf.  
Besonders für Client-Server-Beziehungen geeignet
- ◇ **Weiterentwicklungen von RPC**, insbesondere in objektorientierten verteilten Systemen (z.B. CORBA, JAVA-RMI)

## Verklemmungen von Prozessen

Unter einer **Verklemmung** (*deadlock*) versteht man die **gegenseitige** (allg. : zyklische) **Blockade** von wenigsten zwei Prozessen. Jeder der beteiligten Prozesse wartet auf ein Ereignis (Zustandsänderung, Bedingung), das nur ein anderer der beteiligten Prozesse auslösen kann und das – da dieser Prozess auch wartet – nie eintreten kann.

- **Typische Ursache für eine Verklemmung :**

**Zwei Prozesse warten** jeweils auf die **Zuteilung** eines nur **exklusiv nutzbaren Betriebsmittels**, das aber gerade von dem **jeweils anderen Prozess belegt** ist. → **beide Prozesse** bleiben ständig im Zustand **"wartend"**.

**Beispiel :** Prozess A sammelt Daten, aktualisiert mit diesen Einträge in einer Datei und protokolliert das auf dem Drucker  
Prozess B gibt die gesamte Datei auf dem Drucker aus.

Beide Betriebsmittel (Datei und Drucker) sind jeweils mit gegenseitigem Ausschluss (Semaphor) geschützt.

Prozess A belegt die Datei und aktualisiert einen Eintrag

Prozess B belegt den Drucker und gibt eine Überschrift aus

Prozess B fordert die Datei zum Ausdrucken an → da Datei von A belegt ist, wartet B

Prozess A fordert den Drucker zur Protokollierung an → da Drucker von B belegt ist, wartet A

→ **gegenseitige Blockierung !**

- **Bedingungen für das Auftreten einer Verklemmung :**

- ◇ **Exklusive Nutzung** : Die angeforderten Betriebsmittel sind nur exklusiv nutzbar (gegenseitiger Ausschluss).
- ◇ **Wartebedingung** : Alle beteiligten Prozesse warten auf die Zuteilung von Betriebsmitteln, die von den anderen Prozessen bereits belegt sind, ohne die von ihnen selbst belegten Betriebsmittel freizugeben.
- ◇ **Nichtentziehbarkeit** : Die von einem Prozess belegten Betriebsmittel können ihm nicht zwangsweise entzogen werden.
- ◇ **Zyklisches Warten** :  $n$  ( $n \geq 2$ ) Prozesse warten in einer geschlossenen Kette auf ein Betriebsmittel, das durch den jeweils nächsten Prozess in der Kette gehalten wird.

- **Strategien zur Behandlung von Verklemmungen :**

- ◇ **Ignorieren** (*ignore*) ("Vogel-Strauß-Algorithmus") :  
Dieser "Strategie" liegt die Annahme zu Grunde, daß Verklemmungen relativ selten sind und sich daher der Aufwand für andere Behandlungsstrategien nicht lohnt.  
Zumindest in Echtzeitsystemen (Prozesssteuerungen) nicht tragbar.
- ◇ **Erkennen und Beseitigen** (*detect and recover*) :  
Deadlocks werden prinzipiell zugelassen, das System versucht, diese zu erkennen und dann etwas dagegen zu tun.  
Es gibt mehrere Algorithmen, mit denen das Vorliegen von Verklemmungen festgestellt und die daran beteiligten Prozesse identifiziert werden können. Beseitigung einer erkannten Verklemmung z.B. durch Zurückversetzen eines Prozesses, der ein benötigtes Betriebsmittel besitzt, in einen früheren Zustand (*Rollback*), in dem er dieses Betriebsmittel noch nicht belegt hat, und anschließende Zuteilung dieses Betriebsmittels an einen anderen Prozess. Eine andere – einfachere aber härtere – Möglichkeit hierfür : Gewaltamen Beendigung eines oder mehrerer der beteiligten Prozesse. Einfachere Methode : Zyklische Überprüfung auf Prozesse, die bereits eine längere Zeit blockiert sind und gewaltsame Beendigung derartiger Prozesse. Nachteil : Daten können verloren gehen
- ◇ **Vermeiden** (*avoid*) :  
Bei dieser Strategie wird dafür gesorgt, daß nie alle für eine Verklemmung notwendigen Bedingungen erfüllt sind.  
Beispiele :
  - Vorsorge, daß ein exklusives Betriebsmittel überhaupt nur von einem einzigen festgelegten Prozess benutzt werden kann (z.B. Druckerspöler)
  - Verhinderung der Wartebedingung dadurch, daß jeder Prozess bei der Neuansforderung eines Betriebsmittels zunächst alle bisher belegten Betriebsmittel freigeben muss. Nur bei erfolgreicher Neuansforderung bekommt er auch alle bisherigen Betriebsmittel wieder zurück.
- ◇ **Verhindern** (*prevent*) durch sorgfältige Betriebsmittelvergabe und darauf abgestimmtes Scheduling:  
Das BS darf ein Betriebsmittel nur zuteilen, wenn es "ungefährlich" ist. Hierfür muß ihm der maximale Betriebsmittelbedarf aller Prozesse bekannt sein. Das BS kann dann bei jeder Betriebsmittelanforderung überprüfen, ob der Restvorrat freier Betriebsmittel auch noch nach erfolgter Zuteilung groß genug ist, um bei wenigstens einem Prozess den Maximalbedarf befriedigen zu können. Nur wenn das der Fall ist, erfolgt auch tatsächlich die Zuteilung, sonst wird sie verweigert.

## Speicherverwaltung - Allgemeines

- Der **Arbeitsspeicher** ist ein Betriebsmittel, das **von jedem Prozess benötigt** wird.  
Die **Arbeitsspeicherverwaltung** ist daher eine **zentrale Aufgabe** jedes Betriebssystems.
- **Aufgaben der Arbeitsspeicherverwaltung** (*memory manager*)
  - ◇ Allokation von Speicherbereichen für die Prozesse (für Programmcode und Daten, beim Laden und aufgrund späterer Anforderungen)
  - ◇ Freigabe nicht mehr benötigten Speichers
  - ◇ Führung einer Übersicht über freie und belegte Teile des Arbeitsspeichers
  - ◇ Gegebenenfalls Durchführung einer Speicherbereinigung (Freispeichersammlung, Speicherverdichtung)
  - ◇ Organisation der Auslagerung von Speicherbereichen auf einen Hintergrundspeicher, wenn nicht alle geladenen (laufenden) Prozesse vollständig in den Arbeitsspeicher passen
  - ◇ Realisierung von Speicherschutzmaßnahmen
- Der **tatsächliche Umfang der Speicherverwaltung** wird entscheidend bestimmt von
  - ◇ der **Art des eingesetzten Speichersystems** (Speicher ohne Auslagerung, Speicher mit Auslagerung wie Swapping oder Virtueller Speicher, Speicher mit Bank-Umschaltung)
  - ◇ der **Klasse** und dem **Umfang des Betriebssystems** (Singletasking-BS, Multitasking-BS, Echtzeit-BS usw)
- In modernen DV-Systemen wird die Speicherverwaltung i.a. durch **geeignete Hardwarekomponenten (Speicherverwaltungseinheit, *memory management unit*, MMU)** unterstützt.  
  
In derartigen Systemen weist die Speicherverwaltung i.a. eine **Schichtenstruktur** auf :
  - ◇ untere Schicht : **Verwaltung des realen Arbeitsspeichers**
  - ◇ obere Schicht : **Verwaltung eines virtuellen Speichers**
- **Anmerkungen zur Adressenlage von Programmen**

In einigen – vor allem für Steuerungszwecke konzipierten – Systemen (Einzelprogramm-Systeme, aber auch Mehrprogramm-Echtzeitsysteme) steht der genaue Adressbereich des Programmecodes und der Daten von vorneherein fest. Der entsprechende Programmcode enthält die absoluten Adressen und ist mit diesen Adressen sofort lauffähig.

In **universell einsetzbaren Systemen** (universelle Mehrprogrammsysteme, aber auch Einzelprogrammsysteme) **steht bei der Programmcodeerstellung aber nicht fest, in welchen Adressbereich** das Programm einmal **geladen** werden wird.

Falls nicht virtuelle Speichertechnik verwendet wird, müssen die entsprechenden Programme daher

- entweder **positionsunabhängig** (*position independant*) sein, d.h. alle Adressangaben (sowohl für Code als auch für Daten) müssen relativ zum Befehlszähler angegeben sein,
- oder **frei verschieblich (relokatibel)** sein, d.h. Informationen über beim Laden notwendige Adresskorrekturen (**Relokationsinformationen**) enthalten.

## Verwaltung des realen Arbeitsspeichers

- Die Aufgabe der **Speicherverwaltung** ist **einfach** bei **reiner Einzelprogrammverarbeitung** oder bei **Mehrprogrammverarbeitung** mit einer **Aufteilung des Arbeitsspeichers** in eine **feste Anzahl** von Bereichen (Partitionen, *partitions*) **fester Größe** (die aber nicht notwendigerweise für alle Bereiche gleich sein muß).
- **Komplexer** und **aufwendiger** ist die Verwaltung bei einer **Aufteilung des Speichers** in eine **variable** – sich ständig ändernde – **Anzahl** von Bereichen **variabler Größe**.  
Auf **Anforderung** muß die Speicherverwaltung einen **zusammenhängenden freien Bereich gewünschter Größe** zur Verfügung stellen (→ **Freispeicherverwaltung**).

- **Problem der Speicherfragmentierung :**

Da **dynamisch** eine **wechselnde Anzahl** von Bereichen **unterschiedlicher Größe** **allokiert** und wieder **freigegeben** wird, wird im Laufe der Zeit der Speicher in **belegte** und **freie** Bereiche **fragmentiert** (Speicherzerstückelung). Dies kann dann dazu führen, daß eine **Anforderung** nach einem Speicher **bestimmter Größe nicht erfüllt** werden kann, weil der gewünschte freie Speicher **nicht geschlossen zur Verfügung** steht, obwohl **insgesamt genügend** Speicher **frei** ist.

**Abhilfe** schafft eine **Speicherbereinigung** (Freispeichersammlung, *garbage collection*, bzw Speicherverdichtung, *compaction*). Diese ist sowohl **aufwendig** als auch **Zeit beanspruchend**. Daher ist sie **nicht immer** bzw nicht immer im erforderlichen Umfang **realisiert**.

Für die Durchführung einer Speicherbereinigung existieren **verschiedene Konzepte**.

Üblich ist u.a. ein **im Hintergrund** – mit niedriger Priorität – **laufender Systemprozess** (bei Freispeichersammlung) oder ein **Warten** bis erstmals eine **Speicheranforderung nicht befriedigt** werden kann (bei Speicherverdichtung)

- **Speicherbelegungsübersicht :**

Die Speicherverwaltung benötigt eine **Übersicht** über den **Belegungszustand** (frei oder belegt) des zu verwaltenden Speichers.

Üblich sind die **folgenden Konzepte** :

- ◊ **Belegungs-Bitmap** (*allocation bit map*)

Sie wird angewendet, wenn der Speicher nur in **ganzzahligen Vielfachen einer bestimmten Einheitsgröße** (z.B. 2k) verwaltet wird.

Jedem Einheitsgrößenbereich wird ein **Bit** zugeordnet, dessen Wert anzeigt, ob der Bereich **frei** (0) oder **belegt** (1) ist.

Nachteil : Das Durchsuchen der Bitmap nach freien Bereichen bestimmter Länge ist relativ aufwendig.

Vorteil : Eine Freispeichersammlung ist automatisch ohne zusätzlichen Aufwand realisiert.

- ◊ **Speicherbelegungstabelle**

Sinnvoll anwendbar, wenn die einzelnen **Speicherbereiche** eine **beliebige Größe** haben können.

Für jeden Bereich ist ein Eintrag in der Tabelle enthalten, der u.a. die **Startadresse** und die **Länge** des Bereichs, sowie den **Belegungszustand** enthält.

- ◊ **Speicherbelegungsliste**

Die Informationen über die einzelnen Speicherbereiche (Startadresse, Länge, Belegungszustand) sind in einer **linearen Liste** zusammengefaßt. Dabei ist es sinnvoll, die Listenelemente als Verwaltungsstrukturen den jeweiligen Speicherbereichen direkt voranzustellen.

- **Allokationsstrategien :**

Sequentielles Durchsuchen der Belegungsübersicht nach einem **passenden freien Bereich**

- **First Fit** : Beginn der Suche am Speicheranfang. Der erste passende freie Bereich wird gewählt.

Wenn dieser Bereich größer als benötigt ist, wird der Rest als freier Bereich markiert.

- **Next Fit** : Wie First Fit, nur beginnt die Suche beim letzten allokierten Bereich und nicht am Speicheranfang

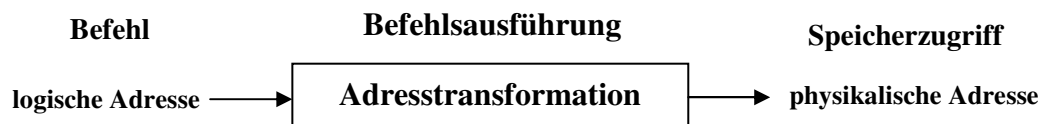
- **Last Fit** : Beginn der Suche am Speicheranfang. Der letzte passende freie Bereich wird gewählt.

- **Best Fit** : Die gesamte Übersicht wird nach einem möglichst genau passenden Bereich durchsucht.

## Virtuelle Speichertechnik (1)

**Virtuelle Speichertechnik** liegt vor, wenn der **Speicheradressraum**, den die **Befehle eines Programms** referieren (→ **logischer Adressraum**), **getrennt** ist vom **Adressraum des realen Arbeitsspeichers**, in dem sich das Programm bei seiner Abarbeitung befindet (→ **physikalischer Adressraum**).

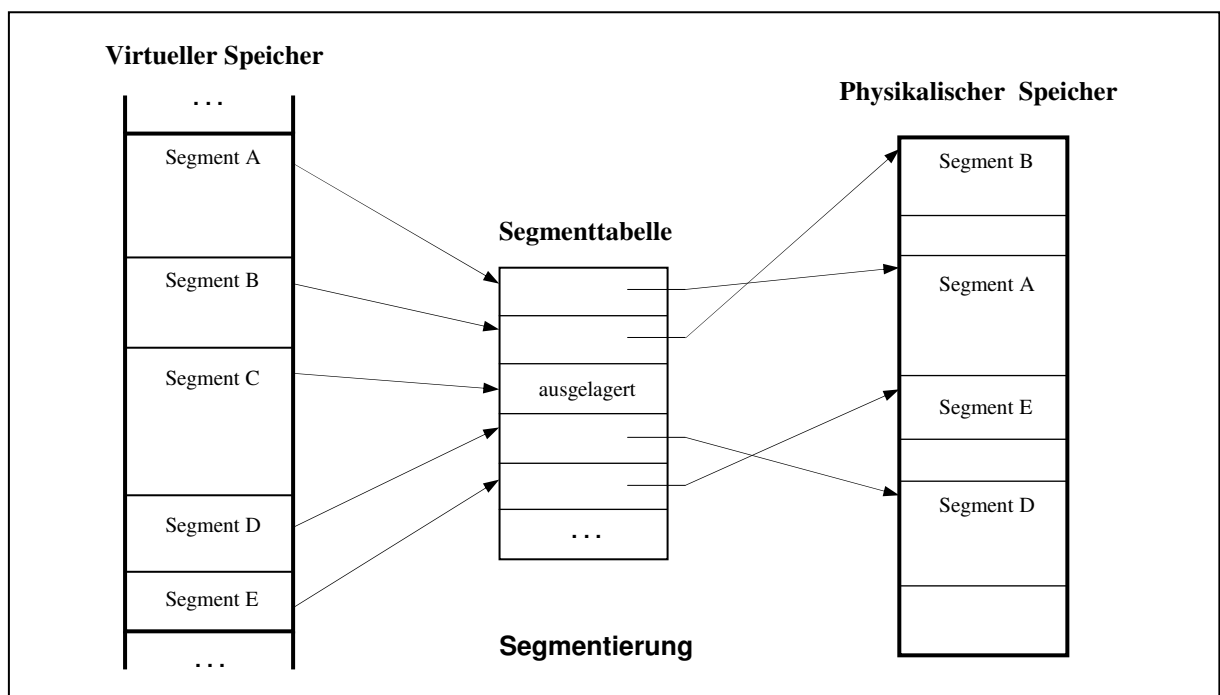
- Die Trennung des logischen Adressraums vom physikalischen Adressraum ermöglicht es, **Programme unabhängig vom tatsächlich vorhandenen Arbeitsspeicher** zu schreiben.
- Der **logische Adressraum** beschreibt einen gedachten, **nicht real vorhandenen** Arbeitsspeicher, den man auch als **virtuellen Speicher** bezeichnet.
  - **logischer Adressraum** = **virtueller Adressraum**  
**logische Adresse** = **virtuelle Adresse**
- I. a. ist der **logische Adressraum größer** als der physikalische Adressraum (meist sogar sehr viel größer). Der **virtuelle Speicher**, in dem sich ein Programm ja logisch befindet, wird auf einen **Hintergrundspeicher** (meist Plattenspeicher) abgebildet.  
Zur **Ausführung** muß ein **Programm** in den echten Arbeitsspeicher **geladen** werden.  
Wegen der **sequentiellen Abarbeitung** wird allerdings i. a. in einem gewissen Zeitabschnitt **nie der gesamte Programmcode** und **nie der gesamte Datenbereich** des Programms benötigt.  
Es reicht daher aus, **nur** die jeweils **aktuell benötigten Programmcode- und Datenbereichs-Teile** (= "working set") im Arbeitsspeicher zu halten.
  - => Ein Programm wird in einzelne **Teilabschnitte zerlegt**, die nur **bei Bedarf** (wenn sie während der Abarbeitung benötigt werden), **in den Arbeitsspeicher geladen** werden.
  - - **Programme** sind in ihrer **Länge nicht** durch die **reale Arbeitsspeichergröße begrenzt**.
    - Es können "**gleichzeitig**" (Multitasking, Timesharing) **mehrere Programme**, deren **Gesamtlänge die reale ASP-Größe übersteigt**, ausgeführt werden.
- Die virtuelle Speichertechnik erlaubt es dem **Programmierer**, sich **nicht** um die spätere **tatsächliche Lage** seines Programms (bzw seiner Abschnitte) im realen ASP zu kümmern.  
Für die **Adressierung** kann er in allen Programmen den einheitlichen – i. a. sehr großen – **virtuellen Adressraum zugrunde legen**, so als ob das der ihm zur Verfügung stehende ASP-Adressraum wäre.  
→ wesentlich **einfachere Programmierung**, wesentlich **größere Flexibilität**.
- Zur **Abarbeitung** der einzelnen Befehle eines Programms ist eine **Transformation** der logischen Adressen in die korrespondierenden physikalischen Adressen erforderlich.  
Diese Transformation findet aber erst **während der Befehlsausführung** statt → **Dynamische Adressumsetzung (DAT)**  
Auch **nach dem Laden** eines Programms bleiben die in ihm erhaltenen **logischen Adressen unverändert** erhalten.



- Wird bei der Abarbeitung eines Programms eine **Adresse** in einem noch **nicht im Arbeitsspeicher** befindlichen Code- oder Datenabschnitt referiert, so muss der entsprechende Abschnitt **nachgeladen** (und gegebenenfalls ein **anderer Abschnitt ausgelagert**) werden.  
→ Ein **Nachladen** erfolgt also immer erst **bei Bedarf** ("on demand").
- Die virtuelle Speichertechnik ist für den **Anwender** (und den Anwendungsprogrammierer) **vollkommen transparent**. Sowohl die **Adressumsetzung** als auch das **Nachladen** von Code- und Datenabschnitten erfolgen **automatisch** und brauchen bei der Programmerstellung nicht berücksichtigt zu werden.

## Virtuelle Speichertechnik (2)

- Die **virtuelle Speichertechnik** wird durch eine **Kombination von Hard- und Software** realisiert :
  - ◇ Die **Adressumsetzung** muß bei jedem Speicherzugriff durchgeführt werden und muß daher sehr schnell geschehen. Sie erfolgt deshalb i.a. mittels spezieller **Hardware-Einrichtungen** (Speicherverwaltungseinheit, *memory management unit*, MMU). Diese überprüft i.a. auch die **Zulässigkeit eines Speicherzugriffs** → Realisierung von **Speicherschutzmechanismen**.
  - ◇ Das **Nachladen** von Programmabschnitten wird dagegen durch spezielle **Betriebssystem-Komponenten** bewirkt. **Grundvoraussetzung** hierfür ist die **Fähigkeit eines Prozessors**, einen **laufenden Befehl zu unterbrechen** (bei erkannter Nachladennotwendigkeit) und nach Behebung der Unterbrechungsursache (erfolgreichem Nachladen) diesen **erneut aufzusetzen** (und dann vollständig auszuführen).
- **Realisierungsmethoden** :
  - ▷ **Segmentierung**
  - ▷ **Seitenadressierung (Paging)**
- **Segmentierung**
  - ◇ **Unterteilung des logischen Adressraums in Abschnitte variabler Größe** entsprechend den **logischen Einheiten** eines Programms (Hauptprogramm, Unterprogramme, Datenbereiche); d.h. Unterteilung nach **logischen Gesichtspunkten**. Diese Abschnitte heißen **Segmente**. Sie stellen die **kleinste Austauschereinheit** dar. Für jedes System ist eine **minimale** u. **maximale Segmentgröße** festgelegt (z.B. zwischen 256 Bytes und 64 kBytes).
  - ◇ Die **logische Adresse** besteht aus :
    - ▷ **Segment-Nummer** (höherwertiger Teil). Sie dient zur Auswahl eines Eintrags in der **Segmenttabelle**
    - ▷ **Offset** ( Displacement) relativ zum Segmentanfang (niederwertiger Teil)
  - ◇ **Probleme** (gleichzeitig **Hauptnachteile** der Segmentierung) :
    - ▷ **Speicherzersplitterung** ("externe Fraktionierung") : Zwischen den im ASp befindlichen Segmenten entstehen nicht belegte Lücken, wenn ein Segment gegen ein kleineres Segment ausgetauscht wird.
      - Es kann Fälle geben, bei denen der für ein Segment **benötigte Speicher nicht geschlossen zur Verfügung steht**, obwohl insgesamt **genügend freier Speicher vorhanden** ist.
      - **Neuordnung der ASp-Belegung** durch das Betriebssystem wird notwendig (Zusammenschieben der Segmente)
    - ▷ **Umständlicher Austauschalgorithmus** (Suche nach "passendem Loch")



## Virtuelle Speichertechnik (3)

### • Seitenadressierung (*Paging*)

- ◇ **Unterteilung des logischen und des physikalischen Adressraums in Abschnitte gleicher Länge.**

Diese Abschnitte werden **Seiten** genannt.

Es besteht **keinerlei Bezug zur logischen Struktur** des Programms

**Typische Seitengröße** : 512 Bytes ... 4 kBytes.

Eine Seite stellt die **Austauscheinheit** dar (→ **Seitenwechsel**)

- ◇ Die **logische Adresse** besteht aus :

- ▷ **logischer Seitennummer** (höherwertiger Teil)

- ▷ **Wortadresse** (Zeilenadresse) relativ zum Seitenanfang (niederwertiger Teil)

Auch die **physikalische Adresse** besteht aus zwei Teilen :

- ▷ **physikalischer Seitennummer** (höherwertiger Teil)

- ▷ **Wortadresse** (niederwertiger Teil), die **unverändert** aus der logischen Adresse übernommen wird

→ Die **Adresstransformation** besteht in einem **Austausch** der **logischen Seitennummer** gegen die **physikalische Seitennummer** (mittels einer – gegebenenfalls auch mehrstufigen – **Seitentabelle**)

- ◇ Da **alle Seiten** die **gleiche Größe** haben, kann **jede Seite** problemlos **durch jede andere Seite ersetzt** werden.  
Für die Auswahl derjenigen Seite, die bei einem Seitenwechsel entfernt werden soll, existieren mehrere Algorithmen (**Seitenwechsel-Algorithmen**), am gebräuchlichsten ist "*least recently used*" (**LRU**)

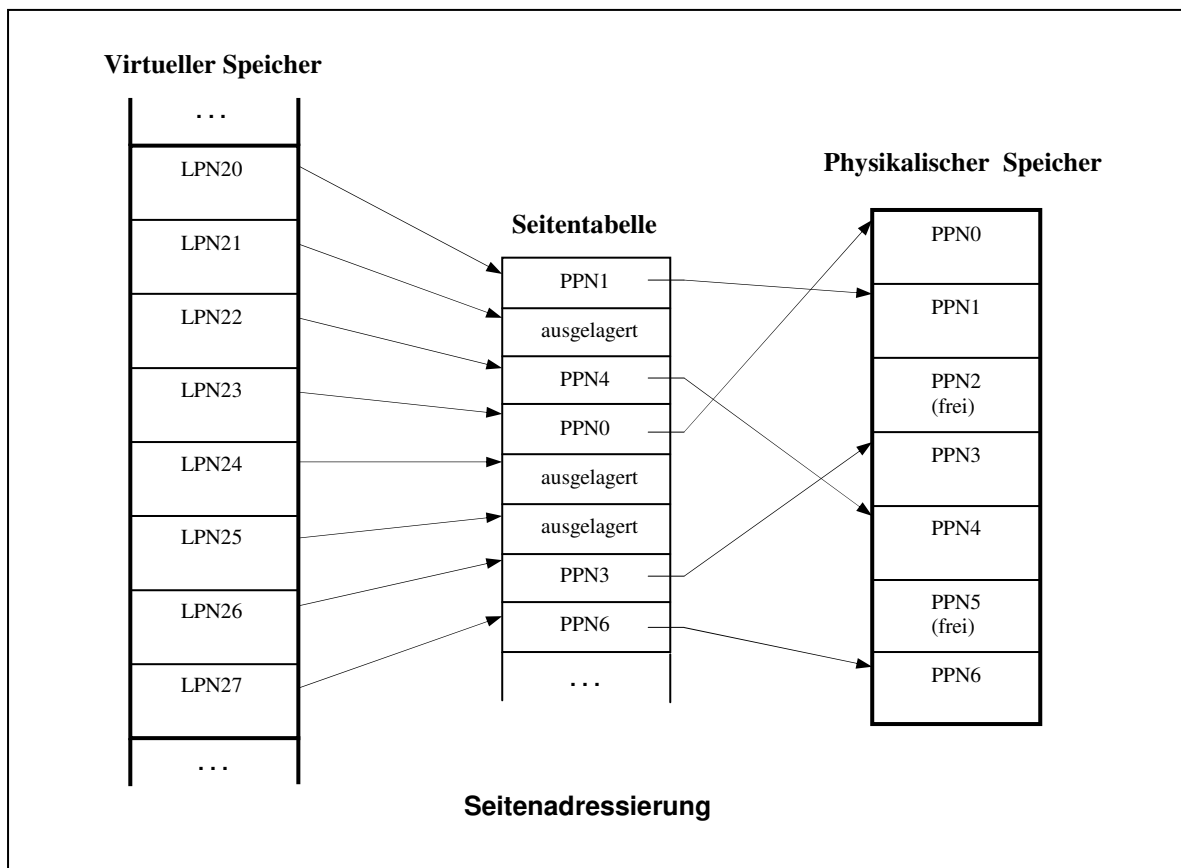
- ◇ **Vorteile** (gegenüber Segmentierung) :

- ▷ **keine Speicherzersplitterung**

- ▷ relativ **einfacher Austauschalgorithmus** (keine Suche nach einem "passenden Loch")

- ▷ **schnellerer Nachladevorgang** (kleinere Abschnitte)

=> **Paging** ist für die Realisierung eines virtuellen Speichers wesentlich **besser geeignet als Segmentierung**.



## Externe Datenverwaltung (1)

### • Dateien

- ◇ **Daten** werden **extern** üblicherweise in Form von **Dateien** gespeichert.  
Die Speicherung erfolgt auf **blockorientierten Hintergrundspeichern**, heute typischerweise auf **Magnetplatten** bzw. Speicherkarten / *memory sticks* (nichtflüchtige Halbleiterspeicher, *solid state disks*)
- ◇ Dateien können – betriebssystemabhängig – **unterschiedlich strukturiert** sein :
  - ▷ Im einfachsten – und am weitesten verbreiteten - Fall bestehen sie aus einer **linearen Folge von Bytes** (→ **sequentielle Dateien**)  
Einen gewissen Sonderfall stellen hierbei häufig **Textdateien** dar (Zeilenstrukturierung !).
  - ▷ Es gibt aber auch Betriebssysteme, die eine **komplexere Strukturierung** – angepasst an die Organisation des Dateiinhalts – verwenden.( → **Direktzugriffs-Dateien, indexsequentielle Dateien**)
- ◇ Auf dem **Datenträger** sind die Dateien **blockweise** abgelegt. Ein Block (häufig **Sektor** genannt) stellt die **Adressiereinheit** auf dem Datenträger und gleichzeitig die **Transfereinheit** von/zum Datenträger dar.  
Aus diesem Grund unterteilen Betriebssysteme Dateien für den **Zugriff** ebenfalls in Blöcke (**Sektoren**).

### • Dateisysteme

- ◇ Betriebssysteme stellen ein – oder auch mehrere – **Dateisystem(e)** zur Verfügung.
- ◇ Ein Dateisystem legt fest :
  - ▷ die **Organisationsform** und die **Bedeutung** und **Verwendung** der auf dem Datenträger vorhandenen **Verwaltungsstrukturen** sowie
  - ▷ **Randbedingungen** für die **Dateien** (z.B. max. Größe) und ihre **Verwendung** (z.B. Aufbau u. Länge von Dateinamen)
- ◇ Im Rahmen des Dateisystems sind **Funktionen zum Bearbeiten von Dateien** implementiert :
  - ▷ **Erzeugen** und **Löschen** von Dateien,
  - ▷ **Lesen** und **Schreiben** von Dateien
- ◇ In vielen heutigen Dateisystemen werden auch **Geräte als** – spezielle – **Dateien betrachtet**

### • Sequentielle Dateien

- ◇ **Sequentielle Dateien** – und nur solche werden hier betrachtet – werden im Normalfall (es gibt Ausnahmen) **nur sequentiell geschrieben**
- ◇ Beim **Lesen** ist in vielen Systemen ein **wahlfreier Zugriff** zu **beliebigen Sektoren** möglich.  
Häufig können derartige Dateien auch wahlfrei **überschrieben** werden.
- ◇ Ein direktes **Entfernen** oder **Einfügen** von Daten **innerhalb** sequentieller Dateien ist **nicht möglich**



## Logische Struktur von Dateisystemen (1)

### • Dateiverzeichnisse (*Directories*)

- ◇ Um eine vernünftige **Zugriffsmöglichkeit** auf die Dateien eines Dateisystems zu haben, werden diese in **Inhaltsverzeichnisse** (Dateiverzeichnisse, Kataloge, Ordner, *directories*) eingetragen.
- ◇ Bei den heutigen Dateisystemen sind **hierarchisch strukturierte Verzeichnisse** (Verzeichnisbaum) üblich :  
In einem Verzeichnis können nicht nur Dateien, sondern wiederum Verzeichnisse eingetragen sein.  
Der Verzeichnisbaum beginnt im **Wurzelverzeichnis** (*root directory*).  
Es gibt Dateisysteme, bei denen **Verzeichnisse als – spezielle – Dateien behandelt** werden.
- ◇ Bei Verzeichnisbäumen werden Dateien nicht durch ihren Namen allein angesprochen, sondern durch einen **Zugriffspfad**, der die Verzeichnisse, über die man zu der Datei kommt, spezifiziert.  
Dies kann erfolgen durch
  - ▷ einen **absoluten Pfad** (Beginn im Wurzelverzeichnis) oder durch
  - ▷ einen **relativen Pfad** (Beginn im aktuellen (Arbeits-) Verzeichnis )  
Die **Syntax** für Datei- und Verzeichnisnamen sowie den Aufbau von Zugriffspfaden ist **betriebssystem- und datei-systemspezifisch**.

### • Logische Laufwerke

- ◇ Die externen Datenträger werden üblicherweise in Form **logischer Laufwerke** betrachtet.  
Dabei können sich auf einem physikalischen Laufwerk mehrere logische Laufwerke befinden (Partitionierung).
- ◇ Es gibt Betriebssysteme, die die **Verzeichnissbäume der einzelnen logischen Laufwerke getrennt halten** (→ Zugriffspfad muß durch Laufwerksangabe ergänzt werden).  
Andere Betriebssysteme fassen alle Einzel-Verzeichnissbäume zu einem einzigen **Gesamt-Verzeichnisbaum** zusammen.

### • Dateispezifische Informationen

- ◇ Im Rahmen des Dateisystems werden für jede **Datei** spezifische **Informationen** benötigt, die auf dem Datenträger abgelegt sein müssen, z.B.
  - **Dateiname** (u. gegebenenfalls **Dateityp**)
  - **Zugriffsrechte** (**Dateiattribute**)
  - **Datum der Erstellung** / der **letzten Änderung** / des **letzten Zugriffs**
  - **Eigentümer**
  - **Größe**
  - **Speicherort**
- ◇ Diese Informationen können enthalten sein :
  - ▷ **im Verzeichniseintrag** der Datei
  - ▷ oder **in anderen Verwaltungsstrukturen** (die dann durch den Verzeichniseintrag referiert werden)

## Logische Struktur von Dateisystemen (2)

### • Dateiverweise (Links)

- ◇ Sie ermöglichen, dass **eine** – nur einmal vorhandene – **Datei** (oder ein Verzeichnis) über **unterschiedliche Verzeichniseinträge** – auch unter **unterschiedlichen** Namen – referiert werden kann. Dies ist nur möglich, wenn die Informationen über diese Datei – außer dem Namen – nur einmal im Dateisystem abgelegt sind.

- ◇ Es gibt **zwei Arten von Links** :

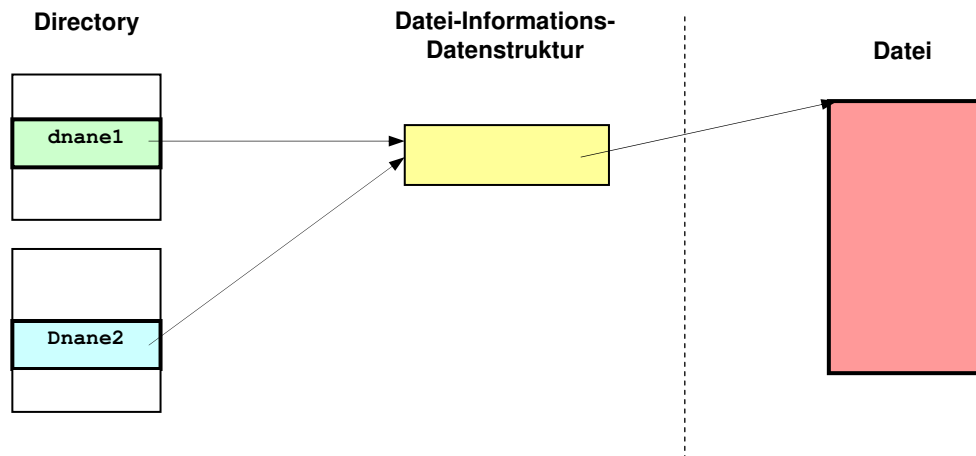
#### ► Hard Links

Sie lassen sich sehr einfach implementieren, wenn die **Datei-Informationen** nicht in den Verzeichniseinträgen sondern in **gesonderten Datenstrukturen** enthalten sind (z.B. *i-nodes* in UNIX).

In den Verzeichniseinträgen befinden sich lediglich **Verweise** auf diese die jeweilige Datei beschreibende Datenstruktur.

Ein Link entsteht, wenn die **gleiche** Informations-**Datenstruktur** durch **mehrere Verweise**, die sich durchaus in unterschiedlichen Verzeichnissen befinden können, referiert wird.

Hard Links sind nur innerhalb ein und desselben logischen Laufwerks möglich.

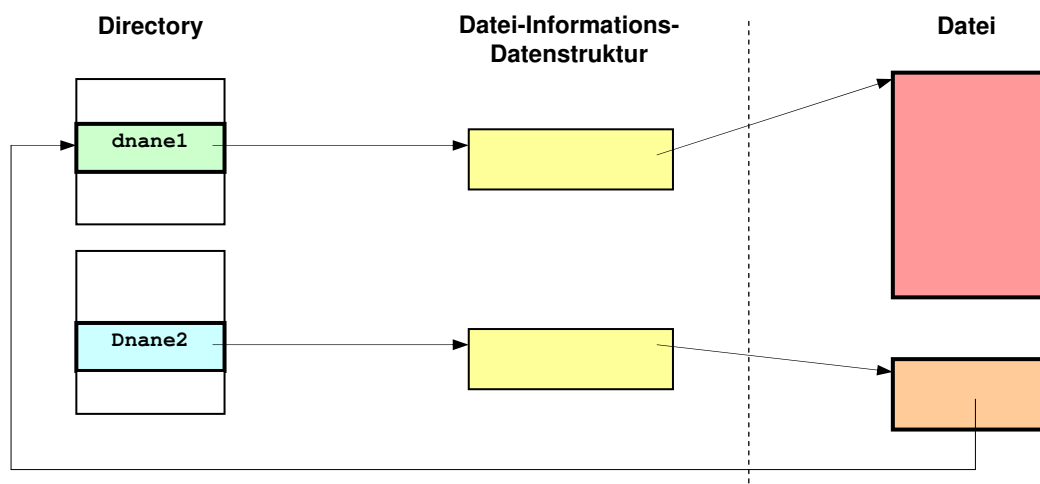


#### ► Soft Links (*Symbolic Links*)

Ein Soft Link ist eine **spezielle Datei**, die als Inhalt den **absoluten Zugriffspfad** zu der eigentlich referierten Datei enthält.

Soft Links sind auch möglich in Systemen, bei denen die Datei-Informationen im Verzeichniseintrag enthalten sind.

Sie können sich auch **über logische Laufwerksgrenzen hinweg** erstrecken.



## Physische Struktur von Dateisystemen (Dateiorganisation)

- Unter der **physischen Struktur** eines Dateisystems (Dateiorganisation) versteht man das **Schema** der **Abbildung** des **Dateiinhalts** auf die durch die **Datenträgergeometrie** (Zylinder, Kopf, Sektor) definierten **physikalischen Sektoren**.

Diese Abbildung erfolgt in zwei Schritten :

- ◊ **Abbildung** des **Dateiinhalts** auf **logische Sektoren**  
Dieser Schritt erfolgt durch den Betriebssystem-Kernel. Das BS betrachtet den Datenträger – unabhängig von der tatsächlichen physikalischen Struktur - als eine lineare Folge von – logischen – Sektoren.
- ◊ **Abbildung** der **logischen Sektoren** auf **physikalische Sektoren**.  
Dieser Schritt wird i.a. durch **Gerätetreiber** und/oder durch in die Datenträger integrierte **Steuereinheiten** realisiert

- **Aufgaben des Betriebssystems im Rahmen der Dateiabbildung auf logische Sektoren :**

- **"Buchführung"** über **freie** (sowie belegte und defekte) **Sektoren**
- **Allokation** von Sektoren bei Erzeugung bzw Erweiterung einer Datei (bzw eines Verzeichnisses)
- **Zuordnung** von **Sektoren** zu **Dateien** (und Verzeichnissen)  
I.a. ist es zulässig, Dateien **beliebige Sektoren** in **beliebiger Reihenfolge** zuzuordnen → **fragmentierte Dateien**
- **"Buchführung"** über die **Reihenfolge** der einer Datei (oder einem Verzeichnis) **zugeordneten Sektoren**
- **Freigabe** der von **gelöschten** Dateien belegten Sektoren
- **Verwaltung** der **Verzeichnisse** und übrigen **dateispezifischen Informationsstrukturen**

- **Verwaltungsstrukturen zur Wahrnehmung der "Buchführungs"-Aufgaben :**

- ◊ **Übersicht** über **freie** und defekte Sektoren
- ◊ **Zuordnungs-Übersicht** Sektoren – Dateien

Aus Effizienzgründen können in diesen Übersichten mehrere aufeinanderfolgende Sektoren jeweils zusammengefasst sein (→ logischer **Datenblock**, **Cluster**)

In der Realisierung dieser Strukturen unterscheiden sich – u.a. - die verschiedenen Dateisysteme.

- **Realisierungsmethoden für die Übersicht über freie und defekte Sektoren**

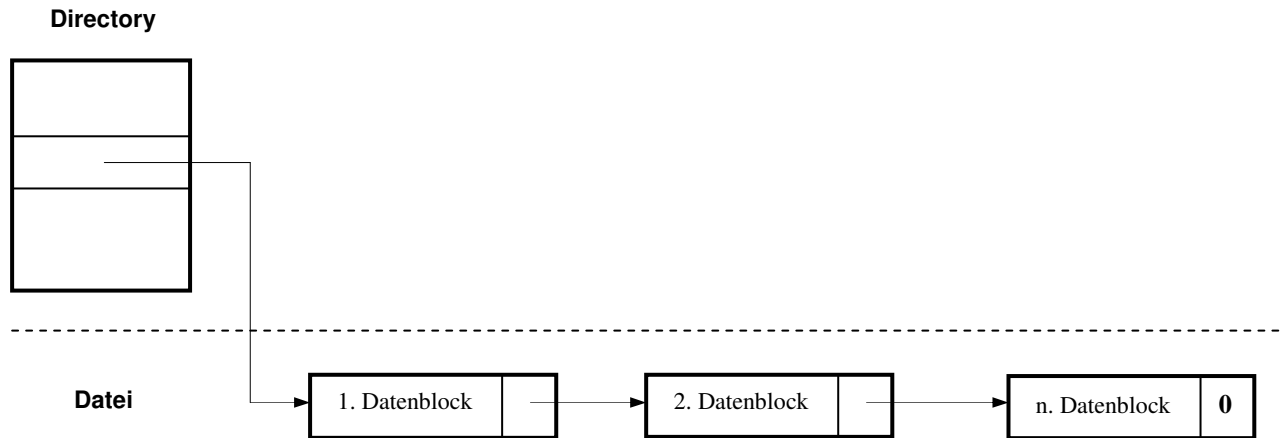
- ◊ Markierung des **Belegungszustands** für alle Sektoren in einer **Bitmap**  
(z.B. Bit=0 → Sektor belegt, Bit=1 → Sektor frei)
- ◊ Markierung **freier** und **defekter Sektoren** in einer **Tabelle** (z.B. Dateibelegungstabelle, s. unten)
- ◊ Zusammenfassen aller **freien Sektoren** in einer **verketteten linearen Liste**
- ◊ **Defekte Sektoren** werden häufig als belegt markiert und in einer **Pseudodatei** zusammengefasst.

- **Realisierungsmethoden für die Sektor-Datei-Zuordnungs-Übersicht :**

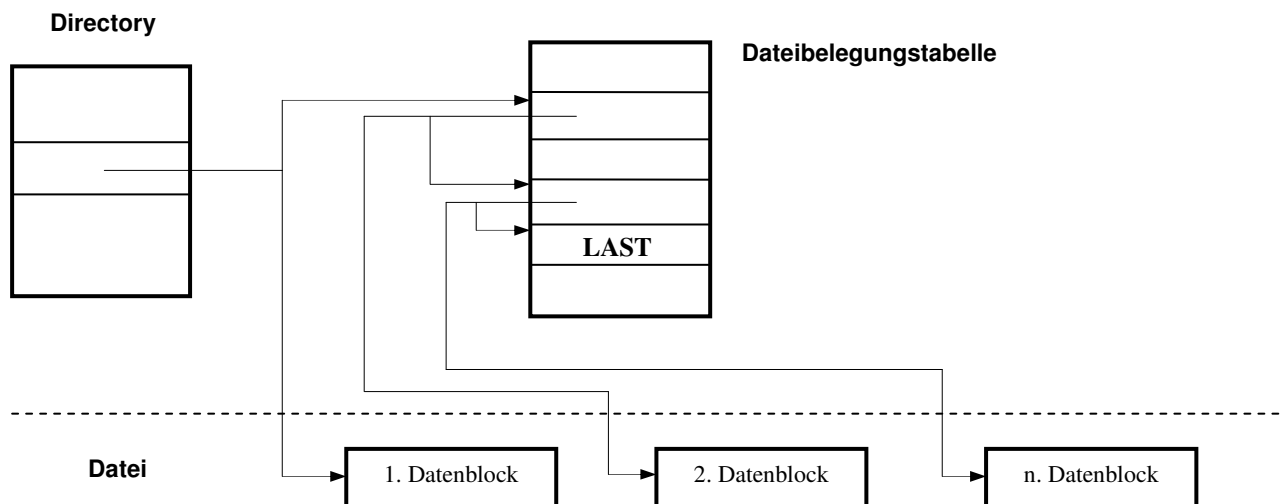
- ◊ **Verweiskette innerhalb** der Sektoren einer Datei (**interne Verkettung**)
- ◊ **Verweiskette** in einer gesonderten **zentralen Dateibelegungstabelle** (**externe Verkettung**).  
Die Dateibelegungstabelle enthält Verweisketten für **alle** Dateien (und Verzeichnisse).  
Gleichzeitig enthält sie die **Übersicht** über **freie** und **defekte Sektoren**. Diese sind jeweils durch **spezielle Eintrags-Werte** gekennzeichnet
- ◊ **Verweistabelle**  
Für jede Datei existiert eine **eigene** Verweistabelle. Die Reihenfolge der Tabellenreihenfolge entspricht der Reihenfolge der Sektoren in der Datei.

## Realisierungsmethoden für die Sektor-Datei-Zuordnungsübersicht

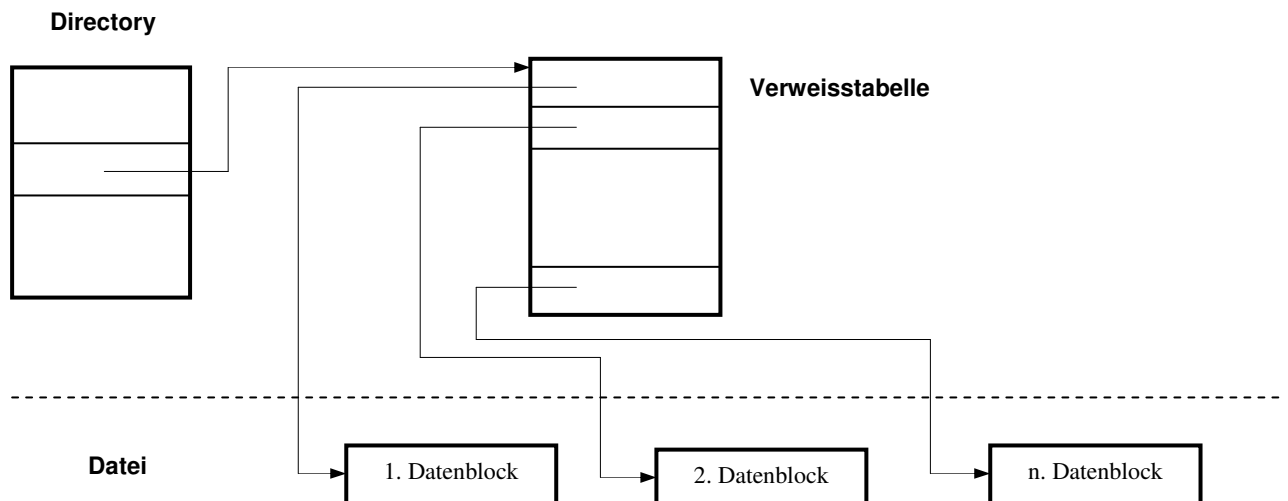
- Verweiskette innerhalb der Datenböcke (Sektoren)



- Verweiskette in einer zentralen Dateibelegungstabelle



- Verweistabelle



## Zusammenhang zwischen physikalischen und logischen Sektoradressen

### • Physikalische Datenträgersorganisation :

#### ◇ Einteilung eines Datenträgers (physikalisches Laufwerk) :

- **Zylinder** (*Spuren, cylinders*) :  $0 \leq C \leq c_{\max}$
- **Oberflächen** (*heads*) :  $0 \leq H \leq h_{\max}$
- **Sektoren** (*sectors*) :  $1 \leq S \leq p_{\max}$     **S (= PSN)** - physikalische Sektornummer

→ Die **kleinste adressierbare Dateneinheit** ist ein **Sektor**.

#### ◇ Physikalische Adresse eines Sektors :

- Angabe der Kombination von C, H und S (**CHS**)
- **Absolute Sektornummer ASN** (auch als **LBA** – *Logical Block Address* bezeichnet) :  $0 \leq \text{ASN} \leq \text{asn}_{\max}$
- **Umrechnung** :

Die Zählung der absoluten Sektornummern beginnt bei Cylinder 0, Oberfläche 0 und PSN 1.  
Sie wird zylinderweise – und innerhalb eines Zylinders oberflächenweise – fortgesetzt.

Mit    **spt**    =  $p_{\max}$     Sektoren/Spur (*sectors per track*)  
      **heads** =  $h_{\max} + 1$     Anzahl Oberflächen

gelten die Beziehungen :

$$\text{ASN} = (H + C * \text{heads}) * \text{spt} + S - 1$$

$$C = \text{ASN DIV} (\text{spt} * \text{heads})$$

$$H = (\text{ASN DIV spt}) \text{ MOD heads}$$

$$S = 1 + \text{ASN MOD spt}$$

**Beispiel :**    spt = 63  
                  heads = 255

C = 0    ⇔    ASN = 574  
H = 9  
S = 8

### • Logische Laufwerksorganisation

#### ◇ Betriebssysteme verwalten i.a. **logische Laufwerke**.

Ein Datenträger (physikalisches Laufwerk) kann ein oder mehrere logische Laufwerke umfassen.

#### ◇ Einteilung eines logischen Laufwerks :

- **logische Sektoren** :  $0 \leq \text{LSN} \leq l_{\max}$     **LSN** - logische Sektornummer

#### ◇ Logische Adresse eines Sektors :

- logische Sektornummer **LSN**

### • Zusammenhang zwischen physikalischer und logischer Adresse :

#### ◇ Vor einem logischen Laufwerk können sich auf dem physikalischen Laufwerk Sektoren befinden, die nicht zu dem logischen Laufwerk gehören (z.B. Sektoren eines anderen logischen Laufwerks) → **verborgene Sektoren**

#### ◇ Umrechnung :

Mit    **avs** = Anzahl verborgener Sektoren  
gilt

$$\text{LSN} = \text{ASN} - \text{avs}$$

**Beispiel :**    avs = 63  
                  ASN = 574    ⇔    LSN = 511

## Geräteverwaltung

- **Geräte** (i.a. E/A-Geräte) werden in der Regel **in das Dateisystem eingebunden** und aus Benutzersicht **wie Dateien behandelt**.

Allerdings können sie u.U. von normalen Dateien abweichende Eigenschaften besitzen, z.B.

- kann auf **reine Ausgabegeräte nur geschrieben**,
- und von **reinen Eingabegeräten nur gelesen** werden.

- Geräte werden in **zwei Gruppen** eingeteilt :

- ◆ **blockorientierte Geräte** (*block devices*)

Der **Datentransfer** von/zu ihnen geschieht nur **blockweise** (Datenblöcke **fester Länge** , typisch 128 Bytes ... 8 kB)

Beispiele : Platte, Band, *memory stick* (übliche externe Speichermedien)

- ◆ **zeichenorientierte Geräte** (*character devices*)

Der **Datentransfer** von/zu ihnen erfolgt **zeichenweise** (byteweise).

Beispiele : Konsole (Tastatur+Bildschirm), Drucker

- Üblicherweise wird **jedem Gerät ein fest vorgegebener Name**, unter dem es **wie eine Datei angesprochen** werden kann, zugeordnet.

- Je nach **Typ** können Geräte **von allen Prozessen** verwendet werden (z.B. Konsole) oder sie sind **exklusiv für einen** Prozess reserviert (z.B.in Multitasking-Systemen der Drucker → **Spool-Prozess**)

- Die – z. Tl. hardwareabhängige – **Software zur Geräteverwaltung** läßt sich in drei Ebenen strukturieren :

- ◆ **Gerätetreiber**

BS-Programme zur direkten Ansteuerung des Geräts. (z.B. zum Einlesen eines Datenblocks)

Da diese Programme stark geräteabhängig sind, sind sie in der untersten BS-Schicht (BIOS) angeordnet.

- ◆ **Unterbrechungsbearbeitung**

Das Gerät erzeugt nach Ausführung eines Befehls (z.B. Datenblock lesen) einen Interrupt, um anzuzeigen, dass es mit seinem Auftrag fertig ist. Der Prozess, der den Gerätebefehl ausgelöst hat, kann daraufhin durch das BS (*Interrupt Service Routine* !) aus dem Zustand "wartend" in den Zustand "bereit" versetzt werden.

- ◆ **Geräteunabhängige Software im BS-Kernel**

Diese Schicht realisiert die **Schnittstelle zum Dateisystem**.

Sie enthält eine einheitliche **Schnittstelle für Treiberaufrufe**, definiert den Gerätenamen, puffert die Daten, ist für den Geräteschutz, die Fehlerbehandlung und die Vergabe exklusiver Geräte zuständig.

## Benutzerverwaltung

### • Benutzeridentifikation

- ◇ In **Mehrbenutzer**-Betriebssystemen, aber auch in einigen Einzelbenutzer-Betriebssystemen, ist es üblich, nur **registrierten Benutzern** den **Systemzugang** zu gestatten.  
Jeder registrierte Benutzer erhält eine eindeutige **Benutzerkennung** (*user ID*).  
In vielen Systemen werden darüberhinaus die Benutzer in (**Arbeits-**)**Gruppen** (*group ID*) zusammengefaßt  
Häufig kann ein Benutzer mehreren Gruppen angehören.
- ◇ Zu Beginn der Arbeit am Rechner (*login*) muß sich der **Benutzer identifizieren** (*user ID* und oft zusätzlich ein **Password** angeben). Für die Durchführung der Identifikation ist ein eigenes Programm zuständig.  
U.U. wird noch eine zusätzliche Echtheitsprüfung (**Authentifikation**) vom System vorgenommen.
- ◇ Bei **besonders kritischer Rechnerumgebung** kann es sogar erforderlich sein, daß sich die Benutzer beim **Aufruf bestimmter Programme nochmals identifizieren** müssen.

### • Ressourcennutzung

- ◇ I.a. wird jedem Benutzer eine **fest eingegrenzte Arbeitsumgebung** zur Verfügung gestellt, die ihn **von anderen Benutzern abschirmt**.  
U.a. gehören hierzu :
  - ▷ Prozesse eines Benutzers können gleichzeitig laufende **Prozesse anderer Benutzer nicht beeinflussen** (Speicherschutz usw.)
  - ▷ Auf **Dateien** eines Benutzers können **andere Benutzer** nur mit dessen **ausdrücklicher Erlaubnis zugreifen** (Zugriffsrechte)
  - ▷ Für **gemeinsam genutzte Ressourcen** und Dateien können **definierte Zugriffsrechte** vergeben werden.
  - ▷ Die **Nutzungsdauer** (CPU-Zeit), der **Zeitraum der Nutzung** (z.B. nur von 8 – 17 Uhr) oder die **maximal zu beanspruchende Plattenkapazität** können **festgelegt** werden.
- ◇ **Spezielle Geräte und Dienste** können **besondere Schutzmaßnahmen** erfordern.  
→ **Zusätzliche Identifikation** und gegebenenfalls **Authentifikation** des Nutzers vor der Nutzung notwendig.

### • Abrechnungsinformationen

- ◇ **Zusätzliche Aufgaben der Benutzerverwaltung** betreffen **Abrechnungsinformationen** :
  - ▷ Ermittlung und Speicherung der Zeit, die ein Benutzer angemeldet ist,
  - ▷ Ermittlung und Speicherung der verbrauchten CPU-Zeit eines Benutzers
  - ▷ Ermittlung und Speicherung der Verweilzeit der Prozesse eines Benutzers
  - ▷ Speicherung des Zugriffs und der Nutzung der Ressourcen (Platte, Drucker usw)

### • Datei-Zugriffsrechte :

- ◇ Die über eine **Datei gespeicherten Informationen** enthalten u.a. auch die **Zugriffsrechte**.  
Der **Umfang** und die **Unterteilung** der Zugriffsrechte hängen vom jeweiligen Betriebssystem ab.
- ◇ Beispiel **UNIX** :      drei **Zugriffsarten** :
  - **read** :      Lesen von der Datei
  - **write** :      Schreiben in die Datei (Ändern, Verkürzen, Erweitern, Löschen)
  - **execute** :    Ausführen der Datei (nur bei Programmen)  
drei **Zugriffsberechtigungsgruppen** :
  - **owner** :      Datei-Eigentümer
  - **group** :      (primäre) Arbeitsgruppe des Dateieigentümers
  - **other** :      alle anderen

Die Kombinationen der Zugriffsarten und der Zugriffsberechtigungsgruppen ergeben insgesamt **9 Zugriffsrechte**

# **Betriebssysteme**

## **Kapitel 3**

### **3. Ausgewählte Dateisysteme**

- 3.1. Logische Strukturierung von PC-Festplatten
- 3.2. FAT12/16-Dateisystem von MS-DOS/WINDOWS
- 3.3 VFAT-Dateisystem von WINDOWS 95/98
- 3.4. FAT32-Dateisystem von WINDOWS 95/98
- 3.5. NTFS von WINDOWS NT
- 3.6. LINUX Extended 2 / Extended 3



## Der Master Boot Block von PC-Festplatten

- **Festplatten** (=physikalische Laufwerke) sind üblicherweise in **Partitionen** unterteilt.  
Die im PC-Bereich verwendeten Festplatten erlauben **maximal 4 Partitionen**.  
Man unterscheidet
  - **primäre Partitionen**, die jeweils genau **einem logischen Laufwerk** entsprechen
  - **erweiterte Partitionen**, die jeweils **mehrere logische Laufwerke** enthalten können
 Logische Laufwerke sind die Einheiten, die von einem Betriebssystem verwaltet werden.  
Sie werden vom Betriebssystem entsprechend dem jeweils verwendeten **Dateisystem** formatiert.  
Auf einem physikalischen Laufwerk können sich logische Laufwerke mit unterschiedlichen Dateisystemen befinden.
- Die Einteilung einer Festplatte in Partitionen erfolgt mit speziellen **Partitionierungsprogrammen** (z.B. **fdisk** bei MS-DOS und Linux, **Diskpart** und der **Festplatten-Manager** bei Windows XP)  
Mit diesen Programmen können i.a. auch Informationen über die vorhandenen Partitionen ausgegeben werden.
- Der **erste physikalische Sektor** (C=0, H=0, S=1) einer Festplatte wird als **Master Boot Block** (Master Boot Sektor, Master Boot Record) bezeichnet.  
Er enthält das **Master Boot Programm** und die **Partitionstabelle**.

In der **Partitionstabelle** sind die Informationen, die die verschiedenen vorhandenen Partitionen beschreiben, abgelegt.  
**Maximal eine** Partition kann eine **erweiterte Partition** sein.

**MS-DOS** und **WINDOWS** gestatten **nur** jeweils **eine primäre** – und i.a. bootfähige - Partition mit einem ihrer Dateisysteme. Weitere logische Laufwerke mit MS-DOS/WINDOWS-Dateisystemen müssen sich in der erweiterten Partition befinden.

Das **Master Boot Programm** wird beim Booten von Festplatten vom BIOS des PC in den Arbeitsspeicher geladen und als erster Code des Boot-Prozesses ausgeführt.

Es durchsucht die Partitionstabelle nach einer **bootfähigen Partition** und lädt dann - im einfachsten Fall - von dem dieser Partition zugeordneten logischen Laufwerk das eigentliche betriebssystemspezifische Boot-Programm, das seinerseits dann das Betriebssystem lädt.

In komplexeren Fällen können weitere Zwischen-Boot-Programme geladen werden (→ **Boot-Manager**), die gegebenenfalls auch nach bootbaren logischen Laufwerken (in Partitionen mit mehreren logischen Laufwerken), u.U. auch auf anderen Festplatten, suchen.

### • Aufbau des Master Boot Blocks

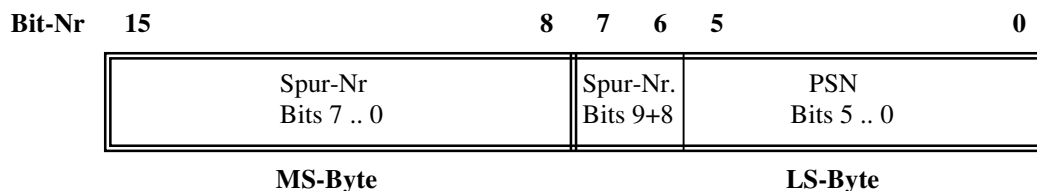
0x000	<b>Master-Boot-Programm + reservierter Bereich</b>	446 Bytes
0x1BE	<b>Partitionstabelle</b> (4 Einträge zu je 16 Bytes)	64 Bytes
0x1FE	<b>Kennung für Bootsektoren (0x55 0xAA)</b>	2 Bytes

## Die Partitionstabelle von PC-Festplatten (1)

### • Aufbau eines Partitionstabelleneintrags

1.	2.	3.	4.	Partition	
0x1BE	0x1CE	0x1DE	0x1EE	Bootflag (0x80 : bootfähig (aktiv), 0x00 : passiv)	1 Byte
0x1BF	0x1CF	0x1DF	0x1EF	Kopf-Nr. des 1. Sektors der Partition	1 Byte
0x1C0	0x1D0	0x1E0	0x1F0	Spur-Nr. und PSN des 1. Sektors der Partition Spur-Nr : 10 Bits (Codierung s. unten) PSN : 6 Bits	2 Bytes
0x1C2	0x1D2	0x1E2	0x1F2	Partitionstyp (mögliche Werte s. unten)	1 Byte
0x1C3	0x1D3	0x1E3	0x1F3	Kopf-Nr. des letzten Sektors der Partition	1 Byte
0x1C4	0x1D4	0x1E4	0x1F4	Spur-Nr. und PSN des letzten Sektors der Partition Spur-Nr : 10 Bits (Codierung s. unten) PSN : 6 Bits	2 Bytes
0x1C6	0x1D6	0x1E6	0x1F6	ASN (LBA) des ersten Sektors der Partition	4 Bytes
0x1CA	0x1DA	0x1EA	0x1FA	Anzahl Sektoren in der Partition	4 Bytes

### • Codierung der Spur-Nr. und PSN



### • Einige Partitionstypen

0x00	Eintrag nicht belegt
0x01	12-Bit FAT
0x02	XENIX
0x04	16-Bit-FAT (Partition kleiner 32 MBytes)
0x05	Extended Partition
0x06	16-Bit-FAT (Partition größer gleich 32 MBytes)
0x07	HPFS (OS/2) / NTFS (Windows NT)
0x09	AIX / COHERENT
0x0A	OS/2 Boot Manager
0x0B	FAT32
0x0C	FAT32 (verwendet LBA INT 13H Erweiterungen)
0x0E	16-Bit-FAT (verwendet LBA INT 13H Erweiterungen)
0x0F	Extended Partition (verwendet LBA INT 13H Erweiterungen)
0x65	NOVELL
0x82	LINUX Swap
0x83	LINUX native (Ext2, Ext3, Reiser usw)

### Beispiel zum Master Boot Block und zur Partitionstabelle von PC-Festplatten

- Master Boot Block (Beispiel)

Adresse		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	:	31	c0	8e	d0	bc	00	7c	8e	c0	8e	d8	bf	1e	06	be	1e	1.....!
00000010	:	7c	50	57	b9	e2	01	f3	a4	b9	00	02	f3	ab	cb	80	fa	!PW.....
00000020	:	8f	7e	02	b2	80	52	52	bb	94	07	8d	af	2a	00	8a	46	.~...RR.....*.F
00000030	:	04	66	8b	7e	08	66	03	3e	b3	06	84	c0	74	0b	80	7e	.f.~.f.>....t..~
00000040	:	00	80	75	05	66	89	3e	84	0b	83	c5	10	83	c3	09	80	.u.f.>.....
00000050	:	fb	b8	75	da	b8	e1	00	c1	e0	02	89	c6	66	8b	ac	00	.u.....f...
00000060	:	08	66	85	ed	75	19	b8	c5	06	be	bb	06	e8	a5	00	89	.f..u.....
00000070	:	c6	e8	9a	00	5a	31	c0	cd	13	cd	18	fb	f4	eb	fc	66	.....Z1.....f
00000080	:	89	2e	b3	06	be	ab	06	b4	42	5a	52	cd	13	b8	d9	06	.....BZR.....
00000090	:	72	d7	a0	00	7c	84	c0	74	03	a1	fe	7d	3d	55	aa	b8	r...!..t...>=U..
000000a0	:	e9	06	75	c5	66	89	ee	5a	e9	55	75	10	00	01	00	00	.u.f..Z.Uu....
000000b0	:	7c	00	00	00	00	00	00	00	00	00	00	45	72	72	6f	72	!.....Error
000000c0	:	20	00	0d	0a	00	4e	6f	20	61	63	74	69	76	65	20	70	....No active p
000000d0	:	61	72	74	69	74	69	6f	6e	00	44	69	73	6b	20	72	65	artition.Disk re
000000e0	:	61	64	20	65	72	72	6f	72	00	4e	6f	20	6f	70	65	72	ad error.No oper
000000f0	:	61	74	69	6e	67	20	73	79	73	74	65	6d	00	49	6e	76	ating system.Inv
00000100	:	61	6c	69	64	20	43	48	53	20	72	65	61	64	00	e8	03	alid CHS read...
00000110	:	00	be	c2	06	60	ac	b4	0e	bb	01	00	cd	10	ac	84	c0	....
00000120	:	75	f4	61	c3	00	00	00	00	00	00	00	00	00	00	00	00	u.a.....
00000130	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000140	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000150	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000160	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000170	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000180	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000190	:	1c	80	b6	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001a0	:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001b0	:	00	00	00	00	00	00	00	00	00	f8	f4	0a	00	e1	b5	80	01
000001c0	:	01	00	83	fe	3f	01	3f	00	00	00	43	7d	00	00	00	00	....?....C>....
000001d0	:	01	02	82	fe	7f	07	82	7d	00	00	86	39	40	00	00	00	.....>...90...
000001e0	:	41	08	0f	fe	ff	ff	08	b7	40	00	63	fb	97	09	00	00	A.....0.c...
000001f0	:	c1	ff	07	fe	ff	ff	6b	b2	d8	09	99	41	21	04	55	aa	.....k....A!.U..
Adresse		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF

- Partitionstabelle (zu obigem Beispiel)

	1. Partition	2. Partition	3. Partition	4. Partition
Partitionstyp	83	82	0F	07
Bootflag	80 (aktiv)	00 (passiv)	00 (passiv)	00 (passiv)
Partitionsanfang (Spur/Kopf/Sektor)	0 / 1 / 1	2 / 0 / 1	264 / 0 / 1	1023/ 0 / 1
Partitionsende (Spur/Kopf/Sektor)	1 / 254 / 63	263 / 254/ 63	1023/ 254/ 63	1023/ 254/ 63
Sektoren in Partition	32 067	4 209 030	160 955 235	69 288 345
Partitionsanfang (ASN)	63	32 130	4 241 160	165 196 395

## Die Erweiterte Partition von PC-Festplatten

- Eine **Erweiterte Partition** (extended partition) kann **mehrere logische Laufwerke** enthalten  
→ Unterteilung in **Unterpartitionen (logische Partitionen)**

Die Definition und Beschreibung der einzelnen logischen Partitionen erfolgt über eine **verkettete Liste** von **Unterpartitionstabellen**.

Jede Unterpartitionstabelle befindet sich in einem **Beschreibungssektor**, der sich – jeweils gefolgt von einem verborgenen Bereich - vor jeder logischen Partition befindet.

Für n logische Partitionen (logische Laufwerke in der erweiterten Partition) werden n Beschreibungssektoren benötigt.

- **Aufbau einer Erweiterten Partition**

Beschreibungs-Sektor mit Unterpartitionstabelle	verborgener Bereich	1. Logische Partition (1. logisches Laufwerk)	Beschreibungs-Sektor mit Unterpartitionstabelle	verborgener Bereich	2. Logische Partition (2. logisches Laufwerk)	...
---	---------------------	--	---	---------------------	--	-----

1. erweiterte Unter-Partition

- **Aufbau eines Beschreibungssektors für logische Partitionen**

0x000	reservierter Bereich (mit 0x00 belegt)	446 Bytes
0x1BE	Unterpartitionstabelle (4 Einträge, max. 2 verwendet)	64 Bytes
0x1FE	Kennung für Bootsektoren (0x55 0xAA)	2 Bytes

Ein Beschreibungssektor für logische Partitionen ist **analog** zum **Master Boot Block aufgebaut**.

Allerdings enthält er kein Master Boot Programm, sondern stattdessen einen reservierten mit 00H vorbelegten Bereich.

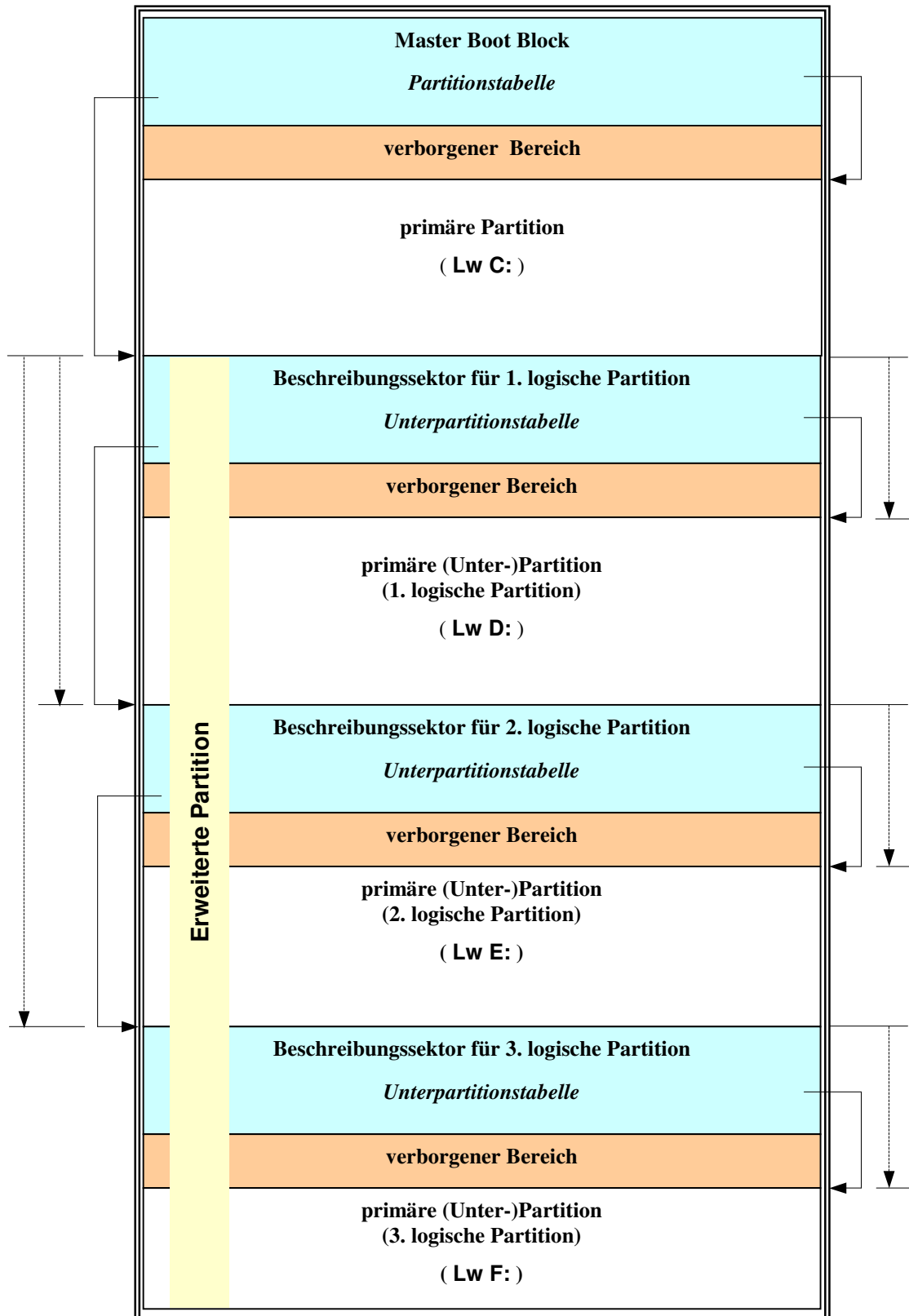
An der Stelle der Partitionstabelle befindet sich eine **Unterpartitionstabelle**.

Deren Aufbau und die Bedeutung ihrer einzelnen Einträge entspricht mit folgenden Abweichungen der Partitionstabelle im Master Boot Block :

- Von den 4 möglichen Einträgen sind maximal 2 belegt.  
Ein Eintrag beschreibt eine **primäre Unter-Partition** → das zugehörige **logische Laufwerk**  
Der zweite Eintrag dient der **Verkettung** zur nächsten logischen Partition. Er beschreibt formal eine **erweiterte Unter-Partition**, die den nächsten Beschreibungssektor, den sich daran anschließenden verborgenen Bereich und den Bereich einer möglichen weiteren logischen Partition umfaßt.
- Der Wert im Feld "**ASN des ersten Sektors der Partition**" der Unterpartitionstabelleneinträge bezieht sich nicht auf den Beginn des physikalischen Laufwerks (die CHS-Werte dagegen schon).  
Im Eintrag für die **primäre Unterpartition** bezieht sich dieser Wert auf den **Beginn** des jeweiligen **Beschreibungssektors** (also auf den Beginn der jeweiligen **erweiterten Unter-Partition**).
- Im Eintrag für die **erweiterte Unterpartition** bezieht sich dieser Wert dagegen auf den **Beginn** der gesamten **erweiterten Partition**

### Prinzipielles Beispiel für die Partitionierung einer PC-Festplatte

- Beispiel : Einteilung der Festplatte in 4 logische MS-DOS/WINDOWS-Laufwerke



## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS (1)

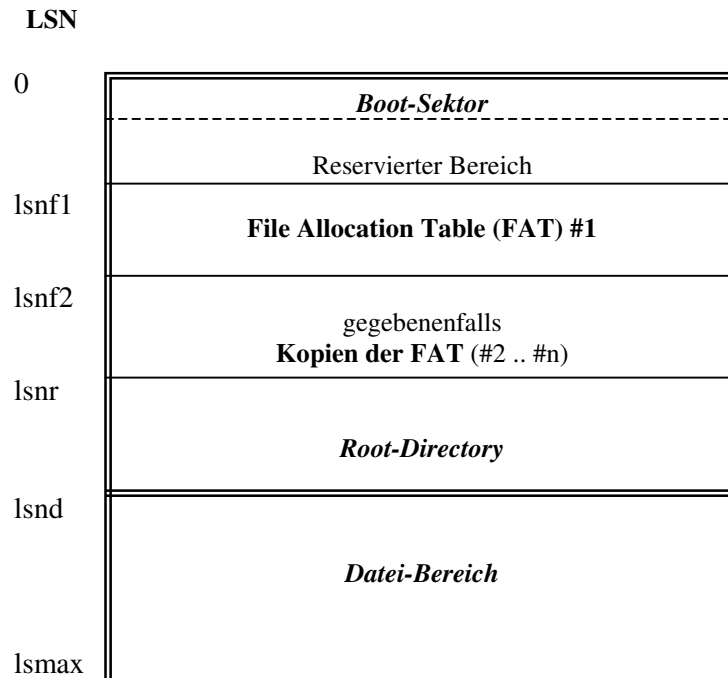
### • Allgemeines

- ✦ Dateisystem, das unter MS-DOS eingeführt und von WINDOWS übernommen wurde
- ✦ **Hierarchisches** – baumartig strukturiertes – Dateisystem.  
Neben einem **Root-Directory** ("Wurzel-Verzeichnis", "Haupt-Verzeichnis") können **Sub-Directories** ("Unter-Verzeichnisse") vorhanden sein.  
→ In Directories können neben Dateien auch Directories eingetragen sein.
- ✦ Das **Root-Directory** hat eine **feste** – beim Formatieren festgelegte – **Größe**.  
→ Anzahl der Einträge im Root-Directory ist begrenzt.  
Sub-Directories können dagegen beliebig groß werden.
- ✦ In der originalen Version sind **Datei-** (u. Directory-) **Namen** auf das **8.3-Format** beschränkt
- ✦ Eine einzige **zentrale Verwaltungsstruktur** dient sowohl zur "Buchführung" über freie und belegte Sektoren als auch zur "Buchführung" über die Sektor-Datei-Zuordnung : **Dateibelegungstabelle (File Allocation Table, FAT)**  
Dabei erfolgt die Verwaltung des Plattenspeicherplatzes clusterweise.  
Ein **Cluster** umfaßt einen oder mehrere (2-er Potenz) Sektoren.  
Neben einer Übersicht über die freien, belegten und defekten Cluster enthält die FAT **Verweisketten** für alle auf dem logischen Laufwerk abgelegten Dateien und Sub-Directories.

### • Aufteilung eines logischen Laufwerks

- ✦ Das Betriebssystem betrachtet ein **logisches Laufwerk** (Festplattenpartition oder Diskette) als eine **kontinuierliche Folge von logischen Sektoren**.  
Ein logischer Sektor wird über eine **logische Sektornummer (LSN)** im Bereich **0 .. lsmax** adressiert.

- ✦ **Aufteilung in Teilbereiche :**



- ✦ **Bootsektor**  
gehört zum reservierten Bereich, enthält im wesentlichen
  - Informationen über die Laufwerks-Parameter (**BIOS-Parameter-Block**)
  - **Bootstrap-Programm**

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS (2)

- Aufbau des Bootsektors** (ab DOS 4.0, erweiterter Bootsektor)

0x000	0xEB 0x3C 0x90 (Sprung zum Start des Bootprogramms + NOP)	3 Bytes
0x003	OEM-Identification (z.B. : MSWIN4.1)	8 Bytes
0x00B	Anzahl Bytes pro Sektor	2 Bytes
0x00D	Anzahl Sektoren pro Cluster	1 Byte
0x00E	Anzahl reservierter Sektoren	2 Bytes
0x010	Anzahl FATs	1 Byte
0x011	Anzahl der Einträge im Root-Directory	2 Bytes
0x013	Gesamtzahl Sektoren im logischen Laufwerk (wenn Kapazität <=32 MB; wenn Kapazität >32MB : 0x0000)	2 Bytes
0x015	Media Descriptor Byte (0xF8 für Hard Disks)	1 Byte
0x016	Anzahl Sektoren pro FAT	2 Bytes
0x018	Anzahl Sektoren pro Spur	2 Bytes
0x01A	Anzahl der Schreib-/Leseköpfe (Oberflächen)	2 Bytes
0x01C	Anzahl verborgener Sektoren in der Partition vor dem logischen Laufwerk ⚙)	4 Bytes
0x020	Gesamtzahl der Sektoren im logischen Laufwerk (wenn Kapazität > 32 MB; wenn Kapazität <=32 MB : reserviert (0x00000000))	◆) 4 Bytes
0x024	Physikalische Laufwerks-Nr (0x80 oder 0x00)	◆) 1 Byte
0x025	reserviert (0x00)	◆) 1 Byte
0x026	Kennung für erweiterten Bootsektor (0x29)	◆) 1 Byte
0x027	Seriennummer des logischen Laufwerks	◆) 4 Bytes
0x02B	Datenträgername (Volume Label)	◆) 11 Bytes
0x036	Dateisystem-Name ("FAT12 " oder "FAT16 ")	◆) 8 Bytes
0x03E	Boot-Code (Bootstrap-Programm)	448 Bytes
0x1FE	Kennung für Bootsektoren (0x55 0xAA)	2 Bytes

⚙) im ursprünglichen Bootsektor (vor DOS 4.0) 2 Bytes groß

◆) im ursprünglichen Bootsektor (vor DOS 4.0) nicht vorhanden

**BIOS-Parameter Block (BPB) :** Einträge von Offset 0x00B bis 0x023 (einschließlich)

### Das FAT12/16-Dateisystem von MS-DOS/WINDOWS (3)

- Beispiel für einen Bootsektor (ab DOS 4.0)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000	EB	3C	90	4D	53	57	49	4E	34	2E	31	00	02	20	01	00	.<.MSWIN4.1.. ..
0010	02	00	02	00	00	F8	FF	00	3F	00	FF	00	3F	00	00	00	.....?....?
0020	C3	DD	1F	00	80	00	29	05	12	69	1E	20	20	20	20	20	.....).i.
0030	20	20	20	20	20	20	46	41	54	31	36	20	20	20	33	C9	FAT16 3.
0040	8E	D1	BC	FC	7B	16	07	BD	78	00	C5	76	00	1E	56	16	....{...x..v..V.
0050	55	BF	22	05	89	7E	00	89	4E	02	B1	0B	FC	F3	A4	06	U."...~..N.....
0060	1F	BD	00	7C	C6	45	FE	0F	38	4E	24	7D	20	8B	C1	99	... .E..8N\$} ...
0070	E8	7E	01	83	EB	3A	66	A1	1C	7C	66	3B	07	8A	57	FC	~....:f... f;..W.
0080	75	06	80	CA	02	88	56	02	80	C3	10	73	ED	33	C9	FE	u.....V.....s.3..
0090	06	D8	7D	8A	46	10	98	F7	66	16	03	46	1C	13	56	1E	..}.F...f..F..V.
00A0	03	46	0E	13	D1	8B	76	11	60	89	46	FC	89	56	FE	B8	.F.....v..`F..V..
00B0	20	00	F7	EF	8B	5E	0B	03	C3	48	F7	F3	01	46	FC	11	....^....H...F..
00C0	4E	FE	61	BF	00	07	E8	28	01	72	3E	38	2D	74	17	60	N.a....(.r>8-t..`
00D0	B1	0B	BE	D8	7D	F3	A6	61	74	3D	4E	74	09	83	C7	20	....}.at=Nt...
00E0	3B	FB	72	E7	EB	DD	FE	0E	D8	7D	7B	A7	BE	7F	7D	AC	;r.....}{...}.
00F0	98	03	F0	AC	98	40	74	0C	48	74	13	B4	0E	BB	07	00	.....@t.Ht.....
0100	CD	10	EB	EF	BE	82	7D	EB	E6	BE	80	7D	EB	E1	CD	16	.....}. ....}
0110	5E	1F	66	8F	04	CD	19	BE	81	7D	8B	7D	1A	8D	45	FE	^f.....}.}..E.
0120	8A	4E	0D	F7	E1	03	46	FC	13	56	FE	B1	04	E8	C2	00	.N....F..V.....
0130	72	D7	EA	00	02	70	00	52	50	06	53	6A	01	6A	10	91	r....p.RP.Sj.j..
0140	8B	46	18	A2	26	05	96	92	33	D2	F7	F6	91	F7	F6	42	.F..&...3.....B
0150	87	CA	F7	76	1A	8A	F2	8A	E8	C0	CC	02	0A	CC	B8	01	...v.....
0160	02	80	7E	02	0E	75	04	B4	42	8B	F4	8A	56	24	CD	13	...~..u..B...V\$..
0170	61	61	72	0A	40	75	01	42	03	5E	0B	49	75	77	C3	03	aar.@u.B.^.Iuw..
0180	18	01	27	0D	0A	55	6E	67	75	65	6C	74	69	67	65	73	..'..Ungueltiges
0190	20	53	79	73	74	65	6D	20	FF	0D	0A	45	2F	41	2D	46	System ...E/A-F
01A0	65	68	6C	65	72	20	20	20	20	FF	0D	0A	44	61	74	65	ehler ...Date
01B0	6E	74	72	61	65	67	65	72	20	77	65	63	68	73	65	6C	ntraeger wechsel
01C0	6E	20	75	6E	64	20	54	61	73	74	65	20	64	72	75	65	n und Taste drue
01D0	63	6B	65	6E	0D	0A	00	00	49	4F	20	20	20	20	20	20	cken....IO
01E0	53	59	53	4D	53	44	4F	53	20	20	20	53	59	53	7F	01	SYSMSDOS SYS..
01F0	00	41	BB	00	07	60	66	6A	00	E9	3B	FF	00	00	55	AA	.A...`fj...;...U.

- Informationen des BPBs aus obigem Bootsektor :

0x003	: OEM-Identifikation	:	MSWIN4.1	
0x00B	: Anzahl Bytes pro Sektor	:	512	(0x0200)
0x00D	: Anzahl Sektoren pro Cluster	:	32	(0x20)
0x00E	: Anzahl reservierter Sektoren	:	1	
0x010	: Anzahl FATs	:	2	
0x011	: Anzahl Einträge im Root-Directory	:	512	(0x0200)
0x020	: Gesamtzahl Sektoren	:	2 088 387	(0x001FDDC3)
0x015	: Media Descriptor Byte	:	0xF8	
0x016	: Anzahl Sektoren pro FAT	:	255	(0x00FF)
0x018	: Anzahl Sektoren pro Spur	:	63	(0x003F)
0x01A	: Anzahl Schreib-/Leseköpfe	:	255	(0x00FF)
0x01C	: Anzahl verborgener Sektoren	:	63	(0x0000003F)



## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS (4)

### • Zusammenhang zwischen Cluster-Nr. und logischer Sektor-Nr.

- ◊ **Sektor** : Zugriffs-/Transfer-Einheit
- Cluster** : Verwaltungseinheit (Allokationseinheit)

- ◊ **CLN** Cluster-Nr
- LSN** logische Sektor-Nr. (des 1. Sektors des Clusters)

Die Zusammenfassung von Sektoren zu Clustern erfolgt **nur** für den **Datei-Bereich** des logischen Laufwerks.  
Die Werte 0 und 1 werden nicht für Cluster-Nummern verwendet.  
⇒ der **erste Cluster** im Datei-Bereich hat die **CLN 2**

$$\text{LSN} = (\text{CLN} - 2) * \text{spc} + \text{lsnd}$$

**spc** Sektoren pro Cluster (im BPB enthalten)  
**lsnd** LSN des ersten Sektors des Dateibereichs  
(aus BPB-Information ermittelbar)

- ◊ **lsnd** läßt sich aus den im BPB enthaltenen Informationen wie folgt ermitteln :

$$\text{lsnd} = \text{ars} + \text{afat} * \text{spf} + \text{erd} * 32 / \text{bps}$$

**ars** Anzahl reservierter Sektoren  
(Boot-Sektor + eventuelle weitere Sektoren)  
**afat** Anzahl FATs  
**spf** Anzahl Sektoren pro FAT  
**erd** Anzahl Einträge im Root-Directory  
**bps** Anzahl Bytes pro Sektor

Reservierter Bereich (Boot-Sektor + ...)	Verwaltungs-Bereich		Datei-Bereich			
	FATs	Root Directory				

z.B. für **spc=4** : **CLN**

**LSN**

2				...
lsnd	lsnd+1	lsnd+2	lsnd+3	...

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS (5)

### • Problematik der Cluster-Bildung :

- ◇ Da die Allokation von Speicherplatz auf dem logischen Laufwerk clusterweise erfolgt, belegt auch die kleinste Datei immer **mindestens einen Cluster**.  
Jede Datei, deren Länge nicht ein Vielfaches der Clustergröße beträgt, belegt mehr Speicher, als sie eigentlich benötigt. Diese "**Verschwendung**" von **Speicherplatz** (→ **Slack**) steigt mit zunehmender Clustergröße.  
Eine Erhöhung der Clustergröße verringert also die Effizienz der Nutzung des Plattenspeichers. Dabei hängt der tatsächliche **Effizienzverlust** von den jeweiligen Dateigrößen ab.

Beispiel :

```
Scanned 7,243 files in 381 directories on drive C:
Cumulative length of all files is 623,803,432 bytes
Cluster size is 16,384 bytes (16K)
```

Cluster Size	Overhang (Bytes)	Efficiency
2K	7,211,992	98.9%
4K	15,156,184	97.6%
8K	33,915,864	94.8%
16K	74,154,968	89.4% <---
32K	162,431,968	79.3%
64K	353,207,256	63.8%

- ◇ Andererseits wird mit steigender Clustergröße - bei gleicher Laufwerkskapazität - der von der FAT belegte Speicherplatz geringer. Dies wirkt der Effizienzverringerng etwas entgegen.
- ◇ Die **maximale Größe einer FAT** wird von der maximal möglichen Clusternummer bestimmt.  
Diese wiederum ist durch die Darstellungslänge von Cluster-Nummern festgelegt.  
Bei einer Länge von 12 Bit (**FAT12**) beträgt sie **4095**, bei einer Länge von 16 Bit (**FAT16**) beträgt sie **65535**.  
⇒ Die maximale FAT-Größe beträgt
  - bei einer **12-Bit-FAT** 12 Sektoren( = **6 kBytes**)
  - bei einer **16-Bit-FAT** 256 Sektoren ( = **128 kBytes**)
- ◇ Eine festliegende **maximale FAT-Größe** erzwingt mit steigender **Laufwerkskapazität** eine **Erhöhung der Clustergröße** :

Laufwerkskapazität	FAT-Type	Sektoren/Cluster	Clustergröße
< 16 MB	12-Bit	8	4 kB
16 MB - 127 MB	16-Bit	4	2 kB
128 MB - 255 MB	16-Bit	8	4 kB
256 MB - 511 MB	16-Bit	16	8 kB
512 MB - 1023 MB	16-Bit	32	16 kB
1024 MB - 2047 MB	16-Bit	64	32 kB
2048 MB - 4096 MB	16-Bit	128	64 kB

### • Grenzen für die Laufwerkskapazität :

- ◇ **2 GB-Grenze** :  
Viele existierende Programme setzen voraus, daß die Clustergröße (Anzahl Bytes pro Cluster) in einem Wort (16 Bits) dargestellt werden kann. 64 kB (=65536) ist hierfür zu groß (→ keine wirkliche Betriebssystemgrenze !)  
→ Die maximale, mit 16 Bits darstellbare, Clustergröße ist 32 kB ( = 32768). → max. Laufwerkskapazität : 2 GB
- ◇ **4-GB-Grenze** :  
MS-DOS/WINDOWS speichern die Anzahl Sektoren pro Cluster in einem Byte. Diese Anzahl muß eine 2-er Potenz sein. Die größte in einem Byte darstellbare 2-er Potenz ist 128 → maximale Clustergröße : 64 kB  
→ max. Laufwerkskapazität : 4 GB

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – File Allokation Table (1)

- Die **File Allokation Table (FAT)** enthält Informationen über die Belegung der einzelnen Cluster des logischen Laufwerks durch die Dateien und Sub-Directories (→ **Dateibelegungstabelle**)  
Aus Sicherheitsgründen ist sie i.a. **mehrfach** (meist zweifach) vorhanden.  
Die FATs befinden sich zwischen Boot-Sektor und Root-Directory .
- Die FAT ist ein **Feld von Einträgen**  
**Jeder Komponente** dieses Feldes ist eindeutig **ein Cluster zugeordnet** (Index der Feldkomponente = Clusternummer).  
In ihr ist eingetragen, ob der entsprechende Cluster
  - ▷ **frei** (d.h. unbelegt),
  - ▷ **defekt** (und damit nicht verwendbar)
  - ▷ oder **belegt** istIm **Fall der Belegung** ist vermerkt,
  - ▷ ob der entsprechende Cluster der **letzte Cluster** einer Datei ist
  - ▷ bzw welches der **Folge-Cluster** der Datei (zu der der Cluster gehört) ist (→ **Verweiskette**).
- Wenn das log. Laufwerk **≤4085 Cluster** im Datei-Bereich aufweist, ist **jeder Eintrag** in der FAT **12 Bit** lang (→ **12-Bit-FAT**),  
wenn es **>4085 Cluster** im Datei-Bereich hat, ist **jeder Eintrag** in der FAT **16 Bit** lang (→ **16-Bit-FAT**).
- Die **ersten beiden Einträge** in der FAT sind immer **reserviert**. Ihnen sind **keine Cluster** im Datei-Bereich **zugeordnet**, d.h. der Datei-Bereich beginnt immer mit der **Cluster-Nummer (CLN) 2**.

Die beiden **reservierten Einträge** umfassen **3 Bytes** (12-Bit-FAT) bzw **4 Bytes** (16-Bit-FAT).  
Das **erste** dieser Bytes enthält das **Media Descriptor Byte** (Mediabyte).

Einige **Beispiele** für Media Descriptor Bytes :

- ◇ 0x**F0** 1.44 MBytes-Diskette
- ◇ 0x**F8** Festplatte, Ram-Disk
- ◇ 0x**F9** 1.2 MBytes-Diskette, 720 KBytes-Diskette
- ◇ 0x**FD** 360 KByte-Diskette (DS, 40 Spuren, 9 Sekt/Spur)

Das **2. und 3.** (bzw **2. bis 4.**) der reservierten Bytes enthalten immer 0x**FF**.

- Die **Werte** in den anschließenden (eigentlichen Nutz-) **Einträgen** der FAT haben folgende **Bedeutung** :
  - ▷ 0x(0)000 Cluster ist frei
  - ▷ 0x(0)001 dieser Wert darf nicht auftreten
  - ▷ 0x(0)002 }  
... } Nummer des Folge-Clusters  
0x(F)FF6 }
  - ▷ 0x(F)FF7 Cluster ist defekt
  - ▷ 0x(F)FF8 }  
... } Cluster ist letzter Cluster einer Datei  
0x(F)FFF }

⇒ Die **FAT** enthält für **jede Datei** und jedes Sub-Directory eine **verkettete Folge** der von dieser Datei / diesem Sub-Directory belegten **Cluster** (→ **Verweiskette**).  
Die Nummer des **ersten Clusters** der Datei / des Sub-Directories (und damit der Start der Kette) steht im **Directory-eintrag**.

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – File Allokation Table (2)

- Anfang einer 16-Bit FAT (Beispiel)

0000	F8 FF FF FF 03 00 04	00-05 00 06 00 07 00 08 00	.....
0010	09 00 0A 00 0B 00 0C	00-FF FF 0E 00 0F 00 10 00	.....
0020	11 00 12 00 13 00 14	00-15 00 16 00 17 00 18 00	.....
0030	19 00 1A 00 1B 00 FF	FF-1D 00 1E 00 1F 00 20 00	.....
0040	21 00 22 00 23 00 24	00-25 00 26 00 27 00 28 00	!.\". #.\$.%&.'.(.
0050	17 17 33 00 FF FF 00	00-FF FF FF FF FF FF FF	..3.....
0060	FF FF FF FF FF FF FF	FF-CA 00 FF FF 37 00 FF FF	.....7...
0070	39 00 3A 00 3B 00 3C	00-3D 00 3E 00 3F 00 40 00	9.:.;.<.=.>.?.@.
0080	41 00 42 00 43 00 44	00-FF FF 46 00 FF FF FF FF	A.B.C.D...F.....
0090	49 00 4A 00 4B 00 4C	00-4D 00 FF FF FF FF FF FF	I.J.K.L.M.....
00A0	FF FF FF FF 53 00 FF	FF-55 00 FF FF FF FF FF FF	....S...U.....
00B0	FF FF FF FF 5B 00 5C	00-5D 00 5E 00 FF FF FF FF	....[.\.].^.....
00C0	FF FF FF FF FF FF FF	FF-65 00 66 00 67 00 68 00	.....e.f.g.h.
00D0	69 00 FF FF 6B 00 6C	00-6D 00 6E 00 6F 00 FF FF	i...k.l.m.n.o...

- Anfang des zugehörigen Root-Directories

0000	49 4F 20 20 20 20 20	20-53 59 53 27 00 00 00 00	IO SYS'....
0010	00 00 00 00 00 00 00	60-6B 0F 02 00 B3 57 00 00	.....`k....W..
0020	4D 53 44 4F 53 20 20	20-53 59 53 27 00 00 00 00	MSDOS SYS'....
0030	00 00 00 00 00 00 00	60-6B 0F 0D 00 D0 75 00 00	.....`k....u..
0040	4B 54 52 4F 4E 5F 33	5F-33 33 20 28 00 00 00 00	KTRON_3_33 (....
0050	00 00 00 00 00 00 00	76 89-2F 11 00 00 00 00 00	.....v./.....
0060	44 49 56 20 20 20 20	20-20 20 20 10 00 00 00 00	DIV .....
0070	00 00 00 00 00 00 30	75-CA 10 4E 00 00 00 00 00	.....0u..N.....
0080	43 4B 45 59 47 52 20	20-43 4F 4D 20 00 00 00 00	CKEYGR COM ....
0090	00 00 00 00 00 00 72	55-31 0F 36 00 DC 0C 00 00	.....rU1.6.....
00A0	43 4F 4D 4D 41 4E 44	20-43 4F 4D 20 00 00 00 00	COMMAND COM ....
00B0	00 00 00 00 00 00 00	60-6B 0F 38 00 DC 65 00 00	.....`k.8..e..
00C0	43 4F 55 4E 54 52 59	20-53 59 53 20 00 00 00 00	COUNTRY SYS ....
00D0	00 00 00 00 00 00 00	60-6B 0F 48 00 F6 2B 00 00	.....`k.H..+..
00E0	47 45 4D 20 20 20 20	20-42 41 54 20 00 00 00 00	GEM BAT ....
00F0	00 00 00 00 00 00 0B	67-37 10 47 00 B0 00 00 00	.....g7.G.....
0100	4D 45 4E 55 20 20 20	20-42 41 54 20 00 00 00 00	MENU BAT ....
0110	00 00 00 00 00 00 42	B1-7A 10 2C 00 6D 00 00 00	.....B.z.,.m...
0120	4D 46 4B 42 47 52 20	20-43 4F 4D 20 00 00 00 00	MFKBGR COM ....
0130	00 00 00 00 00 00 68	7F-56 0D 45 00 20 0D 00 00	.....h.V.E. ....
0140	4D 49 52 52 4F 52 20	20-46 49 4C 21 00 00 00 00	MIRROR FIL!....
0150	00 00 00 00 00 00 34	86-2F 11 1C 00 00 C6 00 00	.....4./.....
0160	52 45 53 45 54 20 20	20-42 41 54 20 00 00 00 00	RESET BAT ....
0170	00 00 00 00 00 00 FD	81-BF 10 2D 00 87 00 00 00	.....-.....
0180	53 45 54 49 4E 54 20	20-42 41 54 20 00 00 00 00	SETINT BAT ....
0190	00 00 00 00 00 00 6D	82-BF 10 2E 00 C1 00 00 00	.....m.....
01A0	53 45 54 4D 47 20 20	20-42 41 54 20 00 00 00 00	SETMG BAT ....
01B0	00 00 00 00 00 00 10	82-BF 10 2F 00 97 00 00 00	...../.....
01C0	53 45 54 4D 4C 20 20	20-42 41 54 20 00 00 00 00	SETML BAT ....
01D0	00 00 00 00 00 00 8E	82-BF 10 30 00 C7 00 00 00	.....0.....
01E0	53 45 54 4D 53 43 20	20-42 41 54 20 00 00 00 00	SETMSC BAT ....
01F0	00 00 00 00 00 00 31	82-BF 10 31 00 BC 00 00 00	.....1...1.....
0200	53 45 54 4D 53 50 20	20-42 41 54 20 00 00 00 00	SETMSP BAT ....
0210	00 00 00 00 00 00 E2	71-27 10 32 00 76 00 00 00	.....q'.2.v....
0220	53 45 54 57 53 20 20	20-42 41 54 20 00 00 00 00	SETWS BAT ....
0230	00 00 00 00 00 00 B2	82-BF 10 35 00 96 00 00 00	.....5.....

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – File Allokation Table (3)

### • Anfang einer 12-Bit FAT (Beispiel)

0000	F9 FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B	....@...`.....
0010	C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60	..... ..@..`
0020	01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02	.....
0030	21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B	! .#@.%`.'...)+
0040	C0 02 2D F0 FF 2F 00 03-31 20 03 33 40 03 35 60	..-../..1 .3@.5`
0050	03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04	.7..9..;..=..?..
0060	41 20 04 43 40 04 45 60-04 47 80 04 49 A0 04 4B	A .C@.E`.G..I..K
0070	C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60	..M..O..Q .S@.U`
0080	05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 5F 00 06	.W..Y..[...]._..
0090	61 20 06 63 40 06 65 60-06 67 80 06 FF 1F 21 FF	a .c@.e`.g.....!
00A0	FF FF FF EF 06 6F 00 07-71 20 07 73 40 07 75 60	.....o..q .s@.u`
00B0	07 77 80 07 79 A0 07 7B-C0 07 7D E0 07 7F 00 08	.w..y..{...}.....
00C0	81 20 08 83 40 08 85 60-08 87 80 08 89 A0 08 8B	. ..@...`.....
00D0	C0 08 8D E0 08 8F 00 09-91 20 09 93 40 09 95 60	..... ..@..`
00E0	09 97 80 09 99 A0 09 9B-C0 09 9D E0 09 9F F0 FF	.....
00F0	A1 20 0A A3 40 0A A5 60-0A FF FF A9 A0 0A AB	. ..@...`.....
0100	C0 0A AD E0 0A AF 00 0B-B1 20 0B B3 40 0B B5 60	..... ..@..`
0110	0B B7 80 0B B9 A0 0B BB-C0 0B BD F0 FF FF 0F 0C	.....
0120	C1 20 0C C3 40 0C C5 60-0C C7 80 0C C9 A0 0C CB	. ..@...`.....

### • Anfang des zugehörigen Root-Directories

0000	49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00	IO SYS'....
0010	00 00 00 00 00 00 00 60-6B 0F 02 00 B3 57 00 00	.....`k....W..
0020	4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00	MSDOS SYS'....
0030	00 00 00 00 00 00 00 60-6B 0F 2E 00 D0 75 00 00	.....`k....u..
0040	52 54 5F 44 4F 53 33 33-20 20 20 28 00 00 00 00	RT_DOS33 (....
0050	00 00 00 00 00 00 38 90-2D 11 00 00 00 00 00 00	.....8.-.....
0060	44 49 56 20 20 20 20 20-20 20 20 10 00 00 00 00	DIV .....
0070	00 00 00 00 00 00 56 90-2D 11 A7 00 00 00 00 00	.....V.-.....
0080	44 52 49 56 45 52 20 20-20 20 20 10 00 00 00 00	DRIVER .....
0090	00 00 00 00 00 00 27 5F-89 10 6A 00 00 00 00 00	.....'...j.....
00A0	53 59 53 20 20 20 20 20-20 20 20 10 00 00 00 00	SYS .....
00B0	00 00 00 00 00 00 23 5F-89 10 69 00 00 00 00 00	.....#...i.....
00C0	55 53 52 20 20 20 20 20-20 20 20 10 00 00 00 00	USR .....
00D0	00 00 00 00 00 00 2E 5F-89 10 6C 00 00 00 00 00	....._...l.....
00E0	55 54 49 4C 20 20 20 20-20 20 20 10 00 00 00 00	UTIL .....
00F0	00 00 00 00 00 00 2B 5F-89 10 6B 00 00 00 00 00	.....+...k.....
0100	41 55 54 4F 45 58 45 43-42 41 54 20 00 00 00 00	AUTOEXECBAT ....
0110	00 00 00 00 00 00 C3 69-89 10 7E 08 CA 00 00 00	.....i...~.....
0120	43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00	COMMAND COM ....
0130	00 00 00 00 00 00 00 60-6B 0F 6D 00 DC 65 00 00	.....`k.m..e..
0140	43 4F 4E 46 49 47 20 20-53 59 53 20 00 00 00 00	CONFIG SYS ....
0150	00 00 00 00 00 00 4E 45-8F 10 BE 00 D8 00 00 00	.....NE.....
0160	43 4F 55 4E 54 52 59 20-53 59 53 20 00 00 00 00	COUNTRY SYS ....
0170	00 00 00 00 00 00 00 60-6B 0F A8 00 F6 2B 00 00	.....`k....+..
0180	4D 46 4B 42 47 52 20 20-43 4F 4D 20 00 00 00 00	MFKBGR COM ....
0190	00 00 00 00 00 00 68 7F-56 0D A0 00 20 0D 00 00	.....h.V....
01A0	E5 4F 54 45 53 34 20 20-54 58 54 20 00 00 00 00	.OTES4 TXT ....
01B0	00 00 00 00 00 00 C2 90-2D 11 7D 08 27 01 00 00	.....-..}..'...
01C0	E5 43 4F 50 59 20 20 20-45 58 45 20 00 00 00 00	.COPY EXE ....
01D0	00 00 00 00 00 00 00 60-6B 0F 97 08 DA 2C 00 00	.....`k.....,
01E0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00	.....
01F0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00	.....

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – File Allokation Table (4)

### • Ermittlung von FAT-Einträgen

◇ Als **Offset** in die FAT dient die **Cluster-Nr** CLN

#### ◇ 16-Bit-FAT

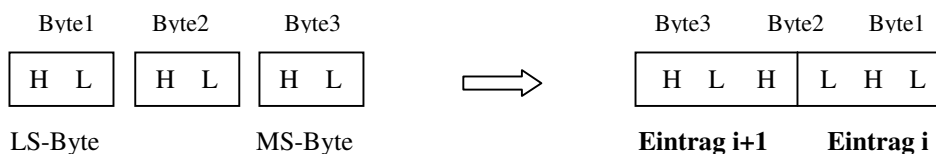
Je **2 Bytes** enthalten **einen** 16-Bit-Eintrag

<b>CLN mit 2 multiplizieren</b> → Byte-Offset des Eintrags in der FAT
<b>FAT-Eintrag :=</b> <b>das durch den Offset bestimmte Wort</b> ( = Byte des Offsets und nächstes Byte )

#### ◇ 12-Bit-FAT

Je **3 Bytes** enthalten **zwei** 12-Bit-Einträge :

- Eintrag i (entspricht Cluster-Nr. CLN=i) (i gerade !!)
- Eintrag i+1 (entspricht Cluster-Nr. CLN=i+1)



<b>CLN mit 1.5 multiplizieren</b>
<b>ganzzahliger Anteil</b> → Byte-Offset des Eintrags in der FAT
<b>das durch den Offset bestimmte Wort</b> ( = Byte des Offsets und nächstes Byte ) der Tabelle entnehmen
<b>CLN gerade ?</b> <div style="display: flex; justify-content: space-around; font-weight: bold; font-size: 1.5em;"> <span>J</span> <span>N</span> </div>
<b>FAT-Eintrag :=</b> <b>niederwertige 12 Bits</b> des entnommenen Worts
<b>FAT-Eintrag:=</b> <b>höherwertige 12 Bits</b> des entnommenen Worts

**Beispiel :**

Anfang der FAT : 0000 F9 FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  
0010 C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60

Eintrag zu CLN=2 : Byte-Offset = 3 → 03 40 → 4003 gerade CLN → **003**

Eintrag zu CLN=3 : Byte-Offset = 4 → 40 00 → 0040 ungerade CLN → **004**

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – Directory-Einträge (1)

### • Directories

Das **Root-Directory** steht an **bestimmter Stelle** und hat eine **festliegende Länge**.

Es kann daher nur eine festliegende **begrenzte Anzahl Einträge** aufnehmen.

Ein **Sub-Directory** dagegen kann an **beliebiger Stelle** im Datenbereich stehen.

Seine Länge und damit die **Anzahl möglicher Einträge** ist **nur** durch die **Kapazität des Datenträgers begrenzt**

### • Aufbau eines Directory-Eintrags :

Jeder Directory-Eintrag umfaßt **32 Bytes**.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	Dateihauptname								Extension		Attr.		reserviert			
0x10	reserviert						Zeit		Datum		CLN		Dateilänge			

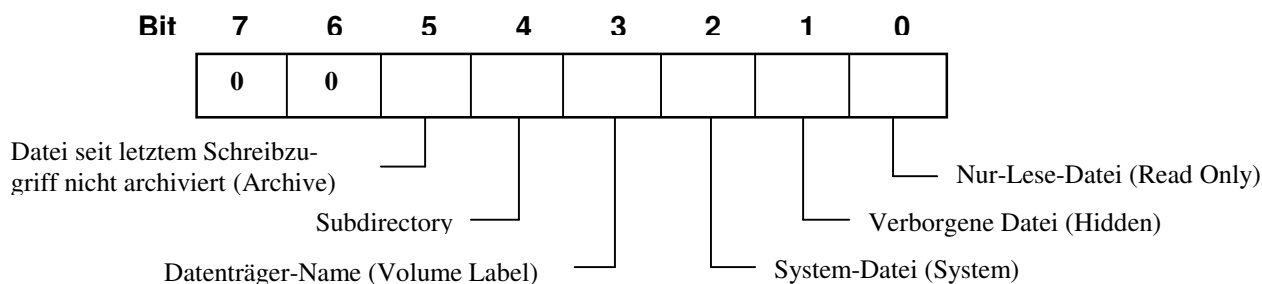
Zeit/Datum : Zeit und Datum der letzten Datei-Änderung  
Attr. : Dateiattribut  
CLN : Cluster-Nummer des Beginns der Datei

### • Besondere Bedeutungen des ersten Bytes des Dateihauptnamens :

- ◇ 0x00 : Der Directory-Eintrag ist noch **nicht verwendet** worden (Ende des belegten Bereichs des Directories)
- ◇ 0xE5 : Der Eintrag ist **gelöscht** und damit frei  
(Die anderen Bytes eines gelöschten Eintrags bleiben bis zum Überschreiben durch einen neuen Eintrag erhalten)
- ◇ 0x05 : Das erste Zeichen des Dateihauptnamnes ist tatsächlich 0xE5
- ◇ 0x2E : - nächstes Byte = 0x20 : Der Eintrag verweist auf das **Directory selbst** (" . ")  
CLN enthält die Nummer des 1. Clusters des Directories selbst.  
- nächstes Byte = 0x2E (und übernächstes Byte = 0x20) :  
Der Eintrag verweist auf das **Eltern-Directory** (" . . ")  
CLN enthält die Nummer des ersten Clusters des Eltern-Directories,  
bzw 0x0000, wenn es sich dabei um das Root-Directory handelt.

### • Attribut-Byte

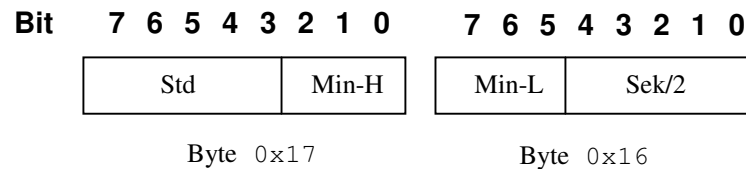
Ein gesetztes Bit definiert die ihm zugeordnete Eigenschaft.



## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – Directory-Einträge (2)

### • Codierung der Zeit

- ◇ Darstellung von Std / Min / Sek dual in 2 Bytes :
  - Std in 5 Bits
  - Min in 6 Bits
  - Sek in 5 Bits → Ablegung von "Doppelsekunden" (Zahlenwert : Sek/2)



- ◇ Beispiel :
 

0x90

1 0 0 1 0 0 0 0

|     18     |     2     |

0x56

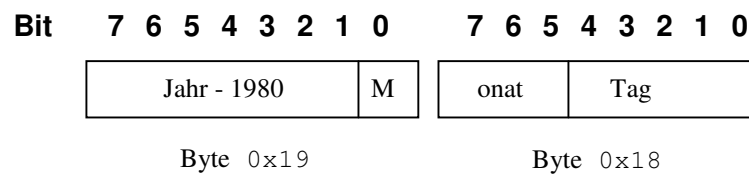
0 1 0 1 0 1 1 0

|     22     |

⇒ 18.02:44

### • Codierung des Datums

- ◇ Darstellung von Jahr / Monat / Tag dual in 2 Bytes :
  - Jahr in 7 Bits → Ablegung des Jahrs relativ zum Jahr 1980 (Zahlenwert : Jahr – 1980)
  - Monat in 4 Bits
  - Tag in 5 Bits



- ◇ Beispiel :
 

0x27

0 0 1 0 0 1 1 1

|     19     |     8     |

0x16

0 0 0 1 0 1 1 0

|     22     |

⇒ 22.8.1999



## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – Directory-Einträge (3)

- Beispiel für den Anfang eines Directories (hier : Root-Directory einer Diskette)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000	42	4F	4F	54	5F	44	4F	53	20	20	20	28	00	00	00	00	BOOT_DOS (....
0010	00	00	00	00	00	00	00	89	74	16	27	00	00	00	00	00	.....t.'.....
0020	49	4F	20	20	20	20	20	20	53	59	53	07	00	00	00	00	IO.....SYS.....
0030	00	00	00	00	00	00	C0	B2	A5	26	02	00	B6	64	03	00	.....&...d..
0040	44	52	56	53	50	41	43	45	42	49	4E	07	00	00	00	00	DRVSPACEBIN.....
0050	00	00	00	00	00	00	C0	B2	A5	26	B5	01	D7	0D	01	00	.....&.....
0060	4D	53	44	4F	53	20	20	20	53	59	53	07	00	00	00	00	MSDOS.....SYS.....
0070	00	00	00	00	00	00	09	75	16	27	3C	02	06	00	00	00	.....u.'<.....
0080	43	4F	4D	4D	41	4E	44	20	43	4F	4D	01	00	00	00	00	COMMAND.COM.....
0090	00	00	00	00	00	00	C0	B2	A5	26	3D	02	72	78	01	00	.....&=.rx..
00A0	E5	4F	4E	46	49	47	20	20	42	41	4B	20	00	00	00	00	.ONFIG BAK.....
00B0	00	00	00	00	00	00	15	A5	0B	27	FA	02	ED	01	00	00	.....&.....
00C0	41	55	54	4F	45	58	45	43	42	41	54	20	00	00	00	00	AUTOEXECBAT.....
00D0	00	00	00	00	00	00	15	A5	0B	27	FB	02	B6	01	00	00	.....&.....
00E0	48	49	4D	45	4D	20	20	20	53	59	53	20	00	00	00	00	HIMEM.....SYS.....
00F0	00	00	00	00	00	00	C0	B2	A5	26	FC	02	A7	82	00	00	.....&.....
0100	45	4D	4D	33	38	36	20	20	45	58	45	20	00	00	00	00	EMM386 EXE.....
0110	00	00	00	00	00	00	C0	B2	A5	26	3E	03	E7	EE	01	00	.....&>.....
0120	41	4E	53	49	20	20	20	20	53	59	53	20	00	00	00	00	ANSI.....SYS.....
0130	00	00	00	00	00	00	C0	B2	A5	26	36	04	07	26	00	00	.....&6..&..
0140	44	4F	53	48	45	59	20	20	43	4F	4D	20	00	00	00	00	DOSKEY.....COM.....
0150	00	00	00	00	00	00	C0	B2	A5	26	4A	04	A7	3D	00	00	.....&J..=..
0160	4B	45	59	42	20	20	20	20	43	4F	4D	20	00	00	00	00	KEYB.....COM.....
0170	00	00	00	00	00	00	C0	B2	A5	26	69	04	37	4E	00	00	.....&i.7N..
0180	4B	45	59	42	4F	41	52	44	53	59	53	20	00	00	00	00	KEYBOARDSYS.....
0190	00	00	00	00	00	00	C0	B2	A5	26	91	04	06	87	00	00	.....&.....
01A0	43	4F	4E	46	49	47	20	20	53	59	53	20	00	00	00	00	CONFIG.....SYS.....
01B0	00	00	00	00	00	00	72	76	16	27	D5	04	76	01	00	00	.....rv.'..v...
01C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....&.....
01D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....&.....
01E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....&.....
01F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....&.....

```
"WIN98"*C:\DIV>dir a:
```

```
Datenträger in Laufwerk A: BOOT_DOS
Seriennummer des Datenträgers: 2F45-15F2
Verzeichnis von A:\
```

```
IO          SYS          222.390    05.05.99    22:22 IO.SYS
DRVSPACE   BIN          69.079    05.05.99    22:22 DRVSPACE.BIN
MSDOS      SYS           6        22.08.99    14:40 MSDOS.SYS
COMMAND    COM          96.370    05.05.99    22:22 COMMAND.COM
AUTOEXEC   BAT           438    11.08.99    20:40 AUTOEXEC.BAT
HIMEM      SYS          33.447    05.05.99    22:22 HIMEM.SYS
EMM386     EXE         126.695    05.05.99    22:22 EMM386.EXE
ANSI       SYS           9.735    05.05.99    22:22 ANSI.SYS
DOSKEY     COM          15.783    05.05.99    22:22 DOSKEY.COM
KEYB       COM          20.023    05.05.99    22:22 KEYB.COM
KEYBOARD   SYS          34.566    05.05.99    22:22 KEYBOARD.SYS
CONFIG     SYS           374    22.08.99    14:51 CONFIG.SYS
          12 Datei(en)                628.906 Bytes
          0 Verzeichnis(se)            825.344 Bytes frei
```

```
"WIN98"*C:\DIV>_
```

## Das FAT12/16-Dateisystem von MS-DOS/WINDOWS – Entlöschten von Dateien

### • Das Löschen einer Datei bewirkt :

- ◇ Im **Directory-Eintrag** der Datei wird das **erste Byte** (= 1. Zeichen des Dateinamens) auf **0xE5** gesetzt.  
→ Dir-Eintrag steht für einen Neueintrag zur Verfügung

Der **Rest** des Dir-Eintrags bleibt **unverändert**

```
00C0  E5 54 44 50 4C 20 20 20-43 20 20 20 00 00 00 00  .TDPL  C  ....
00D0  00 00 00 00 00 00 00 F8 8C-F4 10 6B 00 28 33 00 00  .....k.(3..
```

- ◇ In der **FAT** werden die **Einträge** für alle **Cluster**, die von der Datei **belegt** waren auf **0x(0)000** gesetzt.  
→ Cluster als frei gekennzeichnet

```
0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 FF  a .c@.e`.g..i...
00A0  0F 00 00 00 00 00 00 F0 FF-71 20 07 73 40 07 75 60  .....q .s@.u`
```

- ◇ Die eigentlichen Daten, d.h. der **Dateiinhalt**, werden **nicht gelöscht**.  
Sie bleiben solange auf dem Datenträger, bis sie von einer neuen Datei überschrieben werden.

### • Strategie für das Entlöschten (*undelete*)

- ◇ Folgende Vorgehensweise ermöglicht die Wiederherstellung einer – versehentlich gelöschten – Datei :

- ▷ **Wiederherstellen des 1. Zeichens** des **Dateinamens** im **Dir-Eintrag** der zu entlöschenden Datei
- ▷ Entnahme der **Dateilänge** aus dem Dir-Eintrag
- ▷ Ermittlung der **Anzahl der Cluster**, die von der Datei belegt sind, aus der Dateilänge
- ▷ Entnahme der **CLN des ersten** von der Datei belegten **Clusters**
- ▷ Ermittlung des **Byte-Offsets** in der **FAT** aus dieser CLN
- ▷ **Rekonstruktion** der von der Datei belegten **Cluster-Kette** in der FAT  
(Der Datei werden soviel "freie" Cluster zugeordnet, wie sie entsprechend ihrer Länge belegen muß;  
in allen FATs durchführen !)

**Problem:** Zuordnung der richtigen Cluster, wenn die Datei fraktioniert ist und zeitlich vor oder nach der Datei andere Dateien gelöscht wurden.

Bei **Textdateien** u.U. durch "**Anschauen**" der als frei gekennzeichneten Cluster lösbar,  
bei **Binärdateien** i.a. **keine Chance**.

- ◇ **Voraussetzung** für diese Vorgehensweise : Kenntnis der folgenden Informationen  
(aus BPB entnehmbar bzw. ermittelbar) :

- ▷ Anzahl reservierter Sektoren **ars** → LSN des FAT-Beginns **lsnf1**
- ▷ Anzahl Sektoren pro FAT **spf** → LSN des Beginns weiterer FATs
- ▷ LSN Anfang Root-Directory **lsnr**
- ▷ Sektoren pro Cluster **spc**
- ▷ LSN Anfang Datenbereich **lsnd**

## Das VFAT-Dateisystem von WINDOWS - Allgemeines

- VFAT-Dateisystem = **Virtual FAT**-Dateisystem

**Kompatible Erweiterung** des FAT-Dateisystems.

Es wurde mit WINDOWS95 eingeführt und wird seitdem in allen folgenden WINDOWS-Versionen eingesetzt.

Da die Funktionen zur Realisierung dieses Dateisystems Bestandteil des 32-Bit-Protected-Mode-Codes von WINDOWS95 und nicht des Real-Mode-Codes von MS-DOS (das in WINDOWS 95 nach wie vor enthalten war) waren, wurde es auch **Protected Mode FAT File System** genannt.

- Das VFAT-Dateisystem bietet im wesentlichen **erweiterte Möglichkeiten** in der Vergabe von **Datei- und Directory-Namen**, die die Beschränkungen des 8.3-Namensformats des FAT-Dateisystems aufheben :

- ◇ Datei- und Directory-Namen dürfen bis zu **255 Zeichen** lang sein.  
Ein Dateipfad darf aber maximal nur 259 Zeichen umfassen.

z.B. COPY bsp96.txt **Pruefung\_Betriebssysteme-Sommersemester\_1996**

- ◇ Sie dürfen auch **Leerzeichen** enthalten

z.B. EDIT **"Zweiter Brief an Rosalie"**

- ◇ Sie dürfen auch die **Zeichen + , ; = [ ]** enthalten

z.B. ECHO 1+2=3 >**"Eins+Zwei=Drei"**

- ◇ **Groß- und Kleinbuchstaben** werden in Namenseinträgen **unterschiedlich gespeichert** (aber bezüglich des Namens nach wie vor als gleich betrachtet)

z.B. MKDIR **"Erste Versuche"**

MKDIR **"ERSTE VERSUCHE"** im gleichen Directory nicht möglich

- ◇ Dateinamen werden nicht in Dateihauptname und Extension zerlegt, d.h. es gibt **keine expliziten Extensions**.  
Ein **Punkt** kann aber jederzeit - auch **mehrfach** - in einem Dateinamen – als normales Zeichen **vorkommen**.

z.B. REN vst341.txt vorl\_unterlage.betriebssysteme.kap3.abschn4.1

⇒ **DIR \*** ist **gleichbedeutend** mit **DIR \*.\***

- Um die **Kompatibilität** zum MS-DOS-Modus von WINDOWS 95/98 und zu früheren MS-DOS- und WINDOWS-Versionen zu wahren, wird für jeden VFAT-spezifischen Datei-/Directory-Namen ein **Zweitname (Alias)** im 8.3-Format vergeben.

Ein **VFAT-spezifischer Name** liegt vor, wenn

- ▷ die Namenslänge das **8.3-Format übersteigt**,
- ▷ in Namen **mehr als ein Punkt** (.) enthalten ist,
- ▷ im Namen **Blanks** oder eines der Zeichen **+ , ; = [ ]** enthalten sind,
- ▷ im Namen **Kleinbuchstaben** enthalten sind

Für Namen, die dem **FAT-Dateisystem entsprechen** (8.3-Format, nur Großbuchstaben) wird **kein Zweitname** vergeben.

## Das VFAT-Dateisystem von WINDOWS - Zweitnamen im 8.3-Format

### • Regeln zur Ableitung des Zweitnamens aus dem VFAT-Original-Namen :

- ◇ Der VFAT-Name entspricht dem **8.3-Format**, enthält aber **Kleinbuchstaben**:
  - Umwandlung aller Kleinbuchstaben in Großbuchstaben  
→ der sich ergebende Name ist der Zweitname
- ◇ Der VFAT-Name entspricht **nicht dem 8.3-Format** und/oder enthält **Blanks**:
  - Umwandlung aller Kleinbuchstaben in Großbuchstaben,
  - Entfernung aller Blanks,
  - Umwandlung der Zeichen + , ; = [ ] in jeweils ein Underscore ('\_')
  - Entfernung aller Punkte außer dem letzten,
  - Kürzung des sich so ergebenden Namens auf maximal 6 Zeichen (vor einem eventuellen Punkt)
  - Anhängen des Zeichens Tilde ('~') und einer laufenden Ziffer 1, 2, ... an das sich so ergebende Namensfragment  
→ Hauptname
  - Falls ein Punkt vorhanden war, Anhängen der maximal ersten drei Zeichen nach dem letzten Punkt als Extension  
→ der sich so insgesamt ergebende Name ist der Zweitname
- ◇ Der Zweitname hängt von den in einem Directory bereits vorhandenen Namenseinträgen ab.  
Er kann für den gleichen VFAT-Namen in unterschiedlichen Directories durchaus unterschiedlich sein (laufende Ziffer !).  
Er kann sich beim Kopieren einer Datei von einem Directory in ein anderes ändern.

### • Beispiele :

MS-DOS~1 PIF	995	13.02.96	20:45	MS-DOS Fenster.pif
EDITOR LNK	275	28.01.96	18:37	Editor.lnk
ERU LNK	302	29.01.96	16:57	ERU.lnk
EXPLORER LNK	269	12.02.96	21:48	Explorer.lnk
MS-DOS~2 PIF	995	27.02.96	11:35	MS-DOS Vollbild.pif
TESTDA~1	12	28.02.96	17:01	Test Datei
TESTDA~1 TXT	18	28.02.96	17:02	Test Datei.txt
TESTDA~2 TXT	26	28.02.96	17:03	Test Damenschuhe.txt
CHKDAT ERG	19	27.02.96	13:15	CHKDAT.ERG
TESTDA~1 BEI	17	27.02.96	12:45	Test Datei3.txt.beisp
BOOTLO~1	20.551	15.02.96	19:54	bootlog von Lw c Windows 95
TEF~1 TST	11	28.02.96	16:33	Te F.tst
TESTDA~1 TST	15	28.02.96	16:20	test.dat.tst
DATEIA~1 BAT	1.183	14.02.96	21:37	Datei Autoexec.bat von Lw C
EINS_Z~1	8	28.02.96	20:34	Eins+Zwei=Drei
BRIEFT~1 3	17	27.02.96	13:10	brief.txt.muster.3

### • Probleme

Bei der Bearbeitung von VFAT-Dateisystemen im **MS-DOS-Modus** bzw unter **früheren MS-DOS-/WINDOWS-Versionen** treten bei den folgenden Operationen Probleme auf :

- Umbenennen von Dateien/Directories
- Löschen von Dateien/Directories
- Umsortieren von Directory-Einträgen (Directory Sort)
- Kopieren/Verschieben von Dateien in andere Directories

## Das VFAT-Dateisystem von WINDOWS - Directory-Einträge (1)

- Das **VFAT-Dateisystem** von WINDOWS 95/98 verwendet die **gleiche Datenträgerorganisation** (gleiche Verwaltungsstrukturen usw) wie das **FAT-Dateisystem**.  
Tatsächlich kann jedes logische Laufwerk mit einem FAT-Dateisystem auch als VFAT-Laufwerk verwendet werden.
- **VFAT-spezifische** Directory-Einträge (z.B. lange Dateinamen) werden durch die Verwendung **mehrerer** – jeweils 32 Bytes langer – **Directory-Eintragsfelder** für **eine Datei** / ein Directory realisiert.

- ◊ **ein Eintragsfeld** enthält den **Zweitnamen** im 8.3-Format sowie alle sonstigen Informationen über die Datei / das Directory wie beim FAT-Dateisystem (Attribute, Zeit/Datum letzte Änderung, erste CLN, Dateilänge).  
Es stellt die **Kompatibilität** zum FAT-Dateisystem sicher.

Darüberhinaus werden in diesem Eintrag einige der im FAT-Dateisystem **reservierten Bytes** zur Aufnahme **weiterer Informationen** verwendet :

- **Zeit und Datum der Datei-Erzeugung** (Creation Time and Date)
- **Datum des letzten Dateizugriffs** (Last Access Date)

- ◊ **Aufbau eines Zweitnamens-Directory-Eintragsfeldes :**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	Dateihauptname (Zweitname)								Extension (Zweitn.)		Attr.	res.	<i>E- Zeit</i>			
0x10	<i>E-Datum</i>		<i>Z-Datum</i>		reserviert		Ä-Zeit		Ä-Datum		CLN		Dateilänge			

Ä-Zeit/Ä-Datum : Zeit und Datum der letzten Datei-Änderung  
*E-Zeit/E-Datum* : Zeit und Datum der Datei-Erzeugung  
*Z-Datum* : Datum des letzten Dateizugriffs

### Format der Datums- und Zeitangaben :

- *E-Datum* u. *Z-Datum* : wie Ä-Datum (Jahr-1980, Monat, Tag)
- *E-Zeit* :  
niederwertigstes Byte : 10-msec-Zeitanteile  
restliche Bytes : wie Ä-Zeit (Stunde, Min., Sek/2)

- ◊ Der **VFAT-spezifische Name** ist in **weiteren** unmittelbar davor befindlichen aufeinanderfolgenden **Eintragsfeldern** abgelegt.  
Die Ablage erfolgt im **Uni-Code** (16-Bit-Code).  
Jeder dieser Eintragsfelder kann maximal **13 Zeichen** aufnehmen.  
Das **letzte** Zeichen des Dateinamens muß mit dem **NUL-Zeichen** (0x0000) abgeschlossen sein.  
Es werden soviel Eintragsfelder verwendet, wie zur Darstellung des Namens benötigt werden (bei der Maximallänge von 255 Bytes : 20 Felder).  
Diese Eintragsfelder sind so aufgebaut, daß sie von MS-DOS und früheren WINDOWS-Versionen **nicht irrtümlich** für "normale" Einträge gehalten werden können :
  - An der Stelle, an der sich bei "normalen" Einträgen das **Dateiattribut-Byte** befindet, enthalten sie den Wert 0x0F - eine **"unmögliche"** Kombination der Attribute Read Only, Hidden, System und Volume Label.
  - An der Stelle, an der sich bei "normalen" Einträgen die **CLN** des ersten Clusters der Datei / des Directories befindet, enthalten sie den Wert 0x0000 - eine **nicht existierende CLN**.
 Sie werden daher von MS-DOS und früheren WINDOWS-Versionen ignoriert.



### Das VFAT-Dateisystem von WINDOWS - Directory-Einträge (3)

- Beispiel für das Root-Directory eines logischen Laufwerks mit VFAT-Dateisystem

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000	42	72	00	2E	00	70	00	69	00	66	00	0F	00	8E	00	00	Br...p.i.f.....
0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	.....
0020	01	4D	00	53	00	2D	00	44	00	4F	00	0F	00	8E	53	00	.M.S.-.D.O....S.
0030	20	00	46	00	65	00	6E	00	73	00	00	00	74	00	65	00	.F.e.n.s...t.e.
0040	4D	53	2D	44	4F	53	7E	31	50	49	46	20	00	36	3C	A1	MS-DOS~1PIF .6<.
0050	16	27	16	27	00	00	09	92	16	27	02	00	C7	03	00	00	.'.'.....
0060	43	43	00	00	00	FF	FF	FF	FF	FF	FF	0F	00	FD	FF	FF	CC.....
0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	.....
0080	02	63	00	2E	00	62	00	61	00	74	00	0F	00	FD	20	00	.c...b.a.t....
0090	76	00	6F	00	6E	00	20	00	4C	00	00	00	77	00	20	00	v.o.n. .L...w. .
00A0	01	44	00	61	00	74	00	65	00	69	00	0F	00	FD	20	00	.D.a.t.e.i....
00B0	41	00	75	00	74	00	6F	00	65	00	00	00	78	00	65	00	A.u.t.o.e...x.e.
00C0	44	41	54	45	49	41	7E	31	42	41	54	20	00	88	57	A1	DATEIA~1BAT ..W.
00D0	16	27	16	27	00	00	15	A5	0B	27	04	00	B6	01	00	00	.'.'.....
00E0	41	45	00	78	00	70	00	6C	00	6F	00	0F	00	D7	72	00	AE.x.p.l.o....r.
00F0	65	00	72	00	2E	00	6C	00	6E	00	00	00	6B	00	00	00	e.r...l.n...k...
0100	45	58	50	4C	4F	52	45	52	4C	4E	4B	20	00	25	67	A1	EXPLORERLNK .%g.
0110	16	27	16	27	00	00	8E	76	0E	27	05	00	15	01	00	00	.'.'...v.'.....
0120	52	44	4C	57	41	20	20	20	44	4D	50	20	00	2E	D0	A1	RDLWA DMP ....
0130	16	27	16	27	00	00	D1	A1	16	27	06	00	00	48	00	00	.'.'.....H..
0140	42	74	00	00	00	FF	FF	FF	FF	FF	FF	0F	00	E0	FF	FF	Bt.....
0150	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	.....
0160	01	54	00	65	00	73	00	74	00	20	00	0F	00	E0	44	00	.T.e.s.t. ....D.
0170	61	00	74	00	65	00	69	00	2E	00	00	00	74	00	78	00	a.t.e.i.....t.x.
0180	54	45	53	54	44	41	7E	31	54	58	54	20	00	4A	1B	A2	TESTDA~1TXT .J..
0190	16	27	16	27	00	00	1C	A2	16	27	2A	00	2C	00	00	00	.'.'.....*
01A0	E5	2E	00	70	00	69	00	66	00	00	00	0F	00	6E	FF	FF	...p.i.f.....n..
01B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	.....
01C0	E5	4D	00	53	00	2D	00	44	00	4F	00	0F	00	6E	53	00	.M.S.-.D.O....nS.
01D0	2D	00	42	00	6F	00	78	00	2D	00	00	00	34	00	33	00	-.B.o.x.-...4.3.
01E0	E5	53	2D	44	4F	53	7E	32	50	49	46	20	00	23	C7	A2	.S-DOS~2PIF .#..
01F0	16	27	16	27	00	00	B4	A2	16	27	2B	00	C7	03	00	00	.'.'.....+.....

```
"WIN98"*A:\>dir
```

Datenträger in Laufwerk A: hat keine Bezeichnung

Seriennummer des Datenträgers: 1CDF-3331

Verzeichnis von A:\

```
MS-DOS~1 PIF          967   22.08.99   18:16 MS-DOS Fenster.pif
DATEIA~1 BAT          438   11.08.99   20:40 Datei Autoexec.bat von Lw C
EXPLORER LNK          277   14.08.99   14:52 Explorer.lnk
RDLWA DMP            18.432  22.08.99   20:14 RDLWA.DMP
TESTDA~1 TXT           44   22.08.99   20:16 Test Datei.txt
      5 Datei(en)                20.158 Bytes
      0 Verzeichnis(se)          1.436.672 Bytes frei
```

```
"WIN98"*A:\>
```

## Das FAT32-Dateisystem (1)

### • Allgemeines

- ✧ Weiterentwicklung des FAT-Dateisystems von MS-DOS (FAT12/16)
- ✧ eingeführt mit **Windows95/OSR2** und **Windows98**
- ✧ **nichtkompatibel** zu FAT12/16
- ✧ für Laufwerke sehr **großer Kapazität** geeignet

### • Allgemeine Änderungen gegenüber FAT12/16

- ✧ Erhöhung der **Länge** eines **FAT-Eintrags** auf **32 Bit** (davon sind aber z. Zt. die 4 höchstwertigsten Bits reserviert)
  - maximal mögliche Clusteranzahl :  $2^{32} = 4\,294\,967\,296$  (4 Giga) (derzeit :  $2^{28} = 268\,435\,456$ )
  - bei größeren Laufwerken sind **kleinere Clustergrößen** möglich,  
z.B. : 4 kB (8 Sekt/Cluster) bis 8 GB Laufwerkskapazität, 16 kB (32 Sekt/Cluster) bei 40 GB Laufwerkskapazität
  - effektivere Ausnutzung der Plattenkapazität (**geringerer Slack**)
- ✧ Die Länge einer log. Sektornummer wurde beibehalten (32 Bit)
  - die **maximal. verwaltbare Laufwerkskapazität** beträgt (bei einer Sektorgröße von 512 Bytes) **2 Terabytes**
- ✧ Für das **Root-Directory** existiert – wie für Subdirectories – eine Verweiskette in der FAT.
  - Es **kann** sich an **beliebiger Position** auf dem Laufwerk befinden und kann **beliebig groß** werden
- ✧ **Vergrößerung des reservierten (Boot-)Bereichs** auf dem Laufwerk (i.a. auf 32 Sektoren, aber nicht alle verwendet) :
  - ✧ zusätzliche Felder im BIOS Parameter Block des Bootsektors
  - ✧ Ausdehnung des Bootcodes auf zwei – nicht direkt aufeinanderfolgende - Sektoren → "Bootrecord"
  - ✧ Ablage zusätzlicher Verwaltungsinformationen (Anzahl freier Cluster, Nummer des zuletzt allokierten Clusters) in einem **File System Information Sektor**
  - ✧ Ablage von Backup-Kopien aller relevanten Sektoren des Bootbereichs
- ✧ Das gleichzeitige Beschreiben aller FAT-Kopien ("FAT mirroring") kann abgeschaltet werden
  - Das Betriebssystem kann auch mit einer **anderen Kopie als der ersten FAT** arbeiten

### • Änderungen im BIOS Parameter Block

- ✧ Der Eintrag "Anzahl Einträge im Root Directory" (Offset  $0 \times 11$ , 2 Bytes) wird ignoriert.
- ✧ Der Eintrag "Anzahl Sektoren pro FAT" (Offset  $0 \times 16$ , 2 Bytes) ist immer gleich 0  
Er wird ersetzt durch einen zusätzlichen 4 Bytes großen Eintrag
- ✧ **Zusätzliche Felder :**
  - Offset  $0 \times 24$  : Anzahl Sektoren pro Groß-FAT (4 Bytes)
  - Offset  $0 \times 28$  : FAT-Verwendungs-Flags (2 Bytes)
    - Bit 7 : 0 → FAT mirroring ist eingeschaltet
    - 1 → FAT mirroring ist ausgeschaltet
  - Bit 0 .. 3 : Nummer der aktiven FAT (nur gültig bei ausgeschaltetem FAT mirroring)
  - Offset  $0 \times 2A$  : Dateisystem-Version (2 Bytes)
    - MS-Byte : Haupt-Versions-Nr
    - LS-Byte : Unter-Versions-Nr
  - Offset  $0 \times 2C$  : Start-Cluster-Nummer des Root-Directories (4 Bytes)
  - Offset  $0 \times 30$  : LSN des Sektors mit zusätzlicher Dateisystem-Verwaltungsinformation (File System Information Sektor) (2 Bytes)
  - Offset  $0 \times 32$  : LSN des Backup-Boot-Sektors (2 Bytes)
    - (=  $0 \times 0FFFF$  wenn kein Backup-Boot-Sektor vorhanden)
  - Offset  $0 \times 34$  : Reservierter Bereich (12 Bytes)

### • Änderungen in Directory-Einträgen :

- ✧ **Die Start-CLN ist auf 4 Bytes erweitert**  
Der im FAT12/16-System verwendete Eintrag enthält die beiden LS-Bytes.  
Die beiden MS-Bytes belegen zwei Bytes des bisher reservierten Bereichs



## Das FAT32-Dateisystem (2)

- Aufbau des Bootsektors**

0x000	0xEB 0x58 0x90 (Sprung zum Start des Bootprogramms + NOP)	3 Bytes
0x003	OEM-Identification (z.B. bei Windows98 : MSWIN4.1)	8 Bytes
0x00B	Anzahl Bytes pro Sektor	2 Bytes
0x00D	Anzahl Sektoren pro Cluster	1 Byte
0x00E	Anzahl reservierter Sektoren	2 Bytes
0x010	Anzahl FATs	1 Byte
0x011	<i>reserviert (0x00 0x00)</i>	2 Bytes
0x013	<i>reserviert (0x00 0x00)</i>	2 Bytes
0x015	Media Descriptor Byte (0xF8 für Hard Disks)	1 Byte
0x016	<i>reserviert (0x00 0x00)</i>	2 Bytes
0x018	Anzahl Sektoren pro Spur	2 Bytes
0x01A	Anzahl der Schreib-/Leseköpfe (Oberflächen)	2 Bytes
0x01C	Anzahl verborgener Sektoren in der Partition vor dem logischen Laufwerk	4 Bytes
0x020	Gesamtzahl der Sektoren im logischen Laufwerk	4 Bytes
0x024	<i>Anzahl Sektoren pro FAT</i>	4 Bytes
0x028	<i>FAT-Verwendungs-Flags</i>	2 Bytes
0x02A	<i>FAT32-Dateisystem-Version (MS-Byte : Haupt-Vers., LS-Byte : Unter-Vers.)</i>	2 Bytes
0x02C	<i>Cluster-Nr des Beginns des Root-Directories</i>	4 Bytes
0x030	<i>Logische Sektor-Nr. des File System Information Sektors</i>	2 Bytes
0x032	<i>Logische Sektor-Nr. des Backup Boot Sektors (Beginn Backup-Bereich)</i>	2 Bytes
0x034	<i>reserviert</i>	12 Bytes
0x040	Physikalische Laufwerks-Nr (0x80 oder 0x00)	1 Byte
0x041	reserviert (0x00)	1 Byte
0x042	Kennung für erweiterten Bootsektor (0x29)	1 Byte
0x043	Seriennummer des logischen Laufwerks	4 Bytes
0x047	Datenträgername (Volume Label)	11 Bytes
0x052	Dateisystem-Name ("FAT32 ")	8 Bytes
0x05A	Boot-Code	418 Bytes
0x1FE	Kennung für Bootsektoren (0x55 0xAA)	2 Bytes

### Das FAT32-Dateisystem (3)

- Beispiel für einen Bootsektor

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000 :	eb	58	90	4d	53	57	49	4e	34	2e	31	00	02	20	20	00	.X.MSWIN4.1..
00000010 :	02	00	00	00	00	f8	00	00	3f	00	ff	00	3f	00	00	00	.....?....?
00000020 :	21	59	42	04	2c	44	00	00	00	00	00	00	f4	fd	03	00	?YB..D.....
00000030 :	01	00	06	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000040 :	80	00	29	fe	96	39	85	44	41	54	41	32	00	00	00	00	..>..9.DATA2....
00000050 :	00	00	46	41	54	33	32	20	20	20	fa	33	c9	8e	d1	bc	..FAT32..3....
00000060 :	f8	7b	8e	c1	bd	78	00	c5	76	00	1e	56	16	55	bf	22	..<...x...v...U.U."
00000070 :	05	89	7e	00	89	4e	02	b1	0b	fc	f3	a4	8e	d9	bd	00	..~..N.....
00000080 :	7c	c6	45	fe	0f	8b	46	18	88	45	f9	38	4e	40	7d	25	!..E...F...E.8NE>%
00000090 :	8b	c1	99	bb	00	07	e8	97	00	72	1a	83	eb	3a	66	a1	...s....r....:f.
000000a0 :	1c	7c	66	3b	07	8a	57	fc	75	06	80	ca	02	88	56	02	..if;..W.u....U.
000000b0 :	80	c3	10	73	ed	bf	02	00	83	7e	16	00	75	45	8b	46	...s....~..uE.F
000000c0 :	1c	8b	56	1e	b9	03	00	49	40	75	01	42	bb	00	7e	e8	..U....I@u.B...~
000000d0 :	5f	01	73	26	b0	f8	4f	74	1d	8b	46	32	33	d2	b9	03	..s&..Ot..F23...
000000e0 :	00	3b	c8	77	1e	8b	76	0e	3b	ce	73	17	2b	f1	03	46	..;w...v...;s...+..F
000000f0 :	1c	13	56	1e	eb	d1	73	0b	eb	27	83	7e	2a	00	77	03	..U....s...'.~*.w.
00000100 :	e9	fd	02	be	7e	7d	ac	98	03	f0	ac	84	c0	74	17	3c	....^>.....t.<
00000110 :	ff	74	09	b4	0e	bb	07	00	cd	10	eb	ee	be	81	7d	eb	..t.....>..
00000120 :	e5	be	7f	7d	eb	e0	98	cd	16	5e	1f	66	8f	04	cd	19	....>.....^f....
00000130 :	41	56	66	6a	00	52	50	06	53	6a	01	6a	10	8b	f4	60	AUfj.RP.Sj.j...`
00000140 :	80	7e	02	0e	75	04	b4	42	eb	1d	91	92	33	d2	f7	76	..~..u..B...3..v
00000150 :	18	91	f7	76	18	42	87	ca	f7	76	1a	8a	f2	8a	e8	c0	...v..B...v.....
00000160 :	cc	02	0a	cc	b8	01	02	8a	56	40	cd	13	61	8d	64	10	.....UE...a.d.
00000170 :	5e	72	0a	40	75	01	42	03	5e	0b	49	75	b4	c3	03	18	^r.@u.B.^..lu....
00000180 :	01	27	0d	0a	49	6e	76	61	6c	69	64	20	73	79	73	74	..'..Invalid syst
00000190 :	65	6d	20	64	69	73	6b	ff	0d	0a	44	69	73	6b	20	49	em disk...Disk I
000001a0 :	2f	4f	20	65	72	72	6f	72	ff	0d	0a	52	65	70	6c	61	/O error...Repla
000001b0 :	63	65	20	74	68	65	20	64	69	73	6b	2c	20	61	6e	64	ce the disk, and
000001c0 :	20	74	68	65	6e	20	70	72	65	73	73	20	61	6e	79	20	then press any
000001d0 :	6b	65	79	0d	0a	00	00	49	4f	20	20	20	20	20	20	20	key.....IO
000001e0 :	53	59	53	4d	53	44	4f	53	20	20	20	53	59	53	7e	01	SYSMSDOS SYS~.
000001f0 :	00	57	49	4e	42	4f	4f	54	20	53	59	53	00	00	55	aa	.WINBOOT SYS..U.
Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF

- Inhalt der FAT32-spezifischen Felder des BPBs aus obigem Bootsektor :

0x024 :	Anzahl Sektoren pro FAT	:	17 452	(0x0000442c)
0x028 :	FAT-Verwendungsflags	:	0000h	
0x02A :	Dateisystem-Version	:	0.0	(0x00 0x00)
0x02C :	CLN Beginn Root-Directory	:	261 620	(0x0003fdf4)
0x030 :	LSN des File System Information Sektors	:	1	(0001h)
0x032 :	LSN des Backup Boot Sektors	:	6	(0006h)

## Das FAT32-Dateisystem (4)

### • Aufbau des File System Information Sektors

0x000	1. Signatur (0x52 0x52 0x61 0x41)	4 Bytes
0x004	unbekannt (z. Zt. offensichtlich alles 0x00)	480 Bytes
0x1E4	2. Signatur (0x72 0x72 0x41 0x61)	4 Bytes
0x1E8	Anzahl freier Cluster (== -1, wenn unbekannt)	4 Bytes
0x1EC	CLN des zuletzt allokierten Clusters	4 Bytes
0x1F0	reserviert	12 Bytes
0x1FC	unbekannt (z. Zt. offensichtlich 0x00 0x00)	2 Bytes
0x1FE	Kennnung für Bootsektoren (0x55 0xAA)	2 Bytes

### • Directory-Einträge

- ◇ Das FAT32-Dateisystem arbeitet grundsätzlich mit **langen Dateinamen** (→ VFAT-Dateisystem), d.h. ein Directory-Eintrag belegt i.a. mehrere Eintragsfelder :
  - das Zweitnamens-Eintragfeld, das auch die sonstigen Directory-Informationen enthält
  - bis zu 20 weitere Eintragsfelder zur Aufnahme des (langen) Original-Dateinamens

- ◇ **Aufbau eines Zweitnamens-Directory-Eintragfeldes :**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	Dateihauptname (Zweitname)								Extension (Zweitn.)			Attr.	res.	E- Zeit		
0x10	E-Datum		Z-Datum		<i>CLN High</i>		Ä-Zeit		Ä-Datum		CLN Low		Dateilänge			

Ä-Zeit/Ä-Datum : Zeit und Datum der letzten Datei-Änderung

E-Zeit/E-Datum : Zeit und Datum der Datei-Erzeugung

Z-Datum : Datum des letzten Dateizugriffs

Attr. : Dateiattribut

**CLN High** : High- (MS-) Word der Clusternummer des Datei- (bzw Directory-) Beginns

CLN Low : Low- (LS-) Word der Beginn-Clusternummer

### • FAT-Einträge :

- ◇ 4 Bytes lang  
Beginn mit CLN 0x00000002
- ◇ CLN 0x00000000 : Cluster ist frei  
CLN 0x0FFFFFFF : letzter Cluster einer Cluster-Kette

## Das NTFS von WINDOWS NT (1)

- **NTFS – New Technology File System**

Dateisystem von Windows NT, Windows 2000, Windows XP und Windows Vista

- **Allgemeines**

- ◇ **Hierarchisches Dateisystem**

**Directories** sind **spezielle Dateien**

Das **Root-Directory** kann **beliebig groß** werden

**Directory-Einträge** sind **sortiert** ("*File Name Indexing*") (b+ - Baum)

- ◇ Implementierung der **Schutzmechanismen** von Windows NT auch für den Dateizugriff

→ Dateien sind vor unberechtigtem Zugriff (z.B. vor anderen Benutzern) geschützt

- ◇ **Datei- und Directory-Namen** können bis zu **255 Zeichen** lang sein

Sie können **Leerzeichen** und **mehrere Punkte** enthalten

Groß- und Kleinbuchstaben werden unterschiedlich abgespeichert, aber als gleich behandelt

Abspeicherung der Namen im **Unicode**

Für Dateizugriffe von MS-DOS-Clients (und 16-Bit-Windows-Clients) wird automatisch ein **MS-DOS-Zweitname** (Alias-Name) im **8.3-Format** generiert

- ◇ Ein logisches NTFS-Laufwerk wird als **Volume** bezeichnet

Allokationseinheit : **Cluster** → logische Cluster-Nummer (**LCN**), Beginn mit **LCN 0**

**64 Bit** große Cluster-Nummern

Clustergröße : 512 Bytes (1 Sektor) ... 4 kBytes (8 Sektoren)

**max. Volume (und Datei-) Größe** :  $2^{64} * 4 \text{ Kbytes} \approx 64 * 10^{21} \text{ Bytes}$

- ◇ **Verwaltung** des **freien Speichers** mittels **Bitmap** (pro Cluster ein Bit)

Zusammenfassung defekter Sektoren in einer Datei ("*bad cluster file*")

- ◇ Übersicht **Zuordnung** Datei – Speicherort (Cluster) : **Verweistabelle**

- ◇ Im Dateisystem integrierte Implementierung einer **Datenkomprimierung** auf Dateiebene

- ◇ **Buchführung** über die erfolgreich abgeschlossenen Teiloperationen aller Datei-**Transaktionen** in einer **Log-Datei**.

→ **Fehlerhafte** (unvollständige) **Transaktionen** können **rückgängig** gemacht und das Dateisystem damit in einem konsistenten Zustand gehalten werden (→ *Journaling File System*)

- **NTFS Metadata Files**

- ◇ Ein NTFS-Laufwerk enthält **nur Dateien und freien Speicher**

Auch alle auf dem Laufwerk befindlichen **Verwaltungsinformationen** – einschließlich des Boot-Sektors (1. Sektor eines NTFS-Laufwerks) – befinden sich ebenfalls in Dateien (→ **NTFS Metadata Files**)

- ◇ Die **wichtigsten NTFS Metadata Files** sind :

- ▷ **\$MFT** : Master File Table (MFT)

zentrale Verwaltungsstruktur, enthält für jede Datei (und jedes Directory) einen Eintrag

- ▷ **\$LogFile** : Log-Datei

Buchführung über die Teiloperationen aller Datei-Transaktionen

- ▷ **\$Volume** : Volume File

Informationen über das logische Laufwerk (u.a. Datenträgername, Dirty-Flag, Versions-Nr. des erzeugenden BS)

- ▷ **\$AttrDef** : Attribute Definition File

Definition der im logischen Laufwerk verwendbaren Attribute

- ▷ **Root Directory**

- ▷ **\$Bitmap** : Bitmap zur Buchführung des Belegungszustandes der einzelnen Cluster

Pro Cluster ein Bit : Bit ist gesetzt, wenn Cluster belegt ist

- ▷ **\$Boot** : Boot File

Boot-Sektor + Boot-Programm

- ▷ **\$BadClus** : Bad Cluster File

Zusammenfassung der defekten Cluster des logischen Laufwerks

## Das NTFS von WINDOWS NT (2)

### • Dateiattribute

- ◇ Alle mit einer Datei verknüpften Informationen, wie z.B. ihr Name, Zeitmarken, Zugriffsberechtigungen, aber auch der eigentliche Dateiinhalt sind als **Attribute** der Datei implementiert.  
→ Eine **Datei** ist im NTFS als **Menge von Attributen** definiert  
Ein Attribut besitzt einen **Typ-Code** (*type code*) und einen **Namen** (optional)  
Jeder Typ-Code wird im NT-Quellcode durch eine symbolische Konstante, die mit einem **\$**-Zeichen beginnt, referiert.  
Jedes dieser Attribute besteht aus einer Folge von Bytes ("*stream*") (→ "Wert" des Attributs).  
Jedes Attribut hat einen **Attribut-Header**.  
Jegliche Dateibearbeitung im NTFS besteht aus dem Lesen und Schreiben von Attributen.

- ◇ Die wichtigsten **Standard-Attribute** sind :

- ▷ **Standard-Informations-Attribut** (Typ-Code 0x10 = \$STANDARD\_INFORMATION) :  
Datei-Flags (erweitertes MS-DOS-Datei-Attribut (read only usw)), Zeitmarken
- ▷ **Namens-Attribut** (Typ-Code 0x30 = \$FILE\_NAME) :  
Dateiname (im Unicode), zusätzlich Datei-Flags u. Zeitmarken (wie Standard-Informations-Attribut), Dateigröße, Referenz auf Directory, in dem die Datei eingetragen ist.  
**Hard Links** werden durch **mehrere Namens-Attribute** realisiert.
- ▷ **Zweitnamens-Attribut** ("MS-DOS File Name") : implementiert als weiteres Namens-Attribut (Typ-Code 0x30)
- ▷ **Sicherheits-Attribut** (Typ-Code 0x50 = \$SECURITY\_DESCRIPTOR) :  
Dateibesitzer und Zugriffsberechtigungen
- ▷ **Daten-Attribut** (Typ-Code 0x80 = \$DATA) : eigentlicher Dateiinhalt (nicht für Directories)
- ▷ Attribute zur Implementierung – sortierter – Directories :  
**Index Root Attribute** (Typ-Code 0x90 = \$INDEX\_ROOT)  
**Index Allocation Attribute** (Typ-Code 0xA0 = \$INDEX\_ALLOCATION)  
**Bitmap Attribute** (Typ-Code 0xB0 = \$BITMAP)

- ◇ **normale Dateien** bestehen üblicherweise aus (das Zweitnamens-Attribut ist weggelassen) :

\$STANDARD_INFORMATION	\$FILE_NAME	\$SECURITY_DESCRIPTOR	\$DATA
------------------------	-------------	-----------------------	--------

- ◇ **Directories** bestehen üblicherweise aus (das Zweitnamens-Attribut ist weggelassen) :

\$STANDARD_INF.	\$FILE_NAME	\$SECURITY_DESCR.	\$INDEX_ROOT	\$INDEX_ALLOCATION	\$BITMAP
-----------------	-------------	-------------------	--------------	--------------------	----------

- ◇ Darüberhinaus sind **weitere Standard-Attribute** definiert, die z.Teil nur in bestimmten NTFS Metadata Files oder nur in bestimmten Situationen zur Anwendung kommen, z. B. :

- ▷ **Attribute List Attribute** (Type-Code 0x20 = \$ATTRIBUTE\_LIST)  
Liste der Attribute, aus denen eine Datei besteht.  
Wird nur benötigt, wenn für eine Datei/Sub-Directory mehr als ein MFT-Record erforderlich ist.
- ▷ **Volume Name Attribute** (Type-Code 0x60 = \$VOLUME\_NAME)  
Datenträgername.  
Im Volume File (\$Volume) verwendet
- ▷ **Volume Information Attribute** (Type-Code 0x70 = \$VOLUME\_INFORMATION)  
Informationen über den Zustand eines logischen Laufwerks (Flags, Versions-Nr. des erzeugenden BS)  
Im Volume File (\$Volume) verwendet

- ◇ Zusätzlich sind **benutzerdefinierte Attribute** möglich.  
Sie müssen im Attribute Definition File definiert werden.

### Das NTFS von WINDOWS NT (3)

#### • Der NTFS-Bootsektor

- erster Sektor eines NTFS-Laufwerks (erster Sektor des Clusters mit LCN 0)
- erster Sektor des Datenattributs des Boot Files (File **\$Boot**)
- Aufbau analog zum Bootsektor der FAT-Dateisysteme :  
→ er enthält das Laufwerk beschreibende Informationen sowie das Bootstrap-Programm (*boot loader routine*)
- Die weiteren Sektoren des Boot Files enthalten den Windows NT Loader **NTLDR**

#### Aufbau :

0x000	EB 52 90 (Sprung zum Start des Bootprogramms + NOP)	3 Bytes
0x003	<b>System-ID : "NTFS "</b>	8 Bytes
0x00B	Anzahl Bytes pro Sektor	2 Bytes
0x00D	Anzahl Sektoren pro Cluster	1 Byte
0x00E	<b>reserviert (00 00 00 00 00 00 00)</b>	7 Bytes
0x015	Media Descriptor Byte (0xF8 für Hard Disks)	1 Byte
0x016	<b>reserviert (00 00)</b>	2 Bytes
0x018	Anzahl Sektoren pro Spur	2 Bytes
0x01A	Anzahl der Schreib-/Leseköpfe (Oberflächen)	2 Bytes
0x01C	<b>reserviert (00 00 00 00 00 00 00 00)</b>	8 Bytes
0x024	<b>80 00 80 00 (immer ?)</b>	4 Bytes
0x028	Gesamtzahl Sektoren im logischen Laufwerk	8 Bytes
0x030	<b>Logische Cluster-Nr. des ersten Clusters der MFT (Datenattribut von \$MFT)</b>	8 Bytes
0x038	<b>Logische Cluster-Nr. des ersten Clusters der Backup-MFT(\$MFTMirr)</b>	8 Bytes
0x040	<b>Anzahl Cluster pro MFT-Record</b> (0xF6 bedeutet 1/4)	4 Bytes
0x044	<b>Anzahl Cluster pro Index-Record</b>	4 Bytes
0x048	Seriennummer des logischen Laufwerks	8 Bytes
0x050	<b>reserviert (00 00 00 00)</b>	4 Bytes
0x054	Boot-Code ( <i>boot loader routine</i> )	426 Bytes
0x1FE	Kennnung für Bootsektoren (55 AA)	2 Bytes

## Das NTFS von WINDOWS NT (4)

- Beispiel für einen NTFS-Bootsektor

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000 :	eb	52	90	4e	54	46	53	20	20	20	20	00	02	08	00	00	.R.NTFS .....
00000010 :	00	00	00	00	00	f8	00	00	3f	00	ff	00	6b	b2	d8	09	.....?....k...
00000020 :	00	00	00	00	80	00	80	00	98	41	21	04	00	00	00	00	.....A!.....
00000030 :	04	00	00	00	00	00	00	00	00	00	08	00	00	00	00	00	.....<....8U.
00000040 :	f6	00	00	00	01	00	00	00	09	7b	17	16	f5	38	55	11	.....3.....l....
00000050 :	00	00	00	00	fa	33	c0	8e	d0	bc	00	7c	fb	b8	c0	07	.....3.....3....
00000060 :	8e	d8	e8	16	00	b8	00	0d	8e	c0	33	db	c6	06	0e	00	...S.h..hj....\$.
00000070 :	10	e8	53	00	68	00	0d	68	6a	02	cb	8a	16	24	00	b4	...s.....f...@f
00000080 :	08	cd	13	73	05	b9	ff	ff	8a	f1	66	0f	b6	c6	40	66	.....?.....Af.
00000090 :	0f	b6	d1	80	e2	3f	f7	e2	86	cd	c0	ed	06	41	66	0f	..f..f..A..U.
000000a0 :	b7	c9	66	f7	e1	66	a3	20	00	c3	b4	41	bb	aa	55	8a	..\$.r..U.u....
000000b0 :	16	24	00	cd	13	72	0f	81	fb	55	aa	75	09	f6	c1	01	t.....f'..f...f
000000c0 :	74	04	fe	06	14	00	c3	66	60	1e	06	66	a1	10	00	66	...f;...:..fj
000000d0 :	03	06	1c	00	66	3b	06	20	00	0f	82	3a	00	1e	66	6a	..fP.Sfh.....>..
000000e0 :	00	66	50	06	53	66	68	10	00	01	00	80	3e	14	00	00	.....>.....a.
000000f0 :	0f	85	0c	00	e8	b3	ff	80	3e	14	00	00	0f	84	61	00	..B..\$......fX[.
00000100 :	b4	42	8a	16	24	00	16	1f	8b	f4	cd	13	66	58	5b	07	fXfX...-f3.f....
00000110 :	66	58	66	58	1f	eb	2d	66	33	d2	66	0f	b7	0e	18	00	f.....f..f....6
00000120 :	66	f7	f1	fe	c2	8a	ca	66	8b	d0	66	c1	ea	10	f7	36	.....\$......
00000130 :	1a	00	86	d6	8a	16	24	00	8a	e8	c0	e4	06	0a	cc	b8	.....f.....
00000140 :	01	02	cd	13	0f	82	19	00	8c	c0	05	20	00	8e	c0	66	.....o...fa
00000150 :	ff	06	10	00	ff	0e	0e	00	0f	85	6f	ff	07	1f	66	61	.....<.t.....
00000160 :	c3	a0	f8	01	e8	09	00	a0	fb	01	e8	03	00	fb	eb	fe	.....Fehler beim
00000170 :	b4	01	8b	f0	ac	3c	00	74	09	b4	0e	bb	07	00	cd	10	.....Lesen des Daten
00000180 :	eb	f2	c3	0d	0a	46	65	68	6c	65	72	20	62	65	69	6d	tr.gers...NTLDR
00000190 :	20	4c	65	73	65	6e	20	64	65	73	20	44	61	74	65	6e	fehlt...NTLDR is
000001a0 :	74	72	84	67	65	72	73	00	0d	0a	4e	54	4c	44	52	20	t komprimiert...
000001b0 :	66	65	68	6c	74	00	0d	0a	4e	54	4c	44	52	20	69	73	Neustart mit Str
000001c0 :	74	20	6b	6f	6d	70	72	69	6d	69	65	72	74	00	0d	0a	g+Alt+Entf.....
000001d0 :	4e	65	75	73	74	61	72	74	20	6d	69	74	20	53	74	72	.....U.
000001e0 :	67	2b	41	6c	74	2b	45	6e	74	66	0d	0a	00	00	00	00	
000001f0 :	00	00	00	00	00	00	00	00	83	a8	b6	ce	00	00	55	aa	
Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF

- Inhalt einiger Eintragsfelder :

0x00D :	Anzahl Sektoren pro Cluster	:	0x08	
0x028 :	Gesamtzahl Sektoren im logischen Laufwerk	:	0x00000000004214198	= 69 288 344
0x030 :	LCN Beginn MFT	:	0x00000000000000004	= 4
0x038 :	LCN Beginn Backup-MFT	:	0x00000000000080000	= 524 288
0x040 :	Anzahl Cluster pro MFT-Record	:	0x000000F6	→ 1/4 Cluster → 2 Sektoren
0x044 :	Anzahl Cluster pro Index-Record	:	0x00000001	

## Das NTFS von WINDOWS NT (5)

- **Master File Table (MFT)**

- ◇ **Zentrale Verwaltungsstruktur.**

- ◇ Enthält **für jede** auf dem Laufwerk befindliche **Datei** (einschließlich für sich selbst) **einen Eintrag**.  
Sie ist in **Records** (*file record*) strukturiert.

Recordgröße (abhängig von Clustergröße, beim Formatieren festgelegt) : 1kB, 2kB oder 4kB

Ein Eintrag für eine Datei belegt normalerweise einen Record.

Für stark fragmentierte oder mit sehr viel Attributen versehene Dateien können auch mehrere Records belegt werden.

Der (erste) Record eines Dateieintrags (*Base file record*) wird auch als **Inode** bezeichnet.

Bei einem Dateieintrag, der aus mehr als einem Record besteht, werden die weiteren Records *Extension file records* genannt.

- ◇ **Prinzipeller Aufbau der MFT** (genauer : des Werts des **Datenattributs** der MFT-Datei) :

Record-Nr	
0	<b>MFT</b> (\$MFT)
1	Teilkopie der MFT (\$MFTMirr)
2	<b>Log File</b> (\$LogFile)
3	<b>Volume File</b> (\$Volume)
4	<b>Attribute Definition File</b> (\$AttrDef)
5	<b>Root Directory</b> (.)
6	<b>Bitmap File</b> (\$Bitmap)
7	<b>Boot File</b> (\$Boot)
8	<b>BadCluster File</b> (\$BadClus)
9	
...	weitere NTFS Metadata Files
16	
...	reserviert für <i>extension file records</i> der MFT
24	
	Anwender-Dateien und Directories
	...
	...
	...



## Das NTFS von WINDOWS NT (6)

### • MFT-Einträge

- ◇ Ein **Dateieintrag** in der MFT besteht aus den **Attributen** der Datei.  
Die einzelnen – prinzipiell unterschiedlich langen – Attribute sind in **aufsteigender Reihenfolge** ihres **Typ-Codes** (*type code*) im zugehörigen MFT-Record (bzw in den zugehörigen MFT-Records) angeordnet.
- ◇ Attribute, die **vollständig direkt in der MFT** enthalten sind, heißen **residente Attribute** (*resident attribute*).  
**Kurze Dateien** (Dateien mit "kurzem" Dateiinhalt, d.h. "kleinem" Daten-Attribut) können somit **vollständig in der MFT enthalten** sein, sie belegen **keinen zusätzlichen Platz**
- ◇ Für Attribute, deren Wert für eine direkte Ablage im MFT-Record zu groß (lang) ist (z.B. das Datenattribut bei "längeren" Dateien), wird Speicher in Vielfachen von 2 kB (bzw 4kB bei Clustergröße von 4kB) außerhalb der MFT allokiert (→ "*run*", "*extent*").  
Derartige Attribute heißen **nicht-residente Attribute** (*nonresident attribute*).

Die zu einem nicht-residenten Attribut (z.B. Daten-Attribut) gehörenden Cluster werden bei 0 beginnend fortlaufend durchnummeriert → **virtuelle Cluster-Nummer (VCN)**

Jedes Attribut besteht aus einem **Attribut-Header** und dem eigentlichen **Attribut-Wert** ("*stream*" des Attributs)

Im immer im MFT-Record abgelegten **Attribut-Header** ist vermerkt, ob das betreffende Attribut resident oder nicht-resident ist. Bei nicht-residenten Attributen sind im MFT-Record statt des Attribut-Werts Informationen über dessen Speicherort (**Zuordnung VCN – LCN, Verweistabelle, *Run List***) enthalten

#### ◇ Prinzipeller Aufbau eines MFT-Records :

Jeder MFT-Record enthält vor der Folge von Attributen einen **Record Header** und ist mit einer Endekennung (Bytefolge **0xFFFFFFFF**) abgeschlossen :

Record Header	Attribut	...	Attribut	0xFFFFFFFF
---------------	----------	-----	----------	------------

Der **Record Header** enthält u.a.

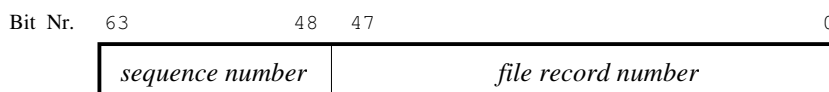
- Nutzungszähler (*sequence number*)  
wird – bei 1 beginnend – vor jeder erneuten Nutzung eines Records (nach Löschen einer Datei) um 1 erhöht.
- Nutzungs-Flag : Kennzeichnung, ob der Record benutzt wird (d.h. sein Inhalt gültig ist)  
Beim Löschen einer Datei werden die Nutzungs-Flags aller von der Datei belegten MFT-Records rückgesetzt
- Directory-Flag : Record beschreibt ein Directory
- Hard-Link-Zähler (*hard link count*)
- Allozierte Größe des Records (*allocated size*), beim Formatieren festgelegt
- tatsächlich belegte Größe des Records (*real size*), auf ein Vielfaches von 8 ergänzt.
- Referenz auf zugehörigen *Base file record*, falls Record ein *Extension file record* ist

### • File Reference

- ◇ **Alle Dateien** in einem NTFS-Laufwerk werden über eine **8 Bytes** große Nummer **referiert** → **File Reference**

Diese setzt sich zusammen aus

- der Nr. des (ersten) MFT-Records (***file record number***) , der den Dateieintrag enthält (niederwertige 6 Bytes)
- und dem Wert des Nutzungszählers (***sequence number***) dieses Records (höherwertige 2 Bytes)



- ◇ Die Ergänzung der MFT-Record-Nummer um die zugehörige *sequence number* ermöglicht ein einfaches Erkennen von Inkonsistenzen (Zugriff zu nicht mehr existierender Datei), wenn der Record nach Löschen einer Datei erneut verwendet wurde.

## Das NTFS von WINDOWS NT (7)

### • Standard-Attribut-Header

#### ◇ gemeinsamer Teil für residente und nicht residente Attribute

0x00	4	Typ-Code
0x04	4	Attribut-Länge (einschl. Header)
0x08	1	Non-resident Flag (0x00 = resident, 0x01 = non-resident)
0x09	1	Länge des Attribut-Namens (N, 0x00 wenn namenlos)
0x0A	2	Offset des Attribut-Namens (bzw Länge des Attribut-Headers, wenn namenlos)
0x0C	2	Modifikations-Flags ( <i>compressed, encrypted, sparse</i> )
0x0E	2	Attribut-Id (?)

#### ◇ Fortsetzung für residente Attribute

0x10	4	Länge des Attribut-Wertes (L)
0x14	2	Länge des Attribut-Headers
0x16	1	Indizierungs-Flag
0x17	1	Füll-Zeichen (0x00)
0x18	2N	Attribut-Name im Unicode (nur wenn nicht namenlos)
2N+0x18	L	Attribut-Wert (" <i>stream</i> ")

#### ◇ Fortsetzung für nicht-residente Attribute

0x10	8	erste VCN des von der Verweistabelle abgedeckten Abschnitts des Attribut-Wertes
0x18	8	letzte VCN des von der Verweistabelle abgedeckten Abschnitts des Attribut-Wertes
0x20	2	Länge des Attribut-Headers ohne Verweistabelle ( <i>Run List</i> )
0x22	2	<i>Compression Unit Size</i> (0x00 wenn nicht komprimiert)
0x24	4	Füll-Zeichen (0x00 0x00 0x00 0x00)
0x28	8	allozierte Größe des Attribut-Wertes (Vielfaches einer Cluster-Größe)
0x30	8	reale nicht komprimierte Größe des Attribut-Wertes (bei Daten-Attribut = Dateigröße)
0x38	8	initialisierte (=komprimierte) Größe des Attribut-Wertes
0x40	2N	Attribut-Name im Unicode (nur wenn nicht namenlos)
2N+0x40		Verweistabelle ( <i>Run List</i> )

## Das NTFS von WINDOWS NT (8)

### • Verweistabelle (*Run List*)

- ◇ Der Attribut-Wert ("*stream*") von nicht-residenten Attributen (z.B. der eigentliche Dateiinhalte als Wert des Datenattributs) ist in Bereichen von einem oder mehreren aufeinanderfolgenden Clustern ("*runs*") außerhalb der MFT abgelegt.  
Existieren mehrere solcher Bereiche, ist der Attribut-Wert **fragmentiert** (→ fragmentierte Datei)  
Jeder dieser Bereiche ("*runs*") läßt sich durch die LCN seines ersten Clusters und seine Länge (=Anzahl der Cluster) definieren.

- ◇ Im MFT-Record ist bei nicht-residenten Attributen nach dem Attribut-Header statt des Attribut-Wertes eine **Verweistabelle**, die die "*runs*" beschreibt, enthalten (***Run List***).

Die Tabelle ist mit dem **0x00-Byte** abgeschlossen

Sie ermöglicht die Zuordnung der einzelnen *Stream*-Bestandteile zu ihrem Speicherort

⇒ **Zuordnung VCN → LCN**

- ◇ **Aufbau eines Tabelleneintrags :**

Die LCN des ersten Clusters eines "*runs*" wird relativ zur LCN des ersten Clusters des vorhergehenden "*runs*" angegeben.

Die LCN des ersten Clusters des ersten "*runs*" ist dessen absolute LCN.

Um Platz zu sparen sind die – relative – LCN und die Länge eines "*runs*" in Feldern variabler Größe abgelegt.

Dies erfordert die zusätzliche Angabe der Größe dieser Felder

0x00	1	Header : niederwertiges Halbbyte : Größe des Längenfeldes (L) höherwertiges Halbbyte : Größe des Feldes mit der relativen LCN (Offset-Feld) (F)
0x01	L	Länge des " <i>runs</i> " in Anzahl Cluster
L+0x01	F	Offset des ersten Clusters des " <i>runs</i> " relativ zur LCN des ersten Clusters des vorhergehenden " <i>runs</i> "

- ◇ **1. Beispiel : nicht fragmentierte Datei**

*Run List* : 21 18 34 56 00

Gruppierung in die einzelnen Tabelleneinträge : 21 18 34 56  
00

⇒ **ein "*run*" :**

Header	0x21	
Länge	(1 Byte)	0x18
Offset	(2 Bytes)	0x5634
VCN	0	0x17
LCN	0x5634	0x564B

- ◇ **2. Beispiel fragmentierte Datei**

*Run List* : 31 38 73 25 34 32 14 01 E5 11 02 31 42 AA 00 03 00

Gruppierung in die einzelnen Tabelleneinträge : 31 38 73 25 34  
32 14 01 E5 11 02  
31 42 AA 00 03  
00

⇒ **3 "*runs*" :**

Header	0x31		0x32		0x31	
Länge	(1 Byte)	0x38	(2 Bytes)	0x0114	(1 Byte)	0x42
Offset	(3 Bytes)	0x342573	(3 Bytes)	0x0211E5	(3 Bytes)	0x0300AA
VCN	0	0x37	0x38	0x14B	0x14C	0x18D
LCN	0x342573	0x3425AA	0x363758	0x36386B	0x393802	0x393843

## Directories beim NTFS (1)

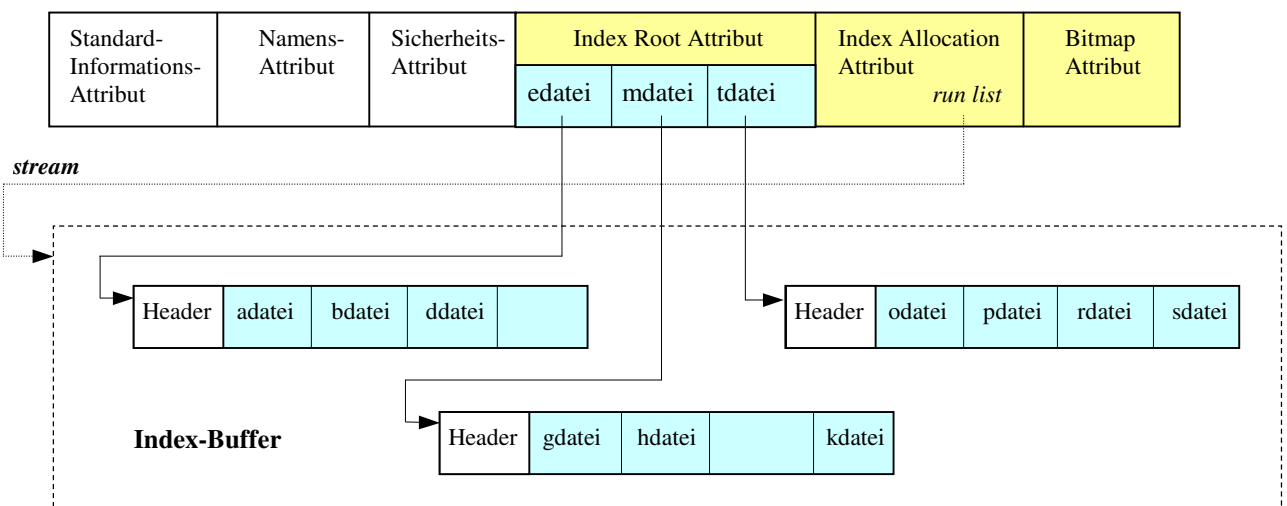
### • Allgemeines

- ◇ Ein NTFS-Directory ist ein **Index** von Datei-(u/o Subdirectory-)Namen.  
Es besteht aus einer Folge von Index-Einträgen, die jeweils ein **Namens-Attribut** und die zugehörige **File Reference** (→ Eintrag in der MFT) enthalten.
- ◇ Die Einträge sind in einem **B+-Baum** lexikographisch aufsteigend **sortiert** abgelegt.  
→ schnelles und effizientes Suchen nach Einträgen.
- ◇ Ein Directory besitzt statt eines Daten-Attributs ein **Index Root Attribut**, ein **Index Allocation Attribut** und ein **Bitmap Attribut**, mit denen der B+-Baum implementiert wird.

### • Prinzipielle Implementierung.

- ◇ Das **Index Root Attribut** (immer resident) bildet den **Wurzel-Knoten** des **B+-Baums**. In ihm finden eine bestimmte Anzahl von sortierten Index-Einträgen Platz.
- ◇ Enthält das Directory mehr Einträge, als im Index Root Attribut Platz haben, werden diese in nicht-residente **Index-Buffer** ausgelagert. Die Größe eines Index-Buffers entspricht der Minimal-Größe eines für ein nicht-residentes Attribut allozierten "runs" (2 kB bzw 4 kB bei 4kB Clustergröße). Jeder Index-Buffer bildet einen **Knoten** im B+-Baum. Jeder Index-Buffer wird von einem Index-Eintrag in einem übergeordneten Knoten (Index Root Attribut bzw übergeordneter Index-Buffer) **referiert**.  
Er enthält die Einträge, die kleiner als der den Buffer referierende Eintrag aber größer als der davor stehende Eintrag sind.
- ◇ Die Gesamtheit der Index-Buffer bildet den Attribut-Wert ("stream") des **Index Allocation Attributs**.  
Dieses Attribut ist immer **nicht-resident**.  
Die von den Index-Buffern belegten Cluster werden bei 0 beginnend durchnummeriert → **VCN**.  
Die Zuordnung der einzelnen Index-Buffer zu ihrem Speicherort (**VCN – LCN Mapping**) wird durch die **Run List** des Index Allocation Attributs hergestellt.
- ◇ Das **Bitmap Attribut** führt Buch über die aktuell **freien bzw belegten VCNs** in den Index-Buffern.  
Jeder VCN ist ein Bit zugeordnet. Das Bit ist gesetzt, wenn die entsprechende VCN verwendet wird.
- ◇ Das Index Allocation Attribut und das Bitmap Attribut sind nur dann vorhanden, wenn für das Directory tatsächlich Index-Buffer alloziert worden sind.

### • Beispiel für die Struktur eines Directories



## Directories beim NTFS (2)

### • Attribut-Wert (*stream*) des Index Root Attributs

0x00	4	Typ-Code des indizierten Attributs (bei Directory-Einträgen : Namens-Attribut 0x30)
0x04	4	immer 00 00 00 01 (?)
0x08	4	Größe der Index-Buffer (?)
0x0C	4	Anzahl Cluster pro Index-Buffer (?)
0x10	4	immer 00 00 00 01 (?)
0x14	4	Größe der Folge der Index-Einträge + 0x10
0x18	4	(?)
...0x1C	4	Flags, Bit 0 : " <i>large index flag</i> ", wenn == 1 existieren Index-Buffer weitere Bits : ?
0x20		Folge der Index-Einträge (hier : Directory-Einträge)

- Der Offset bezieht sich auf den Beginn des Attribut-Werts
- Die Folge der Index-Einträge ist mit einem **speziellen Index-Eintrag abgeschlossen** ("*last index entry*"), der keinen gültigen Directory-Eintrag repräsentiert

### • Struktur eines Index-Eintrags (hier : Directory-Eintrags)

0x00	8	<b>File Reference</b> ( → Eintrag in MFT), nur gültig, wenn das <i>last entry flag</i> nicht gesetzt ist
0x08	2	Länge des Index-Eintrags (L)
0x0A	2	Länge des indizierten Attributs (hier : Namens-Attribut) (M)
0x0C	4	Flags, Bit 0 : " <i>sub node flag</i> ", wenn == 1 Index-Eintrag zeigt auf Index-Buffer Bit 1 : " <i>last entry flag</i> ", wenn == 1 Index-Eintrag ist spezieller letzter Eintrag
0x10	M	indiziertes Attribut (hier : <b>Namens-Attribut</b> ), nur vorhanden, wenn das <i>last entry flag</i> nicht gesetzt ist
L-8	8	VCN des referierten Index-Buffers im Index Allocation Attribut, nur vorhanden, wenn das <i>sub node flag</i> gesetzt ist

- Die einzelnen Index-Einträge können **unterschiedlich lang** sein (Namens-Attribut hat eine variable Länge)
- Obwohl der letzte Index-Eintrag ("*last index entry*") keinen gültigen Directory-Eintrag repräsentiert, kann er einen Index-Buffer referieren (d.h. das *last entry flag* und das *sub node flag* können gleichzeitig gesetzt sein).

## UNIX-Dateisysteme - Allgemeines

- Es gibt **kein einheitliches UNIX-Dateisystem**, sondern je nach UNIX-System unterschiedliche – sich mehr oder weniger stark unterscheidende – Dateisystem-**Varianten**, die aber über eine Reihe **gemeinsamer** Eigenschaften und Kennzeichen verfügen.
- UNIX-Dateisysteme sind – wie die MSDOS/WINDOWS-Dateisysteme – **hierarchisch** strukturiert, d.h. neben einem **Root-Directory** können **Sub-Directories** vorhanden sein.  
Das Root-Directory ist nicht durch die logische Formatierung in der Größe begrenzt, sondern kann beliebig groß werden.  
→ Dateien werden über einen Datei(zugriffs)pfad angesprochen.

- **Datei(zugriffs)pfad**

`[/][dir/[dir/[ .../]]]dateiname`

mit führendem `'/'` : absoluter Pfad (Beginn im Root-Directory)

ohne führendem `'/'` : relativer Pfad (Beginn im aktuellen Directory)

**Beispiel :**                    `/usr/swt1/huber/dpar`    absoluter Pfad  
                              `huber/dpar`                    relativer Pfad (wenn `/usr/swt1` aktuelles Directory ist)

**dateiname :**                - 1 bis 14 (neuere Systeme bis 255) Zeichen  
                              - zulässige Zeichen : Bu Zi `'-'` `'_'` `'.'`  
                              - es existiert keine spezielle Extension,  
                              - das Zeichen `'.'` kann aber – ein- oder mehrmals – enthalten sein

- **Dateitypen :**

- ◇ Regular File (normale Datei)
- ◇ Directory
- ◇ Special File (Gerät)
  - ▷ Block Device Special File
  - ▷ Character Device Special File
- ◇ Named Pipe (FIFO)
- ◇ Symbolic Link (Soft Link)

- **Special Files (Geräte)**

üblicherweise im Directory `/dev` eingetragen

**Beispiele :**                `/dev/dsk1`                    `/dev/lp`  
                              `/dev/tty00`                `/dev/null`

Ein Gerätezugriff erfolgt als Zugriff zu diesen Dateien.  
→ Geräte werden formal wie Dateien behandelt.

- In UNIX können dem Dateipfad **keine Laufwerksbezeichnungen** vorangestellt werden, denn in einem UNIX-System existiert immer **nur ein Dateibaum** – der des Root-Laufwerks.  
Die **Dateisysteme anderer logischer Laufwerke** als des Root-Laufwerks (z.B. anderer Festplattenpartitionen oder von Disketten) werden **in den Dateibaum des Root-Laufwerks "eingehängt"** ( → **mount, umount**)  
Dateien auf den "eingehängten" Laufwerken werden über ein **spezielles Directory** dieses Dateibaums angesprochen.

**Beispiel :**                    `mount /dev/dsk1 /media/floppy`

→ Das Root-Directory von **dsk1** ist als Directory **/media/floppy** ansprechbar

- Die **Sektor-Datei-Zuordnungs-Übersicht** ist durch **Verweistabellen** realisiert

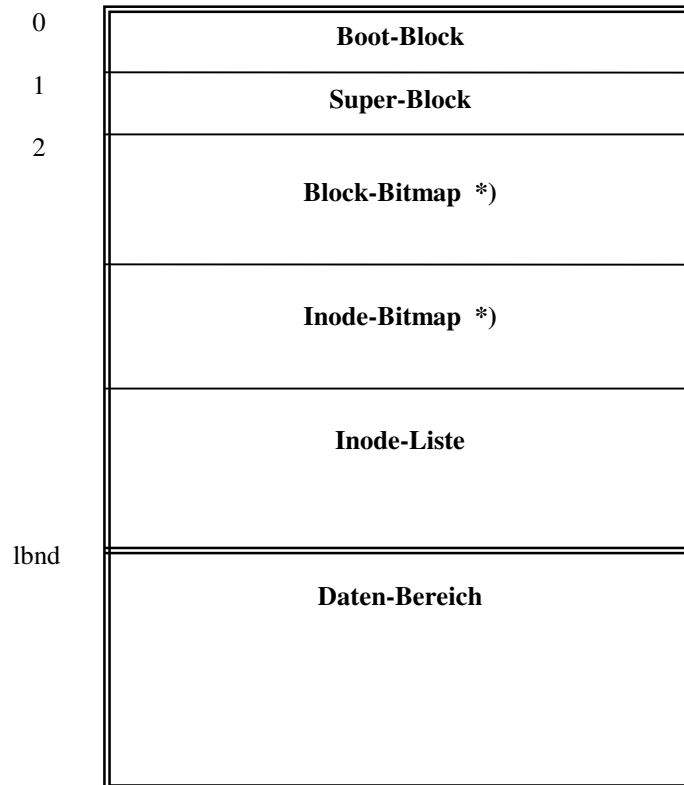
## Logische Aufteilung eines UNIX-Laufwerkes (Prinzip)

- Anmerkung :**

In UNIX-Systemen ist es üblich, logische Laufwerke in **Blöcke** zu unterteilen. → logische Block-Nummer **LBN**  
Je nach System besteht ein Block aus einem oder mehreren Sektoren.

- Aufteilung in Teilbereiche**

LBN



\*) bei einigen Varianten  
nicht vorhanden  
(dort durch (Teil-)Listen  
im Superblock ersetzt)

- ◇ **Boot-Block** : enthält das Bootstrap-Programm (leer, wenn log. Laufwerk nicht bootfähig)
- ◇ **Superblock** : enthält Verwaltungsinformationen zum Dateisystem auf dem logischen Laufwerk
- ◇ **Block-Bitmap** : enthält Belegungsinformationen über die Datenblöcke
- ◇ **Inode-Bitmap** : enthält Belegungsinformationen über die Inodes
- ◇ **Inode-Liste** : Inode = index node  
Jedes Element der Inode-Liste ist einer Datei (bzw einem Directory) zugeordnet und enthält **Beschreibungsinformationen** zu dieser Datei (dies sind Informationen die bei anderen Dateisystemen häufig im Directory-Eintrag für die Datei enthalten sind) sowie eine **Verweistabelle** zur Selektion der einzelnen Datenblöcke der Datei (Sektor-Datei-Zuordnungsübersicht)  
Einer der Inodes ist dem Root-Directory zugeordnet (→ Root-Inode)
- ◇ **Daten-Bereich** : dient zur Ablage der Dateien selbst sowie – bei einigen Dateisystem-Varianten - zur Ablage von Verwaltungsinformationen.  
Die Allokation erfolgt blockweise (ein oder mehrere Sektoren)  
typische Blockgröße : 512 bzw 1024 Bytes

## Logische Aufteilung eines Laufwerkes mit dem Dateisystem LINUX Extended 2 / Extended 3

- Besonderheit :**

Bei den **LINUX**-Dateisystemen **Extended 2** und **Extended 3** wird ein logisches Laufwerk in **Blockgruppen (Bänder)** aufgeteilt.

Die Datenblöcke und die zugehörigen Verwaltungsinformationen (Block-Bitmap, Inode-Bitmap, Inode-Liste) sind auf die einzelnen Blockgruppen aufgeteilt.

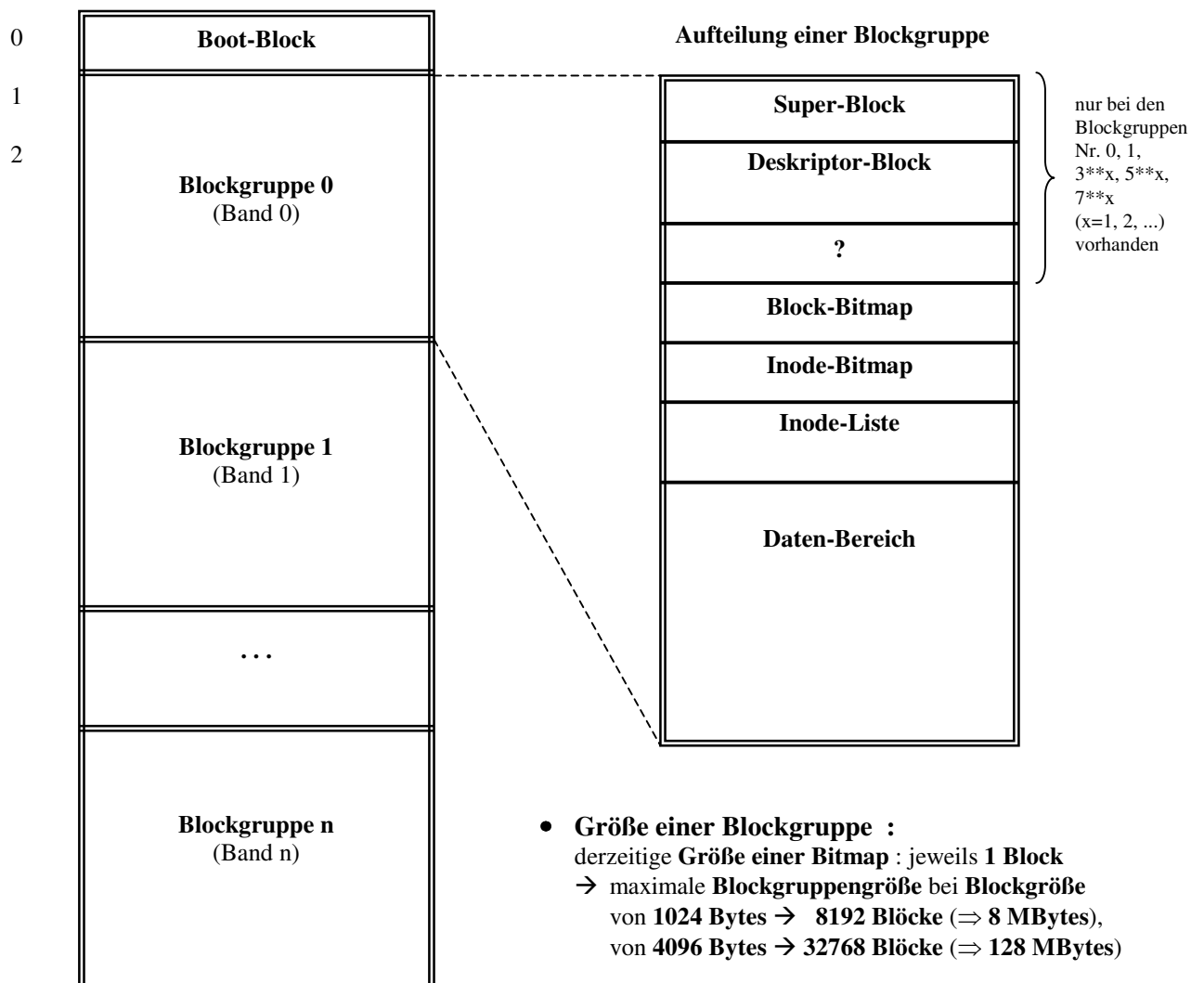
- Zusätzliche Verwaltungsstruktur :**

◇ **Deskriptor-Block** : enthält Beschreibungsinformationen für die einzelnen Blockgruppen

- Aufteilung in Teilbereiche**

**Anm. :** Aus Sicherheitsgründen enthalten die **Blockgruppen** mit den **Nr. 0, 1,  $3**x$ ,  $5**x$ ,  $7**x$**  ( $x=1, 2, \dots$ ) eine **Kopie** des **Superblocks** und des **Deskriptorblocks** (*Sparse Superblock-Technik*).

**LBN**



- Unterschied zwischen Extended 2 und Extended 3 :**

▷ Beide Dateisysteme verwenden die **gleichen Verwaltungsstrukturen**.

▷ **Extended 3** ist ein **Journaling-Dateisystem**. Es behandelt schreibende Dateisystem-Operationen als **Transaktionen**, deren Teiloperationen nach ihrer Ausführung in einem **Journal** gespeichert werden.

Das Journal ist durch eine **spezielle Journal-Datei** realisiert, die sich im **Daten-Bereich** des Laufwerkes befindet.



## Der Super-Block der Dateisysteme LINUX Extended 2 / Extended 3 (1)

- Der **2. Block** jedes logischen UNIX/LINUX-Laufwerks (LBN=1) ist der **Super-Block**.  
Er enthält **Verwaltungsinformationen** zu dem auf dem Laufwerk befindlichen Dateisystem.
- **Aufbau :**

0x000	Anzahl der Inodes	4 Bytes
0x004	Gesamtzahl der Datenblöcke (Größe des logischen Laufwerks)	4 Bytes
0x008	Anzahl reservierter Datenblöcke	4 Bytes
0x00C	Anzahl freier Datenblöcke	4 Bytes
0x010	Anzahl freier Inodes	4 Bytes
0x014	LBN des 1.Datenblocks	4 Bytes
0x018	ld (Blockgröße in Bytes / 1024)	4 Bytes
0x01C	ld (Fragmentgröße in Bytes / 1024)	4 Bytes
0x020	Anzahl Blöcke pro Blockgruppe	4 Bytes
0x024	Anzahl Fragmente pro Blockgruppe	4 Bytes
0x028	Anzahl Inodes pro Blockgruppe	4 Bytes
0x02C	Zeitpunkt des letzten Mountens	4 Bytes
0x030	Zeitpunkt des letzten Schreibzugriffs	4 Bytes
0x034	Mount Count	2 Bytes
0x036	maximaler Mount Count	2 Bytes
0x038	<i>Magic Signature</i> (Kennzeichnung des Dateisystems) (0xEF53)	2 Bytes
0x03A	Gültigkeitsflag ( <i>File System State</i> )	2 Bytes
0x03C	Festlegung des Verhaltens bei Fehlern	2 Bytes
0x03E	<i>Minor Revision Level</i>	2 Bytes
0x040	Zeitpunkt der letzten Dateisystemüberprüfung	4 Bytes
0x044	Maximale Zeit zwischen zwei Überprüfungen	4 Bytes
0x048	Erzeugungs-Betriebssystem (LINUX = 0)	4 Bytes
0x04C	<i>Revision Level</i>	4 Bytes
0x050	Default UID/GID für reservierte Blöcke (UID : 2 LS-Bytes)	4 Bytes
0x054	reserviert	940 Bytes
	u.a. bei Extended 3 an Offset 0xE0 (4 Bytes) : Inode-Nr. der Journal-Datei	

## Der Super-Block der Dateisysteme LINUX Extended 2 / Extended 3 (2)

- **Beispiel** (erster Sektor eines Super-Blocks) :

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000 :	88	17	00	00	00	5e	00	00	b3	04	00	00	87	33	00	00	.....^.....3..
00000010 :	5a	17	00	00	01	00	00	00	00	00	00	00	00	00	00	00	Z.....
00000020 :	00	20	00	00	00	20	00	00	d8	07	00	00	c4	d7	d2	47	.....G
00000030 :	c4	d7	d2	47	1f	00	1e	00	53	ef	00	00	01	00	00	00	...G...S.....
00000040 :	b5	6f	b9	47	00	4e	ed	00	00	00	00	00	01	00	00	00	..o.G.N.....
00000050 :	00	00	00	00	0b	00	00	00	80	00	00	00	30	00	00	00	.....0...
00000060 :	02	00	00	00	01	00	00	00	3e	bf	c6	d1	45	e7	46	5f	.....>...E.F_
00000070 :	a4	24	90	34	64	5a	07	d4	00	00	00	00	00	00	00	00	\$.4dZ.....
00000080 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000090 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000a0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000b0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000c0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	5d	00	.....l.
000000d0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000e0 :	00	00	00	00	00	00	00	00	00	00	00	00	41	22	64	e3	.....a'd.
000000f0 :	43	8a	46	82	ba	66	47	18	cc	9e	00	01	02	00	00	00	C.F..fG.....
00000100 :	00	00	00	00	00	00	00	00	52	9f	34	47	00	00	00	00	.....R.4G....
00000110 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000120 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000130 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000140 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000150 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000160 :	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000170 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000180 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000190 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001a0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001b0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001c0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001d0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001e0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001f0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF

- **Inhalt einiger Eintragsfelder :**

0x000 : Anzahl der Inodes : 0x00001788 = 6 024  
 0x004 : Gesamtzahl der Datenblöcke : 0x00005E00 = 24 064  
 0x008 : Anzahl reservierter Datenblöcke : 0x000004B3 = 1 203  
 0x00C : Anzahl freier Datenblöcke : 0x0000216C = 13 191  
 0x010 : Anzahl freier Inodes : 0x00000F88 = 5 978

0x018 : ld (Blockgröße in Bytes / 1024) : 0x00000000 → Blockgröße : 1024 Bytes = 2 Sektoren

0x020 : Anzahl Blöcke pro Blockgruppe : 0x00002000 = 8 192

0x028 : Anzahl Inodes pro Blockgruppe : 0x000007D8 = 2 008

0x038 : Magic Signature : 0xEF53

Anzahl Blockgruppen = Anzahl Inodes / Anzahl Inodes pro Blockgruppe = 6 024 / 2008 = 3

## Der Deskriptor-Block der Dateisysteme LINUX Extended 2 / Extended 3

- Allgemeines :**

Der Deskriptor-Block enthält die **Beschreibungsinformationen** für die einzelnen **Blockgruppen** (Datenbänder).  
Er folgt unmittelbar auf den Super-Block → Beginn bei LBN = 2.

Pro Blockgruppe enthält er einen Eintrag von **32 Bytes** Länge.

→ Die **Größe** des Deskriptor-Blocks hängt von der **Anzahl der Blockgruppen** ab.

**Beispiel :** 255 Blockgruppen → 8160 Bytes

→ belegt werden 8 Datenblöcke zu je 1024 Bytes

Nicht verwendete Einträge der belegten Datenblöcke werden mit 00-Bytes aufgefüllt.

- Aufbau eines Deskriptor-Block-Eintrags :**

0x00	LBN Beginn Block-Bitmap	4 Bytes
0x04	LBN Beginn Inode-Bitmap	4 Bytes
0x08	LBN Beginn Inode-Liste	4 Bytes
0x0C	Anzahl freier Datenblöcke	2 Bytes
0x0E	Anzahl freier Inodes	2 Bytes
0x10	Anzahl verwendeter Directories	2 Bytes
0x12	Füll-Bytes	2 Bytes
0x14	reserviert	12 Bytes

- Beispiel für einen Deskriptor-Block mit drei Einträgen :**

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000 :	60	00	00	00	61	00	00	00	62	00	00	00	95	12	c0	07	`...a...b.....
00000010 :	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000020 :	60	20	00	00	61	20	00	00	62	20	00	00	6d	16	c5	07	`...a...b...m...
00000030 :	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000040 :	01	40	00	00	02	40	00	00	03	40	00	00	85	0a	d5	07	.e...e...e.....
00000050 :	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000060 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000070 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000080 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000090 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000a0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000b0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000c0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000d0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000e0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000f0 :	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

**Erster Eintrag :** LBN Beginn Block-Bitmap : 0x00000060 = 96  
 LBN Beginn Inode-Bitmap : 0x00000061 = 97  
 LBN Beginn Inode-Liste : 0x00000062 = 98  
 Anzahl freier Datenblöcke : 0x1295 = 4757  
 Anzahl freier Inodes : 0x07C0 = 1984  
 Anzahl Directories : 0x0002 = 2

## Inodes der Dateisysteme LINUX Extended 2 / Extended 3 (1)

### • Allgemeines :

- ◇ Jeder auf einem logischen UNIX-Laufwerk abgelegten Datei (einschließlich Directory) ist eine **Inode** zugeordnet.
- ◇ Eine Inode enthält **Beschreibungs-Informationen** zu einer Datei sowie eine **Verweistabelle** zur Selektion der einzelnen Datenblöcke der Datei.
- ◇ Bei den Dateisystemen LINUX Extended 2 /Extended 3 sind die Inodes auf **mehrere Inode-Listen** aufgeteilt (pro Blockgruppe eine Liste).  
Die **Gesamtzahl inmax** der Inodes sowie die Anzahl der Inodes pro einzelner Liste wird durch die logische Formatierung festgelegt.
- ◇ **Inode-Nummer :** 1 .. inmax  
 Inode 1 : reserviert (für Pseudo-Datei, bestehend aus defekten Blöcken)  
 Inode 2 : für Root-Directory  
 Inode 3 .. 10 : reserviert (z.B Inode 8 für die Journal-Datei bei Extended 3)  
 Inode 11 : erste frei verwendete Inode

**Größe einer Inode bei LINUX Extended 2 / Extended 3 : 128 Bytes**

### • File Mode

- ◇ Erster Eintrag in der Inode (2 Bytes).  
Er besteht aus :
  - ▷ den **Zugriffsrechten** und
  - ▷ dem **Dateityp**
- ◇ **Dateizugriffsrechte :**

Owner		Read ( r )
Group	jeweils	Write ( w )
Other User		Execute ( x )

z.B. **rwX r-X r-X**
- ◇ **Dateitypen :**
  - Normale Datei (Regular File)
  - Directory ( d )
  - Character Device ( c )
  - Block Device ( b )
  - Symbolic Link ( l )
  - Named Pipe (Fifo) ( p )
  - Socket ( s )

### ◇ Aufbau des File Mode Eintrags (2 Bytes)

Bit-Nr      15   14   13   12   11   10   9   8   7   6   5   4   3   2   1   0

Dateityp	spezielle Zugriffs- modifizier	Owner <b>r   w   x</b>	Group <b>r   w   x</b>	Other User <b>r   w   x</b>
----------	-----------------------------------	---------------------------	---------------------------	--------------------------------

0001 Named Pipe  
 0010 Character Device  
 0100 Directory  
 0110 Block Device  
 1000 Regular File  
 1010 Symbolic Link  
 1100 Socket

1-- **setuid**-Bit  
 -1- **setgid**-Bit  
 --1 **sticky**-Bit

## Inodes der Dateisysteme LINUX Extended 2 / Extended 3 (2)

- **Aufbau einer Inode (Größe 128 Bytes) :**

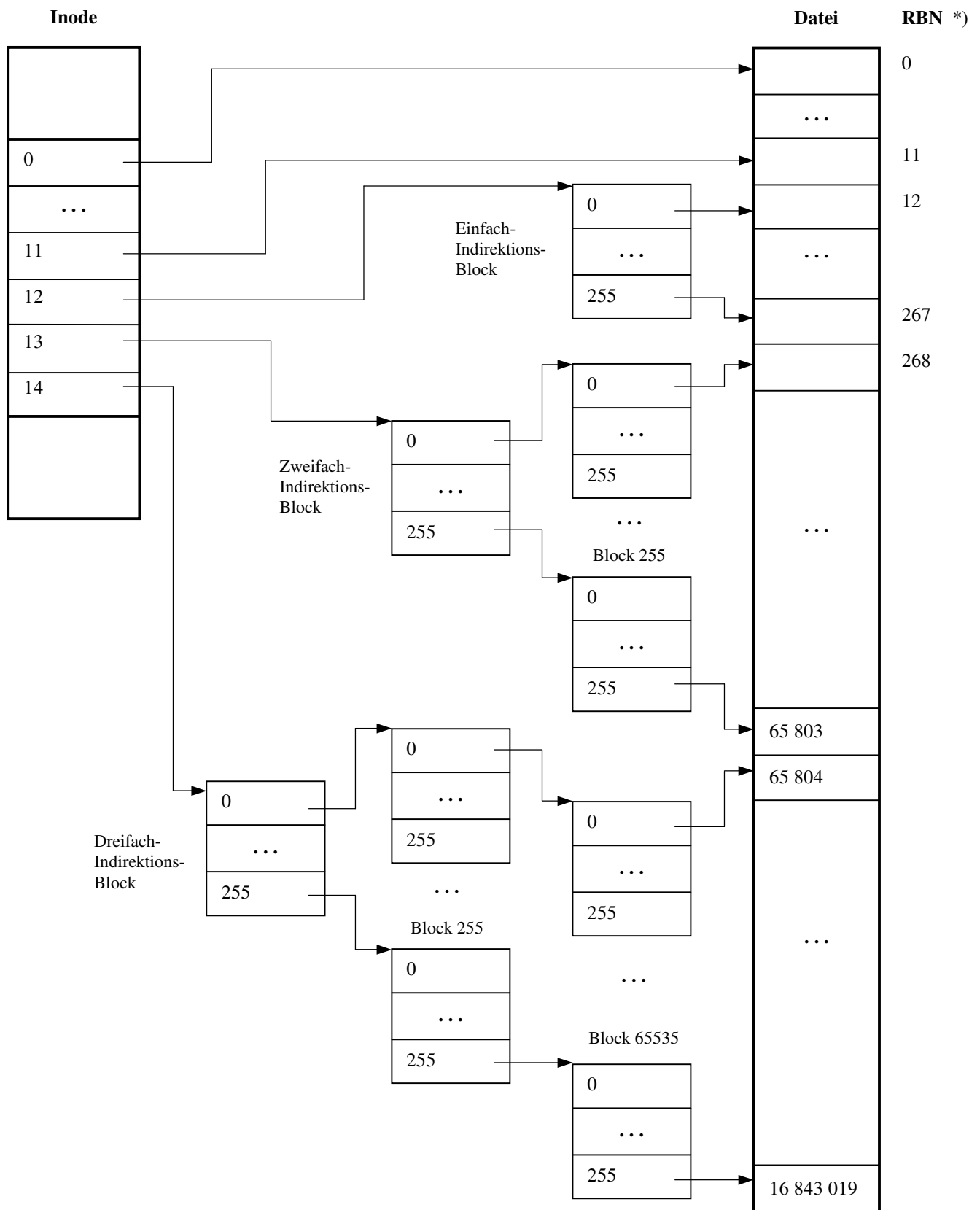
0x00	File Mode (Dateityp und Zugriffsrechte)	2 Bytes
0x02	User ID ( <i>Owner</i> )	2 Bytes
0x04	Dateigröße in Bytes (bei Symbolic Link : Länge des Zielpfades)	4 Bytes
0x08	Zeitpunkt des letzten Zugriffs ( <i>Access Time</i> )	4 Bytes
0x0C	Zeitpunkt der Dateierzeugung ( <i>Creation Time</i> )	4 Bytes
0x10	Zeitpunkt der letzten Änderung ( <i>Modification Time</i> )	4 Bytes
0x14	Zeitpunkt des Löschens der Datei ( <i>Deletion Time</i> )	4 Bytes
0x18	Group ID	2 Bytes
0x1A	Anzahl der Links auf Datei ( <i>Link Count</i> )	2 Bytes
0x1C	Anz. allok. Sektoren für Datei ( <i>Block Count</i> ) (kurzer Symbolic Link : 0)	4 Bytes
0x20	Datei-Attribute ( <i>File Flags</i> )	4 Bytes
0x24	reserviert (OS abhängig)	4 Bytes
0x28	LBN des 1. bis 12. Dateiblocks	12*4 Bytes
0x58	LBN des Einfach-Indirektionsblocks	4 Bytes
0x5C	LBN des Zweifach-Indirektionsblocks	4 Bytes
0x60	LBN des Dreifach-Indirektionsblocks	4 Bytes
bei kurzen Symbolic Links (bis zu 60 Bytes) :  Ziel-Dateipfad		60 Bytes
0x64	Dateiversion	4 Bytes
0x68	File ACL (zukünftige Erweiterung der Zugriffskontrolle)	4 Bytes
0x6C	Directory ACL (zukünftige Erweiterung der Zugriffskontrolle)	4 Bytes
0x70	Fragment-Adresse	4 Bytes
0x74	Fragment-Nummer	1 Byte
0x75	Fragment-Größe	1 Byte
0x76	reserviert (OS abhängig)	10 Bytes

- **Beispiele :**

[illegible][illegible][illegible][illegible]

## Struktur der Datenblock-Verweise in den Dateisystemen LINUX Extended 2 / Extended 3

- Annahmen :** Blockgröße 1024 Bytes, Größe einer LBN 4 Bytes → **max. Dateigröße : 16 GBytes**



\*) **RBN** – relative Blocknummer innerhalb der Datei

## Directory-Einträge der Dateisysteme LINUX Extended 2 / Extended 3

### • Allgemeines :

- ◇ Ein **Directory-Eintrag** stellt lediglich den **Zusammenhang** zwischen dem **Dateinamen** und der Nummer des zu der Datei gehörenden **Inodes** her.
- ◇ In den Dateisystemen LINUX Extended 2 /Extended 3 kann ein Dateiname 1 bis **255 Zeichen** lang sein.
- ◇ Um nicht unnötig Platz zu verschwenden, haben Directory-Einträge eine **variable** – an die Länge des Dateinamens angepaßte – **Länge**. Diese beträgt aber immer ein **Vielfaches von 4 Bytes**.

### • Aufbau eines Directory-Eintrags :

0x00	Inode-Nummer	4 Bytes
0x04	Länge des Directory-Eintrags (immer Vielfaches von 4)	2 Bytes
0x06	Länge des Dateinamens	1 Byte
0x07	Dateityp *)	1 Byte
0x08	Dateiname	1 – 255 Bytes
	Füll-Bytes	(Anzahl angepaßt an durch 4 teilbare Gesamtlänge)

\*) **Codierung des Dateityps :**  
(in den letzten 3 LS-Bits)

000	unbekannt
001	normale Datei
010	Directory
011	Character Device
100	Block Device
101	Named Pipe
110	Socket
111	Symbolic Link

### • Beispiel (Anfang eines Root-Directories) :

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000 :	02	00	00	00	0c	00	01	02	2e	00	00	00	02	00	00	00	.....
00000010 :	0c	00	02	02	2e	2e	00	00	0b	00	00	00	14	00	0a	02	.....
00000020 :	6c	6f	73	74	2b	66	6f	75	6e	64	00	00	d9	07	00	00	lost+found.....
00000030 :	18	00	04	02	67	72	75	62	3b	34	37	33	34	61	31	35	....grub;4734a15
00000040 :	31	00	00	00	0c	00	00	00	0c	00	04	07	62	6f	6f	74	1.....boot
00000050 :	0f	00	00	00	10	00	07	01	6d	65	73	73	61	67	65	00	.....message.
00000060 :	b2	0f	00	00	10	00	08	02	52	65	63	79	63	6c	65	64	.....Recycled
00000070 :	11	00	00	00	34	00	24	01	73	79	6d	73	65	74	73	2d	....4.\$symsets-
00000080 :	32	2e	36	2e	32	32	2e	31	37	2d	30	2e	31	2d	64	65	2.6.22.17-0.1-de
00000090 :	66	61	75	6c	74	2e	74	61	72	2e	67	7a	35	35	65	63	fault.tar.gz55ec
000000a0 :	35	00	00	00	0e	00	00	00	28	00	20	01	53	79	73	74	5.....\$.syst
000000b0 :	65	6d	2e	6d	61	70	2d	32	2e	36	2e	32	32	2e	31	37	em.map-2.6.22.17
000000c0 :	2d	30	2e	31	2d	64	65	66	61	75	6c	74	1c	00	00	00	-0.1-default....
000000d0 :	10	00	07	07	76	6d	6c	69	6e	75	7a	00	1d	00	00	00	....vmlinuz....
000000e0 :	10	00	06	07	69	6e	69	74	72	64	7a	2d	10	00	00	00	....initrdz-....
000000f0 :	24	00	1c	01	63	6f	6e	66	69	67	2d	32	2e	36	2e	32	\$....config-2.6.2
Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF



# **Betriebssysteme**

## **Kapitel 4**

### **4. Einführung in LINUX**

4.1. Von UNIX zu LINUX

4.2. Wesentliche Eigenschaften

4.3. Architektur

4.4. Bootvorgang

4.5. Run-Level

## Von UNIX zu LINUX (1)

### • Allgemeines zu LINUX

LINUX ist ein frei verfügbares **UNIX-artiges** Betriebssystem, das ursprünglich für PCs der Intel-Architektur (damals 80386) entwickelt wurde. Inzwischen stehen aber auch Portierungen auf eine Reihe weiterer Prozessor-Architekturen (z.B. Digital-Alpha, Sparc, MIPS, Power PC, 68k) zur Verfügung..

LINUX vereinigt weitgehend die **Funktionalität** der beiden großen UNIX-Linien (**System V** und **BSD**) und ist kompatibel zum **POSIX-Standard IEEE-1003.1- 2008** (POSIX – *Portable Operating System Interface Specification*).

Der LINUX-Quellcode (und der Quellcode der meisten zugehörigen Dienstprogramme) steht unter der **GNU General Public License** der **Free Software Foundation** (FSF).

Das bedeutet, das jeder das Recht hat, die Software frei – ohne Kosten – zu verwenden, zu kopieren, zu modifizieren und weiterzugeben. Durchgeführte Änderungen müssen ebenfalls frei zur Verfügung gestellt werden.

### • Entwicklung von UNIX

**1965** Bell Labs von AT&T, General Electric und MIT beginnen mit der Entwicklung eines Multi-User-Betriebssystems **MULTICS** (*Multiplexed Information and Computing Service*). Wegen ständiger Misserfolge stellte Bell Labs nach 4 Jahren seine Mitarbeit an dem Projekt ein..

**1969** **Ken Thompson** und **Dennis Ritchie** der **Bell Labs** entwickeln eine abgemagerte Single-User-Version des ursprünglich geplanten MULTICS für die **PDP-7**, die sie **UNICS** (*Uniplexed Information and Computing Service*) nennen. Später wurde die Schreibweise in **UNIX** geändert. Der Quellcode ist in Assembler geschrieben.

**1971** Portierung auf **PDP-11**, Multi-User-Unterstützung, erstes *UNIX Programmers Manual* (**UNIX Version 1**)

**1972** Implementierung großer Teile von UNIX in **C** → Grundlage einer **Portabilität**

**1973** Erste Veröffentlichung über UNIX (**Version 4**) (*Symposium on Operating System Principles* im Oktober)

**1977** Erste Portierung (**UNIX Version 6**) auf eine Nicht-PDP-Maschine : **Interdata 8/32**

Startschuß für weitere UNIX-Portierungen durch andere Firmen.(z.B. XENIX)  
Beginn der weltweiten Verbreitung von UNIX

**1978** Erste Version von **BSD-Unix** (BSD : *Berkley Software Distribution*, University of California Berkley), abgeleitet von UNIX Version 6

**1979** **UNIX Version 7** : wesentliche **Verbesserungen** und **Ergänzungen** sowohl im Kernel als auch in zur Verfügung stehenden Dienstprogrammen  
Beginn von Portierungen im größeren Umfang

In der Folgezeit **zwei Haupt-Entwicklungslinien** von UNIX : **AT&T** und **BSD**, mit zum Teil unterschiedlichen Funktionalitäten, aber auch gegenseitiger Beeinflussung  
AT&T-Versionen werden vor allem im kommerziellen Umfeld, BSD-Versionen an Universitäten eingesetzt.  
Daneben zahlreiche firmenspezifische Nebenlinien.

**1980** **BSD 4** , stark beeinflusst von AT&T-UNIX Version 7

**1983** AT&T UNIX **System V** → "Standard"-UNIX

In der weiteren Entwicklung werden viele Konzepte und Funktionalitäten von BSD-Unix-Versionen aufgenommen.  
Festlegung der System-Schnittstellen (Programmier-Schnittstelle und Benutzer-Kommandos)

**1985** Gründung der **Open Software Foundation** (OSF), ein Zusammenschluß mehrerer Firmen (u.a. IBM, Digital, HP, Nixdorf, Bull, Apollo) mit dem Ziel der Entwicklung einer von AT&T unabhängigen "offenen" und einheitlichen BS-Umgebung.

**1988** Erste Version des **POSIX-Standards** (POSIX 1), entwickelt von IEEE, Ziel : einheitliches API für UNIX-artige BS. Weiterentwicklung unter späterer Beteiligung der Open Group (OSF + X/Open)

**1989** **UNIX System V Release 4,(SVR4)**, letzte "offizielle" UNIX-Version von AT&T.  
In der Folgezeit verkauft AT&T "ihr" UNIX an Novell.

**1994** **BSD 4.4**, letzte Berkley-Unix-Version, University of California stellt Weiterentwicklung von UNIX ein

In der Folgezeit wird die Entwicklung von UNIX vor allem - aber nicht nur - von mehreren Firmen - meist auf der Basis von BSD 4.4 - weitergeführt→ weiterhin **zahlreiche UNIX-Derivate**.  
Heute existieren für nahezu jede Hardware-Plattform z.Tl. mehrere unterschiedliche UNIX-Implementierungen (z.B. Solaris von Sun). Die meisten von ihnen streben **POSIX-Kompatibilität** an

## Von UNIX zu LINUX (2)

### • Entwicklung von LINUX

**1991 Linus Torvalds** (finnischer Student) entwickelt – ausgehend von dem Lehrbetriebssystem **MINIX** – eine eigene Version eines unix-artigen Betriebssystems für 80386-PCs, das er **LINUX** nannte.

Er stellte die Quellen unter die **GNU Public License** und veröffentlichte sie im **Internet**. (**Version 0.1**)

Es bildete sich schnell auf freiwilliger Basis eine Gruppe von LINUX-Enthusiasten, die die Entwicklung dieses Betriebssystems vorantrieben

Da gleichzeitig zahlreiche Nutzer alle Änderungen und Ergänzungen testen konnten, entstanden schnell stabile Versionen.

**1994 Version 1.0** mit erster **TCP/IP**-Unterstützung.

? **Version 1.2** mit vollständiger Unterstützung des **POSIX-1003.1**-Standards

? **Version 2.0** mit **SMP** (*Symmetric Multiprocessor*) -Unterstützung

- ◇ Heute arbeiten - unentgeltlich in ihrer Freizeit - immer mehr Programmierer und Tester über die ganze Welt verstreut und über das Internet kommunizierend an der Weiterentwicklung von LINUX

→ **offener Entwicklungsprozeß** ("*bazaar*" development)

Dieser offene Entwicklungsprozeß führt auch dazu, daß für neue Peripherie-Hardware (Steckkarten) sehr schnell funktionsfähige Treiber bereit stehen.

Das gleiche Entwicklungsmodell wird auch für die zahlreichen LINUX-Dienst- und Anwendungsprogramme, ebenfalls unter der GNU Public License, angewendet.

Im Prinzip sind auf diese Art und Weise fast sämtliche üblichen UNIX-Dienstprogramme sowie zahlreiche Anwendungssysteme für LINUX ebenfalls frei verfügbar.

- ◇ Versions-Nummern des Linux-Kernel werden folgendermassen bezeichnet : **x.y.z** (z.B. 2.4.32).  
Ab dem Kernel 2.6 werden auch vierzählige Bezeichnungen verwendet : **x.y.z.a** (z.B. 2.6.15.3)

Um eine geordnetes Voranschreiten der Entwicklung sicherzustellen, existierten **bis zum Kernel 2.6 zwei Versions-Linien** :

- Versionen der Entwicklungs-Kernel
- Versionen der stabilen Kernel.

**Entwicklungs-Kernel** hatten ein **ungerades y** in der Versions-Nummer, **stabile Kernel** dagegen ein **gerades y**.  
Änderungen und Neuerungen wurden zuerst im Entwicklungs-Kernel eingeführt.

Stellte eine Änderung eine Fehlerbehebung dar, wurde sie – nach ausreichendem Test - auch im stabilen Kernel vorgenommen.

War der Entwicklungs-Kernel x.y.z ausreichend weiterentwickelt und stabil, wurde er zum stabilen Kernel x.(y+1).0.  
Beispiel : aus dem Entwicklungs-Kernel **2.3.99** wurde der stabile Kernel **2.4.0**

Ab dem Kernel 2.6 wurde diese Trennung in Entwicklungs- und stabilen Kernel offensichtlich aufgegeben.

Die Kernel-Entwicklung wird jetzt anders verwaltet.

Die jeweils **neuesten Kernel-Versionen** stehen im Internet unter der URL **<http://www.kernel.org>** zur Verfügung.

- ◇ Der LINUX-Kernel ist zum größten Teil in **C** geschrieben.  
Ein kleinerer – im wesentlichen architekturabhängiger – Teil ist in **Assembler** formuliert.  
Zur Übersetzung wird der **GNU-C-Compiler** bzw der jeweilige GNU-Assembler verwendet.  
Dabei werden im C-Quellcode einige – vom ANSI-Standard abweichende – Besonderheiten des GNU-C-Compilers ausgenutzt (z.B. *inline*-Funktionen).  
Weiterhin wird im Quellcode – in erster Linie zur Geschwindigkeitsoptimierung – durchaus gegen einige Regeln eines "guten" Programmierstils verstoßen (z.B. Verwendung der *goto*-Anweisung, im Mittel ca 1 *goto* / 260 Zeilen)

### • LINUX-Distributionen

Obwohl der LINUX-Kernel-Quellcode von jedermann kostenlos aus dem Internet geladen und zum eigenen Einsatz übersetzt werden kann, werden von mehreren Firmen kostenpflichtige **LINUX-Distributionen** angeboten.

Diese enthalten neben dem **Quellcode** bereits übersetzte **vorkonfigurierte Standard-Kernel** sowie zahlreiche **Dienst- und Anwendungsprogramme** zusammen mit einer **Installations-Software** und ermöglichen so eine i.a. sehr **einfache Installation** des Betriebssystems (z.B. SUSE, DLD, Red Hat, Caldera, Slackware, Debian, Ubuntu).

## Wesentliche Eigenschaften von LINUX (1)

- **Multitasking**

LINUX unterstützt echtes **präemptives** Multitasking

- **Multiuser**

LINUX ermöglicht mehreren Benutzern gleichzeitig mit dem System zu arbeiten und stellt eine ausgereifte Benutzerverwaltung zur Verfügung

- **Multiprozessorfähig**

LINUX läuft ab der Version 2.0 auch auf **Mehrprozessorarchitekturen** (*SMP Symmetric Multiprocessor*). Mehrere Prozesse können echt parallel auf mehreren Prozessoren laufen

- **Portabilität**

Es existieren LINUX-Implementierungen für zahlreiche unterschiedliche CPU-Plattformen.

Die Portierung von LINUX auf eine andere Plattform wird durch die klare Trennung des Source-Codes in architektur-unabhängige und architektur-abhängige Teile unterstützt

- **Virtuelle Speicherverwaltung mittels *Paging***

Die Standard-Seitengröße beträgt 4 kBytes. Sie kann leicht an die von der jeweiligen Hardware-Architektur unterstützte Größe angepaßt werden

- ***Demand Load Executables***

Es werden nur die Teile eines Programms in den Arbeitsspeicher geladen, die zu seiner Ausführung tatsächlich benötigt werden.

Bei der Erzeugung eines neuen Prozesses wird für diesen zunächst der Datenspeicherbereich des Elternprozesses verwendet. Erst wenn einer der beiden Prozesse auf den Datenbereich schreibend zugreift, wird auch für den neu erzeugten Prozeß durch Kopieren ein eigener Datenspeicherbereich bereitgestellt (→ Kopieren nur bei Bedarf, ***Copy-on-Write***)

- ***Shared Libraries***

**Bibliotheks-Code**, der von **mehreren Programmen genutzt** werden kann, wird **nur einmal** in den **Arbeitsspeicher geladen**. Dieser Code wird erst **zur Laufzeit** zum jeweiligen Programm-Code hinzugebunden.

→ dynamisch gebundene Bibliotheken, *Dynamic Link Libraries* (**DLL**)

- **Speicherschutz**

Unter Ausnutzung der **Speicherschutzmechanismen** der jeweiligen CPU verhindert LINUX den Zugriff eines Prozesses zu dem Speicher des Betriebssystems bzw zum Speicher anderer Prozesse.

Der schreibende Zugriff zu schreibgeschütztem Speicher durch den eigenen Prozeß wird ebenfalls verhindert.

Ein fehlerhaftes Programm kann deshalb das System – theoretisch – nicht zum Absturz bringen.

→ entscheidender Beitrag zur **Sicherheit** und zur **Stabilität** des Systems.

- **Dynamischer Cache für Festplatten**

Die Größe des für Festplattenzugriffe verwendeten Caches im Arbeitsspeicher wird dynamisch der jeweiligen Speicherauslastung angepaßt.

Wird mehr Arbeitsspeicher benötigt, wird die Größe des Caches reduziert. Wird Arbeitsspeicher wieder freigegeben, kann der Cache vergrößert werden.

## Wesentliche Eigenschaften von LINUX (2)

- **Unterstützung des POSIX-1003.1-Standards, sowie von System V und BSD**

POSIX 1003.1 definiert eine minimale C-basierte **Programmierschnittstelle** für UNIX-ähnliche Betriebssysteme.

Ab Version 1.2 unterstützt LINUX diesen Standard vollständig.

Einige LINUX-Distributionen haben den offiziellen Zertifizierungsprozeß durchlaufen und dürfen sich deshalb offiziell "**POSIX-kompatibel**" nennen.

Zusätzlich sind weitere Programmierschnittstellen von **System V** und **BSD-UNIX**, die im POSIX-Standard nicht enthalten sind, implementiert.

Im Quellcode vorliegende Software für UNIX läßt sich daher in der Regel ohne größere Probleme für LINUX übersetzen.

→ UNIX-Software ist **quellcode-kompatibel** zu LINUX.

- **Unterstützung verschiedener Formate für ausführbare Dateien**

Das Standardformat für ausführbare Dateien unter LINUX ist das **ELF-Format** (*Executable and Linking Format*).

Daneben kann LINUX auch Programme, die im alten **a.out-Format** vorliegen, ausführen.

Weiterhin ist LINUX – durch Bereitstellung entsprechender **Simulatoren** – in der Lage, **Programme anderer UNIX-Systeme**, die dem **iBCS2-Standard** (*Intel Binary Compatibility Specification 2*) entsprechen, sowie – z. Tl. noch eingeschränkt – **MS-DOS-** und **Windows-**Programme "direkt" auszuführen.

Der Kernel stellt Unterstützung für die Aufnahme weiterer ausführbarer Formate zur Verfügung.

→ Dem Anwender steht auch zahlreiche Software zur Verfügung, die nicht speziell auf LINUX portiert worden ist.

- **Unterstützung verschiedener Dateisysteme**

Neben den derzeitigen LINUX-Standard-Dateisystemen (**Extended 2, Extended 3, Reiser-FS**) unterstützt LINUX **nahezu alle im PC-Bereich gängigen** Dateisysteme sowie das **NFS** (*Network File System*) zum transparenten Zugriff auf Dateisysteme anderer Netzwerkrechner.

- **Unterstützung nationaler Tastaturen und Zeichensätze**

LINUX ermöglicht das Arbeiten mit den unterschiedlichsten nationalen Tastaturen und Zeichensätzen.

Da der von der ISO genormte Zeichensatz *Latin1* auch die deutschen Umlaute enthält, ist die Verwendung anderer Zeichensätze in Deutschland nicht unbedingt erforderlich.

- **Netzwerk-Unterstützung**

LINUX enthält eine vollständige Implementierung von **TCP/IP**.

Zusätzlich unterstützt LINUX auch **SLIP** (*Serial Line Internet Protocol*) und **PLIP** (*Parallel Line Internet Protocol*) sowie **PPP** (*Point to Point Protocol*). Diese ermöglichen einen Zugriff auf TCP/IP-Netzwerke über eine serielle bzw. parallele Verbindung.

Auch andere gängige Netzwerkprotokolle sind implementiert (wie z.B. **Apple Talk, IPX** usw.)

→ LINUX-Rechner lassen sich sehr einfach in **lokale Netze integrieren**.

- **Dynamisch ladbare Module**

Module sind **Kernel-Code-Bestandteile**, die **nicht fest** in den Kernel eingebunden sind, sondern **bei Bedarf geladen** werden.

Besonders geeignet für Module sind Gerätetreiber und Dateisysteme.

Ab der Version 2.2 stellt LINUX einen automatischen Mechanismus für das für ein Anwenderprogramm vollkommen transparente Laden von Modulen zur Verfügung.

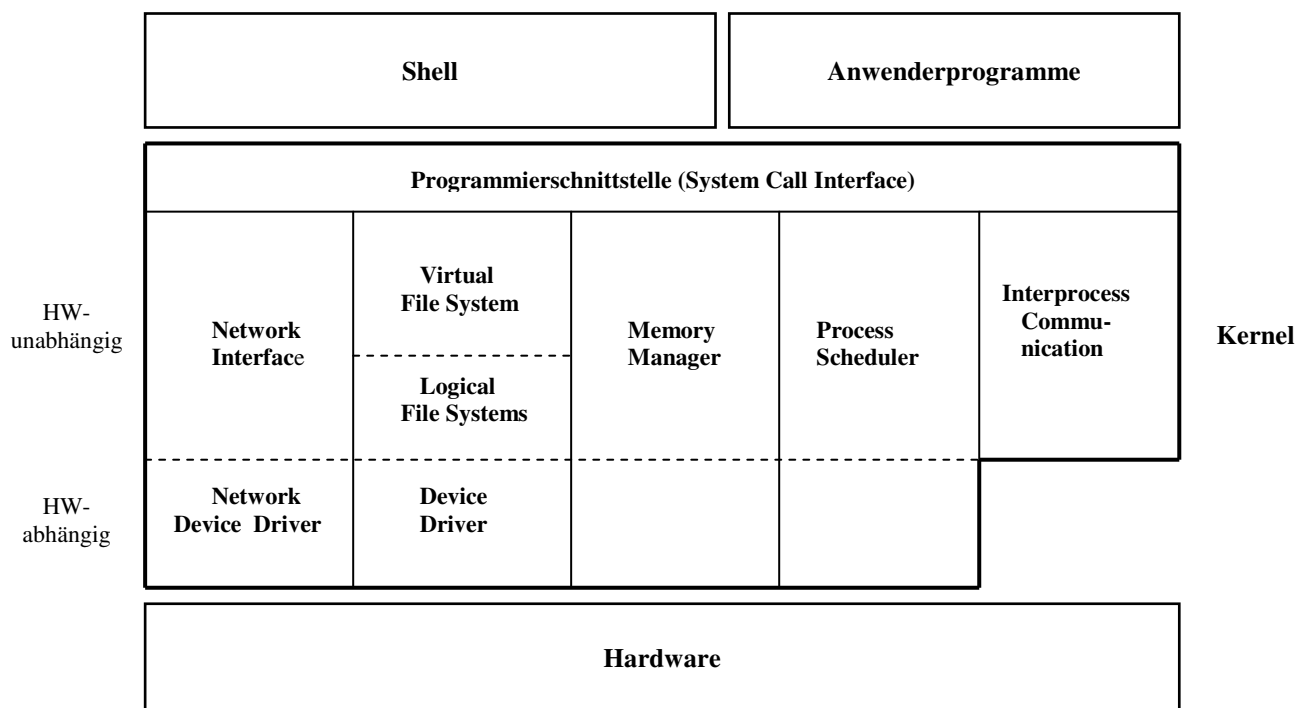
Module erlauben es, einen kleinen kompakten Kernel zu verwenden, dem nur bei Bedarf weitere benötigte Funktionalität hinzugefügt wird.

Außerdem erleichtern Module sehr stark das Erstellen und Testen neuer Treiber und Dateisysteme.

## Architektur von LINUX (1)

### • Grundsätzliche Struktur

- ◇ LINUX besitzt einen intern **modular** aufgebauten **monolitischen Kernel** (keine Microkernel-Architektur).
- ◇ Die einzelnen Module bestehen gegebenenfalls aus mehreren **Schichten** (hardware-unabhängige, hardware-abhängige Schicht)
- ◇ Sowohl die einzelnen Module als auch die einzelnen Schichten innerhalb eines Moduls kommunizieren miteinander weitgehend über wohldefinierte **Schnittstellen**.  
Damit umfaßt der üblicherweise in der Datei **vmlinux** abgelegte **Kernel** die **gesamte Betriebssystem-Kernfunktionalität einschließlich BIOS** (aber ohne Shell).
- ◇ Der Kernel ist weitgehend **konfigurierbar** (**Neukompilierung** und **Kernel-Parameter** zur Boot-Zeit).  
Dadurch läßt er sich optimal an die jeweilige System-Konfiguration anpassen.



### • Kernel-Module

- ◇ Mittlerweile (im derzeitigen Umfang seit Version 2.2) ermöglicht LINUX, einen Teil der konfigurierbaren Kernel-Funktionalität ( vor allem Treiber und Dateisysteme) aus dem eigentlichen **Kernel auszulagern** und als eigenständige Module zur Laufzeit bei **Bedarf nachzuladen** und wieder zu **entfernen**. (→ dynamisch ladbare Treiber und Dateisysteme).
- ◇ Diese Möglichkeit stellt einen **Schritt** in Richtung einer **Microkernel-Architektur** dar (allerdings laufen die nachladbaren Module nicht als eigene Prozesse sondern als direkter Teil des Kernels.)

### • Treiber als eigene Prozesse

- ◇ Es gibt auch **Treiber** für einige **Peripherie-Geräte** (z.B. für Drucker, Maus, Modem, Graphikkarte), die nicht zum Kernel gehören, sondern als eigenständige Programme geladen werden und als **eigenständige Prozesse** laufen.
- ◇ Auch dies zeigt einen Ansatz zu einer Microkernel-Architektur.

## Architektur von LINUX (2)

### • Kernel Threads als Dämonen

- ◇ **Dämonen** sind **Hintergrundprozesse**, die **ohne Benutzerinteraktion** ablaufen.  
Eine Reihe von **System-Aktivitäten** werden durch derartige Dämonen wahrgenommen.  
Diese Dämonen laufen im **System-Mode** und können **direkt** zu den **Datenstrukturen des Kernels** zugreifen.  
Es handelt sich um **Kernel Threads**.
- ◇ **Beispiele** für Kernel Threads :
  - ▷ **kflushd** : regelmäßiges Rausschreiben von Schreibpuffern in den Hintergrundspeicher (Platte)
  - ▷ **kswapd** : Überprüfung auf nicht verwendete physikalische Seiten, Auslagerung und Freigabe dieser Seiten
- ◇ **Anmerkung** : Es gibt auch **Dämonen**, die als "normale" Prozesse im **User-Mode** laufen, z.B.
  - ▷ **cron** : Ausführung beliebiger Kommandos zu vorbestimmten Zeiten
  - ▷ **lpd** : Entgegennahme und Weiterleitung von Druckaufträgen

### • LINUX Shells

- ◇ LINUX ermöglicht den Start **unterschiedlicher Shells** (Kommando-Prozessoren).
- ◇ Shells unixartiger Betriebssysteme stellen i.a. eine Programmiersprache zur Verfügung, mit Hilfe derer man **Shell-Skripte** formulieren kann. Diese ermöglichen den automatisierten und gesteuerten Ablauf von Kommandofolgen.
- ◇ Die gängigsten Shells sind :
  - ▷ **bash** : Bourne Again Shell, **Standard-Shell** von LINUX, kompatibel zu der AT&T-Unix-Shell **sh**, erweitert um Merkmale der C-Shell von BSD-Unix (**csch**), ihre Programmiersprache entspricht weitgehend der **sh**-Programmiersprache, entwickelt von der Free Software Foundation.
  - ▷ **tcsh** : Erweiterte Version der C-Shell (**csch**) von BSD-Unix.  
Zu den Erweiterungen gehören u.a. die Ergänzung von Kommandonamen und ein Kommandozeilen-editor. Die shell-interne Programmiersprache baut auf der Programmiersprache der C-Shell auf, die ihrerseits eine große Ähnlichkeit mit der Programmiersprache C besitzt.
  - ▷ **ksh** : Public Domain Version der Korn-Shell von AT&T, Bestandteil von System V Release 4, häufig in derartigen Systemen die Standard-Shell.  
Sie enthält im wesentlichen die Merkmale der **bash** und der **tcsh**.  
Ihre Programmiersprache folgt der **sh**-Syntax.
- ◇ Die beim **Einloggen** eines **Benutzers** zu startende **Shell** ist **individuell** in der Datei **/etc/passwd** festgelegt.

### • Graphische Benutzeroberfläche

- ◇ **X Window System X11** in der frei verfügbaren Portierung auf x86-Prozessoren **XFree86**.
- ◇ X11 ist ein **netzwerktransparentes** bitmaporientiertes Window-System, das aus einem **X-Server**, **X-Clients** (X-Anwendungen) und dem **X-Protokoll** besteht.
  - ▷ **X-Server** : läuft auf dem die graphische Oberfläche darstellenden Rechner, **Ansprechen der Graphikkarte** (Zeichnen einfacher graphischer Elemente, wie Punkte, Linien, Rechtecke, Text), Entgegennahme von Maus- und Tastatureingaben, Kommunikation mit X-Client mittels X-Protokoll
  - ▷ **X-Client** : Anwendung, die graphische Ausgaben erzeugt, Festlegung der darzustellenden graphischen Objekte, Kommunikation mit X-Server über X-Protokoll, ruft Funktionen der **xlib** auf.  
Spezieller X-Client mit besonderen Rechten : **X-Window-Manager**, dieser steuert das **äußere Erscheinungsbild des Desktops**, er kann andere X-Clients starten und beenden, er kommuniziert sowohl mit dem X-Server (über das X-Protokoll) als auch mit anderen X-Clients (über das **ICCC**-Protokoll, **ICCC** : *Inter-Client Communication Conventions*).  
Für LINUX wurden zahlreiche verschiedene Window Manager entwickelt, z.B. **fvwm**, **olwm**, **icwm**, **kwm** (Window Manager von **KDE**), **Metacity** (Window Manager von **GNOME**) (Überblick siehe [www.xwinman.org](http://www.xwinman.org))
  - ▷ **X-Protokoll** : Kommunikations-Protokoll zwischen X-Client und X-Server,  
Die Kommunikation erfolgt über Unix Domain Sockets (lokal) bzw TCP/IP (Netzwerk)

## Directory-Struktur des Kernel-Quellcodes (1)

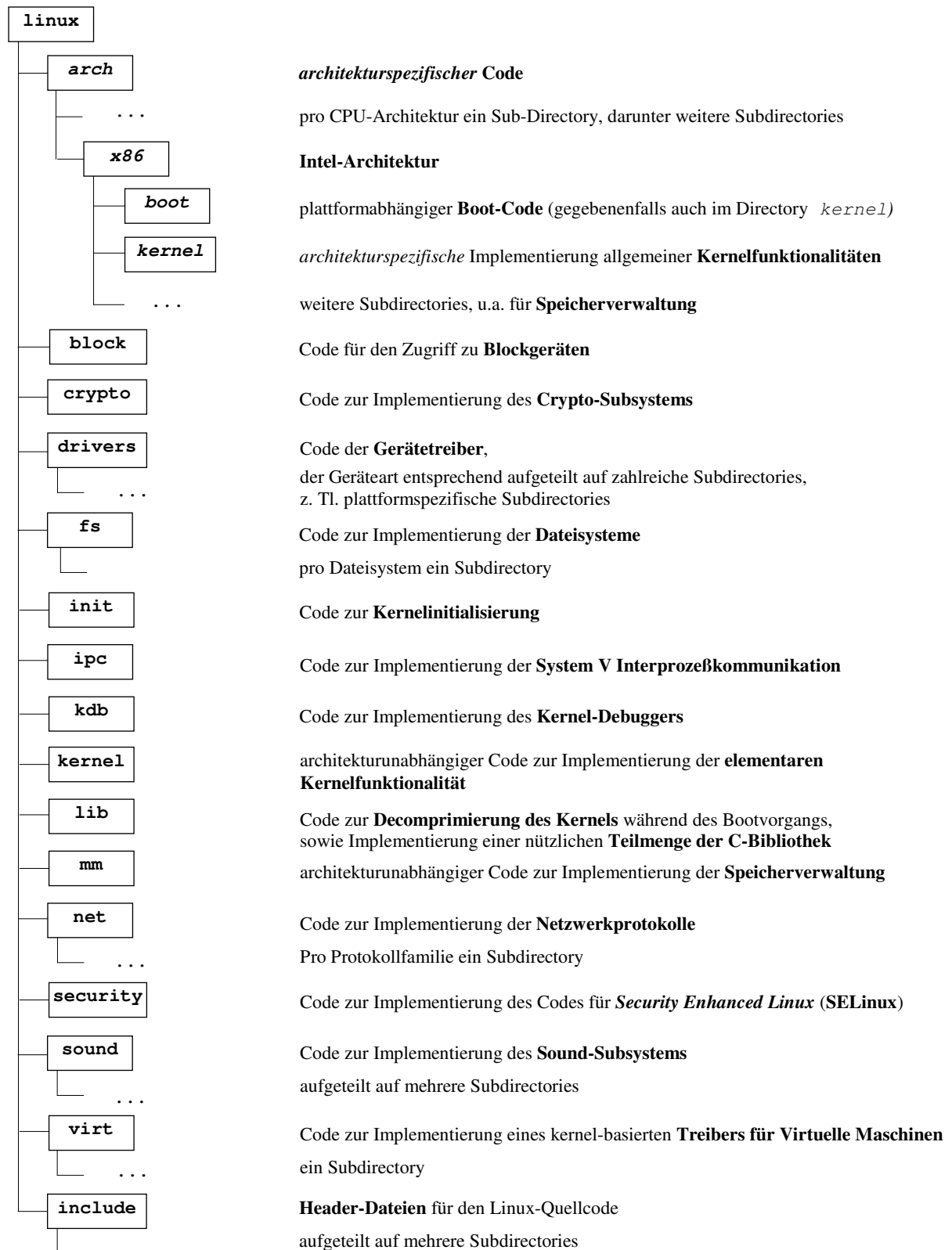
### • Aufteilung des Quellcodes (Kernel 2.6.25)

Der Kernel-Quellcode ist auf zahlreiche Dateien aufgeteilt.

Jede Datei realisiert typischerweise eine bestimmte Funktionalität des Kernels.

Funktionell bzw logisch zusammengehörige Dateien sind in Directories zusammengefaßt.

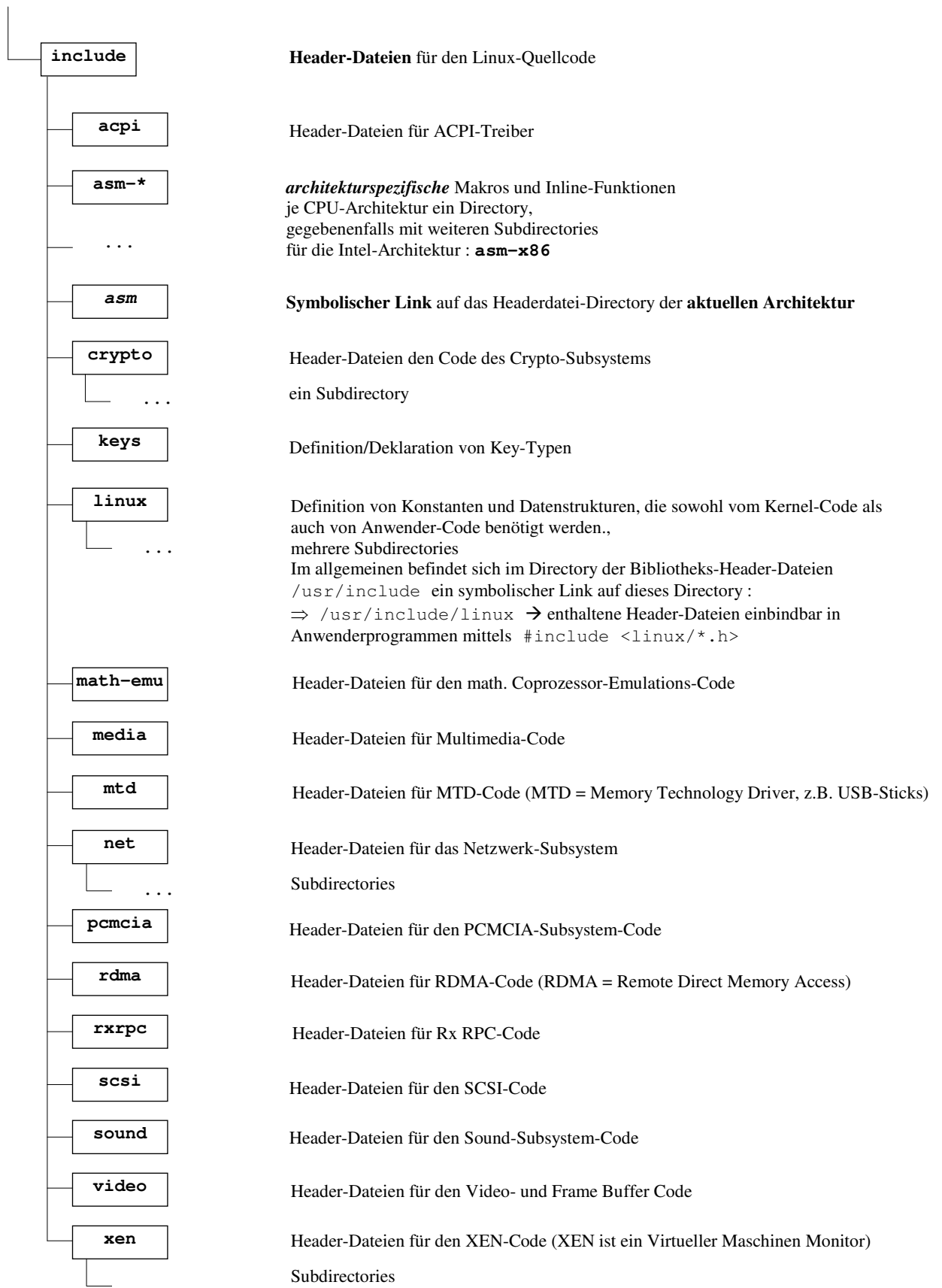
Der **Linux-Quellcode-Directory-Baum** beginnt üblicherweise im Directory **/usr/src/linux**





## Directory-Struktur des Kernel-Quellcodes (2)

### • Strukturierung der Header-Dateien



## Bootloader unter LINUX

### • Allgemeines

- ◇ Der **ausführbare Code** des Betriebssystems LINUX (genauer der **Code des LINUX-Kernels**) befindet sich i.a. in einer **Datei** (häufiger Name : **vmLinux**) auf einem **logischen Laufwerk** (der **Boot-Partition**).  
Zum Laden dieser Kernel-Datei wird üblicherweise ein spezielles Programm, ein sogenannter **Bootloader**, eingesetzt.
- ◇ Z.Zt. kommen zwei verschiedene Bootloader unter LINUX zum Einsatz :
  - ▷ **LILO** (*Linux Loader*)
  - ▷ **GRUB** (*Grand Unified Bootloader*)

### • Gemeinsame Eigenschaften

- ◇ Beide Bootloader vereinigen die Funktionalitäten
  - ▷ eines **Boot-Programms**
  - ▷ und eines **Boot-Managers**Beide ermöglichen somit
  - ▷ sowohl das **Laden des LINUX-Kernels** ( → Booten von LINUX)
  - ▷ als auch das **Laden des Bootsektors anderer Partitionen.** ( → Booten anderer Betriebssysteme)Die zur Verfügung stehenden bootbaren Betriebssysteme werden in einem **Menue** angeboten.  
Aus diesem kann das zu bootende Betriebssystem **ausgewählt** werden.  
Für das Ausbleiben einer expliziten Auswahl kann ein **Default-System** festgelegt werden.
- ◇ Beide Bootloader werten **keine Partitionstabelle** aus.  
Sie können ein Betriebssystem auch **von logischen Laufwerken in Extended Partitionen** und von der **2. Festplatte** booten
- ◇ Einem zu bootenden **Linux-Kernel** können **Boot-Parameter** mitgegeben werden.  
Diese können durch eine Konfigurationsdatei festgelegt sein.  
Sie lassen sich am **Boot-Prompt** ändern bzw ergänzen.
- ◇ Beide Bootloader bestehen jeweils aus **zwei Stufen** :
  - ▷ die **erste Stufe** (*stage 1, primary bootloader*).  
Sie wird im **Master Boot Block (MBR)** oder im **Bootsektor** einer **anderen – primären – Partition** (die auch eine Extended Partition sein darf) installiert.  
(→ "**Installation**" des Bootloaders).  
Ihre Aufgabe ist das Laden der zweiten Stufe.
  - ▷ die **zweite Stufe** (*stage 2, secondary bootloader*).  
Sie befindet sich in einer **eigenen Datei** in der Boot-Partition und enthält den eigentlichen Bootloader-Code, mit dem dann der LINUX-Kernel bzw der Bootsektor anderer Betriebssysteme geladen wird.  
Standardmässig lautet der Zugriffspfad zu dieser Datei :
    - **/boot/boot.b** beim **LILO**
    - **/boot/grub/stage2** beim **GRUB**

### • Wesentliche Unterschiede

- ◇ **LILO** kann nicht direkt zu Dateien zugreifen. Er benötigt eine **Map-Datei**, in der die **physikalischen Adressen** der einzelnen Sektoren der verwendeten Dateien eingetragen sind.  
Die Map-Datei wird bei der Installation von LILO erzeugt.  
Bei **Änderung** des **Orts** oder des **Inhalts** einer **verwendeten Datei** bzw der **Konfigurationsdatei** muß **LILO neu installiert** werden.
- ◇ **GRUB** kann **direkt** zu Dateien in ausgewählten Dateisystemen zugreifen.  
Eine **Änderung** des **Inhalts** der **Haupt-Konfigurationsdatei** ist jederzeit möglich, **ohne** dass GRUB **neu installiert** werden muß.  
Ausserdem kann GRUB auch **Betriebssysteme booten**, die **nicht im Menue** **enthalten** sind.

## Der LINUX-Bootloader LILO

### • Besonderheiten von LILO

- ◇ LILO greift nur mittels der im ROM-BIOS enthaltenen INT13H-Funktionen zu physikalischen Laufwerken zu.  
Die **physikalischen Adressen** der einzelnen Sektoren der zu ladenden Dateien entnimmt es einer eigenen **Map-Datei**. Diese wird bei der Installation von LILO erzeugt.
- ◇ Sämtliche **Bestandteile des LILO-Bootsystems** sowie die von LILO **zu ladenden Sektoren** müssen gegebenenfalls – abhängig vom ROM-BIOS – **innerhalb** der ersten **1024 Zylinder** der Festplatte (d.h. innerhalb der ersten **8 GBytes**) liegen

### • Bestandteile des LILO-Bootsystems

- ▷ **LILO-Bootsektor** (→ MBR oder Bootsektor einer Partition), "*primary bootloader*"
- ▷ **Datei mit 2. Teil des LILO-Codes** (→ `/boot/boot.b`), "*secondary bootloader*"
- ▷ **Map-Datei** mit den **physikalischen Adressen** der zu ladenden Sektoren und den übrigen von LILO benötigten Dateien, → `/boot/map`
- ▷ optional : **Datei** mit von LILO **auszugebenden Text** (z.B. Bootmenu), → `/boot/<dateiname>` (Message-Datei)

### • Konfiguration und Installation von LILO

- ◇ LILO kann sehr **flexibel** an die jeweiligen speziellen Gegebenheiten und **Bedürfnisse** **angepaßt** werden
- ◇ Die Anpassung erfolgt mittels einer **Konfigurationsdatei** (Textdatei) → `/etc/lilo.conf`  
Diese enthält
  - ▷ einen **globalen Abschnitt** (u.a. Angabe des Installationsortes : Parameter `boot=<bootdevice>`, Zugriffspfad der Message-Datei : Parameter `message=<datpfad>`, Boot-Parameter für den Kernel)
  - ▷ einen **speziellen Abschnitt** für jedes zu bootende Betriebssystem
- ◇ **Beispiel :**

```
# LILO Konfigurations-Datei
#
# Start LILO global Section
boot=/dev/hda3           # LILO-Bootsektor in Extended Partition
message=/boot/message
lba32
prompt                   # Erzeugung eines Boot-Prompts
timeout=100              # nach 100*100 ms = 10 s wird das erste BS gebootet
#
# WINDOWS bootable section
other = /dev/hda1
label = Windows
loader=/boot/chain.b
table = /dev/hda
#
# LINUX Kernel 2.4 bootable section
image = /boot/vmlinuz
label = Linux
root = /dev/hda7
initrd = /boot/initrd
append = "disableapic vga=0x0317"
```

- ◇ Die **Installation** erfolgt mittels des **Kommandos** `/sbin/lilo`  
Dieses **wertet** die **Konfigurationsdatei** **aus** und bewirkt :
  - ▷ Anlegen eines **Backups** des bisherigen Inhalts des Ziel- (Boot-) Sektors
  - ▷ **Schreiben** des **LILO-Bootsektors** an den vorgesehenen Ort
  - ▷ Erzeugen einer neuen **Map-Datei**

## Der LINUX-Bootloader GRUB

### • Besonderheiten von GRUB

- ◇ Die zweite Stufe von GRUB enthält *File System Code* für ausgewählte Dateisysteme (derzeit Ext2, Ext3, Reiser, JFS, XFS, Minix, VFAT). Dadurch kann GRUB direkt – über den Zugriffspfad – zu Dateien in diesen Dateisystemen zugreifen. Insbesondere wird die **Haupt-Konfigurationsdatei** bei jedem Bootvorgang neu **eingelese**n. Bei Änderungen ihres Inhalts muss GRUB daher nicht neu installiert werden.
- ◇ GRUB verwendet **eigene Bezeichnungen** für **Laufwerke** und **Partitionen**. Diese weichen von den von LINUX benutzten Gerätepfadnamen ab :
  - Verwendung einer anderen Notation
  - Beginn der Partitionszählung bei 0 statt bei 1,
  - keine Unterscheidung zwischen IDE- und SCSI-Festplatten.

**Beispiel :** **(hd0, 0)** erste primäre Partition der ersten Festplatte (entspricht **/dev/hda1** oder **/dev/sda1**).

Die Zuordnung zwischen den GRUB-Laufwerksbezeichnungen und den LINUX-Gerätepfadnamen speichert GRUB in einer Textdatei, der Map-Datei **/boot/grub/device.map**, ab.

Diese Datei kann gegebenenfalls **manuell geändert** und **angepasst** werden. In einem derartigen Fall muß **GRUB neu installiert** werden.

### • Bestandteile des GRUB-Bootsystems

- ▷ **GRUB-Bootsektor** (→ MBR oder Bootsektor einer Partition), "*primary bootloader*", (Kopie von **/boot/grub/stage1**)
- ▷ **Datei mit 2. Teil des GRUB-Codes** (→ **/boot/grub/stage2**), "*secondary bootloader*"
- ▷ **Map-Datei** **/boot/grub/device.map** (Zuordnung GRUB-Laufwerksbezeichnungen – LINUX-Gerätepfade)
- ▷ **Haupt-Konfigurationsdatei** mit den Informationen für das **Boot-Menue** (**/boot/grub/menu.lst**)
- ▷ **Konfigurationsdatei** für die Installation von GRUB (**/etc/grub.conf**)
- ▷ Datei mit dem **Hintergrundbild** des **Boot-Menues** (z.B. **/boot/message**).

### • Konfiguration und Installation von GRUB

- ◇ **Konfiguration des Menues** : Datei **/boot/grub/menu.lst** (wird bei jedem Bootvorgang neu eingelesen)

**Beispiel :**

```
color white/blue black/light-gray
default 0
timeout 8
gfxmenu (hd1,0)/message

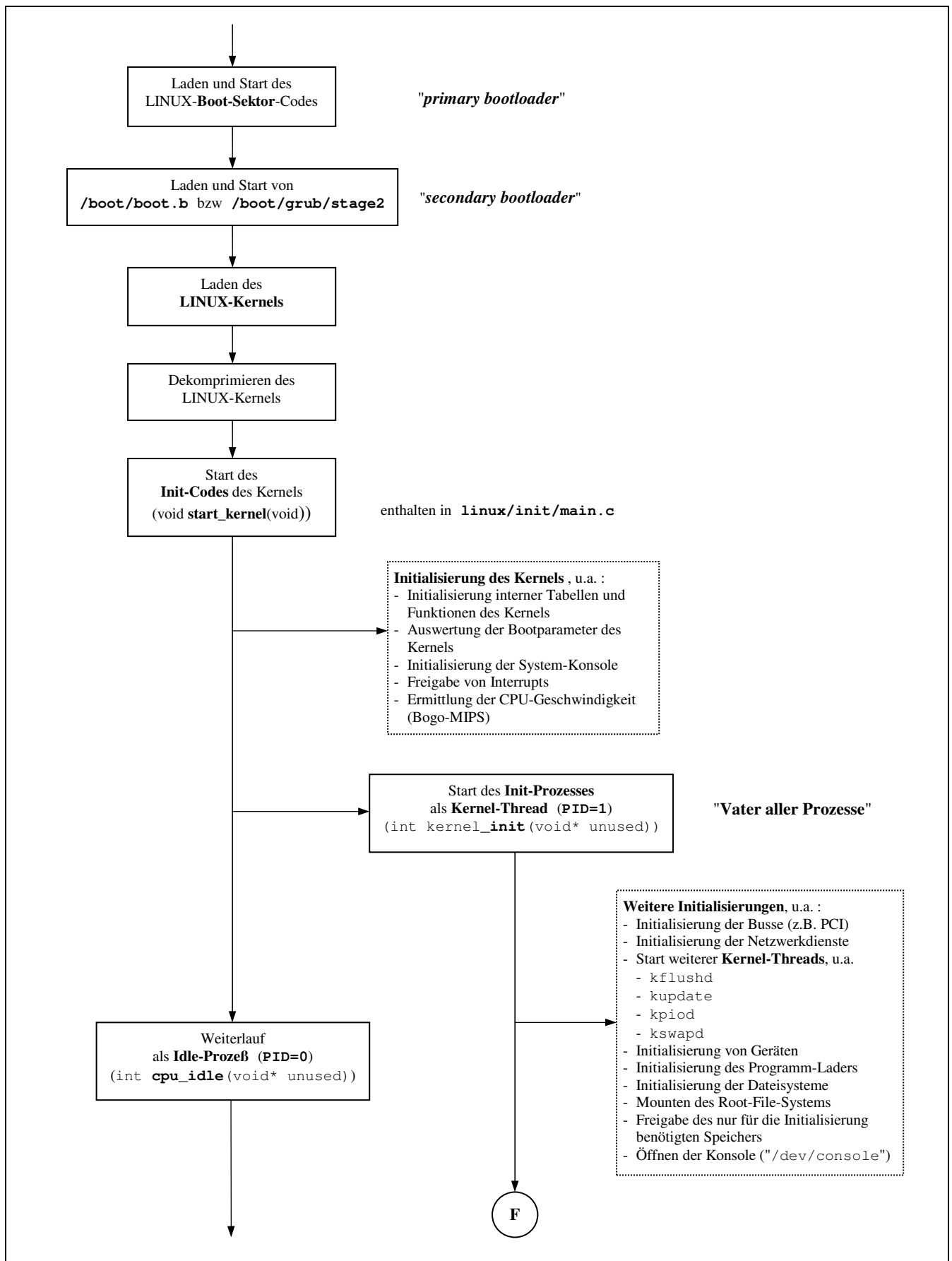
title Windows
    root (hd0,0)
    chainloader +1

title SUSE LINUX 9.2
    kernel (hd1,0)/vmlinuz root=/dev/hdb5 vga=0x31a selinux=0
    initrd (hd1,0)/initrd

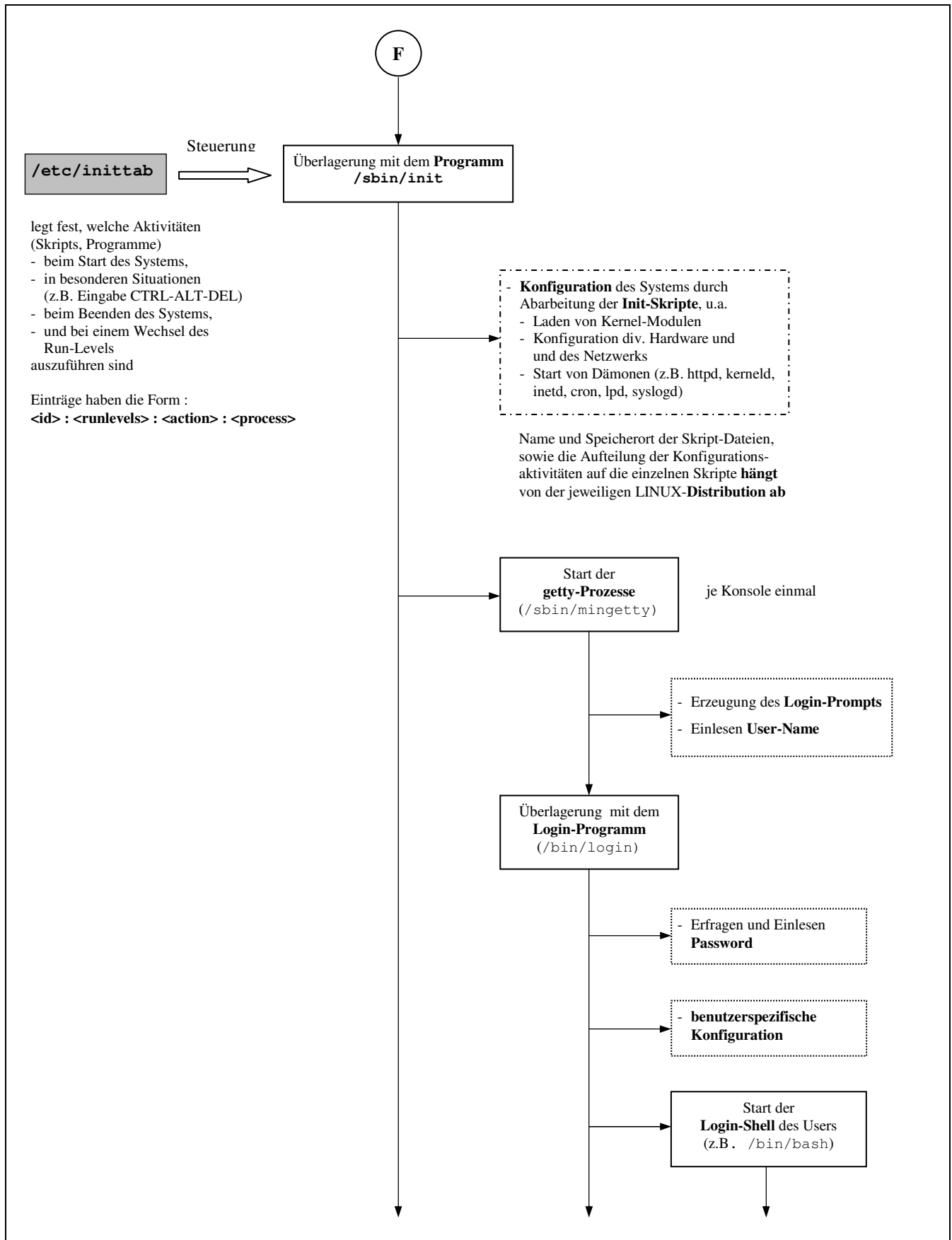
title Speichertest
    kernel (hd1,0)/memtest.bin
```

- ◇ **Konfiguration der Installation** : Datei **/etc/grub.conf**  
Festlegung des Zugriffspfad der beiden Bootloader-Dateien und der Menue-Konfigurationsdatei sowie des Speicherorts des GRUB-Bootsektors
- ◇ Die **Installation** von GRUB erfolgt mittels des Programms **/usr/sbin/grub** → **GRUB-Shell**.
  - ▷ Entweder Verwendung der Kommandos **install** oder **setup**
  - ▷ Oder Verwendung im Batch-Modus (Aufruf mit Option **--batch**)  
**Beispiel :** **grub --batch --device-map=/boot/grub/device.map < /etc/grub.conf**

## Ablauf des Bootvorgangs von LINUX (1)



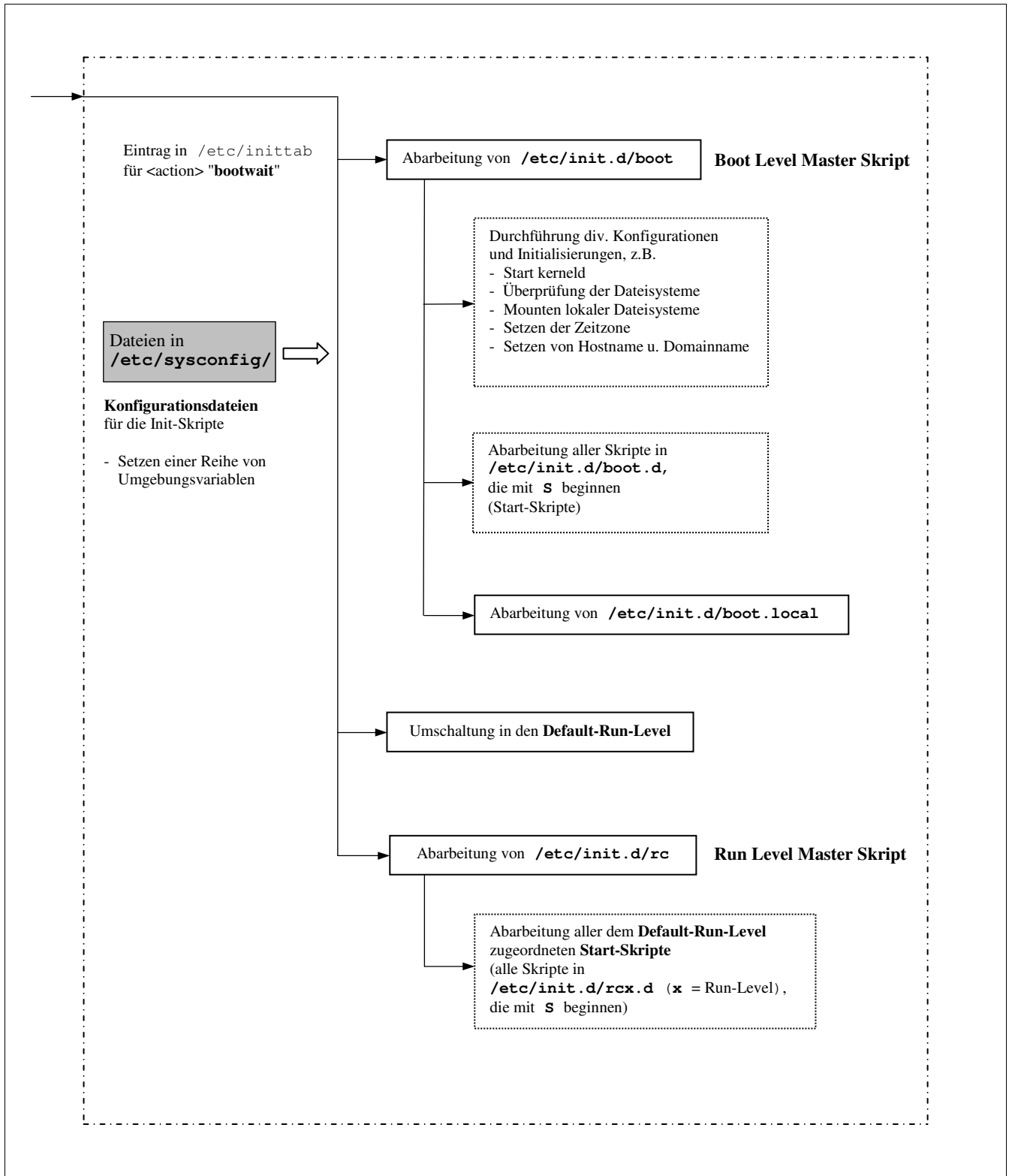
## Ablauf des Bootvorgangs von LINUX (2)



### Ablauf des Bootvorgangs von LINUX (3)

#### • Abarbeitung der Boot-Init-Skripte bei SUSE-LINUX

Bei Suse-Linux befinden sich sämtliche Init-Skripte im Verzeichnis **/etc/init.d** (sowie in den darunterliegenden Verzeichnissen) (entspricht der Festlegung für die Systeminitialisierung im Entwurf des **LINUX Standard Base (LSB)**)



## Runlevel unter LINUX

### • Zweck

Zu jedem Zeitpunkt befindet sich LINUX in einem sogenannten **Runlevel** ("Dienstleistungsstufe"). Der Runlevel definiert den jeweiligen **prinzipiellen Betriebszustand** des Systems. Er legt fest, welche **Grund-Funktionalitäten** das System aktuell aufweist und damit, welche Prozesse überhaupt grundsätzlich existieren können.

Die Datei **/etc/inittab** legt – direkt und indirekt - fest, welche Dienste ("Dienstleistungsprozesse", meist Dämonen) beim Eintritt in welchen Runlevel tatsächlich zu starten sind.

Diese Datei legt auch den **Default-Runlevel** fest, in den das System durch das Booten gelangen soll (`initdefault`, i.a. **3** oder **5**).

Alternativ kann der gewünschte Runlevel beim Booten (z.B. am Boot-Prompt) angegeben werden.

### • Definierte Runlevel

Runlevel	Bedeutung
0	System-Halt
1	Single-User-Mode (Wartungsmodus)
2	Multi-User-Mode ohne Netzwerk (lokaler Mehrbenutzerbetrieb)
3	Multi-User-Mode mit Netzwerk (voller Mehrbenutzerbetrieb)
4	frei
5	Multi-User-Mode mit Netzwerk und X11 (Einloggen über <code>xdm</code> )
6	Reboot
S	Single-User-Mode (Wartungsmodus); vom Bootprompt aus mit US-Tastaturbelegung

### • Änderung des Runlevels

◇ Ein Wechsel des Runlevels während des Betriebs ist - nur durch den User **root** – möglich mit den Kommandos

**init <new\_level>**                      oder                      **telinit <new\_level>**

#### ◇ Ablauf des Wechsels :

- `init` entnimmt der Datei `/etc/inittab`, welche Programme bzw Skripte für den neuen Runlevel auszuführen sind. I.a. wird `/etc/init.d/rc` (*Run Level Master Skript*) mit dem **neuen Runlevel** als **Parameter** zu starten sein.
- Das Skript `rc` ruft alle **Stop-Skripte** (Beginn mit `'K'`) des **alten Runlevels** auf, für die im neuen Runlevel kein Start-Skript existiert (unter Beachtung einer im Namen festgelegten Reihenfolge-Nummer).
- Anschließend ruft `rc` alle **Start-Skripte** (Beginn mit `'S'`) des **neuen Runlevels** auf, für die im alten Zustand kein Stop-Skript existiert (ebenfalls unter Beachtung der im Namen festgelegten Reihenfolge-Nummer).

**Anmerkung :** Start- und Stop-Skripte sind symbolische **Links** auf die **eigentlichen Skripte**, die mit dem Parameter **start** bzw **stop** aufgerufen werden

### • Ermittlung des Runlevels

Der **aktuelle** und der **vorherige Runlevel** kann ermittelt werden mittels des Kommandos

**/sbin/runlevel**

Das Kommando wertet die Datei `/var/run/utmp` aus.



# **Betriebssysteme**

## **Kapitel 5**

### **5. Betriebssystemfunktionen (System Calls) von LINUX**

5.1. Implementierung

5.2. Überblick über System Calls und Fehlercodes

5.3. Beispiele für System Calls

## Implementierung von *System Calls* in LINUX (1)

### • Betriebssystemfunktionen (*System Calls*)

Der Kernel stellt Anwenderprogrammen bestimmte **Funktionalitäten** (Dienstleistungen) zur Verfügung.

I.a. greifen diese Funktionalitäten auf vom Kernel verwaltete Ressourcen zu.

Anwenderprogramme können diese Dienste des Kernels über den Aufruf von **Betriebssystemfunktionen** (*System Calls*) in Anspruch nehmen.

Die Gesamtheit der zur Verfügung stehenden Betriebssystemfunktionen wird auch als **API** (*Application Programming Interface*) des Betriebssystems bezeichnet.

### • Kapselung der Betriebssystemfunktionsaufrufe in C-Funktionen

Unter LINUX sind – wie in UNIX-Systemen üblich – die Aufrufe der Betriebssystemfunktionen in **C-Funktionen gekapselt**. Für jede Betriebssystemfunktion steht – mindestens - eine direkte oder indirekte **Wrapper-Funktion** in der **C-Bibliothek** zur Verfügung.

In einigen Fällen existieren **mehrere C-Bibliotheksfunktionen**, die zum **Aufruf** des **gleichen System Calls** führen (i. a. mit unterschiedlichen Parametern).

Die einzelnen **System Calls** werden mit einem **Namen** bezeichnet, der üblicherweise dem Namen der jeweiligen **kapselnden C-Bibliotheksfunktion** entspricht.

### • Prinzip des Betriebssystemfunktionsaufrufs

Der Aufruf einer Betriebssystemfunktion bedeutet den Aufruf von – **privilegierten – Kernel-Code**.

Dies erfordert einen Übergang vom **User-Mode** (Privileg-Ebene 3 bei x86-Prozessoren) in den **System-Mode** (Ebene 0).

Ein derartiger **Privilegebenenwechsel** wird – u.a. – durch einen **Interrupt** bewirkt.

In LINUX wird hierfür – auf der x86-Architektur – der **Software-Interrupt `int 0x80`** eingesetzt.

Dieser wird von der jeweiligen **Wrapper-Funktion** der C-Bibliothek aufgerufen.

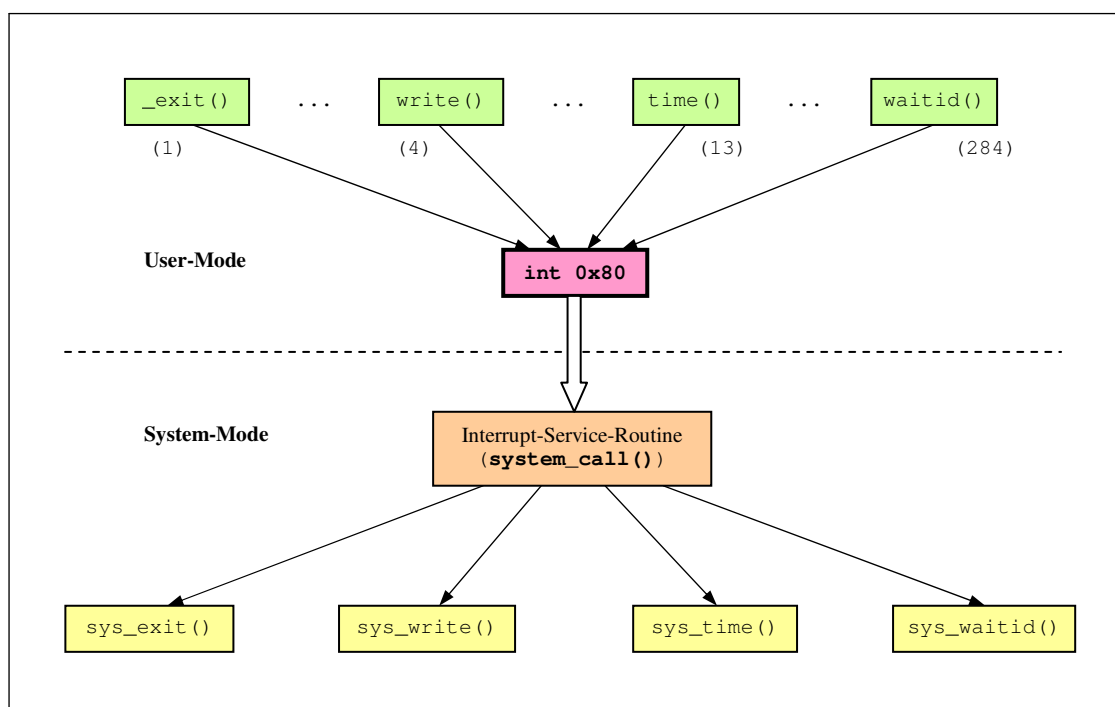
Die verschiedenen Betriebssystemfunktionen werden durch ihnen zugeordnete **Funktions-Nummern** unterschieden.

Z. Zt. (Kernel 2.6.25) sind – mit einigen nicht mehr zugeordneten Lücken – die Nummern **1** bis **326** belegt.

Die Funktions-Nummern sind definiert in `<asm/unistd.h>`. Diese Headerdatei wird üblicherweise indirekt durch `<linux/unistd.h>` eingebunden.

Die **Funktions-Nummer** muß zusammen mit eventuellen **Parametern** der durch den Interrupt aufgerufenen **Interrupt-Service-Routine** übergeben werden. Die Übergabe erfolgt über CPU-Register.

Die Interrupt-Service-Routine ruft dann die zu der Funktions-Nummer gehörende eigentliche Betriebssystemfunktion über eine **Sprungtabelle** - mit der Funktions-Nummer als Index - auf.



## Implementierung von *System Calls* in LINUX (2)

### • Registerverwendung bei *System Calls*

Die C-Bibliotheksfunktion (*Wrapper-Funktion*) schreibt die **Funktions-Nr** und die der eigentlichen Betriebssystemfunktion zu übergebenen **Parameter** in **Prozessor-Register**. Anschließend ruft sie den **int 0x80** auf.

In der x86-Architektur stehen für die Übergabe insgesamt 6 Register zur Verfügung. Damit können **maximal 5 Parameter** übergeben werden.

Benötigt ein System Call mehr als 5 Parameter, werden diese in einer Structure zusammengefaßt und deren Adresse wird als einziger Parameter übergeben

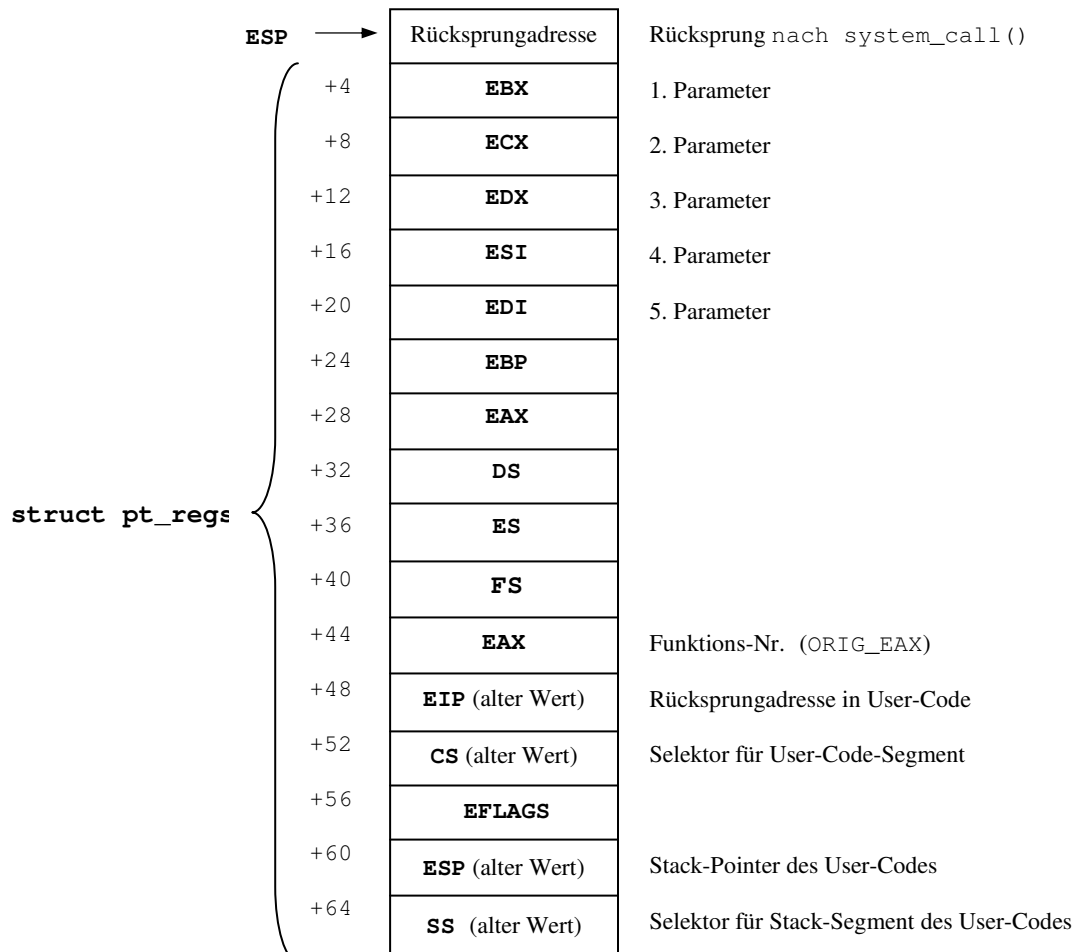
Die Register werden wie folgt verwendet :

<b>EAX</b>	Funktions-Nr
<b>EBX</b>	1. Parameter
<b>ECX</b>	2. Parameter
<b>EDX</b>	3. Parameter
<b>ESI</b>	4. Parameter
<b>EDI</b>	5. Parameter

Durch die Interrupt-Service-Routine (ISR) **system\_call()** werden die Parameter auf dem **Stack** abgelegt und dadurch der von ihr aufgerufenen eigentlichen Betriebssystemfunktion auf dem Stack übergeben.

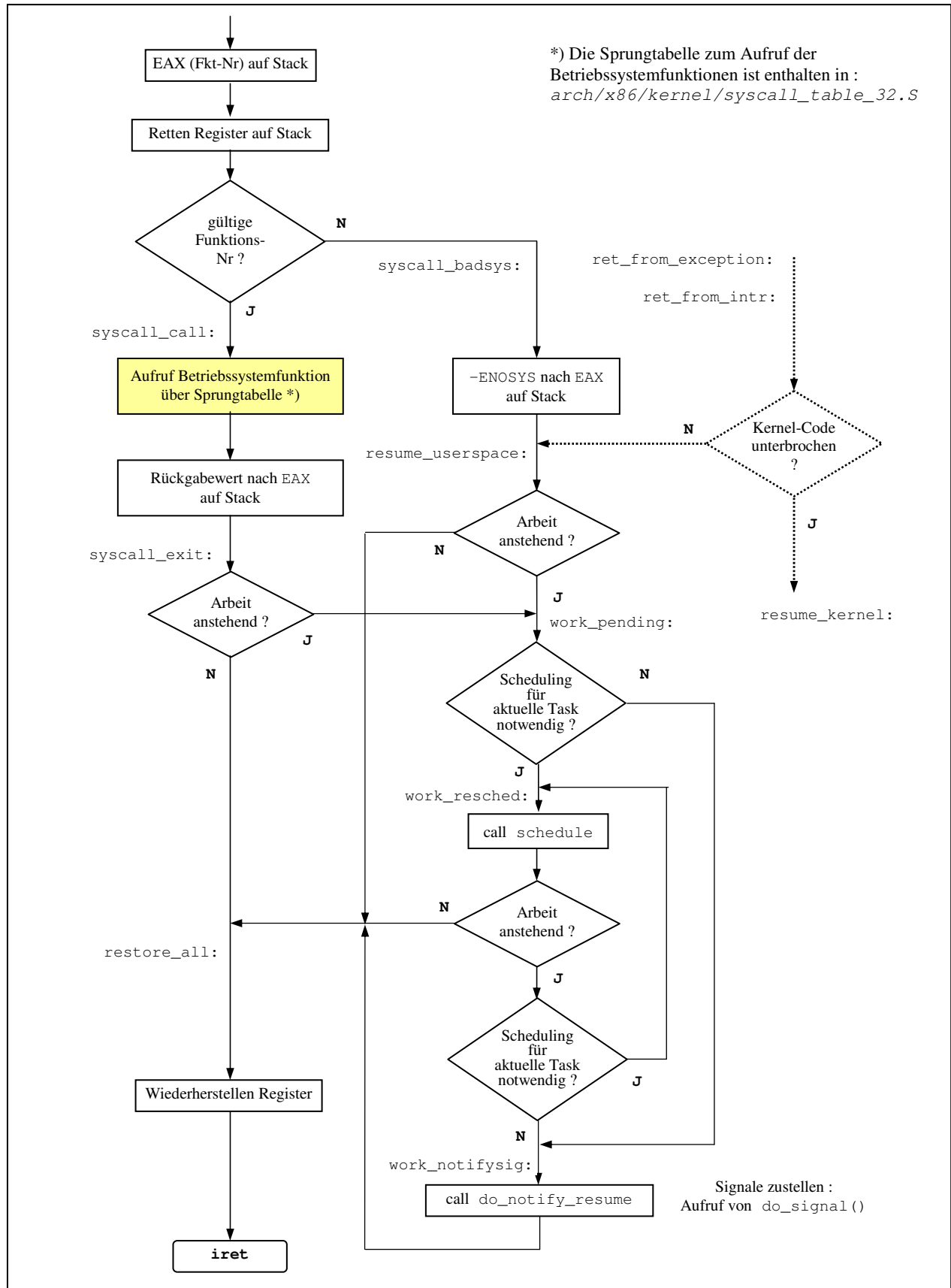
Die Betriebssystemfunktion schreibt ihren **Rückgabewert** in das Abbild des Registers **EAX** auf dem Stack, von wo es die ISR vor dem Rücksprung in den User-Mode in das echte Register **EAX** holt.

### • Stackaufbau in den Betriebssystemfunktionen



### Implementierung von System Calls in LINUX (3)

- **Prinzipielle Arbeitsweise der System-Call-Interrupt-Service-Routine**  
(in Assembler-Code realisiert, enthalten z.B. in `arch/x86/kernel/entry-32.S`)



## Implementierung von *System Calls* in LINUX (4)

### • Rückgabewert von *System Calls*

- ◇ Alle **Betriebssystemfunktionen** erzeugen einen **Rückgabewert** vom Typ **int**.

Dieser Wert ist

- ▷ **0** oder **positiv** oder – bei einigen wenigen Funktionen – **kleiner als -4095** bei **Erfolg**,
- ▷ **negativ** im Bereich **-1 ... -131** im **Fehlerfall**  
(das **Negative** einer den Fehler näher kennzeichnenden **Fehlernummer**)

Er wird von der Interrupt-Service-Routine über das Register **EAX** (x86-Architektur) an die Wrapper-Funktion der C-Bibliothek zurückgegeben.

- ◇ Die **Wrapper-Funktionen** der **C-Bibliothek** setzen den Rückgabewert der Interrupt-Service-Routine wie folgt in den **eigenen Rückgabewert** (ebenfalls von einem ganzzahligen Typ) um :
  - ▷ **0** oder ein **positiver Wert** (bzw ein **negativer Wert <-4095** ) bei **Erfolg**,
  - ▷ **-1** im **Fehlerfall**, wobei die **Fehlernummer** (= der negierte Rückgabewert der Interrupt-Service-Routine) in der globalen Fehlervariablen **errno** abgelegt wird.

### • Prinzipielle Realisierung einer *System-Call-Wrapper-Funktion* in der C-Bibliothek

- ◇ **Beispiel** : System Call `write(...)`

```
ssize_t write(int fd, const void* buf, size_t count)
{
    long __res;
    __asm__("mov eax, 4");           /* Funktions-Nr. 4 --> EAX */
    __asm__("mov ebx, fd");         /* 1. Parameter --> EBX */
    __asm__("mov ecx, buf");        /* 2. Parameter --> ECX */
    __asm__("mov edx, count");      /* 3. Parameter --> EDX */
    __asm__("int 0x80");            /* System Call */
    __asm__("mov __res, eax");      /* Rueckgabewert ist in EAX */

    if ((unsigned long)__res >= (unsigned long)(-131))
    {
        errno=-__res;              /* Setzen von errno */
        __res=-1;
    }
    return __res;
}
```

- ◇ **Anmerkung** :

Für **viele System Calls** werden durch die C-Bibliothek **keine Wrapper-Funktionen** zur Verfügung gestellt. In früheren Versionen der GNU-C-Bibliothek konnten diese aus den sogenannten **\_\_syscall-Makros** generiert werden. Diese Makros (für jede mögliche Parameterzahl (0 ... 5) jeweils einer) waren in der C-Bibliotheks-Header-Datei **<asm/unistd.h>** definiert. In der jetzigen GNU-C-Bibliothek existieren diese Makros nicht mehr.

Alternativ lassen sich **alle Betriebssystemfunktionen** aber indirekt **aufrufen** über die in der Headerdatei **<unistd.h>** deklarierte **generische Wrapper-Funktion**

**int syscall(int sysnr, ...)**

Der Parameter **sysnr** ist die **Funktions-Nummer** des aufzurufenden **System Calls**. Diese wird zweckmässigerweise als symbolische Konstante der Form **\_\_NR\_fktname** (def. in **<asm/unistd.h>**, indirekt eingebunden durch **<linux/unistd.h>**) angegeben.

Die **weiteren Parameter** entsprechen den Parametern, die dem **jeweiligen System Call** zu **übergeben** sind.

Der **Rückgabewert** ist durch den Rückgabewert des aufzurufenden System Calls definiert. Er wird wie bei den direkten **Wrapper-Funktionen** gebildet. Bei **Erfolg** wird i.a. **0** oder ein **positiver Wert** zurückgegeben.

Im Fall eines **Fehlers** wird **-1** zurückgegeben und die Fehlervariable **errno** entsprechend **gesetzt**.

## Überblick über die *System Calls* in LINUX (1)

Fkt-Nr.	Name	Funktionalität
0	<b>restart_syscall</b>	System-Restart
1	<b>_exit</b>	Beendigung des laufenden Prozesses
2	<b>fork</b>	Erzeugen eines Kindprozesses
3	<b>read</b>	Lesen aus einem Datei-Objekt (referiert über File-Deskriptor)
4	<b>write</b>	Schreiben in ein Datei-Objekt (referiert über File-Deskriptor)
5	<b>open</b>	Öffnen einer Datei oder eines Gerätes
6	<b>close</b>	Schließen eines File-Deskriptors
7	<b>waitpid</b>	Warten auf Zustandsänderung (z.B. Beendigung) eines Kindprozesses
8	<b>creat</b>	Erzeugen einer neuen Datei
9	<b>link</b>	Erzeugen eines Hard-Links auf eine existierende Datei
10	<b>unlink</b>	Entfernen eines Dateinamens aus Directory und gegebenenfalls Löschen der Datei
11	<b>execve</b>	Ausführen eines neuen Programms
12	<b>chdir</b>	Änderung des aktuellen (Arbeits-) Directories
13	<b>time</b>	Zeit in Sekunden seit 1.1.1970, 00:00:00 UTC
14	<b>mknod</b>	Erzeugen einer Gerätedatei oder einer Named Pipe
15	<b>chmod</b>	Ändern der Zugriffsberechtigungen für eine Datei
16	<b>lchown</b>	Änderung des Besitzers und der Gruppe einer Datei (sym. Links wird nicht gefolgt)
17	nicht implementiert	(break)
18	<b>oldstat</b>	(veraltet)
19	<b>lseek</b>	Veränderung der Datei-Bearbeitungsposition
20	<b>getpid</b>	Rückgabe der Prozess-ID (PID) des aktuellen Prozesses (ab Kernel 2.4 : TGID)
21	<b>mount</b>	Einhängen (Mounten) eines Dateisystems
22	<b>umount</b>	Aushängen (Unmounten) eines Dateisystems
23	<b>setuid</b>	Setzen der User-ID des aktuellen Prozesses
24	<b>getuid</b>	Rückgabe der realen User-ID des aktuellen Prozesses
25	<b>stime</b>	Setzen der Systemzeit
26	<b>ptrace</b>	Beobachtung und Steuerung eines anderen Prozesses
27	<b>alarm</b>	Setzen eines Timers, nach dessen Ablauf das Signal <b>SIGALARM</b> zugestellt wird
28	<b>oldfstat</b>	(veraltet)
29	<b>pause</b>	Unterbrechung des aktuellen Prozesses bis zur Aktivierung durch ein Signal
30	<b>utime</b>	Änderung der letzten Zugriffs- und der Änderungszeit einer Datei
31	nicht implementiert	(stty)
32	nicht implementiert	(gtty)
33	<b>access</b>	Überprüfung der Zugriffsrechte eines Prozesses zu einer Datei
34	<b>nice</b>	Setzen der Prozeß-Basis-Priorität des aktuellen Prozesses
35	nicht implementiert	(ftime)
36	<b>sync</b>	Herausschreiben aller Datenpuffer des Prozesses, sowie Update der Inodes
37	<b>kill</b>	Senden eines Signals an einen Prozeß
38	<b>rename</b>	Änderung eines Dateinamens
39	<b>mkdir</b>	Erzeugen eines Directories
40	<b>rmdir</b>	Entfernen eines Directories
41	<b>dup</b>	Duplizieren eines File-Deskriptors
42	<b>pipe</b>	Erzeugen einer Pipe
43	<b>times</b>	Ermittlung des Zeitverbrauchs des aktuellen Prozesses
44	nicht implementiert	(prof)
45	<b>brk</b>	Veränderung des Heaps
46	<b>setgid</b>	Setzen der Gruppen-ID des aktuellen Prozesses
47	<b>getgid</b>	Rückgabe der realen Gruppen-ID des aktuellen Prozesses
48	<b>signal</b>	Installieren eines Signal-Handlers
49	<b>geteuid</b>	Rückgabe der effektiven User-ID des aktuellen Prozesses
50	<b>getegid</b>	Rückgabe der effektiven Gruppen-ID des aktuellen Prozesses
51	<b>acct</b>	Ein-/Ausschalten der Zählung beendeter Prozesse
52	<b>umount2</b>	Aushängen (Unmounten) eines Dateisystems
53	nicht implementiert	(lock)
54	<b>ioctl</b>	Manipulation von Geräteparametern
55	<b>fcntl</b>	Manipulation des File-Deskriptors
56	nicht implementiert	(mpx)
57	<b>setpgid</b>	Setzen der Prozeßgruppen-ID

## Überblick über die *System Calls* in LINUX (2)

Fkt-Nr.	Name	Funktionalität
58	nicht implementiert	(ulimit)
59	oldolduname	(veraltet)
60	umask	Setzen der Dateikreierungsmaske (Dateizugriffsrechte-Maske) eines Prozesses
61	chroot	Ändern des Root-Directories
62	ustat	Ermittlung von Informationen über ein (gemountetes) Dateisystem
63	dup2	Duplizieren eines File-Deskriptors
64	getppid	Rückgabe der Prozeß-ID des Elternprozesses vom aktuellen Prozeß
65	getpgrp	Rückgabe der Prozeßgruppen-ID des aktuellen Prozesses
66	setsid	Erzeugung einer neuen Session und Setzen der Prozeßgruppen-ID
67	sigaction	Registrierung eines prozeßeigenen Signalhandlers
68	sgetmask	
69	ssetmask	
70	setreuid	Setzen der realen und der effektiven User-ID des aktuellen Prozesses
71	setregid	Setzen der realen und der effektiven Gruppen-ID des aktuellen Prozesses
72	sigsuspend	Unterbrechung des aktuellen Prozesses bis zur Aktivierung durch ein Signal
73	sigpending	Ermittlung der aktuell anstehenden aber blockierten Signale
74	sethostname	Setzen des Host-(Rechner-)Namens
75	setrlimit	
76	getrlimit	
77	getrusage	
78	gettimeofday	
79	settimeofday	
80	getgroups	Rückgabe der zusätzlichen Gruppen-IDs des aktuellen Prozesses
81	setgroups	Setzen der zusätzlichen Gruppen-IDs des aktuellen Prozesses
82	select	Warten auf Statusänderungen von Datei-Objekten
83	symlink	Erzeugen eines symbolischen Links auf eine existierende Datei
84	oldlstat	(veraltet)
84	readlink	Einlesen eines symbolischen Links (Pfad auf den Link zeigt)
86	uselib	Laden einer Shared Libaray
87	swapon	Anmeldung eines Auslagerungsbereichs beim Kernel
88	reboot	Reboot des Systems bzw Enable/Disable der Wirkung von CTRL-ALT-DEL
89	readdir	Einlesen eines Directory-Eintrags (veraltet, ersetzt durch <code>getdents</code> , Nr. 141)
90	mmap	Allokation eines virtuellen Speicherbereichs (mit mögl. Einblenden einer Datei)
91	munmap	Freigabe eines virtuellen Speicherbereichs
92	truncate	Ändern der Dateigröße
93	ftruncate	Ändern der Dateigröße (Datei referiert über File-Deskriptor)
94	fchmod	Ändern der Zugriffsberechtigungen einer Datei (referiert über File-Deskriptor)
95	fchown	Ändern des Besitzers und der Gruppe einer Datei (referiert über File-Deskriptor)
96	getpriority	Rückgabe der – höchsten – Prozeß-Basis-Priorität eines oder mehrerer Prozesse
97	setpriority	Setzen der Prozeß-Basis-Priorität eines oder mehrerer Prozesse
98	nicht implementiert	(profil)
99	statfs	Ermittlung von Informationen über ein gemountetes Dateisystem
100	fstatfs	Ermittlung von Informationen über ein gemountetes Dateisystem
101	ioperm	Setzen der prozessspezifischen Zugriffsberechtigung für I/O-Ports (speziell für i386)
102	socketcall	Socket-API (mehrere Unterfunktionen)
103	syslog	
104	setitimer	Setzen eines Intervall-Timers
105	getitimer	Ermittlung der aktuellen Werte eines Intervall-iTmers
106	stat	Ermittlung von Inode-Informationen (über Dateipfad, symLinks wird gefolgt)
107	lstat	Ermittlung von Inode-Informationen (über Dateipfad, Links wird nicht gefolgt)
108	fstat	Ermittlung von Inode-Informationen (über File-Deskriptor)
109	olduname	(veraltet)
110	iopl	Setzen des IO-Privilege Levels für den akt. Prozess (speziell für i386)
111	vhangup	
112	nicht implementiert	(idle)
113	vm86old	Umschaltung in den virtuellen 8086 Modus, alte Version (nur für i386)
114	wait4	Warten auf Zustandsänderung (z.B. Beendigung) eines Kindprozesses
115	swapoff	Abmeldung eines Auslagerungsbereichs beim Kernel

## Überblick über die *System Calls* in LINUX (3)

Fkt-Nr.	Name	Funktionalität
116	<b>sysinfo</b>	Ermittlung von Maschineninformationen
117	<b>ipc</b>	System V IPC -API
118	<b>fsync</b>	Herausschreiben aller Daten-Buffer einer Datei (einschließlich Update Inode)
119	<b>sigreturn</b>	Rückkehr von der Ausführung eines Signal-Handlers (nur vom Kernel zu verw.)
120	<b>clone</b>	Erzeugung eines Kindprozesses mit Teilung des Ausführungskontextes (Thread)
121	<b>setdomainname</b>	Setzen des Domain-Namens des Rechners
122	<b>uname</b>	Ermittlung des Namens des und weiterer Informationen über das Betriebssystem
123	<b>modify_ldt</b>	
124	<b>adjtimex</b>	
125	<b>mprotect</b>	
126	<b>sigprocmask</b>	Veränderung der Signalmaske (Liste der aktuell blockierten Signale)
127	nicht implementiert	( <i>create_module</i> )
128	<b>init_module</b>	Laden und Initialisierung eines ladbaren Kernel-Moduls
129	<b>delete_module</b>	Entfernen (Entladen) eines nicht verwendeten Kernel-Moduls
130	nicht implementiert	( <i>get_kernel_syms</i> )
131	<b>quotactl</b>	
132	<b>getpgid</b>	Rückgabe der Prozeßgruppen-ID eines Prozesses
133	<b>fchdir</b>	Änderung des aktuellen (Arbeits-) Directories (referiert über File-Deskriptor)
134	<b>bdflush</b>	Start des <i>bdflush</i> -Dämons sowie Kommunikation mit diesem
135	<b>sysfs</b>	Ermittlung von Dateisystem-Informationen
136	<b>personality</b>	
137	nicht implementiert	( <i>afs_syscall</i> ) Syscall for Andrew File System
138	<b>setfsuid</b>	Setzen der Dateisystem-User-ID des aktuellen Prozesses
139	<b>setfsgid</b>	Setzen der Dateisystem-Gruppen-ID des aktuellen Prozesses
140	<b>_llseek</b>	Veränderung der Datei-Bearbeitungsposition (64 Bits File-Offset)
141	<b>getdents</b>	Einlesen von Directory-Einträgen (ersetzt <i>readdir</i> , Fkt-Nr. 89)
142	<b>_newselect</b>	
143	<b>flock</b>	Setzen / Entfernen einer Dateisperre
144	<b>msync</b>	Ableich einer Datei mit ihrer Speichereinblendung
145	<b>readv</b>	Lesen eines Vektors aus einem Datei-Objekt
146	<b>writev</b>	Schreiben eines Vektors in ein Datei-Objekt
147	<b>getsid</b>	Rückgabe der Session-ID
148	<b>fdatasync</b>	Herausschreiben aller Daten-Buffer einer Datei (ohne Update Inode)
149	<b>_sysctl</b>	Lesen und Schreiben von Kernel-Parametern
150	<b>mlock</b>	Sperren von Prozess-Speicher-Bereichen für Swapping
151	<b>munlock</b>	Freigabe von Prozess-Speicher-Bereichen für Swapping
152	<b>mlockall</b>	Sperren sämtlichen Prozess-Speichers für Swapping
153	<b>munlockall</b>	Freigabe sämtlichen Prozess-Speichers für Swapping
154	<b>sched_setparam</b>	Setzen der statischen (Realtime-) Priorität eines Prozesses
155	<b>sched_getparam</b>	Rückgabe der statischen (Realtime-) Priorität eines Prozesses
156	<b>sched_setscheduler</b>	Setzen der Scheduling-Policy u. der statischen Priorität eines Prozesses
157	<b>sched_getscheduler</b>	Rückgabe der statischen Priorität eines Prozesses
158	<b>sched_yield</b>	freiwillige Freigabe der CPU durch einen Prozeß
159	<b>sched_get_priority_max</b>	Rückgabe des max. zulässigen Wertes für die statische Priorität
160	<b>sched_get_priority_min</b>	Rückgabe de min. zulässigen Wertes für die statische Priorität
161	<b>sched_rr_get_interval</b>	Rückgabe der Zeitscheibe für Prozesse mit Policy <i>SCHED_RR</i>
162	<b>nanosleep</b>	
163	<b>mremap</b>	Größenänderung und/oder Verschiebung eines virtuellen Speicherbereichs
164	<b>setresuid</b>	
165	<b>getresuid</b>	
166	<b>vm86</b>	Umschaltung in den virtuellen 8086 Modus, neue Version (nur für i386)
167	nicht implementiert	( <i>query_module</i> )
168	<b>poll</b>	Warten auf ein Ereignis bei Datei-Objekten
169	<b>nfsservctl</b>	
170	<b>setresgid</b>	
171	<b>getresgid</b>	
172	<b>prctl</b>	
173	<b>rt_sigreturn</b>	



## Überblick über die *System Calls* in LINUX (4)

Fkt-Nr.	Name	Funktionalität
174	<b>rt_sigaction</b>	
175	<b>rt_sigprocmask</b>	
176	<b>rt_sigpending</b>	
177	<b>rt_sigtimedwait</b>	
178	<b>rt_sigqueueinfo</b>	
179	<b>rt_sigsuspend</b>	
180	<b>pread</b>	Lesen aus Datei-Objekten ab einer bestimmten Position
181	<b>pwrite</b>	Schreiben in Datei-Objekte ab einer bestimmten Position
182	<b>chown</b>	Änderung des Besitzers und der Gruppe einer Datei (symLinks wird gefolgt)
183	<b>getcwd</b>	Ermittlung des aktuellen (Arbeits-) Directories
184	<b>capget</b>	
185	<b>capset</b>	
186	<b>sigaltstack</b>	
187	<b>sendfile</b>	Kopieren von Daten zwischen Datei-Objekten
188	nicht implementiert	(getpmsg) streams1
189	nicht implementiert	(putpmsg) streams2
190	<b>vfork</b>	Erzeugen eines Kindprozesses
ab Kernel 2.4 :		
191	<b>ugetrlimit</b>	
192	<b>mmap2</b>	
193	<b>truncate64</b>	
194	<b>ftruncate64</b>	
195	<b>stat64</b>	
196	<b>lstat64</b>	
197	<b>fstat64</b>	
198	<b>lchown32</b>	
199	<b>getuid32</b>	
200	<b>getgid32</b>	
201	<b>geteuid32</b>	
202	<b>getegid32</b>	
203	<b>setreuid32</b>	
204	<b>setregid32</b>	
205	<b>getgroups32</b>	
206	<b>setgroups32</b>	
207	<b>fchown32</b>	
208	<b>setresuid32</b>	
209	<b>getresuid32</b>	
210	<b>setresgid32</b>	
211	<b>getresgid32</b>	
212	<b>chown32</b>	
213	<b>setuid32</b>	
214	<b>setgid32</b>	
215	<b>setfsuid32</b>	
216	<b>setfsgid32</b>	
217	<b>pivot_root</b>	
218	<b>mincore</b>	
219	<b>madvise</b>	
220	<b>getdents64</b>	
221	<b>fcntl64</b>	
222	nicht implementiert	(reserviert für TUX )
223	nicht implementiert	(reserviert für Security)
224	<b>gettid</b>	Rückgabe der Thread-ID des aktuellen Prozesses
225	<b>readahead</b>	
ab Kernel 2.6 :		
226	<b>setxattr</b>	
227	<b>lsetxattr</b>	
228	<b>fsetxattr</b>	
229	<b>getxattr</b>	
230	<b>lgetxattr</b>	
231	<b>fgetxattr</b>	

## Überblick über die *System Calls* in LINUX (5)

Fkt-Nr.	Name	Funktionalität
232	listxattr	
233	llistxattr	
234	flistxattr	
235	removexattr	
236	lremovexattr	
237	fremovexattr	
238	tkill	Senden eines Signals an einen Thread (Verwendung der TID)
239	sendfile64	
240	futex	
241	sched_setaffinity	
242	sched_getaffinity	
243	set_thread_area	
244	get_thread_area	
245	io_setup	
246	io_destroy	
247	io_getevents	
248	io_submit	
249	io_cancel	
250	fadvise64	
251	nicht implementiert	
252	exit_group	
253	lookup_dcookie	
254	epoll_create	
255	epoll_ctl	
256	epoll_wait	
257	remap_file_pages	
258	set_tid_address	
259	timer_create	
260	timer_settime	
261	timer_gettime	
262	timer_getoverrun	
263	timer_delete	
264	clock_settime	
265	clock_gettime	
266	clock_getres	
267	clock_nanosleep	
268	statfs64	
269	fstatfs64	
270	tgkill	Senden eines Signals an einen Thread (Verwendung der TGID und der TID)
271	utimes	
272	fadvise64_64	
273	nicht implementiert	(vserver)
274	mbind	
275	get_mempolicy	
276	set_mempolicy	
277	mq_open	
278	mq_unlink	
279	mq_timedsend	
280	mq_timedreceive	
281	mq_notify	
282	mq_getsetattr	
283	kexec_load	
284	waitid	Warten auf Zustandsänderung (z.B. Beendigung) eines Kindprozesses
mindestens ab Kernel 2.6.16 :		
285	nicht implementiert	
286	add_key	
287	request_key	
288	keyctl	
289	ioprio_set	
290	ioprio_get	

## Überblick über die *System Calls* in LINUX (6)

Fkt-Nr.	Name	Funktionalität
291	<code>inotify_init</code>	
292	<code>inotify_add_watch</code>	
293	<code>inotify_rm_watch</code>	
294	<code>migrate_pages</code>	
295	<code>openat</code>	Öffnen einer Datei oder eines Gerätes relativ zu einem Directory File Descriptor
296	<code>mkdirat</code>	Erzeugen eines Directories relativ zu einem Directory File Descriptor
297	<code>mknodat</code>	Erzeugen einer Gerätedatei oder einer Named Pipe relativ zu einem Dir File Descr.
298	<code>fchownat</code>	
299	<code>futimesat</code>	
300	<code>fstatat64</code>	
301	<code>unlinkat</code>	
302	<code>renameat</code>	
303	<code>linkat</code>	
304	<code>symlinkat</code>	
305	<code>readlinkat</code>	
306	<code>fchmodat</code>	
307	<code>faccessat</code>	
308	<code>pselect6</code>	Warten auf Statusänderungen von Datei-Objekten
309	<code>ppoll</code>	Warten auf ein Ereignis bei Datei-Objekten
310	<code>unshare</code>	
mindestens ab Kernel 2.6.22 :		
311	<code>set_robust_list</code>	
312	<code>get_robust_list</code>	
313	<code>splice</code>	direktes Kopieren zwischen File-Deskriptoren (einer muss eine Pipe referieren)
314	<code>sync_file_range</code>	
315	<code>tee</code>	Duplizieren von Daten von einer Pipe zu einer anderen
316	<code>vmsplice</code>	Einblenden von User-Speicher in eine Pipe
317	<code>move_pages</code>	
318	<code>getcpu</code>	
319	<code>epoll_pwait</code>	
320	<code>utimensat</code>	
321	<code>signalfd</code>	
322	<code>timerfd</code>	
323	<code>eventfd</code>	
mindestens ab Kernel 2.6.25 :		
324	<code>fallocate</code>	
325	<code>timerfd_settime</code>	
326	<code>timerfd_gettime</code>	

## System-Fehlercodes (Fehlernummern) in LINUX

### • Fehlercodes und Fehlermeldungen

- ◇ Die einzelnen **System Calls** erzeugen im Fehlerfall eine **Fehlernummer** (System-Fehlercode), die von den kapselnden Bibliotheksfunktionen in der globalen System-Variablen **errno** abgelegt wird.
- ◇ Die **möglichen System-Fehlercodes** sind in der Headerdatei **<asm-generic/errno.h>** definiert. Diese Header-Datei wird **indirekt** über die Headerdatei **<errno.h>** eingebunden :

```
#include <errno.h>
    → #include <bits/errno.h>
        → #include <linux/errno.h>
            → #include <asm/errno.h>
                → #include <asm-generic/errno.h>
```

- ◇ Den einzelnen Fehlercodes sind beschreibende **Fehlermeldungen** zugeordnet. Die Anfangsadressen der diese Fehlermeldungen enthaltenden Strings sind in einem **Pointer-Array** zusammengefaßt, das durch die globale Pointer-Variablen **sys\_errlist** referiert wird. Der **Fehlercode** dient als **Index** in dieses Array. Die Größe dieses Arrays ist in der globalen System-Variablen **sys\_nerr** enthalten.

### • Ermittlung der zu einem Fehlercode gehörenden Fehlermeldung

- ◇ ANSI-C-Standardbibliotheksfunktion

```
char* strerror(int errcode);
#include <string.h>
```

Diese Funktion gibt einen **Pointer** auf den **Fehlermeldungs-String** zurück, der dem als Parameter übergebenen **Fehlercode** **errcode** zugeordnet ist.

- ◇ ANSI-C-Standardbibliotheksfunktion

```
void perror(const char* msg);
#include <stdio.h>
```

Diese Funktion gibt den als Parameter übergebenen String **msg** nach **stderr** aus, gefolgt von einem Doppelpunkt, einem Blank und der **Fehlermeldung**, die dem in **errno** gespeicherten Fehlercode entspricht.

## Überblick über die System-Fehlercodes in LINUX (1)

### • Inhalt der Bibliotheks-Headerdatei <asm-generic/errno.h> (1. Teil)

```
#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

#define EPERM          1      /* Operation not permitted */
#define ENOENT          2      /* No such file or directory */
#define ESRCH          3      /* No such process */
#define EINTR          4      /* Interrupted system call */
#define EIO            5      /* I/O error */
#define ENXIO          6      /* No such device or address */
#define E2BIG          7      /* Arg list too long */
#define ENOEXEC        8      /* Exec format error */
#define EBADF          9      /* Bad file number */
#define ECHILD         10     /* No child processes */
#define EAGAIN         11     /* Try again */
#define ENOMEM         12     /* Out of memory */
#define EACCES         13     /* Permission denied */
#define EFAULT         14     /* Bad address */
#define ENOTBLK        15     /* Block device required */
#define EBUSY          16     /* Device or resource busy */
#define EEXIST         17     /* File exists */
#define EXDEV          18     /* Cross-device link */
#define ENODEV         19     /* No such device */
#define ENOTDIR        20     /* Not a directory */
#define EISDIR         21     /* Is a directory */
#define EINVAL         22     /* Invalid argument */
#define ENFILE         23     /* File table overflow */
#define EMFILE         24     /* Too many open files */
#define ENOTTY         25     /* Not a typewriter */
#define ETXTBSY        26     /* Text file busy */
#define EFBIG          27     /* File too large */
#define ENOSPC         28     /* No space left on device */
#define EPIPE          29     /* Illegal seek */
#define EROFS          30     /* Read-only file system */
#define EMLINK         31     /* Too many links */
#define EPIPE          32     /* Broken pipe */
#define EDOM           33     /* Math argument out of domain of func */
#define ERANGE         34     /* Math result not representable */
#define EDEADLK        35     /* Resource deadlock would occur */
#define ENAMETOOLONG   36     /* File name too long */
#define ENOLCK         37     /* No record locks available */
#define ENOSYS         38     /* Function not implemented */
#define ENOTEMPTY      39     /* Directory not empty */
#define ELOOP          40     /* Too many symbolic links encountered */
#define EWOULDBLOCK    EAGAIN /* Operation would block, Error-Code 11 */
#define ENOMSG         42     /* No message of desired type */
#define EIDRM          43     /* Identifier removed */
#define ECHRNG         44     /* Channel number out of range */
#define EL2NSYNC       45     /* Level 2 not synchronized */
#define EL3HLT         46     /* Level 3 halted */
#define EL3RST         47     /* Level 3 reset */
#define ELNRNG         48     /* Link number out of range */
#define EUNATCH        49     /* Protocol driver not attached */
#define ENOCSI         50     /* No CSI structure available */
#define EL2HLT         51     /* Level 2 halted */
#define EBADE          52     /* Invalid exchange */
#define EBADR          53     /* Invalid request descriptor */
#define EXFULL         54     /* Exchange full */
#define ENOANO         55     /* No anode */
#define EBADRQC        56     /* Invalid request code */
#define EBADSLT        57     /* Invalid slot */
#define EDEADLOCK      EDEADLK /* Errorcode 35 */
#define EBFONT         59     /* Bad font file format */
#define ENOSTR         60     /* Device not a stream */
#define ENODATA        61     /* No data available */
#define ETIME          62     /* Timer expired */
#define ENOSR          63     /* Out of streams resources */
#define ENONET         64     /* Machine is not on the network */
```

## Überblick über die System-Fehlercodes in LINUX (2)

### • Inhalt der Bibliotheks-Headerdatei <asm-generic/errno.h> (2. Teil)

```
#define ENOPKG 65 /* Package not installed */
#define EREMOTE 66 /* Object is remote */
#define ENOLINK 67 /* Link has been severed */
#define EADV 68 /* Advertise error */
#define ESRMNT 69 /* Srmount error */
#define ECOMM 70 /* Communication error on send */
#define EPROTO 71 /* Protocol error */
#define EMULTIHOP 72 /* Multihop attempted */
#define EDOTDOT 73 /* RFS specific error */
#define EBADMSG 74 /* Not a data message */
#define EOVERFLOW 75 /* Value too large for defined data type */
#define ENOTUNIQ 76 /* Name not unique on network */
#define EBADFD 77 /* File descriptor in bad state */
#define EREMCHG 78 /* Remote address changed */
#define ELIBACC 79 /* Can not access a needed shared library */
#define ELIBBAD 80 /* Accessing a corrupted shared library */
#define ELIBSCN 81 /* .lib section in a.out corrupted */
#define ELIBMAX 82 /* Attempting to link in too many shared libraries */
#define ELIBEXEC 83 /* Cannot exec a shared library directly */
#define EILSEQ 84 /* Illegal byte sequence */
#define ERESTART 85 /* Interrupted system call should be restarted */
#define ESTRPIPE 86 /* Streams pipe error */
#define EUSERS 87 /* Too many users */
#define ENOTSOCK 88 /* Socket operation on non-socket */
#define EDESTADDRREQ 89 /* Destination address required */
#define EMSGSIZE 90 /* Message too long */
#define EPROTOTYPE 91 /* Protocol wrong type for socket */
#define ENOPROTOPT 92 /* Protocol not available */
#define EPROTONOSUPPORT 93 /* Protocol not supported */
#define ESOCKTNOSUPPORT 94 /* Socket type not supported */
#define EOPNOTSUPP 95 /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT 96 /* Protocol family not supported */
#define EAFNOSUPPORT 97 /* Address family not supported by protocol */
#define EADDRINUSE 98 /* Address already in use */
#define EADDRNOTAVAIL 99 /* Cannot assign requested address */
#define ENETDOWN 100 /* Network is down */
#define ENETUNREACH 101 /* Network is unreachable */
#define ENETRESET 102 /* Network dropped connection because of reset */
#define ECONNABORTED 103 /* Software caused connection abort */
#define ECONNRESET 104 /* Connection reset by peer */
#define ENOBUFS 105 /* No buffer space available */
#define EISCONN 106 /* Transport endpoint is already connected */
#define ENOTCONN 107 /* Transport endpoint is not connected */
#define ESHUTDOWN 108 /* Cannot send after transport endpoint shutdown */
#define ETOOMANYREFS 109 /* Too many references: cannot splice */
#define ETIMEDOUT 110 /* Connection timed out */
#define ECONNREFUSED 111 /* Connection refused */
#define EHOSTDOWN 112 /* Host is down */
#define EHOSTUNREACH 113 /* No route to host */
#define EALREADY 114 /* Operation already in progress */
#define EINPROGRESS 115 /* Operation now in progress */
#define ESTALE 116 /* Stale NFS file handle */
#define EUCLEAN 117 /* Structure needs cleaning */
#define ENOTNAM 118 /* Not a XENIX named type file */
#define ENAVAIL 119 /* No XENIX semaphores available */
#define EISNAM 120 /* Is a named type file */
#define EREMOTEIO 121 /* Remote I/O error */
#define EDQUOT 122 /* Quota exceeded */
#define ENOMEDIUM 123 /* No medium found */
#define EMEDIUMTYPE 124 /* Wrong medium type */
#define ECANCELED 125 /* Operation Canceled */
#define ENOKEY 126 /* Required key not available */
#define EKEYEXPIRED 127 /* Key has expired */
#define EKEYREVOKED 128 /* Key has been revoked */
#define EKEYREJECTED 129 /* Key was rejected by service */
#define EOWNERDEAD 130 /* Owner died */
#define ENOTRECOVERABLE 131 /* State not recoverable */
#endif
```

## LINUX System Call `time`

- **Funktionalität :** Rückgabe der seit 1.1.1970, 00:00:00 UTC, vergangenen **Zeit** in **Sekunden**.

- **Interface :**

```
time_t time(time_t* tp);
```

- **Header-Datei :** `<time.h>`
- **Parameter :** `tp` Pointer auf eine Variable, die die Zeit in Sekunden aufnehmen kann.  
Falls die übergebene Adresse `!= NULL` ist, wird der Rückgabewert zusätzlich unter dieser Adresse abgelegt.
- **Rückgabewert :** - **Zeit in Sekunden** seit 1.1.1970, 00:00:00 UTC, bei Erfolg  
- `(time_t)-1`, im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **13**  
→ `sys_time(...)` (in `kernel/time.c`)
- **Anmerkungen :** 1. Der Datentyp `time_t` ist in der Header-Datei `<time.h>` **definiert** als **long**.  
2. Zur **Umwandlung** der sekundenbasierten Zeit (Typ `time_t`) in eine **komponenten-strukturierte Zeit** sowie in eine **Komponenten-Stringdarstellung** stehen geeignete **Funktionen der C-Standardbibliothek** zur Verfügung

## Ergänzungen zum LINUX System Call `time` (1)

### • Datentyp `struct tm`

- ◇ Dieser Datentyp dient zur **komponenten-strukturierten Darstellung** von **Zeiten**.
- ◇ In der Header-Datei `<time.h>` ist er wie folgt definiert :

```
struct tm
{
    int tm_sec;           /* Seconds.    [0-60] (1 leap second) */
    int tm_min;           /* Minutes.    [0-59]                  */
    int tm_hour;          /* Hours.      [0-23]                  */
    int tm_mday;          /* Day.        [1-31]                  */
    int tm_mon;           /* Month.      [0-11]                  */
    int tm_year;          /* Year - 1900.                  */
    int tm_wday;          /* Day of week. [0-6]                  */
    int tm_yday;          /* Days in year [0-365]               */
    int tm_isdst;         /* DST.        [-1/0/1]               */
};
```

### • Zeit-Umwandlungs-Funktionen der C-Standard-Bibliothek

- ◇ In der C-Standard-Bibliothek sind mehrere Funktionen zur Umwandlung der Zeitdarstellung definiert.
- ◇ Einige dieser in der Header-Datei `<time.h>` deklarierten Funktionen sind :

- **Umwandlung** der durch `ptime` referierten **sekundenbasierten Zeit** in eine **komponenten-strukturierte Zeit-Darstellung**

```
struct tm * gmtime(const time_t * ptime);
```

**Funktionswert** : Pointer auf `struct tm` - Variable, die Zeit als **UTC-Zeit** enthält

```
struct tm * localtime(const time_t * ptime);
```

**Funktionswert** : Pointer auf `struct tm` - Variable, die Zeit als **lokale Zeit** enthält

- **Umwandlung** der durch `ptime` referierten **sekundenbasierten Zeit** in eine **Komponenten-Stringdarstellung**

```
char* ctime(const time_t * ptime);
```

**Funktionswert** : Pointer auf C-String (Abschluss mit `'\n'`), der Zeit komponentenweise darstellt (umgewandelt in **lokale Zeit**)

- **Umwandlung** der durch `timeptr` referierten **komponenten-strukturierten Zeit** in eine **Stringdarstellung**

```
char* asctime(const struct tm * timeptr);
```

**Funktionswert** : Pointer auf C-String (Abschluss mit `'\n'`), der Zeit komponentenweise darstellt



## Ergänzungen zum LINUX System Call `time` (2)

- Demonstrationsbeispiel zum System Call `time` und zu den Zeit-Umwandlungsfunktionen

```
/* ----- */
/* C-Quelldatei timedemo_m.c                               */
/* main()-Funktion fuer das Programm timedemo              */
/* Demo-Programm zum Linux-System-Call time()              */
/* und den Zeit-Konvertierungsfunktionen der C-Standard-Bibliothek */
/* ----- */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void outTimeComp(struct tm * zeitKomp)
{ printf("tm_sec   : %d\n", zeitKomp->tm_sec);
  printf("tm_min   : %d\n", zeitKomp->tm_min);
  printf("tm_hour   : %d\n", zeitKomp->tm_hour);
  printf("tm_mday   : %d\n", zeitKomp->tm_mday);
  printf("tm_mon    : %d\n", zeitKomp->tm_mon);
  printf("tm_year   : %d\n", zeitKomp->tm_year);
  printf("tm_wday   : %d\n", zeitKomp->tm_wday);
  printf("tm_yday   : %d\n", zeitKomp->tm_yday);
  printf("tm_isdst  : %d\n", zeitKomp->tm_isdst);
}

int main(void)
{ time_t actZeit;
  struct tm actZeitKomp;
  actZeit = time(&actZeit);
  actZeitKomp = *gmtime(&actZeit);
  printf("\nAktuelle Zeit :\n");
  printf("\nin sek. seit 1.1.1970 00:00:00 UTC      : %ld\n", (long)actZeit);
  printf("\nin Komponenten-Stringdarstellung (UTC)    : %s", asctime(&actZeitKomp));
  printf("\nin Komponenten-Stringdarstellung (local) : %s", ctime(&actZeit));
  printf("\nWerte der Zeitkomponenten : \n");
  outTimeComp(&actZeitKomp);
  return EXIT_SUCCESS;
}

/* ----- */

Start und Ausgabe des Programms :

Aktuelle Zeit :

in sek. seit 1.1.1970 00:00:00 UTC      : 1077727321

in Komponenten-Stringdarstellung (UTC)    : Wed Feb 25 16:42:01 2004

in Komponenten-Stringdarstellung (local) : Wed Feb 25 17:42:01 2004

Werte der Zeitkomponenten :
tm_sec   : 1
tm_min   : 42
tm_hour   : 16
tm_mday   : 25
tm_mon    : 1
tm_year   : 104
tm_wday   : 3
tm_yday   : 55
tm_isdst  : 0

/- ----- */
```

## LINUX System Call `open`

- **Funktionalität :** **Öffnen** einer existierenden bzw **Erzeugen** einer neuen **Datei** oder Öffnen eines existierenden **Directories** oder eines existierenden **Gerätes**.  
Bei Erfolg Rückgabe eines die geöffnete Datei / Directory / Gerät referierenden **File-Deskriptors**.  
Hierbei handelt es sich um den kleinsten nicht offenen File-Deskriptor des Prozesses.  
Die **Bearbeitungsposition** wird auf den **Dateianfang** gesetzt.

- **Interface :**

```
int open(const char* path, int flags, ... /*mode_t mode */);
```

- **Header-Datei :** **<fcntl.h>**

**<sys/stat.h>** (nur wenn Parameter *mode* verwendet wird)

- **Parameter :**

*path* Zugriffspfad der Datei / des Directories bzw des Gerätes

*flags* Zugriffs- und Verwendungsflags

Die zulässigen Flags sind definiert in **<bits/fcntl.h>** (durch **<fcntl.h>** eingebunden).

Genau eines der folgenden Flags muß angegeben werden :

**O\_RDONLY** Datei soll für nur lesenden Zugriff geöffnet werden

**O\_WRONLY** Datei soll für nur schreibenden Zugriff geöffnet werden

**O\_RDWR** Datei soll für lesenden und schreibenden Zugriff geöffnet werden

Zusätzlich können eines oder mehrere der folgenden Flags bitweise oder-verknüpft hinzugefügt werden (die Aufzählung der Flags ist nicht vollständig) :

**O\_CREAT** Falls die Datei nicht existiert, soll sie erzeugt werden.

In diesem Fall ist auch der 3. Parameter *mode* anzugeben.

Nur für Dateien möglich, nicht für Directories oder Geräte.

**O\_EXCL** Wenn zusammen mit **O\_CREAT** verwendet u. Datei existiert → Fehlschlag

**O\_TRUNC** Die Länge einer existierenden Datei soll auf 0 gesetzt werden

**O\_APPEND** Schreiben soll nur am Dateiende zulässig sein

**O\_NONBLOCK** Öffnen im non-blocking Mode : z.B. kein Warten bis Lese-Daten verfügbar sind, sondern Beendigung des Lesens mit Fehler **EAGAIN**

**O\_SYNC** Öffnen im sync Mode : Der **write** System Call kehrt erst nach dem tatsächlichen Schreiben der Daten auf das Speichermedium zurück

*mode* Dateizugriffsrechte (nur notwendig für **O\_CREAT**, wird sonst ignoriert)

Mehrere Rechte können bitweise oder-verknüpft werden.

Die **tatsächlichen Zugriffsrechte** der Datei ergeben sich aus der bitweisen UND-Verknüpfung dieses Parameters mit der negierten Dateikreierungsmaske : **mode & ~umask**

Symbolische Konstanten für die Zugriffsrechte sind definiert in **<sys/stat.h>**.

s. System Call **creat()**.

- **Rückgabewert :** - **File-Deskriptor**, bei Erfolg  
- **-1** im Fehlerfall, **errno** wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 5  
→ **sys\_open(...)** (in **fs/open.c**)

- **Anmerkungen :** 1. Der Datentyp **mode\_t** ist – indirekt – in der Header-Datei **<sys/stat.h>** definiert als **unsigned int**.  
2. Das **Öffnen** eines **Directories** ist **nur** für den **lesenden Zugriff** zulässig.  
3. Ursprünglich diente die Wrapper-Funktion **open(...)** nur zum Öffnen von Objekten, deren Größe mit 31 Bits darstellbar ist. In neueren 32-Bit-Systemen kann sie auch auf Objekte angewendet werden, deren Größe mit bis zu 63 Bits darstellbar ist (**Large File Mode**) (Bibliotheks-Source-Code muss mit **\_FILE\_OFFSET\_BITS==64** übersetzt worden sein)  
Alternativ existiert hierfür auch als User-Interface **open64(...)** mit gleichen Parametern

## LINUX System Call **creat**

- **Funktionalität :** **Erzeugen** einer neuen **Datei** und **Öffnen** derselben zum Schreiben.  
Bei Erfolg Rückgabe eines die geöffnete Datei referierenden **File-Deskriptors**.  
Hierbei handelt es sich um den kleinsten nicht offenen File-Deskriptor des Prozesses.  
Die **Bearbeitungsposition** wird auf den **Dateianfang** gesetzt.

- **Interface :**

```
int creat(const char* path, mode_t mode );
```

- **Header-Datei :** `<fcntl.h>`  
`<sys/stat.h>`

- **Parameter :** *path* Zugriffspfad der Datei  
*mode* Dateizugriffsrechte.  
Mehrere Rechte können bitweise oder-verknüpft werden.  
Die **tatsächlichen Zugriffsrechte** der Datei ergeben sich aus der bitweisen UND-Verknüpfung dieses Parameters mit der negierten Dateikreierungsmaske *umask* des Prozesses : ***mode & ~umask***

In `<sys/stat.h>` sind definiert :

<b>S_IRUSR</b> ( <b>S_IREAD</b> )	Besitzer (User) darf Datei lesen	} <b>S_IRWXU</b>
<b>S_IWUSR</b> ( <b>S_IWRITE</b> )	Besitzer (User) darf Datei schreiben	
<b>S_IXUSR</b> ( <b>S_IEXEC</b> )	Besitzer (User) darf Datei ausführen	
<b>S_IRGRP</b>	Gruppe darf Datei lesen	} <b>S_IRWXG</b>
<b>S_IWGRP</b>	Gruppe darf Datei schreiben	
<b>S_IXGRP</b>	Gruppe darf Datei ausführen	
<b>S_IROTH</b>	andere Benutzer (Andere) dürfen Datei lesen	} <b>S_IRWXO</b>
<b>S_IWOTH</b>	andere Benutzer dürfen Datei schreiben	
<b>S_IXOTH</b>	andere Benutzer dürfen Datei ausführen	
<b>S_ISUID</b>	suid-Bit wird gesetzt	
<b>S_ISGID</b>	sgid-Bit wird gesetzt	
<b>S_ISVTX</b>	sticky-Bit wird gesetzt	

- **Rückgabewert :** - **File-Deskriptor**, bei Erfolg  
- **-1** im Fehlerfall, *errno* wird entsprechend gesetzt

- **Implementierung :** System Call Nr. **8**  
→ `sys_creat(...)` (in `fs/open.c`)  
→ `sys_open(...)` (in `fs/open.c`)

- **Anmerkung :**
  1. Der Datentyp **mode\_t** ist – indirekt – in der Header-Datei `<sys/stat.h>` definiert als **unsigned int**.
  2. `creat()` entspricht dem Aufruf von `open()` mit dem Parameter `flags = O_CREAT|O_WRONLY|O_TRUNC`

## LINUX System Call **lseek**

- **Funktionalität :** Setzen der **Datei-Bearbeitungsposition**.  
Die zu setzende Bearbeitungsposition wird als Byte-Offset relativ zu einer Bezugsposition angegeben.  
Rückgabe der neuen Bearbeitungsposition in Bytes relativ zum Dateianfang.
- **Interface :**

```
off_t lseek(int fd, off_t offset, int whence);
```

  - **Header-Datei :** `<unistd.h>`
  - **Parameter :**

<i>fd</i>	File-Deskriptor der Datei, für die die Bearbeitungsposition zu setzen ist. Es darf sich um keinen Deskriptor für eine Pipe, einen Socket oder ein FIFO handeln (→ Fehler)
<i>offset</i>	neue Bearbeitungsposition in Bytes relativ zur Bezugsposition <i>whence</i>
<i>whence</i>	Bezugsposition Einer der folgenden – in <code>&lt;unistd.h&gt;</code> definierten – Werte : <b>SEEK_SET</b> Bezugsposition ist der Dateianfang <b>SEEK_CUR</b> Bezugsposition ist die aktuelle Bearbeitungsposition <b>SEEK_END</b> Bezugsposition ist das Dateieende
  - **Rückgabewert :**
    - neue **Bearbeitungsposition** in Bytes relativ zum Dateianfang, bei Erfolg
    - **(off\_t)-1**, im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **19**  
→ `sys_lseek(...)` (in `fs/read_write.c`)
- **Anmerkungen :**
  1. Der Datentyp `off_t` ist – indirekt - in der Header-Datei `<unistd.h>` **definiert** als **long**.
  2. Eine Positionierung über das aktuelle Dateieende hinaus ist zulässig.  
Falls später ab dieser neuen Position geschrieben wird, liefern anschließende Leseoperationen in der "Lücke" 00-Bytes
  3. Die sinnvolle Anwendung des System Calls ist auf Dateien mit einer Länge  $\leq 2$  GBytes  
( $= 2^{**}(32-1)$ ) beschränkt
  4. Der System Call kann auch verwendet werden, um die **Länge einer Datei zu ermitteln** :  
→ **Positionierung** auf das **Dateieende** :  
`off_t len = lseek(fd, 0, SEEK_END);`

## LINUX System Call `_llseek`

- **Funktionalität :** Setzen der **Datei-Bearbeitungsposition** mittels eines 64 Bits Offsets  
( *Large File System* Unterstützung)  
Die zu setzende Bearbeitungsposition wird als Byte-Offset relativ zu einer Bezugsposition angegeben. Dieser wird in zwei 32-Bit-Werte ( höherwertiger Teil und niederwertiger Teil) zerlegt an die Betriebssystemfunktion übergeben.  
Rückgabe der neuen Bearbeitungsposition in Bytes relativ zum Dateianfang über einen als Pointer übergebenen Parameter

- **Interface** (nur für die Anwendung innerhalb der C-Bibliothek) :

```
int _llseek(int fd, long off_high, long off_low, loff_t* res, int whence)
```

- **Header-Datei :** Prototyp ist in keiner Header-Datei enthalten
- **Parameter :**
  - `fd` File-Deskriptor der Datei, für die die Bearbeitungsposition zu setzen ist. Es darf sich um keinen Deskriptor für eine Pipe, einen Socket oder ein FIFO Handeln (→ Fehler)
  - `off_high` höherwertiger Teil des – in Bytes relativ zur Bezugsposition `whence` angegebenen – Offsets der neuen Bearbeitungsposition (höherwertige 32 Bits)
  - `off_low` niederwertiger Teil des – in Bytes relativ zur Bezugsposition `whence` angegebenen – Offsets der neuen Bearbeitungsposition (niederwertige 32 Bits)
  - `res` Pointer auf einen 64 Bits großen Speicherbereich, in dem die neue Bearbeitungsposition – in Bytes relativ zum Dateianfang – abgelegt wird
  - `whence` Bezugsposition  
Einer der folgenden - in `<unistd.h>` definierten – Werte :
    - SEEK\_SET** Bezugsposition ist der Dateianfang
    - SEEK\_CUR** Bezugsposition ist die aktuelle Bearbeitungsposition
    - SEEK\_END** Bezugsposition ist das Dateieinde
- **Rückgabewert :**
  - `0`, bei Erfolg
  - `-1`, im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **140**  
→ `sys_llseek(...)` (in `fs/read_write.c`)

- **Interface für den Aufruf in Anwendercode** (Funktion der C-Bibliothek, die `_llseek()` aufruft) :

```
off64_t lseek64(int fd, off64_t offset, int whence)
```

- **Header-Datei :** `<unistd.h>`
- **Parameter :** `offset` Offset (64 Bits) der neuen Bearbeitungsposition in Bytes relativ zu `whence`
- **notwendig für Verwendung :** `#define _LARGEFILE64_SOURCE` (vor `#include <unistd.h>`)
- **Rückgabewert :**
  - neue Bearbeitungsposition (64 Bits) in Bytes relativ zum Dateianfang, bei Erfolg
  - `(loff_t) (-1)`, im Fehlerfall, `errno` wird entsprechend gesetzt
- **Anmerkungen :** Der Datentyp `loff_t` ist ebenso wie der Typ `off64_t` – indirekt – in der Header-Datei `<unistd.h>` definiert als `long long`.

## LINUX System Call **write**

- **Funktionalität :** **Schreiben** in ein durch einen File-Deskriptor referiertes Objekt.  
Es werden – maximal - eine anzugebende Anzahl Bytes, die aus einem anzugebenden Buffer entnommen werden, geschrieben
- **Interface :**

```
ssize_t write(int fd, const void* buf, size_t count);
```

  - **Header-Datei :** `<unistd.h>`
  - **Parameter :**
    - fd*      File-Deskriptor des Objekts, in das geschrieben werden soll
    - buf*      Anfangsadresse des Buffers, dem die zu schreibenden Daten entnommen werden sollen
    - count*   maximale Anzahl der zu schreibenden Bytes
  - **Rückgabewert :**
    - **Anzahl** der tatsächlich **geschriebenen Bytes**    bei Erfolg
    - **(`ssize_t`)-1**    im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **4**  
→ `sys_write(...)` (in `fs/read_write.c`)
- **Anmerkung :** Der Datentyp **`ssize_t`** ist - indirekt - in der Header-Datei `<unistd.h>` als **`int`** und der Typ **`size_t`** ist in der – von `<unistd.h>` eingebundenen - ANSI-C-Header-Datei `<stddef.h>` als **`unsigned int`** definiert.

## LINUX System Call **read**

- **Funktionalität :** **Lesen** aus einem durch einen File-Deskriptor referierten Objekt.  
Es werden – maximal - eine anzugebende Anzahl Bytes gelesen.  
Die gelesenen Bytes werden in einem anzugebenden Buffer abgelegt.  
Die Bearbeitungsposition wird um die Anzahl der gelesenen Bytes weiter gesetzt
- **Interface :**

```
ssize_t read(int fd, void* buf, size_t count);
```

  - **Header-Datei :** **<unistd.h>**
  - **Parameter :**
    - fd*        File-Deskriptor des Objekts, aus dem gelesen werden soll
    - buf*        Anfangsadresse des Buffers, in dem die zu lesenden Daten abgelegt werden sollen
    - count*    maximale Anzahl der zu lesenden Bytes
  - **Rückgabewert :**
    - **Anzahl** der tatsächlich **gelesenen Bytes** bei Erfolg
    - **(ssize\_t)-1** im Fehlerfall, **errno** wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **3**  
→ **sys\_read(...)** (in *fs/read\_write.c*)
- **Anmerkung :**
  1. Der Datentyp **ssize\_t** ist - indirekt - in der Header-Datei **<unistd.h>** als **int** und der Typ **size\_t** ist in der – von **<unistd.h>** eingebundenen - ANSI-C-Header-Datei **<stddef.h>** als **unsigned int** definiert.
  2. Das **Lesen von Directories** ist mit diesem System Call **nicht möglich**.  
Hierfür ist der System Call **getdents()** einzusetzen.

## LINUX System Call `close`

- **Funktionalität :** Schließen eines File-Deskriptors (und des dadurch referierten Objekts).

- **Interface :**

```
int close(int fd);
```

- **Header-Datei :** `<unistd.h>`
- **Parameter :** *fd* zu schließender File-Deskriptor
- **Rückgabewert :**
  - 0 bei Erfolg
  - -1 im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. 6  
→ `sys_close(...)` (in `fs/open.c`)
- **Anmerkung :**
  - Der Rückgabewert von `close()` einer zum Schreiben geöffneten Datei sollte immer überprüft werden.
  - Bei vielen Systemen werden die letzten geschriebenen Daten i.a. erst beim Schließen der Datei tatsächlich auf das Speichermedium geschrieben.
  - Bei fehlgeschlagenen Schließen (z.B. im NFS auf einem anderen Rechner) tritt i.a. ein Datenverlust auf.



## Beispiel für die Anwendung von LINUX System Calls

```
/* C-Quelldatei filecopy_m.c
   Implementiert ein Programm zum Kopieren einer Datei
   Einfaches Demonstrationsbeispiel zur Verwendung von LINUX System Calls
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>

#define BUFFSIZE 512

int filecopy(char* qdat, char* zdat)
{
    int fdq, fdz;
    int ret = 0;
    static char buff[BUFFSIZE];
    int anz;

    if ((fdq=open(qdat, O_RDONLY))<0)
        ret = -1; /* Fehler beim Oeffnen der Quelldatei */
    else
    {
        if ((fdz=open(zdat, O_WRONLY|O_CREAT|O_TRUNC, S_IWUSR|S_IRUSR))<0)
            /* oder : if ((fdz=creat(zdat, S_IWUSR|S_IRUSR))<0) */
            ret = -2; /* Fehler beim Oeffnen der Zieldatei */
        else
        {
            while ((anz=read(fdq, buff, BUFFSIZE))>0)
                write(fdz, buff, anz);
            if (close(fdz)==-1)
                ret=-3; /* Fehler beim Schliessen der Zieldatei */
        }
        close(fdq); /* Fehler beim Schliessen der Quelldatei ist unproblematisch */
    }
    return ret;
}

int main(int argc, char *argv[])
{
    int iRet;
    if (argc<3)
    {
        printf("Aufruf : filecopy <quelldatei> <zieldatei>\n");
        iRet = 1;
    }
    else
    {
        if ((iRet = filecopy(argv[1], argv[2]))==0)
            printf("Datei %s nach %s erfolgreich kopiert\n", argv[1], argv[2]);
        else
            printf("Datei %s nicht kopiert : Fehler (%d) ist aufgetreten (errno : %d)\n",
                argv[1], iRet, errno);
    }
    return iRet;
}
```

# **Betriebssysteme**

## **Kapitel 6**

### **6. Einige Kernel-Konzepte und -Mechanismen von LINUX**

- 6.1. Listenverwaltung
- 6.2. Warteschlangen
- 6.3. Synchronisation im Kernel
- 6.4. Interruptbearbeitung

## Listenverwaltung im Linux-Kernel (1)

### • Allgemeines

- ◇ An vielen Stellen arbeitet der Linux-Kernel mit **doppelt verketteten Listen** (z.B. Warteschlangen, Prozessliste usw). Die verschiedenen Listen unterscheiden sich u.a. auch in dem Typ ihrer Elemente.
- ◇ Zur **Implementierung** und Verwaltung derartiger Listen wird im LINUX-Quellcode ein **vom Typ der Listenelemente unabhängiger** – generischer – **Mechanismus** eingesetzt.

### • Implementierung

- ◇ Den Kern der Implementierung bildet der in der Headerdatei **include/linux/list.h** definierte **Structure-Typ**

```
struct list_head {  
    struct list_head *next, *prev;  
}
```

Dieser Typ fasst die **Pointer** für die **Vorwärts- und Rückwärtsverkettung** der Listenelemente zusammen.

- ◇ **Jedes Element** einer **verketteten Liste** muss eine **Komponente dieses Typs besitzen**. Die Position dieser Komponente innerhalb des Listenelements eines bestimmten Typs spielt keine Rolle (natürlich müssen aber alle Elemente einer Liste jeweils vom gleichen Typ, d.h. gleich aufgebaut, sein).

**Beispiel** (Prozessliste) :

```
struct task_struct {  
    // ...  
    struct list_head tasks;  
    // ...  
}
```

- ◇ Als **Listenkopf** dient i.a. ebenfalls eine Instanz des Typs `struct list_head`. Diese wird üblicherweise mit dem ebenfalls in **include/linux/list.h** definierten **Makro LIST\_HEAD erzeugt und initialisiert** :

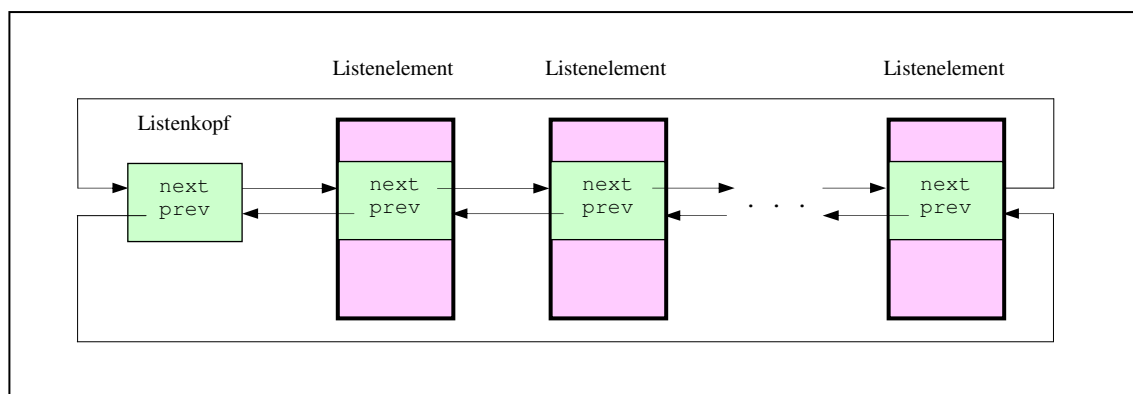
```
#define LIST_HEAD(name)    struct list_head name = {&(name), &(name)}
```

Der **Makro INIT\_LIST\_HEAD** dient zur **Initialisierung** eines durch den Pointer **ptr** referierten **Listenkopfes**.

```
#define INIT_LIST_HEAD(ptr) do { (ptr)->next = (ptr); (ptr)->prev = (ptr); } while(0)
```

Über den Listenkopf kann zum **ersten** und **letzten Element** einer doppelt verketteten Liste in immer der gleichen von der **Listengröße unabhängigen Zeit** zugegriffen werden.

- ◇ **Prinzipieller Aufbau** doppelt verketteter Listen :



- ◇ Zum weiteren **Aufbau** und zur **Bearbeitung** doppelt verketteter Listen existieren eine Reihe – meist als **inline-Funktionen** realisierte – **Makros** ("Listen-Bearbeitungsfunktionen").

## Listenverwaltung im Linux-Kernel (2)

- "Listen-Bearbeitungsfunktionen" (Auswahl)

- ◇ Sie sind ebenfalls in der Headerdatei `include/linux/list.h` definiert.

- ◇ Einfügen eines neuen Listenelementes **neu** hinter das vorhandene Element **act** mittels `list_add(neu, act)`  
Wenn **act==Listenkopf** : Einfügen an den **Listenanfang**

**Implementierung :**

```
static inline void list_add(struct list_head* neu, struct list_head* act)
{ __list_add(neu, act, act->next); }
```

**Implementierung der internen Hilfsfunktion \_\_list\_add() :**

```
static inline void __list_add(struct list_head* neu,
                             struct list_head* prev, struct list_head* next)
{
    next->prev = neu;
    neu->next = next;
    neu->prev = prev;
    prev->next = neu;
}
```

- ◇ Einfügen eines neuen Listenelementes **neu** vor das vorhandene Element **act** mittels `list_add_tail(neu, act)`  
Wenn **act==Listenkopf** : Einfügen an das **Listenende**

**Implementierung :**

```
static inline void list_add_tail(struct list_head* neu, struct list_head* act)
{ __list_add(neu, act->prev, act); }
```

- ◇ Entfernen des Listenelements **entry** mittels `list_del(entry)`

```
static inline void list_del(struct list_head* entry)
```

- ◇ Überprüfung, ob die durch den **Listenkopf head** referierte **Liste leer** ist, mittels `list_empty(head)`

**Implementierung :**

```
static inline int list_empty(const struct list_head* head)
{ return head->next == head; }
```

- ◇ Iteration über **alle Elemente** einer durch den **Listenkopf head** referierten Liste mittels des Makros `list_for_each(pos, head)` (*pos* und *head* sind vom Typ `struct list_head*`)

**Implementierung :**

```
#define list_for_each(pos, head) \
    for (pos = (head)->next; prefetch(pos->next), pos != head; pos = pos->next)
```

**Modifizierter Makro**, bei dem die Iteration **sicher** gegen das **Entfernen** des jeweils aktuellen **Listenelements** ist :

```
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != head; pos = n, n = pos->next)
```

- Ermittlung eines Pointers auf den Anfang eines Listenelements

- ◇ Mittels des Makros `list_entry()`  
(indirekt definiert in `include/linux/list.h`, in Verbindung mit `include/linux/kernel.h`)

```
#define list_entry(ptr, type, member) \
    ((type *) ((char *) (ptr) - (unsigned long) (&((type *) 0)->member)))
```

- ◇ Beispielsweise wird der Aufruf

```
list_entry(tmp, struct task_struct, tasks) mit struct list_head* tmp
```

expandiert zu

```
((struct task_struct *) ((char *) (tmp) - (unsigned long) (&((struct task_struct *) 0)->tasks)))
```

## Warteschlangen im Linux-Kernel (1)

### • Allgemeines

- ◇ Häufig hängt die **Fortsetzung** eines Prozesses vom Eintreten einer bestimmten **Bedingung** bzw **Ereignisses** ab. (z.B. Warten auf das Lesen eines Datenbereichs von der Festplatte oder auf das Ende eines Kindprozesses). In einem derartigen Fall trägt sich der wartenden Prozeß in eine **Warteschlange** (*wait queue*) ein und gibt die CPU frei. (→ Aufruf des Schedulers). → Der Prozeß "**legt sich schlafen**".
- ◇ Prinzipiell können Prozesse auf das gleiche Ereignis warten.  
Für **jedes Ereignis**, auf das gewartet wird, existiert eine **eigene Warteschlange**.  
→ alle Prozesse, die auf das gleiche Ereignis warten, befinden sich in der gleichen Warteschlange.
- ◇ **Tritt das Ereignis ein**, so werden **alle Prozesse**, die sich in der zugeordneten Warteschlange befinden, **aufgeweckt**. Durch Aufnahme in die **Run Queue** werden sie wieder Kandidaten für die Auswahl des nächsten laufenden Prozesses durch den Scheduler.  
Wenn ein Prozeß wieder fortgesetzt wird, **entfernt er sich** als erstes **selbst** aus der Warteschlange.

### • Realisierung von Warteschlangen

- ◇ Eine Warteschlange wird durch eine zyklisch **doppelt verkettete Liste** realisiert, deren Elemente jeweils einen Pointer auf eine **Task-Struktur** enthalten. Eine Task-Struktur dient zur Beschreibung eines Prozesses (Prozeßdeskriptor).
- ◇ Zur Beschreibung eines **Elements** einer derartigen Liste dient der folgende in **include/linux/wait.h** definierte Structurtyp :

```
struct __wait_queue {
    unsigned int flags;                /* Flags zur Beeinflussung des Aufweckens */
    wait_queue_func_t func;           /* Funktion, die beim Aufwecken aufgerufen werden soll */
    struct task_struct* task;         /* Pointer auf Prozeßverwaltungsstruktur */
    struct list_head task_list;       /* Verkettungspointer zum Aufbau der Liste */
};

typedef struct __wait_queue wait_queue_t;
```

- ◇ Die Komponente **func** des Typs **struct \_\_wait\_queue** legt die Funktion fest, die beim Aufwecken des Prozesses ausgeführt werden soll ("**Aufweckfunktion**").  
Der Typ dieser Komponente ist der – ebenfalls in **include/linux/wait.h** definierte – **Funktionspointer-Typ** **wait\_queue\_func\_t** :  

```
typedef int (*wait_queue_func_t)(wait_queue_t* wait, unsigned mode, int sync, void* key);
```

  
Defaultmässig wird als "Aufweckfunktion" verwendet  
(deklariert in **include/linux/wait.h**, definiert in **kernel/sched.c**) :

```
int default_wake_function(wait_queue_t* wait, unsigned mode, int sync, void* key);
```

- ◇ Die Komponente **flags** des Typs **struct \_\_wait\_queue** enthält Flags zur Beeinflussung des Aufweckens. Derzeit ist **nur ein Flag** definiert :
  - **WQ\_FLAG\_EXCLUSIVE**  
Es dient zur Markierung von Prozessen, die bei einer Beschränkung exklusiv aufzuweckender Prozesse zu berücksichtigen sind.
- ◇ Eine **Warteschlange** wird durch einen **Listenkopf** repräsentiert und angesprochen (**Warteschlangenkopf**). Zur Beschreibung von Warteschlangenköpfen dient der folgende ebenfalls in **include/linux/wait.h** definierte Structurtyp :

```
struct __wait_queue_head {
    spinlock_t lock;                /* Locking-Variable, in SMP-Systemen benötigt (Spin-Lock) */
    struct list_head task_list;     /* Verkettungspointer auf das erste und letzte Listenelement */
};

typedef struct __wait_queue_head wait_queue_head_t;
```

## Warteschlangen im Linux-Kernel (2)

### • Einrichten von Warteschlangen

- ◇ Eine Warteschlange wird durch **Erzeugung** eines **Warteschlangenkopfes** eingerichtet.
- ◇ Zur **vereinfachten Erzeugung** und **Initialisierung** eines Warteschlangenkopfes ist in **include/linux/wait.h** der folgende **Makro** definiert :

```
#define DECLARE_WAIT_QUEUE_HEAD(name) wait_queue_head_t name = { \
    .lock = SPIN_LOCK_UNLOCKED, .task_list = { &(amp;name).task_list, &(amp;name).task_list } }
```

### • Initialisierung von Warteschlangenelementen

- ◇ Zur Initialisierung eines Warteschlangenelementes lässt sich die folgende in **include/linux/wait.h** definierte **inline-Funktion** einsetzen (Daneben existiert ein Makro für die Erzeugung und gleichzeitige Initialisierung) :

```
static inline void init_waitqueue_entry(wait_queue_t* q, struct task_struct* p)
{ q->flags = 0; q->task = p; q->func = default_wake_function; }
```

### • Kernelfunktionen zum Modifizieren von Warteschlangen

- ◇ In einer Multitasking-Umgebung muss ein verändernder Zugriff zu Strukturen, wie den Warteschlangen, **synchronisiert** erfolgen. Diesen synchronisierten Zugriff stellen die folgenden **Kernel-Funktionen** zur Verfügung :

- ◇ **Einfügen eines Elementes in eine Warteschlange** (definiert in **linux/wait.c**):

```
void add_wait_queue(wait_queue_head_t* q, wait_queue_t* wait);
void add_wait_queue_exclusive(wait_queue_head_t* q, wait_queue_t* wait);
```

#### ▷ Bedeutung der Parameter :

**q** : Pointer auf den Warteschlangenkopf  
**wait** : Pointer auf das neu hinzuzufügende Element

#### ▷ Die Funktion **add\_wait\_queue()**

- sorgt dafür, dass das Flag **WQ\_FLAG\_EXCLUSIVE** in der Komponente **flags** des durch **wait** referierten Elements **nicht gesetzt** ist und
- fügt das Element an den **Anfang** der durch **q** referierten Warteschlange ein.  
Hierfür ruft sie die in **include/linux/wait.h** definierte **inline-Funktion** auf :

```
static inline void __add_wait_queue(wait_queue_head_t* head, wait_queue_t* neu)
{ list_add(&neu->task_list, &head->task_list); }
```

#### ▷ Die Funktion **add\_wait\_queue\_exclusive()**

- **setzt** das Flag **WQ\_FLAG\_EXCLUSIVE** in der Komponente **flags** des durch **wait** referierten Elements und
- fügt das Element an das **Ende** der durch **q** referierten Warteschlange ein.  
Hierfür ruft sie die in **include/linux/wait.h** definierte **inline-Funktion** auf :

```
static inline void __add_wait_queue_tail(wait_queue_head_t* head, wait_queue_t* neu)
{ list_add_tail(&neu->task_list, &head->task_list); }
```

- ◇ **Entfernen eines Elements aus einer Warteschlange** (definiert in **linux/wait.c**):

```
void remove_wait_queue(wait_queue_head_t* q, wait_queue_t* wait);
```

#### ▷ Bedeutung der Parameter :

**q** : Pointer auf den Warteschlangenkopf  
**wait** : Pointer auf das zu entfernende Element

- ▷ Die Funktion entfernt das über **wait** referierte Element aus der durch **q** referierten Warteschlange.  
Hierfür ruft sie die in **include/linux/wait.h** definierte **inline-Funktion** auf :

```
static inline void __remove_wait_queue(wait_queue_head_t* head, wait_queue_t* old)
{ list_del(&old->task_list); }
```

## Warteschlangen im Linux-Kernel (3)

### • Eintragen eines Prozesses in eine Warteschlange

- ◇ Prozesse, die in eine Warteschlange eingetragen werden, werden entweder in den Zustand **TASK\_INTERRUPTABLE** oder in den Zustand **TASK\_UNINTERRUPTABLE** versetzt.  
Ein Prozess, der sich im Zustand **TASK\_UNINTERRUPTABLE** befindet, kann **nur** durch den **Eintritt des Ereignisses**, auf das er wartet, wieder **aufgeweckt** werden.  
Ein Prozess, der sich im Zustand **TASK\_INTERRUPTABLE** befindet, kann **auch** durch ein **Signal** bzw durch den **Ablauf** einer gewählten **Zeit** wieder **aufgeweckt** werden.  
Ein Prozess, der aufgeweckt wird, wird in den Zustand **TASK\_RUNNING** versetzt.
- ◇ Es existieren die folgenden drei in **kernel/sched.c** definierten Kernelfunktionen mit denen der aktuelle Prozess sich "schlafenlegen", d.h. in eine **Warteschlange eintragen** kann.  
Nach dem Eintrag des Prozesses in die Warteschlange **rufen** alle drei Funktionen den **Scheduler** auf.  
Sie werden erst dann wieder fortgesetzt, wenn dem jeweiligen Prozess – nach dem Aufwecken – vom Scheduler die CPU wieder zugeteilt wird.  
Nach der Fortsetzung tragen sie den Prozess wieder aus der Warteschlange aus.

- ◇ "Schlafenlegen" im Zustand **TASK\_UNINTERRUPTABLE**

```
void sleep_on(wait_queue_head_t* q);
```

Bedeutung des Parameters **q** : Pointer auf den Warteschlangenkopf

- ◇ "Schlafenlegen" im Zustand **TASK\_INTERRUPTABLE** (aufweckbar auch durch ein Signal)

```
void interruptable_sleep_on(wait_queue_head_t* q);
```

Bedeutung des Parameters **q** : Pointer auf den Warteschlangenkopf

- ◇ "Schlafenlegen" im Zustand **TASK\_INTERRUPTABLE** (aufweckbar auch durch Zeitablauf)

```
long interruptable_sleep_on_timeout(wait_queue_head_t* q, long timeout);
```

Bedeutung der Parameter : **q** : Pointer auf den Warteschlangenkopf  
**timeout** : Zeit nach deren Ablauf der Prozess spätestens geweckt werden soll

- ◇ **Beispiel : Prinzipielle Realisierung der Funktion sleep\_on()**

(vereinfachte Darstellung, Belegung und Freigabe des Spin-Locks bei Veränderung der Warteschlange ist jeweils weggelassen)

```
void sleep_on(wait_queue_head_t* q)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_UNINTERRUPTABLE;
    __add_wait_queue(q, &wait);
    schedule();
    __remove_wait_queue(q, &wait);
}
```

**Wesentlicher Unterschied** zu den beiden anderen **sleep\_on**-Funktionen :

**interruptable\_sleep\_on()** und **interruptable\_sleep\_on\_timeout()** setzen den Zustand des aktuellen (und schlafenzulegenden) Prozesses auf **TASK\_INTERRUPTABLE**.

## Warteschlangen im Linux-Kernel (4)

### • Aufwecken von in einer Warteschlange eingetragenen Prozessen

- ◇ Hierfür lassen sich einige in `include/linux/wait.h` definierte **Makros** einsetzen :

```
#define wake_up(x)          __wake_up(x, TASK_INTERRUPTABLE|TASK_UNINTERRUPTABLE, 1, NULL)
#define wake_up_nr(x, nr)   __wake_up(x, TASK_INTERRUPTABLE|TASK_UNINTERRUPTABLE, nr, NULL)
#define wake_up_all(x)      __wake_up(x, TASK_INTERRUPTABLE|TASK_UNINTERRUPTABLE, 0, NULL)
#define wake_up_interruptable(x)      __wake_up(x, TASK_INTERRUPTABLE, 1, NULL)
#define wake_up_interruptable_nr(x, nr) __wake_up(x, TASK_INTERRUPTABLE, nr, NULL)
#define wake_up_interruptable_all(x)   __wake_up(x, TASK_INTERRUPTABLE, 0, NULL)
```

Alle Makros rufen die in `kernel/sched.c` definierte Funktion `void __wake_up(...)` auf

Die Gruppe der **Makros** `wake_up_interruptable_xx()` wecken **nur Prozesse** auf, die sich im Zustand **TASK\_INTERRUPTABLE** befinden, während die **übrigen Makros auch Prozesse**, die sich im Zustand **TASK\_UNINTERRUPTABLE** befinden, aufwecken.

Bedeutung des Makro-Parameters **x** : Pointer auf den Warteschlangenkopf

- ◇ Die von allen `wake_up`-Makros aufgerufene Funktion `__wake_up()` ist wie folgt deklariert :

```
void __wake_up(wait_queue_head_t* q, unsigned int mode, int nr_exclus, void* key);
```

Bedeutung der Parameter **q** : Pointer auf den Warteschlangenkopf  
**mode** : Maske von Prozesszuständen, die die aufzuweckenden Prozesse festlegt  
**nr\_exclus** : Max. Anzahl von Prozessen mit gesetztem **WQ\_FLAG\_EXCLUSIVE**, die aufgeweckt werden dürfen  
**key** : ? (offensichtlich z.Zt. nicht verwendet, ==NULL)

Die Funktion

- belegt den mit der Warteschlange verknüpften Spin-Lock
- ruft zur Durchführung ihrer eigentlichen Arbeit die Kernel-Funktion `__wake_up_common()` auf :  
`__wake_up_common(q, mode, nr_exclus, 0, key);`
- und gibt den belegten Spin-Lock anschliessend wieder frei.

- ◇ Die Funktion `__wake_up_common()` veranlasst das Aufwecken der in der Warteliste eingetragenen Prozesse. Sie ist in `kernel/sched.c` wie folgt definiert :

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                             int nr_exclusive, int sync, void *key)
{ struct list_head *tmp, *next;
  list_for_each_safe(tmp, next, &q->task_list) {
    wait_queue_t *curr;
    unsigned flags;
    curr = list_entry(tmp, wait_queue_t, task_list);
    flags = curr->flags;
    if (curr->func(curr, mode, sync, key) &&
        (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
      break;
  }
}
```

Die Funktion **iteriert** über die in der **Warteliste** eingetragenen **Prozesse** und **ruft** deren jeweilige "**Aufweckfunktion**" (referiert durch die Listenelement-Komponente **func**) **auf**. Standardmässig ist diese Komponente auf die Anfangsadresse der **Default-"Aufweckfunktion"** `default_wake_function()` gesetzt.

Die "**Aufweckfunktion**" sorgt dafür, dass der jeweils **aufgeweckte Prozesse** in den Zustand **TASK\_RUNNING** versetzt und – sofern er noch nicht enthalten ist – in die **Run Queue** aufgenommen wird.

Die **Iteration endet**, wenn

- alle in der Warteliste eingetragenen Prozesse aufgeweckt worden sind oder
- die durch `nr_exclusive` festgelegte Anzahl von `WQ_FLAG_EXCLUSIVE`-Prozessen aufgeweckt worden sind.



## Warteschlangen im Linux-Kernel (5)

- **Aufwecken von in einer Warteschlange eingetragenen Prozessen, Forts.**

- ◇ Die Default-"Aufweckfunktion" `default_wake_function()` ist in `kernel/sched.c` wie folgt definiert :

```
int default_wake_function(wait_queue_t *curr, unsigned mode, int sync, void *key)
{
    task_t *p = curr->task;
    return try_to_wake_up(p, mode, sync);
}
```

Sie ruft die Kernel-Funktion `try_to_wake_up()` auf, die den aufgeweckten Prozess in den Zustand **TASK\_RUNNING** versetzt und seine Aufnahme in die *Run Queue* bewirkt.

- ◇ Die Funktion `try_to_wake_up()` ist ebenfalls in `kernel/sched.c` definiert  
Nachfolgend ist ihre Definition **auszugsweise** wiedergegeben :

```
static int try_to_wake_up(task_t * p, unsigned int state, int sync)
{
    int cpu, this_cpu, success = 0;
    // ...
    long old_state;
    // ...
    old_state = p->state;
    if (!(old_state & state))           // Prozess befindet sich in keinem Wartezustand
        goto out;
    if (p->array)                       // Prozess ist bereits in der Run Queue enthalten
        goto out_running;
    // ...

#ifdef CONFIG_SMP
    // ...                               // nur für SMP-Systeme relevanter Code
#endif /* CONFIG_SMP */

    // ...
    activate_task(p, rq, cpu == this_cpu);
    // ...
    success = 1;
out_running:
    p->state = TASK_RUNNING;
out:
    // ...
    return success;
}
```

Die Funktion `activate_task()` bewirkt das **Einfügen** des aufgeweckten Prozesses in die *Run Queue*. Hierzu ruft sie – indirekt über eine **weitere Zwischen-Funktion** – die Funktion `enqueue_task()` auf.

- **Zusammenfassender Überblick über das Aufwecken "schlafender" Prozesse :**

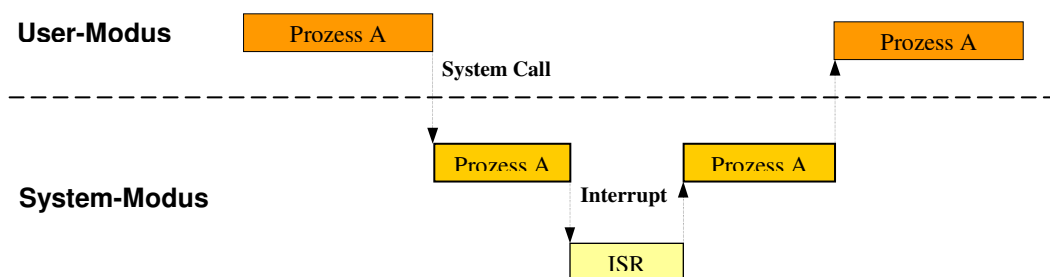
```
wake_up-Makro
→ __wake_up()
  → __wake_up_common()
    → default_wake_function()      (allgemein : prozess-spezifische "Aufweckfunktion")
      → try_to_wake_up()
        → activate_task()          (bewirkt das Einfügen des Prozesses in die Run Queue)
          → __activate_task()
            → enqueue_task()
              Setzen Prozesszustand auf TASK_RUNNING
```

## Synchronisation im Linux-Kernel

### • Notwendigkeit der Synchronisation im Kernel

- ◇ **Gemeinsam genutzte Ressourcen** müssen vor **konkurrierendem** (parallelem bzw pseudo-parallel) **Zugriff** durch **mehrere Prozesse** (allg. : unabhängige Anweisungsfolgen, *threads of execution*) **geschützt** werden, wenn mindestens einer der Zugriffe diese **Ressourcen** verändert (→ Schutz vor **Race Conditions**).
- ◇ Dies gilt auch für die **Datenstrukturen des Kernels**.  
Zu diesen kann nur der im **System-Modus** ausgeführte **Kernel-Code** zugreifen.
- ◇ **Kernel-Code** wird ausgeführt in
  - ▷ im Kontext von User-Prozessen ausgeführten **System Calls**
  - ▷ **Interrupt Service Routinen**
  - ▷ verzögertem Reaktions-Code (**Soft-IRQs**, **Tasklets**)
  - ▷ **Kernel-Threads**
- ◇ Eine im System-Modus ausgeführte Anweisungsfolge wird auch als **Kernel-Kontroll-Pfad** (*kernel control path*) bezeichnet.
- ◇ In folgenden **Situationen** ist ein **konkurrierender Zugriff** zu **Kernel-Daten** durch zwei oder mehr Kernel-Kontroll-Pfade prinzipiell möglich :
  - ▷ aktiviertes SMP (*symmetric multiprocessing*) auf **Multi-Prozessor-Maschinen** :  
Mehrere Prozesse können z.B. gleichzeitig System Calls ausführen.
  - ▷ Eintritt eines **Interrupts während** der Ausführung eines **System Calls** bzw einer **Interrupt Service Routine**
  - ▷ Prozesswechsel während sich ein nicht-blockierter Prozess im System-Modus befindet (**Kernel-Präemption**, ab Kernel 2.6 möglich) und der neue Prozess anschliessend auch einen System Call ausführt.
- ◇ Die Anweisungsfolge, in der ein Zugriff zu einer bestimmten Datenstruktur (allg. : Resource), die nicht konkurrierend verwendet werden darf, erfolgt, wird auch als **kritischer Abschnitt** (*critical region*) bezeichnet.  
Es dürfen sich **niemals zwei** (oder mehr) **Kernel-Kontroll-Pfade** gleichzeitig in einem zur gleichen Datenstruktur gehörenden kritischen Abschnitt befinden → **Synchronisation** erforderlich

### • Konkurrierende Ausführung von System Call und Interrupt Service Routine



### • Synchronisations-Mechanismen im Kernel

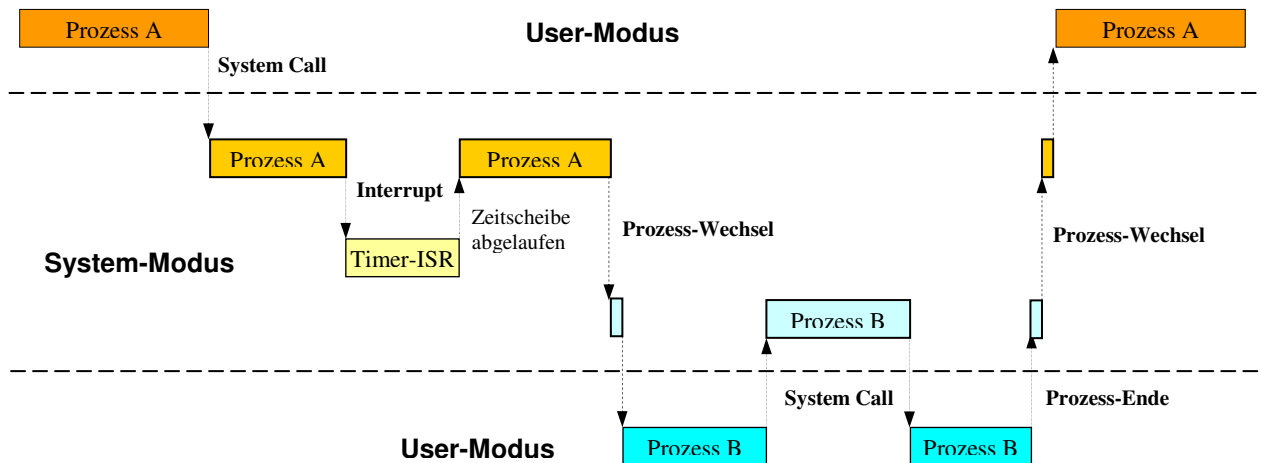
- ◇ Im Kernel sind mehrere Synchronisations-Mechanismen implementiert.  
Die wichtigsten sind :
  - ▷ **atomare Operationen**
  - ▷ **Spin-Locks**
  - ▷ **Semaphore**

## Anmerkungen zur Kernel-Präemption

### • Nicht-präemptiver Kernel

- ◇ Bis einschliesslich der Version **2.4** war der Linux-Kernel **nicht-präemptiv** (*non preemptive*).  
Bei einem nicht-präemptiven Kernel kann ein Prozess nur dann aus der CPU **verdrängt** werden (wegen Zeitscheibenablauf oder durch einen höherprioren Prozess), wenn er sich im (genauer : unmittelbar vor der Rückkehr in den) **Benutzer-Modus** befindet (am Ende eines System Calls bzw einer Interruptbearbeitung). Eine Verdrängung im System-Modus (**innerhalb** eines System Calls) ist **nicht möglich**

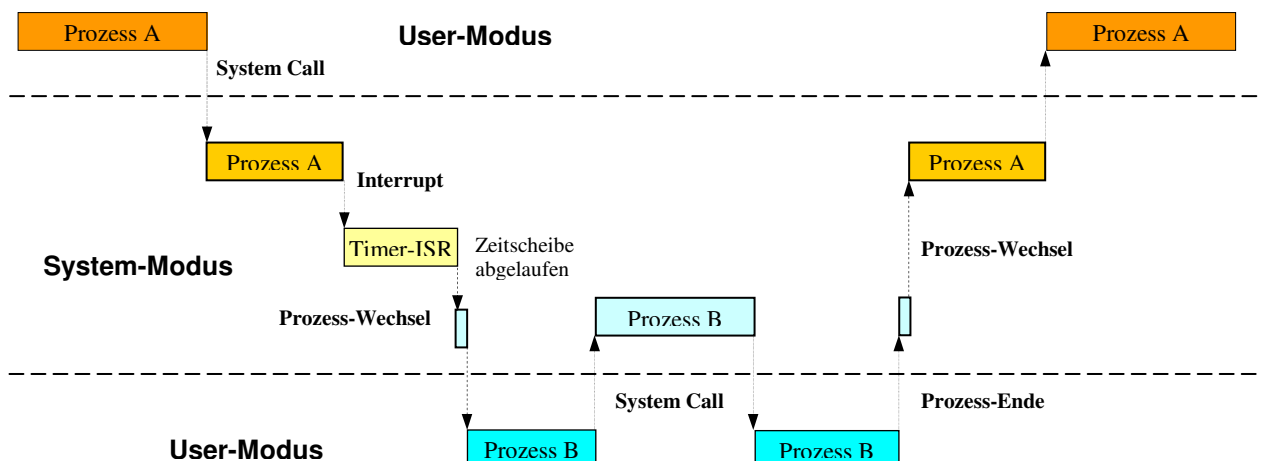
#### ◇ Prinzipielles Beispiel :



### • Präemptiver Kernel

- ◇ Ab der Version **2.6** ist der Linux-Kernel **präemptiv**.  
Eine Verdrängung ist auch dann möglich, wenn der aktuelle Prozess sich im System-Modus (also z.B. inmitten eines System Calls) befindet.  
Dies wird durch folgende Situation realisiert : Während der Ausführung des System Calls tritt ein Interrupt auf, durch den die Zeitscheibe des aktuellen Prozesses beendet (Timer-Interrupt) oder die Blockade eines höherprioren Prozesses aufgehoben wird (z.B. Interrupt nach I/O-Beendigung).  
Am Ende der Interrupt Service Routine – vor Beendigung des System Calls – erfolgt ein Prozesswechsel.  
→ Ein wegen Zeitscheibenablauf oder Prioritätsverdrängung notwendiger Prozesswechsel kann **schneller** erfolgen
- ◇ Die Kernel-Präemption kann im Kernel **temporär abgeschaltet** werden → weiterer Synchronisations-Mechanismus

#### ◇ Prinzipielles Beispiel :



## Temporäre Verhinderung konkurrierender Kernel-Kontroll-Pfade

### • Ein-/Ausschalten der Kernel-Präemption

- ◇ Ob Kernel-Präemption implementiert ist, lässt sich beim Compilieren des Linux-Kernels festlegen. Durch Definition der Preprozessor-Konstanten **CONFIG\_PREEMPT** wird der Kernel für **Kernel-Präemption konfiguriert**.
- ◇ Grundsätzlich darf eine **Unterbrechung** von **Kernel-Code** aber **nur dann** stattfinden, wenn sichergestellt ist, dass in dem unterbrochenen Kontroll-Pfad nicht gerade Daten manipuliert werden, zu denen auch in dem neu aktivierten Kontroll-Pfad zugegriffen werden könnte.  
→ Es muss möglich sein, die Kernel-Präemption **temporär auszuschalten**.
- ◇ Ob aktuell Kernel-Präemption stattfinden darf, also eingeschaltet ist, wird durch die Komponente **preempt\_count** der für jeden Prozess existierenden **thread\_info**-Struktur festgelegt (s. Datenstrukturen zur Prozessverwaltung). Hat diese Komponente für den aktuellen Prozess den Wert **0**, ist **Kernel-Präemption zulässig**, bei einem Wert **>0** ist sie dagegen unzulässig.  
Zur Manipulation dieser Komponente dienen die Makros (definiert in **include/linux/preempt.h**)
  - ▷ **inc\_preempt\_count()** (Aufruf vor Eintritt in einen kritischen Abschnitt)
  - ▷ **dec\_preempt\_count()** (Aufruf nach Verlassen des kritischen Abschnitts)
- ◇ Als weitere – obige Makros aufrufende – Makros sind definiert (ebenfalls in **include/linux/preempt.h**):
  - ▷ **preempt\_disable()** (ruft **inc\_preempt\_count()** auf)  
→ Deaktivierung der Kernel-Präemption
  - ▷ **preempt\_enable()** (ruft **dec\_preempt\_count()** auf)  
→ Aktivierung der Kernel-Präemption, wenn durch das Decrementieren **preempt\_count** (des aktuellen Prozesses) **==0** geworden ist, Überprüfung auf einen anstehenden Prozesswechsel und gegebenenfalls Aufruf des Schedulers.

### • Sperren/Freigabe von Interrupts

- ◇ Ein **kritischer Abschnitt** eines Kernel-Kontroll-Pfads darf auch **nicht** durch eine **Interrupt Service Routine**, die ebenfalls zu den im kritischen Abschnitt manipulierten Daten zugreift, **unterbrochen** werden.
- ◇ Dies kann durch **Sperren** der **Interrupts** (IRQs) während der Ausführung des kritischen Abschnitts sichergestellt werden. Nach **Beendigung** des **kritischen Abschnitts** müssen die **Interrupts** wieder **freigegeben** werden.
- ◇ Jede CPU-Architektur stellt **Assembler-Befehle** zur Verfügung, mit denen Interrupts (genauer IRQs) temporär gesperrt und wieder freigegeben werden können (bei i386: **cli** (Sperren IRQs) und **sti** (Freigabe IRQs)). Üblicherweise ist der Interrupt-Freigabe-Zustand durch ein Flag des Status-Registers (bei i386: **IFLAG** im **EFLAGS**-Register) festgelegt.
- ◇ Im Linux-Quellcode sind diese maschinenabhängigen Befehle in **Inline-Assembler-Code** verwendende **Makros** gekapselt.  
In **include/asm/system.h** sind definiert (Kernel 2.6):
  - ▷ **local\_irq\_disable()** Sperren IRQs (bei i386: **\_\_asm\_\_ \_\_volatile\_\_ ("cli" : : : "memory")**)
  - ▷ **local\_irq\_enable()** Freigabe IRQs (bei i386: **\_\_asm\_\_ \_\_volatile\_\_ ("sti" : : : "memory")**)
- ◇ Da Interrupt Service Routinen prinzipiell auch geschachtelt ausgeführt werden können, kann es sein, dass bei Eintritt in den kritischen Abschnitt Interrupts bereits gesperrt waren. Falls eine derartige Situation auftreten kann, darf nach Beendigung des kritischen Abschnitts nicht einfach eine Freigabe der Interrupts erfolgen, sondern es muss der vorher vorhandene Interrupt-Freigabe-Zustand wiederhergestellt werden.  
Dies ermöglichen die folgenden ebenfalls in **include/asm/system.h** definierten Makros (Kernel 2.6):
  - ▷ **local\_save\_flags(x)** Speichern des Status-Registers (und damit des Interrupt-Freigabe-Zustands) in **x**
  - ▷ **local\_irq\_save(x)** Speichern des Status-Registers (und damit des Interrupt-Freigabe-Zustands) in **x** und Sperren IRQs
  - ▷ **local\_irq\_restore(x)** Wiederherstellen des in **x** abgelegten alten Inhalts des Status-Registers (und damit des alten Interrupt-Freigabe-Zustands)

## Atomare Operationen im Linux-Kernel (1)

### • Allgemeines

- ◇ Eine **atomare Operation** ist **unteilbar**. Sie **kann nicht unterbrochen** werden. Sie wird entweder **vollständig** oder **garnicht** ausgeführt.
- ◇ Ressourcen (hier : Daten), zu denen nur mittels atomarer Operationen zugegriffen werden kann, sind vor **Race Conditions geschützt**.  
Der **kritische Abschnitt** wird auf die **atomare Operation reduziert**. Zwei auf die gleiche Resource (Daten) wirkende atomare Operationen in mehreren Kernel-Kontroll-Pfaden können **niemals gleichzeitig** (bzw verschränkt) sondern immer **nur nacheinander** (sequentiell) ausgeführt werden.  
→ Ein **konkurrierender Zugriff** ist daher **nicht möglich**.
- ◇ Atomare Operationen stellen somit einen sehr **einfachen Synchronisations-Mechanismus** zur Verfügung.
- ◇ Im Kernel sind nur **sehr einfache Operationen** als **atomare Operationen** realisiert (z.B. Inc/Dec eines Zählers). Typischerweise handelt es sich um **read-modify-write**-Operationen. Diese bestehen aus **drei Teiloperationen** :
  - Kopieren eines Werts vom Speicher in ein CPU-Register
  - Verändern des Werts im CPU-Register
  - Zurückschreiben des Werts in den Speicher
 Eine atomare Operation darf zwischen diesen Teiloperationen nicht unterbrochen werden.  
Alle vom Linux-Kernel unterstützten Prozessoren stellen hierfür Möglichkeiten bereit.
- ◇ **Beispiel : Intel x86-Architektur**
  - Realisierung der drei Teiloperationen in einem **einzigen Maschinenbefehl**  
→ Schutz vor Unterbrechung durch Interrupts → Atomarität bei Einzel-CPU-Systemen
  - **lock-Befehl**  
→ Verhinderung der Speicherbus-Freigabe innerhalb des nächsten Befehls → Atomarität bei SMP-Systemen

### • Atomarer Datentyp und atomare "Funktionen"

- ◇ Der Kernel-Code definiert einen **Datentyp `atomic_t`** und die auf diesen anwendbaren **atomaren Operationen**. Dieser Datentyp ist weder zu einem der Standard-Datentypen zuweisungs-kompatibel, noch lassen sich auf ihn die herkömmlichen arithmetischen Operationen ausführen.
- ◇ Da die Implementierung der atomaren Operationen von den Eigenschaften der jeweiligen CPU-Architektur abhängt, sind diese – zusammen mit dem Datentyp `atomic_t` – im **architekturspezifischen Teil** des Linux-Quell-Codes enthalten. Sie sind unter Verwendung von **Assembler-Anweisungen** als **Makros** bzw **inline-Funktionen** definiert.  
→ Headerdatei **`include/asm/atomic.h`**
- ◇ Der Datentyp `atomic_t` ist als Structure-Typ definiert, der einen `int`-Wert kapselt:  

```
typedef struct { volatile int counter; } atomic_t;
```
- ◇ Die wichtigsten definierten **"Funktionen"** zur **atomaren Manipulation** von Variablen dieses Typs sind :

```
int atomic_read(atomic_t* v)
void atomic_set(atomic_t* v, int i)
void atomic_inc(atomic_t* v)
void atomic_dec(atomic_t* v)
void atomic_add(int i, atomic_t* v)
void atomic_sub(int i, atomic_t* v)
int atomic_inc_and_test(atomic_t* v)
int atomic_dec_and_test(atomic_t* v)
int atomic_sub_and_test(int i, atomic_t* v)
int atomic_add_negative(int i, atomic_t* v)
```

```
Lesen des Werts der atomaren Variablen *v
Setzen der atomaren Variablen *v auf den Wert i
Incrementieren der atomaren Variablen *v
Decrementieren der atomaren Variablen *v
Addieren von i zur atomaren Variablen *v
Subtrahieren von i von der atomaren Variablen *v
Incrementieren von *v. Wenn das Ergebnis 0 ist,
Rückgabe von 1, sonst Rückgabe von 0
Decrementieren von *v. Wenn das Ergebnis 0 ist,
Rückgabe von 1, sonst Rückgabe von 0
Subtrahieren von i von *v. Wenn das Ergebnis 0
ist, Rückgabe von 1, sonst Rückgabe von 0
Addieren von i zu *v. Wenn das Ergebnis negativ
ist, Rückgabe von 1, sonst Rückgabe von 0
```

## Atomare Operationen im Linux-Kernel (2)

### • Anmerkungen zur Implementierung

- ◇ Zur **Initialisierung** von **atomic\_t**-Variablen darf nur der folgende – ebenfalls in **include/asm/atomic.h** definierte – Makro verwendet werden (**i** ist der Initialisierungswert) :

```
#define ATOMIC_INIT(i)    { (i) }
```

**Beispiel :** `atomic_t u = ATOMIC_INIT(0);`

- ◇ Die "Funktionen" zum **Lesen und Setzen einer atomaren Variablen** sind als **define-Makros** definiert :

```
#define atomic_read(v)      ((v)->counter)  
#define atomic_set(v,i)    (((v)->counter) = (i))
```

- ◇ Die **übrigen Operationen** sind als **inline-Funktionen** mit **Inline-Assembleranweisungen** definiert.

- ◇ **Beispiele** für die **Intel x86-Architektur** (GNU-Assembler-Syntax für erweiterte Inline-Assembler-Anweisungen):

```
#ifndef CONFIG_SMP  
#define LOCK "lock ; "  
#else  
#define LOCK ""  
#endif  
  
static __inline__ void atomic_add(int i, atomic_t *v)  
{  
    __asm__ __volatile__(  
        LOCK "addl %1,%0"  
        : "=m" (v->counter)  
        : "ir" (i), "m" (v->counter));  
}  
  
static __inline__ int atomic_dec_and_test(atomic_t *v)  
{  
    unsigned char c;  
  
    __asm__ __volatile__(  
        LOCK "decl %0; sete %1"  
        : "=m" (v->counter), "=qm" (c)  
        : "m" (v->counter) : "memory");  
    return c != 0;  
}
```

Bei **SMP-Systemen** verhindert die **lock-Anweisung**, dass der Speicherbus während der Ausführung des die **atomic\_t**-Variable verändernden Befehls (**addl** bzw **decl**) durch einen anderen Prozessor belegt werden kann.

Bei Einzel-CPU-Systemen ist die **lock-Anweisung** nicht erforderlich (→ bedingte Compilierung)

### • Lokale atomare Operationen bei SMP-Systemen

- ◇ Für **SMP-Systeme** ist die Möglichkeit der **lokalen** (auf eine einzelne CPU beschränkten) **Atomarität** implementiert.
- ◇ In der Headerdatei **include/asm/local.h** sind hierfür definiert :
  - ▷ der Datentyp **local\_t** (wie der Datentyp **atomic\_t** als ein eine Ganzzahlvariable kapselnder Structure-Typ)
  - ▷ sowie die folgenden **atomaren Manipulations-"funktionen"** (analog zu den "Funktionen" für den Typ **atomic\_t**)
    - unsigned long **local\_read**(local\_t\* v)
    - void **local\_set**(local\_t\* v, unsigned long i)
    - void **local\_inc**(local\_t\* v)
    - void **local\_dec**(local\_t\* v)
    - void **local\_add**(unsigned long i, local\_t\* v)
    - void **local\_sub**(unsigned long i, local\_t\* v)

## Spinlocks im Linux-Kernel (1)

### • Allgemeines

- ◇ **Locks** werden eingesetzt, wenn der **kritische Abschnitt**, in dem zu bestimmten Daten zugegriffen wird, aus **mehreren Anweisungen** besteht. Sie stellen sicher, dass sich immer **nur ein Kontroll-Pfad** in dem **kritischen Abschnitt** befinden kann.  
Der **Lock** ist an die jeweiligen **Daten gebunden**, er **schützt** diese **vor konkurrierendem Zugriff**.  
Der **erste Kontroll-Pfad**, der in den kritischen Abschnitt eintreten will, **belegt** den **Lock**. Er **hält** diesen während der Ausführung des kritischen Abschnitts und **gibt** ihn nach Verlassen des kritischen Abschnitts wieder **frei**.  
Jeder **andere Kontroll-Pfad**, der ebenfalls in den kritischen Abschnitt eintreten möchte, muss **warten**, bis der Kontroll-Pfad, der den **Lock** besitzt, diesen wieder **freigibt** und er seinerseits den **Lock belegen** kann.
- ◇ **Spinlocks** sind Locks, auf die ein anfordernder Kontroll-Pfad bei Belegung **aktiv warten** muss (*busy waiting*).  
Wird ein angeforderter Spinlock bereits von einem anderen Kontroll-Pfad gehalten, so muss der wartende Kontroll-Pfad in einer **Schleife** ständig **überprüfen** (*spinning*), ob der **Lock freigegeben** ist.
- ◇ Spinlocks sind der im **Linux-Quellcode** am **häufigsten verwendete** Locking-Mechanismus.  
In dem oben beschriebenen Sinne werden sie in **SMP-Systemen** eingesetzt.  
In **Einzel-Prozessor-Systemen mit Kernel-Präemption** bewirken die Operationen zum Belegen und Freigeben eines Spinlocks ein **Deaktivieren** bzw **Aktivieren** der Kernel-Präemption.  
In Einzel-Prozessor-Systemen **ohne Kernel-Präemption** besitzen die Spinlock-Operationen eine **leere Funktionalität**.  
Bei derartigen Systemen besteht die einzige Möglichkeit zur konkurrierenden Abarbeitung von Kernel-Kontroll-Pfaden in der Unterbrechung von Kernel-Code durch Interrupt Service Routinen. In derartigen Situationen wird eine gegebenenfalls notwendige Synchronisation durch das Sperren von Interrupts realisiert.
- ◇ Die meisten Datenstrukturen des Kernels sind mit einem eigenen Spinlock versehen, der gehalten werden muss, wenn kritische Komponenten der Struktur bearbeitet werden.  
Spinlocks werden dabei immer dann sinnvoll eingesetzt, wenn der kritische Abschnitt aus nur relativ wenigen Anweisungen besteht. Da das aktive Warten auf Freigabe CPU-Zeit verbraucht, sollte die Wartezeit nicht länger als die für zwei Prozesswechsel benötigte Zeit betragen.

### • Implementierung

- ◇ Spinlocks sind **architekturabhängig** und deshalb weitgehend in **Assembler** implementiert.
- ◇ In mehreren Headerdateien sind der **Datentyp `spinlock_t`** und **Operationen** auf diesem Typ (als Makros bzw Inline-Assembler-Funktionen) definiert.  
Dabei existieren **unterschiedliche Implementierungen** für **SMP-** und **Einzel-Prozessor-Systeme**.  
Das **Benutzungs-Interface** wird durch die Headerdatei **`include/linux/spinlock.h`** bereitgestellt.  
Diese Headerdatei bindet weitere Headerdateien ein, u.a. auch **`include/linux/spinlock_types.h`** sowie **`include/asm/spinlock.h`**.
- ◇ In der Headerdatei **`include/linux/spinlock_types.h`** sind u.a. definiert :
  - Der Structure-Datentyp **`spinlock_t`**  
Die **SMP-Version** dieses Typs kapselt im wesentlichen eine Variable zur Aufnahme des **Belegungszustands** des Spinlocks (Typ **`unsigned int`**).  
Bei der **Einzel-Prozessor-Version** (und Nicht-Debug-Version) handelt es sich um eine **komponentenlose Structure**.
  - Der Makro **`SPIN_LOCK_UNLOCKED`**  
Er dient zur **Initialisierung** einer neu erzeugten **`spinlock_t`**-Variablen mit dem Zustand "**nicht belegt**".  
In der SMP-Version wird hierdurch die Belegungszustands-Variable auf den Wert **`1`** gesetzt.  
Durch die **Belegung** eines Spinlocks wird deren Wert in **`0`** geändert.
  - Der Makro **`DEFINE_SPINLOCK(x)`** :  
Er dient zur statischen Erzeugung und Initialisierung einer **`spinlock_t`**-Variablen mit dem Namen **`x`**.  
**`#define DEFINE_SPINLOCK(x) spinlock_t x = SPIN_LOCK_UNLOCKED`**

## Spinlocks im Linux-Kernel (2)

### • Spinlock-Operationen

- ◇ Die Headerdatei `include/linux/spinlock.h` definiert zahlreiche **Spinlock-Operationen** als **Makros**. Die Makros greifen auf Hilfs-"Funktionen" zurück, die z. Teil auch als Makros oder als Inline-(Assembler-)Funktionen realisiert sind.
- ◇ Die beiden wichtigsten dieser Operationen sind :
  - ▷ **spin\_lock(\*lock)** Belegen des Locks `*lock`  
In der Version für Einzelprozessor-Systeme mit Kernel-Präemption :  
Deaktivierung der Kernel-Präemption (Aufruf von `preempt_disable()`)
  - ▷ **spin\_unlock(\*lock)** Freigabe des Locks `*lock`  
In der Version für Einzelprozessor-Systeme mit Kernel-Präemption :  
Aktivierung der Kernel-Präemption (Aufruf von `preempt_enable()`)
- ◇ Spin-Locks können auch in Interrupt-Service-Routinen eingesetzt werden.  
Wenn ein Lock, der auch in einer ISR verwendbar sein soll, in einem Kernel-Kontroll-Pfad belegt wird, müssen vor der Anforderung des Locks die lokalen Interrupts gesperrt werden.  
Andernfalls könnte die ISR den bereits belegten Lock anfordern und als Folge in der aktiven Überprüfungsschleife auf seine Freigabe warten.  
Der unterbrochene, den Lock besitzende Code kann aber erst weiterlaufen und den Lock freigeben, wenn die ISR beendet ist. Diese wartet ihrerseits aber auf Freigabe des Locks → Ein **Deadlock** tritt auf.  
Zur Vermeidung derartiger Situationen lassen sich die zwei folgenden Spinlock-Operationen einsetzen :
  - ▷ **spin\_lock\_irqsave(\*lock, flags)**  
Speichern des Status-Registers (und damit des Interrupt-Freigabe-Zustands) in der Variablen `flags`,  
Sperren der IRQs,  
Belegen des Locks `*lock`
  - ▷ **spin\_unlock\_irqrestore(\*lock, flags)**  
Freigabe des Locks `*lock`  
Wiederherstellen des in `flags` gespeicherten alten Werts des Statusregisters (und damit des alten Interrupt-Freigabe-Status)
- ◇ Es existieren noch weitere Spinlock-Operationen

### • Verwendung von Spinlocks

- ◇ **Beispiel :**

```
spinlock_t mylock = SPIN_LOCK_UNLOCKED;

spin_lock(&mylock);

/* kritischer Abschnitt */

spin_unlock(&lock);
```

### • Spezialisierte Spinlocks

- ◇ Im Linux-Kernel finden noch zwei spezialisierte Spinlock-Typen Anwendung :
  - ▷ **Reader/Writer-Locks** (Datentyp `rwlock_t`)  
Locks, die es ermöglichen, dass zwar nur ein Kontroll-Pfad zu den geschützten Daten zugreifen darf, wenn diese verändert werden, dass aber beliebig viele Kontrollpfade zugreifen dürfen, wenn nur gelesen wird.
  - ▷ **Seq-Locks** (Datentyp `seqlock_t`), ab Kernel 2.6  
Implementierung eines einfachen Locking-Mechanismus mittels Verwaltung eines Sequenzzählers für die Benutzung durch viele lesende und wenig schreibende Kontroll-Pfade



## Semaphore im Linux-Kernel

### • Allgemeines

- ◇ **Semaphore** stellen ebenfalls einen **Locking-Mechanismus** zur Verfügung, der es ermöglicht, dass mehrere konkurrierende Kontrollpfade (Prozesse) in **kritischen Abschnitten** zu den gleichen Datenbereichen **synchronisiert zugreifen** können.
- ◇ Semaphore implementieren **Sleeping Locks**.  
Im **Unterschied zu Spinlocks** müssen Kontrollpfade (Prozesse), die einen Lock auf ein bereits belegtes Semaphore erlangen wollen, **nicht aktiv** das **Freiwerden** des Semaphors **erfragen**.  
Vielmehr werden sie in den **Wartezustand** versetzt, aus dem sie **beim Freiwerden aufgeweckt** werden.  
Sie **belegen** also während des Wartens **nicht die CPU**.
- ◇ Semaphore eignen sich daher auch für die **Synchronisation längerer kritischer Abschnitte**

### • Realisierung von Semaphoren mittels Warteschlangen

- ◇ Im LINUX-Kernel werden Semaphore mittels **Warteschlangen** realisiert.

Zur Beschreibung eines Semaphores dient der Structure-Typ **struct semaphore** (definiert in **include/asm/semaphore.h**)

```
struct semaphore {  
    atomic_t count;           /* Belegungskennung */  
    int sleepers              /* Anzahl wartender Prozesse */  
    wait_queue_head_t wait;   /* Warteschlangenkopf */  
};
```

- ◇ Die Belegungskennung (Komponente **count**) gibt im Ausgangszustand an, wieviel Prozesse sich gleichzeitig in dem kritischen Bereich befinden dürfen.  
Meist ist dieser Wert **== 1** (→ **Mutex**, *mutual exclusion*)
- ◇ Ein Semaphore gilt als **belegt**, wenn die Komponente **count** einen Wert **kleiner gleich 0** hat.
- ◇ Ein Prozeß, der den Semaphore belegen will, muß den Wert von **count** überprüfen.  
Ist dieser Wert größer **0**, so kann er den Semaphore belegen.  
Ein Prozeß, der den **Semaphore belegt**, muss **count** um **1 dekrementieren**.  
Jeder **weitere Prozeß**, der den Semaphore ebenfalls belegen möchte, findet ihn als bereits belegt vor und trägt sich in die **Warteschlange** des Semaphors ein.
- ◇ Wenn der den Semaphore besitzende Prozeß diesen wieder **frei** gibt, **incrementiert** er **count** um **1** und **weckt** anschließend die **wartenden Prozesse** wieder auf.  
Der als **erster** wieder **laufende Prozeß** findet den Semaphore als frei vor und **belegt** ihn seinerseits.  
Alle anderen aufgeweckten Prozesse stellen erneut fest, dass der Semaphore belegt ist und tragen sich wieder in die Warteschlange ein.

### • Kernelfunktionen zum Belegen und Freigeben von Semaphore

- ◇ Die folgenden Funktionen sind tatsächlich in Assembler realisiert (in **include/asm/semaphore.h**).  
Nachstehend ist ihre jeweilige prinzipielle Funktionalität in C formuliert.

- ◇ **Belegen eines Semaphors (P-Operation) :**

```
void down(struct semaphore* sem)  
{  
    while (atomic_read(&sem->count) <= 0)  
        sleep_on(&sem->wait);  
    atomic_dec(&sem->count);  
}
```

- ◇ **Freigabe eines Semaphors (V-Operation) :**

```
void up(struct semaphore* sem)  
{  
    atomic_inc(&sem->count);  
    wake_up(&sem->wait);  
}
```

## Interruptbearbeitung in LINUX

### • Allgemeines

- ◇ Zur Kommunikation der Hardware mit dem Betriebssystem werden – vor allem in einem Multitasking-BS – meist **Interrupts** eingesetzt.
- ◇ Wenn ein Gerät eine – i.a. durch einen Gerätetreiber initiierte – Operation beendet hat, sendet es eine **Interruptanforderung (Interrupt Request, IRQ, Hardware Interrupt)** an die CPU.  
Falls der entsprechende Interrupt freigegeben ist, reagiert die CPU mit der Unterbrechung des gerade in Abarbeitung befindlichen Codes und dem Sprung zu einer **Interrupt Service Routine (ISR, Interrupt Handler)**.  
Innerhalb der ISR erfolgt dann die weitere Kommunikation mit dem Gerät (z.B. Übertragung von Daten).  
Nach Beendigung der ISR wird zu dem unterbrochenen Code zurückgekehrt und dieser wird weiter abgearbeitet.
- ◇ Die – i.a. durch den **Init-Code** eines **Gerätetreibers** installierte – **Interrupt Service Routine** ist Bestandteil des Betriebssystems und wird im **System- (Kernel-) Mode** abgearbeitet.
- ◇ Üblicherweise werden die Interruptanforderungen nicht direkt, sondern über einen **Interrupt Controller** an die CPU gesandt.  
Die verschiedenen Eingangsleitungen eines Interrupt Controllers legen die jeweilige Nummer des IRQs (**IRQ-Nr.**) fest.  
Im **Controller** ist es u.a. möglich, **bestimmte IRQs zu sperren** bzw freizugeben.  
Gespernte IRQs werden nicht an die CPU weitergegeben.
- ◇ In der CPU selbst können **alle** – maskierbaren – **Interrupts** gemeinsam gesperrt werden  
(bei der Intel-x86-Architektur : Befehl für Sperren **CLI**, Befehl für Freigabe **STI**).
- ◇ Zu **jeder IRQ-Nr.** gehört eine **eigene Interrupt Service Routine**.
- ◇ Der genaue Aufbau und weitere Einzelheiten des durch die Hardware festgelegten Interrupt-Systems sind naturgemäß stark **architekturabhängig**.

### • "Teilen" eines Interrupts (*Interrupt Sharing*)

- ◇ Im Idealfall sollte jedem im System vorhandene Gerät eine **eigene IRQ-Nr** zugeordnet sein.  
Wenn die Anzahl der Geräte die Anzahl der zur Verfügung stehenden IRQ-Nummern übersteigt, ist dies nicht möglich.  
In einem derartigen Fall müssen sich mehrere Geräte dieselbe IRQ-Nr. teilen → **Interrupt Sharing**.
- ◇ Innerhalb der Interrupt Service Routine für die entsprechende IRQ-Nr. muß dann festgestellt werden, welches Gerät den IRQ erzeugt hat, damit die richtige Bearbeitung stattfinden kann.
- ◇ Unter LINUX wird *Interrupt Sharing* durch die Installation mehrerer **gerätespezifischer Interruptbearbeitungs-routinen**, die innerhalb der ISR aufgerufen werden, realisiert.

### • Prinzipieller Aufbau einer Interruptbearbeitung unter LINUX

- ◇ Die Interruptbearbeitung unter LINUX erfolgt nicht mittels einer geschlossen-homogen aufgebauten ISR sondern in einer **modularen** aus – im wesentlichen **fünf – Schichten** bestehenden Struktur :
  - ▷ **"allgemeine" ISR**
  - ▷ **Interruptbearbeitungs-Steuer-Routine (Interrupt Flow Handler)**
  - ▷ **Bearbeitungsroutinen** für den **Interrupt Controller** (z.B. Acknowledge-Routine, Beendigungs-Routine)
  - ▷ **gerätespezifische Interruptbearbeitungsroutine(n)** (früher als *Top Half Handler* bezeichnet)
  - ▷ gegebenenfalls Routine zur **verzögerten** (späteren) **Ausführung** weniger wichtiger Aufgaben  
(früher als *Bottom Half Handler* bezeichnet, ab Kernel 2.4 durch – anders implementierte – **Tasklets** ersetzt)
- ◇ Ein – nicht gesperrter – IRQ bewirkt über einen der jeweiligen IRQ-Nr. zugeordneten Einsprungpunkt in der Interrupt-Vektor-Tabelle den Aufruf der **"allgemeinen" ISR**.  
Diese ruft über eine weitere Funktion den **Interrupt Flow Handler** auf.  
Der Flow Handler steuert den Ablauf der Interruptbearbeitung. Er wickelt die Kommunikation mit dem Interrupt Controller durch Aufruf entsprechender **controller-spezifischer Bearbeitungsroutinen** ab und veranlasst – über eine Zwischenfunktion den Aufruf der **gerätespezifischen Interruptbearbeitungsroutine(n)**  
In dieser/diesen wird im wesentlichen die **zeitkritische Kommunikation mit der Hardware** abgewickelt. Hierbei können **weitere Interrupts gesperrt** sein. Deshalb soll(en) diese Routine(n) **so kurz und schnell wie möglich** sein.
- ◇ Später – zu einem "sicheren" Zeitpunkt – wird ein eventuelles **Tasklet** ausgeführt.

## Ergänzungen zur Interruptbearbeitung in LINUX

### • IRQs und Interrupt-Nummern bei der IA32-Architektur

- ◇ Alle – maskierbaren – IRQs werden vom Interrupt Controller auf einen einzigen **INTR-Eingang** der CPU gegeben.
- ◇ Die CPU unterscheidet die verschiedenen Interrupts mittels einer **Interrupt-Nr** (Interrupt-Vector, *type vector*).
- ◇ IRQ-Nr. und Interrupt-Nr sind nicht identisch aber **eindeutig** einander **zugeordnet**.  
Im Interrupt Controller erfolgt eine Abbildung der jeweiligen IRQ-Nr auf die zugehörige Interrupt-Nr.  
Es gilt die Beziehung : **Interrupt-Nr = IRQ-Nr + 32**.  
In der IA32-Architektur mit dem klassischen 8256A Interrupt Controller werden die **IRQ-Nummern 0 ... 15** verwendet, diesen entsprechen die **Interrupt-Nummern 32 ... 47**.  
In moderneren IA32-Systemen mit dem IO-APIC stehen **IRQ-Nummern** bis zu **223** zur Verfügung.
- ◇ Im **Linux-Quellcode** wird mit **IRQ-Nummern** gearbeitet.

### • Weitere Interruptarten bei der IA32-Architektur

- ◇ Neben den mittels **IRQs** erzeugten **externen – maskierbaren – Interrupts** gehören zum Interrupt-System der **Intel-Prozessoren** :
  - ▷ ein – ebenfalls externer – **nicht-maskierbarer Interrupt (NMI)**, der über eine Interruptanforderung auf einem eigenen CPU-Eingang (NMI) ausgelöst wird.
  - ▷ **interne** durch die CPU selbst erzeugte Interrupts (→ **Exceptions**) unterschiedlichen Typs.  
Die CPU löst derartige Exceptions beim Erkennen bestimmter **Fehlersituationen** aus (z.B. Division durch 0, *Page Fault* usw)
  - ▷ durch den **Befehl INT xx** erzeugte **Software-Interrupts** (z.B. `INT 0x80` für System Calls)
- ◇ Auch diese Interrupts werden innerhalb der CPU durch **Interrupt-Nummern** unterschieden.  
Alle Interrupt-Nummern gehören zu dem gleichen **einheitlichen Nummern-Raum**, der den Bereich **0 ... 255** umfasst.  
Für sie müssen ebenfalls geeignete **Interrupt Service Routinen** installiert werden.  
Die **ISR für Exceptions** (*exception handler*) senden i.a. (Ausnahme gegebenenfalls *page fault*) ein **Signal** an den die Fehlersituation erzeugenden Prozess.
- ◇ Die Startadressen aller Interrupt Service Routinen müssen in einer von der CPU hardwaremässig verwendeten **Interrupt Deskriptor Tabelle** zusammengefasst sein.

### • SoftIRQs

- ◇ Ein ab dem Kernel 2.4 in Linux implementierter Mechanismus zur **verzögerten Ausführung** von **Kernel-Aktivitäten**.
- ◇ Die Implementierung und Funktionsweise von SoftIRQs **orientiert** sich an der Bearbeitung von Interrupts.  
SoftIRQs sind aber **keine Interrupts**.
- ◇ Auch SoftIRQs werden durch **Nummern** unterschieden. Für die im Kernel standardmässig eingesetzten SoftIRQ-Nummern sind zusätzlich durch einen Aufzählungstyp **symbolische Namen** festgelegt.  
Z.Zt (Kernel 2.6) umfasst der SoftIRQ-Nummern-Raum den Bereich **0 ... 31**, die Nummern **0 ... 5** besitzen symbolische Namen.
- ◇ Die als Reaktion auf die Auslösung eines SoftIRQs auszuführende Funktion (**SoftIRQ-Handler**) muss **installiert** werden. Dies erfolgt durch Setzen der der SoftIRQ-Nummer entsprechenden Komponente in einer **SoftIRQ-Vektor-Tabelle** (`softirq_vec[32]`) mittels der Funktion `open_softirq()` (definiert in *kernel/softirq.c*)
- ◇ Ein **SoftIRQ** wird **ausgelöst** durch den Aufruf der Funktion `raise_softirq(int nr)`.  
Diese Funktion setzt das der SoftIRQ-Nr entsprechende Bit in einer Markierungs-Variablen und aktiviert den als Kernel-Thread laufenden **SoftIRQ-Dämon ksoftirqd** (durch Setzen dessen Zustands auf ablaufbereit (`TASK_RUNNING`)).
- ◇ Der SoftIRQ-Dämon führt die **SoftIRQ-Handler**, die durch die gesetzten Bits in der Markierungs-Variablen gekennzeichnet sind, **asynchron** zur restlichen Kernel-Aktivität aus ("Bedienung" der SoftIRQs).  
(Funktion `do_softirq()`, definiert in *kernel/softirq.c*).
- ◇ **Tasklets** (s. oben) sind als **spezielle SoftIRQs** implementiert

## Datenstrukturen zur Interruptbearbeitung in LINUX (1)

- **Structure-Typ struct irqaction** (definiert in `include/linux/interrupt.h`)

Dieser Typ dient zur Beschreibung von Informationen über eine für einen bestimmten IRQ auszuführende **gerätespezifische Interruptbearbeitungsroutine** (*Top Half Handler*).

Durch Bildung einer linearen vorwärts verketteten Liste von Variablen dieses Typs lassen sich die Bearbeitungsroutinen, die sich einen Interrupt teilen (*Interrupt Sharing*), zusammenfassen.

```
typedef irqreturn_t (*irq_handler_t)(int, void *);

struct irqaction {
    irq_handler_t handler;
    unsigned long flags;
    cpumask_t mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
    int irq;
    struct proc_dir_entry *dir;
};
```

### Bedeutung einiger Structure-Komponenten :

**handler** : Startadresse der Interruptbearbeitungsroutine  
Der Rückgabotyp dieser Routinen ist definiert in `include/linux/irqreturn.h` :  
`typedef int irqreturn_t;`

**flags** : Verwendungsflags der Interruptbearbeitungsroutine (def. in `include/linux/interrupt.h`), u.a. :  
**IRQF\_DISABLED** : wenn gesetzt, darf Routine nicht durch weitere Interrupts unterbrochen werden  
(*"fast" interrupt handler*)  
**IRQF\_SHARED** : wenn gesetzt, Interruptbearbeitungsroutine erlaubt *Interrupt Sharing*

**name** : Name des Gerätes, das den die Bearbeitungsroutine aktivierenden Interrupt erzeugt hat

**dev\_id** : spezifische ID des Gerätetyps, Achtung : darf nicht als Pointer verwendet werden !

**next** : Pointer auf nächste struct irqaction -Variable (bei *Interrupt Sharing*),  
bzw **NULL**, falls keine weitere Routine existiert

- **Structure-Typ struct irq\_chip** (definiert in `include/linux/irq.h`)

Dieser Typ abstrahiert einen Interrupt Controller.

Im wesentlichen enthält er Pointer auf Funktionen zur Realisierung von controller-spezifischen Operationen

```
struct irq_chip {
    const char *name;
    unsigned int (*startup)(unsigned int irq);
    void (*shutdown)(unsigned int irq);
    void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
    void (*ack)(unsigned int irq);
    void (*mask)(unsigned int irq);
    void (*mask_ack)(unsigned int irq);
    void (*unmask)(unsigned int irq);
    void (*eoi)(unsigned int irq);
    void (*end)(unsigned int irq);
    /* Pointer auf weitere Funktionen */
};
```

### Bedeutung einiger Structure-Komponenten :

**name** : Name des Controllers (bei IA32-Architektur z.B. "XT-PIC" oder "IO-APIC")

**startup** und **enable** : Freigabe des IRQs mit Nr. `irq` (Parameter) im Controller

**shutdown** und **disable** : Sperren des IRQs mit Nr. `irq` (Parameter) im Controller

**ack** : Acknowledge-Routine

**end** : Beendigungs-Routine

## Datenstrukturen zur Interruptbearbeitung in LINUX (2)

- **Structure-Typ struct irq\_desc** (definiert in `include/linux/irq.h`)

Dieser Typ faßt die Informationen über einen IRQ einer bestimmten Nr. (IRQ-Typ) zusammen

```
struct irq_desc {
    irq_flow_handler_t  handle_irq;
    struct irq_chip      *chip;
    struct msi_desc      *msi_desc;
    void                *handler_data;
    void                *chip_data;
    struct irqaction      *action          /* IRQ action list */
    unsigned int         status;           /* IRQ status */
    unsigned int         depth;            /* nested irq disables */
    unsigned int         wake_depth;       /* nested wake enables */
    unsigned int         irq_count;        /* For detecting broken IRQs */
    unsigned int         irqs_unhandled;   /* For spurious unhandled IRQs */
    unsigned long        last_unhandled;   /* Aging timer for unhandled count */
    spinlock_t          lock;
    /* einige weitere von #define-Konstanten abhängige Komponenten */
    const char           *name;            /* Flow Handler name for /proc/interrupts output */
};
```

### Bedeutung einiger Structure-Komponenten :

**handle-irq** : Pointer auf Interruptbearbeitungs-Steuer-Routine (*Flow Handler*), wenn == 0 : `__do_IRQ0`

**chip** : Pointer auf die `struct irq_chip`-Variable, die den Interrupt Controller beschreibt

**action** : Pointer auf die Liste der die Bearbeitungsrouinen beschreibenden `struct irqaction`-Variablen

**status** : Status des IRQs, beschrieben durch eines oder eine Kombination mehrerer der folgenden in `include/linux/irq.h` definierten Flags.

Die Flags dürfen nur durch die controller-spezifischen Routinen verändert werden, die auch die korrespondierenden hardwaremaessigen Einstellungen (z.B. im Interrupt Controller) vornehmen müssen.

U.a. existieren die folgenden Flags :

<b>IRQ_INPROGRESS</b>	IRQ wird bereits von einer CPU bedient (notwendig für SMP-Systeme)
<b>IRQ_DISABLED</b>	IRQ ist gesperrt
<b>IRQ_PENDING</b>	IRQ steht an, korespondierender Handler wird noch nicht ausgeführt
<b>IRQ_REPLAY</b>	IRQ wurde gesperrt, während ein noch nicht bedienter Interrupt ansteht
<b>IRQ_AUTODETECT</b>	verwendet für automatische Detektion und Konfiguration von IRQs
<b>IRQ_WAITING</b>	anstehender IRQ noch nicht erkannt (für automat. Detektion)
<b>IRQ_LEVEL</b>	IRQ ist zustands-getriggert
<b>IRQ_MASKED</b>	IRQ ist maskiert, sollte nicht wieder erkannt werden
<b>IRQ_PER_CPU</b>	IRQ kann nur auf einer einzigen CPU auftreten (nur für SMP-Systeme relevant)
<b>IRQ_NOREQUEST</b>	IRQ für Interrupt-Sharing verwendbar, keine exklusive Zuordnung zu einem Gerät

**depth** : Anzahl verschachtelter Sperrungen dieses Interrupts

Wenn ==0 : Interrupt ist nicht gesperrt, er darf auch hardwaremässig freigegeben werden.

**lock** : Spinlock-Variable (nur für SMP-Architekturen relevant)

- **Interruptbearbeitungs-Steuer-Routine (*Flow Handler*)**

◇ Sie dient zur Steuerung des Ablaufs der Interruptbearbeitung (insbesondere der Interaktion mit dem Interrupt Controller) unter Berücksichtigung des jeweiligen Interrupt-Typs (z.B. flankengetriggert, pegelgetriggert)

◇ **Funktionspointer-Typ für *Flow Handler*** (def. in `include/linux/irq.h`):

```
typedef void (*irq_flow_handler_t)(unsigned int irq, struct irq_desc* desc)
```

◇ Im Kernel sind einige **Default *Flow Handler*** implementiert (def. in `kernel/irq/chip.c`):

- ▷ `handle_edge_irq` für flankengetriggerte Interrupts
- ▷ `handle_level_irq` für pegelgesteuerte Interrupts
- ▷ `handle_fasteoi_irq` für moderne Interrupt Controller, die kaum Interaktion benötigen (nur `chip->eoi()`)
- ▷ `handle_simple_irq` für einfache Interrupts, die keinerlei Steuerung der Hardware benötigen
- ▷ `handle_percpu_irq` für Interrupts die nur von einer bestimmten CPU bedient werden können (kein Locking !)

## Datenstrukturen zur Interruptbearbeitung in LINUX (3)

- **Interrupt-Beschreibungstabelle `irq_desc`** (definiert in `kernel/irq/handle.c`)

Tabelle (Array), die IRQ-Beschreibungsinformationen für jede mögliche IRQ-Nr. enthält.

Für jede mögliche IRQ-Nr. existiert eine Array-Komponente (vom Typ `struct irq_desc`):

```
struct irq_desc irq_desc[NR_IRQS];
```

`NR_IRQS` ist die im System maximal mögliche Anzahl von IRQs .

Für die x86-Architektur gilt :

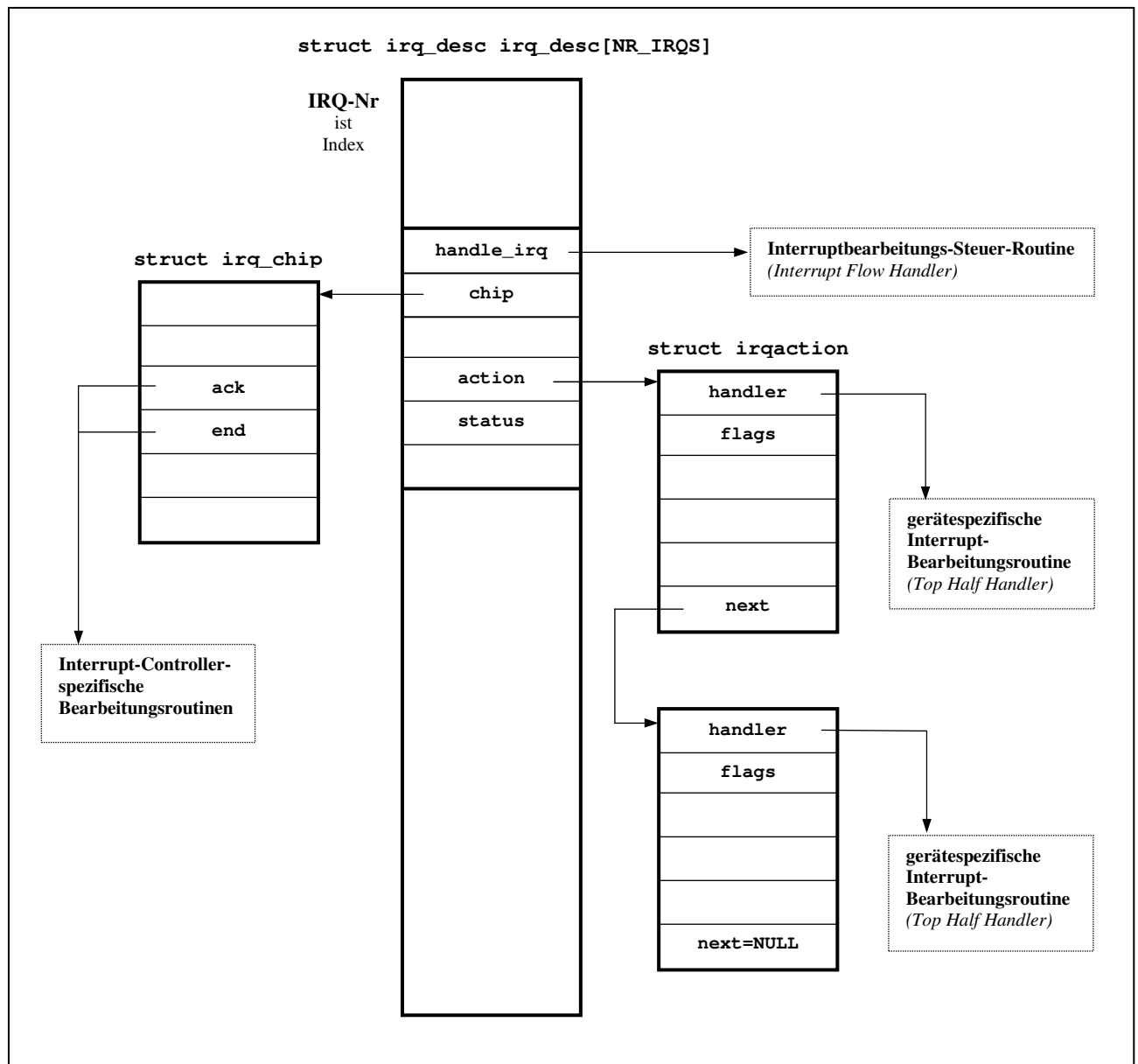
```
#define NR_IRQS 224
```

Die **IRQ-Nr.** dient als **Tabellen-Auswahl-Index**.

- **Anmerkung :**

Die Pseudodatei `"/proc/interrupts"` enthält eine **Auflistung** der **im System belegten Interrupts** (IRQs) mit **Angabe** (Name) des verwendeten **Interrupt Controllers**, an den der Name des Flow Handlers angehängt wird.

- **Überblick**



## Gerätespezifische Interruptbearbeitungsroutinen in LINUX

- **Prototyp einer gerätespezifischen Interruptbearbeitungsroutine**

```
irqreturn_t example_interrupt(int irq, void* dev_id);
```

**Bedeutung der Parameter :**

**irq** : IRQ-Nr.

**dev\_id** : spezifische ID des Gerätetyps

**Funktionswert** : Der Typ **irqreturn\_t** definiert in *include/linux/irqreturn.h* als `int` :  
`typedef int irqreturn_t;`

Zulässige Werte :

- **IRQ\_HANDLED** (==1), wenn Routine IRQ bearbeitet hat
- **IRQ\_NONE** (==0), wenn Routine für den IRQ nicht zuständig war

- **Installation einer gerätespezifischen Interruptbearbeitungsroutine** (z.B. durch einen Gerätetreiber)

mittels der in *include/linux/interrupt.h* deklarierten und in *kernel/irq/manage.c* definierten Kernel-Funktion

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long irqflags,  
               const char * devname, void *dev_id);
```

**Bedeutung der Parameter :**

**irq** : IRQ-Nr, für die die Routine installiert werden soll

**handler** : Startadresse der zu installierenden Interruptbearbeitungsroutine

**irqflags** : Verwendungsflags der Interruptbearbeitungsroutine (s. *struct irqaction*)

**devname** : Name des Geräts, für das die Routine zuständig ist

**dev\_id** : spezifische ID des Gerätetyps

Diese Funktion legt unter Verwendung der übergebenen Parameter eine neue *struct irqaction* -Variable an und hängt diese mittels der Funktion *setup\_irq()* an das Ende der *struct irqaction* -Liste für den IRQ mit der Nummer *irq* an.

Das Anhängen an die *struct irqaction* -Liste ist nur möglich, wenn es sich bei der zu installierenden Routine um die erste für diesen IRQ handelt oder wenn für alle bereits installierten Routinen das *IRQF\_SHARED*-Flag gesetzt ist. Im letzteren Fall muß auch für die neu zu installierende Routine das *IRQF\_SHARED*-Flag (Parameter *irqflags*) gesetzt sein. Die Komponente *dev\_id* muß für alle Routinen in einer Liste (gleiche IRQ-Nr.) unterschiedlich sein.

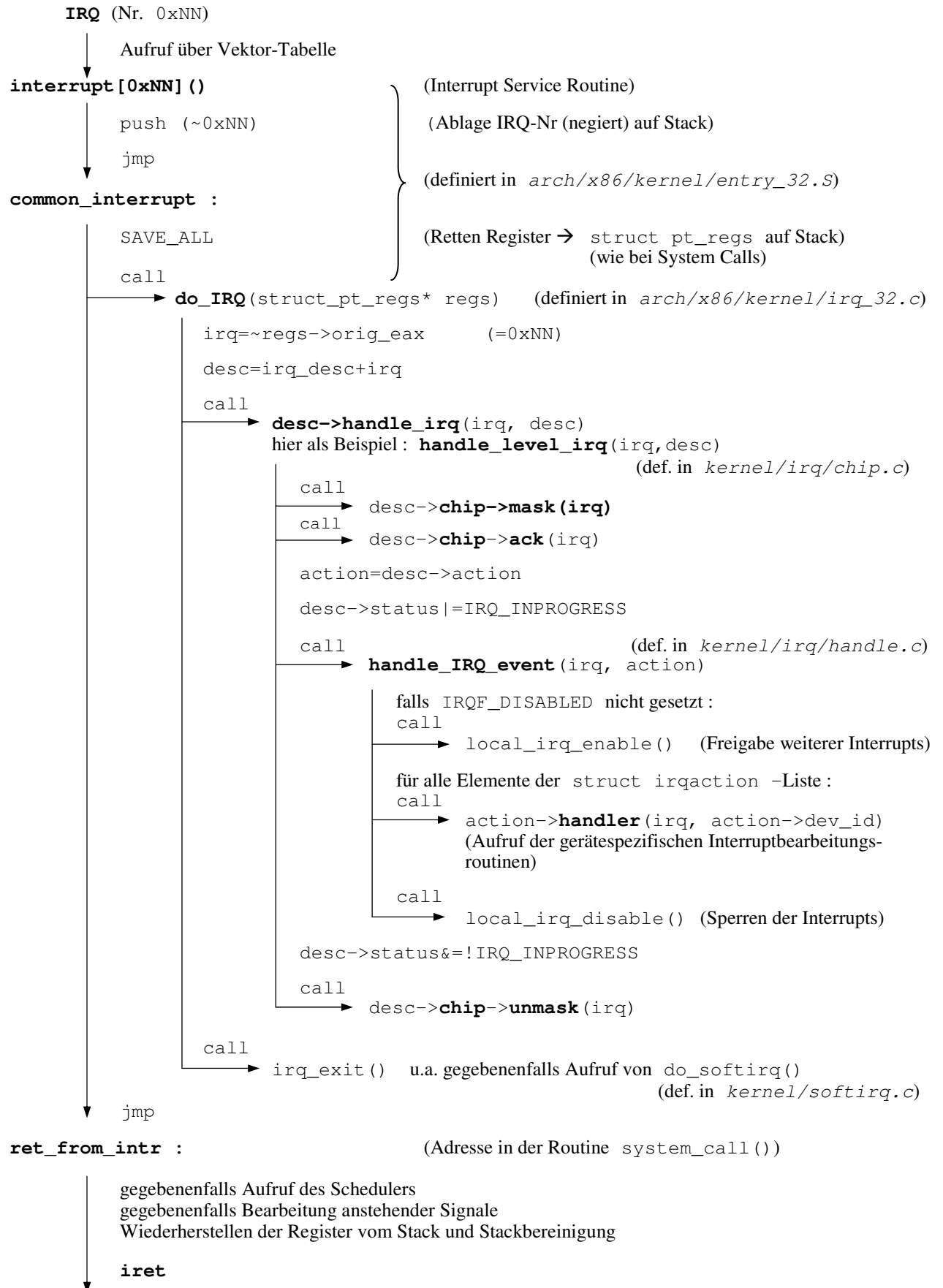
**Funktionswert** : - 0, bei erfolgreicher Installation der Routine,  
- ein Wert != 0, bei Mißerfolg

- **Registernachbildung auf dem Stack (Structure-Typ *struct pt\_regs*)**

(def. in *include/asm-x86/ptrace.h*)

```
struct pt_regs {  
    long bx;  
    long cx;  
    long dx;  
    long si;  
    long di;  
    long bp;  
    long ax;  
    int ds;  
    int es;  
    int fs;  
    long orig_ax;  
    long ip;  
    int cs;  
    long flags;  
    long sp;  
    int ss;  
};
```

## Prinzipieller Ablauf einer IRQ-Bedienung in LINUX





## **Bottom Half Handler in LINUX (1)**

### • Allgemeines

Mit einer Interruptbearbeitung in Zusammenhang stehende Aktivitäten, die nicht unbedingt sofort erledigt werden müssen, werden in sogenannte **Bottom Half Handler** ausgelagert und später – außerhalb der eigentlichen Interruptbearbeitung – ausgeführt.

### • Gründe für die Einführung von Bottom Halfs

- ▷ Verringerung der Interrupt-Latenz-Zeit.  
Die Bearbeitung von IRQs, die nicht durch andere Interrupts unterbrochen werden dürfen ("schnelle" Interrupts), muß so schnell wie möglich erfolgen, damit weitere IRQs nicht zu lange warten müssen.
- ▷ Der aktuell bearbeitete IRQ wird – durch die controller-spezifische ACK-Routine – während der Abarbeitung der gerätespezifischen Interruptbearbeitungsroutine im Interrupt Controller gesperrt.  
Diese Sperrung sollte so kurz wie möglich dauern, damit nicht IRQs der gleichen Nummer verloren gehen.
- ▷ Eine Reihe von Aktivitäten müssen nicht bei jedem Auftritt des entsprechenden IRQs ausgeführt werden.  
Diese werden in einen **Bottom Half Handler** ausgelagert.  
Der **Top Half Handler** (gerätespezifische Bearbeitungsroutine) entscheidet, ob und wann der zugehörige **Bottom Half Handler** ausgeführt werden soll. Soll er ausgeführt werden, muß er vom **Top Half Handler** **aktiviert** werden.

**Achtung :** Die folgenden Ausführungen beziehen sich auf den **Kernel 2.2**  
Ab dem **Kernel 2.4** hat sich das Konzept für **Bottom Halfs** geändert.

### • Ausführung der Bottom Half Handler (Kernel 2.2)

Der Aufruf der **Bottom Half Handler** erfolgt durch die Kernel-Funktion

```
void do_bottom_half(void);
```

Diese Funktion startet – über den Aufruf einer weiteren Funktion – nacheinander alle aktivierten und freigegebenen **Bottom Half Handler**.

Noch vor ihrer Ausführung werden diese wieder als deaktiv markiert.

Der **Aufruf** der Funktion `do_bottom_half` erfolgt in den folgenden drei Situationen :

- ▶ innerhalb des **Schedulers** vor Auswahl der nächsten laufenden Task
- ▶ vor der **Rückkehr** aus einem **System Call**
- ▶ unmittelbar vor der **Beendigung** der im Rahmen der IRQ-Bedienung ausgeführten Routine `do_IRQ()`, also **nach jedem IRQ** (dies wird in zukünftigen LINUX-Versionen voraussichtlich geändert)

**Bottom Half Handler** werden **nur dann ausgeführt**,

- wenn **nicht gerade** – der gleiche oder ein anderer – **Bottom Half Handler läuft**  
(→ **Bottom Half Handler** dürfen **nicht durch Bottom Half Handler unterbrochen** werden)
- wenn **nicht gerade** ein **IRQ bedient** wird (genauer : eine gerätespezifische Interruptbearbeitungsroutine ausgeführt wird).

Eine derartige Situation kann auftreten, wenn die Bedienung eines IRQs durch einen erneuten IRQ unterbrochen wird.

Die Abarbeitung *eines Bottom Half Handlers* **darf** jederzeit **durch** einen **Interrupt unterbrochen** werden

## **Bottom Half Handler in LINUX (2)**

- **Datenstrukturen zur Verwaltung der *Bottom Half Handler* (Kernel 2.2)**

- ◇ ***Bottom Half* Aktivitäts-"Register"** (definiert in `kernel/softirq.c`)

```
unsigned long bh_active;
```

Jedem *Bottom Half Handler* ist ein **Bit** zugeordnet. → maximal **32** verschiedene *Bottom Half Handler* sind möglich.  
Das einem bestimmten *Bottom Half Handler* zugeordnete Bit wird gesetzt (i.a. vom *Top Half Handler*), wenn dieser Handler ausgeführt werden soll. → Der Handler wird **aktiviert**.  
Nach seiner Ausführung wird das Bit wieder rückgesetzt.

- ◇ ***Bottom Half* Masken-"Register"** (definiert in `kernel/softirq.c`)

```
unsigned long bh_mask;
```

Auch in diesem "Register" ist **jedem *Bottom Half Handler* ein Bit zugeordnet**.  
Unabhängig von seiner Aktivierung im Aktivitäts-"Register" kann die **Ausführung** jedes *Bottom Half Handlers* durch **Rücksetzen** des ihm zugeordneten Bits im Masken-"Register" **gesperrt** werden.  
Durch Setzen des entsprechenden Bits wird die Ausführung des Handlers freigegeben.  
→ die **bitweise UND**-Verknüpfung von **bh\_active** und **bh\_mask** legt fest, ob ein *Bottom Half Handler* **ausgeführt** werden soll.

- ◇ ***Bottom Half Handler* Pointer-Tabelle** (definiert in `kernel/softirq.c`)

```
void (*bh_base[32])(void);
```

Jedem möglichen *Bottom Half Handler* ist eine Array-Komponente zugeordnet.  
Für jeden installierten *Bottom Half Handler* ist seine Anfangsadresse (Funktions-Pointer) eingetragen.  
Für nicht installierte Handler enthält die entsprechende Komponente den `NULL`-Pointer.

- ◇ **Namenloser Aufzählungstyp zur Festlegung symbolischer Namen für *Bottom Half Handler***  
(definiert in `include/linux/interrupt.h`)

In den Verwaltungs-"Registern" und in der Pointer-Tabelle werden *Bottom Half Handler* über eine **Nummer** (Bit-Nr, Index) ausgewählt.

Die Nummern `0` bis `16` sind defaultmäßig bestimmten Handlern zugeordnet.

Zur **symbolischen Referierung** dieser Handler über einen Namen dient der folgende **Aufzählungstyp** :

```
enum {  
    TIMER_BH = 0,  
    CONSOLE_BH,  
    TQUEUE_BH,  
    DIGI_BH,  
    SERIAL_BH,  
    RISCOM8_BH,  
    SPECIALIX_BH,  
    AURORA_BH,  
    ESP_BH,  
    NET_BH,  
    SCSI_BH,  
    IMMEDIATE_BH,  
    KEYBOARD_BH,  
    CYCLADES_BH,  
    CM206_BH,  
    JS_BH,  
    MACSERIAL_BH,  
    ISICOM_BH  
};
```

Handlern, die **öfter aufgerufen** werden, ist eine **niedrigere Nummer** zugeordnet

### **Bottom Half Handler in LINUX (3)**

- **Kernelfunktionen zur Verwaltung der *Bottom Half Handle* (Kernel 2.2) :**

- ◇ **Installation eines *Bottom Half Handlers*** (definiert in `include/asm-i386/softirq.h`) :

```
inline void init_bh(int nr, void(*routine)(void));
```

Bedeutung der Parameter :

**nr** : Nummer des zu installierenden *Bottom Half Handlers*  
**routine** : Anfangsadresse des zu installierenden *Bottom Half Handlers*

Die Funktion schreibt die Adresse `routine` in die Komponente mit dem Index `nr` des Pointer-Arrays **bh\_base**.

- ◇ **Deinstallation eines *Bottom Half Handlers*** (definiert in `include/asm-i386/softirq.h`) :

```
inline void remove_bh(int nr);
```

Bedeutung des Parameters :

**nr** : Nummer des zu deinstallierenden *Bottom Half Handlers*

Die Funktion entfernt den Eintrag in der Komponente mit dem Index `nr` aus dem Pointer-Array **bh\_base**.

- ◇ **Aktivieren eines *Bottom Half Handlers*** (definiert in `include/asm-i386/softirq.h`) :

```
inline void mark_bh(int nr);
```

Bedeutung des Parameters :

**nr** : Nummer des zu aktivierenden *Bottom Half Handlers*

Die Funktion setzt im Aktivitäts-"Register" **bh\_active** das Bit mit der Nummer `nr`.

- ◇ **Sperren eines *Bottom Half Handlers*** (definiert in `include/asm-i386/softirq.h`) :

```
inline void disable_bh(int nr);
```

Bedeutung des Parameters :

**nr** : Nummer des zu sperrenden *Bottom Half Handlers*

Die Funktion setzt im Masken-"Register" **bh\_mask** das Bit mit der Nummer `nr` zurück

- ◇ **Freigeben eines *Bottom Half Handlers*** (definiert in `include/asm-i386/softirq.h`) :

```
inline void enable_bh(int nr);
```

Bedeutung des Parameters :

**nr** : Nummer des freizugebenden *Bottom Half Handlers*

Die Funktion setzt im Masken-"Register" **bh\_mask** das Bit mit der Nummer `nr`.

## Beispiel für die Installation einer Interruptbearbeitung in LINUX

- **Beispiel : Installation der Interruptbearbeitung für den Keyboard-Treiber (Kernel 2.2)**

Die Installation sowohl des *Top Half Handlers* als auch des *Bottom Half Handlers* erfolgt in der **init ()** -Funktion des Treibers.

```
/* Top Half Handler (geraetespezifische Interruptbearbeitungsroutine) -- */
static void keyboard_interrupt(int irq, void* dev_id, struct pt_regs* regs)
{
    /* ... */

    mark_bh(KEYBOARD_BH);          /* Aktivieren der Bottom Half Handlers */

    /* ... */
}

/* Bottom Half Handler ----- */
static void kbd_bh(void)
{
    unsigned char leds = getleds();

    if (leds != ledstate) {
        ledstate = leds;
        kbd_leds(leds);
    }
}

/* Installationsfunktion des Keyboard-Treibers ----- */
int kbd_init(void)
{
    /* ... */

    request_irq(KEYBOARD_IRQ, keyboard_interrupt, 0, "keyboard", NULL);

    /* ... */

    init_bh(KEYBOARD_BH, kbd_bh);
    mark_bh(KEYBOARD_BH);

    /* ... */
}
```

# **Betriebssysteme**

## **Kapitel 7**

### **7. Prozessverwaltung in LINUX**

- 7.1. Allgemeines
- 7.2. Prozess-Zustände
- 7.3. Prozess-Identifikation
- 7.4. Datenstrukturen zur Prozessverwaltung
- 7.5. Erzeugung und Beendigung von Prozessen
- 7.6. Scheduling

## Prozesse unter LINUX – Allgemeines

### • Allgemeine Eigenschaften

- ◇ LINUX ist ein **Multitasking-Betriebssystem**, das **preemptives Multitasking** einsetzt.  
→ **Mehrere Prozesse** können **gleichzeitig** existieren und (quasi-)parallel abgearbeitet werden.  
Jeder Prozess läuft als **eigenständige Scheduling-Einheit** in einem **eigenen virtuellen Adressraum** und kann mit anderen Prozessen nur über vom Betriebssystem-Kernel zur Verfügung gestellten Mechanismen interagieren (→ **Interprozesskommunikation**).  
Einen gewissen Sonderfall stellen **Threads** dar. Dies sind Prozesse, die sich den **gleichen Arbeitsspeicher** (außer Stack) teilen. LINUX behandelt Threads (leichtgewichtige Prozesse) und "normale" Prozesse (schwergewichtige Prozesse) gleichartig. Beide werden unter dem Begriff **Prozess** oder **Task** zusammengefaßt.
- ◇ Auf **einer CPU** kann zu einem bestimmten Zeitpunkt immer **nur ein Prozess** laufen.  
Der **aktuell laufende** Prozess **gibt** die **CPU frei**,
  - wenn er auf ein **Ereignis warten** muß oder
  - wenn ihm diese **zwangsweise entzogen** wird (z.B. Ablauf der Zeitscheibe) oder
  - **freiwillig**
- ◇ Die Auswahl des **nächsten laufenden** Prozesses erfolgt durch den **Scheduler**.

### • Scheduler

- ◇ Aufgabe des **Schedulers** ist es, die CPU gerecht zwischen den einzelnen ablaufbereiten Prozessen aufzuteilen.  
Die **Effizienz** eines Schedulers hängt in erster Linie von dem für die **Zuteilung der Zeitscheiben** an die einzelnen Prozesse eingesetzten **Algorithmus** ab.
- ◇ Der Scheduler von LINUX verwendet hierfür die Kombination von **drei** verschiedenen **Algorithmen** :
  - ▷ einen "klassischen" UNIX-Scheduling-Algorithmus für "normale" Prozesse
  - ▷ zwei Scheduling-Algorithmen für "*Realtime*"-Prozesse.  
Dabei bedeutet "*realtime*" unter LINUX lediglich, daß derartige Prozesse gegenüber den "normalen" ("*nonrealtime*") Prozessen bei der CPU-Zuteilung grundsätzlich bevorzugt werden ("*soft realtime*").
- ◇ Alle drei Scheduling-Algorithmen (**Scheduling-Klassen**) berücksichtigen **Prozess-Prioritäten**, wobei für zwei der Scheduling-Klassen ein **Zeitscheibenverfahren** eingesetzt wird.  
→ **Prioritäts-Scheduling** (*priority scheduling*) mit **unterlagener Zeitscheibensteuerung**

### • Prozess-Prioritäten

- ◇ Prioritäten sind ganze Zahlen, die Prozessen zugeordnet sind und Einfluss auf deren Behandlung durch den Scheduler haben.
- ◇ Es werden **verschiedene Prioritätsarten** unterschieden :
  - ▷ **Dynamische Priorität** :  
Sie bestimmt die **Zuteilungsreihenfolge der CPU** an die rechenbereiten Prozesse  
Je **kleiner die Prioritäts-Zahl** ist, desto **größer ist die Priorität** und desto größer ist die Chance bei der nächsten CPU-Zuteilung berücksichtigt zu werden.  
Ihr Wert liegt im Bereich **0** (höchste Priorität) . . . **MAX\_PRIO-1** (==**139**) (niedrigste Priorität).  
Der Bereich **0** . . . **MAX\_RT\_PRIO-1** (==**99**) ist den "*Realtime*"-Prozessen vorbehalten.  
Prozesse mit höherer dynamischer Priorität (kleinerer Prioritätszahl) werden immer gegenüber solchen mit niedrigerer dynamischer Priorität bei der CPU-Zuteilung bevorzugt.
  - ▷ **Statische-Priorität** :  
Sie bestimmt die **Größe der Zeitscheibe**.  
Je größer die Priorität (je kleiner die Prioritätszahl) ist, desto größer ist die Zeitscheibe.  
Ausserdem bildet die statische Priorität bei "normalen" Prozessen den Ausgangswert für die dynamische Priorität
  - ▷ **"Realtime"-Priorität** :  
Eine Prioritäts-Kennzahl für "*Realtime*"-Prozesse . Sie bestimmt deren dynamische Priorität.
- ◇ Die – defaultmäßig vergewene – Priorität von Prozessen kann mit **System Calls verändert** werden.

## Prozesszustände unter LINUX

### • Prozesszustände :

LINUX unterscheidet die folgenden Prozesszustände :

#### ◇ TASK\_RUNNING

Der Prozess ist lauffähig (wartet auf die CPU) oder läuft aktuell (besitzt die CPU).

→ Zusammenfassung der beiden prinzipiellen elementaren Zustände "**bereit**" und "**aktiv**"

Auch in einem Einzel-CPU-System können sich mehrere Prozesse gleichzeitig in diesem Zustand befinden.

Alle im Zustand TASK\_RUNNING befindlichen Prozesse liegen in der *Run Queue*.

#### ◇ TASK\_INTERRUPTABLE

Der Prozess wartet auf ein Ereignis. Der Wartezustand kann – ausser durch den Eintritt des Ereignisses – auch durch ein Signal wieder beendet – d.h. der Prozess wieder ablaufbereit – werden.

#### ◇ TASK\_UNINTERRUPTABLE

Der Prozess wartet auf ein Ereignis. Er kann nicht durch ein Signal wieder ablaufbereit werden.

#### ◇ TASK\_STOPPED

Der Prozess ist angehalten worden, entweder durch ein Signal (SIGSTOP, SIGTSTP, SIGTTIN oder SIGTTOU) oder durch einen anderen ihn überwachenden Prozess (z.B. Debugger, System Call ptrace).

Durch das Signal SIGCONT kann er wieder in den Zustand TASK\_RUNNING überführt werden

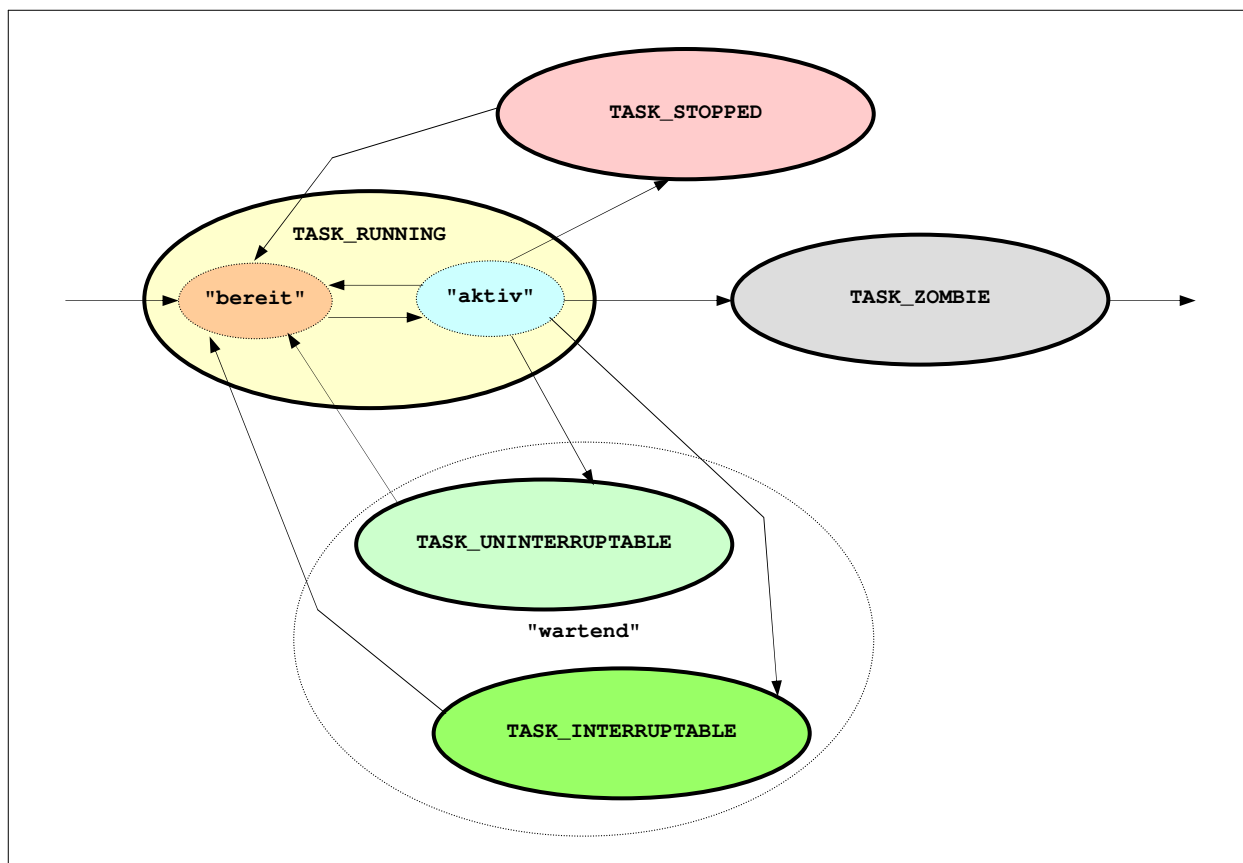
#### ◇ TASK\_ZOMBIE (ab Kernel 2.6.11 als EXIT\_ZOMBIE bezeichnet)

Der Prozess ist beendet. Er belegt aber noch einen Eintrag in der Prozessliste.

Dieser Zustand wird beibehalten, bis der Elternprozess das Ende seines Kindprozesses erfragt (System Call wait).

Erst dann wird der Prozess aus der Prozessliste ausgetragen.

### • Zustandsübergänge



## Prozess-Identifikation in LINUX (1)

### • Prozess-ID (PID)

Jeder Prozeß unter LINUX besitzt eine **Prozessidentifikationsnummer (PID *process identifier*)**, die zu seiner eindeutigen Identifikation verwendet wird. Es handelt sich um eine positive ganze Zahl im Bereich 0 bis 32767, die an jeden neu kreierte Prozess fortlaufend vergeben wird.

Zwei Prozesse haben eine fest zugeordnete PID :

- ▷ Idle-Prozess PID 0
- ▷ Init-Prozess PID 1

Ein Prozess kann seine PID mit dem System Call **getpid()** ermitteln.

Ab Kernel 2.4 liefert der System Call **gettid()** die eigentliche – als **Thread-ID** bezeichnete – PID zurück.

### • Threadgruppen-ID (TGID)

Ab Kernel 2.4 ist in Linux das **Thread-Gruppen-Konzept** von POSIX implementiert.

Alle vom selben Eltern-Prozess mittels des System Calls **clone()** mit gesetztem **CLONE\_THREAD**-Flag erzeugten Kindprozesse (== Threads) werden zu einer Thread-Gruppe zusammengefasst. Jeder dieser Threads besitzt eine **eigene PID**, die zutreffenderweise **Thread-ID** genannt wird, alle Threads der Thread-Gruppe teilen sich aber **dieselbe Thread-gruppen-ID** (die gleich der PID des erzeugenden Eltern-Prozesses ist). Bei einem Prozess, der nur aus einem Thread besteht, sind PID und TGID identisch.

Ab Kernel 2.4 gibt der System Call **gettid()** die **TGID** zurück

### • Prozess-ID des Elternprozesses (PPID)

Jeder Prozeß – außer dem Idle-Prozess – besitzt einen Erzeuger-Prozess (Vater-Prozess, **Elternprozess**, *parent process*).

Ein Eltern-Prozess kann beliebig viele Kindprozesse haben.

Stirbt ein Prozess, so werden alle seine noch **lebenden Kindprozesse** vom **Init-Prozess adoptiert**. → Der Init-Prozess wird zum neuen Elternprozess der "verwaisten" Prozesse.

Die PID des Elternprozesses eines Prozesses wird als **PPID** (*parent process id*) bezeichnet.

Ein Prozess kann seine PPID mittels des System Calls **getppid()** ermitteln.

### • Prozessgruppen-ID (PGID)

Jeder Prozess gehört zu einer **Prozessgruppe** (*process group*).

Eine Prozessgruppe bildet die Zusammenfassung von Prozessen, die auf irgendeine Art und Weise zusammengehören. (z.B. Prozesse, die gemeinsam an einer Aufgabe arbeiten, Prozesse die über eine Kommandozeile mit einer sie verknüpfenden Pipe gestartet werden). Eine Prozessgruppe kann auch nur aus einem einzigen Prozess bestehen.

Ein Prozess innerhalb einer Prozessgruppe ist der **Prozessgruppen-Führer** (*process group leader*). Seine PID ist die ID der Prozessgruppe (**Prozessgruppen-ID**, PGID, *process group id*).

Mit dem System Call **setpgid()** kann ein Prozess eine neue Prozessgruppe anlegen oder Mitglied einer existierenden Prozessgruppe werden. Diesen System Call darf ein Prozess nur für sich selbst oder einen seiner Kindprozesse aufrufen.

Mit den System Calls **getpgid()** und **getpgrp()** kann die PGID eines Prozesses ermittelt werden.

### • Session-ID

Jede Prozessgruppe gehört zu einer **Session**. → Jeder Prozeß besitzt auch eine **Session-ID**

Eine Session kann mehrere Prozessgruppen umfassen (z.B. Vordergrund-Prozessgruppe, Hintergrund-Prozessgruppe).

Eine Session hat einen Prozess als **Session-Führer** (*session leader*). Die PID des Session-Führers ist die Session-ID. Beispielsweise gehören alle über dasselbe Terminal gestarteten Prozesse zu der gleichen Session. Die **Login-Shell** des Terminals ist der **Session-Führer**.

Ein Prozess kann mittels des System Calls **setsid()** eine neue Session erzeugen und sich selbst zum neuen Session-Führer und zum neuen Prozessgruppen-Führer machen.

Alle von ihm erzeugten Kindprozesse werden Mitglieder "seiner" Session.

Die Session-ID eines Prozesses kann mittels des System Calls **getsid()** ermittelt werden.



## LINUX System Calls `getpid`, `gettid` und `getppid`

### System Call `getpid`

- **Funktionalität :** Rückgabe der **PID** (ab Kernel 2.4 der **TGID**) des aktuellen Prozesses
- **Interface :**

`pid_t getpid(void);`

  - **Header-Datei :** `<unistd.h>`
- **Implementierung :** System Call Nr. **20**  
→ `sys_getpid(...)` (in `kernel/timer.c`)
- **Anmerkung :** Der Datentyp `pid_t` ist in der Header-Datei `<unistd.h>` **definiert als `int`**.

### System Call `gettid`

- **Funktionalität :** Rückgabe der **Thread-ID** (== **PID**) des aktuellen Prozesses
- **Interface :**

`pid_t gettid(void);`

  - **Header-Datei :** `<unistd.h>` (für Prototype von `syscall(...)`)  
`<linux/unistd.h>` (für symbolische Funktions-Nr.)
- **Implementierung :** System Call Nr. **224**  
→ `sys_gettid(...)` (in `kernel/timer.c`)
- **Anmerkungen:**
  1. Der Datentyp `pid_t` ist in der Header-Datei `<unistd.h>` **definiert als `int`**.
  2. Der System Call existiert ab dem Kernel 2.4
  3. Der System Call muss über die **generische Wrapper-Funktion** `syscall(...)` aufgerufen werden : `syscall(__NR_gettid)`

### System Call `getppid`

- **Funktionalität :** Rückgabe der **PID** des Elternprozesses des aktuellen Prozesses
- **Interface :**

`pid_t getppid(void);`

  - **Header-Datei :** `<unistd.h>`
- **Implementierung :** System Call Nr. **64**  
→ `sys_getppid(...)` (in `kernel/timer.c`)
- **Anmerkung :** Der Datentyp `pid_t` ist in der Header-Datei `<unistd.h>` **definiert als `int`**.

## Prozess-Identifikation in LINUX (2)

- **User-ID (UID) und Gruppen-ID (GID)**

Die (reale) UID und die (reale) GID eines Prozesses bestimmen die **Identität des Benutzers**, der den **Prozess gestartet** hat (Prozess-Besitzer).

Beim Login eines Benutzers werden seine in der Datei `/etc/passwd` eingetragene Benutzer-Nummer (**User-ID**) und (primäre) Gruppen-Nummer (**Gruppen-ID**) zur **UID** und **GID** der **Login-Shell**, die diese dann an alle ihre Kindprozesse vererbt.

- **Effektive User-ID (EUID) und effektive Gruppen-ID (EGID)**

Sie bestimmen die tatsächlichen Zugriffsrechte des Prozesses.

Normalerweise sind sie gleich der (realen) UID bzw der (realen) GID des Prozesses

Bei der Ausführung von Programmen, bei denen das **SUID**-Bit und/oder das **SGID**-Bit gesetzt ist (sogenannte **SUID**- bzw **SGID**-Programme), werden sie jedoch auf die User-ID bzw Gruppen-ID des **Besitzers** der **Programm-Datei** gesetzt. Damit kann der das Programm ausführende Prozess mit den Zugriffsrechten des Programm-Besitzers bzw dessen Gruppe laufen, obwohl er von einem anderen User gestartet worden ist.

- **Gespeicherte User-ID (SUID) und gespeicherte Gruppen-ID (SGID)**

Normalerweise sind auch diese gleich der (realen) UID bzw der (realen) GID des Prozesses.

Bei der Ausführung von **SUID**- bzw **SGID**-Programmen werden sie gleich der **EUID** bzw der **EGID** gesetzt.

Dies ermöglicht einen mit den Rechten des Programm-Besitzers laufenden Prozess, diese Rechte – vorübergehend – gegen seine ursprünglichen Rechte (die des Prozess-Besitzers) einzutauschen (durch Setzen von **EUID=UID** bzw **EGID=GID**) und anschließend diese Rechte wieder anzunehmen (durch Setzen von **EUID=SUID** bzw **EGID=SGID**)

- **Dateisystem-User-ID (FSUID) und Dateisystem-Gruppen-ID (FSGID)**

Diese legen die Zugriffsrechte zum Dateisystem fest.

Normalerweise sind sie gleich der **EUID** bzw der **EGID**.

Für besondere Fälle können sie aber auf hiervon abweichende Werte gesetzt werden.

- **Ermitteln und Verändern der User-ID und der Gruppen-ID**

Hierfür stehen mehrere System Calls zur Verfügung.

Grundsätzlich gilt, daß ein Prozess mit der **EUID==0** (**root**-Rechte) alle Arten der User-ID und der Gruppen-ID auf beliebige Werte setzen darf.

Jeder mit einer anderen **EUID** laufende Prozess darf

- seine **EUID** (bzw **EGID**) nur auf seine (reale) **UID** (bzw (reale) **GID**) oder auf seine **SUID** (bzw **SGID**) setzen
- seine (reale) **UID** (bzw (reale) **GID**) auf seine **EUID** (bzw **EGID**) setzen

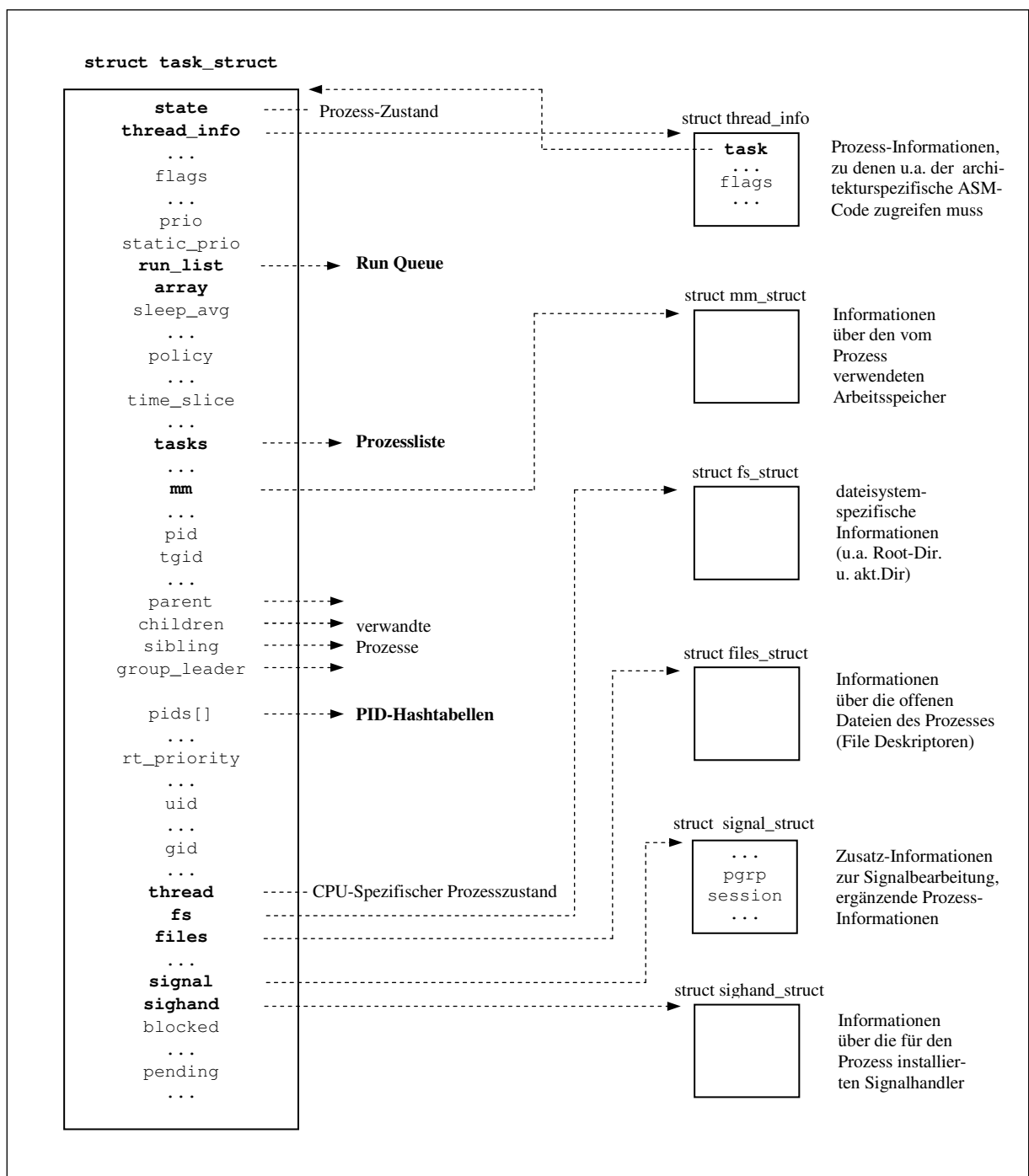
Es stehen folgende **System Calls** zur Verfügung :

- ▷ **setuid()** Setzen der **EUID** (keine **root**-Rechte) bzw der **EUID**, **UID** und **SUID** (**root**-Rechte)
- ▷ **setgid()** Setzen der **EGID** (keine **root**-Rechte) bzw der **EGID**, **GID** und **SGID** (**root**-Rechte)
- ▷ **setreuid()** Setzen der (realen) **UID** und der effektiven User-ID (**EUID**)
- ▷ **setregid()** Setzen der (realen) **GID** und der effektiven Gruppen-ID (**EGID**)
- ▷ **setfsuid()** Setzen der Dateisystem-User-ID (**FSUID**)
- ▷ **setfsgid()** Setzen der Dateisystem-Gruppen-ID (**FSGID**)
- ▷ **getuid()** Rückgabe der (realen) **UID**
- ▷ **getgid()** Rückgabe der (realen) **GID**
- ▷ **geteuid()** Rückgabe der effektiven User-ID (**EUID**)
- ▷ **getegid()** Rückgabe der effektiven Gruppen-ID (**EGID**)

## Datenstrukturen zur Prozessverwaltung in LINUX (1)

### • Prozessdeskriptor (Prozessverwaltungsstruktur, Task-Struktur, *task structure*)

- ◇ Zur Prozessverwaltung legt LINUX für jeden Prozeß einen Prozeßdeskriptor an.  
Dies ist eine Datenstruktur, in der sämtliche **prozessspezifischen Beschreibungs- und Verwaltungsinformationen** zusammengefasst sind, wobei einige Informationen in **Unterstrukturen** enthalten sind, die durch entsprechende Pointer-Komponenten referiert werden.
- ◇ Ein Prozessdeskriptor wird durch den in der Headerdatei `include/linux/sched.h` definierten Structure-Typ **struct task\_struct** beschrieben.  
Mittels typedef ist als weiterer Typname **task\_t** definiert: **typedef struct task\_struct task\_t;**
- ◇ **Prinzipieller Überblick** über den Prozessdeskriptor :



## Datenstrukturen zur Prozessverwaltung in LINUX (2-1)

- **Prozessverwaltungsstruktur struct task\_struct (1.Teil)** (Kernel 2.6.8)

Dieser einen Prozess beschreibende Structure-Datentyp ist definiert in `include/linux/sched.h`

```
struct task_struct {
    volatile long state;          /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;          /* per process flags, defined below */
    unsigned long ptrace;

    int lock_depth;              /* Lock depth */

    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    long interactive_credit;
    unsigned long long timestamp;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
    struct sched_info sched_info;
#endif
    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a ptracer.
     */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

    /* task state */
    struct linux_binfmt *binfmt;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
    /* ??? */
    unsigned long personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively. (p->father can be replaced with
     * p->parent->pid)
     */
    struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent;      /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */

    /* Fortsetzung der Definition s. nächste Seite */
}
```

## Datenstrukturen zur Prozessverwaltung in LINUX (2-2)

- **Prozessverwaltungsstruktur struct task\_struct (2.Teil) (Kernel 2.6.8)**

```
/* Fortsetzung der Definition von struct task_struct */

/* PID/PID hash table linkage. */
struct pid pids[PIDTYPE_MAX];

wait_queue_head_t wait_chldexit; /* for wait4() */
struct completion *vfork_done; /* for vfork() */
int __user *set_child_tid; /* CLONE_CHILD_SETTID */
int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */

unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
unsigned long utime, stime;
unsigned long nvcsw, nivcsw; /* context switch counts */
u64 start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-
specific */
unsigned long min_flt, maj_flt;
/* process credentials */
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsgid;
struct group_info *group_info;
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
unsigned keep_capabilities:1;
struct user_struct *user;
/* limits */
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
/* file system info */
int link_count, total_link_count;
/* ipc stuff */
struct sysv_sem sysvsem;
/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* namespace */
struct namespace *namespace;
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;

sigset_t blocked, real_blocked;
struct sigpending pending;

unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;

void *security;
struct audit_context *audit_context;

/* Fortsetzung der Definition s. nächste Seite */
```

## Datenstrukturen zur Prozessverwaltung in LINUX (2-3)

- **Prozessverwaltungsstruktur struct task\_struct (3.Teil) (Kernel 2.6.8)**

```
/* Fortsetzung der Definition von struct task_struct */

/* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
    spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock,
write_lock_irq(&tasklist_lock); */
    spinlock_t proc_lock;
/* context-switch lock */
    spinlock_t switch_lock;

/* journalling filesystem info */
    void *journal_info;

/* VM state */
    struct reclaim_state *reclaim_state;

    struct dentry *proc_dentry;
    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use. */
/*
 * current io wait handle: wait queue entry to use for io waits
 * If this thread is processing aio, this points at the waitqueue
 * inside the currently handled kiocb. It may be NULL (i.e. default
 * to a stack based synchronous wait) if its doing sync IO.
 */
    wait_queue_t *io_wait;
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy;
    short il_next; /* could be shared with used_math */
#endif

/* TASK_UNMAPPED_BASE */
    unsigned long map_base;
    struct list_head private_pages; /* per-process private pages */
    int private_pages_count;
};
```

## Datenstrukturen zur Prozessverwaltung in LINUX (2-4)

- Ergänzende Thread-Informationsstruktur **struct thread\_info** (Kernel 2.6.8)

◇ Dieser Structure-Datentyp ist definiert in **include/asm/thread\_info.h**

Er fasst process-spezifische Daten zusammen, zu denen vom architektur-spezifischen Assembler-Code zugegriffen werden muss

```
struct thread_info {
    struct task_struct *task;           /* main task structure */
    struct exec_domain *exec_domain;    /* execution domain */
    unsigned long flags;                /* low level flags */
    unsigned long status;                /* thread-synchronous flags */
    __u32 cpu;                           /* current CPU */
    int preempt_count;                   /* 0 => preemptable, <0 => BUG */

    mm_segment_t addr_limit;             /* thread address space:
                                           0-0xBFFFFFFF for user-thead
                                           0-0xFFFFFFFF for kernel-thread
                                           */
    struct restart_block restart_block;

    unsigned long previous_esp;          /* ESP of the previous stack in case
                                           of nested (IRQ) stacks
                                           */
    __u8 supervisor_stack[0];
};

#define PREEMPT_ACTIVE 0x10000000
#ifdef CONFIG_4KSTACKS
#define THREAD_SIZE (4096)
#else
#define THREAD_SIZE (8192)
#endif
```

◇ Die Komponente **task** der Thread-Informationsstruktur ist ein Zeiger auf den zugehörigen Prozessdeskriptor  
→ doppelte Verkettung zwischen dem Prozessdeskriptor und der zugehörigen Thread-Informationsstruktur

◇ Die Komponente **flags** der Thread-Informationsstruktur enthält verschiedene prozess-spezifische Steuerungs- und Informations-Flags.

U.a. gehören zu diesen Flags :

- ▷ **TIF\_SIGPENDING** : ein gesetztes Flag zeigt an, dass dem Prozess zugestellte **Signale** auf Bearbeitung warten
- ▷ **TIF\_NEED\_RESCHED** : ein gesetztes Flag zeigt an, das bei nächster Gelegenheit der **Scheduler** aufgerufen werden soll (**need\_resched-Flag**).

◇ Die Komponente **preempt\_count** ist der **Präemptionszaehler**. Dieser dient zur Überprüfung, ob der aktuell ausgeführte Kernel-Code durch Präemption (erzwungener Prozesswechsel) unterbrochen werden darf.

**Kernel-Präemption** ist zulässig, wenn **preempt\_count == 0** ist (s.a. "6.3 Synchronisation im Kernel")

Veränderung des Werts von **preempt\_count** mittels den Makros (definiert in **include/linux/preempt.h**) :

- ▷ **inc\_preempt\_count()** (Aufruf vor Eintritt in den kritischen Abschnitts)
- ▷ **dec\_preempt\_count()** (Aufruf nach Verlassen des kritischen Abschnitts)

## Datenstrukturen zur Prozessverwaltung in LINUX (3)

### • Speicherallokation für einen Prozessdeskriptor und die Thread-Informationsstruktur

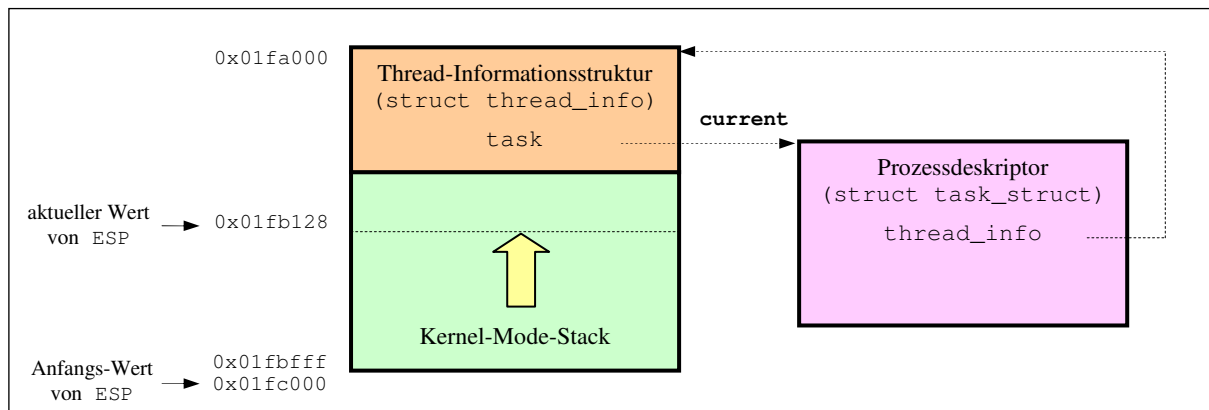
- ◇ Für jeden Prozess muss neben dem Prozessdeskriptor und der Thread-Informationsstruktur auch ein **Kernel-Mode-Stack** (bei der x86-Architektur : Stack der Ebene 0) bereitgestellt werden.  
Bei der Erzeugung eines neuen Prozesses belegt der Kernel für den **Stack** zusammen mit der **Thread-Informationsstruktur** einen i.a. **8 kBytes** großen dynamisch allozierten Speicherbereich (2 aufeinanderfolgende Speicherseiten). Für den **Prozessdeskriptor** wird ein davon unabhängiger Speicherbereich alloziert. Eine Komponente der Thread-Informationsstruktur zeigt auf diesen Speicherbereich
- ◇ Der Typ des von der Thread-Informationsstruktur und dem Stack belegten Speicherbereichs wird durch die folgende, in der Headerdatei **include/linux/sched.h** definierte, Union beschrieben :

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

Die Konstante **THREAD\_SIZE** ist in der Headerdatei **include/asm/thread\_info.h** definiert.

Die Thread-Informationsstruktur belegt den "unteren" Teil (niedrigere Adressen) des Bereichs, während der Stack von "oben" (höhere Adressen) her nach "unten" wächst. Zu Beginn (Eintritt in den Kernel-Mode) enthält der Stackpointer (ESP) die Adresse des unmittelbar auf den 8k-Speicherbereich folgenden Bytes (leerer Stack, bei der x86-Architektur zeigt der Stackpointer immer auf das letzte besetzte Wort im Stack)

Da die Thread-Informationsstruktur weniger als 50 Bytes belegt, kann die Stack-Grösse bis nahezu 8 kBytes wachsen.



### • Ermittlung der Adresse des Prozessdeskriptors des aktuellen Prozesses (current Macro)

- ◇ Die Adresse des Prozessdeskriptors des aktuellen Prozesses lässt sich im Kernel-Modus – und nur dort ist sie relevant und wird als **Prozessreferenz** verwendet – indirekt über die Adresse der zugehörigen Thread-Informationsstruktur aus dem Wert des Stackpointers **ESP** ermitteln, indem dessen letzten 13 Bits gleich 0 gesetzt werden ( $8k == 2^{13}$ ).

```
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    __asm__("mov ti, esp");
    __asm__("and ti, 0xfffffe00");
    return ti;
}
```

definiert in  
**include/asm/thread\_info.h**

```
static inline struct task_struct * get_current(void)
{
    return current_thread_info()->task;
}

#define current get_current()
```

definiert in  
**include/asm/current.h**



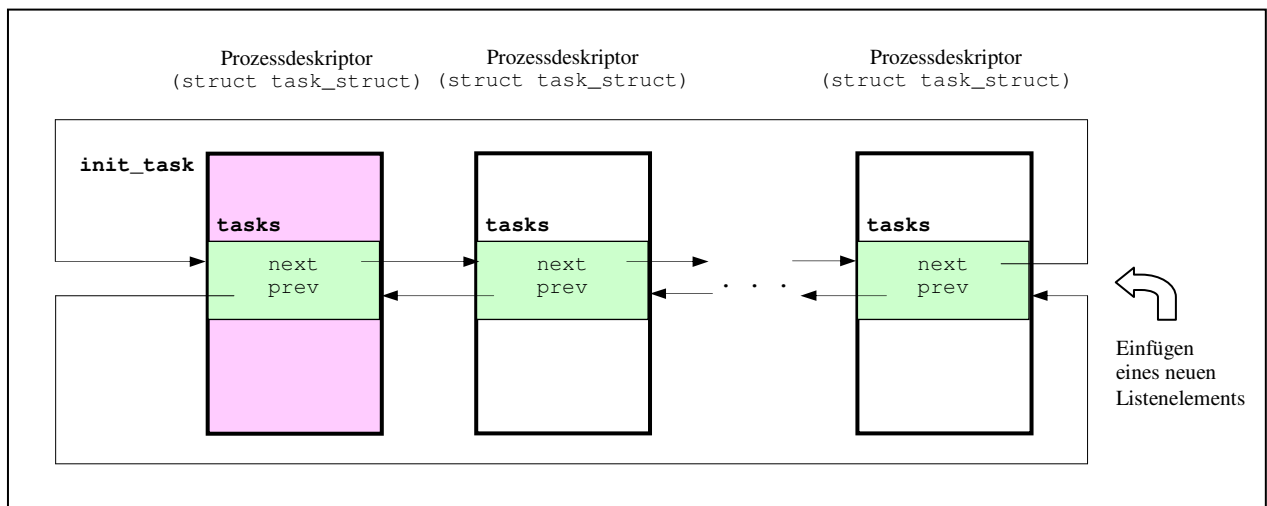
## Datenstrukturen zur Prozessverwaltung in LINUX (4)

### • Prozessliste

- ◇ Alle Prozessdeskriptoren sind über eine **doppelt verkettete Liste** miteinander verknüpft  
→ Sie bilden die **Prozessliste**.
- ◇ Die Verkettung der Listenelemente erfolgt über die Komponente **tasks** des Prozessdeskriptors. Diese Komponente fasst die Pointer auf das nächste und das vorherige Listenelement zusammen. Sie ist von dem in der Headerdatei **include/linux/list.h** definierten Structure-Typ (s. "6.1. Listenverwaltung")

```
struct list_head {
    struct list_head *next, *prev;
}
```

- ◇ **Listenkopf** ist der Prozessdeskriptor der **Idle-Task** (PID=0). Dieser wird über den Namen **init\_task** referiert. Der Prozessdeskriptor eines **neu erzeugten Prozesses** wird an das **Listenende** eingefügt.



- ◇ Die **maximal mögliche Anzahl von Prozessen** wird durch die Kernel-Variablen **max\_threads** festgelegt, die mit einem an die **physikalische Arbeitsspeichergröße** angepassten Wert initialisiert wird (in der Funktion **\_\_init\_fork\_init()** in **kernel/fork.c**). Ab Kernel 2.4 ist dieser Wert so gewählt, dass **maximal 1/8 des Arbeitsspeichers** durch Thread-Informationsstrukturen einschließlich Kernel-Mode-Stacks belegt werden kann. (z.B. bei 256 MB Speicher → 4096 Prozesse maximal). **Mindestens** wird der Wert aber auf **20** gesetzt.
- ◇ Zur **Ermittlung eines Pointers auf den Anfang eines Prozessdeskriptors** aus einem Pointer auf die enthaltene Komponente **tasks** vom Typ **struct list\_head** lässt sich der Macro **list\_entry** anwenden. Dieser Macro ist prinzipiell wie folgt definiert (indirekt in **include/linux/list.h**, in Verbindung mit **include/linux/kernel.h**):

```
#define list_entry(ptr, type, member) \
    ((type *) ((char *) (ptr) - (unsigned long) (&((type *) 0) -> member)))
```

Der Aufruf

```
list_entry(tmp, struct task_struct, tasks) mit struct list_head tmp
```

wird expandiert zu

```
((struct task_struct *) ((char *) (tmp) - (unsigned long) (&((struct task_struct *) 0) -> tasks)))
```

## Datenstrukturen zur Prozessverwaltung in LINUX (5)

### • Liste der ablaufbereiten Prozesse (*Run Queue*)

- ◇ Zusammenfassung aller sich im Zustand **TASK\_RUNNING** befindlichen Prozesse.  
Bei einem anstehenden Prozesswechsel wählt der Scheduler aus dieser Liste den nächsten zu aktivierenden Prozess aus.
- ◇ In SMP-Systemen existiert **pro CPU** jeweils eine **eigene Run Queue** (seit dem Kernel 2.6).
- ◇ Eine Run Queue besteht nicht aus einer einzigen verketteten Liste, sondern setzt sich aus **mehreren doppelt verketteten Teillisten** zusammen.  
In jede Teilliste sind **Prozesse derselben Priorität** zusammengefasst.  
Die Teillisten sind auf **zwei Guppen** aufgeteilt :
  - Gruppe der Teillisten mit Prozessen, deren **Zeitscheibe noch nicht abgelaufen** ist (*active queue*)
  - Gruppe der Teillisten mit Prozessen, deren **Zeitscheibe abgelaufen** ist (*expired queue*).Prinzipiell wird in jeder Gruppe für jede Prioritätsstufe eine Teilliste existieren (die natürlich auch leer sein kann).
- ◇ Jeder ablaufbereite Prozess kann sich zu jedem Zeitpunkt immer nur in einer der Teillisten befinden  
Die zu einer Teilliste gehörenden Prozesse sind über die Komponente **run\_list** (Typ `struct list_head`) ihres Prozessdeskriptors miteinander **verknüpft**.  
Zusätzlich verweist die Komponente **array** des Prozessdeskriptors auf die Teillisten-Gruppe (*active queue* oder *expired queue*) in der sich der Prozess aktuell befindet.
- ◇ Eine **Run Queue** wird im wesentlichen mit Hilfe der beiden folgenden Structure-Datentypen **implementiert** :
  - ▷ **struct prio\_array**  
Diese Datenstruktur enthält als Komponente u.a. ein Array des Element-Typs `struct list_head`.  
In diesem Array sind die **Listenköpfe** der einzelnen Teillisten einer **Teillisten-Gruppe** zusammengefasst.  
Der **Array-Element-Index** entspricht der **Priorität** der Prozesse in der jeweiligen Teilliste.  
→ Die Teillisten und damit die in ihnen enthaltenen Prozesse sind nach der Priorität geordnet.
  - ▷ **struct runqueue**  
Pro CPU wird eine Variable dieses Typs als Element des Arrays **runqueues** angelegt (in Einprozessor-Systemen besitzt das Array nur ein Element) :  

```
struct runqueue runqueues[NR_CPUS];
```

 (def. in `kernel/sched.c`)  
Die Datenstruktur `struct runqueue` enthält u.a. als Komponente ein **zwei-elementiges Array** des Element-Typs **struct prio\_array**  
Ein Element dieses **struct prio\_array**-Arrays ist für die Teillisten der *active queue* zuständig, das andere für die Teillisten der *expired queue*.

### • Structure-Typ **struct prio\_array**

- ◇ **Definition** : (enthalten in `kernel/sched.c`)

```
struct prio_array {  
    unsigned int nr_active;  
    unsigned long bitmap[BITMAP_SIZE];  
    struct list_head queue[MAX_PRIO];  
};
```

- ◇ **Bedeutung der Komponenten** :

- ▷ **nr\_active** **Gesamtanzahl** aller Prozesse, die sich insgesamt in den Teillisten befinden
- ▷ **bitmap** **Prioritäts-Bitmap**, jeder möglichen Priorität ist ein Bit zugeordnet.  
Ein Bit ist gesetzt, wenn die zugehörige Teilliste wenigstens einen Prozess enthält
- ▷ **queue** **Array von Listenköpfen** für die einzelnen Teillisten.  
Für jede mögliche Priorität existiert ein Array-Element

- ◇ **Alternativer Typname** : **prio\_array\_t** (definiert mittels `typedef` in `include/linux/sched.h`)

## Datenstrukturen zur Prozessverwaltung in LINUX (6)

### • Structure-Typ `struct runqueue`

◇ Vereinfachte Definition : (enthalten in `kernel/sched.c`)

```
struct runqueue {
    spinlock_t lock;
    unsigned long nr_running;
#ifdef CONFIG_SMP
    unsigned long cpu_load;
#endif
    unsigned long long nr_switches;
    unsigned long expired_timestamp, nr_uninterruptible;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;

#ifdef CONFIG_SMP
    struct sched_domain *sd;

    /* For active balancing */
    int active_balance;
    int push_cpu;

    task_t *migration_thread;
    struct list_head migration_queue;
#endif

#ifdef CONFIG_SCHEDSTATS
    // . . .
#endif
};
```

◇ Bedeutung einiger Komponenten :

- ▷ **nr\_running** Anzahl aller ablaufbereiten Prozesse in der Run Queue (unabhängig von der Priorität, der Scheduling-Klasse und dem Ablauf/Nicht-Ablauf einer Zeitscheibe)
- ▷ **expired\_timestamp** Zeitpunkt (in Jiffies), zu dem nach Wechsel der *active queue* erstmals ein Prozess auf die *expired queue* umgesetzt worden ist
- ▷ **curr** Pointer auf Prozessdeskriptor des aktuell laufenden Prozesses
- ▷ **idle** Pointer auf Prozessdeskriptor des Idle-Prozesses (der Idle-Prozess ist in keiner Teilliste enthalten)
- ▷ **active** Pointer auf `struct prio_array` für die *active queue*
- ▷ **expired** Pointer auf `struct prio_array` für die *expired queue*
- ▷ **arrays** Array mit den beiden `struct prio_array`-Elementen, die die Listenköpfe für die *active queue* und die *expired queue* enthalten.

◇ Alternativer Typname : `runqueue_t` (definiert mittels `typedef` in `kernel/sched.c`)

### • Funktionen zur Manipulation der Run Queue (definiert in `kernel/sched.c`)

◇ Einfügen eines Prozesses in eine Teillisten-Gruppe (*active queue* oder *expired queue*):

**static void enqueue\_task(struct task\_struct \*p, prio\_array\_t \*array)**

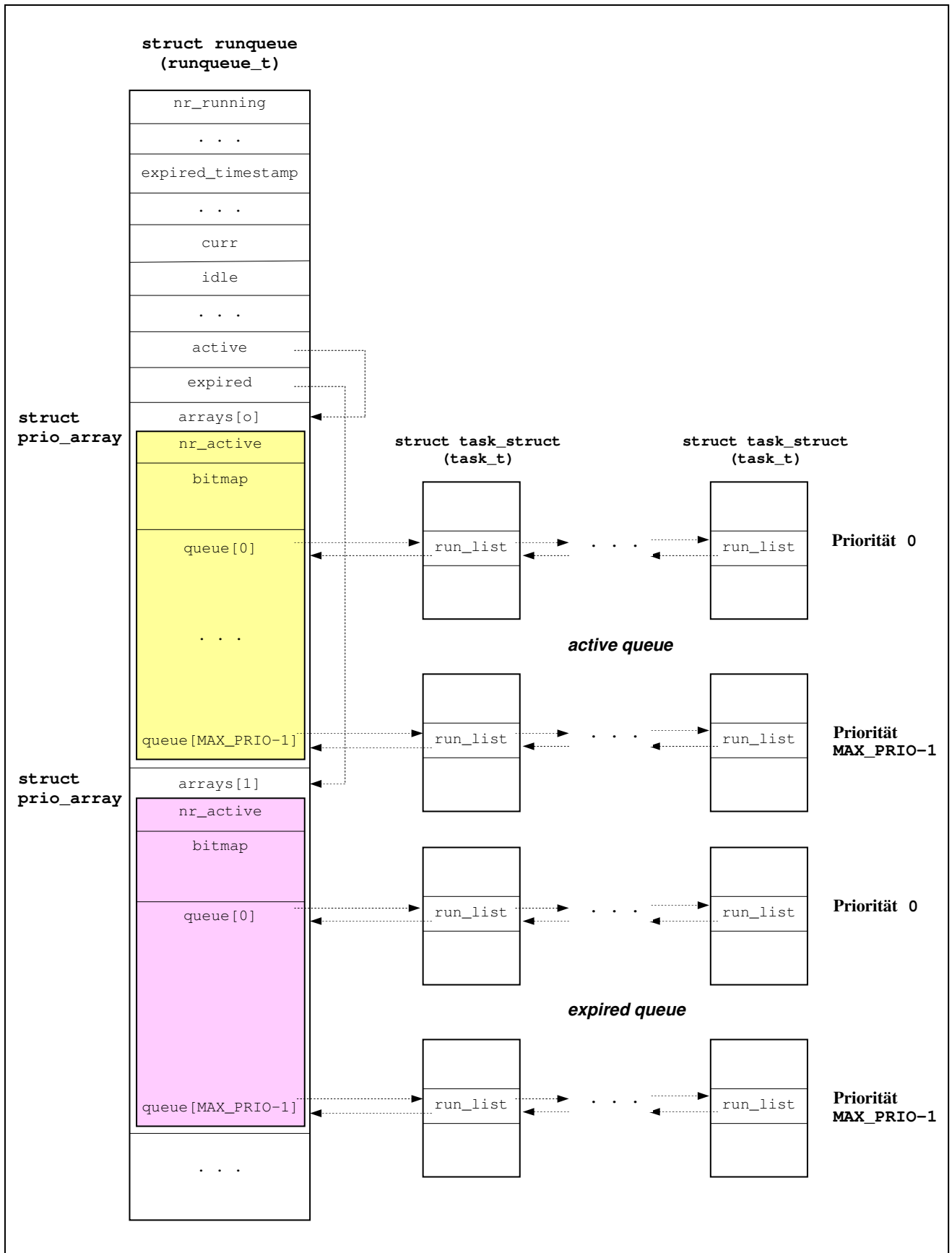
Der durch `p` referierte Prozess wird an das Ende der Teilliste seiner dynamischen Priorität eingefügt.

◇ Entfernen eines Prozesses aus einer Teillisten-Gruppe (*active queue* oder *expired queue*):

**static void dequeue\_task(struct task\_struct \*p, prio\_array\_t \*array)**

## Datenstrukturen zur Prozessverwaltung in LINUX (7)

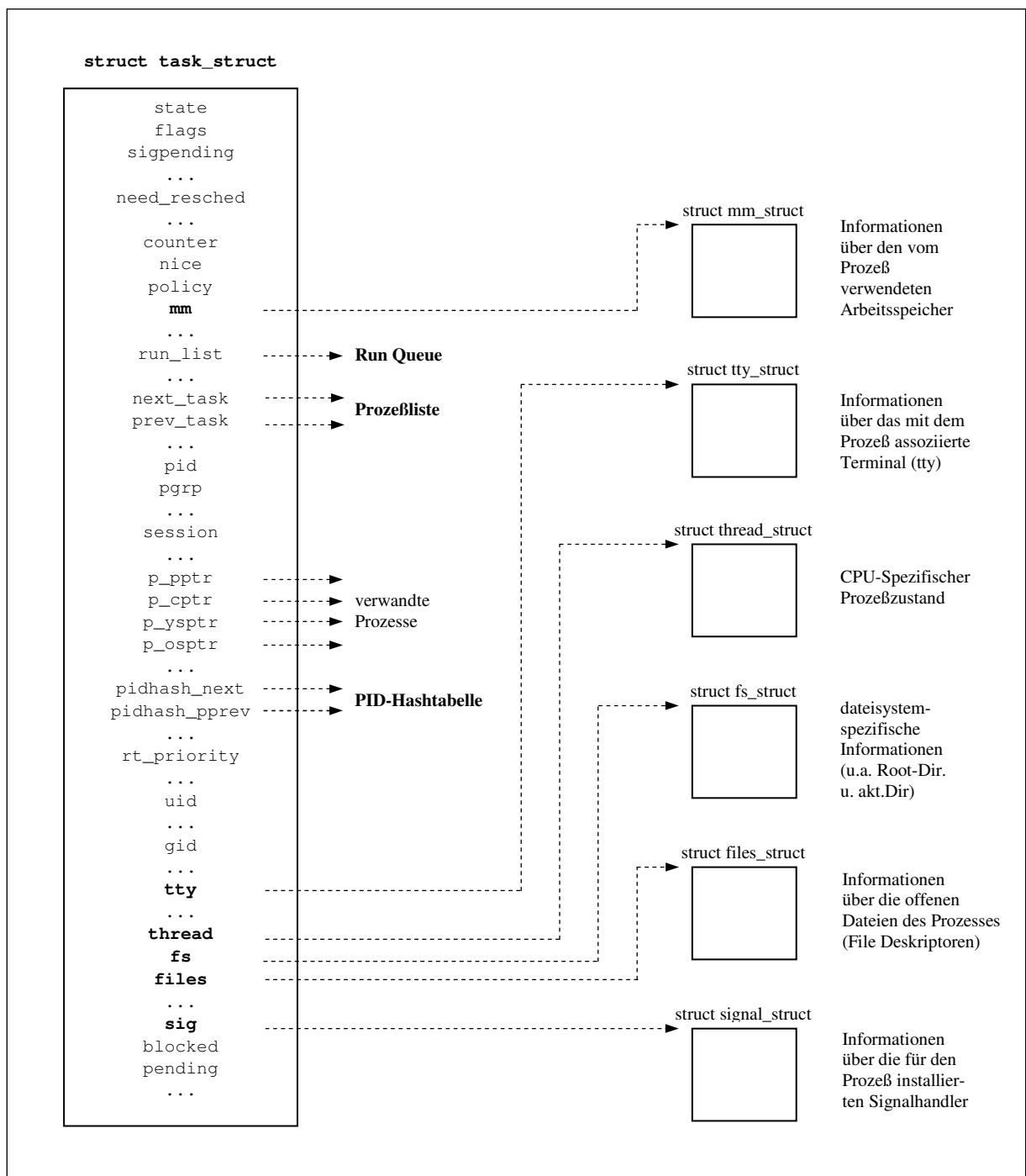
- Überblick über den Aufbau einer Run Queue



## Datenstrukturen zur Prozeßverwaltung in LINUX (1) – Kernel 2.4

### • Prozeßdeskriptor (Prozessverwaltungsstruktur, Task-Struktur, *task structure*)

- ◇ Zur Prozeßverwaltung legt LINUX für jeden Prozeß einen Prozeßdeskriptor an.  
Dies ist eine Datenstruktur, in der sämtliche **prozeßspezifischen Beschreibungs- und Verwaltungsinformationen** zusammengefasst sind, wobei einige Komponenten auf weitere Strukturen verweisen.
- ◇ Ein Prozeßdeskriptor wird durch den in der Headerdatei `include/linux/sched.h` definierten Structure-Typ **struct task\_struct** beschrieben.
- ◇ **Prinzipieller Überblick** über den Prozeßdeskriptor :



## Datenstrukturen zur Prozeßverwaltung in LINUX (2-1) – Kernel 2.4

- **Prozeßverwaltungsstruktur `struct task_struct` (1.Teil)** (Kernel 2.4.18)

Dieser einen Prozeß beschreibende Structure-Datentyp ist definiert in `include/linux/sched.h`

```
struct task_struct {
    /*
     * offsets of these are hardcoded elsewhere - touch with care
     */
    volatile long state;          /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags;          /* per process flags, defined below */
    int sigpending;
    mm_segment_t addr_limit;      /* thread address space:
                                   0-0xBFFFFFFF for user-thread
                                   0-0xFFFFFFFF for kernel-thread
                                   */
    struct exec_domain *exec_domain;
    volatile long need_resched;
    unsigned long ptrace;

    int lock_depth;               /* Lock depth */

    /*
     * offset 32 begins here on 32-bit platforms. We keep
     * all fields in a single cacheline that are needed for
     * the goodness() loop in schedule().
     */
    long counter;
    long nice;
    unsigned long policy;
    struct mm_struct *mm;
    int processor;
    /*
     * cpus_runnable is ~0 if the process is not running on any
     * CPU. It's (1 << cpu) if it's running on a CPU. This mask
     * is updated under the runqueue lock.
     *
     * To determine whether a process might run on a CPU, this
     * mask is AND-ed with cpus_allowed.
     */
    unsigned long cpus_runnable, cpus_allowed;
    /*
     * (only the 'next' pointer fits into the cacheline, but
     * that's just fine.)
     */
    struct list_head run_list;
    unsigned long sleep_time;

    struct task_struct *next_task, *prev_task;
    struct mm_struct *active_mm;
    struct list_head local_pages;
    unsigned int allocation_order, nr_local_pages;

    /* task state */
    struct linux_binfmt *binfmt;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
    /* ??? */
    unsigned long personality;
    int did_exec:1;

    /* Fortsetzung der Definition s. nächste Seite */
}
```

## Datenstrukturen zur Prozeßverwaltung in LINUX (2-2) – Kernel 2.4

- Prozeßverwaltungsstruktur **struct task\_struct** (2.Teil) (Kernel 2.4.18)

```
/* Fortsetzung der Definition von struct task_struct */

pid_t pid;
pid_t pgrp;
pid_t tty_old_pgrp;
pid_t session;
pid_t tgid;
/* boolean value for session group leader */
int leader;
/*
 * pointers to (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->p_pptr->pid)
 */
struct task_struct *p_opptr, *p_pptr, *p_cptra, *p_ysptr, *p_osptr;
struct list_head thread_group;

/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;

wait_queue_head_t wait_chldexit; /* for wait4() */
struct completion *vfork_done; /* for vfork() */
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
struct tms times;
unsigned long start_time;
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
/* mm fault and swap info: this can arguably be seen as either mm-specific or
thread-specific */
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnsnap;
int swappable:1;
/* process credentials */
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsuid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
/* limits */
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
/* file system info */
int link_count, total_link_count;
struct tty_struct *tty; /* NULL if no tty */
unsigned int locks; /* How many file locks are being held */
/* ipc stuff */
struct sem_undo *semundo;
struct sem_queue *semsleeping;

/* Fortsetzung der Definition s. nächste Seite */
```

## Datenstrukturen zur Prozeßverwaltung in LINUX (2-3) – Kernel 2.4

- Prozeßverwaltungsstruktur **struct task\_struct** (3.Teil) (Kernel 2.4.18)

```
/* Fortsetzung der Definition von struct task_struct */

/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* signal handlers */
spinlock_t sigmask_lock; /* Protects signal and blocked */
struct signal_struct *sig;

sigset_t blocked;
struct sigpending pending;

unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;

/* Thread group tracking */
u32 parent_exec_id;
u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
spinlock_t alloc_lock;

/* journalling filesystem info */
void *journal_info;
};
```



## Datenstrukturen zur Prozeßverwaltung in LINUX (3) – Kernel 2.4

### • Speicherallokation für einen Prozeßdeskriptor

- ◇ Prozeßdeskriptoren werden zusammen mit dem **Kernel-Mode-Stack** des jeweiligen Prozesses (bei der x86-Architektur : Stack der Ebene 0) in **dynamisch allozierten Speicherbereichen** abgelegt.  
Bei der Erzeugung eines neuen Prozesses stellt der Kernel hierfür einen **8 kBytes** großen Speicherbereich (2 aufeinanderfolgende Speicherseiten) bereit.
- ◇ Der Typ dieses Speicherbereichs wird durch die folgende, in der Headerdatei `include/linux/sched.h` definierte, Union beschrieben :

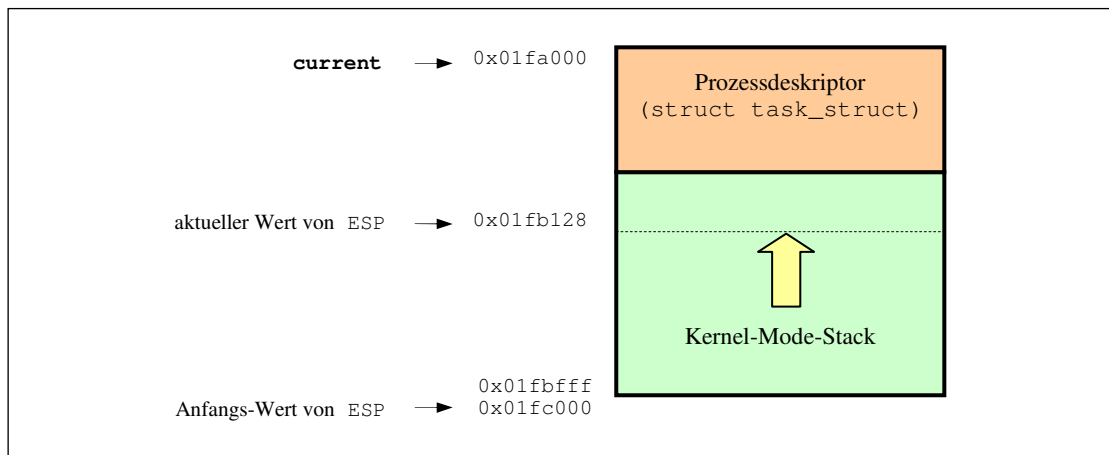
```
# define INIT_TASK_SIZE    2048*sizeof(long)

union task_union {
    struct task_struct task;
    unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};
```

Der Prozeßdeskriptor belegt den "unteren" Teil (niedrigere Adressen) des Bereichs, während der Stack vom "oben" (höhere Adressen) her nach "unten" wächst.

Zu Beginn (Eintritt in den Kernel-Mode) enthält der Stackpointer (ESP) die Adresse des unmittelbar auf den 8k-Speicherbereich folgenden Bytes (leerer Stack, bei der x86-Architektur zeigt der Stackpointer immer auf das letzte besetzte Wort im Stack)

Da der Prozeßdeskriptor weniger als 1000 Bytes belegt, kann der Stack bis zu einer Größe von ca 7200 Bytes wachsen.



### • Das `current` Macro

- ◇ Die **Adresse des Prozeßdeskriptors** des **aktuellen Prozesses** lässt sich im Kernel-Modus – und nur dort ist sie relevant und wird als Prozeßreferenz verwendet – aus dem Wert des Stackpointers **ESP** ermitteln, indem dessen **letzten 13 Bits gleich 0** gesetzt werden ( $8k = 2^{13}$ ).  
Hierzu dient der in der Headerdatei `include/asm/current.h` prinzipiell wie folgt definierte **current** Macro :

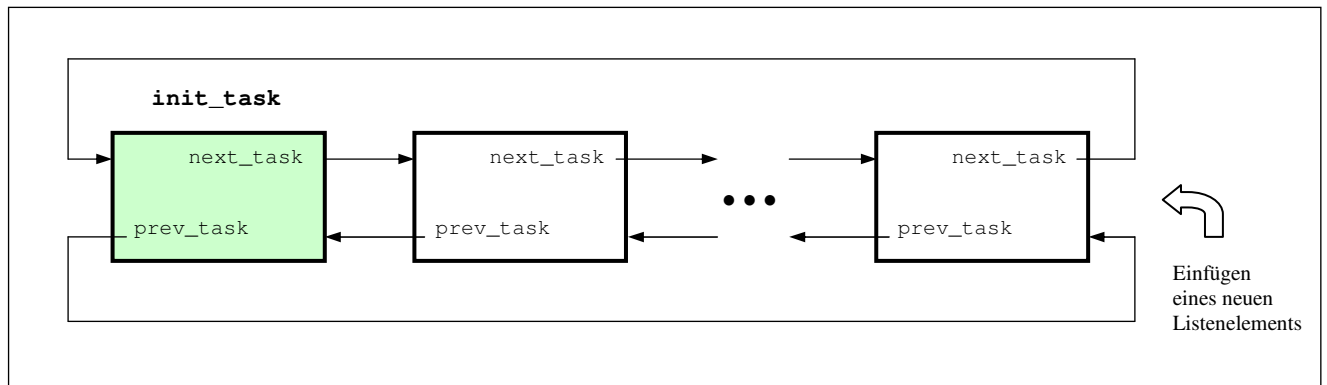
```
static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__("mov current, esp");
    __asm__("and current, 0xfffffe000");
    return current;
}

#define current get_current()
```

## Datenstrukturen zur Prozeßverwaltung in LINUX (4) – Kernel 2.4

### • Prozeßliste

- ◇ Alle Prozeßdeskriptoren sind über eine **doppelt verkettete Liste** miteinander verknüpft → Sie bilden die **Prozeßliste**. Die **Verknüpfungspointer** sind in der Struktur `struct task_struct` als Komponenten **`next_task`** und **`prev_task`** enthalten
- ◇ **Listenkopf** ist der Prozeßdeskriptor der **Idle-Task** (PID=0). Dieser wird über den Namen **`init_task`** referiert. Der Prozeßdeskriptor eines **neu erzeugten Prozesses** wird an das **Listenende** eingefügt.



- ◇ Die **maximal mögliche Anzahl von Prozessen** wird durch die Kernel-Variable **`max_threads`** festgelegt, die mit einem an die **physikalische Arbeitsspeichergröße angepassten** Wert initialisiert wird (in `linux/fork.c`). Zur Zeit (Kernel 2.4.18) ist dieser Wert so gewählt, dass **maximal 1/8 des Arbeitsspeichers** durch Prozeßdeskriptoren (einschließlich Kernel-Mode-Stacks) belegt werden kann. (z.B. bei 256 MB Speicher → 4096 Prozesse maximal)
- ◇ **Anmerkung :**  
Bis einschließlich dem Kernel 2.2. hat zusätzlich eine **Prozesstabelle** existiert. Diese war als statisch allokiertes Array von **Pointern auf Prozeßdeskriptoren** realisiert, in dem jeder Prozeß genau einen Eintrag belegt hat.  
In `kernel/sched.c` war sie definiert als  

```
struct task_struct* task[NR_TASKS];
```

  
Ihre Größe `NR_TASKS` hat die **maximal mögliche Prozessanzahl** festgelegt.  
In `include/linux/tasks.h` war dieser Wert definiert als **512**.

### • PID-Hashtabelle `pidhash`

- ◇ Dies ist eine Tabelle von Pointern auf Prozeßdeskriptoren. Sie dient zum schnellen **Ermitteln der Adresse des Prozeßdeskriptors** eines **über seine PID referierten Prozesses**. Eine die `PID` als Parameter verwendende **Hash-Funktion (`pid_hashfn()`)**, in `include/linux/sched.h` definiert) sorgt für eine gleichmäßige Abbildung des `PID`-Nummernbereichs auf die Indices dieser Tabelle. Für unterschiedliche `PIDs`, die auf denselben Tabellen-Index abgebildet werden, werden **Hash-Listen** unter Verwendung von innerhalb der Prozeßverwaltungsstrukturen (`struct task_struct`) angesiedelten Verknüpfungspointern (`pidhash_next` und `pidhash_pprev`) gebildet
- ◇ Die `PID`-Hashtabelle ist definiert in `kernel/fork.c` :  

```
struct task_struct* pidhash[PIDHASH_SZ];
```

  
Die Tabellengröße `PIDHASH_SZ` ist in `include/linux/sched.h` definiert :  

```
#define PIDHASH_SZ (4096 >> 2)
```

  
→ derzeit (Kernel 2.4.18) beträgt sie also **1024**.

## Datenstrukturen zur Prozeßverwaltung in LINUX (5) – Kernel 2.4

### • Liste der ablaufbereiten Prozesse (*Run Queue*)

- ◇ Zusammenfassung aller sich im Zustand **TASK\_RUNNABLE** befindlichen Prozesse.  
Bei einem anstehenden Prozesswechsel wählt der Scheduler aus dieser Liste den nächsten zu aktivierenden Prozeß aus.
- ◇ Die Verkettung der Listenelemente erfolgt über die Komponente **run\_list** des Prozeßdeskriptors.  
Diese Komponente faßt die Pointer auf das nächste und das vorherige Listenelement zusammen.  
Sie ist von dem in der Headerdatei `include/linux/list.h` definierten Structure-Typ

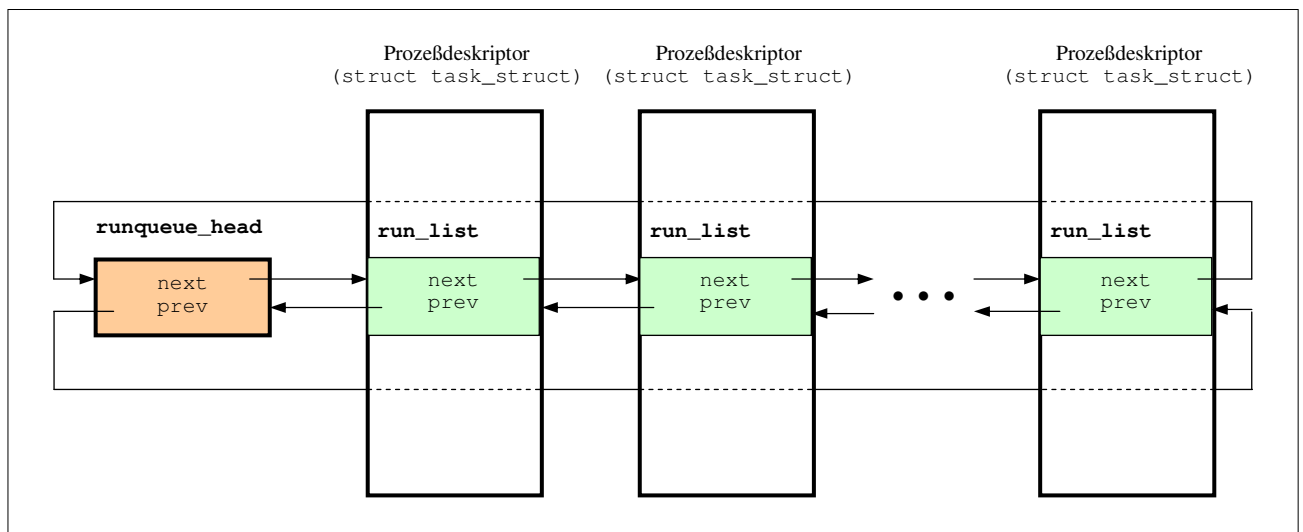
```
struct list_head {
    struct list_head *next, *prev;
}
```

Dieser Structure-Typ ermöglicht eine **generische Verwaltung von Listenelementen** beliebigen Typs, sofern eine Komponente dieses Typs in ihnen enthalten ist.

Entsprechende "**Bearbeitungsfunktionen**" sind als – z.Tl. durch `inline`-Funktionen realisierte – Makros ebenfalls in `include/linux/list.h` definiert

- ◇ Der **Listenkopf** der *Run Queue* wird durch eine statisch-globale in `kernel/sched.c` definierte Variable gebildet :

```
static struct list_head runqueue_head;
```



- ◇ Zur **Ermittlung eines Pointers auf den Anfang eines Prozeßdeskriptors** aus einem Pointer auf die enthaltene Komponente `run_list` vom Typ `struct list_head` wird der folgende in `include/linux/list.h` definierte Makro eingesetzt :

```
#define list_entry(ptr, type, member) \
    ((type *) ((char *) (ptr) - (unsigned long) (&((type *) 0) -> member)))
```

Der Aufruf

`list_entry(tmp, struct task_struct, run_list)` mit `struct list_head tmp`  
wird expandiert zu

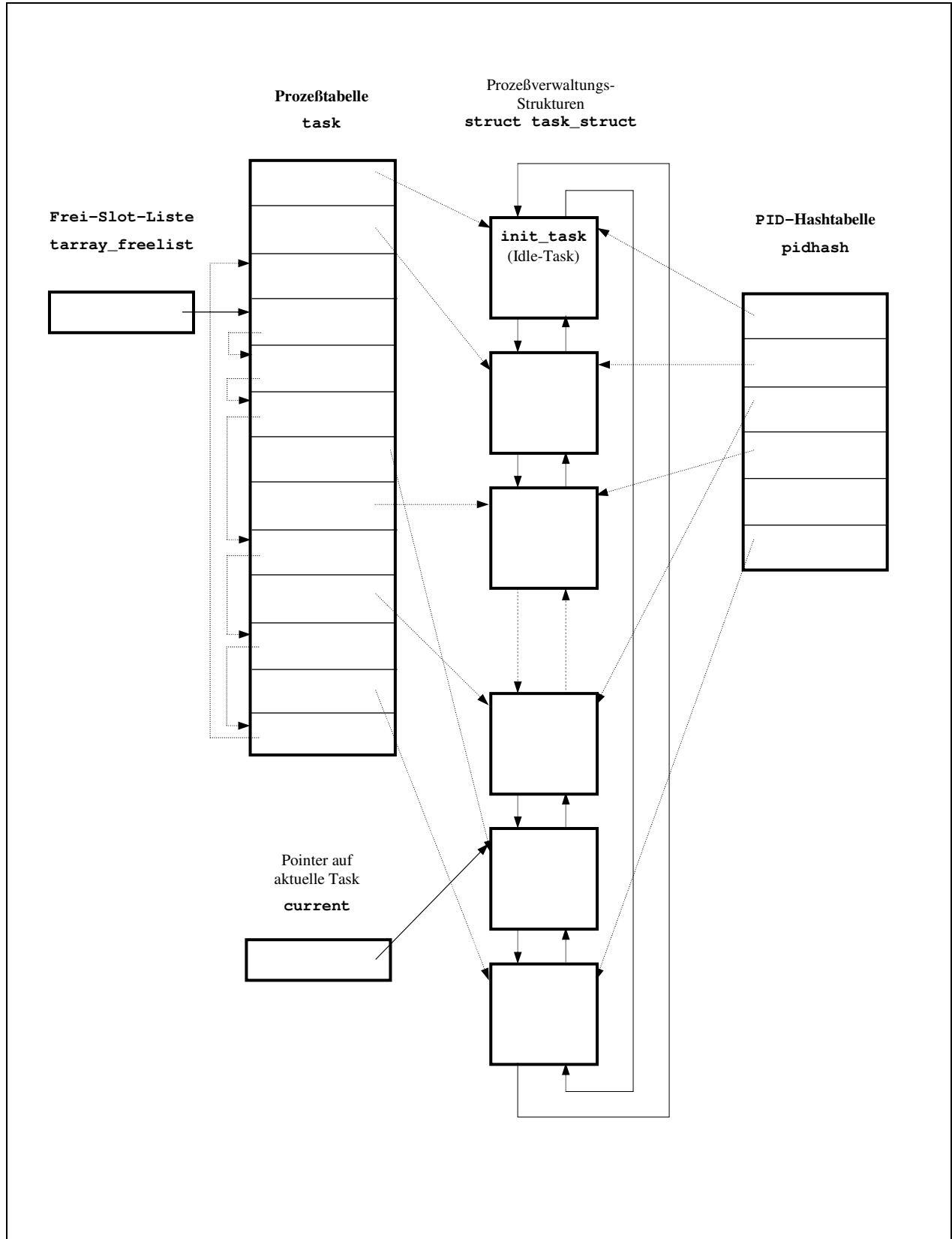
```
((struct task_struct *) ((char *) (tmp) - (unsigned long) (&((struct task_struct *) 0) -> run_list)))
```

- ◇ Zur **Verwaltung der Run Queue** sind u.a. die folgenden `inline`-Funktionen definiert :

- ▷ `void add_to_runqueue(struct task_struct * p)` (in `kernel/sched.c`)  
Einfügen `p` an den Anfang der *Run Queue*
- ▷ `void del_from_runqueue(struct task_struct * p)` (in `include/linux/sched.h`)
- ▷ `void move_last_runqueue(struct task_struct * p)` (in `kernel/sched.c`)
- ▷ `void move_first_runqueue(struct task_struct * p)` (in `kernel/sched.c`)

## Datenstrukturen zur Prozeßverwaltung in LINUX (6) – Kernel 2.2

- Überblick (Kernel 2.2)



## Erzeugung von Prozessen in LINUX (1)

### • System Call **fork**

- ◇ Standardmäßig werden neue Prozesse in LINUX – wie in allen UNIX-artigen Betriebssystemen i.a. üblich – mit dem System Call **fork()** erzeugt.

Dieser System Call kreiert einen **neuen Prozess**, der eine **fast identische Kopie** des aufrufenden Prozesses ist. Der neue Prozeß ist **Kindprozess** des erzeugenden **Elternprozesses**.

Bei **Erfolg** findet eine **Rückkehr** sowohl in den **Elternprozess** als auch in den **Kindprozess** statt (*fork* → Gabel(ung) ! ) Beide Prozesse führen dasselbe Programm aus. Sie setzen es jeweils mit der auf den System Call folgenden Anweisung fort.

Eine **Unterscheidung** erlaubt lediglich der **Rückgabewert** des System Calls :

- im **Elternprozess** : **PID** des erzeugten **Kindprozesses**
- im **Kindprozess** : **0**

Durch Auswertung dieses Rückgabewertes ist es möglich, in beiden Prozessen **unterschiedlich zu verzweigen** und diese damit unterschiedlich fortzusetzen.

→ Typische Formulierung :

```
switch (fork())
{
    case -1 : /* im Elternprozess Fehlermeldung */
        break;
    case 0 : /* Code für Kindprozess */
        break;
    default : /* Code für Elternprozess */
        break;
}
```

- ◇ Der neu erzeugte **Kindprozess** besitzt einen **eigenen virtuellen Adressraum**, der aber in **denselben physikalischen Adressraum** wie beim **Elternprozess** abgebildet wird. → **Kopieren der Seitentabellen** !
  - ⇒ Beide Prozesse verwenden **dasselbe Codesegment** (nur ausführender Zugriff).
  - ⇒ Beide Prozesse verwenden **zunächst** auch **dasselbe Datensegment** – solange beide Prozesse nur **lesend** zugreifen. Erst wenn einer der beiden Prozesse **schreibend** zum Datensegment zugreift, wird dieses für den Kindprozess in einen **eigenen physikalischen Adressraum** kopiert → *Copy-on-Write*.
- ◇ Der **Kindprozess erbt** fast alle **Attribute des Elternprozesses**. (Die Prozessverwaltungsstruktur wird weitgehend kopiert.) Insbesondere werden **geerbt** (als Kopie) :
  - ▷ alle offenen **File-Deskriptoren** (einschließlich eventueller Umleitungen der Standardgeräte). Duplizierung : Sie können unabhängig geschlossen werden.
  - ▷ reale User-/Gruppen-ID (**UID/GID**) und effektive User-/Gruppen-ID (**EUID/EGID**)
  - ▷ **Prozessgruppen-ID, Session-ID**
  - ▷ **Dateikreierungsmaske**
  - ▷ **Arbeitsdirectory, Root-Directory**
  - ▷ **Environment**
  - ▷ **Signalmaske** und **Signalhandler**
  - ▷ **Ressourcen-Begrenzungen** (*resource limits*)
- ◇ Es bestehen **nur wenige Unterschiede** zwischen Eltern- u. Kindprozess, im wesentlichen sind es die folgenden :
  - ▷ **Rückgabewert** von **fork()**
  - ▷ **PID** und **PPID**
  - ▷ die gesammelten **Ausführungszeiten** werden beim Kindprozess auf **0** gesetzt.
  - ▷ **Dateiverriegelungen** (*File Locks*) werden nicht vererbt
  - ▷ gestartete **Timer** werden ausgeschaltet
  - ▷ hängende (noch nicht zugestellte) **Signale** werden nicht vererbt

## LINUX System Call **fork**

- **Funktionalität :** Erzeugung eines **Kindprozesses**.  
Bei Erfolg kehrt der System Call sowohl im Elternprozess als auch im Kindprozess zurück.
- **Interface :**

`pid_t fork(void);`

  - **Header-Datei :** `<unistd.h>`  
`<sys/types.h>`
  - **Parameter :** keine
  - **Rückgabewert :**
    - bei **Erfolg**
      - im **Elternprozess** : **PID** des erzeugten **Kindprozesses**
      - im **Kindprozess** : **0**
    - im **Fehlerfall** nur im Elternprozess : **(pid\_t) (-1)**, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. **2**
  - `sys_fork(...)` (in `arch/x86/kernel/process_32.c`)
  - `do_fork(...)` (in `kernel/fork.c`)

**do\_fork()** führt im wesentlichen nacheinander aus :

  - ▷ Erzeugen und Setzen aller prozessspezifischen Datenstrukturen
    - `p=copy_process()` (in `kernel/fork.c`)
    - `p=dup_task_struct()` (in `kernel/fork.c`)
    - `p=alloc_task_struct()` (in `kernel/fork.c`) (Makro)
    - `p=alloc_thread_info()` (in `include/asm/thread_info.h`)
    - Kopieren des Prozessdeskriptors
    - Kopieren der Thread-Informationsstruktur
    - Kopieren der Unterstrukturen
    - Setzen aller nichtkopierbaren Komponenten
    - Einhängen des Prozessdeskriptor in die Taskliste
    - `list_add_tail_rcu()`, (in `include/linux/list.h`)
  - ▷ Einhängen des Prozessdeskriptors in die *Run Queue*  
`wake_up_new_task(p)`, (in `kernel/sched.c`)

- **Anmerkungen:**
  1. Der Datentyp **pid\_t** ist in der Header-Datei `<sys/types.h>` **definiert** als **int**.
  2. In LINUX existiert ein **weiterer System Call** mit gleichem Interface und **ähnlicher Funktionalität** :

`pid_t vfork(void);`

Dieser System Call ist insbesondere für Fälle vorgesehen, bei denen sich der erzeugte Kindprozess unmittelbar nach seiner Erzeugung mit einem neuen Programm überlagert

Die **Implementierung** dieses System Calls führt ebenfalls zum Aufruf von `do_fork()` :

System Call Nr. **190**

- `sys_vfork(...)` (in `arch/x86/kernel/process_32.c`)
- `do_fork(...)` (in `kernel/fork.c`)

### Beispiel zum LINUX System Call **fork**

```
/* -----*/
/* Programm forktest                                     */
/* -----*/
/* Beispiel zum System Call fork und einigen anderen System Calls */
/* -----*/

#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>

int forktest()
{
    int iRetPID;
    printf("\nStart von forktest()\n");
    fflush(stdout);
    iRetPID=fork();
    switch (iRetPID)
    {
        case -1 : printf("\nFehlschlag von fork()\n");
                  break;
        case 0  : printf("\nI'm the child  pid = %d\n", getpid());
                  printf("My father has  pid = %d\n", getppid());
                  break;
        default : printf("\nI'm the father pid = %d\n", getpid());
                  printf("My Child has   pid = %d\n", iRetPID);
    }
    return getpid();
}

int main(int argc, char *argv[])
{
    int erg = forktest();
    printf("\nProgrammende : %d\n", erg);
    return EXIT_SUCCESS;
}

/* ----- -/
```

#### **Start und - mögliche - Ausgabe des Programms :**

Start von forktest()

I'm the father pid = 2718  
My Child has pid = 2719

Programmende : 2718

I'm the child pid = 2719  
My father has pid = 1

Programmende : 2719

```
/- ----- */
```

## Erzeugung von Prozessen in LINUX (2)

### • System Call `clone`

- ◇ LINUX stellt einen weiteren System Call zur Erzeugung eines Kindprozesses zur Verfügung.  
Dieser System Call ermöglicht es, daß sich Eltern- und Kindprozess Teile des **Ausführungskontextes teilen** (z.B. auf dem gleichen virtuellen und physikalischen Arbeitsspeicher arbeiten, dieselben – nicht duplizierten – File-Deskriptoren und Signalhandler verwenden).  
**Teilen** bedeutet, daß beide Prozesse mit **denselben Ressourcen** arbeiten : eine Änderung durch den einen Prozess bedeutet dieselbe Änderung für den anderen Prozess  
(Unterschied zum Vererben : Änderungen an geerbten Ressourcen durch einen Prozess haben keine Auswirkung auf den anderen Prozess).  
Geteilt werden können :
  - ▷ virtueller (und physikalischer) Arbeitsspeicher
  - ▷ Dateisystem-Informationen (u.a. Arbeits-Directory, Root-Directory, Dateikreierungsmaske)
  - ▷ alle offenen Dateien (Dateideskriptor-Tabelle)
  - ▷ alle Signalhandler
  - ▷ der Elternprozess (Elternprozess des aufrufenden Prozesses wird auch Elternprozess des Kindprozesses)
  - ▷ die Thread-Gruppe (aufrufender Prozess und Kindprozess gehören zur gleichen Thread-Gruppe)Der Umfang des tatsächlich zu teilenden Ausführungskontextes ist dem System Call durch einen Flags-Parameter mitzuteilen.  
Die Hauptanwendung dieses System Calls liegt in der Erzeugung von **Threads**.
- ◇ Dem System Call ist als Parameter die Startadresse einer **int-Funktion**, die mit einem Pointerparameter aufgerufen wird, zu übergeben.  
Der erzeugte **Kindprozess startet** mit der **Abarbeitung dieser Funktion**. Ihr formaler Parameter ist ebenfalls dem System Call als Parameter zu übergeben.  
Wenn die **Funktion endet**, wird auch der **Prozess (Thread) beendet**.  
Der Rückgabewert der Funktion ist der Exit-Code des Kindprozesses.
- ◇ Obwohl sich Eltern- und Kindprozess den verwendeten Arbeitsspeicher teilen können, ist es grundsätzlich nicht möglich, daß beide Prozesse mit demselben Stack (User Mode Stack) arbeiten.  
Vor Aufruf von `clone()` muß daher durch den Elternprozess ein ausreichender Speicherbereich für den **Stack des Kindprozesses** allokiert werden. Die Anfangsadresse des Stacks (höchste Adresse !) ist dem System Call ebenfalls als Parameter zu übergeben.
- ◇ Der **Rückgabewert** von `clone()` im Elternprozess entspricht dem Rückgabewert von `fork()`.  
Im Kindprozess findet keine Rückkehr statt, sondern es wird die im System Call übergebene Funktion gestartet.
- ◇ Dieser System Call wird durch die gleiche – allerdings mit z.Tl unterschiedlichen Parametern aufgerufene – Funktion wie der System Call `fork()` (und `vfork()`) implementiert (→ `do_fork()`).
- ◇ Der System Call `clone()` sollte i.a. **nicht direkt von Anwenderprogrammen** benutzt werden.  
Er wird aber in zahlreichen **Bibliotheken** zur **Implementierung von Threads** verwendet.
- ◇ Für Threads, die der **gleichen Thread-Gruppe** angehören, liefert der System Call `getpid()` den **gleichen Wert** (und zwar die `TGID`).  
Eine Unterscheidung ist mittels der **Thread-ID**, die der System Call `gettid()` zurückgibt, möglich. Diese ist aber genaugenommen die eigentliche im Prozessdeskriptor gespeicherte `PID`.



## LINUX System Call `clone`

- **Funktionalität :** Erzeugung eines **Kindprozesses**.  
Kindprozess und Elternprozess können sich bestimmte Teile des **Ausführungskontextes teilen**, d.h. mit denselben Strukturen arbeiten (Unterschied zu `erben` = kopieren !).  
Bei Erfolg startet der System Call den Kindprozess mit der Ausführung einer anzugebenden Funktion und kehrt im Elternprozess zurück.  
Der Elternprozess wird mit der auf dem System Call folgenden Anweisung fortgesetzt.  
Die Hauptanwendung liegt in der Implementierung von **Threads**.

- **Interface :**

```
int clone(int (*fn)(void* arg), void* stck, int flags, void* arg);
```

- **Header-Datei :** `<sched.h>`  
`<signal.h>` (definiert die Signal-Nummern, s. Parameter `flags`)
- **Parameter :**
  - `fn` Pointer auf die durch den Kindprozess auszuführende **Funktion**.  
Die Beendigung dieser Funktion führt zur Beendigung des Kindprozesses.  
Der Rückgabewert der Funktion ist der Exit-Code des Kindprozesses
  - `stck` Pointer auf den vom Kindprozess zu verwendenden **Stack** (höchste Adresse !)
  - `flags` Im niederwertigen Byte ist die **Signal-Nr.**, die bei Beendigung des Kindprozesses an den Elternprozess zu senden ist, anzugeben (default : `SIGCHLD`).  
Diese kann zur **Festlegung** des vom Eltern- und Kindprozess **geteilten Ausführungskontextes** mit einer der folgenden Konstanten (Auswahl) bitweise oder verknüpft werden (Die Konstanten sind in der durch `<sched.h>` eingebundenen Header-Datei `<bits/sched.h>` definiert):
    - CLONE\_VM** Teilung des virtuellen (u. physikalischen) Speichers
    - CLONE\_FS** Teilung der Dateisystem-Informationen (Arbeits-Dir. usw)
    - CLONE\_FILES** Teilung der File-Deskriptor-Tabelle
    - CLONE\_SIGHAND** Teilung der Signalhandler-Tabelle
    - CLONE\_PTRACE** falls Elternprozess (Aufrufer) überwacht wird, wird auch der Kindprozess überwacht
    - CLONE\_PARENT** Elternprozess des Aufrufers wird Elternprozess des Kindes
    - CLONE\_THREAD** Aufrufer und Kindprozess gehören zur gleichen Thread-Gruppe**CLONE\_THREAD** und **CLONE\_SIGHAND** müssen immer zusammen gesetzt werden
  - `arg` **Parameter** der der durch `fn` referierten Funktion übergeben wird
- **Rückgabewert :**
  - bei **Erfolg**
    - im **Elternprozess** : **PID** des erzeugten **Kindprozesses**
    - im **Kindprozess** : keine Rückkehr (Funktion `fn` wird ausgeführt)
  - im **Fehlerfall** nur im Elternprozess : **-1**, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **120**
  - `sys_clone(...)` (in `arch/x86/kernel/process_32.c`)
  - `do_fork(...)` (in `kernel/fork.c`)
- **Anmerkungen:**
  1. Die Ausführung der anzugebenden Funktion im Kindprozess ist durch die Wrapper-Funktion in der C- Bibliothek realisiert. Der eigentlichen Betriebssystemfunktion `sys_clone()` werden nur die Parameter `stck` und `flags` übergeben.
  2. Dieser System Call wird in erster Linie zur Implementierung einer Thread-Bibliothek eingesetzt. Von Anwenderprogrammen wird er daher nur selten direkt aufgerufen werden.

### Beispiel zum LINUX System Call **clone** (1)

```
/* ----- */
/* C-Quelldatei clonetest_m.c */
/* main()-Funktion fuer das Programm clonetest */
/* Demo-Programm zu den Linux-System-Calls clone() und getpid() */
/* ----- */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <sched.h>
#include <linux/unistd.h>

pid_t gettid(void)
{ return syscall(__NR_gettid); }

#define STACK_SIZE 8192

int thrfunc(void*);
extern void delay(unsigned hms); /* Verzoeigerung um hms Hundert-Millisekunden */

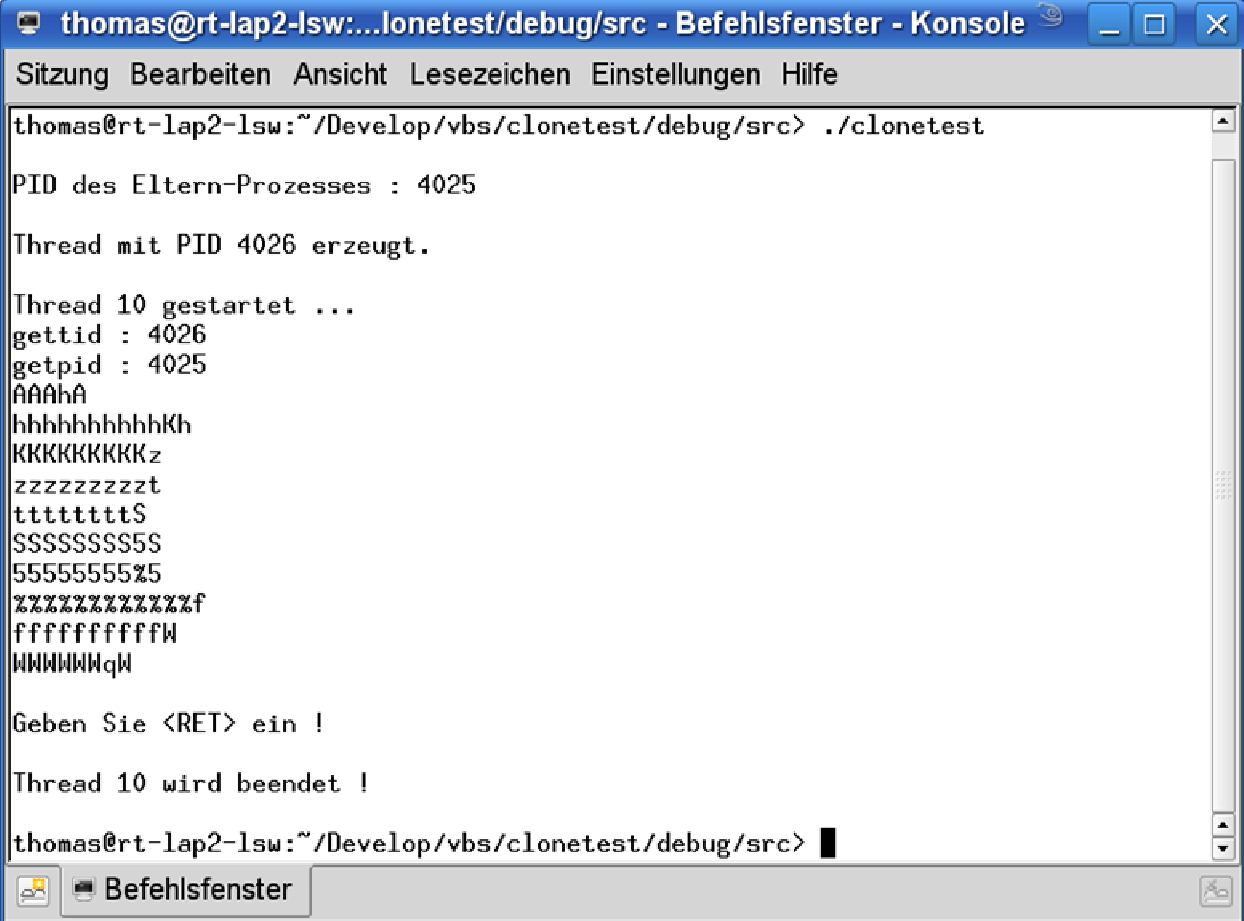
int ende = 0;
char zeich = 'A';

int main(void)
{ unsigned dly = 10;
  char* thrstack = malloc(STACK_SIZE) + STACK_SIZE;
  int scret;
  printf("\nPID des Eltern-Prozesses : %d\n", getpid());
  scret = clone(thrfunc, thrstack, CLONE_VM|CLONE_THREAD|CLONE_SIGHAND, &dly);
  if (scret== -1)
    printf("\nThread konnte nicht erzeugt werden !\n");
  else
  { printf("\nThread mit PID %d erzeugt.\n", scret);
    while (toupper(zeich)!='Q')
    { zeich =getchar();
      getchar();
    }
    ende = 1;
    printf("\nGeben Sie <RET> ein !\n");
    getchar();
  }
  return 0;
}

int thrfunc(void* par)
{ unsigned vzg = *(unsigned*)par;
  char zei;
  int i = 0;
  printf("\nThread %u gestartet ... \n", vzg);
  printf("gettid : %d\n", gettid());
  printf("getpid : %d\n", getpid());
  fflush(stdout);
  while (!ende)
  { zei = zeich;
    putchar(zeich);
    fflush(stdout);
    delay(vzg);
    i++;
  }
  printf("\nThread %u wird beendet !\n", vzg);
  fflush(stdout);
  return i;
}
```

### Beispiel zum LINUX System Call `clone` (2)

- Start und Ausgabe des Programms `clonetest` (Beispiel)



```
thomas@rt-lap2-lsw:~/Develop/vbs/clonetest/debug/src - Befehlsfenster - Konsole
Sitzung Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
thomas@rt-lap2-lsw:~/Develop/vbs/clonetest/debug/src> ./clonetest

PID des Eltern-Prozesses : 4025

Thread mit PID 4026 erzeugt.

Thread 10 gestartet ...
gettid : 4026
getpid : 4025
AAAAhA
hhhhhhhhhhKh
KKKKKKKKKz
zzzzzzzzzt
ttttttttS
SSSSSSSS5S
55555555%5
xxxxxxxxxxxx%f
fffffffffffW
WWWWWWqW

Geben Sie <RET> ein !

Thread 10 wird beendet !

thomas@rt-lap2-lsw:~/Develop/vbs/clonetest/debug/src> █
```

## Ausführung eines neuen Programms in LINUX

### • Allgemeines

Ein **neues Programm** wird in LINUX dadurch gestartet, daß sich der **aktuelle Prozess** mit dem neuen Programm **überlagert** : Das vom aktuellen Prozess bisher ausgeführte Programm wird vollständig durch das neue Programm ersetzt., d.h. der **virtuelle Speicherbereich** des Prozesses wird durch Code, Daten, Stack und Heap des neuen Programms **überschrieben**. Der **gleiche** seine Identität beibehaltende **Prozess** wird mit der Ausführung des **neuen Programms fortgesetzt**.

Wenn das von einem Prozess aktuell ausgeführte Programm auch nach Start eines neuen Programms noch weiterlaufen soll, muß dieser Prozess daher erst mit `fork()` eine Kopie seiner selbst erzeugen und diese muss sich dann mit dem neuen Programm überlagern. (z.B. Start und Ausführung eines Programms durch einen Kommandoprozessor)

### • System Call `execve`

- ◇ Die **Überlagerung** eines Prozesses mit einem **neuen Programm** erfolgt mit dem System Call `execve()` .

Diesem System Call sind neben dem **Zugriffspfad** der auszuführenden Programmdatei, Pointer auf die **Programmparameter** und das **Environment** als Parameter zu übergeben.

#### Anmerkung :

Die Gesamtgröße der Programmparameter und des Environments darf einen bestimmten Maximalwert nicht überschreiten. Der Wert ist begrenzt auf den Wert `PAGE_SIZE*MAX_ARG_PAGES` .

Die Seitengröße `PAGE_SIZE` (definiert in `include/asm-x86/page.h`) beträgt defaultmäßig 4k Bytes.

Die maximale Anzahl Seiten für Programmparameter und Environment `MAX_ARG_PAGES` ist definiert in `include/linux/binfmts.h` und beträgt defaultmäßig 32 .

Damit ergibt sich eine **Default-Maximalgröße** für **Programmparameter und Environment** von **128k** Bytes.

- ◇ Neben der eigentlichen Wrapper-Funktion existieren in der C-Bibliothek **5 weitere Funktionen** zum Aufruf dieses System Calls, die sich in der Übergabeart der Programmparameter und des Environments voneinander unterscheiden.
- ◇ **Wirkungen der Programmüberlagerung :**
  - Die meisten Prozessattribute bleiben unverändert (u.a. `PID`, `PPID`, reale `UID`, reale `GID`, `PGID`, `Session-ID`, `Arbeits-Directory`, `Root-Directory`, `Dateikreierungsmaske`, `Signalmaske`, `Ressourcen-Limits`, angesammelte Ausführungszeiten)
  - Die geöffneten Datei-Deskriptoren bleiben geöffnet, soweit für sie nicht das `close-on-exec`-Flag gesetzt ist.
  - Anstehende Signale werden rückgesetzt
  - Installierte Signalhandler werden durch die entsprechenden Default-Handler ersetzt.
  - Die effektive User-ID (`EUID`) und die effektive Gruppen-ID (`EGID`) können sich ändern, wenn das `SUID`-Bit bzw das `SGID`-Bit gesetzt ist, andernfalls bleiben sie gleich

### • ANSI-C-Bibliotheksfunktion `system()`

Die ANSI-C-Standardbibliothek stellt die Funktion (Headerdatei : `<stdlib.h>`)

```
int system(const char* cmd);
```

zur Verfügung, mit der das durch den Zugriffspfad `cmd` referierte Programm ausgeführt werden kann.

Die Funktion ruft intern die System Calls `fork()`, `execve()` und `waitpid()` auf.

Der die Funktion aufrufende Prozess wird nicht überlagert, sondern erzeugt einen neuen Prozess, der sich mit der Standard-Shell des Systems (== Kommandoprozessor) überlagert . Diese führt das durch `cmd` referierte Programm aus, indem sie sich wiederum mit diesem überlagert.

Der aufrufende Prozess wartet auf das Ende des von ihm erzeugten Prozesses und wird dann fortgesetzt.

Funktionswert : - bei Aufruf mit `cmd==NULL` : 0, wenn kein Kommandoprozessor verfügbar ist, !=0 sonst  
- Beendigungsstatus des gestarteten Programms (Format s. Rückgabewert von `waitpid()`) bei Erfolg  
- -1 im Fehlerfall

## LINUX System Call **execve** (1)

- **Funktionalität :** **Überlagerung** eines Prozesses mit einem **neuen Programm**.  
Ausführung des neuen Programms durch den Prozess.  
Dem Programm werden Programmparameter und ein Environment übergeben.

- **Interface :**

```
int execve(const char* path, char* const argv[], char* const envp[]);
```

- **Header-Datei :** **<unistd.h>**

- **Parameter :** *path* Pointer auf Datei-Zugriffspfad des auszuführenden Programms.  
Die referierte Datei muss sein :
  - direkt ausführbare Datei (Binär-Datei)
  - oder eine einem Interpreter zu übergebende Datei (automat. Aufruf des Interpreters)
  - oder eine – ebenfalls durch einen Interpreter abzuarbeitende – Skript-Datei.  
Im letzteren Fall muss die erste Zeile der Datei das folgende Format besitzen :  
"#! interpreter [arg]"  
interpreter muss der Zugriffspfad eines Interpreter-Programms sein (z.B Shell, perl usw)  
Beispiel: "#! /bin/bash" für Shell-Skripts

*argv* Array von Pointern auf die Programmparameter ("Kommandozeilenparameter").  
*argv*[0] muss ein Pointer auf den Programmnamen sein.  
Das Array muss mit dem `NULL`-Pointer abgeschlossen sein.

*envp* Array von Pointern auf die Environmentvariablen.  
Das Array muss mit dem `NULL`-Pointer abgeschlossen sein.

- **Rückgabewert :**
  - bei **Erfolg** kehrt der System Call nicht zurück (Abarbeitung des neuen Programms !)
  - **-1** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. **11**
  - `sys_execve(...)` (in `arch/x86/kernel/process_32.c`)
  - `do_execve(...)` (in `fs/exec.c`)

- **Anmerkungen:**
  1. Im Fall einer Shell-Skriptdatei darf die erste Zeile nicht mehr als 127 Zeichen umfassen
  2. Das `SUID`- und `SGID`-Bit werden bei Skriptdateien von LINUX ignoriert
  3. In der C-Bibliothek existieren noch 5 weitere Funktionen zum Aufruf dieses System Calls.  
Sie unterscheiden sich durch eine unterschiedliche Übergabe der Programmparameter und des Environments.

## LINUX System Call `execve` (2)

- Weitere C-Bibliotheksfunktionen zum Aufruf des System Calls `execve`

- **Header-Datei** für alle Funktionen : `<unistd.h>`

- ◆ 

```
int execl(const char* path, char* const arg0, ... /* NULL */);
```

- Übergabe der **Kommandozeilenparameter** als **Parameterliste**: `arg0, arg1, arg2, ...`
  - `arg0 == argv[0]` Pointer auf Programmname
  - Abschluss mit `NULL`-Pointer
- Übergabe des **aktuellen Environments** (referiert durch Systemvariable `extern char** environ;`)

- ◆ 

```
int execlp(const char* path, char* const arg0, ... /* NULL */);
```

- Übergabe der **Kommandozeilenparameter** als **Parameterliste**: `arg0, arg1, arg2, ...`
- Übergabe des **aktuellen Environments**
- Wenn `path` keinen Slash (`'/'`) enthält, wird nach der Datei in den `PATH`-Directories gesucht, enthält `path` einen Slash (`'/'`), wird `path` als vollständiger Zugriffspfad interpretiert

- ◆ 

```
int execle(const char* path, char* const arg0, ... /* NULL */,  
            char* const envp[]);
```

- Übergabe der **Kommandozeilenparameter** als **Parameterliste**: `arg0, arg1, arg2, ...`
- Übergabe eines **Arrays von Pointern** auf die **Environmentvariablen**

- ◆ 

```
int execv(const char* path, char* const argv[]);
```

- Übergabe der **Kommandozeilenparameter** als **Pointerarray** (wie bei `execve()`)
- Übergabe des **aktuellen Environments**

- ◆ 

```
int execvp(const char* path, char* const argv[]);
```

- Übergabe der **Kommandozeilenparameter** als **Pointerarray** (wie bei `execve()`)
- Übergabe des **aktuellen Environments**
- Wenn `path` keinen Slash (`'/'`) enthält, wird nach der Datei in den `PATH`-Directories gesucht, enthält `path` einen Slash (`'/'`), wird `path` als vollständiger Zugriffspfad interpretiert

## Ausführbare Dateiformate in LINUX

### • Allgemeines

- ◇ LINUX unterstützt die Ausführung von **Binärdateien unterschiedlicher Formate** sowie die Ausführung von **interpretierbaren Dateien**.
  - ◇ Die standardmäßig unterstützten Formate ausführbarer Dateien werden bei der System-Initialisierung registriert :
  - ◇ Durch **nachladbare Module** können weitere Formate installiert werden.
  - ◇ Die Registrierung der einzelnen Formate erfolgt mittels der Funktion
- `int register_binfmt(struct linux_binfmt* fmt);` (*fs/exec.c*)
- ◇ Zur Registrierung während der System-Initialisierung erfolgt der Aufruf dieser Funktion durch spezielle Binärformat-Initialisierungsfunktionen, wie .z.B:

```
int init_elf_format_binfmt(void) (fs/binfmt_elf.c)
```

### • Standardmäßig verfügbare Ausführungsformate :

- ▷ **elf** (*Executable and Linkable Format*) : heutiges Standard-Binärformat von LINUX
- ▷ **a.out** "altes" Binärformat von LINUX (u.a. ineffektivere Unterstützung von Shared Libraries)
- ▷ **script** Format zur Ausführung von Skript-Dateien (z.B. Shell-Skripts). Der hierfür aufzurufende Interpreter muss in der ersten Zeile der Skript-Datei angegeben sein. Dem Interpreter wird die Datei zur Ausführung übergeben.
- ▷ **misc** Ebenfalls ein Format zur Ausführung von Applikationen, die einen externen Interpreter benötigen. Der zu verwendende Interpreter wird anhand spezieller Kennzeichen der Applikations-Datei (z.B. Magic Number oder Extension) erkannt. Dieses Format wird z.B. verwendet, um Java-Byte-Code (*class-Dateien*) oder Windows-Programme (*exe-Dateien*) "direkt" auszuführen. Die unterstützten Applikations-Typen müssen installiert werden

### • Structure Datentyp `struct linux_binfmt`

- ◇ Dieser in der Headerdatei `include/linux/binfmts.h` definierte Datentyp dient zur Beschreibung eines Ausführungsformats.  
Er fasst Pointer auf die für die Unterstützung eines Ausführungsformats benötigten Funktionen zusammen :

```
/*
 * This structure defines the functions that are used to load the binary formats
 * that linux accepts.
 */
struct linux_binfmt {
    struct linux_binfmt * next;
    struct module *module;
    int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
    int (*load_shlib)(int fd);
    int (*core_dump)(long signr, struct pt_regs * regs);
    unsigned long min_coredump; /* minimal dump size */
};
```

- ◇ Für jedes registrierte Ausführungsformat existiert eine Variable dieses Typs.  
Alle entsprechenden Variablen sind in einer **vorwärts verketteten Liste** zusammengefaßt, die über die in der Datei `fs/exec.c` definierten Kernel-Variablen
- ```
static struct linux_binfmt* formats
```
- referiert wird.
- ◇ Beim Versuch, eine ausführbare Datei zu laden, wird vom `execve`-System Call mit der in `fs/exec.c` definierten Funktion `search_binary_handler(...)` diese Liste nach einer "passenden" Lade-Funktion (Komponente `load_binary`) und damit nach einem passenden Ausführungsformat durchsucht.

## LINUX-Ausführungsformat `binfmt_misc` (1)

### • Grundsätzliches

- ◇ Das Ausführungsformat `binfmt_misc` ist ein **generisches Ausführungsformat**. Es ermöglicht, **Anwendungen unterschiedlichster (Binär-)Formate** "direkt" – allein durch **Aufruf** mit dem **Programm-Dateinamen** (bzw Dateipfad) – zu starten. Die derartig startbaren (Binär-)Formate (Applikations-Typen) müssen **registriert** werden.
- ◇ Tatsächlich wird für jedes unterstützte (Binär-)Format ein **Interpreter-Programm** aufgerufen, dem die jeweilige Programmdatei zur Abarbeitung übergeben wird. Der zu verwendende Interpreter wird entweder aus der **Dateinamen-Extension** oder anhand **spezieller Bytefolgen** innerhalb der Programmdatei ("*Magic Number*") erkannt. Die Erkennungsmethode, die für die Erkennung benötigten Informationen und das zuständige Interpreter-Programm müssen bei der Registrierung eines Formats (Applikations-Typs) angegeben werden.
- ◇ Die Implementierung dieses Ausführungsformats ist in der Quelldatei `fs/binfmt_misc.c` enthalten
- ◇ Die verschiedenen registrierten (Binär-)Formate werden durch Variable des Structure-Typs `Node` beschrieben, die in einer **doppelt verketteten Liste** (Komponente vom Typ `struct list_head`) zusammengefasst sind.
- ◇ Die zur Beschreibung des Ausführungsformats `binfmt_misc` dienende Variable vom Typ `struct linux_binfmt` trägt den Namen `misc_format` (definiert in `fs/binfmt_misc.c`).
- ◇ Ein **Zugriffs-Interface** zur Verwendung dieses Ausführungsformats wird durch das **/proc-Dateisystem** realisiert.

### • Verwendung des Ausführungsformats `binfmt_misc`

- ◇ Bevor dieses Ausführungsformat verwendet werden kann, muss es als "Dateisystem" **gemountet** werden :
  - ▷ Kommando : `mount -t binfmt_misc none /proc/sys/fs/binfmt_misc`
  - ▷ oder Eintrag in `etc/fstab` : `none /proc/sys/fs/binfmt_misc binfmt_misc defaults 0 0`
  - Im Verzeichnis `/proc/sys/fs/binfmt_misc` werden die "Dateien" `status` und `register` angelegt
- ◇ Durch **Schreiben** von `0` in die Datei `status` (z.B. mit `echo 0 > /proc/sys/fs/binfmt_misc/status`) kann `binfmt_misc` **temporär deaktiviert** werden, durch **Schreiben** von `1` kann es **wieder aktiviert** werden. (Root-Rechte erforderlich)

### • Registrierung von "direkt" startbaren (Binär-)Formaten (Root-Rechte erforderlich)

- ◇ Für jedes zu registrierende Format muss in die "Datei" `register` ein String folgender Form geschrieben werden :  
**:name:type:offset:magic:mask:interpreter:**

Bedeutung der einzelnen Felder des Strings :

- **name** : frei wählbarer Kennungs-String für das Format → Anlage einer Typ-"Datei" unter diesem Namen
- **type** : Erkennungstyp : **E** für Extension, **M** für Magic Number
- **offset** : Offset der *Magic Number* in Bytes vom Dateianfang, **default** : `0` (bei Extension leeres Feld)
- **magic** : *Magic Number* (als String mit darstellbaren Zeichen u/o sedez. Zeichenersatzdarstellung `\x. .`) bzw Extension (ohne ".")
- **mask** : optionale Ausblend-Maske für *Magic-Number*-String, **default** : für jedes Zeichen `\xff`
- **interpreter** : absoluter Zugriffspfad des aufzurufenden Interpreter-Programms

- ◇ Zweckmässiger Weise werden diese Strings mit dem **echo**-Kommando in die Datei `register` geschrieben.

#### ◇ Beispiel :

- ▷ **Windows-Programme** (EXE-Dateien, Magic Number `MZ` am Dateianfang, Interpreter `wine`)  
`echo ':DOSWin:M::MZ::/usr/bin/wine:' > /proc/sys/fs/binfmt_misc/register`  
→ Datei `/proc/sys/fs/binfmt_misc/DOSWin` wird angelegt

- ◇ Wenn eine Registrierung **permant** beim **Systemstart** erfolgen soll, sollte das entsprechenden Kommando zweckmässigerweise in eine beim Systemstart abgearbeitete **Init-Script-Datei** aufgenommen werden (z.B. `/etc/init.d/boot.local` bei SUSE-Linux).

- ◇ Durch **Schreiben** von `-1` in die jeweilige **Typ-"Datei"** kann ein registriertes Format wieder **deregistriert** werden.



## LINUX-Ausführungsformat `binfmt_misc` (2)

### • "Direkte" Ausführung von Java-Byte-Code-Dateien

- ◇ Das generische Ausführungsformat `binfmt_misc` ermöglicht auch die "direkte" Ausführung von Java-Byte-Code-Dateien (`class`-Dateien).
- ◇ **Voraussetzungen :**
  - ▷ `binfmt_misc` muss **gemountet** sein (s. Verwendung des Ausführungsformats `binfmt_misc`)
  - ▷ Java-Byte-Code muss als startbares (Binär-)Format **registriert** werden
- ◇ Java-Byte-Code wird durch das Programm **java** (→ JVM, *Java Virtual Machine*) interpretierend abgearbeitet. Diesem Programm ist aber nicht der Name einer `class`-Datei zu übergeben, sondern der **Name einer Klasse** (dies ist i.a. der Hauptnamens-Teil der Datei, ohne Extension `.class` und ohne eventuellen Directory-Pfad). Wenn die Datei, die die Klasse enthält, sich nicht im aktuellen Verzeichnis befindet, sucht `java` diese in einem durch die Environment-Variable `CLASSPATH` festgelegten Directory-Pfad.
  - Das von `binfmt_misc` aufzurufende **Interpreter-Programm** muss ein **Wrapper-Programm** sein,
    - das aus dem ihm übergebenen Dateinamen (bzw Dateipfad) den Hauptnamen (== Klassennamen) extrahiert und den eigentlichen Java-Interpreter `java` unter Übergabe dieses Klassennamens aufruft,
    - das einen gegebenenfalls enthaltenen Directory-Pfad der `CLASSPATH`-Environment-Variablen hinzufügt
    - und das Programmparameter, die der `class`-Datei eventuell übergeben werden, an den Aufruf der JVM weiterreicht

Ein derartiges Wrapper-Programm lässt sich z.B. sehr einfach als **Shell-Skript** realisieren.

- ◇ Wrapper-Programm **javawrapper** (sinnvollerweise in einem Execution-Dir enthalten, z.B. `/usr/local/bin`)

```
#!/bin/sh
CLASS=$1

# if classname is a link, we follow it (this could be done easier - how?)
if [ -L "$1" ] ; then
    CLASS=`ls --color=no -l $1 | tr -s '\t ' ' ' | cut -d ' ' -f 11`
fi
CLASSN=`basename $CLASS .class`
CLASSP=`dirname $CLASS`

FOO=$PATH
PATH=$CLASSPATH
if [ -z "`type -p -a $CLASSN.class`" ] ; then
    # class is not in CLASSPATH
    if [ -e "$CLASSP/$CLASSN.class" ] ; then
        # append dir of class to CLASSPATH
        if [ -z "${CLASSPATH}" ] ; then
            export CLASSPATH=$CLASSP
        else
            export CLASSPATH=$CLASSP:$CLASSPATH
        fi
    else
        # uh! now we would have to create a symbolic link - really
        # ugly, i.e. print a message that one has to change the setup
        echo "Hey! This is not a good setup to run $1 !"
        exit 1
    fi
fi
PATH=$FOO

shift
/usr/java/default/bin/java $CLASSN $@
```

- ◇ **Registrierung** von `class`-Dateien als ausführbares (Binär-)Format

```
echo ':Java:M::\xca\xfe\xba\xbe::/usr/local/bin/javawrapper:' >
    /proc/sys/fs/binfmt_misc/register
```

## LINUX System Call `_exit`

- **Funktionalität :** **Beendigung** eines Prozesses :  
Alle vom Prozess belegten **Ressourcen** werden **freigegeben** (Speicher, IPC-Datenstrukturen, Signalhandler-Tabelle usw)  
Alle dem Prozess gehörenden **File-Deskriptoren** werden geschlossen.  
Alle **Kindprozesse** werden vom **Init-Prozess adoptiert**.  
An den **Elternprozess** wird das **Signal SIGCHLD** (*child status changed*) gesandt.  
Der als Parameter übergebene Wert *status* wird als **Exit-Code** dem **Elternprozess** verfügbar gemacht (Ermittlung durch Elternprozess mit einem der `wait` System Calls)  
Überführung des Prozesses in den Zustand **TASK\_ZOMBIE** (Prozesslisteneintrag bleibt bestehen, bis der Elternprozess das Prozessende mit einem der `wait` System Calls erfragt hat)  
Aufruf des **Schedulers** → endgültige Freigabe der CPU durch den Prozess

- **Interface :**

```
void _exit(int status);
```

- **Header-Datei :** `<unistd.h>`
- **Parameter :** *status* Exit-Code des Prozesses (wird dem Elternprozess verfügbar gemacht)
- **Rückgabewert :** Die Funktion kehrt nicht zum Aufrufer zurück
- **Implementierung :** System Call Nr. 1  
→ `sys_exit(...)` (in `kernel/exit.c`)  
→ `do_exit(...)` (in `kernel/exit.c`)
- **Anmerkungen:**
  1. Andere Möglichkeit zur Beendigung eines Prozesses :  
**Empfang** eines "tödlich" wirkenden **Signals** (z.B. `SIGKILL`)
  2. Der System Call `_exit()` wird von der ANSI-C-Standardbibliotheksfunktion `exit()` und durch die Beendigung von `main()` aufgerufen, nachdem diese alle mit der ANSI-C-Standardbibliotheksfunktion `atexit()` registrierten Funktionen aufgerufen haben
  3. Bei einem direkten Aufruf von `_exit()` werden die mit `atexit()` registrierten Funktionen nicht aufgerufen.

## LINUX System Call `wait4`

- **Funktionalität :** Warten auf die **Beendigung** eines **Kindprozesses**.

Der System Call **blockiert** den aufrufenden Prozess bis ein durch den Parameter `pid` spezifizierter Kindprozess endet (bzw bis der Prozess ein von ihm zu behandelndes Signal bzw ein ihn beendendes Signal erhält).

(Der Prozess geht in den Zustand `TASK_INTERRUPTABLE`, aus dem ihn der Empfang eines Signals, z.B. `SIGCHLD`, wieder in den Zustand `TASK_RUNNING` überführt).

Falls zum Zeitpunkt des System Calls ein durch den Parameter `pid` spezifizierter Kindprozess bereits beendet ist (→ "Zombie"), kehrt `wait4()` sofort zurück.

Der System Call kehrt ebenfalls sofort zurück, falls kein durch den Parameter `pid` spezifizierter Kindprozess existiert

- **Interface :**

```
pid_t wait4(pid_t pid, int* pstat, int options, struct_rusage* rusage);
```

- **Header-Datei :** `<sys/resource.h>`  
`<sys/wait.h>`

- **Parameter :**
  - `pid` Spezifikation der Prozesse, auf die gewartet werden soll.  
Zulässige Werte :
    - `<-1` Es soll auf irgendeinen Kindprozess, dessen PGID gleich dem Betrag von `pid` ist, gewartet werden
    - `-1` Es soll auf irgendeinen Kindprozess gewartet werden
    - `0` Es soll auf irgendeinen Kindprozess gewartet werden, dessen PGID gleich der des aufrufenden Prozesses ist
    - `>0` Es soll auf den Prozess mit der `PID==pid` gewartet werden
  - `pstat` Pointer auf eine Variable, in die Status-Information über die Prozessbeendigung abgelegt werden soll  
Diese Status-Information enthält Angaben über die Beendigungsart des Kindprozesses und – bei Beendigung mit `_exit()` – dessen Exit-Code.
  - `options` Angabe, wie der System Call in speziellen Fällen reagieren soll.  
Mögliche Werte sind `0` oder die bitweise ODER-Verknüpfung von :
    - WNOHANG** Sofortige Rückkehr des System Calls, wenn kein Kindprozess zum Zeitpunkt des Aufrufs beendet ist
    - WUNTRACED** Der System Call soll auch zurückkehren, wenn ein durch `pid` spezifizierter Prozess gestoppt ist
  - `rusage` Pointer auf eine Variable, in der Information über die Ressourcenverwendung des beendeten Kindprozesses (und aller seiner Kinder) abgelegt werden soll
- **Rückgabewert :**
  - **PID** des beendeten bzw gestoppten Prozesses
  - `0`, wenn das `options`-Flag **WNOHANG** gesetzt war und **kein Prozess beendet** ist
  - `-1` im **Fehlerfall** (insbesondere, wenn kein durch `pid` spezifizierter Prozess existiert)

- **Implementierung :** System Call Nr. **114**  
→ `sys_wait4(...)` (in `kernel/exit.c`)

- **Anmerkungen:**
  1. Der Datentyp `pid_t` ist in der Headerdatei `<sys/wait.h>` **definiert** als `int`.
  2. Es existiert noch ein weiterer System Call zum Warten auf das Ende eines Kindprozesses (`waitpid`), der intern aber auch durch `wait4` realisiert wird.

## LINUX System Call `waitpid`

- **Funktionalität :** **Warten** auf die **Beendigung** eines **Kindprozesses**.

Der System Call **blockiert** den aufrufenden Prozess bis ein durch den Parameter `pid` spezifizierter Kindprozess endet (bzw bis der Prozess ein von ihm zu behandelndes Signal bzw ein ihn beendendes Signal erhält).

(Der Prozess geht in den Zustand `TASK_INTERRUPTABLE`, aus dem ihn der Empfang eines Signals, z.B. `SIGCHLD`, wieder in den Zustand `TASK_RUNNING` überführt).

Falls zum Zeitpunkt des System Calls ein durch den Parameter `pid` spezifizierter Kindprozess bereits beendet ist (→ "Zombie"), kehrt `waitpid()` sofort zurück.

Der System Call kehrt ebenfalls sofort zurück, falls kein durch den Parameter `pid` spezifizierter Kindprozess existiert

- **Interface :**

```
pid_t waitpid(pid_t pid, int* pstat, int options);
```

- **Header-Datei :** `<sys/wait.h>`

- **Parameter :**
    - `pid` Spezifikation der Prozesse, auf die gewartet werden soll.  
Zulässige Werte : s. System Call `wait4()`
    - `pstat` Pointer auf eine Variable, in die Status-Information über die Prozessbeendigung abgelegt werden soll  
Diese Status-Information enthält Angaben über die Beendigungsart des Kindprozesses und – bei Beendigung mit `_exit()` – dessen Exit-Code.
    - `options` Angabe, wie der System Call in speziellen Fällen reagieren soll.  
Mögliche Werte : s. System Call `wait4()`

- **Rückgabewert :**
    - **PID** des beendeten bzw gestoppten Prozesses
    - **0**, wenn das `options`-Flag **WNOHANG** gesetzt war und **kein Prozess beendet** ist
    - **-1** im **Fehlerfall** (insbesondere, wenn kein durch `pid` spezifizierter Prozess existiert)

- **Implementierung :** System Call Nr. **7**

- `sys_waitpid(...)` (in `kernel/exit.c`)
  - `sys_wait4(...)` (in `kernel/exit.c`)

- **Anmerkungen:**
  1. Der Datentyp `pid_t` ist in der Headerdatei `<sys/wait.h>` **definiert** als `int`.
  2. Dieser System Call wird nur noch aus Kompatibilitätsgründen bereitgestellt.  
Der Aufruf `waitpid(pid, pstat, options)`  
entspricht `wait4(pid, pstat, options, NULL)`

- **Weitere C-Bibliotheks-Funktion zum Aufruf dieses System Calls**

In der C-Bibliothek existiert eine weitere Funktion zum Aufruf dieses System Calls, mit der auf das Ende eines **beliebigen Kindprozesses** gewartet werden kann (Headerdatei `<sys/wait.h>` :

```
pid_t wait (int* pstat);
```

Diese Funktion ist definiert zu

```
inline pid_t wait(int* pstat)
{
    return waitpid(-1, pstat, 0);
}
```



## Scheduling in LINUX (1)

### • Allgemeines

- ◇ Der Scheduler hat die Aufgabe, die **CPU-Zeit** auf die einzelnen **rechenbereiten Prozesse** (Zustand `TASK_RUNNING`) **aufzuteilen**. Alle rechenbereiten Prozesse befinden sich in der **Run Queue**.  
Ab dem Kernel 2.6 wurde der Scheduler als **O(1)-Scheduler** implementiert, d.h. die Zeit zur Auswahl des jeweils nächsten zu aktivierenden Prozesses ist unabhängig von der Anzahl der ablaufbereiten Prozesse.  
Mit der Kernel-Version 2.6.23 wurde dieser Scheduler durch einen *Completely Fair Scheduler* ersetzt, der eine leichte Abhängigkeit von der Anzahl der ablaufbereiten Prozesse besitzt ( $O(\log n)$ ).  
Im folgenden wird der ältere O(1)-Scheduler betrachtet.
- ◇ Der LINUX-Scheduler arbeitet **prioritätsgesteuert** mit unterlagertem **Zeitscheibenverfahren**, wobei er **drei verschiedene Scheduling-Algorithmen** (Scheduling-Klassen !) kombiniert. Seine **Hauptaufgaben** sind :
  - ▷ **Überwachung der Zeitscheiben** → Feststellung eines durch **Zeitscheibenablauf** notwendigen **Taskwechsels**
  - ▷ Auswahl des **nächsten zu aktivierenden Prozesses** aus der *Run Queue* bei einem anstehenden **Taskwechsel**
- ◇ Die Funktionalität des Schedulers von LINUX wird im wesentlichen durch zwei in der Datei `kernel/sched.c` definierte Kernel-Funktionen realisiert :
  - ▷ `void scheduler_tick()` ("periodischer Scheduler")
  - ▷ `void schedule()` ("Haupt-Scheduler")
- ◇ Die "periodische Scheduler"-Funktion `scheduler_tick()` überwacht die Zeitscheiben .  
Bei **Ablauf der Zeitscheibe** des jeweils aktuellen Prozesses **setzt** sie dessen `TIF_NEED_RESCHED`-Flag und kennzeichnet somit die Notwendigkeit eines Prozesswechsels.  
Diese Funktion wird bei **jedem Timer-Interrupt** ("*timer tick*") durch die Timer-ISR **aufgerufen**.
- ◇ Die "Haupt-Scheduler"-Funktion `schedule()` wählt den nächsten zu aktivierenden Prozess aus.  
Sie wird immer dann **aufgerufen**, wenn ein **Prozesswechsel notwendig** wird.  
Dies wird im wesentlichen in zwei verschiedenen Situationen im System der Fall sein :
  - ▷ Wenn der **aktuelle Prozess** die **CPU abgeben** muß, weil er auf ein **Ereignis warten** muß (d.h. in eine Warteschlange gestellt wird). Dies erfolgt i.a. während der Abarbeitung eines **System Calls**, mit dem der aktuelle Prozess etwas anfordert (z.B. I/O), indirekt durch die Funktion `sleep_on()` .
  - ▷ Am **Ende** jedes **System Calls** und jeder **Interrupt-Bearbeitung** (Sprung zur Marke `ret_from_intr`: in der Routine `system_call()`) wenn für die aktuelle Task das `TIF_NEED_RESCHED`- Flag **gesetzt** ist.  
U.a. wird dieses Flag gesetzt, wenn die Zeitscheibe für den aktuellen Prozess abgelaufen ist (von der Funktion `scheduler_tick()`) oder dieser die CPU freiwillig abgibt (System Call `sched_yield()`) . Da zumindest der **Timer-Interrupt** regelmässig (alle 1 msec == Timer-Tick) aufgerufen und dabei gegebenenfalls (Ablauf der Zeitscheibe) das `TIF_NEED_RESCHED`-Flag durch die "periodische Scheduler"-Funktion gesetzt wird, ist sichergestellt, daß die "Haupt-Scheduler"-Funktion **regelmässig** aufgerufen wird (⇒ **preemptives Multitasking**)

### • Prozess-Arten und Scheduling-Klassen (*Scheduling Policy*)

- ◇ Die **drei** im Scheduler kombinierten **Scheduling-Algorithmen** korrespondieren mit **drei verschiedenen Scheduling-Klassen**. Jeder Prozess ist einer dieser drei Klassen zugeordnet (→ Komponente **policy** des Prozessdeskriptors).
- ◇ Abhängig von der zugeordneten Scheduling-Klasse werden zwei prinzipielle **Prozess-Arten** in LINUX unterschieden :
  - ▷ **"normale" Prozesse** : ⇒ Scheduling-Klasse `SCHED_NORMAL`
  - ▷ **"Realtime"-Prozesse** : ⇒ Scheduling-Klassen `SCHED_FIFO` und `SCHED_RR`
- ◇ Nur Prozesse mit **Root-Berechtigung** können "Realtime"-Prozesse sein
- ◇ "Realtime"-Prozesse haben grundsätzlich eine höhere Priorität als "normale" Prozesse und werden – sofern sie ablaufbereit sind – immer vor diesen ausgeführt.  
→ Einem "normalen" Prozess wird nur dann die CPU zugeteilt, wenn kein "Realtime"-Prozeß ablaufbereit ist.
- ◇ Die beiden **Scheduling-Klassen** für "Realtime"-Prozesse **unterscheiden** sich hinsichtlich der **Zuordnung von Zeitscheiben**.
- ◇ Die die Scheduling-Klassen **kennzeichnenden Konstanten** sind definiert in `include/linux/sched.h`.

## Scheduling in LINUX (2)

### • Eigenschaften der Scheduling-Klassen

- ◇ Ein Prozess **erbt** die Scheduling-Klasse seines Elternprozesses.
- ◇ Sie kann – zusammen mit der "Realtime"-Priorität – mittels des System Calls
  - ▷ `int sched_setscheduler(pid_t pid, int policy, const struct sched_param* p);`für den aktuellen oder einen anderen Prozess **verändert** werden.  
Ein Setzen der Scheduling-Klassen `SCHED_FIFO` oder `SCHED_RR` setzt Root-Berechtigung voraus.
- ◇ Mittels des System Calls
  - ▷ `int sched_getscheduler(pid_t pid);`kann die Scheduling-Klasse des aktuellen oder eines anderen Prozesses **ermittelt** werden.
- ◇ **Scheduling-Klasse `SCHED_NORMAL`** (*Default LINUX Time-sharing Scheduling*)

Für Prozesse dieser Scheduling-Klasse muß die "**Realtime**"-Priorität `==0` sein ("normale" Prozesse).  
Ihre **dynamische Priorität** liegt zwischen `MAX_RT_PRIO` (`==100`) (höchste Priorität) ... `MAX_PRIO-1` (`==139`) (niedrigste Priorität)  
Einem Prozess der Klasse `SCHED_NORMAL` wird erst dann die **CPU zugeteilt**, wenn es **keinen rechenbereiten "Realtime"-Prozess** (dynamische Priorität `<= MAX_RT_PRIO-1` (`==99`)) gibt.  
Von den rechenbereiten `SCHED_NORMAL`-Prozessen wird immer der **erste Prozess höchster dynamischer Priorität** ausgewählt, dessen **Zeitscheibe noch nicht abgelaufen** ist.  
Einem Prozess, dessen **Zeitscheibe abgelaufen** ist, wird die **CPU entzogen**.  
Erst wenn für **alle anderen Prozesse** dieser Scheduling-Klasse – auch niedrigerer dynamischer Priorität – die **Zeitscheibe abgelaufen** ist, kann einem Prozess **erneut die CPU zugeteilt** werden.
- ◇ **Scheduling-Klasse `SCHED_RR`** (*Round Robin Scheduling*)

Für Prozesse dieser Scheduling-Klasse muß die "**Realtime**"-Priorität `>=1` betragen ("Realtime"-Prozesse).  
Ihre **dynamische Priorität** hat einen Wert `<= MAX_RT_PRIO-1` (`==99`)  
Wenn ein Prozess der Klasse `SCHED_RR` rechenbereit wird, **verdrängt** er beim nächsten Scheduling jeden laufenden **Prozess niedrigerer dynamischer Priorität** (also auch jeden "normalen" Prozess).  
Prozesse **gleicher dynamischer Priorität** werden in der **Reihenfolge**, in der sie in die **Run Queue** aufgenommen wurden, abgearbeitet.  
Nach **Ablauf seiner Zeitscheibe** wird einem `SCHED_RR`-Prozess die **CPU entzogen**.  
Der Prozess wird an das **Ende der Run Queue** (genauer das Ende der seiner Priorität zugeordneten Teilliste der Run Queue) gestellt. Er kann erst dann wieder **weiterlaufen**, wenn **alle anderen** rechenbereiten `SCHED_RR`-Prozesse der **gleichen** dynamischen Priorität **ebenfalls ihre Zeitscheibe aufgebraucht** haben.  
(→reines "**Round Robin**" für Prozesse gleicher dynamischer Priorität)  
Wenn ein Prozess dieser Scheduling-Klasse vor Ablauf seiner Zeitscheibe durch einen Prozess höherer dynamischer Priorität **verdrängt** wurde, **verbleibt** er an seiner **Position** in der **Run Queue**, so dass er seine Zeitscheibe beenden kann, bevor anderen Prozessen mit gleicher dynamischer Priorität die CPU zugeteilt wird.
- ◇ **Scheduling-Klasse `SCHED_FIFO`** (*First in First out Scheduling*)

Bezüglich ihrer **Priorität** gilt das **gleiche** wie für Prozesse der Klasse `SCHED_RR`.  
Im Unterschied zu diesen existieren für sie aber **keine Zeitscheiben**.  
Ein die CPU besitzender `SCHED_FIFO`-Prozess **läuft** daher **solange**, bis er
  - die CPU wegen des Wartens auf ein Ereignis freigeben muß
  - oder die CPU freiwillig abgibt (System Call `sched_yield()`)
  - oder beendet ist
  - oder durch einen Prozess mit höherer dynamischer Priorität verdrängt wird.Ein Prozess der Klasse `SCHED_FIFO`, der durch einen Prozess **höherer** dynamischer **Priorität verdrängt** wurde, **verbleibt** an seiner **Position** in der **Run Queue**. Dadurch läuft er vor allen anderen Prozessen der gleichen dynamischen Priorität, die nach ihm in die **Run Queue** aufgenommen worden sind, weiter.

## Scheduling in LINUX (3 - 1)

### • Prozess-Prioritäten

LINUX unterscheidet **drei verschiedene Prioritäten**, die jedem Prozess zugeordnet werden und in entsprechenden **Komponenten des Prozessdeskriptors** abgelegt sind. :

- ▷ **Statische Priorität** (Komponente `static_prio`)
- ▷ **Dynamische Priorität** (Komponente `prio`)
- ▷ **"Realtime"-Priorität** (Komponente `rt_priority`)

### • "Realtime"-Priorität

- ◇ Die "Realtime"- Priorität eines Prozesses ist in der Komponente `rt_priority` seines Prozessdeskriptors enthalten.
- ◇ Sie ist eine ganze Zahl im Bereich `0 ... MAX_USER_RT_PRIO-1 (== 99)`.  
Im Unterschied zu den beiden anderen Prioritäts-Arten bedeutet hier eine **grössere Zahl** eine **höhere Priorität**  
Für "**normale**" Prozesse hat die "Realtime"-Priorität immer den Wert `0`.  
Für "**Realtime**"-Prozesse liegt sie im Bereich `1 ... 99`.  
Die "Realtime"-Priorität unterscheidet damit auch "normale" Prozesse von "Realtime"-Prozessen
- ◇ Für "**Realtime**"-Prozesse legt sie die **dynamische Priorität** fest.  
Es gilt die Beziehung :  
$$\text{dynamische Priorität} = \text{MAX\_USER\_RT\_PRIO} - 1 - \text{"Realtime"-Priorität}$$
- ◇ Ein Prozess **erbt** die "Realtime"-Priorität seines Elternprozesses.
- ◇ Mit den System Calls
  - ▷ `int sched_setscheduler(pid_t pid, int policy, const struct sched_param* p)`
  - ▷ `int sched_setparam(pid_t pid, const struct sched_param* p)`kann die "Realtime"- Priorität (und dynamische Priorität) des aktuellen oder eines anderen Prozesses **geändert** werden.  
Ein Setzen der "Realtime"- Priorität auf einen Wert `>0` ist nur für die Scheduling-Klassen `SCHED_FIFO` und `SCHED_RR` zulässig und erfordert **Root-Berechtigung**.

### • Statische Priorität

- ◇ Die statische Priorität eines Prozesses ist in der Komponente `static_prio` seines Prozessdeskriptors abgelegt.
- ◇ Sie ist eine ganze Zahl im Bereich `MAX_RT_PRIO (==100)` (höchste Priorität) ... `MAX_PRIO-1 (==139)` (niedrigste Priorität).
- ◇ Die statische Priorität bestimmt die **Grösse der Zeitscheibe**.  
Je kleiner ihr Wert ist (grössere Priorität !), desto grösser ist die Zeitscheibe.  
Zeitscheiben liegen im Bereich `5 ms ... 800 ms` (Standardwert: `100 ms` für `static_prio==120`)
- ◇ Bei "**normalen**" Prozessen bildet die statische Priorität ausserdem den **Ausgangswert** für die **dynamische Priorität**
- ◇ Die statische Priorität ist eng mit dem – vom User-Mode aus setzbaren – **nice-Wert** verknüpft  
Der **nice-Wert** ist eine traditionell in UNIX-Systemen verwendete ganze Zahl zur Prioritätskennzeichnung.  
Er liegt zwischen `-20` (hohe Priorität) und `+19` (niedrige Priorität).  
Es gilt die Beziehung  
$$\text{statische Priorität} = \text{MAX\_RT\_PRIO} + \text{nice} + 20$$
- ◇ Ein Prozess **erbt** die statische Priorität seines Elternprozesses.
- ◇ Mit den System Calls
  - ▷ `int nice(int inc)`
  - ▷ `int setpriority(int which, int who, int prio)`kann die statische Priorität des aktuellen Prozesses bzw beliebiger Prozesse **geändert** werden.  
Eine **Erhöhung** der statischen Priorität erfordert bei beiden System Calls **Root-Berechtigung**.



## Scheduling in LINUX (3 - 2)

### • Dynamische Priorität

- ◇ Die dynamische Priorität eines Prozesses ist in der Komponente **prio** seines Prozessdeskriptors abgelegt.
- ◇ Sie ist ein ganze Zahl im Bereich **0** (höchste Priorität) ... **MAX\_PRIO-1** (==139) (niedrigste Priorität). Der Bereich **0** ... **MAX\_RT\_PRIO-1** (==99) ist den "Realtime"-Prozessen vorbehalten.
- ◇ Die dynamische Priorität bestimmt die **Zuteilungsreihenfolge der CPU** an die rechenbereiten Prozesse. Prozesse mit **höherer dynamischer Priorität (kleinerer Prioritätszahl)** werden immer gegenüber solchen mit niedrigerer dynamischer Priorität **bevorzugt**.
- ◇ Für "**normale**" Prozesse wird die dynamische Priorität aus der statischen Priorität unter Berücksichtigung der **Interaktivität** des jeweiligen Prozesses ermittelt.  
Dies erfolgt mittels der in `kernel/sched.c` definierten Kernel-Funktion :  

```
static int effective_prio(task_t *p)
```

**Interaktive Prozesse** bekommen eine **höhere dynamische Priorität**.  
Dabei wird **nach jedem Ablauf einer Zeitscheibe** eines Prozesses dessen dynamische Priorität **neu bestimmt**.
- ◇ Für "**Realtime**"-Prozesse ist die dynamische Priorität durch die "**Realtime**"-Priorität festgelegt (s. oben).

### • Zeitscheibenwert (Größe einer Zeitscheibe)

- ◇ Der Zeitscheibenwert beschreibt die Größe einer Zeitscheibe in **Timer-Ticks**.  
Ein Timer-Tick ist gleich der – architekturabhängigen – **Periodendauer** des **Timer-Interrupts**.  
Bei der x86-Architektur beträgt diese ab dem Kernel 2.6 **1 ms**  
(HZ=1000, definiert in `include/asm/param.h`)
- ◇ Der Zeitscheibenwert eines Prozesses wird durch dessen **statische Priorität** festgelegt (Ausnahme : erste Zeitscheibe).  
Zu seiner Ermittlung dient die in `kernel/sched.c` definierte Kernel-Funktion  

```
static unsigned int task_timeslice(task_t *p)
```

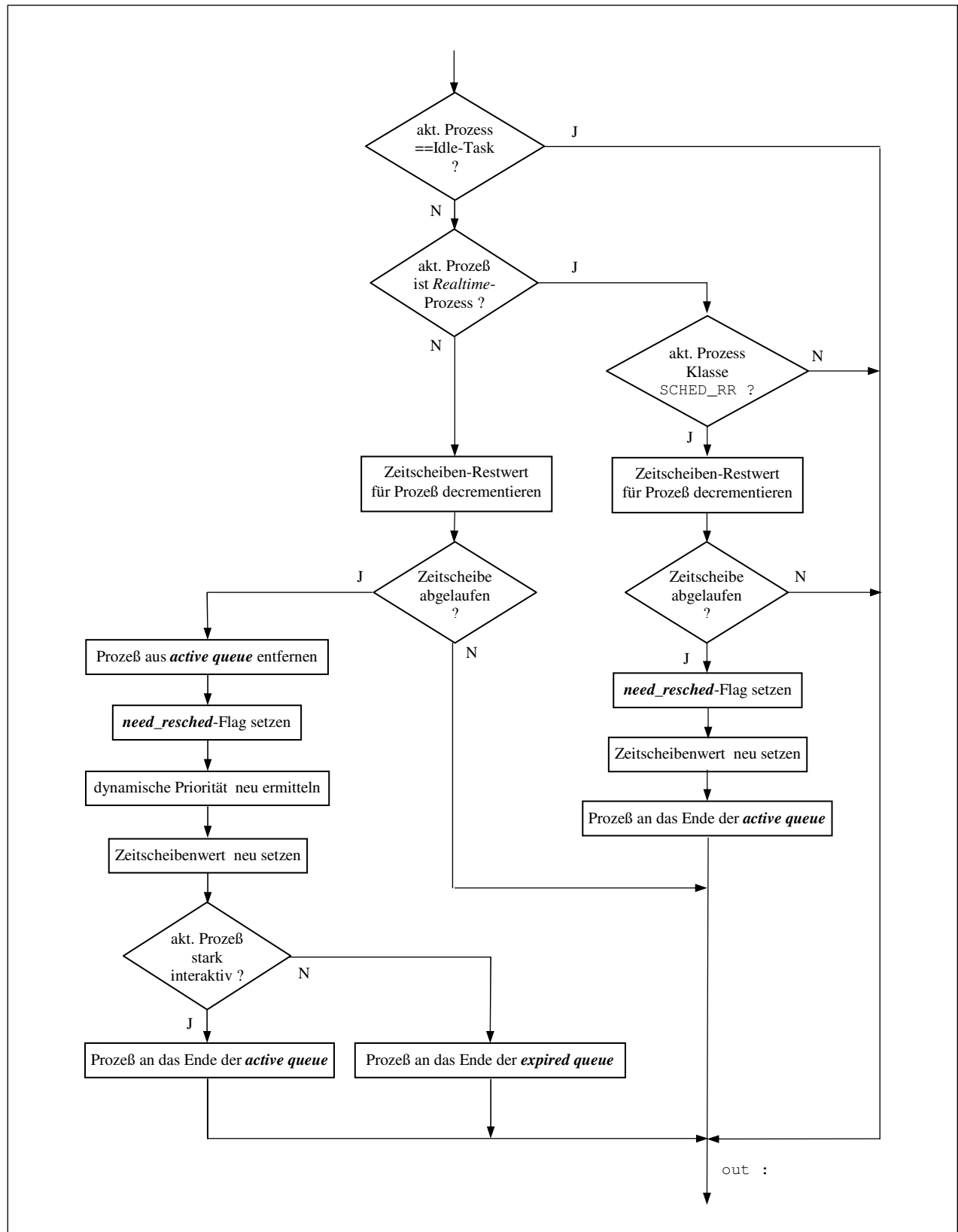
  
Die Funktion wird nach jedem Ablauf einer Zeitscheibe für den betreffenden Prozess erneut aufgerufen.  
Je grösser die statische Priorität ist (kleinerer Wert von `static_prio`), desto grösser ist der Zeitscheibenwert.
- ◇ Der Zeitscheibenwert eines Prozesses wird vor Beginn einer Zeitscheibe in der Komponente **time\_slice** seines Prozessdeskriptors abgelegt.  
Mit **jedem Timer-Interrupt** wird diese Komponente für den jeweils aktuell laufenden Prozess **um 1 decrementiert**.  
(in der durch die Timer-ISR aufgerufenen "periodischen Scheduler"-Funktion `scheduler_tick()`)  
Eine **Zeitscheibe** ist **abgelaufen**, wenn der Wert dieser Komponente **0** geworden ist.
- ◇ Die **erste Zeitscheibe** eines neu erzeugten Prozesses wird durch **Teilung** des im Elternprozess aktuell vorhandenen Zeitscheibenrests gewonnen.  
Nach der Erzeugung eines Prozesses verbleibt dem Elternprozess also nur noch die Hälfte seines zum Zeitpunkt der Prozesserschöpfung vorhandenen Zeitscheibenrests.

### • Einige weitere scheduler-relevante Komponenten des Prozessdeskriptors

- ◇ **timestamp** speichert den **Zeitpunkt** (in Jiffies = Timer-Ticks seit System-Start) zu dem der **Prozess zuletzt gelaufen** ist.
- ◇ **sleep\_avg** ("Schlafguthaben") gibt an, wie häufig und wie lange sich der Prozess in einem Wartezustand befunden hat. Der Wert ist ein **Mass** für die **Interaktivität** des Prozesses.  
Beim Übergang des Prozesses in den ablaufbereiten Zustand wird dieser Wert um die Zeitdauer (in Timer-Ticks), die der Prozess im Wartezustand verbracht hat, erhöht. Bei der Freigabe der CPU durch den Prozess (Prozesswechsel) wird der Wert um die Zeitdauer (in Timer-Ticks), die der Prozess gelaufen ist, erniedrigt.
- ◇ **first\_time\_slice** gibt an, ob sich der Prozess in seiner **ersten Zeitscheibe** befindet ( ==1) oder nicht ( == 0)

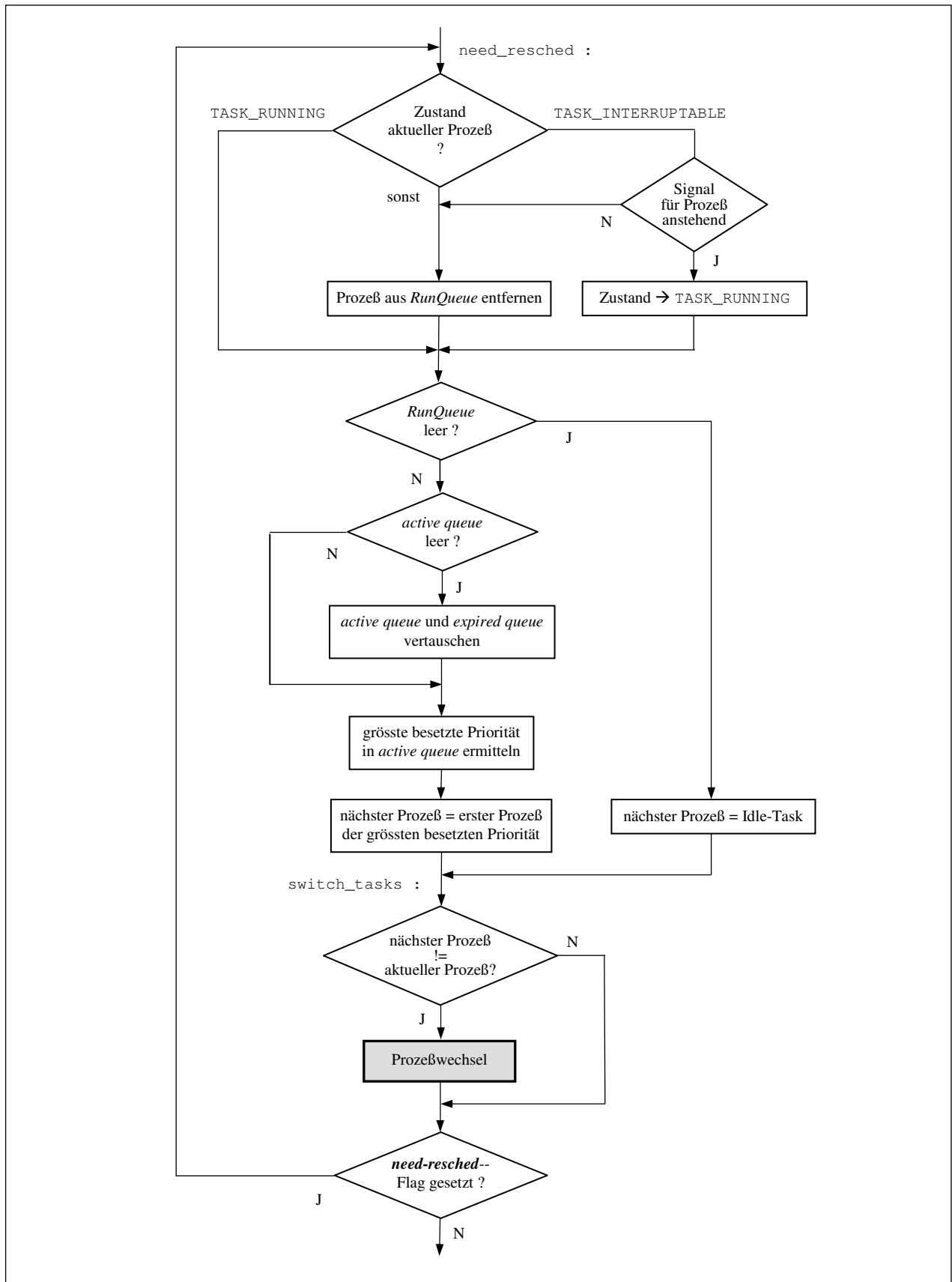
## Prinzipielle Arbeitsweise des Schedulers von LINUX (1)

- "Periodischer Scheduler" `scheduler_tick()`



## Prinzipielle Arbeitsweise des Schedulers von LINUX (2)

- "Haupt-Scheduler" `schedule()`



# **Betriebssysteme**

## **Kapitel 8**

### **8. Arbeitsspeicherverwaltung in LINUX**

- 8.1. Allgemeines
- 8.2. Virtueller Arbeitsspeicher eines Prozesses
- 8.3. Verwaltung des physikalischen Speichers
- 8.4. Paging

## Arbeitsspeicherverwaltung in LINUX – Allgemeines (1)

### • Virtuelle Speichertechnik

- ◇ Als multitaskingfähiges Betriebssystem setzt LINUX **virtuelle Speichertechnik** ein.  
Sein Speicherverwaltungssystem muß eine **Abbildung** der **virtuellen Adressen** auf die realen **physikalischen Adressen** realisieren.  
→ Die Speicherverwaltung von LINUX umfaßt :
  - Verwaltung des virtuellen Speichers
  - Verwaltung des physikalischen Speichers
- ◇ Das Speicherverwaltungssystem stellt die folgenden Fähigkeiten bereit :
  - ▷ großer virtueller Adressraum
  - ▷ Schutz vor unzulässigen Speicherzugriffen
  - ▷ Einblenden von Dateien in den Speicheradressraum
  - ▷ "faire" Zuteilung des physikalischen Speichers an die einzelnen Prozesse
  - ▷ gemeinsame Nutzung von physikalischen Speicherbereichen durch mehrere Prozesse (*Shared Memory*)
- ◇ LINUX realisiert die virtuelle Speichertechnik mittels **Paging**.  
Die Umsetzung der virtuellen Adressen in die physikalischen Adressen erfolgt über **Seitentabellen**  
Wird in einem Prozess eine – virtuelle – Adresse referiert, für die kein Eintrag in einer Seitentabelle existiert, erzeugt der Prozessor eine **Page Fault Exception**. Der dadurch aktivierte **Page Fault Handler** versucht die Seite mit der referierten Adresse in den physikalischen Arbeitsspeicher zu laden.

### • Virtueller Adressraum

- ◇ Der virtuelle Adressraum ist häufig größer als der tatsächlich zur Verfügung stehende physikalische Adressraum  
Seine Größe hängt von der jeweiligen CPU-Architektur ab.  
Bei **32-Bit**-Prozessoren beträgt sie i.a. (ohne Einsatz von Segmentierung) **4 GBytes**.
- ◇ Der virtuelle Adressraum ist aufgeteilt in
  - einen **Systembereich** (*kernel space*, Kernel-"Segment") für Code und Daten des Betriebssystems und
  - einen **Nutzerbereich** (*user space*, User-Bereich, Nutzer-"Segment") für Code und Daten der Anwenderprozesse.

Der **Systembereich** befindet sich im **oberen Teil** des gesamten virtuellen Adressraums.  
Seine Startadresse und seine Größe sind architekturabhängig.

Für die **x86-Prozessoren** gilt : Standardmäßige Startadresse : **0xC0000000** (=3 GBytes)

⇒ Nutzerbereich 0x00000000 .. 0xBFFFFFFF (3 GBytes)

Systembereich 0xC0000000 .. 0xFFFFFFFF (1 GByte)

Änderung über Konstante `PAGE_OFFSET` in "`include/asm/page.h`" möglich

- ◇ **Jedem Prozess** steht der Nutzerbereich als **eigener virtueller Adressraum** zur Verfügung.  
→ Jeder Prozess verfügt über eigene Seitentabellen.

### • Physikalischer Speicher

- ◇ Beim System-Start wird der **gesamte physikalische Arbeitsspeicher** bis zur Größe des Systembereichs (bei IA32 : 1 Gbytes) in den virtuellen **Adressraum des Kernels** (Systembereich) eingeblendet (ab `PAGE_OFFSET`, i.a. ist das der Beginn des Systembereichs).  
→ **physikalische Adresse == virtuelle (System-)Adresse – PAGE\_OFFSET**  
Der gegebenenfalls darüberhinausgehende Speicher wird als **HighMem** bezeichnet.
- ◇ Die vom **Kernel-Code** und den **Kernel-Daten** einschließlich der **Kernel-Seitentabellen** belegten Seiten dürfen **niemals** ausgelagert werden.  
Sie sind daher als **reserviert** und damit als nicht-swapbar gekennzeichnet.

## Arbeitsspeicherverwaltung in LINUX – Allgemeines (2)

### • Speicherschutz

- ◇ Da jeder Prozess in einem eigenen virtuellen Adressraum läuft, sind die von den Prozessen verwendeten – virtuellen - Adressen vollständig voneinander entkoppelt.  
→ Der Speicher eines Prozesses ist **vor dem Zugriff durch einen anderen Prozess geschützt**.
- ◇ Code kann im **Nutzer-Modus** (User-Modus) oder im höherprivilegierten **System-Modus** (Kernel-Modus) abgearbeitet werden  
Für LINUX auf **x86-Prozessoren** gilt :
  - Nutzer-Modus : Privileg-Ebene ("Ring") **3**
  - System-Modus : Privileg-Ebene ("Ring") **0**

**Betriebssystem-** (Kernel-)Code läuft im **System-Modus**.

**Anwenderprogramm**-Code läuft im **Nutzer-Modus**. In diesem Modus ist ein Zugriff zum Speicherbereich des Kernels (Systembereich) nicht möglich.

→ Der **Kernel-Speicherbereich** ist **vor dem Zugriff durch Anwenderprogramme geschützt**.

Üblicherweise startet jeder Nutzer-Prozess im Nutzer-Modus.

Ein Zugriff zu Systemressourcen ist nur durch einen Übergang in den System-Modus möglich.

Dies erfolgt durch - mittels Software-Interrupts realisierte - System Calls. Ein System Call bewirkt die Abarbeitung von - im System-Modus ablaufenden - Betriebssystem-Code

Analog bewirkt auch die Abarbeitung von Hardware-Interrupt- und Exception-Service-Routinen einen Übergang in den System-Modus.

- ◇ Der Speicher eines Prozesses ist **vor einem Zugriff mittels einer unzulässigen Zugriffsart durch den eigenen Prozess geschützt**.  
Beispielsweise ist ein schreibender Zugriff zu Code-Speicherbereichen oder zu Read-Only-Daten-Speicherbereichen unzulässig.
- ◇ Die **Überprüfung auf Speicherschutzverletzungen** erfolgt i.a. durch die Hardware (MMU).  
Beim Auftritt einer Speicherschutzverletzung löst die MMU eine Exception aus, die von einem im Kernel implementierten Exception-Handler bedient werden muß.  
In nicht von ihm selbst behandelbaren Fällen sendet der Exception Handler das Signal `SIGSEGV` an den verursachenden Prozess. Wenn dieser hierfür keinen Signal-Handler installiert hat, wird der Prozess beendet.

### • Shared Memory

- ◇ Trotz getrennter virtueller Adressräume für verschiedene Prozesse, ist es möglich, daß sich **mehrere Prozesse gemeinsame physikalische Speicherbereiche teilen** :  
In den **Seitentabellen verschiedener Prozesse** verweisen Einträge auf die **gleichen physikalischen Seiten**.  
→ Der entsprechende **physikalische Speicherbereich** ist in den **virtuellen Speicherbereich mehrerer Prozesse** eingeblendet. Dies kann – und wird häufig auch – unter verschiedenen virtuellen Adressen der Fall sein.
- ◇ Shared Memory findet u.a. Anwendung für :
  - ▷ Threads
  - ▷ Zustand nach `fork()` (→ "Copy-on-write")
  - ▷ Verwendung derselben Code-Bereiche und Read-Only-Daten-Bereiche wenn das gleiche Programm durch mehrere Prozesse ausgeführt wird.
  - ▷ Shared Libraries
  - ▷ Interprozeß-Kommunikation

## Arbeitsspeicherverwaltung in LINUX – Allgemeines (3)

### • Anmerkung zur Segmentierung bei der x86-Architektur

- Bei Prozessoren der x86-Architektur wird das – abschaltbare - Paging durch eine Segmentierung überlagert. Diese **Segmentierung** läßt sich de facto dadurch **abschalten**, daß alle Segmente mit einer **Basis-Adresse** von **0** definiert werden  
→ **virtuelle Adresse** (genauer : deren Offset-Teil) == **lineare Adresse** ("Flat Memory Model")  
Diese Möglichkeit wird bei LINUX genutzt.

Folgende Segmente sind definiert :

- ◇ User-Code-Segment (Basis = 0, Größe = TASK\_SIZE (== PAGE\_OFFSET, default : 3 GBytes))
- ◇ User-Daten-Segment (Basis = 0, Größe = TASK\_SIZE (== PAGE\_OFFSET, default : 3 GBytes))
- ◇ Kernel-Code-Segment (Basis = 0, Größe = 4 GBytes)
- ◇ Kernel-Daten-Segment (Basis = 0, Größe = 4 GBytes)

Anmerkung : `#define TASK_SIZE (PAGE_OFFSET) in "include/asm/processor.h"`

- Jeder Prozess verfügt über eine eigene **LDT**.  
Die **Deskriptoren** für das **User-Code-Segment** und das **User-Daten-Segment** sind in den Einträgen mit der Slot-Nr. **1** und **2** enthalten  
→ User-Code-Segment-Selektor **0x0F**  
User-Daten-Segment-Selektor **0x17**
  - Die **Deskriptoren** für das **Kernel-Code-Segment** und das **Kernel-Daten-Segment** sind in der **GDT** enthalten (Slot Nr. **2** und **3**)  
Die GDT enthält auch **Deskriptoren** für das **User-Code-Segment** und das **User-Daten-Segment** der jeweils aktuellen Task (diese sind aber für alle User-Tasks gleich) (Slot Nr. **4** und **5**)  
→ Kernel-Code-Segment-Selektor **0x10**  
Kernel-Daten-Segment-Selektor **0x18**  
User-Code-Segment-Selektor **0x23**  
User-Daten-Segment-Selektor **0x2B**
- Für **jede Task** werden **2 weitere Einträge** in der GDT belegt : einen für ihr **TSS** und einen für ihre **LDT**.  
Die Ablage dieser Deskriptoren beginnt bei dem Eintrag in der Slot-Nr. **12** (TSS-Deskriptor für Task 0)  
→ TSS-Selektor für Task 0 **0x60**  
LDT-Selektor für Task 0 **0x68**

## Speicherbelegung eines LINUX-Prozesses

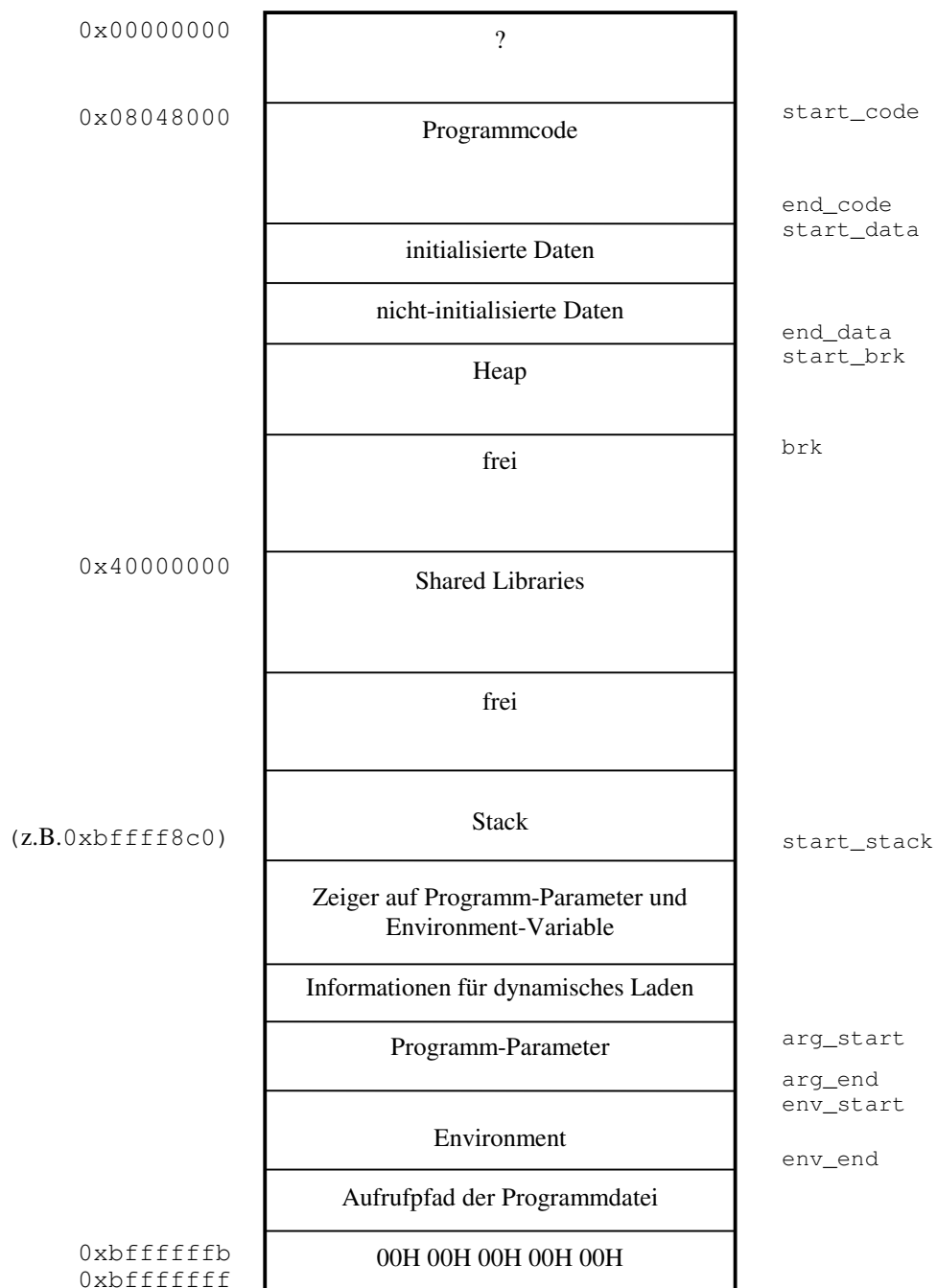
- Vorbemerkung**

Der von einem LINUX-Prozess verwendete Speicher ist **virtueller Speicher**.

Die genaue Aufteilung dieses virtuellen Adressraums hängt u.a. vom Format der vom Prozess ausgeführten Binärdatei ab.

Ein mittels `fork()` erzeugter Prozess übernimmt zunächst die Speicher-Struktur des Vaterprozesses.

- Aufteilung des virtuellen Adressraums bei einer Binärdatei vom ELF-Format**





## Virtuelle Speicherbereiche eines LINUX-Prozesses

### • Unterscheidung einzelner Speicherbereiche

Der von einem Prozess verwendete – virtuelle – Speicher ist nicht homogen, sondern zerfällt in **mehrere Bereiche**, die für unterschiedliche Zwecke unterschiedlich verwendet werden.

Die einzelnen Bereiche unterscheiden sich hinsichtlich

- des Typs der durch sie repräsentierten "Objekte" (Code, Daten, Shared Libraries, Heap, Mapped Files, Framebuffer usw)
- ihrer Verwendungs- und zulässigen Zugriffsart
- den Behandlungsroutinen für Zugriffsfehler und den Strategien für ihre Freigabe u. Auslagerung auf Sekundärspeicher

⇒ Abstraktion : **Virtueller Speicherbereich** (*Virtual Memory Area, VMA*)

#### Eigenschaften von VMAs :

- Jeder VMA repräsentiert einen zusammenhängenden Bereich des Adressraums eines Prozesses
- Zwischen VMAs können Lücken sein.
- VMAs können sich nicht überschneiden

### • Beschreibung von VMAs

Ein VMA wird durch den Structure-Typ **struct vm\_area\_struct** beschrieben.

Dieser Typ definiert u.a.

- Anfangs- und Endadresse des Speicherbereichs
- Schutzattribute für Speicherseiten aus diesem Bereich
- Informationen über den Typ des Speicherbereichs (einschließlich aktuelle Zugriffsrechte auf den Speicherbereich und Festlegungen, welche Schutzattribute gesetzt werden dürfen)
- gegebenenfalls einen Verweis auf die eingeblendete Datei
- Verweis auf eine Struktur, die Pointer auf die für den Speicherbereich anzuwendenden Behandlungsroutinen (z.B. für Zugriffsfehler, für das Auslagern u. Rückladen, für das Entfernen/Ausblenden) enthält

Für jeden vom Prozess verwendeten Speicherbereich wird eine Variable vom Typ **struct vm\_area\_struct** angelegt. (→ **VMA-Struktur**)

Die einzelnen VMA-Strukturen eines Prozesses bilden eine **vorwärts verkettete** nach Adressen aufwärts **sortierten Liste**. Zusätzlich sind die VMA-Strukturen in einem **RB-Baum** (*red black tree*) miteinander verknüpft.

Ein RB-Baum ist ein ausgeglichener (balanzierter) **sortierter Binärbaum**, der ein effektives Suchen nach einem bestimmten Element (bzw Einfügen eines Elements) ermöglicht.

Jedem Knoten ist eine Farbe (*red* oder *black*) so zugeordnet, dass eine (bei einem Einfügen oder Entfernen erforderliche) Neubalanzierung des Baumes erleichtert wird.

### • Anmerkungen zu den Behandlungsroutinen für VMAs

Für jeden Typ des von einem VMA repräsentierten "Objekts" (Code, Shared Memory, Mapped File, Heap usw.) existiert ein spezieller Satz von Behandlungsroutinen.

Pointer auf diese Behandlungsroutinen sind in einer für jeden "Objekt"-Typ angelegten statischen Structure-Variablen zusammengefaßt → Structure-Typ **struct vm\_operations\_struct** (→ VM-Operations-Struktur)

Für Operationen, die für einen bestimmten Typ nicht zur Verfügung stehen, enthält die entsprechende Structure-Variable den NULL-Pointer.

Die VM-Operations-Struktur stellt ein **abstrahiertes Interface** zum Aufruf der Speicher-Behandlungsroutinen zur Verfügung

### • Anwendung der VMAs bei einer Page Fault Exception

Der Page Fault Handler durchsucht die VMA-Strukturen des Prozesses nach dem Speicherbereich, in den die den Page Fault verursachende Adresse fällt. Falls es sich um eine für den Prozess gültige Adresse handelt, existiert ein derartiger Speicherbereich. Über die in seiner VMA-Struktur referierten VM-Operations-Struktur lässt sich die richtige Behandlungsroutine aufrufen.

**Prozessspezifische Speicherverwaltungsstrukturen (1)**

- **Structure-Typ `struct vm_area_struct`** (definiert in `<linux/mm.h>`)

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task. A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */

struct vm_area_struct {
    struct mm_struct * vm_mm;          /* The address space we belong to. */
    unsigned long vm_start;           /* Our start address within vm_mm. */
    unsigned long vm_end;             /* The first byte after our end address
                                       within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;           /* Access permissions of this VMA. */
    unsigned long vm_flags;          /* Flags, listed below. */

    struct rb_node vm_rb;

    /*
     * For areas with an address space and backing store,
     * linkage into the address_space->i_mmap prio tree, or
     * linkage to the list of like vmas hanging off its node, or
     * linkage of vma in the address_space->i_mmap_nonlinear list.
     */
    union {
        struct {
            struct list_head list;
            void *parent;           /* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
        } vm_set;

        struct raw_prio_tree_node prio_tree_node;
    } shared;

    /*
     * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
     * list, after a COW of one of the file pages. A MAP_SHARED vma
     * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
     * or brk vma (with NULL file) can only be in an anon_vma list.
     */
    struct list_head anon_vma_node;    /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;       /* Serialized by page_table_lock */

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;           /* Offset (within vm_file) in PAGE_SIZE
                                       units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;           /* File we map to (can be NULL). */
    void * vm_private_data;          /* was vm_pte (shared mem) */
    unsigned long vm_truncate_count; /* truncate_count or restart_addr */

#ifdef CONFIG_MMU
    atomic_t vm_usage;               /* refcount (VMAs shared if !MMU) */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
};
```

## Prozessspezifische Speicherverwaltungsstrukturen (2)

- **Structure-Typen `struct rb_node` und `rb_root`** (definiert in `<linux/rbtree.h>`)

Der Datentyp `struct rb_node` beschreibt einen Knoten eines RB-Baums

Der Datentyp `struct rb_root` enthält einen Pointer auf den Wurzelknoten eines RB-Baums

```
struct rb_node
{
    unsigned long  rb_parent_color;
#define   RB_RED    0
#define   RB_BLACK  1
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};

struct rb_root
{
    struct rb_node *rb_node;
};
```

- **Structure-Typ `struct vm_operations_struct`** (definiert in `<linux/mm.h>`)

Dieser Datentyp fasst die Pointer auf die für einen VMA geltenden Behandlungsroutinen zusammen (Operations-Tabelle)

```
/*
 * These are the virtual MM functions - opening of an area, closing and
 * unmapping it (needed to keep files on disk up-to-date etc), pointer
 * to the functions called when a no-page or a wp-page exception occurs.
 */

struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    struct page * (*nopage)(struct vm_area_struct * area, unsigned long address,
                           int *type);
    int (*populate)(struct vm_area_struct * area, unsigned long address,
                   unsigned long len, pgprot_t prot, unsigned long pgoff,
                   int nonblock);

    /* notification that a previously read-only page is about to become
     * writable, if an error is returned it will cause a SIGBUS */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct page *page);

#ifdef CONFIG_NUMA
    int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);
    struct mempolicy * (*get_policy)(struct vm_area_struct *vma, unsigned long addr);
    int (*migrate)(struct vm_area_struct *vma, const nodemask_t *from,
                  const nodemask_t *to, unsigned long flags);
#endif
};
```

### Bedeutung einiger Funktionen :

- ▷ **open ()** wird aufgerufen, wenn der angegebene VMA zu dem Speicherbereich eines Prozesses hinzugefügt wird (häufig nicht benötigt, dann == NULL)
- ▷ **close ()** wird aufgerufen, wenn der angegebene VMA aus dem Speicherbereich eines Prozesses entfernt wird (häufig nicht benötigt, dann == NULL)
- ▷ **nopage ()** wird vom Page-Fault-Handler aufgerufen, wenn auf eine Seite zugegriffen wird, die sich nicht im physikalischen Speicher befindet
- ▷ **populate ()** wird gegebenenfalls beim Einblenden von Dateien benötigt

### Prozessspezifische Speicherverwaltungsstrukturen (3)

- **Structure-Typ `struct mm_struct`** (definiert in `<linux/sched.h>`)

Zentrale Verwaltungsstruktur für den von einem Prozess verwendeten Speicher.

Sie enthält u.a. einen Pointer auf den Beginn der Liste der VMA-Strukturen des Prozesses sowie einen Pointer auf den Wurzelknoten des RB-Baums der VMA-Strukturen.

Zwei Prozesse, die dieselbe `mm_struct`-Struktur besitzen, arbeiten mit identischen Speicherbereichen (→ Threads !)

```
struct mm_struct {
    struct vm_area_struct * mmap;           /* list of VMAs */
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache;     /* last find_vma result */
    unsigned long (*get_unmapped_area) (struct file *filp, unsigned long addr,
   unsigned long len, unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
    unsigned long mmap_base;                /* base of mmap area */
    unsigned long task_size;                /* size of task vm space */
    unsigned long cached_hole_size;         /* if non-zero, the largest hole below free_area_cache */
    unsigned long free_area_cache;          /* first hole of size cached_hole_size or larger */
    pgd_t * pgd;
    atomic_t mm_users;                      /* How many users with user space? */
    atomic_t mm_count;                      /* How many references to "struct mm_struct" (users count as 1) */
    int map_count;                          /* number of VMAs */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;             /* Protects page tables and some counters */

    struct list_head mmlist;                /* List of maybe swapped mm's. These are globally strung together off
   * init_mm.mmlist, and are protected by mmlist_lock */

    /* Special counters, in some configurations protected by the
     * page_table_lock, in other configurations by being atomic.
     */
    mm_counter_t _file_rss;
    mm_counter_t _anon_rss;

    unsigned long hiwater_rss;              /* High-watermark of RSS usage */
    unsigned long hiwater_vm;              /* High-water virtual memory usage */

    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;

    unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */

    unsigned dumpable:2;
    cpumask_t cpu_vm_mask;

    /* Architecture-specific MM context */
    mm_context_t context;

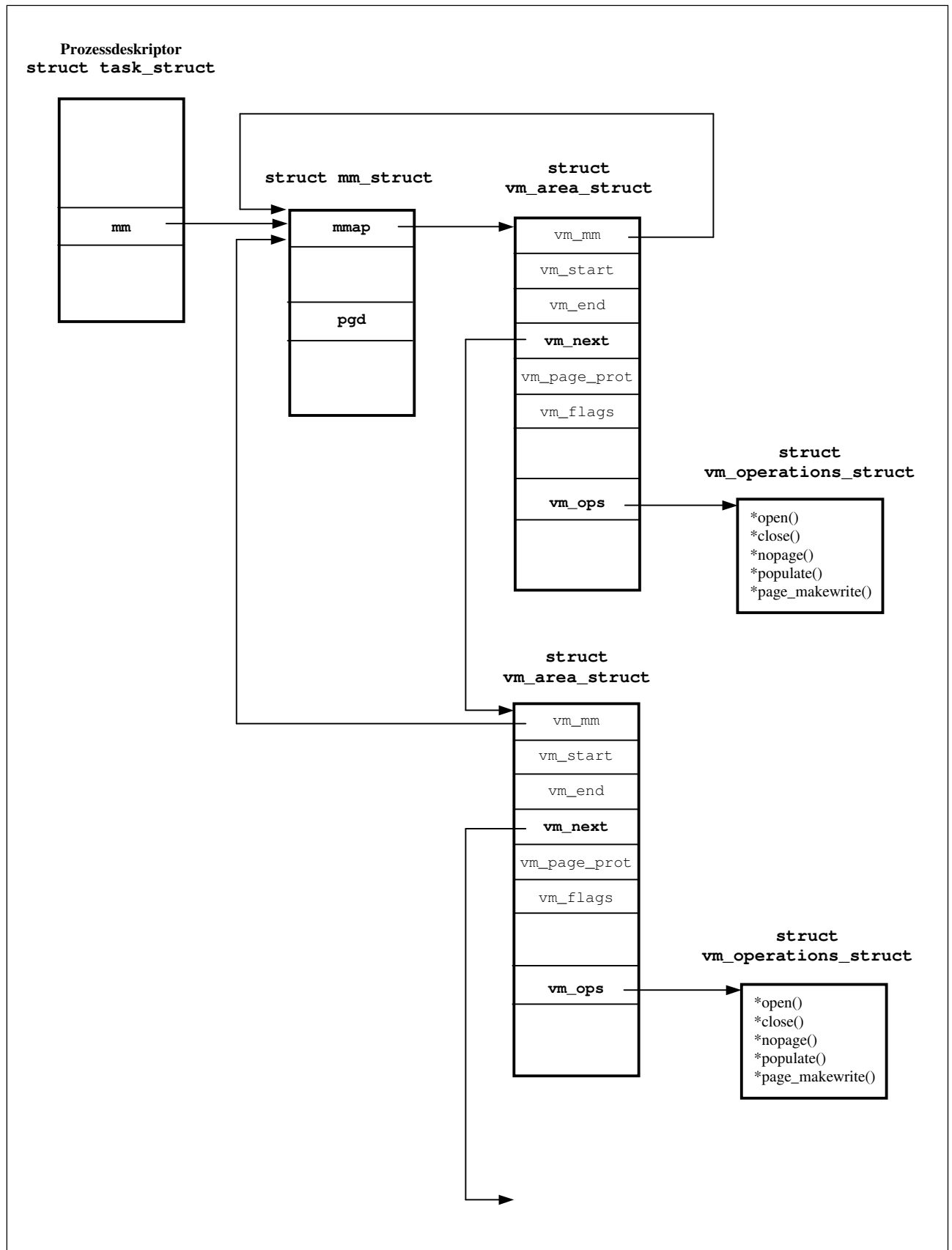
    /* Token based thrashing protection. */           /* → 2 Komponenten */
    /* coredumping support */                       /* → 3 Komponenten */
    /* aio bits */                                   /* → 2 Komponenten */
};
```

#### Bedeutung einiger Structure-Komponenten :

- ▷ **`mm_rb`** : enthält Pointer auf den Wurzelknoten des RB-Baums
- ▷ **`mmap_cache`** : Cache für zuletzt bearbeiteten VMA
- ▷ **`pgd`** : Adresse des Page Directories der Task

## Prozessspezifische Speicherverwaltungsstrukturen (4)

### • Überblick



## LINUX System Call **brk**

- **Funktionalität :** Veränderung (Vergrößerung bzw Verkleinerung) des **Heaps** durch Veränderung (Erhöhung bzw Erniedrigung) des **brk-Werts**.  
Der **brk-Wert** ist die erste Adresse nach dem Heap-Ende (= Endadresse des Heaps + 1)

- **Interface :**

```
int brk(void* new_brk);
```

- **Header-Datei :** **<unistd.h>**
- **Parameter :** *new\_brk* neuer brk\_Wert (neue Endadresse des Heaps+1)  
Der Wert muß größer als die Endadresse des Code-Segments (end\_code) sein und um wenigstens 16 kB unterhalb des Stacks liegen
- **Rückgabewert :** - 0 bei Erfolg  
- -1 im Fehlerfall (*new\_brk* > alter brk-Wert und Setzen auf *new\_brk* nicht möglich)  
*errno* wird auf den Wert *ENOMEM* gesetzt
- **Implementierung :** System Call Nr. 45  
→ *sys\_brk(...)* (in *mm/mmap.c*)  
Falls ein neuer VMA benötigt wird, wird dieser mittels *do\_mmap(...)* allokiert  
(s. System Call *mmap*)

- **Anmerkung :**

Die eigentliche Betriebssystemfunktion *sys\_brk()* kehrt immer mit dem jeweils aktuellen (alten bzw veränderten) brk-Wert als Funktionswert zurück. Die Wrapper-Funktion setzt eine globale Bibliotheksvariable (*\_\_curbrk*) auf diesen Wert und erzeugt den Rückgabewert -1, wenn dieser Wert < *new\_brk* ist, andernfalls den Wert 0. (z.B. kehrt *brk(NULL)* mit dem Wert 0 (Erfolg !) zurück)

- **Beispiel für Anwendung :**

Der System Call wird von den **ANSI-C-Bibliotheksfunktionen** zur **dynamischen Speicherallokation** (*malloc()*, *free()* usw.) verwendet.  
In Benutzerprogrammen wird er kaum eingesetzt.

- **Alternatives Interface zum Aufruf des System Calls**

Die C-Bibliothek (nicht ANSI-C) stellt eine weitere Funktion zum Aufruf des System Calls mit geändertem Interface zur Verfügung.  
Diese Funktion ruft *brk()* auf.

```
void* sbrk(ptrdiff_t increment);
```

- **Header-Datei :** **<unistd.h>**
- **Parameter :** *increment* Differenz des neuen brk-Werts zum alten brk-Wert
- **Rückgabewert :** - **alter brk-Wert**, wenn erfolgreich  
(der Aufruf endet auch erfolgreich mit *increment==0*  
→ Ermittlung des aktuellen brk-Wertes)  
- **(void\*) (-1)** im Fehlerfall, *errno* wird auf den Wert *ENOMEM* gesetzt

## LINUX System Call `mmap` (1)

- **Funktionalität :** Allokation eines **virtuellen Speicherbereichs** (VMA), in den gleichzeitig eine **Datei** oder ein **Gerät** eingeblendet werden kann.  
Wird keine Datei bzw kein Gerät eingeblendet, liegt **anonymes Einblenden** vor.  
Der neu allokierte VMA wird in die VMA-Liste (u. ggfls den VMA-RB-Baum) des Prozesses eingehängt.  
Aneinander angrenzende gleichartige VMAs werden zu einem einzigen VMA verschmolzen

- **Interface :**

```
void* mmap(void* start, size_t length, int prot,  
            int flags, int fd, off_t offset);
```

- **Header-Datei :**    `<unistd.h>`  
                      `<sys/mman.h>`
- **Parameter :** *start*    "vorgeschlagene" Startadresse des neuen VMAs (meist gleich `NULL` gesetzt)  
*length*    Größe des zu allozierenden Speicherbereichs in Bytes  
*prot*    Zugriffsrechte für den zu allozierenden Speicherbereich  
          Mögliche – gegebenenfalls bitweis oder-verknüpfte – Werte :  
          **PROT\_EXEC**        ausführender Zugriff zulässig  
          **PROT\_READ**        lesender Zugriff zulässig  
          **PROT\_WRITE**        schreibender Zugriff zulässig  
          **PROT\_NONE**        kein Zugriff zulässig  
*flags*    Einblend-Optionen  
          Mögliche – gegebenenfalls bitweis oder-verknüpfte – Werte (s. gesonderte Übersicht) :  
          **MAP\_FIXED**        angegebene Startadresse (*start*) muß gültig sein  
          **MAP\_SHARED**        Änderungen im Speicher auch in der eingeblendeten Datei  
          **MAP\_PRIVATE**        Änderungen im Speicher nicht in der eingeblendeten Datei  
          **MAP\_ANON (YMOUS)**    anonymes Einblenden  
          **MAP\_DENYWRITE**    "Normale" Schreibzugriffe zur eingeblendeten Datei sind  
                                  unzulässig  
          **MAP\_GROWSDOWN**    Speicherbereich soll zu niedrigeren Adressen erweiterbar sein  
          **MAP\_EXECUTABLE**    Speicherbereich soll ausführbaren Code aufnehmen  
          **MAP\_LOCKED**        Speicherbereich soll als nicht-auslagerbar in den  
                                  physikalischen Speicher eingeblendet werden  
*fd*        Filedeskriptor der einzublendenden Datei/Gerät,  
          wird bei anonymen Einblenden ignoriert.  
*offset*    Offset innerhalb der Datei, ab dem die Datei eingeblendet werden soll,  
          muß ein Vielfaches der Seitengröße sein.
- **Rückgabewert :** - **tatsächliche Startadresse** des allokierten Speicherbereichs, falls erfolgreich  
                      - **(void\*) (-1)** im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **90**  
                          → `old_mmap(...)` (in `arch/i386/kernel/sys_i386.c`)  
                          → `do_mmap(...)` (in `mm/mmap.c`)

## LINUX System Call `mmap` (2)

- **Anmerkungen zu den Zugriffsrechten** (Parameter `prot`)

- ✦ Eine Datei kann nur mit den Zugriffsrechten eingeblendet werden, für die sie geöffnet worden ist.
- ✦ Die tatsächlich realisierbaren Zugriffsrechte hängen von der Hardware-Plattform ab : Für die x86-Architektur gilt beispielsweise, daß bei `PROT_WRITE` auch `PROT_READ` und `PROT_EXEC`, bei `PROT_EXEC` auch `PROT_READ` und bei `PROT_READ` auch `PROT_EXEC` automatisch mitgesetzt werden

- **Einblend-Optionen** (Parameter `flags`)

Mögliche – gegebenenfalls bitweis oder-verknüpfte - Werte :

- MAP\_FIXED** Der zu allozierende Speicherbereich muß an der angegebenen Startadresse `start` beginnen. In diesem Fall muß `start` auf den Beginn einer Seite zeigen. Ist eine Allokation ab dieser Adresse nicht möglich, endet der System Call fehlerhaft.
- MAP\_SHARED** Änderungen des Inhalts des allokierten Speicherbereichs werden auch in der eingeblendeten Datei vorgenommen und damit mit anderen Prozessen, die die gleiche Datei geöffnet haben, geteilt.
- MAP\_PRIVATE** Änderungen des Inhalts des allokierten Speicherbereichs werden nicht in der eingeblendeten Datei vorgenommen und damit nicht mit anderen Prozessen geteilt. Es muß genau eines der beiden Flags **MAP\_SHARED** oder **MAP\_PRIVATE** angegeben werden, auch wenn `MAP_ANON` gesetzt ist.
- MAP\_ANON (YMOUS)** Es wird keine Datei oder Gerät in den zu allozierenden Speicherbereich eingeblendet  
→ **anonymes Einblenden**  
In diesem Fall wird der Parameter `fd` ignoriert.
- MAP\_DENYWRITE** Zu der eingeblendeten Datei sind "normale" Schreibzugriffe (mittels `write()`) unzulässig.
- MAP\_GROWSDOWN** Der zu allozierende Speicherbereich soll zu niedrigeren Adressen erweiterbar sein : Bei einem Zugriff zu einer bisher nicht für den Prozess allozierten Adresse wird ein weiterer Speicherbereich unmittelbar vor dem Speicherbereich alloziert und anonym eingeblendet, so daß der Zugriff möglich wird und nicht zur Erzeugung des Signals `SIGSEGV` führt. (wird u.a. vom Kernel zur Vergrößerung des Stacks verwendet)
- MAP\_EXECUTABLE** Der zu allozierende Speicherbereich soll ausführbaren Code aufnehmen
- MAP\_LOCKED** Der zu allozierende Speicherbereich soll nicht-auslagerbar in den physikalischen Speicher eingeblendet werden (erfordert root-Rechte).

- **Beispiele für Anwendung :**

- ✦ Laden einer ausführbaren Programm-Datei in den Arbeitsspeicher durch den Kernel
- ✦ Einblenden von dynamisch ladbaren Bibliotheken (DLL's, Shared Libraries) in den Adressraum eines Prozesses durch den Kernel
- ✦ Einblenden von Dateien durch einen Prozess.
  - schnellerer Dateizugriff (Umgehung eines Kernel-Datei-Buffers, Zugriff mittels Pointer statt System Calls)
  - Dateiinhalt steht auch nach dem Schließen der eingeblendeten Datei zur Verfügung
  - Prozess kann Daten im Speicher ablegen, die über das Prozessende in der eingeblendeten Datei erhalten bleiben (→ Persistenz)
- ✦ Allokation von für besondere Zwecke benötigten Speicher durch einen Prozess.  
(Anonymes Einblenden oder Einblenden des – nur NUL-Bytes enthaltenden - Gerätes `"/dev/zero"`)  
z.B. durch einen Just-in-time-Compiler, der in den als ausführbar gekennzeichneten Speicherbereich Maschinenbefehle ablegt, die anschließend ausgeführt werden können.



## LINUX System Call `munmap`

- **Funktionalität :** **Freigabe eines virtuellen Speicherbereichs (VMA)**, verbunden mit dem Ausblenden einer ggfls. eingeblendet gewesenen Datei.  
Es lassen sich auch Teile eines VMAs freigeben.  
Der freizugebende Bereich kann sich auch über mehrere VMAs erstrecken, die allerdings unmittelbar aneinander anschließen müssen.

- **Interface :**

```
int munmap(void* start, size_t length);
```

- **Header-Datei :** `<sys/mman.h>`

- **Parameter :**
  - `start` Anfangsadresse des freizugebenden Speicherbereichs  
Die Adresse muß page-aligned sein.  
Die Angabe einer Adresse, die in keinem VMA liegt (also zu keinem allokierten Speicherbereich gehört) ist zulässig, der System Call endet fehlerfrei ohne irgendeine Wirkung.
  - `length` Größe des freizugebenden Speicherbereichs in Bytes  
(wird gegebenenfalls auf nächstes Vielfache einer Seitengröße aufgerundet)

- **Rückgabewert :**
  - `0` bei Erfolg
  - `-1` im Fehlerfall, `errno` wird auf den Wert `EINVAL` gesetzt

- **Implementierung :** System Call Nr. **91**
  - `sys_munmap(...)` (in `mm/mmap.c`)
  - `do_munmap(...)` (in `mm/mmap.c`)

- **Anmerkungen :**
  - 1.) Ein zukünftiger Zugriff zu einer Adresse des freigegebenen Bereichs führt zu einer Page Fault Exception und in Folge zum Senden des Signals `SIGSEGV` an den Prozess.
  - 2.) Alle VMAs eines Prozesses werden automatisch freigegeben, wenn der Prozess endet oder mittels eines `exec()` System Calls ein neues Programm ausführt

### Beispiel zum Einblenden einer Datei in den Arbeitsspeicher

```
// C-Quell-Datei mapbsp_m.c

// Beispiel zu den System Calls map und unmap

// Sedezimale Darstellung und Änderung von Bytes an beliebiger Dateiposition
// Zugriff zum Dateiinhalte nach Einblenden der Datei in den Arbeitsspeicher

// Der Zugriffspfad zu der zu bearbeitenden Datei ist als
// Kommandozeilenparameter zu übergeben

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/mman.h>

extern void errorExit(char* msg);

void changeBytesInFile(int fd);

int main(int argc, char** argv)
{ int fd;
  if (argc<2)
    printf("\nAufruf : mapbsp <dateipfad>\n\n");
  else
  {
    if ((fd=open(argv[1], O_RDWR))===-1)
      errorExit("open");
    changeBytesInFile(fd);
    if (close(fd)===-1)
      errorExit("close");
  }
  return 0;
}

void changeBytesInFile(int fd)
{ off_t dlen;
  char* fAddr;
  int pos;
  unsigned neu;
  if ((dlen=lseek(fd, 0, SEEK_END))===(off_t)(-1))
    errorExit("lseek");
  printf("\nDateigröße : %ld Bytes\n", (long)dlen);
  fAddr=mmap(NULL, (size_t)dlen, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  if (fAddr==(void*)(-1))
    errorExit("mmap");
  while (((printf("\nPosition ? "), scanf("%d", &pos))!=EOF))
  {
    getchar(); // Entfernen NL-Character (Eingabe RET) aus Eingabepuffer
    if (pos<0 || pos>=dlen)
      printf("Position ausserhalb der Datei !\n");
    else
    {
      printf("alt : %02x neu ? ", *(fAddr+pos)&0xff);
      if ((neu=getchar())!='\n' && neu!=EOF)
      { ungetc(neu, stdin);
        if (scanf("%x", &neu)!=EOF)
          *(fAddr+pos)=neu;
      }
    }
  }
  putchar('\n');
  if (munmap(fAddr, dlen)===-1)
    errorExit("munmap");
}
```

## Datenstrukturen zur Verwaltung des physikalischen Speichers (1)

- **Verwaltungstabelle für physikalische Seiten `mem_map`** (definiert in `include/linux/mm.h`)

Tabelle (Array) zur Aufnahme von für die Verwaltung benötigten Beschreibungsinformationen der physikalischen Seiten im System :

```
extern mem_map_t *mem_map;
```

Jeder physikalischen Seite ist genau ein Eintrag (Array-Element) zugeordnet.

Der Index des Eintrags entspricht der physikalischen Seitennummer (*physical page frame, physical frame number, PFN*)

- **Structure-Typ `mem_map_t (struct page)`** (definiert in `include/linux/mm.h`)

Dieser Typ dient zur Aufnahme von Beschreibungsinformationen für eine physikalische Seite

```
typedef struct page {
    /* these must be first (free area handling) */
    struct page *next;
    struct page *prev;
    struct inode *inode;
    unsigned long offset;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags; /* atomic flags,
                        some possibly updated asynchronously */
    struct wait_queue *wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
} mem_map_t;
```

### Bedeutung der Structure-Komponenten :

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------------------------------------------------------------|-----------------|--------------------------------------------|----------------------|----------------------------------|-----------------|-------------------------------------|--------------------|---------------------------------------------------------------|----------------------|---------------------------------------------------------|----------------------|--|-----------------------------|--|---------------|------------------------------------------------|----------------|--|----------------------|-----------------------------|----------------|-----------------------------------------------------|--------------------|-----------------------------------------|
| <b>next</b> und <b>prev</b>            | : Pointer zur Verwaltung zusammengehöriger <code>mem_map</code> -Komponenten in doppelt verketteten Listen<br>(freie Seiten in <code>free_area[]</code> oder alle belegten Seiten mit Inhalten derselben Datei)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>inode</b>                           | : Referenz der Datei, zu der der Inhalt der Seite gehört,<br>bei Seiten des Swap-Caches (enthaltene Info ist aus einem Auslagerungsbereich) wird der spezielle reservierte Inode <code>swapper_inode</code> eingetragen                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>offset</b>                          | : Offset des Seiteninhalts in der Datei,<br>bei Seiten des Swap-Caches : Auslagerungsort (Format wie beim Seitentabelleneintrag ausgelagerter Seiten)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>next_hash</b> und <b>pprev_hash</b> | : Pointer zur Referenzierung der Seite in einer Hash-Liste<br>(z.B. für Swap-Cache und Buffer-Cache)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>count</b>                           | : Referenzzähler (Anzahl der Referenzen auf diese Seite in Seitentabellen)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>flags</b>                           | : Zugriffs- und Verwendungsflags der Seite : <table style="margin-left: 20px;"> <tr> <td><b>PG_locked</b></td><td>Seite ist verriegelt (gesperrt) (während Platten-I/O gesetzt)</td></tr> <tr> <td><b>PG_error</b></td><td>I/O-Fehler ist für diese Seite aufgetreten</td></tr> <tr> <td><b>PG_referenced</b></td><td>zur Seite ist zugegriffen worden</td></tr> <tr> <td><b>PG_dirty</b></td><td>in die Seite ist geschrieben worden</td></tr> <tr> <td><b>PG_uptodate</b></td><td>der Seiteninhalt ist gültig (nach fehlerfreiem Lesen gesetzt)</td></tr> <tr> <td><b>PG_free_after</b></td><td>Seite soll nach Schreiben auf Platte freigegeben werden</td></tr> <tr> <td><b>PG_decr_after</b></td><td></td></tr> <tr> <td><b>PG_swap_unlock_after</b></td><td></td></tr> <tr> <td><b>PG_DMA</b></td><td>Seite liegt in einem DMA-fähigen Adressbereich</td></tr> <tr> <td><b>PG_Slab</b></td><td></td></tr> <tr> <td><b>PG_swap_cache</b></td><td>Seite gehört zum Swap-Cache</td></tr> <tr> <td><b>PG_skip</b></td><td>Seite kennzeichnet Lücke im physikalischen Speicher</td></tr> <tr> <td><b>PG_reserved</b></td><td>zur Seite darf nicht zugegriffen werden</td></tr> </table> | <b>PG_locked</b> | Seite ist verriegelt (gesperrt) (während Platten-I/O gesetzt) | <b>PG_error</b> | I/O-Fehler ist für diese Seite aufgetreten | <b>PG_referenced</b> | zur Seite ist zugegriffen worden | <b>PG_dirty</b> | in die Seite ist geschrieben worden | <b>PG_uptodate</b> | der Seiteninhalt ist gültig (nach fehlerfreiem Lesen gesetzt) | <b>PG_free_after</b> | Seite soll nach Schreiben auf Platte freigegeben werden | <b>PG_decr_after</b> |  | <b>PG_swap_unlock_after</b> |  | <b>PG_DMA</b> | Seite liegt in einem DMA-fähigen Adressbereich | <b>PG_Slab</b> |  | <b>PG_swap_cache</b> | Seite gehört zum Swap-Cache | <b>PG_skip</b> | Seite kennzeichnet Lücke im physikalischen Speicher | <b>PG_reserved</b> | zur Seite darf nicht zugegriffen werden |
| <b>PG_locked</b>                       | Seite ist verriegelt (gesperrt) (während Platten-I/O gesetzt)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_error</b>                        | I/O-Fehler ist für diese Seite aufgetreten                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_referenced</b>                   | zur Seite ist zugegriffen worden                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_dirty</b>                        | in die Seite ist geschrieben worden                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_uptodate</b>                     | der Seiteninhalt ist gültig (nach fehlerfreiem Lesen gesetzt)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_free_after</b>                   | Seite soll nach Schreiben auf Platte freigegeben werden                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_decr_after</b>                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_swap_unlock_after</b>            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_DMA</b>                          | Seite liegt in einem DMA-fähigen Adressbereich                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_Slab</b>                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_swap_cache</b>                   | Seite gehört zum Swap-Cache                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_skip</b>                         | Seite kennzeichnet Lücke im physikalischen Speicher                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>PG_reserved</b>                     | zur Seite darf nicht zugegriffen werden                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>wait</b>                            | : Pointer auf Warteschlange der Prozesse, die auf eine Entsperrung der Seite warten<br>(eine Seite ist gesperrt, wenn das <code>flags</code> -Bit <code>PG_locked</code> gesetzt ist)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |
| <b>buffers</b>                         | : Pointer auf die Verwaltungsstruktur eines Blockpuffers ( <i>buffer head</i> ), wenn die Seite Bestandteil eines Blockpuffers ist                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                  |                                                               |                 |                                            |                      |                                  |                 |                                     |                    |                                                               |                      |                                                         |                      |  |                             |  |               |                                                |                |  |                      |                             |                |                                                     |                    |                                         |

## Datenstrukturen zur Verwaltung des physikalischen Speichers (2)

- **Verwaltungstabelle für freie physikalische Seiten `free_area`** (definiert in `mm/page_alloc.c`)

```
#define NR_MEM_LISTS 10
static struct free_area_struct free_area[NR_MEM_LISTS];
```

Die einzelnen Tabelleneinträge enthalten jeweils Pointer auf eine doppelt verkettete Ringliste freier Speicherblöcke einer bestimmten – durch den Eintragsindex festgelegten - Ordnung :

Eintrag `i` referiert Speicherblöcke von jeweils  $2^{**i}$  aufeinanderfolgenden freien Seiten (Ordnung `i`)

(d.h. Eintrag `0` referiert einzelne freie Seiten,

Eintrag `1` referiert Speicherblöcke von jeweils zwei aufeinanderfolgenden freien Seiten, usw)

- **Structure-Typ `struct free_area_struct`** (definiert in `mm/page_alloc.c`)

```
/* The start of this MUST match the start of "struct page" */
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int * map;
};
```

### Bedeutung der Structure-Komponenten :

|             |                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>next</b> | : Pointer auf erste Seite des ersten freien Speicherblocks der jeweiligen Ordnung                                                                                                                                                                                                                                                                                                                 |
| <b>prev</b> | : Pointer auf erste Seite des letzten freien Speicherbereichs der jeweiligen Ordnung<br>neu hinzu kommende freie Speicherbereiche werden an den Anfang der Liste gestellt                                                                                                                                                                                                                         |
| <b>map</b>  | : Pointer auf eine Belegungs-Bitmap<br>In dieser Bitmap ist jeweils ein Bit für zwei aufeinanderfolgende Speicherblöcke der jeweiligen Ordnung zuständig ( <b>Buddy</b> -System).<br>Das Bit ist gesetzt, wenn einer der beiden Speicherblöcke frei und der andere – wenn auch nur teilweise – reserviert ist.<br>Wenn beide Speicherblöcke frei oder beide belegt sind, ist das Bit rückgesetzt. |

## Allokation und Freigabe physikalischer Seiten

- **Allgemeines**

Linux ermöglicht eine **blockweise** Allokation und Freigabe physikalischer Seiten.

Hierfür setzt es den **Buddy-Algorithmus** ein.

Dieser stellt sicher, daß freie Seiten immer zu größtmöglichen Blöcken zusammengefaßt sind.

D.h. es existieren niemals zwei aufeinanderfolgende freie Speicherblöcke, die zu einem größeren Speicherblock zusammengefaßt werden könnten.

Die Anzahl der Seiten eines Speicherblocks muß immer eine Zweierpotenz sein.

Ein Speicherblock hat die **Ordnung**  $i$ , wenn er  $2^{**i}$  Seiten umfaßt.

Seine Anfangsadresse ist immer ein Vielfaches seiner Größe in Bytes.

- **Allokation eines physikalischer Speicherblocks**

Zur Allokation eines physikalischen **Speicherblocks** ruft der Kernel die (in `mm/page_alloc.c` definierte) Funktion

```
unsigned long __get_free_pages(int gfp_mask, unsigned long order)
```

auf.

Der Parameter `gfp_mask` steuert die Arbeitsweise der Funktion.

Der Parameter `order` legt die Ordnung und damit die Größe des zu allozierenden Speicherblocks fest.

Falls die Anzahl freier Seiten kleiner als ein vorgegebener Schwellenwert ist, wird versucht – unabhängig vom Kernel-Thread `kswapd` - allokierte Seiten freizubekommen. Hierfür wird die Funktion **`try_to_free_pages()`** (def. in `mm/vmscan.c`) aufgerufen, die ihrerseits die auch von `kswapd` verwendete Funktion **`do_try_to_free_pages()`** (def. in `mm/vmscan.c`) aufruft. Konnten keine Seiten freigegeben werden, kehrt `__get_free_pages()` mit dem Wert `0` erfolglos zurück.

Steht – gegebenenfalls nach erfolgreicher Freigabe einiger Seiten - eine ausreichende Anzahl freier Seiten zur Verfügung, sucht `__get_free_pages()` durch Überprüfung von `free_area[order]` nach einem freien Speicherblock der angeforderten Ordnung.

Existiert ein derartiger Block, wird er aus der Freiliste seiner Ordnung ausgetragen und seine Anfangsadresse als Funktionswert zurückgegeben.

Wird dagegen kein freier Block der angeforderten Ordnung gefunden, so wird nach einem freien Speicherblock der jeweils nächsthöheren Ordnung gesucht, solange bis ein derartiger Block gefunden wird oder die gesamte `free_area[]`-Tabelle erfolglos durchsucht worden ist.

Wird ein Block höherer Ordnung gefunden, wird er **geteilt** und aus der Freiliste seiner Ordnung ausgetragen. Seine Anfangsadresse wird als Funktionswert zurückgegeben. Der über die angeforderte Größe hinausgehende Teilblock wird – gegebenenfalls weiter geteilt – in die entsprechenden Liste(n) freier Speicherblöcke (`free_area[]`) eingetragen.

Wird auch kein freier Speicherblock höherer Ordnung gefunden, kehrt die Funktion mit dem Wert `0` erfolglos zurück.

- **Freigabe eines physikalischen Speicherblocks**

Zur Freigabe eines physikalischen Speicherblocks ruft der Kernel die - in `mm/page_alloc.c` definierten – Funktionen

```
void free_pages(unsigned long addr, unsigned long order)
```

bzw (Freigabe einer Seite)

```
void __free_page(struct page* page)
```

auf.

Beide Funktionen rufen die – ebenfalls in `mm/page_alloc.c` definierte – Funktion

```
static inline void free_pages_ok(unsigned long map_nr, unsigned long order)
```

auf

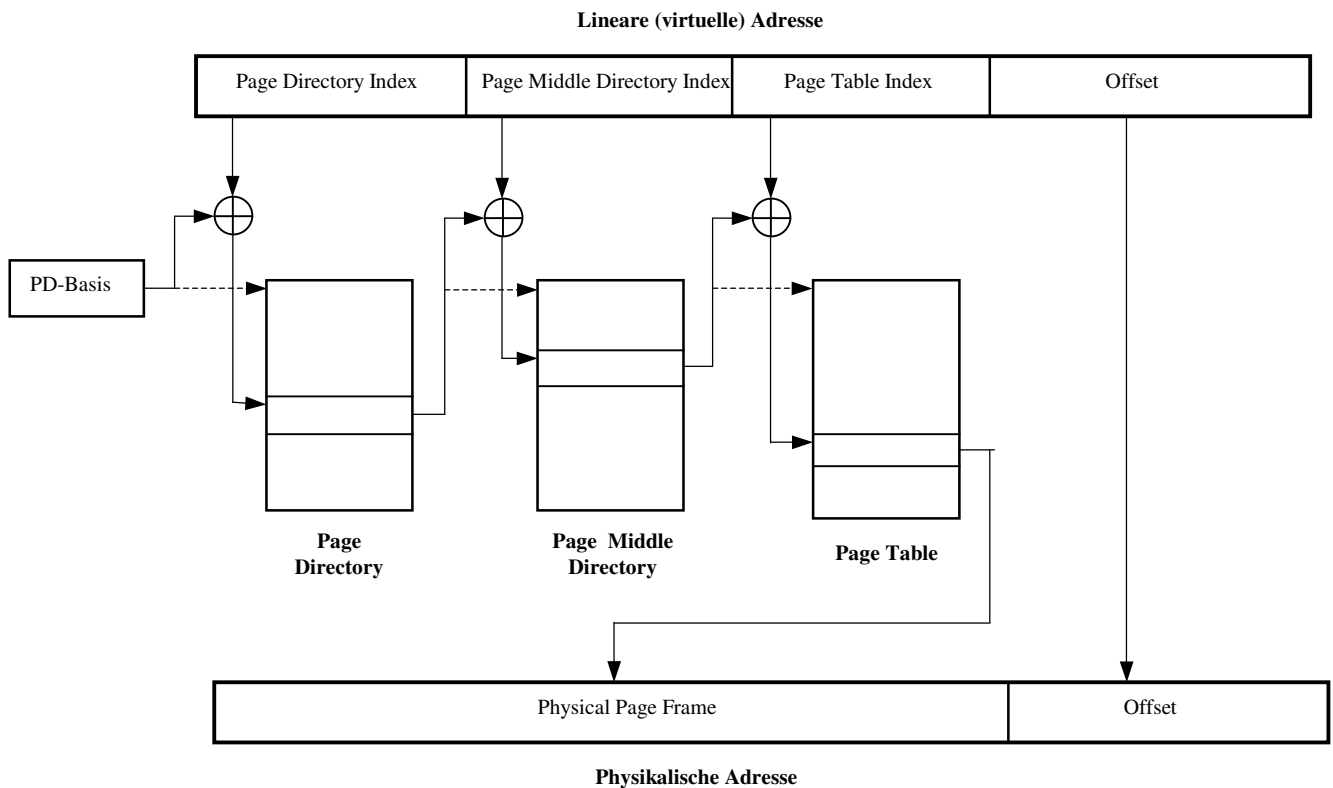
Diese Funktion versucht, den freizugebenden Block mit seinen Nachbar- (*buddy*-)Blöcken zu einem möglichst großen freien Block zusammenzufassen.

Hierzu wird überprüft, ob der jeweilige Nachbarblock ebenfalls frei ist. Wenn ja, wird er aus der Freiliste seiner Ordnung ausgetragen und mit dem freizugebenden Block zu einem Block doppelter Größe zusammengefaßt. Nach jeder derartigen Zusammenfassung wird analog versucht, wiederum einen freien Block doppelter Größe zu bilden.

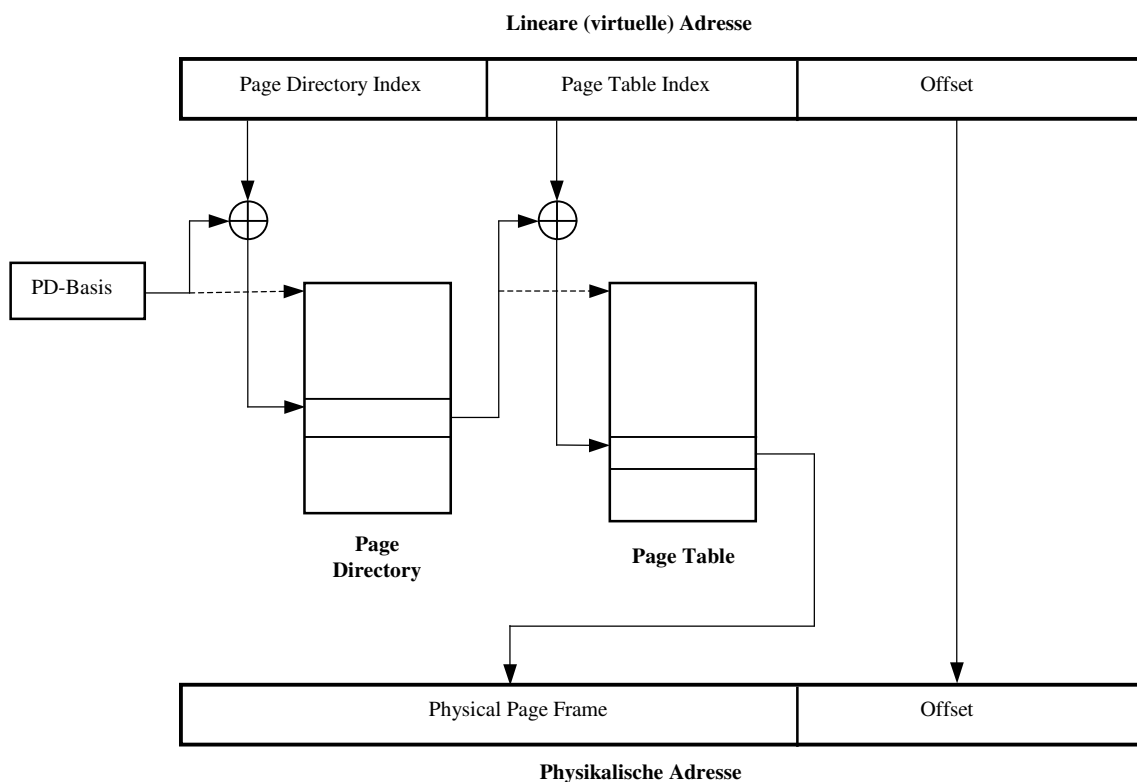
Auf diese Art und Weise entsteht immer ein größtmöglicher freier Speicherblock. Dieser wird dann in die Freiliste seiner Ordnung eingetragen.

## Adressumsetzung beim Paging in LINUX

- Generisches (architekturunabhängiges) Model



- Adressumsetzung durch die Paging Unit der x86-Architektur



## Seitentableneinträge unter LINUX

### • Allgemeines

Eine – logische und physikalische – Seite beginnt immer an einer durch die Seitengröße teilbaren Adresse, d.h. die **niederwertigen Stellen** einer **Seitenbasisadresse** sind **0** ( $\rightarrow$  *page-aligned*).

Die Seitentableneinträge (*page table entries*) enthalten daher nur die **höherwertigen relevanten Bits** der physikalischen Seitenbasisadresse ( $\rightarrow$  *physical page frame*, physikalische Seiten-Nummer).

Die **niederwertigen** – in der Adresse mit 0 besetzten – Bits werden für die Ablage von **Zugriffskontrollinformationen** (*access control information*, Schutzbits, *protection bits*) verwendet.

Die Anzahl dieser Bits und die Bedeutung der tatsächlich verwendeten Bits sind ebenso wie die Größe eines Seitentableneintrags abhängig von der Hardware-Plattform.

### • Zugriffskontrollinformations-Flags

Linux verwendet die folgenden Flags zur Festlegung und Kennzeichnung von Zugriffskontrollinformationen.

Die Lage dieser Flags ist hardware-abhängig und in der architekturenspezifischen Header-Datei *pgtable.h* definiert ( $\rightarrow$  z.B. *include/asm-i386/pgtable.h*)

#### ◇ **\_PAGE\_PRESENT (P-Bit)**

gesetzt : Seite befindet sich im physikalischen Speicher (der Eintrag enthält physikalische Seiten-Nummer)  
rückgesetzt : Seite ist ausgelagert (der Rest des Eintrags enthält Information über den Auslagerungsort)

#### ◇ **\_PAGE\_RW (RW-Bit)**

gesetzt : Seite ist les- und schreibbar  
rückgesetzt : Seite ist nur lesbar (*read only*)

#### ◇ **\_PAGE\_USER (US-Bit)**

gesetzt : Benutzerbereichs-Seite  
rückgesetzt : Systembereichs-Seite

#### ◇ **\_PAGE\_ACCESSED (A-Bit)**

gesetzt : zur Seite ist kürzlich zugegriffen worden  
rückgesetzt : kein kürzlicher Zugriff zur Seite, sie kann für das Auslagern bevorzugt werden

#### ◇ **\_PAGE\_DIRTY (D-Bit)**

gesetzt : in die Seite ist geschrieben worden, sie muß beim Seitenwechsel gesichert (ausgelagert) werden  
rückgesetzt : die Seite ist nicht verändert worden, sie kann beim Seitenwechsel verworfen werden

#### ◇ **\_PAGE\_WT (PWT-Bit)**

gesetzt : die Cache-Strategie für die Seite ist *writethrough*  
rückgesetzt : die Cache-Strategie für die Seite ist *writeback*

#### ◇ **\_PAGE\_PCD (PCD-Bit)**

gesetzt : Seite wird nicht gecacht (*page caching disabled*) (sinnvoll z.B. für *memory mapped I/O*)

### • Seitentableneintrag bei der x386-Architektur

|    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|----|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|----|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Eintrag für ausgelagerte Seite :

|                                     |    |    |    |   |   |   |                     |   |   |   |   |   |   |
|-------------------------------------|----|----|----|---|---|---|---------------------|---|---|---|---|---|---|
| 31                                  | 12 | 11 | 10 | 9 | 8 | 7 | 6                   | 5 | 4 | 3 | 2 | 1 | 0 |
| Seitennummer im Auslagerungsbereich |    |    |    |   |   | 0 | Auslagerungsbereich |   |   |   |   |   | 0 |

## Allgemeines zum Paging unter LINUX

### • Grundprinzip der Allokation physikalischer Seiten

Die Allokation physikalischer Seiten (d.h. ihre Zuordnung zu virtuellen Seiten durch entsprechende Einträge in den Seitentabellen eines Prozesses) findet erst dann statt, wenn es unumgänglich erforderlich ist, d.h. so spät wie möglich. Das bedeutet, daß die Allokation eines virtuellen Speicherbereichs (VMAs) i.a. noch keinen physikalischen Speicher bereitstellt, sondern lediglich den Speicherbereich für den Prozess als gültigen Adreßbereich reserviert.

Die Bereitstellung physikalischen Speichers erfolgt in der Regel – seitenweise - erst dann, wenn zu einer Adresse des Speicherbereichs zugegriffen wird.

⇒ **Demand Paging**

### • Demand-Paged Executables

Durch den **exec ()** –System-Call wird die Datei, die das auszuführende Programm enthält, lediglich – mittels Allokation eines entsprechenden VMAs - in den virtuellen Speicherbereich des Prozesses eingeblendet und nur die Seite, die den Programmstart enthält, wird tatsächlich in den physikalischen Speicher geladen.

Das Laden der weiteren Programmteile erfolgt immer erst dann, wenn zu ihnen zugegriffen wird.

### • Copy-On-Write

Eine – durch Kopieren zu erzeugende – neue physikalische Seite wird erst dann bereitgestellt, wenn versucht wird, in die ursprüngliche Seite zu schreiben. Hierfür wird die ursprüngliche Seite als **schreibgeschützt** gekennzeichnet, während zu dem entsprechenden VMA **Schreibzugriffe zulässig** sind.

**Beispiele für Anwendungen :**

- ◇ Bei der Erzeugung eines neuen Prozesses mittels des **fork ()** –System-Calls werden die VMAs und Seitentabellen des ursprünglichen Prozesses kopiert  
→ **Beide Prozesse** arbeiten mit **demselden** virtuellen und **physikalischen Speicher**.  
Die physikalischen Seiten werden als schreibgeschützt gekennzeichnet.  
Erst wenn einer der beiden Prozesse zu einer Seite **schreibend zugreift**, wird die entsprechende **physikalische Seite kopiert**. Ursprüngliche und kopierte Seite werden anschließend als schreibbar markiert. Bei beiden Prozessen ist jetzt die gleiche virtuelle Seite auf **verschiedene physikalische Seiten** abgebildet.
- ◇ Bei der Allokation (anonymes Einblenden) eines beschreibbaren virtuellen Speicherbereiches werden sämtliche Seiten dieses Speicherbereichs auf eine einzige "leere" schreibgeschützte physikalische Seite abgebildet.  
Erst bei einem Schreibzugriff zu einer Seite wird für diese eine entsprechende Kopie der physikalischen Seite angelegt, die dann als beschreibbar gekennzeichnet wird

### • Seitenfehler (Page Fault)

Wird eine – virtuelle – Speicheradresse referiert,

- für die kein Eintrag in einer Seitentabelle enthalten ist
- oder deren Seite in ihrem Seitentableneintrag als ausgelagert gekennzeichnet ist
- oder für die die festgelegten Schutzrechte verletzt werden (Schreibzugriff zu schreibgeschützter Seite),

tritt ein **Seitenfehler** auf.

→ Der Prozessor erzeugt – unter Bereitstellung von Informationen über die Ursache - eine **Page Fault Exception**.

CPUs der x386-Architektur schreiben die den Fehler verursachende – virtuelle - Adresse in das Register CR2 und legen einen Fehlercode auf den Stack.

Die diese Exception bearbeitende **Interrupt Service Routine** (primär realisiert durch die Funktion **do\_page\_fault ()** im Modul `arch/i386/mm/fault.c`) überprüft zunächst, ob die Adresse innerhalb eines für den Prozess allokierten VMAs liegt. Falls das nicht der Fall ist, wird an den Prozeß das Signal **SIGSEGV** geschickt.

Handelt es sich jedoch um eine **gültige virtuelle Adresse** des Prozesses, wird die Funktion **handle\_mm\_fault ()** (def. in `mm/memory.c`) aufgerufen. Diese Funktion **versucht** – durch Aufruf weiterer Funktionen - die **Fehlerursache** zu **beseitigen**, d.h. eine physikalische Seite zu allokiieren, eine Seite aus dem Hintergrundspeicher (Datei oder Auslagerungsbereich) nachzuladen oder für die Seite eine beschreibbare Kopie anzulegen (Copy-On-Write !).

Falls vorhanden, werden hierfür die über die Komponente `vm_ops` der den VMA beschreibenden `vm_area_struct`-Variablen bereitgestellten **VMA-spezifischen Behandlungsroutinen** eingesetzt, andernfalls Default-Behandlungsroutinen.



## Auslagerungsbereiche unter LINUX

### • Freigabe physikalischer Seiten

Um für Neu-Allokationen Seiten verfügbar zu haben, ist es i.a. notwendig, allokierte Seiten wieder freizugeben.

Hierfür ist primär der **Kernel-Thread kswapd** zuständig. Er wird einmal pro Sekunde aktiviert

Falls die Anzahl freier physikalischer Seiten kleiner als ein vorgegebener Schwellenwert ist, versucht **kswapd** Seiten freizugeben (→ Aufruf der Funktion `do_try_to_free_pages()`, def. in `mm/vmscan.c`).

Kandidaten für die Freigabe sind "alte" Seiten. Das sind Seiten, zu denen in letzter Zeit nicht zugegriffen worden ist (→ LRU-(*Least Recently Used*) Algorithmus)

**kswapd** durchsucht periodisch die VMAs sämtlicher swapbarer Tasks nach freigebbaren Seiten. Dabei werden die Tasks mit der größeren Anzahl physikalischer Seiten im Arbeitsspeicher bevorzugt.

#### Regeln für die Freigabe von Seiten :

- Seiten des Systembereichs können nicht ausgelagert werden
- Reservierte (*reserved*) und verriegelte (*locked*) Seiten können nicht ausgelagert werden
- Bei Seiten, die von mehreren Tasks verwendet werden (z.B. *System V Shared Memory*) wird lediglich der Referenz- (Benutzungszähler) dekrementiert.  
Eine Freigabe der Seite erfolgt erst dann, wenn der Referenzzähler 0 geworden ist
- Nicht veränderbare (*read-only*) Seiten und Seiten, die nicht verändert worden sind (*not dirty*) werden nur verworfen. Ihr Inhalt kann jederzeit wieder hergestellt werden (aus Datei oder Auslagerungsbereich).
- Modifizierte (*dirty*) Seiten werden ausgelagert (*swapped out*), d.h. in einen Auslagerungsbereich geschrieben. Dies ist dann nicht erforderlich, wenn sich die Seite im Auslagerungs-Cache (*Swap Cache*) befindet.

### • Auslagerungsbereiche

Seiten, deren Inhalt verändert worden ist, sind bei ihrer Freigabe zu **sichern**.

Diese Sicherung erfolgt durch Ablage in einem **Auslagerungsbereich** auf einem externen Speichermedium.

Unter LINUX sind **zwei Arten** von Auslagerungsbereichen möglich :

- **Auslagerungsdatei** (*Swap File*, **Swap-Datei**)
- **Auslagerungspartition** (*Swap Partition*, *Swap Device*, **Swap-Gerät**)

Üblicherweise wird der Begriff **Swap Device** für **beide Arten** verwendet.

**Auslagerungspartitionen** sind **effektiver** als Auslagerungsdateien.

In einer Auslagerungspartition ist eine Seite immer in aufeinanderfolgenden Datenblöcken (Sektoren) abgelegt.

Bei einer Auslagerungsdatei kann eine Seite – abhängig von der Fragmentierung der Datei – auch in nicht aufeinanderfolgenden Datenblöcken abgelegt sein. In einem derartigen Fall dauert der Zugriff zu der Seite länger.

LINUX erlaubt die parallele Verwendung von mehreren Auslagerungsbereichen (Dateien und/oder Partitionen).

Defaultmäßig ist deren maximale Anzahl auf 8 festgelegt.

(Konstante **MAX\_SWAPFILES** in `include/linux/swap.h`)

Die Anzahl der tatsächlich verwendeten Auslagerungsbereiche kann während des Betriebs mit den System Calls

- **swapon** (Anmeldung eines Auslagerungsbereichs beim Kernel)
- **swapoff** (Abmeldung eines Auslagerungsbereichs beim Kernel)

verändert werden.

Jedem Auslagerungsbereich ist eine **Priorität** zugewiesen.

Zur Auslagerung einer neuen Seite verwendet LINUX den Auslagerungsbereich mit der höchsten Priorität, auf dem sich noch Platz befindet.

Sind mehrere nichtvolle Bereiche gleicher Priorität vorhanden, so werden diese zyklisch verwendet (*Round Robin*)

Es wird versucht, neu auszulagernde Speicherseiten sequentiell in Gruppen (Clustern) in den Auslagerungsbereich zu schreiben.

Dadurch sollen unnötige Kopfbewegungen der Festplatte beim aufeinanderfolgenden Auslagern von Seiten vermieden werden.

## Datenstrukturen zur Verwaltung der Auslagerungsbereiche

- **Auslagerungsbereichs-Tabelle `swap_info`** (definiert in `mm/swapfile.c`)

Tabelle (Array) zur Aufnahme von Informationen über die beim System angemeldeten Auslagerungsbereiche :

```
struct swap_info_struct  swap_info[MAX_SWAPFILES];
```

Jedem Auslagerungsbereich ist ein Eintrag in der Tabelle zugeordnet.

Die einzelnen Einträge sind über einen `next`-Index in einer vorwärts verketteten Liste logisch (nach der Priorität) sortiert.

- **Structure-Typ `struct swap_info_struct`** (definiert in `include/linux/swap.h`)

Dieser Typ dient zur Beschreibung eines Auslagerungsbereichs

```
struct swap_info_struct {  
    unsigned int flags;  
    kdev_t swap_device;  
    struct dentry * swap_file;  
    unsigned short * swap_map;  
    unsigned char * swap_lockmap;  
    unsigned int lowest_bit;  
    unsigned int highest_bit;  
    unsigned int cluster_next;  
    unsigned int cluster_nr;  
    int prio;                /* swap priority */  
    int pages;  
    unsigned long max;  
    int next;               /* next entry on swap list */  
};
```

### Bedeutung einiger Structure-Komponenten :

- |                     |                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>swap_device</b>  | : Geräte-Nummer des Swap-Grätes, wenn Auslagerungsbereich ein Swap-Gerät ist,<br>== 0, wenn Auslagerungsbereich eine Swap-Datei ist.                                                                                                                                                                                                                         |
| <b>swap_file</b>    | : Verweis auf Datei bzw Partition, die den Auslagerungsbereich realisiert                                                                                                                                                                                                                                                                                    |
| <b>swap_map</b>     | : Zeiger auf eine Tabelle von <code>short</code> -Werten, in der jeder Seite des Auslagerungsbereichs<br>ein Eintrag zugeordnet ist. In diesem Eintrag werden die Anzahl der die jeweilige Seite<br>nutzenden Prozesse gezählt.<br>Ist dieser Wert == 0, so ist die Seite nicht belegt.<br>Ist dieser Wert == 0x8000, so kann die Seite nicht benutzt werden |
| <b>swap_lockmap</b> | : Zeiger auf eine Bitmap-Tabelle, in der jeder Seite des Auslagerungsbereichs ein Bit zugeord-<br>net ist.<br>Ein Bit wird gesetzt, wenn zu der betreffenden Seite aktuell ein I/O-Zugriff stattfindet<br>→ weitere I/O-Zugriffe zu derselben Seite können verhindert werden                                                                                 |
| <b>lowest_bit</b>   | : Nr. der ersten freien Seite im Auslagerungsbereich (da die erste Seite ein Verwaltungs-Header<br>ist, ist dieser Wert niemals 0).                                                                                                                                                                                                                          |
| <b>highest_bit</b>  | : Nr. der letzten freien Seite im Auslagerungsbereich                                                                                                                                                                                                                                                                                                        |
| <b>prio</b>         | : dem Auslagerungsbereich zugeordnete Priorität                                                                                                                                                                                                                                                                                                              |
| <b>pages</b>        | : Anzahl der im Auslagerungsbereich verfügbaren Seiten                                                                                                                                                                                                                                                                                                       |
| <b>max</b>          | : maximale Anzahl von Seiten, die der Kernel nutzen darf                                                                                                                                                                                                                                                                                                     |
| <b>next</b>         | : Index des nächsten Auslagerungsbereichs in einer logisch sortierten Liste                                                                                                                                                                                                                                                                                  |

# **Betriebssysteme**

## **Kapitel 9**

### **9. Externe Datenverwaltung in LINUX**

- 9.1. Virtuelles Dateisystem (VFS)
- 9.2. Datenstrukturen zur externen Datenverwaltung
- 9.3. Registrierung und Mounten von Dateisystemen
- 9.4. System Calls zur externen Datenverwaltung

## Das Virtuelle Dateisystem (VFS) von LINUX (1)

### • Aufgabe

- ◇ Linux unterstützt eine Vielzahl von **unterschiedlichen Dateisystem-Typen** zur **externen Speicherung von Daten**. Neben den **Linux-eigenen Dateisystemen** (Extended-2, Extended-3, Reiser) können nahezu alle im **PC-Bereich gängigen Dateisysteme** (u.a. die verschiedenen FAT-Dateisysteme, das NTFS, ISO9660-CD-ROM-Dateisystem, OS/2-HPFS, Macintosh-HFS, Amiga-FFS usw) sowie diverse **Netzwerk-Dateisysteme** (NFS, Coda, AFS, SMB usw) verwendet werden.

Prinzipiell lässt sich jedes beliebige Dateisystem in Linux integrieren. Die Implementierung erfolgt entweder direkt im Kernel oder in nachladbaren Kernel-Modulen.

- ◇ Trotz der Vielfalt unterstützter Dateisystem-Typen existiert eine **einheitliche Schnittstelle** von System Calls zur **externen Datenverwaltung** : Dateien in unterschiedlichen Dateisystemen werden mit denselben Betriebssystem-funktionen angesprochen und bearbeitet. Das jeweilige reale Dateisystem muß dem Aufrufer nicht bekannt sein. Dies ermöglicht eine von den realen Dateisystemen abstrahierende Software-Schicht die als **Virtuelles Dateisystem** (*Virtual File System*, genauer *Virtual File System Switch*, VFS) bezeichnet wird.

Das VFS

- stellt den einzelnen Anwender-Prozessen die System Calls zur Dateiverwaltung zur Verfügung,
- verwaltet interne Strukturen
- übergibt die Erledigung der durch die System Calls angeforderten Aufgaben an das jeweilige reale Dateisystem
- implementiert selbst einige Standard-Aufgaben (z.B. System Call `open()` für normale Dateien)

- ◇ Mittels des VFS lassen sich auch

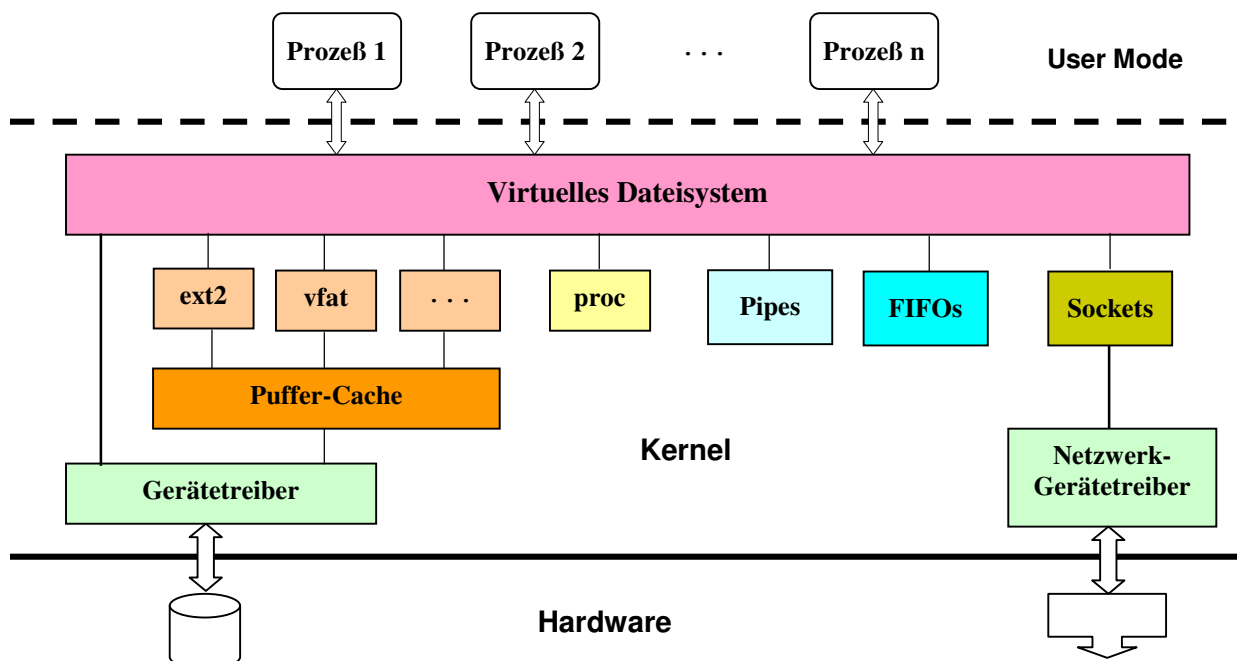
- ▷ **Dateisysteme**, die **in normalen Dateien** enthalten sind
- ▷ sowie **Pseudo-Dateisysteme**, die real gar keine externen Daten verwalten (z.B. das `/proc`-Dateisystem, das einen dateibasierten Zugriff zu internen Datenstrukturen des Kernels ermöglicht)

bearbeiten.

- ◇ Weiterhin ermöglicht das VFS auch einen **dateiartigen Zugriff** (gleiche System Calls) zu :

- ▷ **Geräten**
- ▷ **Pipes**
- ▷ **Named Pipes (FIFOs)**
- ▷ **Sockets**

### • Schichtenstruktur der externen Datenverwaltung



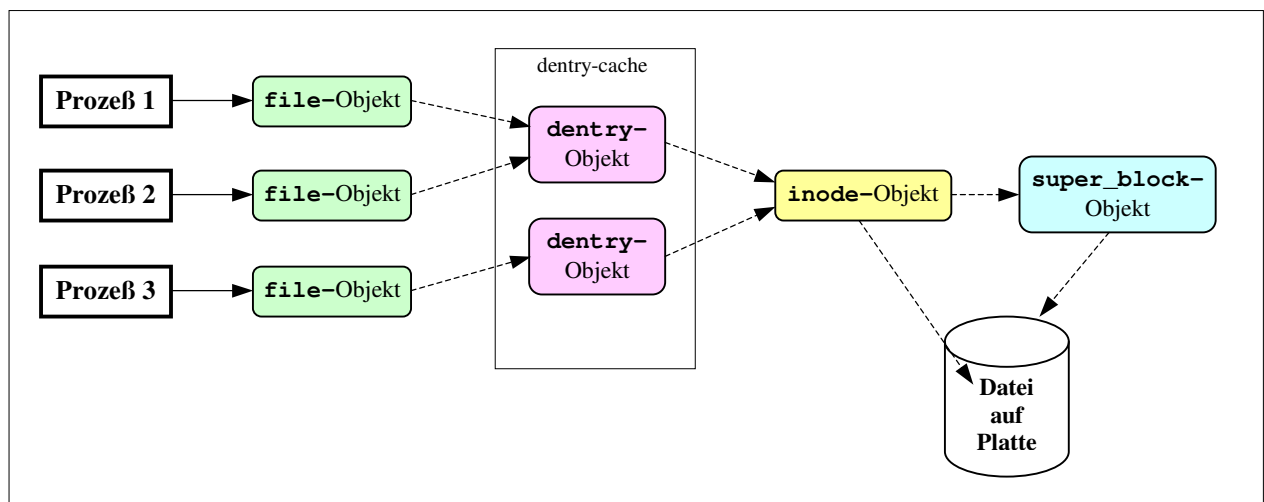
## Das Virtuelle Dateisystem (VFS) von LINUX (2)

### • Dateimodell des VFS

- ◇ Die **Grundidee** des VFS besteht in der Einführung eines **allgemeinen Dateimodells**, das in der Lage ist, alle zu unterstützenden Dateisysteme und Dateitypen zu repräsentieren.
- ◇ Dieses Modell **orientiert** sich an den **Strukturen UNIX-artiger Dateisysteme**.  
Die tatsächlichen Strukturen der unterschiedlichen realen Dateisysteme weichen hiervon zum Teil erheblich ab. Sie müssen daher durch die Implementierung des jeweiligen **speziellen Dateisystems** in die Strukturen des VFS-Dateimodells **überführt** werden.  
Bei Pseudo-Dateisystemen und den dateiartig behandelten übrigen Kommunikationsobjekten (Pipes, Sockets usw) existieren überhaupt keine derartigen Strukturen. In diesen Fällen müssen sie geeignet **nachgebildet** werden.
- ◇ Eine durch einen vom VFS bereitgestellten System Call aktivierte Betriebssystemfunktion, deren Ausführung vom jeweiligen Dateisystem abhängt (z.B. `read()` → `sys_read()`) muß die entsprechende von dem speziellen Dateisystem bereitgestellte eigentliche Bearbeitungsfunktion (z.B. `fat_file_read()` für VFAT) aufrufen. Sinnvollerweise geschieht dies über einen geeigneten Funktionspointer.
- ◇ Prinzipiell ist das VFS-Dateimodell **objekt-orientiert**.  
Allerdings ist es aus Effizienzgründen nicht in einer objekt-orientierten Sprache (z.B. C++) codiert, sondern **in C nachgebildet**.  
Die verschiedenen Objekte des Modells werden durch **C-Structures**, die neben reinen Datenkomponenten auch **Funktionspointer** (Methoden !) enthalten, realisiert.
- ◇ Das VFS-Dateimodell besteht aus den folgenden **Objekt-Typen** :
  - ▷ **Superblock (`struct super_block`)**  
Objekte dieses Typs enthalten die für die Verwaltung eines bestimmten Dateisystems notwendigen Informationen. Sie stellen Funktionen zum Zugriff zu den Verwaltungsstrukturen des Dateisystems bereit.
  - ▷ **Inode (`struct inode`)**  
Objekte dieses Typs enthalten die wesentlichen Informationen über eine bestimmte Datei. Sie stellen die Funktionen für die Dateiverwaltung zur Verfügung (z.B. `create()`, `mkdir()`, `rename()`)
  - ▷ **File (`struct file`)**  
Objekte dieses Typs enthalten die prozessrelevanten Informationen für eine geöffnete Datei. Sie enthalten die Funktionen für die Dateibearbeitung (z.B. `write()`, `read()`, `lseek()`)
  - ▷ **Directory Entry (`struct dentry`)**  
Objekte dieses Typs enthalten Informationen über die Zuordnung eines Dir-Eintrags (Name) zu der jeweiligen Datei (Inode). Sie bilden den *dentry cache*, der den Zugriff zu den zuletzt verwendeten Dir-Einträgen beschleunigt. Sie stellen Funktionen zur Verwaltung und Verwendung von `dentry`-Objekten bereit.

### • Interaktion zwischen Prozessen und VFS-Objekten (Beispiel)

- ◇ Drei Prozesse haben dieselbe Datei geöffnet, Prozess 1 und Prozess 2 verwenden denselben Hard-Link



## Datenstrukturen des VFS von LINUX (1)

### • Structure-Typ: `struct super_block`

- ◇ Diese Datenstruktur bildet den Typ für **Superblock-Objekte**. Sie ist in `include/linux/fs.h` definiert.
- ◇ Für jedes gemountetes Dateisystem wird ein Objekt dieses Typs angelegt.  
Alle Superblock-Objekte sind in einer **doppelt verketteten Ringliste** zusammengefasst
- ◇ Die enthaltenen **VFS-relevanten Informationen** entsprechen bei realen Dateisystemen grobenteils Informationen, die in speziellen auf der Platte befindlichen Beschreibungsblöcken (z.B. Superblock bei ext2, Bootsektor bei VFAT) enthalten sind.  
Daneben können **dateisystemspezifische Informationen**, die nicht vom VFS sondern von der jeweiligen Dateisystem-Implementierung benötigt werden (z.B. Belegungs-Bitmaps), enthalten sein.  
Da diese Informationen verändert werden, aber mit entsprechenden Informationen auf der Platte übereinstimmen müssen, ist ein periodisches Rausschreiben der `dirty` Superblock-Objekte erforderlich.
- ◇ **Wesentliche Komponenten** dieser Struktur sind (Kernel 2.6.16) :

|                                               |                               |                                                               |
|-----------------------------------------------|-------------------------------|---------------------------------------------------------------|
| <code>struct list_head</code>                 | <code>s_list</code>           | Pointer der Superblock-Liste                                  |
| <code>dev_t</code>                            | <code>s_dev</code>            | Geräte-Identifizier                                           |
| <code>unsigned long</code>                    | <code>s_blocksize</code>      | Blockgröße in Bytes                                           |
| <code>unsigned char</code>                    | <code>s_blocksize_bits</code> | ld (Blockgröße)                                               |
| <code>unsigned char</code>                    | <code>s_dirt</code>           | Dirty-Flag (Superblock ist verändert worden)                  |
| <code>unsigned long long</code>               | <code>s_maxbytes</code>       | maximale Dateigröße                                           |
| <code>unsigned char</code>                    | <code>s_rd_only</code>        | Read-only-Flag                                                |
| <code>struct file_system_type *</code>        | <code>s_type</code>           | Dateisystem-Typ                                               |
| <b><code>struct super_operations *</code></b> | <b><code>s_op</code></b>      | Pointer auf Superblock-Operationen                            |
| <code>struct dqout_operations *</code>        | <code>dq_op</code>            | Pointer auf Quota-Operationen                                 |
| <code>unsigned long</code>                    | <code>s_flags</code>          | Mount-Flags                                                   |
| <code>unsigned long</code>                    | <code>s_magic</code>          | Dateisystemkennung (magic number)                             |
| <code>struct dentry *</code>                  | <code>s_root</code>           | Pointer auf <code>dentry</code> -Objekt des Mount-Directories |
| <code>struct mutex</code>                     | <code>s_lock</code>           | Superblock-Semaphore                                          |
| <code>int</code>                              | <code>s_count</code>          | Superblock-Referenzzähler                                     |
| <code>struct list_head</code>                 | <code>s_inodes</code>         | Liste aller Inodes                                            |
| <code>struct list_head</code>                 | <code>s_dirty</code>          | Liste aller geänderten Inodes                                 |
| <code>struct block_device *</code>            | <code>s_bdev</code>           | Blockgerät, auf dem sich Dateisystem befindet                 |
| <code>void*</code>                            | <code>s_fs_info</code>        | Pointer auf dateisystemspezifische Information                |

### • Superblock-Operationen `struct super_operations`

- ◇ Diese Struktur enthält Pointer auf die – dateisystemspezifischen – Superblock-Operationen.  
Diese Funktionen dienen zum Lesen und Schreiben einzelner Inodes, zum Schreiben des Super-Blocks sowie zum Auslesen von Dateisysteminformationen. Sie überführen die konkrete Darstellung dieser Informationen auf dem Datenträger in die allgemeine Form im VFS-Superblock-Objekt.  
Häufig ist für ein bestimmtes Dateisystem nur eine Teilmenge der Operationen definiert.  
Für nicht implementierte Funktionen ist der **NULL**-Pointer eingetragen.

```
struct super_operations
{
    struct inode* (*alloc_inode) (struct super_block*);
    void (*destroy_inode) (struct inode*);
    void (*read_inode) (struct inode *);
    void (*dirty_inode) (struct inode *);
    void (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*drop_inode) (struct inode*);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs) (struct super_block*, int);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct statfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    /* 3 weitere Operationen, die z. Zt. nur vom Reiser-FS verwendet werden */
};
```

## Datenstrukturen des VFS von LINUX (2)

### • Structure-Typ: `struct inode`

- ◇ Diese Datenstruktur bildet den Typ für **Inode-Objekte**. Sie ist in `include/linux/fs.h` definiert.
- ◇ **Pro Datei** existiert ein **Inode-Objekt**, unabhängig von der Anzahl von Links, über die die Datei referiert werden kann. In dem Inode-Objekt sind alle vom VFS benötigten Informationen über die Datei zusammengefasst. In einigen Dateisystemen wird ein Großteil dieser Informationen durch entsprechende Datenstrukturen auf dem Datenträger (dort auch häufig Inode genannt, z.B. bei UNIX-artigen Dateisystemen) zur Verfügung gestellt, bei anderen sind sie – zumindest teilweise – im Directory-Eintrag enthalten (z.B. FAT-Dateisysteme). Neben den **vom VFS allgemein** verwendeten können auch **dateisystemspezifische Informationen** abgelegt sein.
- ◇ Jedem Inode-Objekt ist eine in dem zugehörigen Dateisystem eindeutige **Inode-Nummer** zugeordnet.
- ◇ Jedes Inode-Objekt ist in einer von drei möglichen doppelt verketteten Ringlisten enthalten :
  - ▷ **Liste der verwendeten** und nicht geänderten **Inodes**  
Das erste und das letzte Element dieser Liste werden durch die statisch-globale Variable `struct list_head inode_in_use` (definiert in `fs/inode.c`) referiert
  - ▷ **Liste der verwendeten** und **geänderten Inodes**  
Das erste und das letzte Element dieser Liste werden durch die Komponente `s_dirty` des zugehörigen Superblock-Objekts referiert
  - ▷ **Liste der freien** (nicht verwendeten) **Inodes**.  
Das erste und das letzte Element dieser Liste werden durch die statisch-globale Variable `struct list_head inode_unused` (definiert in `fs/inode.c`) referiert
- ◇ Für einen schnellen Zugriff sind alle verwendeten (geänderten und nicht geänderten) Inodes zusätzlich in einer **Hashtabelle** enthalten (ref. durch `struct hlist_head* inode_hashtable` (definiert in `fs/inode.c`)). Der **Hash-Wert** (=Tabellenindex) wird aus der **Inode-Nummer** und der **Adresse** des zugehörigen **Superblock-Objekts** gebildet. Inodes mit **gleichem Hash-Wert** sind in je einer doppelt verketteten Liste (**Hash-Liste**) zusammengefaßt.
- ◇ **Wesentliche Komponenten** dieser Struktur sind (Kernel 2.6.16) :

|                                               |                               |                                                       |
|-----------------------------------------------|-------------------------------|-------------------------------------------------------|
| <code>struct hlist_node</code>                | <code>i_hash</code>           | Pointer der Hash-Liste                                |
| <code>struct list_head</code>                 | <code>i_list</code>           | Pointer der Inode-Liste, in der sich Inode befindet   |
| <code>struct list_head</code>                 | <code>i_dentry</code>         | Pointer der Liste der zugehörigen Dentry-Objekte      |
| <code>unsigned long</code>                    | <code>i_ino</code>            | Inode-Nummer                                          |
| <code>atomic_t</code>                         | <code>i_count</code>          | Verwendungszähler                                     |
| <code>umode_t</code>                          | <code>i_mode</code>           | Dateityp und Zugriffsrechte (File Mode)               |
| <code>unsigned int</code>                     | <code>i_nlink</code>          | Anzahl der Hard Links auf die Datei                   |
| <code>uid_t</code>                            | <code>i_uid</code>            | User ID                                               |
| <code>gid_t</code>                            | <code>i_gid</code>            | Group ID                                              |
| <code>dev_t</code>                            | <code>i_rdev</code>           | Identifizier des Geräts                               |
| <code>loff_t</code>                           | <code>i_size</code>           | Dateilänge in Bytes                                   |
| <code>struct timespec i_atime,</code>         | <code>i_mtime, i_ctime</code> | Zeit des letzten Dateizugriffs, -änderung, -erzeugung |
| <code>unsigned long</code>                    | <code>i_blksize</code>        | Blockgröße in Bytes                                   |
| <code>unsigned long</code>                    | <code>i_version</code>        | Versions-Nummer (increment. nach jeder Verwendung)    |
| <code>unsigned long</code>                    | <code>i_blocks</code>         | Anzahl Datenblöcke der Datei                          |
| <code>spinlock_t</code>                       | <code>i_lock</code>           | Spinlock                                              |
| <code>struct mutex</code>                     | <code>i_mutex</code>          | Semaphor zur Zugriffssteuerung                        |
| <b><code>struct inode_operations *</code></b> | <b><code>i_op</code></b>      | Pointer auf Inode-Operationen                         |
| <code>struct file_operations *</code>         | <code>i_fop</code>            | Pointer auf File-Operationen                          |
| <code>struct super_block *</code>             | <code>i_sb</code>             | Pointer auf zugehöriges Superblock-Objekt             |
| <code>wait_queue_head_t</code>                | <code>i_wait</code>           | Inode-Warteschlange                                   |
| <code>struct file_lock *</code>               | <code>i_flock</code>          | Pointer auf Liste der Dateisperren                    |
| <code>struct address_space *</code>           | <code>i_mapping</code>        | Pointer auf Speicher, in den Datei eingeblendet ist   |
| <code>struct pipe_inode_info *</code>         | <code>i_pipe</code>           | ggfls Pointer auf Pipe-Beschreibungsstruktur          |
| <code>struct block_device *</code>            | <code>i_bdev</code>           | ggfls Pointer auf BlockDev-Beschreibungsstruktur      |
| <code>struct cdev *</code>                    | <code>i_cdev</code>           | ggfls Pointer auf CharDev-Beschreibungsstruktur       |
| <code>unsigned long</code>                    | <code>i_state</code>          | Inode-Status                                          |
| <code>unsigned int</code>                     | <code>i_flags</code>          | Dateisystem-Mount-Flag                                |
| <code>unsigned char</code>                    | <code>i_sock</code>           | true, falls Socket                                    |
| <code>atomic_t</code>                         | <code>i_writcount</code>      | Zähler für schreibende Prozesse                       |
| <code>union {void* generic_ip;} u</code>      |                               | dateisystemspezifische Information                    |

### Datenstrukturen des VFS von LINUX (3)

#### • Inode-Operationen: `struct inode_operations`

- ◇ Diese Struktur fasst Pointer auf die mit einem Inode-Objekt assoziierten – i.a. dateisystemspezifischen und vom jeweiligen Dateityp abhängigen – Funktionen zusammen. Diese Funktionen dienen hauptsächlich der **Dateiverwaltung**. Sie werden meist direkt in der Implementierung der entsprechenden Betriebssystemfunktionen aufgerufen.
- ◇ Das Eintragen des Pointers auf die jeweiligen Inode-Operationen in das Inode-Objekt (Komponente **i\_op**) erfolgt zusammen mit dem Füllen der übrigen Felder des Objekts durch die zugehörige Superblock-Operation **read\_inode()**.

**Prinzipielles Codebeispiel** hierfür (hier für ext2-Dateisystem) :

```
if (S_ISREG(inode->i_mode)) {
    inode->i_op = &ext2_file_inode_operations;
    inode->i_fop = &ext2_file_operations;
    inode->i_mapping->a_ops = &ext2_aops;
}
else if (S_ISDIR(inode->i_mode)) {
    inode->i_op = &ext2_dir_inode_operations;
    inode->i_fop = &ext2_dir_operations;
    inode->i_mapping->a_ops = &ext2_aops;
}
else if (S_ISLNK(inode->i_mode)) {
    inode->i_op = &page_symlink_inode_operations;
    inode->i_mapping->a_ops = &ext2_aops;
}
```

- ◇ Auch von den Inode-Operationen ist für ein bestimmtes Dateisystem und einen bestimmten File-Type häufig nur eine Teilmenge definiert. Für nicht implementierte Funktionen ist der **NULL**-Pointer eingetragen.
- ◇ Definition von `struct inode_operations` (in `include/linux/fs.h`) (Kernel 2.6.16) :

```
struct inode_operations {
    int (*create) (struct inode *, struct dentry *, int);
    struct dentry * (*lookup) (struct inode *, struct dentry *);
    int (*link) (struct dentry *, struct inode *, struct dentry *);
    int (*unlink) (struct inode *, struct dentry *);
    int (*symlink) (struct inode *, struct dentry *, const char *);
    int (*mkdir) (struct inode *, struct dentry *, int);
    int (*rmdir) (struct inode *, struct dentry *);
    int (*mknod) (struct inode *, struct dentry *, int, int);
    int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char *, int);
    int (*follow_link) (struct dentry *, struct nameidata *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct dentry *, struct iattr *);
    int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    void (*truncate_range) (struct inode *, loff_t, l_offt);
};
```

#### • Weitere Inode-Operationen für Blockgeräte

- ◇ Für Blockgeräte existieren weitere Inode-Operationen. Pointer auf diese Funktionen sind in der Struktur **struct block\_device\_operations** zusammengefasst. (def. in `include/linux/fs.h`)
- ◇ Referierung dieser Struktur: in `struct inode` : **struct block\_device\* i\_bdev**  
in `struct block_device` (`include/linux/fs.h`) : **struct gendisk\* bd\_disk**  
in `struct gendisk` (`include/linux/genhd.h`) : **struct block\_device\_operations \*fops**



## Datenstrukturen des VFS von LINUX (4)

### • Structure-Typ: **struct dentry**

- ◇ Diese Datenstruktur bildet den Typ für **Dentry-Objekte**. Sie ist in `include/linux/dcache.h` definiert.
- ◇ Jeder zugriffene Directory-Eintrag wird vom VFS in ein Dentry-Objekt (Directory-Entry-Objekt) transformiert. Das bedeutet, dass für jede Komponente eines Zugriffspfad ein eigenes Dentry-Objekt erzeugt wird (einschließlich eines für das Root-Directory).  
Ein Dentry-Objekt stellt den **Zusammenhang** zwischen dem jeweiligen **Pfadnamen** eines Eintrags und dem **Inode-Objekt**, das die dadurch ausgewählte Datei beschreibt, her.
- ◇ Die Gesamtheit der Dentry-Objekte bilden den **dentry cache**, der ein **schnelleres Auflösen von Pfadnamen** (d.h. Auffinden des zugehörigen Inode-Objekts) ermöglicht.  
Für den schnellen Zugriff zum Cache existiert eine **Hash-Tabelle** `struct hlist_head* dentry_hashtable` (definiert in `fs/dcache.c`). Der **Hash-Wert** wird aus dem **Pfadnamen** gebildet. Dentry-Objekte mit **gleichem Hash-Wert** sind jeweils über eine doppelt-verkettete Liste (**Hash-Liste**) zusammengefasst.
- ◇ **Dieselbe Datei** kann durch **mehrere Directory-Einträge** (Hard Links) referiert werden.  
Für eine derartige Datei existiert nur **ein Inode-Objekt**, aber für **jeden** – zugriffenen – **Directory-Eintrag** wird ein **eigenes Dentry-Objekt** angelegt.  
Alle zum selben Inode-Objekt gehörenden Dentry-Objekte sind in einer doppelt-verketteten Liste zusammengefasst, die durch die Komponente `i_dentry` des Inode-Objekts referiert wird.
- ◇ Jedes Dentry-Objekt enthält einen **Verwendungszähler** (Komponente `d_count`), der die Anzahl der aktuellen Verwendungen des Objekts durch den Kernel speichert.  
Dieser Zähler wird `==0` wenn das Objekt (d.h. der entsprechende Directory-Eintrag) nicht mehr verwendet wird.  
Alle nicht mehr verwendeten Dentry-Objekte (Zustand *unused*) werden nicht sofort wieder freigegeben, sondern verbleiben zunächst im **dentry cache**. Sie werden an den Anfang einer doppelt verketteten **LRU-Liste** (LRU – *least recently used*) gesetzt. Erst wenn Speicherplatz für neue Dentry-Objekte benötigt wird, werden Objekte vom Ende der Liste, an dem sich ja die am längsten nicht verwendeten Objekte befinden, entfernt.  
Solange ein Dentry-Objekt in der Liste enthalten ist, kann bei einer erneuten Verwendung der darüber referierten Datei sehr schnell das zugehörige Inode-Objekt, das sich ja auch noch im Speicher befindet, gefunden werden.
- ◇ Wesentliche **Komponenten** dieser Struktur sind (Kernel 2.6.16) :

|                                                |                             |                                                                |
|------------------------------------------------|-----------------------------|----------------------------------------------------------------|
| <code>atomic_t</code>                          | <code>d_count</code>        | Verwendungszähler                                              |
| <code>unsigned int</code>                      | <code>d_flags</code>        | Dentry-Flags                                                   |
| <code>spinlock_t</code>                        | <code>d_lock</code>         | per dentry lock                                                |
| <b><code>struct inode *</code></b>             | <b><code>d_inode</code></b> | Pointer auf zugehöriges Inode-Objekt                           |
| <code>struct dentry *</code>                   | <code>d_parent</code>       | Pointer auf Dentry-Objekt des Eltern-Directories               |
| <code>struct hlist_node</code>                 | <code>d_hash</code>         | Pointer der Hash-Liste                                         |
| <code>struct list_head</code>                  | <code>d_lru</code>          | Pointer der LRU-Liste (wenn <code>dcount=0</code> )            |
| <code>struct list_head</code>                  | <code>d_child</code>        | Pointer der Liste der übrigen Dentry-Objekte im Eltern-Dir.    |
| <code>struct list_head</code>                  | <code>d_subdirs</code>      | Pointer der Liste von Sub-Directories (wenn Eintrag Dir. ist)  |
| <code>struct list_head</code>                  | <code>d_alias</code>        | Pointer der Liste der Dentry-Objekte für gleiches Inode-Objekt |
| <code>int</code>                               | <code>d_mounted</code>      | true (1), wenn Mount-Point                                     |
| <b><code>struct qstr</code></b>                | <b><code>d_name</code></b>  | Dateipfad                                                      |
| <code>unsigned long</code>                     | <code>d_time;</code>        | von der Methode <code>*d_revalidate()</code> verwendet         |
| <b><code>struct dentry_operations *</code></b> | <b><code>d_op</code></b>    | Pointer auf Dentry-Operationen                                 |
| <code>struct super_block *</code>              | <code>d_sb;</code>          | Pointer auf zugehöriges Superblock-Objekt                      |
| <code>void *</code>                            | <code>d_fsdata;</code>      | dateisystemabhängige Daten                                     |
| <code>unsigned char</code>                     | <code>d_iname[16]</code>    | Kurzname für Datei                                             |

- ◇ Die Struktur **`struct qstr`** dient zur Darstellung von Strings ("*Quick String*").  
Sie ist wie folgt in `include/linux/dcache.h` definiert :

```
struct qstr {
    unsigned int hash;
    unsigned int len;
    const unsigned char * name;
};
```

## Datenstrukturen des VFS von LINUX (5)

### • Dentry-Operationen: `struct dentry_operations`

- ◇ Diese Struktur umfasst Pointer auf Funktionen zur Verwaltung und Verwendung von Dentry-Objekten. Die Adresse einer derartigen Struktur ist in der Komponente `d_op` eines Dentry-Objekts enthalten
- ◇ Diese Funktionen werden nur durch einige wenige Dateisysteme zur Verfügung gestellt, d.h. im allgemeinen ist für alle Komponenten ein `NULL`-Pointer eingetragen. Das VFS führt dann in ihm selbst angesiedelte Default-Funktionen aus.
- ◇ Definition von `struct dentry_operations` (in `include/linux/dcache.h`) (Kernel 2.6.16):

```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, int);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete) (struct dentry *);
    void (*d_release) (struct dentry *);
    void (*d_iput) (struct dentry *, struct inode *);
};
```

## Datenstrukturen des VFS von LINUX (6)

### • Structure-Typ: **struct file**

- ◇ Diese Datenstruktur bildet den Typ für **File-Objekte**. Sie ist in `include/linux/fs.h` definiert.
- ◇ Für jede von einem **Prozess geöffnete Datei** wird ein **File-Objekt** angelegt.
- ◇ Prinzipiell kann dieselbe Datei von mehreren Prozessen – gegebenenfalls auch für unterschiedliche – Zugriffsarten geöffnet werden.  
Da der Bearbeitungszustand einer derartigen Datei i.a. in jedem Prozess unterschiedlich sein wird, werden für jeden **Prozess** eigene entsprechende **Bearbeitungsinformationen** benötigt. Diese sind in jeweils einem **eigenen File-Objekt** zusammengefasst.  
Während die die Datei beschreibenden Informationen in einem **nur einmal vorhandenen Inode-Objekt** enthalten sind.
- ◇ Die wesentlichste in einem File-Objekt enthaltene Information ist der **File-Pointer**, der die jeweilige aktuelle Bearbeitungsposition innerhalb der Datei darstellt.
- ◇ Jedes File-Objekt ist in einer von zwei möglichen doppelt verketteten Ringlisten enthalten :
  - ▷ **Liste der verwendeten File-Objekte**  
Jedes Element der Liste wird von wenigstens einem Prozess verwendet.  
Da sich mehrere Prozesse geöffnete Dateien auch teilen können (Threads !), ist die Verwendung desselben File-Objekts durch mehrere Prozesse möglich.  
Das erste und das letzte Element dieser Liste werden durch die statisch-globale Variable (Listenkopf) `struct list_head anon_list` (definiert in `fs/file_table.c`) referiert
  - ▷ **Liste der freien (nicht verwendeten) File-Objekte**  
In diese Liste wird ein File-Objekt, das von keinem Prozess mehr verwendet wird, aufgenommen.  
Wird später ein neues File-Objekt benötigt, wird versucht, es dieser Liste zu entnehmen.  
Die Liste dient somit als Speicher-Cache für File-Objekte.  
Der Kernel stellt sicher, dass die Liste immer wenigstens `NR_RESERVED_FILES` (defaultmässig = 10, definiert in `include/linux/fs.h`) umfasst.  
Das erlaubt dem Super-User auch dann eine Datei zu öffnen, wenn kein dynamischer Speicher im System mehr alloziert werden kann.  
Das erste und das letzte Element dieser Liste werden durch die statisch-globale Variable (Listenkopf) `struct list_head free_list` (definiert in `fs/file_table.c`) referiert
- ◇ Die **Komponenten** dieser Struktur sind (Kernel 2.6.16) :

|                                              |                           |                                                     |
|----------------------------------------------|---------------------------|-----------------------------------------------------|
| <code>struct list_head</code>                | <code>f_list</code>       | Pointer der Liste, in der sich File-Objekt befindet |
| <code>struct dentry *</code>                 | <code>f_dentry</code>     | Pointer auf das assoziierte Dentry-Objekt           |
| <code>struct vfsmount *</code>               | <code>f_vfsmnt</code>     | Pointer auf das assoziierte gemountete Dateisystem  |
| <b><code>struct file_operations *</code></b> | <b><code>f_op</code></b>  | Pointer auf File-Operationen                        |
| <code>atomic_t</code>                        | <code>f_count</code>      | Verwendungszähler des File-Objekts                  |
| <code>unsigned int</code>                    | <code>f_flags</code>      | Öffnungs-Flags der Datei                            |
| <code>mode_t</code>                          | <code>f_mode</code>       | aktuelle Zugriffsart zur Datei                      |
| <b><code>loff_t</code></b>                   | <b><code>f_pos</code></b> | aktuelle Bearbeitungsposition (File Pointer)        |
| <code>struct fown_struct</code>              | <code>f_owner</code>      | Owner-Daten für asynchrone I/O mittels Signalen     |
| <code>unsigned int</code>                    | <code>f_uid</code>        | User ID                                             |
| <code>unsigned int</code>                    | <code>f_gid</code>        | Group ID                                            |
| <code>struct file_ra_state</code>            | <code>f_ra</code>         | Read-ahead-Status                                   |
| <code>unsigned long</code>                   | <code>f_version</code>    | Versions-Nummer (increment. nach jeder Verwendung)  |
| <code>void *</code>                          | <code>f_security</code>   | Security Module                                     |
| <code>void *</code>                          | <code>private_data</code> | Informationen für Geräte-Treiber                    |
| <code>struct list_head</code>                | <code>f_ep_links</code>   | Pointer der Liste der Eventpoll-Links               |
| <code>spinlock_t</code>                      | <code>f_ep_lock</code>    | Eventpoll Lock                                      |
| <code>struct address_space *</code>          | <code>f_mapping</code>    | Page Cache Mapping                                  |

## Datenstrukturen des VFS von LINUX (7)

### • File-Operationen: struct file\_operations

- ◇ Diese Struktur umfasst Pointer auf die dateisystemspezifischen **Dateibearbeitungsfunktionen**. Die meisten dieser Funktionen dienen der Realisierung der Funktionalität bzw einer Teilfunktionalität von System Calls in dem jeweiligen Dateisystem. Sie werden daher i.a. direkt in der Implementierung der entsprechenden im VFS angesiedelten Betriebssystemfunktion aufgerufen.
- ◇ Bei **Geräten** werden diese Funktionen durch den jeweiligen **Gerätetreiber** bereitgestellt.
- ◇ Beim Füllen eines Inode-Objekts mittels der Superblock-Operation `read_inode()` wird in diesem auch ein Pointer auf die **Standard-File-Operationen** des jeweiligen Dateisystems mit abgelegt (Komponente `i_fop`). Mit diesem Pointer wird dann die Komponente `f_op` des beim Öffnen einer Datei neu angelegten File-Objekts initialisiert.
- ◇ Auch für diese Funktionen gilt, dass – abhängig vom Dateisystem und vom speziellen File-Typ – gegebenenfalls nur eine Teilmenge implementiert sein muss. Für nicht implementierte Funktionen ist in einem solchen Fall ein **NULL**-Pointer in die Pointer-Struktur eingetragen. Gegebenenfalls wird abhängig von der Funktion dann eine Standardimplementierung ausgeführt oder ein Fehler erzeugt.
- ◇ Definition von `struct file_operations` (in `include/linux/fs.h`) (Kernel 2.6.16):

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long,
   unsigned long, unsigned long);

    int (*check_flags) (int);
    int (*dir_notify) (struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    int (*open_exec) (struct inode *);
};
```

## Datenstrukturen des VFS von LINUX (8)

### • Kurzbeschreibung der Aufgaben einzelner File-Operationen

|                                                                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>loff_t (*llseek) (struct file *, loff_t, int);</pre> <p>Veränderung der Bearbeitungsfunktion innerhalb der Datei</p>                                                                                                                                                                                                                                                         |
| <pre>ssize_t (*read) (struct file *, char *, size_t, loff_t *);</pre> <p>Lesen einer Anzahl Bytes aus einer Datei ab einer bestimmten Position, Ablage der gelesenen Bytes in einen Buffer im User-Adressraum. Anschließend Erhöhung der Bearbeitungsposition</p>                                                                                                                 |
| <pre>ssize_t (*write) (struct file *, const char *, size_t, loff_t *);</pre> <p>Schreiben einer Anzahl Bytes in eine Datei ab einer bestimmten Position, Entnahme der zu schreibenden Bytes aus einem Buffer im User-Adressraum. Anschließend Erhöhung der Bearbeitungsposition</p>                                                                                               |
| <pre>int (*readdir) (struct file *, void *, filldir_t);</pre> <p>Ermittlung des nächsten Eintrags in einem Directory.<br/>Diese Funktion muss nur für den File-Typ "directory" implementiert werden</p>                                                                                                                                                                           |
| <pre>unsigned int (*poll) (struct file *, struct poll_table_struct *);</pre> <p>Überprüfung, ob Daten von einer Datei gelesen oder in eine Datei geschrieben werden können.<br/>Diese Funktion ist i.a. nur für Geräte und Sockets sinnvoll.</p>                                                                                                                                  |
| <pre>int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);</pre> <p>Einstellung gerätespezifischer Parameter durch Senden eines Kommandos an ein Gerät.<br/>Diese Funktion muss nur für Geräte implementiert werden.</p>                                                                                                                                     |
| <pre>int (*mmap) (struct file *, struct vm_area_struct *);</pre> <p>Abbildung einer Datei bzw eines Teils einer Datei ab einer bestimmten Position in den Arbeitsspeicher des User-Adressraums</p>                                                                                                                                                                                |
| <pre>int (*open) (struct inode *, struct file *);</pre> <p>Ergänzende Funktionalität zum Öffnen einer Datei. Die Funktion muss i.a. nur für Geräte implementiert werden.<br/>Das Öffnen normaler Dateien wird vollständig durch die VFS-Funktion <code>sys_open()</code> realisiert (u.a. Anlegen bzw "Füllen" eines File-Objekts).</p>                                           |
| <pre>int (*flush) (struct file *);</pre> <p>Die Funktion wird vom VFS beim Schließen einer Referenz zu einer offenen Datei aufgerufen, d.h. wenn die Komponente <code>f_count</code> des File-Objekts decremementiert wird. Ihre tatsächliche Aufgabe ist dateisystemabhängig, z.B. kann sie das Heraus-schreiben gepufferter Daten in die Datei bewirken.</p>                    |
| <pre>int (*release) (struct inode *, struct file *);</pre> <p>Diese Funktion wird vom VFS bei der Freigabe eines File-Objekts aufgerufen, d.h. wenn die letzte Referenz zu einer offenen Datei geschlossen wird (<code>f_count=0</code> geworden ist). Sie ist in erster Linie für Geräte vorgesehen.</p>                                                                         |
| <pre>int (*fsync) (struct file *, struct dentry *, int datasync);</pre> <p>Schreibt alle Schreib-Datenpuffer in die Datei heraus</p>                                                                                                                                                                                                                                              |
| <pre>int (*fasync) (int, struct file *, int);</pre> <p>Diese Funktion wird vom VFS aufgerufen, wenn sich ein Prozess mit Hilfe des System Calls <code>fcntl()</code> für eine asynchrone Benachrichtigung (durch das Signal <code>SIGIO</code>) beim Eintreffen von Daten an- oder abmeldet.<br/>Der dritte Parameter bestimmt, ob Anmeldung (=1) oder Abmeldung (=0) erfolgt</p> |
| <pre>int (*lock) (struct file *, int, struct file_lock *);</pre> <p>Setzen und Überprüfen von Dateisperren. Die Funktion wird von der VFS-Funktion <code>sys_fcntl()</code> aufgerufen.</p>                                                                                                                                                                                       |

## Prozessspezifische Datenstrukturen zur externen Datenverwaltung in LINUX (1)

### • Typ zur Darstellung einer Menge von File-Deskriptoren : **fd\_set**

- ◇ **Geöffnete Dateien** werden in den System Calls von LINUX durch – prozessbezogene – **File-Deskriptoren** referiert. Ein File-Descriptor ist eine positive ganze Zahl zwischen 0 und – zur Zeit üblicherweise – 1023. D.h. ein Prozess kann damit maximal 1024 Dateien gleichzeitig offen haben. Grundsätzlich werden für jeden neu erzeugten Prozess 3 Dateien (genauer Geräte) automatisch geöffnet, die über die folgenden File-Deskriptoren angesprochen werden :
  - 0 Standardeingabe
  - 1 Standardausgabe
  - 2 Standard-Fehlerrausgabe
- ◇ Zur Beschreibung einer **Menge von File-Deskriptoren** dient der Datentyp **fd\_set**. Dieser Typ realisiert eine **Bitmap**, in der jedes Bit entsprechend seiner Position einem File-Descriptor zugeordnet ist. Ein File-Descriptor ist in der Menge enthalten, wenn das ihm zugeordnete Bit gesetzt ist.

- ◇ Der Typ **fd\_set** ist definiert in der Headerdatei `include/linux/types.h` im Zusammenhang mit der von dieser eingebundenen Headerdatei `include/linux/posix_types.h` als Structure-Typ :

```
typedef struct
{ unsigned long fds_bits[__FDSET_LONGS];
} fd_set;
```

wobei `__FDSET_LONGS` definiert ist zu 32.

### • Structure-Typ : **struct fs\_struct** (definiert in `include/linux/fs_struct.h`)

- ◇ Diese Struktur enthält **prozessspezifische Informationen** über das **Gesamt-Dateisystem**. Im wesentlichen sind das
  - ▷ **Root-Directory** des Prozesses (Pointer auf das entsprechende Dentry-Objekt)
  - ▷ Aktuelles **Arbeits-Directory** des Prozesses (Pointer auf das entsprechende Dentry-Objekt)
  - ▷ Maske **umask** für die **Zugriffsberechtigungen** (Dateiattribut) bei neu erzeugten Dateien (**Datei kreierungsmaske**). Diese Maske legt die Zugriffsberechtigungen fest, die bei neu erzeugten Dateien **nicht gesetzt** werden dürfen. Die beim Erzeugen einer neuen Datei anzugebenden Zugriffsrechte (z.B. Parameter `mode` bei `creat()`) werden mit der invertierten Maske `umask` bitweise UND-verknüpft. Die Dateiattribut-Bits `suid`, `sgid`, `sticky` können in der Maske nicht enthalten sein.
- ◇ Weiterhin ist u.a. ein **Verwendungszähler** für die Anzahl der Prozesse, die sich dieselbe `fs_struct`-Variable teilen, als Komponente enthalten. (für Threads möglich, s. System Call `clone()`)
- ◇ Definition (Kernel 2.6.11) :

```
struct fs_struct {
    atomic_t count;
    rwlock_t lock;
    int umask;
    struct dentry * root, * pwd, * alroot;
    struct vfsmount * rootmnt, * pwdmnt, * alrootmnt;
};
```

- ◇ Der **Prozessdeskriptor** (Prozessverwaltungsstruktur) enthält einen Pointer auf die für den Prozess relevante `fs_struct`-Variable (Komponente **fs**)

## Prozessspezifische Datenstrukturen zur externen Datenverwaltung in LINUX (2)

- **Structure-Typ: struct files\_struct**

(definiert in `include/linux/sched.h`, ab Kernel 2.6.x in `include/linux/file.h`)

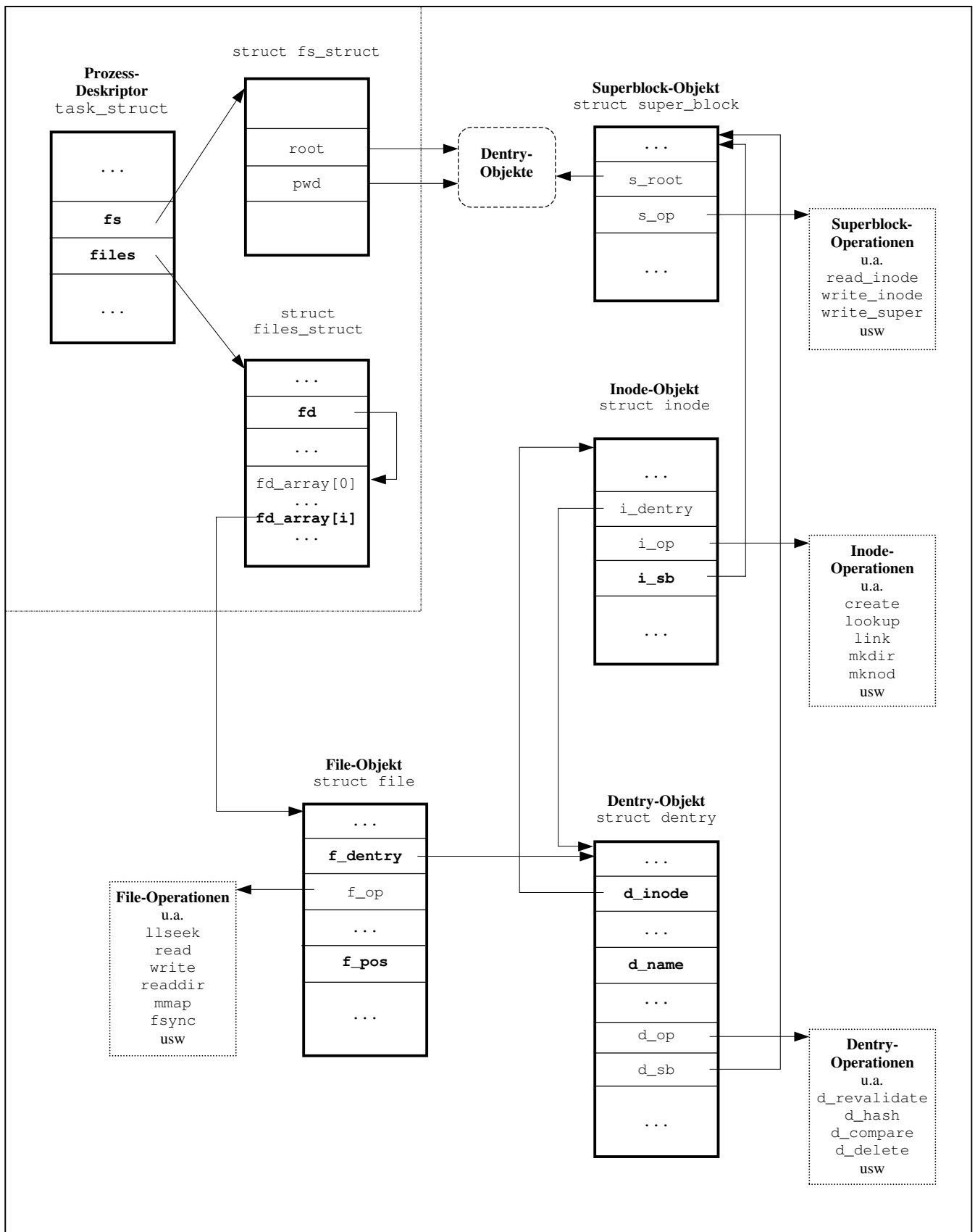
- ◇ Diese Datenstruktur enthält Informationen über die **geöffneten Dateien** eines Prozesses.
- ◇ Definition (Kernel 2.6.11):

```
struct files_struct {  
    atomic_t count;                /* Verwendungszähler */  
    spinlock_t file_lock;  
    int max_fds;                   /* Grösse des File-Objekt-Arrays */  
    int max_fdset;                 /* max. mögliche Anz. von File-Deskriptoren */  
    int next_fd;                   /* Index der nächsten freien Array-Komponente */  
    struct file ** fd;             /* zeigt anfangs auf fd_array */  
    fd_set * close_on_exec;        /* zeigt anfangs auf close_on_exec_init */  
    fd_set * open_fds;             /* zeigt anfangs auf open_fds_init */  
    fd_set close_on_exec_init;     /* Menge der bei exec() zu schliessenden File-Deskr. */  
    fd_set open_fds_init;          /* Menge der File-Deskriptoren */  
    struct file * fd_array[NR_OPEN_DEFAULT]; /* Array von File-Objekt-Pointern */  
};
```

- ◇ Als wesentliche Komponente ist ein Pointer **fd** auf ein **Array von File-Objekt-Pointern** (`struct file **`) enthalten.  
Beim Öffnen einer Datei wird die Adresse des die Datei beschreibenden File-Objekts in eine freie Komponente dieses Arrays eingetragen (freie Komponenten enthalten den `NULL`-Pointer).  
Der **Index** der entsprechenden Array-Komponente ist der **File-Deskriptor**, über den die Datei in System Calls referiert werden kann.  
Es ist möglich, dass das gleiche File-Objekt über mehrere File-Deskriptoren referiert wird. Dies kann durch geeignete System Calls bewirkt werden (z.B. `dup2()`). In einem derartigen Fall zeigen mehrere Array-Komponenten auf dasselbe File-Objekt.
- ◇ Die Struktur enthält selbst auch ein Array von File-Objekt-Pointern → Komponente **fd\_array**.  
Die Grösse dieses Arrays `NR_OPEN_DEFAULT` ist in `include/linux/sched.h` standardmässig definiert zu 32.  
Defaultmässig zeigt die Komponente `fd` auf dieses Array. Falls ein Prozess mehr als `NR_OPEN_DEFAULT` offene Dateien benötigt, alloziert der Kernel ein neues größeres Array, auf das dann `fd` umgesetzt wird.
- ◇ Die Komponente **open\_fds** referiert die **Menge der File-Deskriptoren** der jeweils aktuell geöffneten Dateien. Normalerweise wird diese durch die ebenfalls in der Struktur enthaltene Komponente **open\_fds\_init** gebildet.
- ◇ Die Komponente **close\_on\_exec** referiert die **Menge der File-Deskriptoren** derjenigen **Dateien**, die beim **Überlagern des Prozesses** (System Call `execve()`) **geschlossen** werden sollen. Normalerweise wird diese Menge durch die ebenfalls in der Struktur enthaltene Komponente **close\_on\_exec\_init** gebildet.
- ◇ Die für einen Prozess relevante `files_struct`-Variable wird im **Prozessdeskriptor** (Prozessverwaltungsstruktur) des Prozesses durch die Komponente **files** referiert.
- ◇ Auch eine `files_struct`-Variable kann von mehreren Prozessen gleichzeitig verwendet werden (Threads!). Die Anzahl der Prozesse, die sich ein- und dieselbe `files_struct`-Variable teilen, ist in deren Komponente **count** abgelegt.

## Zusammenhang der Datenstrukturen zur externen Datenverwaltung in LINUX

### • Überblick





## Registrierung von Dateisystemen

### • Allgemeines

- ◇ Der Begriff "**Dateisystem**" wird in **doppeltem Sinn** verwendet :
  - ▷ Einerseits wird darunter eine **bestimmte Organisationsform** der externen Daten einschließlich der Strukturen und Funktionen zur ihrer Verwaltung verstanden (z.B. Ext2, VFAT, NTFS usw).  
Wo zur Unterscheidung notwendig wird hier für diese Bedeutung auch der Begriff **Dateisystem-Typ** verwendet.
  - ▷ Andererseits bezeichnet man damit auch die **konkrete Realisierung** dieser Organisationsform auf einem **bestimmten Datenträger** / einer bestimmten Partition (z.B. Ext2-Dateisystem auf hda2)
- ◇ Bevor unter LINUX ein konkretes Dateisystem eines bestimmten Typs (z.B. eine Festplatten-Partition mit dem Ext2-Dateisystem oder dem VFAT-Dateisystem) verwendet werden kann, muss
  - ▷ das Dateisystem dem Typ nach **registriert** werden
  - ▷ das konkrete Dateisystem **eingebunden** (gemountet, *mounted*) werden
- ◇ Durch die **Registrierung** eines Dateisystem(-Typ)s wird der für die Verwendung konkreter Systeme dieses Typs notwendige Code (die "**Implementierung** des Dateisystems") im Kernel verfügbar gemacht.  
Dies kann erfolgen
  - ▷ zum **Boot-Zeitpunkt** (für alle Dateisysteme, deren Code direkt im Kernel eingebunden ist)
  - ▷ zu einem späteren Zeitpunkt durch **Laden eines Kernel-Modules**, das den entsprechenden Code enthält.

### • Structure-Datentyp **struct file\_system\_type** (definiert in *include/linux/fs.h*)

- ◇ Dieser Datentyp dient zur **Repräsentierung** von **registrierten Dateisystemen** im Kernel.
- ◇ Für jedes registrierte Dateisystem wird ein Objekt dieses Typs angelegt (Filesystem-Typ-Objekt).  
Alle derartigen Objekte sind in einer **einfach verketteten Liste** zusammengefaßt.  
Der Anfang dieser Liste wird durch die in *fs/super.c* definierte statisch-globale Variable `struct file_system_type *file_systems` referiert.
- ◇ Definition (Kernel 2.6.11)

```
struct file_system_type {
    const char* name;                /* Name des Dateisystems */
    int fs_flags;                    /* Registrierungs-Flags */
    struct super_block *(*read_super) /* Funktion zum Einlesen des Super-Blocks */
        (struct file_system_type *, void *, int, const char*, void*);
    void (*kill_sb) (struct super_block*); /* Funktion zum Zerstören eines Super-Blocks */
    struct module *owner;
    struct file_system_type *next;    /* Pointer auf nächstes Listenelement */
    struct list_head fs_supers;
};
```

### • Dateisystem-Registrierung

- ◇ Die Registrierung eines Dateisystems erfolgt mit der in *fs/super.c* definierten Funktion  
`int register_filesystem(struct file_system_type * fs)`
- ◇ Diese Funktion hängt das durch den übergebenen Parameter *fs* referierte Filesystem-Typ-Objekt in die entsprechende Objekt-Liste ein. Sie wird aufgerufen
  - zum **Boot-Zeitpunkt** für alle im Kernel direkt implementierten Dateisysteme
  - beim **Laden eines Moduls** für das durch das Modul implementierte Dateisystem.
- ◇ Beim **Entladen eines Moduls** wird das Dateisystem wieder **deregistriert** (das entsprechende Filesystem-Typ-Objekt aus der Liste ausgetragen durch Aufruf der ebenfalls in *fs/super.c* definierten Funktion  
`int unregister_filesystem(struct file_system_type * fs)`
- ◇ Eine **Liste** der aktuell **registrierten Dateisysteme** ist in der Datei **/proc/filesystems** enthalten

## Mounten von Dateisystemen (1)

### • Allgemeines

- ◇ Das Mounten (Einbinden) eines konkreten Dateisystems ist erst möglich, nachdem der entsprechende Dateisystem-Typ registriert worden ist.
- ◇ Jedes Dateisystem hat sein eigenes Root-Directory.  
Das Dateisystem, dessen Root-Directory zum **Root-Directory des Gesamtsystems** wird, wird als **Root-Dateisystem** (*root filesystem*) bezeichnet.  
Dieses Dateisystem muß zum **Boot-Zeitpunkt gemountet** werden.
- ◇ Andere Dateisysteme können beliebig später gemountet werden.  
Durch das Mounten wird ihr jeweiliges Root-Directory in ein anzugebendes Directory des bis dahin existierenden System-Directory-Baums eingehängt. Dieses Directory wird als **Mount-Punkt** (*mount point*) bezeichnet.
- ◇ Eine **Liste** der aktuell **gemounteten Dateisysteme** (Gerät, Mount-Punkt, Typ) ist in der Datei **/proc/mounts** enthalten.

### • Structure-Datentyp **struct vfsmount** (definiert in *include/linux/mount.h*)

- ◇ Dieser Datentyp dient zur **Verwaltung der gemounteten Dateisysteme**.
- ◇ Pro gemountetem Dateisystem wird eine Variable dieses Typs angelegt.  
Alle Variablen sind in einer doppelt verketteten Liste zusammengefasst → **Liste der gemounteten Dateisysteme** (Mount-Liste).  
Das erste Element dieser Liste wird durch die statisch-globale Variable `struct list_head vfsmntlist` referiert (def. in *fs/namespace.c*)
- ◇ Definition (Kernel 2.6.11) :

```
struct vfsmount
{
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent; /* fs we are mounted on */
    struct dentry *mnt_mountpoint; /* dentry of mountpoint */
    struct dentry *mnt_root; /* root of the mounted tree */
    struct super_block *mnt_sb; /* pointer to superblock */
    struct list_head mnt_mounts; /* list of children, anchored here */
    struct list_head mnt_child; /* and going through their mnt_child */
    atomic_t mnt_count; /* Verwendungszaehler */
    int mnt_flags;
    int mnt_expiry_mark; /* true if marked for expiry */
    char *mnt_devname; /* Name of device e.g. /dev/hda1 */
    struct list_head mnt_list; /* Pointer der Mount-Liste */
    struct list_head mnt_fslink; /* link in fs-specific expiry list */
    struct namespace *mnt_namespace; /* containing namespace */
};
```

## Mounten von Dateisystemen (2)

### • Mounten des Root-Dateisystems

- ◇ Das Mounten des Root-Dateisystems erfolgt während der System-Initialisierung. Hierfür wird im Init-Prozeß die Funktion **mount\_root()** (def. in `fs/super.c`) aufgerufen. Diese
  - ▷ **erzeugt** das **Superblock-Objekt** für das **Root-Dateisystem**  
Hierfür ruft sie nacheinander für die bisher registrierten (in der Filesystem-Typ-Objekt-Liste enthaltenen) Dateisystem-Typen die **allgemeine** Funktion **read\_super()** (def. in `fs/super.c`) auf, bis ein Superblock-Objekt erfolgreich eingelesen werden konnte.  
`read_super()` führt im wesentlichen aus :
    - **Allozierung** eines neuen **Superblock-Objekts** (mittels **alloc\_super()**, def. in `fs/super.c`)
    - Aufruf der Funktion **insert\_super()** (def. in `fs/super.c`) zum Einfügen des Superblock-Objekts in die – zu Beginn noch leere – Superblock-Objekt-Liste.
    - Aufruf der jeweiligen **dateisystemspezifischen** Methode **read\_super()** zum Setzen der Komponenten des Superblock-Objekts.  
Diese Methode ruft bei Erfolg u.a. auch die allgemeine Funktion **d\_alloc\_root()** (def. in `fs/dcache.c`) auf, die ein **Dentry-Objekt** und ein **Inode-Objekt** für das Root-Directory **anlegt**.
    - Bei **Misserfolg** der dateisystemspezifischen Methode **read\_super()** wird das allozierte Superblock-Objekt mittels **remove\_super()** (def. in `fs/super.c`) wieder aus der Liste ausgetragen und **beseitigt**  
Ein Aufruf von `read_super()` ist nur für den Dateisystem-Typ, der mit dem Typ des Root-Dateisystems übereinstimmt, erfolgreich.
  - ▷ **erzeugt** eine **struct vfsmnt -Variable** mittels **alloc\_vfsmnt()** (def. in `fs/namespace.c`), setzt deren Komponenten für das Root-Dateisystem und fügt sie als erstes Element in die Liste der gemounteten Dateisysteme ein (mittels **graft\_tree()**, die ihrerseits **attach\_mnt()** aufruft, beide def. in `fs/namespace.c`)
  - ▷ **setzt** die Komponenten **root** und **pwd** der **struct fs\_struct -Struktur** des **Init-Prozesses** auf das Dentry-Objekt des Root-Directories.

### • Mounten weiterer Dateisysteme

- ◇ Das Mounten weiterer Dateisysteme wird durch den **System Call mount()** bewirkt.
- ◇ Die dadurch aktivierte Betriebssystemfunktion **sys\_mount()** (def. in `fs/namespace.c`) führt zum Aufruf von :
  - ▷ **do\_mount()** (def. in `fs/namespace.c`)
    - ▷ **do\_add\_mount()** (def. in `fs/namespace.c`)
      - ▷ **do\_kern\_mount()** (def. in `fs/super.c`)
        - ▷ **alloc\_vfsmnt()** (def. in `fs/namespace.c`)  
Erzeugung einer `struct vfsmount -Variablen`
        - ▷ **get\_sb\_bdev()** (def. in `fs/super.c`)
          - ▷ **read\_super()** (def. in `fs/super.c`)  
Erzeugung und Einlesen eines neuen Superblock-Objekts,  
sowie eines Dentry- und eines Inode-Objekts für das Root-Dir des gemounteten Dateisystems
      - ▷ **graft\_tree()** (def. in `fs/namespace.c`)  
Einfügen der neuen `struct vfsmount -Variablen` in die Mount-Liste

### • Unmounten eines Dateisystems

- ◇ Das Unmounten eines Dateisystems erfolgt mit dem **System Call umount()**.
- ◇ Die dadurch aktivierte Betriebssystemfunktion **sys\_umount()** (def. in `fs/namespace.c`) führt zum Aufruf von :
  - ▷ **do\_umount()** (def. in `fs/namespace.c`)

## LINUX System Call `mount`

- **Funktionalität :** **Einhängen** (Mounten) eines **Dateisystems** in den System-Directory-Baum.  
Das nach Ort (i.a. *block device special file*) und Typ anzugebende Dateisystem wird in ein ebenfalls anzugebendes Directory (Mount-Directory, Mount-Punkt) eingehängt.  
Zusätzlich können Mount-Optionen spezifiziert werden.  
Nach dem Einhängen ist der Inhalt des Root-Directories des eingehängten Dateisystems über das Mount-Directory zugänglich

- **Interface :**

```
int mount(const char* specfile, const char* dir, const char* type,
          unsigned long flags, const void* data);
```

- **Header-Datei :** `<sys/mount.h>`

- **Parameter :** *specfile* **Ort** des **einzuhängenden Dateisystems**, i.a. Dateipfad eines blockorientierten. Geräts (*block device special file*, z.B. `/dev/hdb3`), bzw. `NULL`-Pointer wenn nicht benötigt (Dateisystem nicht auf Gerät)
  - dir* Zugriffspfad des Directories, in das eingehängt werden soll (**Mount-Directory**)
  - type* Name für den **Typ** des **einzuhängenden Dateisystems**  
Der Name muß dem Linux-Kernel bekannt sein, d.h. zu einem registrierten Dateisystem gehören (z.B. : `"ext2"`, `"vfat"`, `"nfs"`, `"iso9660"` usw).  
Eine Liste der registrierten Namen ist in `/proc/filesystems` enthalten
  - flags* **allgemeine Mountoptionen**, zulässig sind
    - in den 16 höherwertigen Bits der Wert **`0xC0ED`** (*Magic Number*) (**`MS_MGC_VAL == 0xC0ED0000`**)
    - in den 16 niederwertigen Bits :  
eine bitweis oder-verknüpfte Kombination eines oder mehrerer der folgenden Flags (definiert in `sys/mount.h`):
 

|                                   |                                                                        |
|-----------------------------------|------------------------------------------------------------------------|
| <b><code>MS_RDONLY</code></b>     | nur lesender Dateizugriff im gemounteten Dateisystems                  |
| <b><code>MS_NOSUID</code></b>     | <code>suid</code> und <code>sgid</code> -Flags sollen ignoriert werden |
| <b><code>MS_NODEV</code></b>      | Zugriff zu Gerätedateien ist nicht zulässig                            |
| <b><code>MS_NOEXEC</code></b>     | Ausführung von Programmen ist unzulässig                               |
| <b><code>MS_ASYNC</code></b>      | Schreiboperationen sofort auf das Medium ausführen                     |
| <b><code>MS_REMOUNT</code></b>    | Remount des Dateisystems (ggfls mit ander. Optionen)                   |
| <b><code>MS_MANDLOCK</code></b>   | Mandatory File Locks sind zulässig                                     |
| <b><code>MS_NOATIME</code></b>    | Zugriffszeit soll nicht angepaßt werden                                |
| <b><code>MS_NODIRATIME</code></b> | Zugriffszeit bei Directories soll nicht angepaßt werden                |
  - data* **dateisystemspezifische Mount-Optionen**, werden vom jeweiligen Dateisystem interpretiert und ausgewertet.  
bzw. `NULL`-Pointer wenn nicht benötigt

- **Rückgabewert :**
    - `0` bei **Erfolg**
    - `-1` im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. **21**
  - `sys_mount(...)` (in `fs/namespace.c`)
  - `do_mount(...)` (in `fs/namespace.c`)

- **Anmerkungen:** Der System Call darf nur durch einen Prozess, der mit Root-Berechtigung (`UID = 0`) läuft, aufgerufen werden.

## LINUX System Call `umount`

- **Funktionalität :** **Aushängen** (Unmounten) eines gemounteten **Dateisystems** aus dem System-Directory-Baum. Das auszuhängende Dateisystem ist über sein Mount-Directory (Directory, in das es eingehängt worden ist) zu referieren.
- **Interface :**

```
int umount (const char* dir);
```

  - **Header-Datei :** `<sys/mount.h>`
  - **Parameter :** *dir* Zugriffspfad des Directories, in das das auszuhängende Dateisystem gemountet war (Mount-Directory)
  - **Rückgabewert :** - 0 bei **Erfolg**  
- -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **22**
  - `sys_umount(...)` (in `fs/namespace.c`)
  - `do_umount(...)` (in `fs/namespace.c`)
- **Anmerkungen:**
  1. Der System Call darf nur durch einen Prozess, der mit Root-Berechtigung (UID = 0) läuft, aufgerufen werden.
  2. In der ursprünglichen Version von `umount` war als Parameter der Dateipfad des Gerätes, auf dem sich das eingehängte Dateisystem befindet, anzugeben. In Linux 0.98p4 wurde als zweite Möglichkeit die alternative Angabe des Mount-Directories hinzugefügt. Ab Linux 2.3.99-pre7 muß ausschließlich das Mount-Directory angegeben werden. (Da es jetzt zulässig ist, dasselbe Dateisystem in mehr als ein Mount-Directory einzuhängen, reicht die Spezifizierung des Dateisystems über das Gerät nicht aus).

## System Calls zur Externen Datenverwaltung - Überblick

### • Überblick über wesentliche System Calls des VFS

|                     |                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------|
| <b>mount ()</b>     | Einhängen (Mounten) eines Dateisystems in den System-Directory-Baum                                             |
| <b>umount ()</b>    | Aushängen (Unmounten) eines gemounteten Dateisystems                                                            |
| <b>sysfs ()</b>     | Ermittlung von Informationen über einen Dateisystem-Typ                                                         |
| <b>statfs ()</b>    | Ermittlung von Informationen über ein gemountetes Dateisystem (referiert über einen Dateipfad)                  |
| <b>fstatfs ()</b>   | Ermittlung von Informationen über ein gemountetes Dateisystem (referiert über einen File-Deskriptor)            |
| <b>ustat ()</b>     | Ermittlung von Informationen über ein gemountetes Dateisystem (referiert über die Geräte-ID)                    |
| <b>chroot ()</b>    | Änderung des Root-Directories                                                                                   |
| <b>chdir ()</b>     | Änderung des aktuellen (Arbeits-) Directories (neues Directory referiert über Zugriffspfad)                     |
| <b>fchdir ()</b>    | Änderung des aktuellen (Arbeits-) Directories (neues Directory referiert über File-Deskriptor)                  |
| <b>getcwd ()</b>    | Ermittlung des aktuellen (Arbeits-) Directories                                                                 |
| <b>mkdir ()</b>     | Erzeugung eines neuen Directories                                                                               |
| <b>rmdir ()</b>     | Entfernen eines Directories                                                                                     |
| <b>getdents ()</b>  | Einlesen von Directory-Einträgen                                                                                |
| <b>link ()</b>      | Erzeugung eines weiteren Directory-Eintrags für eine existierende Datei (Hard Link)                             |
| <b>unlink ()</b>    | Entfernung eines Directory-Eintrags für eine Datei und wenn es letzter Link war, auch der Datei selbst          |
| <b>rename ()</b>    | Umbenennung eines Directory-Eintrags                                                                            |
| <b>symlink ()</b>   | Erzeugung eines Soft Links für eine existierende Datei                                                          |
| <b>readlink ()</b>  | Einlesen eines symbolischen Links (Ermittlung des Pfades auf den der Link zeigt)                                |
| <b>chown ()</b>     | Änderung des Besitzers und der Gruppe einer Datei (referiert über Zugriffspfad, symb. Links wird gefolgt)       |
| <b>fchown ()</b>    | Änderung des Besitzers und der Gruppe einer Datei (referiert über File-Deskriptor)                              |
| <b>lchown ()</b>    | Änderung des Besitzers und der Gruppe einer Datei (referiert über Zugriffspfad, symb. Links wird nicht gefolgt) |
| <b>chmod ()</b>     | Änderung der Zugriffsberechtigungen für eine Datei (referiert über Zugriffspfad)                                |
| <b>fchmod ()</b>    | Änderung der Zugriffsberechtigungen für eine Datei (referiert über File-Deskriptor)                             |
| <b>utime ()</b>     | Änderung der letzten Zugriffs- und der letzten Änderungszeit einer Datei                                        |
| <b>stat ()</b>      | Ermittlung von Inode-Informationen (referiert über Zugriffspfad, symbolischen Links wird gefolgt)               |
| <b>fstat ()</b>     | Ermittlung von Inode-Informationen (referiert über File-Deskriptor)                                             |
| <b>lstat ()</b>     | Ermittlung von Inode-Informationen (referiert über Zugriffspfad, symbolischen Links wird nicht gefolgt)         |
| <b>access ()</b>    | Überprüfung der Zugriffsrechte des aktuellen Prozesses zu einer Datei                                           |
| <b>umask ()</b>     | Setzen der Default-Dateiattributs-Maske für den aktuellen Prozeß                                                |
| <b>mknod ()</b>     | Erzeugung einer normalen Datei, einer Gerätedatei oder einer Named Pipe                                         |
| <b>open ()</b>      | Öffnen einer existierenden Datei oder Erzeugung einer neuen Datei                                               |
| <b>creat ()</b>     | Erzeugung einer neuen Datei                                                                                     |
| <b>close ()</b>     | Schließen einer geöffneten Datei                                                                                |
| <b>dup ()</b>       | Duplizieren eines File-Deskriptors (Wahl des neuen File-Deskriptors durch den Kernel)                           |
| <b>dup2 ()</b>      | Duplizieren eines File-Deskriptors (Angabe des neuen File-Deskriptors durch Aufrufer)                           |
| <b>fcntl ()</b>     | Manipulation eines File-Deskriptors                                                                             |
| <b>lseek ()</b>     | Setzen der Dateibearbeitungs-Position                                                                           |
| <b>_llseek ()</b>   | Setzen der Dateibearbeitungs-Position mittels eines 64-Bits-Offsets                                             |
| <b>read ()</b>      | Lesen aus einem Datei-Objekt                                                                                    |
| <b>write ()</b>     | Schreiben in ein Datei-Objekt                                                                                   |
| <b>readv ()</b>     | Lesen eines Vektors aus einem Datei-Objekt                                                                      |
| <b>writev ()</b>    | Schreiben eines Vektors in ein Datei-Objekt                                                                     |
| <b>pread ()</b>     | Lesen aus einem Datei-Objekt ab einer bestimmten Position                                                       |
| <b>pwrite ()</b>    | Schreiben in ein Datei-Objekt ab einer bestimmten Position                                                      |
| <b>sendfile ()</b>  | Kopieren von Daten zwischen Datei-Objekten                                                                      |
| <b>truncate ()</b>  | Änderung der Größe einer Datei (referiert über Zugriffspfad)                                                    |
| <b>ftruncate ()</b> | Änderung der Größe einer Datei (referiert über File-Deskriptor)                                                 |
| <b>fsync ()</b>     | Herausschreiben aller Daten-Buffer einer Datei sowie Update des Inodes der Datei                                |
| <b>fdatasync ()</b> | Herausschreiben aller Daten-Buffer einer Datei ohne Update des Inodes der Datei                                 |
| <b>msync ()</b>     | Herausschreiben der Änderungen einer in den Arbeitsspeicher eingeblendeten Datei                                |
| <b>flock ()</b>     | Setzen / Entfernen einer Dateisperre                                                                            |
| <b>select ()</b>    | Warten auf eine Statusänderung von Datei-Objekten                                                               |
| <b>poll ()</b>      | Warten auf ein Ereignis bei Datei-Objekten                                                                      |

## Implementierung von VFS System Calls - Beispiel

### • Implementierung des System Calls `read()`

- ◇ Der System Call `read()` führt zum Aufruf der in `fs/read_write.c` definierten Betriebssystemfunktion `ssize_t sys_read(unsigned int fd, char * buf, size_t count)`
- ◇ Diese Funktion führt im wesentlichen die folgenden Schritte aus :
  - ▷ Ermittlung der **Adresse file** des durch den übergebenen File-Deskriptor `fd` referierten **File-Objekts**. Hierfür ruft sie die in `fs/file_table.c` definierte Funktion `fget()` auf. Die Adresse des File-Objekts ergibt sich zu `file=current->files->fd[fd]`. `fget()` bewirkt weiterhin eine **Erhöhung des Verwendungszählers** des File-Objekts `file->f_count`.
  - ▷ **Überprüfung**, ob die in `file->f_mode` eingetragenen **Zugriffsrechte** einen lesenden Zugriff zulassen
  - ▷ Falls lesender Zugriff zulässig ist, Aufruf der Funktion `locks_verify_area()` (def. in `include/linux/fs.h`) zur **Überprüfung**, ob für den zu lesenden Dateibereich keine **Dateisperre** für lesenden Zugriff besteht.
  - ▷ Falls keine Dateisperre besteht, Aufruf der **dateisystemspezifischen** File-Operation `read()` über `file->f_op->read` zum **Lesen der Daten**. Diese Funktion **erhöht** auch die aktuelle Bearbeitungsposition `file->f_pos` um die Anzahl der gelesenen Bytes. Diese Anzahl wird als Funktionswert zurückgegeben.
  - ▷ Aufruf von `fput()` (def. in `fs/file_table.c`) zur **Erniedrigung des Verwendungszählers** des File-Objekts.
  - ▷ Rückgabe der Anzahl der gelesenen Bytes bzw eines Fehlercodes
- ◇ **Quellcode** der Funktion `sys_read()` (in `fs/read_write.c`)

```
ssize_t sys_read(unsigned int fd, char * buf, size_t count)
{
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        if (file->f_mode & FMODE_READ) {
            ret = locks_verify_area(FLOCK_VERIFY_READ, file->f_dentry->d_inode,
                                   file, file->f_pos, count);

            if (!ret) {
                ssize_t (*read)(struct file *, char *, size_t, loff_t *);
                ret = -EINVAL;
                if (file->f_op && (read = file->f_op->read) != NULL)
                    ret = read(file, buf, count, &file->f_pos);
            }
        }

        /* ... */

        fput(file);
    }
    return ret;
}
```

## LINUX System Call `mkdir`

- **Funktionalität :** Erzeugen eines neuen Directories mit einem anzugebenden Zugriffspfad (Pfadnamen) und anzugebenden Zugriffsrechten.

- **Interface :**

```
int mkdir(const char* pname, mode_t mode);
```

- **Header-Datei :** `<sys/stat.h>`

- **Parameter :**
  - `pname` Pfadname (Zugriffspfad) des neu zu erzeugenden Directories. Falls bereits ein Directory-Eintrag mit diesem Pfadnamen existiert oder eines der Elemente des Pfades kein Directory oder kein Link auf ein Directory ist, endet der System Call fehlerhaft.

`mode` Zugriffsrechte.

Bitweise ODER-Verknüpfung von einer oder mehrerer der folgenden in `<sys/stat.h>` definierten Konstanten :

|                                                |                                                 |   |                      |
|------------------------------------------------|-------------------------------------------------|---|----------------------|
| <code>S_IRUSR</code> ( <code>S_IREAD</code> )  | Besitzer (User) darf Directory lesen            | } | <code>S_IRWXU</code> |
| <code>S_IWUSR</code> ( <code>S_IWRITE</code> ) | Besitzer (User) darf Dir. schreiben             |   |                      |
| <code>S_IXUSR</code> ( <code>S_IEXEC</code> )  | Besitzer (User) darf Dir. traversieren          |   |                      |
| <code>S_IRGRP</code>                           | Gruppe darf Directory lesen                     | } | <code>S_IRWXG</code> |
| <code>S_IWGRP</code>                           | Gruppe darf Directory schreiben                 |   |                      |
| <code>S_IXGRP</code>                           | Gruppe darf Directory traversieren              |   |                      |
| <code>S_IROTH</code>                           | andere Benutzer (Andere) dürfen Directory lesen | } | <code>S_IRWXO</code> |
| <code>S_IWOTH</code>                           | andere Benutzer dürfen Directory schreiben      |   |                      |
| <code>S_IXOTH</code>                           | andere Benutzer dürfen Directory traversieren   |   |                      |
| <code>S_ISUID</code>                           | suid-Bit wird gesetzt                           |   |                      |
| <code>S_ISGID</code>                           | sgid-Bit wird gesetzt                           |   |                      |
| <code>S_ISVTX</code>                           | sticky-Bit wird gesetzt                         |   |                      |

Die durch den Parameter angegebenen Zugriffsrechte werden mit der invertierten Dateikreierungsmaske `umask` des Prozesses bitweise UND-verknüpft.

Die **tatsächlichen Zugriffsrechte** des Directories ergeben sich somit zu :  
**`mode & ~umask`**

- **Rückgabewert :**
  - 0 bei **Erfolg**
  - -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 39
  - `sys_mkdir(...)` (in `fs/namei.c`)
  - `vfs_mkdir(...)` (in `fs/namei.c`)

- **Anmerkungen:**
  1. Der Datentyp `mode_t` ist – indirekt – in der Headerdatei `<sys/stat.h>` definiert als **`unsigned int`**.
  2. Das neue erzeugte Directory bekommt die effektive User-ID des erzeugenden Prozesses.
  3. Falls das Directory, in dem das neu erzeugte Directory eingetragen wird (Eltern-Directory), das `sgid`-Bit gesetzt hat, erbt das neue Directory die Gruppen-ID seines Eltern-Directories. In diesem Fall wird auch sein `sgid`-Bit gesetzt.  
Ist das `sgid`-Bit des Eltern-Directories nicht gesetzt, bekommt das neue Directory die Gruppen-ID des erzeugenden Prozesses.



## LINUX System Call **rmdir**

- **Funktionalität :** **Löschen** eines Directories.

Das durch seinen Pfadnamen referierte Directory muss **leer** sein.

- **Interface :**

```
int rmdir(const char* dir);
```

- **Header-Datei :** **<unistd.h>**
- **Parameter :** *dir* Pfadname (Zugriffspfad) des Directories, das gelöscht werden soll
- **Rückgabewert :**
  - 0 bei **Erfolg**
  - -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. **40**
  - `sys_rmdir(...)` (in `fs/namei.c`)
  - `vfs_rmdir(...)` (in `fs/namei.c`)

- **Anmerkungen:**
  1. Zu dem Eltern-Directory des zu löschenden Directories muß Schreibzugriff bestehen
  2. Wenn für das Eltern-Directory des zu löschenden Directories das `sticky`-Bit gesetzt ist, muss die effektive `UID` des löschenden Prozesses gleich der `UID` des zu löschenden Directories sein, oder der löschende Prozeß muss mit Root-Rechten laufen
  3. Das aktuelle Directory und das Root-Directory eines Prozesses können nicht gelöscht werden.

## LINUX System Call `mknod`

- **Funktionalität :** Erzeugen eines neuen Eintrags (eines "Knotens", *node*) im Gesamt-Dateisystem. Der Eintrag kann eine **Gerätedatei**, ein **Fifo** (*named pipe*) oder eine **normale Datei** sein.

- **Interface :**

```
int mknod(const char* pname, mode_t mode, dev_t dev);
```

- **Header-Datei :** `<sys/stat.h>`  
`<sys/types.h>`

- **Parameter :**
  - `pname` Pfadname (Zugriffspfad) des neu zu erzeugenden Eintrags. Falls bereits ein Directory-Eintrag mit diesem Pfadnamen existiert oder eines der Zwischen-Elemente des Pfades kein Directory oder kein Link auf ein Directory ist, endet der System Call fehlerhaft.
  - `mode` Typ des Eintrags und Zugriffsrechte zum Eintrag. Bitweise ODER-Verknüpfung einer Typangabe und der Zugriffsrechte. Zur **Typangabe** dient eine der folgenden in `<sys/stat.h>` defin. Konstanten :
    - S\_IFCHR** zeichenorientiertes Gerät (*character device special file*)
    - S\_IFBLK** blockorientiertes Gerät (*block device special file*)
    - S\_IFIFO** Fifo (*named pipe*)
    - S\_IFREG** normale Datei (hierfür ist auch der Wert 0 zulässig)Für die **Zugriffsrechte** ist eine bitweise ODER-Verknüpfung von einer oder mehreren der hierfür in `<sys/stat.h>` definierten Konstanten anzugeben. s. System Call `mkdir()` oder `creat()`. Die durch den Parameter angegebenen Zugriffsrechte werden mit der invertierten Dateikreierungsmaske `umask` des Prozesses bitweise UND-verknüpft. Die **tatsächlichen Zugriffsrechte** ergeben sich somit zu : **`mode & ~umask`**
  - `dev` Identifizierung des Gerätes durch Haupt-Nr. (*major number*) und Unter-Nr. (*minor number*). Wird ignoriert, wenn Eintrag keine Gerätedatei ist

- **Rückgabewert :**
  - 0 bei **Erfolg**
  - -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 14
  - `sys_mknod(...)` (in `fs/namei.c`)
  - `vfs_mknod(...)` (in `fs/namei.c`)
  - `vfs_create(...)` (in `fs/namei.c`)

- **Anmerkungen:**
  1. Der Datentyp `mode_t` ist – indirekt – in der Headerdatei `<sys/types.h>` definiert als **unsigned int**.
  2. Der Datentyp `dev_t` ist – indirekt – in der Headerdatei `<sys/types.h>` definiert als **unsigned long long**.
  3. Die Erzeugung von Gerätedateien und normalen Dateien erfordert Root-Rechte
  4. Der neu erzeugte Eintrag bekommt die effektive User-ID des erzeugenden Prozesses.
  5. Falls das Directory, in dem der neu erzeugte Eintrag eingetragen wird (Eltern-Directory), das `sgid`-Bit gesetzt hat, erbt das mit dem neuen Eintrag verknüpfte Objekt die Gruppen-ID seines Eltern-Directories. Ist das `sgid`-Bit des Eltern-Directories nicht gesetzt, bekommt das neue Objekt die Gruppen-ID des erzeugenden Prozesses.
  6. Falls `pname` bereits existiert, wird `errno` auf `EEXIST` gesetzt.

## LINUX System Call `link`

- **Funktionalität :** **Erzeugen** eines neuen **Hard-Links** auf eine existierende Datei.  
Festlegung eines alternativen Pfadnamens, unter dem dieselbe Datei referiert werden kann.  
Alle eine Datei referierenden Pfadnamen sind gleichberechtigt; es ist nicht möglich, festzustellen, welches der Originalname (erster ursprünglicher Pfadname) war

- **Interface :**

```
int link(const char* old, const char* new);
```

- **Header-Datei :** `<unistd.h>`
- **Parameter :**
  - old* existierender Pfadname (Zugriffspfad) für die Datei
  - new* neuer alternativer Pfadname (Zugriffspfad) für die Datei
- **Rückgabewert :**
  - 0 bei **Erfolg**
  - -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. 9
  - `sys_link(...)` (in `fs/namei.c`)
  - `vfs_link(...)` (in `fs/namei.c`)
- **Anmerkungen:**
  1. Hard-Links sind nur innerhalb desselben Dateisystems möglich, d.h. beide Pfadnamen müssen sich auf dasselbe Dateisystem beziehen.
  2. Hard-Links auf Directories sind nicht möglich
  3. Der neue Pfadname darf nicht existieren.
  4. Zu dem Directory, in dem der neue Name eingetragen werden soll, muss Schreibzugriff bestehen

## LINUX System Call **symlink**

- **Funktionalität :** **Erzeugen eines Soft-Links** (symbolischen Links).  
Ein Soft-Link zeigt auf einen anderen Pfadnamen (Zielpfad).  
Die meisten System Calls, die einen Dateizugriff bewirken, folgen dem symbolischen Link, d.h. sie substituieren den Link-Namen durch den Zielpfad.  
Somit wird auch durch einen Soft-Link ein alternativer Pfadname für eine Datei festgelegt.

- **Interface :**

```
int symlink(const char* old, const char* new);
```

- **Header-Datei :** **<unistd.h>**
- **Parameter :**  
*old*     Pfadname auf den der Soft-Link zeigen soll (Zielpfad)  
*new*     Pfadname des Soft-Links (neuer Pfadname, "alternativer" Pfadname für *old*)
- **Rückgabewert :**
  - 0    bei **Erfolg**
  - -1   im **Fehlerfall**, *errno* wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **83**
  - `sys_symlink(...)` (in *fs/namei.c*)
  - `vfs_symlink(...)` (in *fs/namei.c*)
- **Anmerkungen:**
  1. Soft-Links sind auch über Dateisystemgrenzen hinweg möglich, d.h. beide Pfadnamen können sich auf unterschiedliche Dateisysteme beziehen.
  2. Soft-Links können auch auf Directories zeigen.
  3. Der Pfadname des Soft-Links (neuer Pfadname) darf nicht existieren.
  4. Zu dem Directory, in dem der Name des Soft-Links eingetragen werden soll, muss Schreibzugriff bestehen.
  5. Das Dateisystem, in dem der Soft-Link eingetragen werden soll, muss symbolische Links unterstützen.
  6. Der Zielpfad (Parameter *old*) wird nicht auf Vorhandensein überprüft.  
D.h. ein Soft-Link kann auch auf einen nicht-existenten Zielpfad zeigen (→ **dangling link**).
  6. Die folgenden System Calls folgen nicht symbolischen Links sondern beziehen sich direkt auf einen Soft-Link :
    - `lchown()`
    - `lstat()`
    - `readlink()`
    - `rename()`
    - `unlink()`

## LINUX System Call `unlink`

- **Funktionalität :** **Löschen** eines **Namens** (Directory-Eintrags, Hard Links) aus dem Dateisystem.  
Der Name kann eine reguläre Datei, einen Symbolischen Link, ein Gerät (Gerätedatei), einen Fifo (*Named Pipe*) oder einen Socket referieren.  
Wenn es sich um den letzten Hard-Link auf eine Datei handelt und kein Prozess die Datei geöffnet hat, wird die **Datei gelöscht**.  
Handelt es sich um den letzten Hard-Link auf eine Datei, die noch von einem anderen Prozess verwendet wird, wird die Datei erst nach dem Schließen des letzten File Deskriptors, durch den sie referiert wird, gelöscht.  
Wenn durch den Namen ein symbolischer Link referiert wird, wird dieser entfernt.  
Wenn der Name einen Socket, Fifo oder Gerät referiert, wird er gelöscht, auch wenn Prozesse, die das referierte Objekt geöffnet haben, dieses noch weiterverwenden können.
- **Interface :**

```
int unlink(const char* path);
```

  - **Header-Datei :** `<unistd.h>`
  - **Parameter :** `path` Zugriffspfad des zu löschenden Namens
  - **Rückgabewert :**
    - 0 bei **Erfolg**
    - -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. 10
  - `sys_unlink(...)` (in `fs/namei.c`)
  - `vfs_unlink(...)` (in `fs/namei.c`)
- **Anmerkungen:**
  1. Das Löschen eines Directory-Namens ist mit diesem System Call nicht möglich.
  2. Zu dem Directory, in dem der zu löschende Name eingetragen ist, muss Schreibzugriff bestehen.
  3. Wenn für das Directory, in dem der zu löschende Name eingetragen ist, das `sticky-Bit` gesetzt ist, muss die effektive `UID` des löschenden Prozesses gleich der `UID` des mit dem zu löschenden Namen verknüpften Objekts sein, oder der löschende Prozeß muss mit Root-Rechten laufen

## LINUX System Call `dup2`

- **Funktionalität :** **Duplizieren** eines **File-Deskriptors** (Quell-Deskriptor) in einen anderen File-Deskriptor (Ziel-Deskriptor).  
Wenn der Ziel-Deskriptor offen war, wird er zuvor geschlossen.  
Beide File-Deskriptoren referieren anschließend das gleiche File-Objekt.  
Damit verwenden sie dieselbe Bearbeitungsposition, dieselben *File Locks* und dieselben Dateiflags, außer dem *close-on-exec* Flag.  
Sie können alternativ für den Zugriff zum gleichen Datei-Objekt (echte Datei, Pipe, usw) verwendet werden.

- **Interface :**

```
int dup2(int oldfd, int newfd);
```

- **Header-Datei :** `<unistd.h>`
- **Parameter :**
  - `oldfd` Ursprünglicher Deskriptor (Quell-Deskriptor), der in den Ziel-Deskriptor dupliziert werden soll.
  - `newfd` Deskriptor, in den der Quell-Deskriptor dupliziert werden soll (Ziel-Deskriptor)
- **Rückgabewert :**
  - `newfd` bei Erfolg
  - `-1` im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **63**
  - `sys_dup2(...)` (in `fs/fcntl.c`)
  - `dupfd(...)` (in `fs/fcntl.c`)

## LINUX System Calls **stat**, **lstat** und **fstat** (1)

- **Funktionalität :** Ermittlung von (**Inode**-)Informationen über einen **Dateisystem-Eintrag**
  - ◇ **stat** referiert den Dateisystem-Eintrag über einen **Zugriffspfad**.  
Wenn der Zugriffspfad ein symbolischer Link ist, werden die Informationen über den durch den Link referierten Eintrag ermittelt, d.h. dem **Link wird gefolgt**.
  - ◇ **lstat** referiert den Dateisystem-Eintrag ebenfalls über einen **Zugriffspfad**.  
Wenn der Zugriffspfad ein symbolischer **Link** ist, wird diesem **nicht gefolgt**, d.h. es werden die Informationen über den Link selbst ermittelt
  - ◇ **fstat** referiert den Dateisystem-Eintrag über einen **Filedeskriptor**.

Alle drei System Calls legen die ermittelten Informationen in einer durch den Aufrufer bereitzustellenden Variablen des Typs **struct stat** ab.

- **Interface :**

```
int stat(const char* path, struct stat* buf);  
int lstat(const char* path, struct stat* buf);  
int fstat(int fd, struct stat* buf);
```

- **Header-Dateien :** **<sys/types.h>**  
**<sys/stat.h>**  
**<unistd.h>**
- **Parameter :**
  - path* Zugriffspfad des zu untersuchenden Dateisystem-Eintrags
  - fd* Filedeskriptor des zu untersuchenden Dateisystem-Eintrags
  - buf* Pointer auf Variable in der die ermittelten Informationen abzulegen sind
- **Rückgabewert :**
  - **0** bei **Erfolg**
  - **-1** im **Fehlerfall**, *errno* wird entsprechend gesetzt

- **Implementierung :**
  - stat** System Call Nr. **106**
    - `sys_newstat(...)` (in `fs/stat.c`)
    - `vfs_stat(...)` (in `fs/stat.c`)
  - lstat** System Call Nr. **107**
    - `sys_newlstat(...)` (in `fs/stat.c`)
    - `vfs_lstat(...)` (in `fs/stat.c`)
  - fstat** System Call Nr. **108**
    - `sys_newfstat(...)` (in `fs/stat.c`)
    - `vfs_fstat(...)` (in `fs/stat.c`)

## LINUX System Calls **stat**, **lstat** und **fstat** (2)

### • Anmerkungen :

1. Der Datentyp **struct stat** fasst die von den System Calls **stat**, **lstat** und **fstat** gelieferten Informationen über einen Dateisystem-Eintrag zusammen.

Er ist in der von `<sys/stat.h>` eingebundenen Headerdatei `<bits/stat.h>` wie folgt definiert :

#### **struct stat**

```
{ dev_t      st_dev;      // Geräte-Nr des Geräts, auf dem sich der Eintrag befindet
  ino_t      st_ino;      // Inode-Nr. (im virtuellen Dateisystem)
  mode_t     st_mode;     // File Mode (Dateityp und Zugriffsrechte)
  nlink_t    st_nlink;    // Link Count
  uid_t      st_uid;      // User-ID des Dateibesitzers
  gid_t      st_gid;      // Group-ID des Dateibesitzers
  dev_t      st_rdev;     // eigene Geräte-Nr., wenn Eintrag ein Gerät ist (sonst 0)
  unsigned short __pad2;
  off_t      st_size;     // Dateigrösse (0 bei Geräten)
  unsigned long st_blksize; // bevorzugte (optimale) Blockgrösse für effiziente I/O
  unsigned long st_blocks; // Anzahl der allozierten 512-Bytes-Blöcke
  time_t     st_atime;    // Zeit des letzten Zugriffs (time of last access)
  time_t     st_mtime;    // Zeit der letzten Inhalts-Änderung (time of last modification)
  time_t     st_ctime;    // Zeit der letzten Inode-Änderung (time of last change)
};
```

2. Die für die Structure-Komponenten verwendeten Typen sind – indirekt – definiert in `<sys/types.h>` :

```
- dev_t      als unsigned long long
- ino_t      als unsigned long
- mode_t     als unsigned int
- nlink_t    als unsigned int
- uid_t      als unsigned int
- gid_t      als unsigned int
- off_t      als long
- time_t     als long
```

3. Die **Geräte-Nr.** (Typ **dev\_t**) fasst die **Major-Nr.** und die **Minor-Nr.** eines Gerätes zusammen.

Die beiden Bestandteile können mittels der folgenden in `<sys/sysmacros.h>` definierten **Makros** ermittelt werden (Parameter muss die Geräte-Nr. sein) :

▷ **major(dev\_nr)** liefert die **Major-Nr.**

▷ **minor(dev\_nr)** liefert die **Minor-Nr.**

4. Die Codierung des **Dateityps** und der **Zugriffsrechte** im **File Mode** (Typ **mode\_t**) entspricht der Codierung im Inode des Ext2/Ext3-Dateisystems.

Mit den folgenden – in `sys/stat.h` definierten – **Makros** kann der **Dateityp** überprüft werden (Parameter muss der File Mode sein) :

```
▷ S_ISREG(mode) liefert true, wenn normale Datei
▷ S_ISDIR(mode) liefert true, wenn Directory
▷ S_ISCHR(mode) liefert true, wenn Character Device
▷ S_ISBLK(mode) liefert true, wenn Block Device
▷ S_ISFIFO(mode) liefert true, wenn Named Pipe
▷ S_ISLNK(mode) liefert true, wenn symbolischer Link
▷ S_ISSOCK(mode) liefert true, wenn Socket
```



## LINUX System Call `getdents`

- **Funktionalität :** Einlesen von Directory-Einträgen.

Das einzulesende Directory muss zum Lesen geöffnet sein und durch seinen Filedeskriptor referiert werden.

Beim ersten Aufruf nach dem Öffnen des Directories werden die ersten Directoryeinträge eingelesen. Es werden soviel Einträge gelesen, wie in dem beim Aufruf anzugebenden Ablagebuffer vollständig Platz haben.

Bei jedem weiteren Aufruf werden – entsprechend dem Platz im Ablagebuffer – die jeweils nächsten Einträge eingelesen.

- **Interface :**

```
int getdents(unsigned int fd, struct dirent* dirp, unsigned int count);
```

- **Header-Dateien :** `<unistd.h>`

`<linux/types.h>`  
`<linux/dirent.h>`  
`<linux/unistd.h>`

In der C-Bibliothek existiert keine spezifische *Wrapper*-Funktion für diesen System Call .

Zweckmässigerweise wird er über die generische *Wrapper*-Funktion `syscall()` aufgerufen :

```
syscall(__NR_getdents, fd, dirp, count);
```

- **Parameter :** `fd` Filedeskriptor des einzulesenden Directories

`dirp` Pointer auf einen Speicherbereich, in dem die eingelesenen Directory-Einträge abgelegt werden (Ablagebuffer).

Directory-Einträge werden durch Structures des folgenden Typs beschrieben (definiert in `<linux/dirent.h>`):

```
struct dirent
{ long d_ino;                /* inode number */
  off_t d_off;               /* offset from dir-start to next dirent */
  unsigned short d_reclen;   /* length of this dirent */
  char d_name [NAME_MAX+1]; /* file name (null-terminated) */
};
```

`count` Größe des Speicherbereichs in Bytes

- **Rückgabewert :** - Anzahl der gelesenen Bytes bei Erfolg  
- 0, wenn das Directory-Ende erreicht ist  
- -1 im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 141

→ `sys_getdents(...)` (in `fs/readdir.c`)  
→ `vfs_readdir(...)` (in `fs/readdir.c`)

- **Anmerkung :** 1. Die Konstante **NAME\_MAX** legt die maximale Länge eines Dateinamens fest.  
Ihr Wert ist in `<linux/limits.h>` definiert zu 255

- 2. Für die Anwendung in User-Programmen sind in der C-Bibliothek eine Reihe POSIX-kompatibler Directory-Bearbeitungsfunktionen implementiert (dekl. in `<dirent.h>` ), u.a. :

```
DIR* opendir(const char* path); /* DIR ist ein in der Bibliothek */
struct dirent* readdir(DIR* dirp); /* intern verwendeter Datentyp */
int closedir(DIR* dirp); /* zur Beschreibung von */
void rewinddir(DIR* dirp); /* "Directory-Streams" */
```

## Demonstrationsbeispiel zum LINUX System Call `getdents`

```
// -----  
// C-Quell-Datei listdir_m.c  
// Demonstrationsbeispiel zum SystemCall getdents  
// Auflisten eines Directories  
// -----  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <errno.h>  
  
#define __USE_GNU  
  
#include <fcntl.h>  
#include <unistd.h>  
#include <linux/types.h>  
#include <linux/dirent.h>  
#include <linux/unistd.h>  
  
#define DEF_DIR "../.../.../..."  
  
int getdents(unsigned fd, struct dirent* dirp, unsigned count)  
{ syscall(__NR_getdents, fd, dirp, count) };  
  
int main(int argc, char *argv[])  
{  
    int iRet = 0;  
    char* dirpfad = DEF_DIR;  
    int fd;  
    struct dirent * pEntry = malloc(sizeof(struct dirent));  
    struct dirent* entryp = pEntry;  
    printf("\nGroesse struct dirent : %d\n", sizeof(struct dirent));  
    printf("Adr. d_ino      : %p\n", &entryp->d_ino);  
    printf("Adr. d_off       : %p\n", &entryp->d_off);  
    printf("Adr. d_reclen    : %p\n", &entryp->d_reclen);  
    printf("Adr. d_name      : %p\n", &entryp->d_name);  
    printf("Groesse d_name  : %d\n", sizeof(entryp->d_name));  
    char* endp=NULL;  
  
    if (argc>1)  
        dirpfad=argv[1];  
    fd = open(dirpfad, O_RDONLY);  
    printf("\nfd      : %d  errno : %d\n", fd, errno);  
    while ((iRet=getdents(fd, entryp, sizeof(struct dirent)))>0)  
    { printf("\ngelesene Bytes : %d", iRet);  
      endp=(char*)entryp+iRet;  
      while ((char*)entryp < endp)  
      { printf("\n%s", entryp->d_name);  
        entryp=(struct dirent*)((char*)entryp+entryp->d_reclen);  
      }  
      entryp = pEntry;  
    }  
    printf("\n\nerrno : %d", errno);  
    printf("\nfertig\n");  
    close(fd);  
    free(pEntry);  
  
    // nur zum Test des SystemCalls symlink  
    iRet = symlink("../", "ParDir");  
    printf("\niRet : %d  errno : %d\n", iRet, errno);  
    //unlink("ParDir");  
  
    return iRet;  
}
```

# **Betriebssysteme**

## **Kapitel 10**

### **10. Interprozesskommunikation in LINUX**

10.1. Überblick

10.2. Signale

10.3. Pipes

10.4. Fifos (Named Pipes)

10.5. System V IPC

## Interprozesskommunikation in LINUX (Überblick)

- **Überblick über die Möglichkeiten zur Interprozesskommunikation**

LINUX stellt alle wesentlichen Möglichkeiten zur Interprozesskommunikation (IPC), die in unixartigen Systemen üblich sind, zur Verfügung.

Die wichtigsten dieser Möglichkeiten sind :

- **Signale**

Information eines Prozesses oder einer Prozessgruppe durch den Kernel oder durch einen anderen Prozess über ein asynchron eingetretenes Ereignis.

- **Pipes**

Unidirektionale Kommunikation (Senden und Empfangen von Bytefolgen) zwischen verwandten Prozessen (Eltern- und Kindprozesse, "Geschwister"-Prozesse).

Der "Nachrichtenkanal" besteht i.a. aus einem hierfür vom Kernel bereitgestellten Arbeitsspeicherbereich.

Eine Pipe existiert maximal solange wie die an der Kommunikation beteiligten Prozesse.

- **Named Pipes (FIFOs)**

Unidirektionale Kommunikation zwischen beliebigen Prozessen.

Eine Named Pipe besitzt einen Eintrag im Dateisystem. Sie kann auch außerhalb der an einer Kommunikation beteiligten Prozesse existieren.

Auch die durch eine Named Pipe übertragenen Daten werden über einen vom Kernel bereitgestellten Arbeitsspeicherbereich übertragen und nicht über eine tatsächliche Ablage in einer Datei auf einem Datenträger.

- **System V IPC :**

IPC-Mechanismen, die ursprünglich durch UNIX System V eingeführt worden sind.

Sie ermöglichen eine Kommunikation zwischen beliebigen Prozessen.

Sie werden durch geeignete vom Kernel im Arbeitsspeicher angelegte und verwaltete Objekte (Datenstrukturen) realisiert.

Ein angelegtes Objekt existiert solange, bis es explizit gelöscht wird.

- ◆ **Semaphore**

Gegenseitige Synchronisation von Prozessen

- ◆ **Message Queues**

Asynchrone Übermittlung von Nachrichten (Bytefolgen) zwischen Prozessen

- ◆ **Shared Memory**

Zugriff zu gemeinsam nutzbaren Speicherbereichen durch mehrere Prozesse.

Derartige Speicherbereiche sind in den virtuellen Speicher aller beteiligten Prozesse eingeblendet.

- **Sockets**

Das Socket-API ermöglicht eine bidirektionale Kommunikation zwischen beliebigen Prozessen.

Einsatz in erster Linie zur Kommunikation im Netzwerk. Unterstützung unterschiedlicher Netzwerkprotokolle.

Auch für die Kommunikation zwischen lokalen Prozessen verwendbar (→ UNIX-Domain-Sockets).

Im letzteren Fall besitzen Sockets einen Eintrag im Dateisystem.

## Signale in LINUX (1)

### • Allgemeines

- ◇ Signale sind eine der **ältesten IPC-Methoden** in UNIX-Systemen.  
Sie werden benutzt, um einen oder mehrere Prozesse über den Eintritt eines **asynchronen Ereignisses** zu informieren.  
Sie können vom **Kernel** oder einem **anderen Prozess** ausgesandt werden.  
Für eine bidirektionale Kommunikation sind Signale i.a. nicht geeignet.
- ◇ Dem **Empfänger-Prozess** stehen mehrere Möglichkeiten zur **Reaktion** auf ein Signal zur Verfügung :
  - ▷ Er kann das empfangene Signal **ignorieren** (außer Signale `SIGKILL` und `SIGSTOP`).
  - ▷ Er kann mit einer vom Kernel auszuführenden **Default-Aktion** reagieren.
  - ▷ Er kann mit der Ausführung eines eigenen von ihm installierten **Signal-Handlers** reagieren (**Abfangen** des Signals, "*catching the signal*") (außer Signale `SIGKILL` und `SIGSTOP`).
  - ▷ Er kann den Empfang von Signalen **blockieren** (außer Signale `SIGKILL` und `SIGSTOP`).  
Blockierte Signale gehen nicht verloren, sondern werden vom Kernel später – nach Aufhebung der Blockade – zugestellt.
- ◇ Ein Prozess kann **nur dann** ein Signal an einen **anderen Prozess senden**,
  - ▷ wenn er mit **Root-Berechtigung** läuft (`EUID==0`).
  - ▷ wenn seine **UID** oder **EUID** **gleich der UID** oder **SUID** des Empfängerprozesses ist.
  - ▷ wenn er das Signal `SIGCONT` an einen Prozeß, der derselben Session angehört, sendet.
- ◇ Der LINUX-Kernel unterscheidet zwei verschiedene **Signal-Kategorien** :
  - ▷ **Klassische UNIX-Signale** (einfache Signale, "*Nonrealtime*"-Signale)
    - Ursprünglich enthielt ein derartiges Signal keine Zusatz-Info, auch nicht über den Sender des Signals.  
In neueren LINUX-Versionen steht jedoch bei diesen Signalen die ursprünglich nur für verlässliche Signale vorgesehene Zusatz-Info ebenfalls zur Verfügung
    - Ein Prozess kann nicht unterscheiden, ob ihm das gleiche Signal nur einmal oder mehrmals hintereinander gesandt wurde → Signale können verloren gehen.
    - Die verschiedenen Signale (außer `SIGUSR1` und `SIGUSR2`) haben eine **feststehende Bedeutung**, die jeweils einer bestimmten Erzeugungs-Situation zugeordnet ist.
  - ▷ **Verlässliche Signale** (nach POSIX.4-Standard, "*Realtime*"-Signale)
    - Signal enthält Zusatz-Info, u.a. auch über den Sender des Signals.
    - Mehrere hintereinander zugesandte gleiche Signale werden in eine Warteschlange gestellt, so daß kein Signal verloren gehen kann.
    - Die verschiedenen Signale haben eine **prozess-konfigurierbare Bedeutung**.

### • Signal-Nummern und Signalnamen

- ◇ Signale werden durch positive ganze Zahlen, den **Signal-Nummern**, repräsentiert.  
Die **klassischen UNIX-Signale** umfassen den Bereich : **1 . . . 31**.  
Die **verlässlichen Signale** umfassen den Bereich **ab 32**.  
Die höchste Signal-Nummer ist architekturabhängig. Bei der x86-Architektur beträgt sie **64**.
- ◇ Zur einfacheren Handhabung sind für die **klassischen UNIX-Signale** symbolische Namen (→ **Signalnamen**) definiert.  
(für die C-Bibliothek in der durch `<signal.h>` eingebundenen Headerdatei `<bits/signal.h>`, für den LINUX-Quellcode in der Datei `include/asm/signal.h`)  
Die Namen beginnen alle mit **SIG. . .**.  
Für die kleinste Signalnummer der **verlässlichen Signale** ist der symbolische Name **SIGRTMIN** und für die größte Signalnummer der Name **SIGRTMAX** definiert.

## Signale in LINUX (2)

### • Default-Aktionen für Signale

Für jedes Signal ist eine der folgenden vier möglichen Default-Aktionen festgelegt :

- **Ignorieren** des Signals
- **Anhalten** des Prozesses (Prozess-Zustand → `TASK_STOPPED`, der Prozess existiert weiter und kann später wieder mit dem Signal `SIGCONT` in den Zustand `TASK_RUNNING` versetzt werden)
- **Beenden** des Prozesses
- **Beenden** des Prozesses mit **Core Dump** (Beenden m. CD)

### • Überblick über die klassischen UNIX-Signale

| Signal-Nr | Signalname | Bedeutung                                                                      | Default-Aktion     |
|-----------|------------|--------------------------------------------------------------------------------|--------------------|
| 1         | SIGHUP     | Verbindung zum steuernden Terminal ist geschlossen (Hangup)                    | Beendigung         |
| 2         | SIGINT     | User hat Interrupt-Character (^C) gesendet                                     | Beendigung         |
| 3         | SIGQUIT    | User hat Quit-Character (^\) gesendet                                          | Beendigung m. CD   |
| 4         | SIGILL     | Prozess versuchte illegalen Maschinenbefehl auszuführen                        | Beendigung m. CD   |
| 5         | SIGTRAP    | Prozess ist auf Breakpoint gelaufen                                            | Beendigung m. CD   |
| 6         | SIGABRT    | von <code>abort()</code> Funktion erzeugt                                      | Beendigung m. CD   |
| 7         | SIGBUS     | Prozess hat Hardware-Fehler erzeugt                                            | Beendigung         |
| 8         | SIGFPE     | Prozess hat arithmetischen Fehler (z.B. Überlauf, Div. d. 0) erzeugt           | Beendigung m. CD   |
| 9         | SIGKILL    | Nicht abfangbare Programmbeendigung                                            | Beendigung         |
| 10        | SIGUSR1    | Bedeutung durch Prozess frei wählbar                                           | Beendigung         |
| 11        | SIGSEGV    | Unerlaubter Speicherzugriff                                                    | Beendigung m. CD   |
| 12        | SIGUSR2    | Bedeutung durch Prozess frei wählbar                                           | Beendigung         |
| 13        | SIGPIPE    | Prozess hat in Pipe geschrieben, die keine Leseprozesse hat                    | Beendigung         |
| 14        | SIGALRM    | Ablauf eines vom Prozess mit <code>alarm()</code> gesetzten Timers             | Beendigung         |
| 15        | SIGTERM    | Abfangbare Aufforderung zur Programmbeendigung                                 | Beendigung         |
| 16        | SIGSTKFLT  | Prozess hat Stackfehler verursacht                                             | Beendigung         |
| 17        | SIGCHLD    | Information über Beendigung bzw Anhalten eines Kindprozesses                   | Ignorieren         |
| 18        | SIGCONT    | Fortsetzen eines angehaltenen Prozesses                                        | Fortsetzen/Ignor.  |
| 19        | SIGSTOP    | Nichtabfangbares Anhalten des Prozesses                                        | Anhalten           |
| 20        | SIGSTP     | User hat Suspend-Character gesendet (^Z)                                       | Anhalten o. Ignor. |
| 21        | SIGTTIN    | Leseversuch vom steuernden Terminal durch Hintergrundprozess                   | Anhalten o. Ignor. |
| 22        | SIGTTOU    | Schreibversuch auf steuerndes Terminal durch Hintergrundprozess                | Anhalten o. Ignor. |
| 23        | SIGURG     | Auftritt einer "dringenden Bedingung" an einem Socket                          | Beendigung         |
| 24        | SIGXCPU    | Prozess hat seine CPU-Ressourcen-Grenze überschritten                          | Beendigung         |
| 25        | SIGXFSZ    | Prozess hat seine Filegrößen-Ressourcen-Grenze überschritten                   | Beendigung         |
| 26        | SIGVTALARM | Ablauf eines vom Prozess mit <code>setitimer()</code> gesetzten Timers         | Beendigung         |
| 27        | SIGPROF    | Ablauf des " <i>profile</i> "-Timers                                           | Beendigung         |
| 28        | SIGWINCH   | Änderung der Größe des Terminal-Fensters                                       | Ignorieren         |
| 29        | SIGIO      | Auftritt eines asynchronen I/O-Ereignisses (alt. Name : <code>SIGPOLL</code> ) | Beendigung         |
| 30        | SIGPWR     | System hat Fehler in der Stromversorgung entdeckt                              | Beendigung         |
| 31        | SIGSYS     | falscher System Call (alternativer Name : <code>SIGUNUSED</code> )             | Beendigung         |

### • Anmerkungen

- Die Signale `SIGSTOP` und `SIGKILL` sowie die Signale Nr. 32 und 33 können weder blockiert, noch ignoriert, noch abgefangen werden.
- Das Signal `SIGCONT` führt immer zum Aktivieren eines gestoppten Prozesses (`TASK_STOPPED` → `TASK_RUNNING`) .  
**Zusätzlich** kann für den aktivierten Prozess ein **Signal-Handler** installiert werden.
- Ein explizites Ignorieren von `SIGCHLD` ist nicht möglich. In diesem Fall wird vom Kernel `wait4()` aufgerufen.
- Default-Aktion für alle Signale mit Signal-Nr > 31 ist "Beendigung"

## Datenstrukturen zur Signalbearbeitung in LINUX (1)

- **Datentyp `sigset_t`** (definiert in `include/asm-i386/signal.h` bzw. für Anwender in `<signal.h>`)

Dieser Datentyp dient zur Darstellung von **Signalmengen**.

Jeder Signal-Nummer entspricht ein Bit : Signal-Nummer 1 ist das niederwertigste Bit zugeordnet usw.

Ein Signal ist in der Menge enthalten, wenn das entsprechende Bit gesetzt ist.

Ursprünglich war der Datentyp als `unsigned long` definiert → Begrenzung auf 32 Signale.

Jetzt ist `sigset_t` ein Structure-Typ mit einem Array von `unsigned long` als einzige Komponente.

Die Anzahl der Array-Elemente ist architekturabhängig. Für die x86-Architektur beträgt diese Anzahl 2. Damit können **64 Signale** erfaßt werden (in `include/asm-i386/signal.h` definierte Konstante `_NSIG`).

**Anmerkung :** Für den **Anwender-Code** ist `_NSIG` in `<bits/signum.h>` (eingebunden durch `<signal.h>`) zu **65** (`== SIGRTMAX + 1`) definiert

```
#define _NSIG          64
#define _NSIG_BPW      32
#define _NSIG_WORDS    (_NSIG / _NSIG_BPW)

typedef struct {
    unsigned long sig[_NSIG_WORDS];
} sigset_t;
```

- **Funktionen zur Manipulation von Signalmengen**

POSIX definiert die folgenden 5 Funktionen zur Manipulation von Signalmengen.

Sie sind in der Headerdatei `<signal.h>` deklariert.

- ◇ **Entfernen aller Signale** aus einer durch **set** referierten Signalmenge

```
int sigemptyset(sigset_t* set);
```

Funktionswert :    **0**    bei Erfolg  
                  **-1**   im Fehlerfall

- ◇ **Aufnahme aller** verfügbaren **Signale** in eine durch **set** referierte Signalmenge

```
int sigfillset(sigset_t* set);
```

Funktionswert :    **0**    bei Erfolg  
                  **-1**   im Fehlerfall

- ◇ **Hinzufügen des Signals** mit der Nummer **signum** in eine durch **set** referierte Signalmenge

```
int sigaddset(sigset_t* set, int signum);
```

Funktionswert :    **0**    bei Erfolg  
                  **-1**   im Fehlerfall

- ◇ **Entfernen des Signals** mit der Nummer **signum** aus einer durch **set** referierten Signalmenge

```
int sigdelset(sigset_t* set, int signum);
```

Funktionswert :    **0**    bei Erfolg  
                  **-1**   im Fehlerfall

- ◇ **Überprüfung**, ob das **Signal** mit der Nummer **signum** in der durch **set** referierten Signalmenge **enthalten** ist

```
int sigismember(sigset_t* set, int signum);
```

Funktionswert :    **1**    wenn enthalten  
                  **0**    wenn nicht enthalten  
                  **-1**   im Fehlerfall

## Datenstrukturen zur Signalbearbeitung in LINUX (2)

- **Structure-Typ struct sigaction**

(definiert in `include/asm-i386/signal.h` bzw. für Anwender – indirekt – in `<signal.h>`)

Datentyp zur Repräsentation der Reaktion eines Prozesses auf den Empfang eines bestimmten Signals

```
/* Type of a signal handler. */
typedef void (*__sighandler_t)(int);

#define SIG_DFL    ((__sighandler_t)0)    /* default signal handling */
#define SIG_IGN    ((__sighandler_t)1)    /* ignore signal */

struct sigaction {
    union {
        __sighandler_t _sa_handler;
        void (*_sa_sigaction)(int, struct siginfo *, void *);
    } _u;
    sigset_t sa_mask;
    unsigned long sa_flags;
    void (*sa_restorer)(void);
};

#define sa_handler    _u._sa_handler
#define sa_sigaction  _u._sa_sigaction
```

### Bedeutung der Structure-Komponenten :

- sa\_handler** : Startadresse des auszuführenden Signal-Handlers (Standard, Parameter = Signal-Nummer))  
Spezielle Werte : **SIG\_DFL** Default-Aktion des Kernels soll ausgeführt werden  
**SIG\_IGN** Das Signal soll ignoriert werden
- sa\_sigaction** : alternativ zu **sa\_handler** (falls in **sa\_flags** das Flag **SA\_SIGINFO** gesetzt ist) :  
Startadressen des auszuführenden Signals-Handlers (mit erweiterter Parameterliste)
- sa\_mask** : Menge von Signalen, die während der Ausführung des Signal-Handlers zusätzlich blockiert sein sollen
- sa\_flags** : Flags zur Beeinflussung des Verhaltens des das Signal bearbeitenden Prozesses  
(s. System Call `sigaction`)
- sa\_restorer** : reserviert, kann ignoriert werden (in POSIX nicht definiert)

- **Structure-Typ struct sighand\_struct** (definiert in `include/linux/sched.h`)

Dieser Typ fasst die Aktionen, die von einem Prozess für die verschiedenen Signale auszuführen sind, zusammen.

```
struct sighand_struct {
    atomic_t      count;
    struct k_sigaction action[_NSIG];
    spinlock_t    siglock;
};
```

### Bedeutung der Structure-Komponenten :

- action** : Array von Datenstrukturen, die die Reaktionen eines Prozesses auf den Empfang der verschiedenen Signale beschreiben.  
Der für diese Komponente verwendete Datentyp `struct k_sigaction` kapselt den Datentyp `struct sigaction` (Definition in: `include/asm-i386/signal.h`) :
- ```
struct k_sigaction {
    struct sigaction sa;
};
```
- Die Konstante `_NSIG` legt die Anzahl der insgesamt verfügbaren Signale fest.  
Sie ist in `include/asm-i386/signal.h` zu 64 definiert.
- count** : Anzahl der Prozesse (Threads !), die sich die Structure-Variable teilen
- siglock** : Locking-Variable, stellt exklusiven Zugriff bei SMP-Systemen sicher



### Datenstrukturen zur Signalbearbeitung in LINUX (3)

- **Structure-Typ struct siginfo (siginfo\_t)**

(definiert in `include/asm-generic/siginfo.h` bzw. für Anwender. – indirekt in – in `<signal.h>`)

Dieser Typ dient zur Aufnahme von Signal-Zusatz-Informationen.

Ursprünglich war er für verlässliche Signale ("*Realtime*"-Signale) vorgesehen. In Linux wird er auch für die klassischen UNIX-Signale eingesetzt.

Da er auch die Signal-Nummer enthält, kann er als Typ zur Beschreibung von Signalen aufgefasst werden.

```
#define SI_MAX_SIZE      128
#define SI_PAD_SIZE      ((SI_MAX_SIZE/sizeof(int)) - 3)

typedef struct siginfo {
    int      si_signo;      /* Signal number */
    int      si_errno;      /* An errno value */
    int      si_code;       /* Signal code */

    union {
        int  _pad[SI_PAD_SIZE];

        /* kill() */
        struct {
            pid_t _pid;      /* sender's pid */
            uid_t _uid;      /* sender's uid */
        } _kill;

        /* POSIX.1b signals */
        struct {
            pid_t _pid;      /* sender's pid */
            uid_t _uid;      /* sender's uid */
            sigval_t _sigval;
        } _rt;

        /* weitere Union-Komponenten unterschiedlicher Zusammensetzung
           für einige spezielle Signal-Nummern */

    } _sifields;
} siginfo_t;

/* How these fields are to be accessed. */

#define si_pid      _sifields._kill._pid
#define si_uid      _sifields._kill._uid

#define si_value     _sifields._rt._sigval
```

#### Bedeutung einiger Structure-Komponenten :

- |                 |  |
|-----------------|--|
| <b>si_signo</b> | : Signal-Nummer  |
| <b>si_errno</b> | : Eine mit dem Signal verknüpfte Fehlernummer (errno-Wert des sendenden Prozesses zum Zeitpunkt des Sendens des Signals)                                       |
| <b>si_code</b>  | : Codierung der Art des Absenders des Signals,<br>z.B. SI_USER      Signal von User-Prozess erzeugt (mit kill())<br>SI_KERNEL    Signal ist vom Kernel erzeugt |
| <b>si_pid</b>   | : PID des sendenden Prozesses (0 für Kernel)   |
| <b>si_uid</b>   | : (Real) UID des sendenden Prozesses (0 für Kernel)  |

## Datenstrukturen zur Signalbearbeitung in LINUX (4)

- **Structure-Typ `struct sigqueue`** (definiert in `include/linux/signal.h`)

Dieser Typ definiert ein Element einer verketteten Liste von Signalen (die jeweils durch eine `siginfo_t`-Komponente beschrieben werden). Ursprünglich war diese Liste nur für "*Realtime*"-Signale vorgesehen. Da in Linux der Typ `siginfo_t` aber auch für klassische UNIX-Signale eingesetzt wird, werden auch derartige Signale in die Liste aufgenommen. Aber im Unterschied zu den "*Realtime*"-Signalen wird bei aufeinanderfolgenden klassischen UNIX-Signalen der gleichen Nummer nur ein Listenelement angelegt.

```
struct sigqueue
{
    struct list_head list;
    spinlock_t *lock;
    int flags;
    siginfo_t info;
    struct user_struct *user;
};
```

### Bedeutung einiger Structure-Komponenten :

- ▷ **list** : Zusammenfassung der Verkettungs-Pointer auf nächstes und vorheriges Listenelement
- ▷ **info** : Beschreibung des Signals

- **Structure-Typ `struct sigpending`** (definiert in `include/linux/signal.h`)

Dieser Typ fasst eine verkettete Liste von Signalen sowie eine Signalmenge zusammen. Er wird im Prozessdeskriptor zur Beschreibung der anstehenden Signale eingesetzt.

```
struct sigpending
{
    struct list_head list;
    sigset_t signal;
};
```

### Bedeutung der Structure-Komponenten :

- ▷ **list** : Listenkopf mit Pointer auf das erste und letzte Listenelement
- ▷ **signal** : Signalmenge

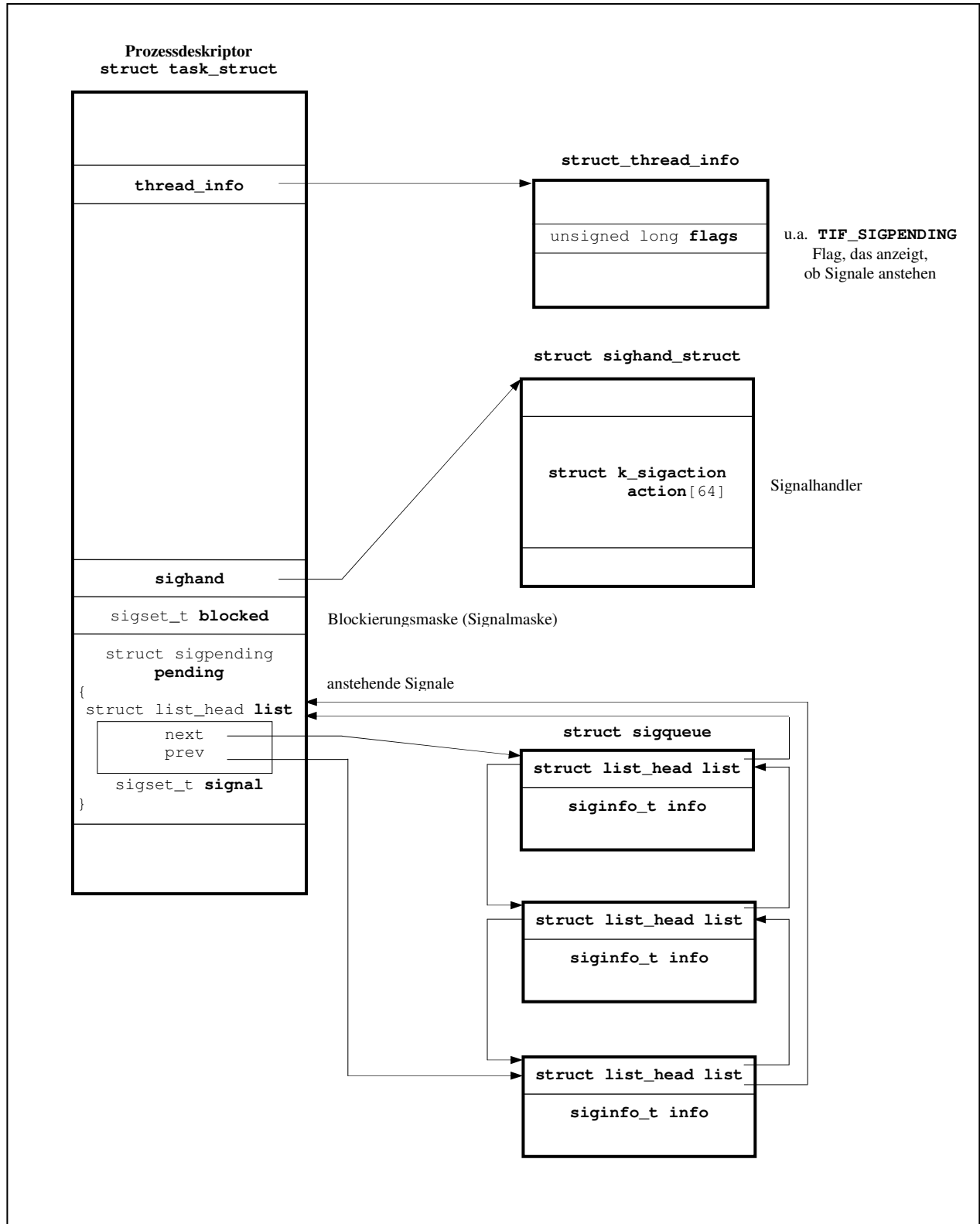
- **TIF\_SIGPENDING-Flag in der Thread-Informationsstruktur**

Die Komponente **flags** der Thread-Informationsstruktur (Typ `struct thread_info`) eines Prozesses enthält u.a. das Flag

- ▷ **TIF\_SIGPENDING** : ein **gesetztes Flag** zeigt an, dass an den Prozess **nicht blockierte Signale** gesandt wurden und auf Bearbeitung warten

## Datenstrukturen zur Signalbearbeitung in LINUX (5)

- Überblick über die signalbezogenen Komponenten der Prozessverwaltungsstruktur



## LINUX System Call `kill`

- **Funktionalität :** Senden eines Signals an einen Prozess bzw eine Gruppe von Prozessen.

- **Interface :**

```
int kill(pid_t pid, int sig);
```

- **Header-Datei :** `<signal.h>`

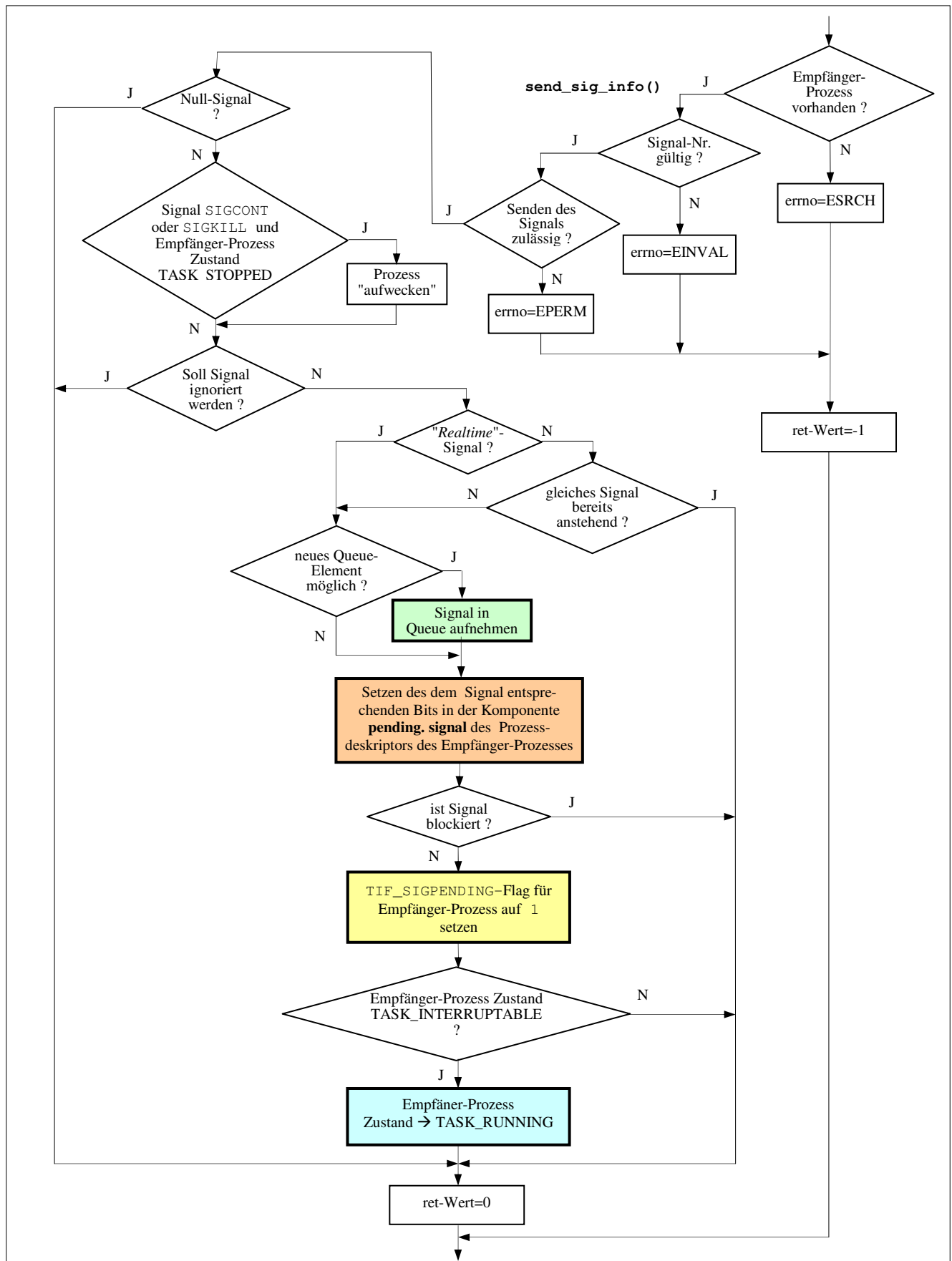
- **Parameter :**
  - `pid` legt den Empfängerprozess bzw die Empfängerprozesse des Signals fest :
    - `>0` Das Signal wird an den Prozess mit der PID `pid` gesendet.
    - `0` Das Signal wird an alle Prozesse in der Prozessgruppe des aktuellen (sendenden) Prozesses geschickt.
    - `-1` Das Signal wird an alle Prozesse außer den Init-Prozess (PID==1) gesendet (Reihenfolge : von höherer PID zu niedrigerer PID)
    - `<-1` Das Signal wird an alle Prozesse der Prozessgruppe mit der PGID==`-pid` geschickt
  - `sig` Nummer des zu sendenden Signals
- **Rückgabewert :**
  - `0` bei Erfolg
  - `-1` im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 37
  - `sys_kill(...)` (in `kernel/signal.c`)
  - `kill_something_info(...)` (in `kernel/signal.c`)
  - `kill_pg_info(...)` (in `kernel/signal.c`)
  - bzw
  - `kill_proc_info(...)` (in `kernel/signal.c`)
  - `send_sig_info(...)` (in `kernel/signal.c`)

- **Anmerkungen:**
  1. Der Datentyp `pid_t` ist in der Header-Datei `<signal.h>` definiert als `int`.
  2. Das Senden eines Signals ist nur zulässig,
    - wenn der aufrufende (sendende) Prozess die `EUID==0` hat (Root-Berechtigung)
    - wenn die `UID` oder `EUID` des aufrufenden Prozesses gleich der `UID` oder `SUID` des Empfängerprozesses ist.
    - wenn das Signal `SIGCONT` an einen Prozess gesendet wird, der der gleichen Session wie der aufrufende Prozess angehört
  3. Wird `kill()` mit `sig==0` aufgerufen, so wird **kein Signal gesandt**, sondern lediglich **geprüft**, ob das **Senden** eines Signals an den Empfängerprozess **möglich** ist.  
→ Prüfung auf Fehler erfolgt.  
Meist wird dies verwendet, um festzustellen, ob ein bestimmter **Prozess** noch **existiert**.  
Falls ja, → Rückgabewert `=0`  
Falls nein → Rückgabewert `=-1` und `errno=ESRCH`
  4. An den Init-Prozess (PID==1) können nur solche Signale geschickt werden, für die dieser einen Signal-Handler installiert hat (alle anderen werden ignoriert)

## Senden von Signalen in LINUX

### • Prinzipielle Funktionalität von `kill()` (vereinfacht):



## LINUX-Demonstrations-Programm zum Senden von Signalen

```
/* C-Quelldatei sigsend_m.c
   Demo-Programm sigsend zum Senden von Signalen
   Signal-Nummer und Empfangs-Prozess (PID) koennen
   als Programm-Parameter uebergeben werden,
   andernfalls werden sie erfragt
*/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>

int sendSignal(signr, procid)
{ int iRet = 0;
  if (kill(procid, signr) == 0)
    printf("\nSignal %d an Prozess (PID) %d gesendet\n\n", signr, procid);
  else
  { printf("\nFehler beim Senden des Signals, errno : %d\n\n", errno);
    iRet = 1;
  }
  return iRet;
}

int getIntVal(char* frage)
{ int val = -1;
  printf(frage);
  scanf("%d", &val);
  return val;
}

int main(int argc, char *argv[])
{
  int iRet = 0;
  int sig;
  int proc;
  printf("\nSignal-Sendeprozess, PID : %d\n", getpid());
  switch(argc)
  {
    case 1 : while ( ((sig = getIntVal("\nSignal-Nummer ? "))!=-1)
                    &&((proc = getIntVal("Empfangs-Prozess ? "))!=-1))
      iRet = sendSignal(sig, proc);
      break;
    case 2 : sscanf(argv[1], "%d", &sig);
      proc=getIntVal("\nEmpfangs-Prozess ? ");
      iRet = sendSignal(sig, proc);
      break;
    default: sscanf(argv[1], "%d", &sig);
      sscanf(argv[2], "%d", &proc);
      iRet = sendSignal(sig, proc);
      break;
  }

  return iRet;
}
```

Dieses Programm dient zusammen mit dem Programm **sigrec** (s. LINUX-Demonstrationsprogramm zum Empfang von Signalen, V-BS-A29-00-TH-xx) als **Demonstrationsbeispiel für das Senden und Empfangen von Signalen** .

## Empfang von Signalen in LINUX (1)

### • Überprüfung auf anstehende Signale

An einen **Prozess** gesandte Signale werden von diesem erst dann **empfangen**, d.h. durch Ausführung der festgelegten Bearbeitungsroutine **behandelt**, wenn er – gegebenenfalls nach vorausgegangenem Scheduling – von der Ausführung eines **System Calls** oder der Bedienung eines **Interrupts** zurückkehrt.

Unmittelbar vor der Rückkehr des – ja dann aktuellen – Prozesses in den User Mode findet eine **Überprüfung** seines **TIF\_SIGPENDING-Flags** statt.

Ist dieses Flag gesetzt, stehen für den Prozess nichtblockierte Signale an, die noch nicht bearbeitet worden sind.

In diesem Fall wird zur Einleitung der Bearbeitung dieser Signale die Kernel-Funktion **do\_signal(...)** aufgerufen.

### • Behandlung anstehender Signale durch **do\_signal()**

Dieser – in `arch/i386/kernel/signal.c` definierten – Funktion wird u.a. die Anfangsadresse `regs` der auf dem System Stack des Prozesses abgelegten Registerinhaltsstruktur (Typ `struct pt_regs`, s. Interruptbearbeitung) übergeben.

Diese Struktur enthält den Ausführungs-Kontext des Prozesses im User Mode.

Die Funktion `do_signal()` – genauer die von ihr aufgerufene Funktion `get_signal_to_deliver()` – durchsucht in einer Schleife die Menge der möglichen Signale nach dem jeweils nächsten nichtblockierten anstehenden Signal (→ Signalnummer `signr`).

Abhängig von dem für das Signal in `current->sighand->action[signr-1].sa.sa_handler` eingetragenen **Pointer** auf die **Signalbehandlungsroutine** wird dieses wie folgt behandelt :

#### ◇ **SIG\_IGN** ist eingetragen :

Das Signal wird **ignoriert**, die Suche wird fortgesetzt nach einem weiteren Signal.

Handelt es sich bei dem zu ignorierenden Signal um `SIGCHLD`, wird vor der Weitersuche der Prozess durch Aufruf der Funktion `sys_wait4(...)` (Implementierung des System Calls `wait4()`) dazu veranlasst, beendete – und gegebenenfalls angehaltene – Kindprozesse "einzusammeln". Beendete Kindprozesse werden dadurch "automatisch" aus dem Zustand `TASK_ZOMBIE` und damit aus der Prozessliste entfernt.

#### ◇ **SIG\_DFL** ist eingetragen :

Die **Defaultaktion** des Kernels wird ausgeführt :

- ▷ Die Signale `SIGCONT`, `SIGCHLD` und `SIGWINCH` werden ignoriert (bei `SIGCHLD` wird `sys_wait4()` nicht aufgerufen), die Suche wird fortgesetzt nach einem weiteren Signal.
- ▷ Die Signale `SIGSTP`, `SIGTTIN` und `SIGTTOU` werden ignoriert, wenn die Prozessgruppe des Prozesses "verwaist" ist ( d.h. nicht an ein TTY gebunden ist), die Suche wird fortgesetzt nach einem weiteren Signal.
- ▷ Andernfalls wird bei einem dieser Signale ebenso wie beim Signal `SIGSTOP`
  - der Prozess angehalten (→ Prozesszustand `TASK_STOPPED`),
  - der Elternprozess – sofern von diesem nicht das Flag `SA_NOCLDSTOP` für das Signal `SIGCHLD` gesetzt ist (s. System Call `sigaction`) – über den Zustandswechsel seines Kindprozesses informiert (Senden des Signals `SIGCHLD` mittels der Funktion `notify_parent(...)`, die ihrerseits `send_sig_info(...)` aufruft, beide Funktionen sind in `kernel/signal.c` definiert)
  - und anschliessend der Scheduler aufgerufen.
- ▷ Bei allen anderen Signalen wird der Prozess durch Aufruf von `do_exit()` beendet.  
Bei den Signalen `SIGQUIT`, `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGFPE` und `SIGSEGV` wird vorher ein **Core Dump** durchgeführt (Aufruf von `current->binfmt->core_dump(...)`).

#### ◇ Die Startadresse eines **prozesseigenen Signal-Handlers** ist eingetragen :

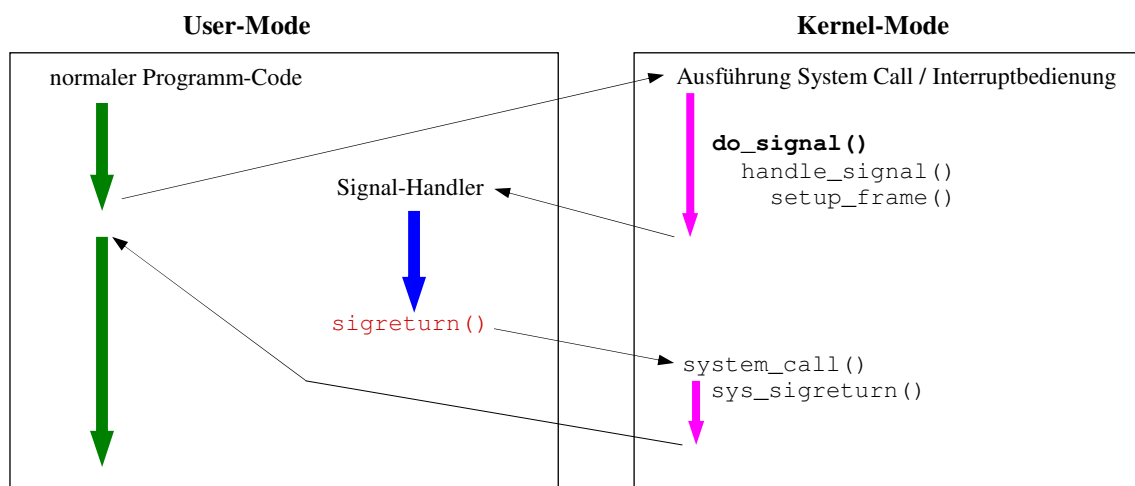
Dieser Signal-Handler muss ausgeführt werden

Hierfür ruft die Funktion `do_signal()` die in `arch/i386/kernel/signal.c` definierte Funktion `handle_signal(...)` auf.

## Empfang von Signalen in LINUX (2)

### • Ausführung eines prozesseigenen Signal-Handlers

- ◇ Zur Ausführung eines installierten prozesseigenen Signal-Handlers wird von `do_signal()` die Funktion **`handle_signal(...)`** aufgerufen.  
Nach der Beendigung von `handle_signal()` wird auch `do_signal()` beendet. Falls weitere Signale für den Prozess anstehen, können diese erst nach einem erneuten Aufruf von `do_signal()` empfangen werden.
- ◇ `handle_signal()` werden u.a. als Parameter übergeben :
  - die Signal-Nummer
  - Pointer auf die dem Signal entsprechende Komponente des Signalreaktions-Beschreibungs-Arrays (Pointer auf Typ `struct k_sigaction`), enthält u.a. die Startadresse des Signal-Handlers
  - Anfangsadresse `regs` der auf dem System Stack des Prozesses abgelegten Registerinhaltsstruktur (= Ausführungs-Kontext im User Mode)
- ◇ Der prozesseigene Signal-Handler kann nicht direkt von `handle_signal()` aufgerufen werden :  
Die Funktion `handle_signal()` läuft im Kernel-Mode, der Signal-Handler dagegen wird im User-Mode ausgeführt und ein direkter Aufruf von User-Mode-Code durch Kernel-Mode-Code ist nicht möglich.
- ◇ Vielmehr **manipuliert** `handle_signal()` mittels der von ihr aufgerufenen Funktion **`setup_frame()`** (bzw **`setup_rt_frame()`**), beide definiert in `arch/i386/kernel/signal.c` die auf dem Kernel-Stack abgelegten Register **EIP** (Instruction Pointer) und **ESP** (Stack Pointer) so, daß nach Rückkehr von `do_signal()` und anschließender Beendigung des System Calls bzw der Interrupt Service Routine der prozesseigene Signal-Handler statt des unterbrochenen Prozess-Codes ausgeführt wird.  
`setup_frame()` (bzw `setup_rt_frame()`) rettet außerdem den im Kernel-Stack abgelegten User-Mode-Ausführungs-Kontext auf dem User-Mode-Stack.  
Darüberhinaus **manipuliert** die Funktion den **User-Mode-Stack** so, dass nach Beendigung des Signal-Handlers der System Call **`sigreturn`** (Nr. 119) ausgeführt wird.
- ◇ Weiterhin führt `handle_signal()` aus :
  - Falls für das Signal das Flag `SA_ONESHOT` gesetzt ist, wird die Startadresse des auszuführenden Signal-Handlers auf `SIG_DFL` gesetzt (→ der prozesseigene Signal-Handler wird nur einmal ausgeführt)
  - Falls für das Signal das Flag `SA_NODEFER` nicht gesetzt ist, wird die Maske der blockierten Signale (Blockierungsmaske) um das aktuelle Signal und die vom Prozess festgelegten weiteren Signale, die während der Bearbeitung des aktuellen Signals zusätzlich blockiert sein sollen, erweitert.
- ◇ Die durch den System Call **`sigreturn`** ausgeführte Betriebssystemfunktion **`sys_sigreturn()`** kopiert den auf dem User-Mode-Stack geretteten User-Mode-Ausführungs-Kontext des ursprünglich unterbrochenen Prozess-Codes wieder auf den Kernel-Mode-Stack zurück und stellt den ursprünglichen Zustand des User-Mode-Stacks wieder her. Außerdem setzt sie die Blockierungsmaske des Prozesses wieder auf den ursprünglichen Wert zurück.  
Nach Beendigung des System Calls `sigreturn` wird dann der ursprüngliche Prozess-Code an der unterbrochenen Stelle fortgesetzt.





## Empfang von Signalen in LINUX (3)

### • Restart von durch Signale unterbrochenen System Calls

- ◇ In einigen Fällen muß in der Ausführung eines System Calls auf den Eintritt eines bestimmten Ereignisses **gewartet** werden (z.B. Warten auf Daten von einem Plattenlaufwerk).  
In einer derartigen Situation wird der betreffende Prozess in den Zustand `TASK_UNINTERRUPTABLE` oder `TASK_INTERRUPTABLE` versetzt.
- ◇ Wird an einen im Zustand **TASK\_INTERRUPTABLE** wartenden Prozess ein Signal geschickt, so wird der Prozess aufgeweckt, d.h. in den Zustand `TASK_RUNNING` überführt, ohne dass das erwartete Ereignis eingetreten ist.  
Da der System Call in diesem Fall nicht erfolgreich beendet werden kann, wird die entsprechende Betriebssystem-Routine mit einem der **Fehler-Codes** `EINTR`, `ERESTARTSYS`, `ERESTARTNOHAND` oder `ERESTARTNOINTR` als Rückgabewert beendet.  
Von diesen Fehler-Codes wird lediglich `EINTR` an den aufrufenden User-Mode-Prozess zurückgegeben (→ Setzen von `errno` auf `EINTR`).  
Die anderen Fehler-Codes werden vom Kernel (genauer von `do_signal()` bzw `handle_signal()`) verwendet, um gegebenenfalls einen **automatischen Restart** des nicht erfolgreich beendeten System Calls zu veranlassen.
- ◇ Die **Behandlung** eines nicht erfolgreich beendeten (durch ein Signal "unterbrochenen") **System Calls** hängt neben dem zurückgegebenen **Fehler-Code** auch von der vom Prozeß für das Signal festgelegten **Reaktionsweise** ab :

Reaktion auf Signal	Fehler-Code			
	<code>EINTR</code>	<code>ERESTARTSYS</code>	<code>ERESTARTNOHAND</code>	<code>ERESTARTNOINTR</code>
<b>Ignorieren</b>	Beenden	Restart	Restart	Restart
<b>Default-Aktion</b>	Beenden	Restart	Restart	Restart
<b>eigener Handler</b>	Beenden	SA_RESTART-abhängig	Beenden	Restart

**Beenden** : Beenden des System Calls mit `errno = EINTR`

**Restart** : automatischer Restart des System Calls

**SA\_RESTART-abhängig** : Wenn `SA_RESTART` gesetzt, automatischer Restart des System Calls, sonst Beenden

- ◇ Zum Zeitpunkt des Signal-Empfangs (unmittelbar vor der Rückkehr in den User-Mode, Aufruf von `do_signal()`) befindet sich der **Rückgabewert** der Betriebssystemfunktion in der Komponente `eax` der auf dem System-Stack **abgelegten Registerinhaltsstruktur**.  
Der in der Komponente `orig_eax` der abgelegten Registerinhalte enthaltene Wert ermöglicht eine **Unterscheidung** zwischen den verschiedenen Ursachen für den vorangegangenen Übergang vom User-Mode in den Kernel-Mode :  
**Interrupt** (negativer Wert < -1), **interne Exception** (-1), **System Call** (positiver Wert = System Call Nummer).  
Diese Unterscheidung ist notwendig, da ein Restart nur für System Calls erfolgen darf
- ◇ Ein automatischer Restart eines System Calls wird dadurch erreicht, dass durch `do_signal()` bzw `handle_signal()` die auf dem Kernel-Stack abgelegten User-Mode-Registerinhalte entsprechend manipuliert werden :
  - Die Komponente `eax` der Registerinhaltsstruktur wird auf die System-Call-Nr gesetzt (diese ist in der Komponente `orig_eax` enthalten)
  - Die Komponente `eip` wird zurückgesetzt, so dass sie auf den Befehl `int 80x` zeigt.
 Nach einer Rückkehr vom System Call in den User-Mode wird somit sofort wieder der System Call ausgeführt.

## Empfang von Signalen in LINUX (4)

- **Restart von durch Signale unterbrochenen System Calls, Forts**

- ◇ Falls das **empfangene Signal** explizit **ignoriert** wird oder falls für dieses Signal die **Default-Aktion** ausgeführt wird, erfolgt die Überprüfung des System Call Rückgabewertes und gegebenenfalls die Initialisierung des System Call Restarts am Ende der Funktion **do\_signal()**

Der entsprechende Code-Ausschnitt lautet :

```
/* . . . */

/* Did we come from a system call? */
if (regs->orig_eax >= 0) {
    /* Restart the system call - no handlers present */
    if (regs->eax == -ERESTARTNOHAND ||
        regs->eax == -ERESTARTSYS ||
        regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
}

/* . . . */
```

- ◇ Falls für das Signal ein **prozesseigener Signal-Handler** installiert ist, erfolgt die Überprüfung und gegebenenfalls Initialisierung des Restarts zu Beginn der Funktion **handle\_signal()**.

Der entsprechende Code-Ausschnitt lautet :

```
/* . . . */

/* Are we from a system call? */
if (regs->orig_eax >= 0) {
    /* If so, check system call restarting.. */
    switch (regs->eax) {
        case -ERESTARTNOHAND:
            regs->eax = -EINTR;
            break;

        case -ERESTARTSYS:
            if (!(ka->sa.sa_flags & SA_RESTART)) {
                regs->eax = -EINTR;
                break;
            }
            /* fallthrough */
        case -ERESTARTNOINTR:
            regs->eax = regs->orig_eax;
            regs->eip -= 2;
    }
}

/* . . . */
```

## LINUX System Call **sigaction** (1)

- **Funktionalität :** **Registrierung** (Installation) und/oder Ermittlung eines **prozesseigenen Signalhandlers** sowie weiterer mit der Signalbearbeitung zusammenhängender Informationen (Signalbehandlungsstruktur)

- **Interface :**

```
int sigaction(int sig, const struct sigaction* act, struct sigaction* oldact);
```

- **Header-Datei :** **<signal.h>**

- **Parameter :**
  - sig* Nummer des Signals, für das ein Handler registriert werden soll, zulässig sind alle gültigen Signalnummern außer SIGKILL und SIGSTOP
  - act* falls !=NULL : Pointer auf Datenstruktur, die die neue Reaktion eines Prozesses auf den Empfang eines Signals der Nummer *sig* festlegt (neue Signalbehandlungsstruktur).

Die relevanten Komponenten des Datentyps struct sigaction sind :

*sa\_handler* (Typ: void(\*) (int)) Startadresse des Signalhandlers  
zulässig sind auch die Werte

- SIG\_IGN → das Signal soll ignoriert werden
- SIG\_DFL → Ausführung der Default-Aktion des Kernels

*sa\_sigaction* (Typ: void(\*) (int, siginfo\_t\*, void\*))  
alternative Startadresse eines Signalhandlers mit erweiterter Parameterliste, zu benutzen, wenn Flag SA\_SIGINFO gesetzt ist

*sa\_mask* (Typ: sigset\_t) Menge von Signalen, die während der Ausführung des Signalhandlers zusätzlich blockiert sein sollen (nur wirksam, wenn das Flag SA\_NOMASK (==SA\_NODEFER) nicht gesetzt ist.)

Das Signal, das zur Ausführung des Signalhandlers geführt hat, wird in einem derartigen Fall ebenfalls blockiert.

*sa\_flags* (Typ: unsigned long) Flags, die das Verhalten des das Signal bearbeitenden Prozesses beeinflussen.

Mögliche Werte :

0 oder die gegebenenfalls bitweise ODER-Verknüpfung von :

- SA\_NOCLDSTOP
- SA\_ONESHOT (== SA\_RESETHAND)
- SA\_RESTART
- SA\_NOMASK (== SA\_NODEFER)
- SA\_SIGINFO

Bedeutung der einzelnen Flags siehe unten

*oldact* falls !=NULL : Pointer auf Datenstruktur, in der die bisher installierte Signalbehandlungsstruktur abgelegt wird (→ Ermittlung des bisher installierten Handlers)

- **Rückgabewert :**
  - 0 bei **Erfolg**
  - -1 im **Fehlerfall**, *errno* wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 67
  - *sys\_sigaction(...)* (in *arch/i386/kernel/signal.c*)
  - *do\_sigaction(...)* (in *kernel/signal.c*)

- **Anmerkung :** Der Datentyp **struct sigaction** ist in der durch **<signal.h>** eingebundenen Headerdatei **<bits/sigaction.h>** definiert.

## LINUX System Call **sigaction** (2)

- **Bedeutung der Flags** (Komponente **sa\_flags** des Parameters **act**) :

<b>SA_NOCLDSTOP</b>	Dieses Flag hat nur für das Signal <code>SIGCHLD</code> eine Bedeutung. Wenn dieses Flag für das Signal <code>SIGCHLD</code> gesetzt ist, wird der Prozess nicht über das Anhalten eines seiner Kindprozesse (mittels eines der Signale <code>SIGSTOP</code> , <code>SIGSTP</code> , <code>SIGTTIN</code> oder <code>SIGTTOU</code> ) informiert, d.h. in diesem Fall wird an den Prozess <b>kein</b> Signal <code>SIGCHLD</code> gesendet.
<b>SA_ONESHOT</b> (== <b>SA_RESETHAND</b> )	Dieses Flag bewirkt, dass der prozesseigene Signalhandler nur <b>einmal</b> aufgerufen wird. Nach seinem Aufruf wird die Default-Reaktion des Kernels wieder hergestellt (Die Funktion <code>handle_signal()</code> setzt die Adresse des auszuführenden Handlers auf <code>SIG_DFL</code> ).
<b>SA_RESTART</b>	Dieses Flag bewirkt, dass System Calls, die durch das Signal unterbrochen worden sind, nach Beendigung des Signalhandlers erneut ausgeführt werden. Defaultmäßig (Flag nicht gesetzt) wird in einem solchen Fall der System Call mit einem Fehler beendet und <code>errno</code> wird auf <code>EINTR</code> gesetzt.
<b>SA_NOMASK</b> (== <b>SA_NODEFER</b> )	Dieses Flag bewirkt, dass während der Abarbeitung des Signalhandlers keine zusätzlichen Signale blockiert werden (auch nicht das die Ausführung des Signalhandlers bewirkende Signal)
<b>SA_SIGINFO</b>	Dieses Flag muß gesetzt werden, wenn ein Signalhandler mit erweiterter Parameterliste installiert wird. In diesem Fall ist die Startadresse des Signalhandlers in der Komponente <code>sa_sigaction</code> (statt der Komponente <code>sa_handler</code> ) des Parameters <code>act</code> abzu- legen.

## LINUX System Call **sigprocmask**

- **Funktionalität :** Veränderung und/oder Ermittlung der Signalmaske (Menge der blockierten Signale) eines Prozesses

- **Interface :**

```
int sigprocmask(int what, const sigset_t* set, sigset_t* oldset);
```

- **Header-Datei :** `<signal.h>`

- **Parameter :**
  - what* Festlegung, wie die Signalmaske mittels *set* zu verändern ist.  
Als mögliche Werte sind in der durch `<signal.h>` eingebundenen Headerdatei `<bits/sigaction.h>` definiert :
    - **SIG\_BLOCK** Die in *set* übergebene Signalmenge ist der aktuellen Signalmaske hinzuzufügen
    - **SIG\_UNBLOCK** Die in *set* übergebene Signalmenge ist aus der aktuellen Signalmaske zu entfernen
    - **SIG\_SETMASK** Die in *set* übergebene Signalmenge wird zur neuen Signalmaske
  - set* falls `!=NULL` : Pointer auf eine Signalmenge, die zur Veränderung der aktuellen Signalmaske dient.
  - oldset* falls `!=NULL` : Pointer auf eine Variable, in der die bisherige Signalmaske abgelegt wird (→ Ermittlung des bisherigen Signalmaske)
- **Rückgabewert :**
  - **0** bei Erfolg
  - **-1** im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** System Call Nr. 126  
→ `sys_sigprocmask(...)` (in `kernel/signal.c`)

- **Anmerkungen :**
  1. Der Datentyp **sigset\_t** ist in der Headerdatei `<signal.h>` definiert.
  2. Signale, die nicht ignoriert werden können, können auch nicht blockiert werden. Der Versuch wird nicht als Fehler angezeigt.

## LINUX System Call **sigpending**

- **Funktionalität :** Ermittlung der aktuell **anstehenden** aber **blockierten Signale**.

- **Interface :**

```
int sigpending(sigset_t* set);
```

- **Header-Datei :** **<signal.h>**
- **Parameter :**     *set*     Pointer auf eine Variable, in der die Menge der anstehenden aber blockierten Signale abgelegt wird.
- **Rückgabewert :**   - **0**     bei **Erfolg**  
                      - **-1**    im **Fehlerfall**, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **73**
  - `sys_sigpending(...)`     (in `kernel/signal.c`)
  - `do_sigpending(...)`     (in `kernel/signal.c`)
- **Anmerkung :**       Der Datentyp **sigset\_t** ist in der Headerdatei `<signal.h>` definiert.

## LINUX System Calls **pause** und **sigsuspend**

### System Call **pause**

- **Funktionalität :** **Warten auf ein Signal.**

Der System Call setzt den Prozesszustand auf `TASK_INTERRUPTIBLE` und ruft anschließend den Scheduler auf.

Durch Senden eines – nicht blockierten – Signals, das nicht ignoriert werden soll, an den Prozess wird dieser wieder in den Zustand `TASK_RUNNING` versetzt.

Falls ein Signalhandler für das empfangene Signal installiert ist, wird dieser nach der erneuten Aktivierung des Prozesses ausgeführt. Anschließend kehrt der System Call zurück.

Falls durch das Signal der Prozess beendet wird, kehrt der System Call nicht zurück.

- **Interface :**

```
int pause(void);
```

- **Header-Datei :** `<unistd.h>`

- **Parameter :** keine

- **Rückgabewert :** - `-1`, `errno` wird auf `EINTR` gesetzt

- **Implementierung :** System Call Nr. **29**

→ `sys_pause(void)` (in `arch/i386/kernel/sys_i386.c`)

### System Call **sigsuspend**

- **Funktionalität :** **Warten auf ein Signal** nach vorheriger - temporärer – Änderung der **Signalmaske**

Der System Call setzt die Signalmaske auf den durch den Parameter bestimmten Wert , überführt anschließend den Prozess in den Zustand `TASK_INTERRUPTIBLE` und ruft danach den Scheduler auf.

Durch Senden eines – nicht blockierten – Signals, das nicht ignoriert werden soll, an den Prozess wird dieser wieder in den Zustand `TASK_RUNNING` versetzt.

Falls ein Signalhandler für das empfangene Signal installiert ist, wird dieser nach der erneuten Aktivierung des Prozesses ausgeführt. Anschließend wird die Signalmaske auf den ursprünglichen Wert wieder zurückgesetzt und der System Call kehrt zurück.

Falls ein zur Prozessbeendigung führendes Signal empfangen wird, kehrt der System Call nicht zurück.

- **Interface :**

```
int sigsuspend(const sigset_t* mask);
```

- **Header-Datei :** `<signal.h>`

- **Parameter :** `mask` Pointer auf die temporär zu setzende Signalmaske

- **Rückgabewert :** - `-1`, `errno` wird auf `EINTR` gesetzt

- **Implementierung :** System Call Nr. **72**

→ `sys_sigsuspend(...)` (in `arch/i386/kernel/signal.c`)

## LINUX-Demonstrations-Programm zum Empfang von Signalen

```
/* C-Quelldatei sigrec_m.c
   Demo-Programm sigrec zum Empfangen von Signalen
   Durch einen Programmparameter wird festgelegt, fuer welches Signal ein Signal-Handler
   installiert werden soll. Ein Aufruf ohne Programmparameterwert fuehrt zum Installieren
   eines (des gleichen) Signal-Handlers fuer alle Signale .
*/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

void mySigHandler(int sig)
{ printf("\nSignal %d empfangen !\n", sig); }

void myExtSigHandler(int sig, struct siginfo* si, void* dummy)
{ printf("\nSignal %d von PID %u empfangen !\n", sig, si->si_pid); }

void infInstError(int sig)
{ fprintf(stderr, "\nHandler fuer Signal %d nicht installiert, errno : %d\n", sig, errno); }

void installSigHandler(int sig, int ext)
{
    sigset_t sigmask;
    struct sigaction sa;
    sigfillset(&sigmask);
    sa.sa_mask = sigmask;
    if (ext==0)
    {
        sa.sa_handler = mySigHandler;
        sa.sa_flags = 0;
    }
    else
    {
        sa.sa_sigaction = myExtSigHandler;
        sa.sa_flags = SA_SIGINFO;
    }
    if (sig>0)
    {
        if (sigaction(sig, &sa, NULL)==-1)
            infInstError(sig);
    }
    else
    {
        int i;
        for (i = 1; i<_NSIG; i++)
            if (sigaction(i, &sa, NULL)==-1)
                infInstError(i);
    }
}

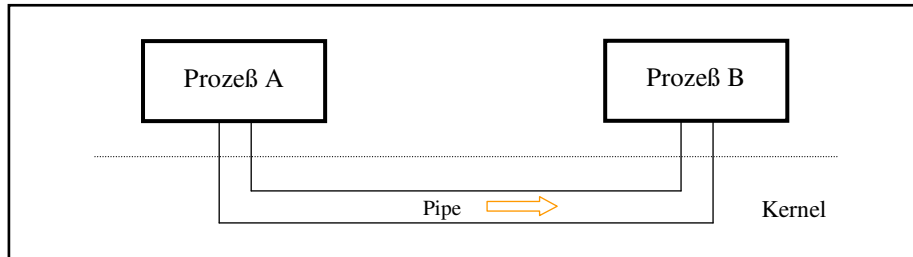
int main(int argc, char *argv[])
{
    int ext = 1; /* legt Typ des Signal-Handlers fest : 0 - einfacher, 1 - erweiterter */
    sigset_t blockmask; /* fuer sigsuspend() */
    sigemptyset(&blockmask);
    if (argc==1)
        installSigHandler(0, ext);
    else
    {
        int sig;
        sscanf(argv[1], "%d", &sig);
        installSigHandler(sig, ext);
    }
    printf("\nIch warte auf Signale !!!\n");
    while(1)
    {
        sigsuspend(&blockmask);
        printf("\nIch warte erneut !!! \n");
    }
    return EXIT_SUCCESS;
}
```



## Unidirektionale Pipes in LINUX (1)

### • Allgemeines

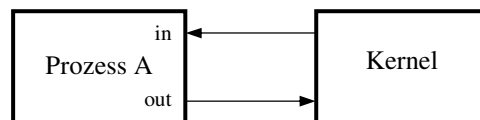
- ◇ Unidirektionale Pipes sind eine der ältesten **IPC-Methoden** in UNIX.  
Sie ermöglichen eine **unidirektionale Kommunikation** zwischen **verwandten Prozessen** (Eltern-/Kind-Prozesse, "Geschwister"-Prozesse).
- ◇ Der "Nachrichtenkanal" besteht i.a. aus einem hierfür vom Kernel bereitgestellten Arbeitsspeicherbereich (**Kommunikations-Buffer**).  
Eine Pipe existiert maximal solange wie die an der Kommunikation beteiligten Prozesse existieren.



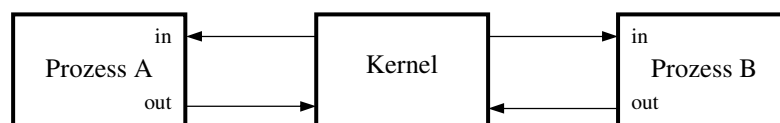
- ◇ Pipes werden häufig eingesetzt, um die Standardausgabe eines Prozesses mit der Standardeingabe eines anderen Prozesses zu verbinden.
- ◇ U.a. stellen alle gängigen UNIX-Shells in der Kommandozeile einen Pipe-Operator zur Verfügung, der dies realisiert.  
Beispiel für dessen Anwendung : `ls | more`

### • Prinzip des Aufbaus einer Pipe-Kommunikation

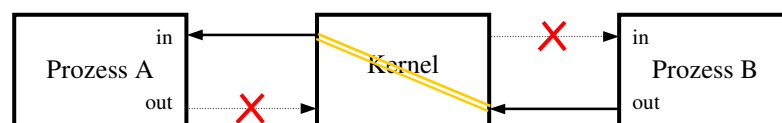
- ◇ Prozess A ruft den **System Call `pipe()`** auf.  
→ der Kernel erzeugt **zwei File-Deskriptoren**, einen für Eingabe, einen für Ausgabe, die eine Kommunikation über den Kernel ermöglichen.



- ◇ Prozess A erzeugt einen **Kindprozess B**. Dieser erbt die File-Deskriptoren der Pipe.



- ◇ Der Elternprozess A und der Kindprozess B **schließen** jeweils einen **File-Deskriptor** für entgegengesetzte Kommunikationsrichtungen.



- ◇ hier : Aufbau einer **Kommunikation vom Kindprozess B zum Elternprozess A**
  - **Schreiben** in Pipe (hier durch Kindprozess) mittels des System Calls **`write()`**
  - **Lesen** aus Pipe (hier durch Elternprozess) mittels des System Calls **`read()`**
  - Der Zugriff zur Pipe ist nur rein **sequentiell** möglich → Positionierung (mittels **`lseek()`**) nicht möglich

## LINUX System Call `pipe`

- **Funktionalität :** **Erzeugen einer Pipe.**  
Öffnen der Schreibseite der Pipe für blockierendes Schreiben und der Leseseite der Pipe für blockierendes Lesen.  
Bereitstellung von zwei File-Deskriptoren (Pipe-Deskriptoren) für den Zugriff zur Lese- bzw Schreibseite der Pipe.

- **Interface :**

```
int pipe(int fd[2]);
```

- **Header-Datei :** `<unistd.h>`
- **Parameter :** `fd` Array, in dem die beiden bereitgestellten File-Deskriptoren zurückgegeben werden.  
In `fd[0]` wird der File-Deskriptor für Lesen (Leseseite der Pipe),  
in `fd[1]` wird der File-Deskriptor für Schreiben (Schreibseite der Pipe) abgelegt.
- **Rückgabewert :** - `0` bei **Erfolg**  
- `-1` im **Fehlerfall**, `errno` wird entsprechend gesetzt
- **Implementierung :** System Call Nr. **42**  
→ `sys_pipe(...)` (in `arch/i386/kernel/sys_i386.c`)  
→ `do_pipe(...)` (in `fs/pipe.c`)

- **Anmerkungen zur Verwendung einer Pipe :**

1. Die Pipe-Deskriptoren können mittels `close()` geschlossen werden
2. Bei – defaultmässig eingestelltem – blockierendem I/O-Verhalten blockiert Lesen aus der Pipe (mittels `read()`) solange, bis mindestens 1 Byte verfügbar ist.
3. Bei – defaultmässig eingestelltem – blockierendem I/O-Verhalten blockiert Schreiben in die Pipe (mittels `write()`) solange, bis Platz für alle zu schreibenden Daten vorhanden ist.
4. Mittels des System Calls `fcntl()` kann für die Lese- und/oder die Schreibseite der Pipe nichtblockierendes I/O-Verhalten (*non-blocking mode*) eingestellt werden (zusätzliches Einbinden von `fcntl.h` erforderlich):  
`fcntl(fd[0], F_SETFL, (long)O_NONBLOCK)` für die Leseseite bzw  
`fcntl(fd[1], F_SETFL, (long)O_NONBLOCK)` für die Schreibseite  
In diesem Fall warten `read()` bzw `write()` nicht, sondern kehren bei nicht-vorhandenen Lesedaten bzw bei nicht-ausreichendem Platz für die Schreibdaten mit dem Fehler `EAGAIN` zurück.
5. Wenn die Schreibseite der Pipe geschlossen wird, liefert `read()` nach dem Lesen des letzten Bytes den Wert `0` → Kennzeichen für `EOF`.
6. Beim Versuch in eine Pipe zu schreiben, deren Leseseite geschlossen wurde, wird das Signal `SIGPIPE` vom Kernel an den schreibenden Prozess geschickt.  
Sowohl beim Ignorieren als auch beim Abfangen des Signals (prozesseigener Signalhandler) kehrt `write()` mit einem Fehler zurück und `errno` wird auf `EPIPE` gesetzt.

### Beispiel zum LINUX System Call pipe

```
/* ----- */
/* Programm pipetest                               */
/* ----- */
/* Testprogramm für unidirektionale Pipes           */
/* Erzeugen einer Pipe                             */
/* Erzeugen eines Kindprozesses                     */
/* Senden eines Strings von Kind zum Elternprozess über Pipe */
/* ----- */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int iRet = EXIT_SUCCESS;
    int fd[2];
    pid_t iChildPID;
    char *szMsg = "Hello Papa !";
    char acBuff[80]={'\0'};

    if (pipe(fd)<0)
    { printf("\nPipe kann nicht erzeugt werden\n");
      iRet = 1;
    }
    else
    { iChildPID=fork();
      switch (iChildPID)
      {
          case -1 : printf("\nKindprozeß kann nicht erzeugt werden\n");
                    iRet = 1;
                    break;
          case 0 : /* im Kindprozess */
                    close(fd[0]); /* Kindprozess schliesst Leseseite der Pipe */
                    printf("\nKindprozeß schreibt in Pipe : %s\n", szMsg);
                    if (write(fd[1], szMsg, strlen(szMsg)+1)<0)
                    {
                        printf("\nSchreiben in Pipe fehlgeschlagen\n");
                        iRet = 1;
                    }
                    close(fd[1]);
                    break;
          default : /* im Elternprozess */
                    close(fd[1]); /* Elternprozess schliesst Schreibseite der Pipe */
                    printf("\nElternprozeß liest aus Pipe : ");
                    if (read(fd[0], acBuff, sizeof(acBuff)-1) >0)
                        printf("%s\n", acBuff);
                    else
                    {
                        printf("\nLesen fehlgeschlagen\n");
                        iRet = 1;
                    }
                    break;
      }
    }
    return iRet;
}

/* ----- */

Ausgabe des Programms :

Kindprozeß schreibt in Pipe : Hello Papa !

Elternprozeß liest aus Pipe : Hello Papa !

*/
```

## Unidirektionale Pipes in LINUX (2)

### • Umleitung von `stdin` bzw `stdout` in eine Pipe

- ◇ Die Umleitung von `stdin` bzw `stdout` in die Lese- bzw Schreibseite einer Pipe wird erreicht durch **Duplizierung** des betreffenden Pipe-Deskriptors in den Deskriptor für `stdin` (`=0`) bzw für `stdout` (`=1`).
- ◇ Zum Duplizieren eines File-Deskriptors dient der System Call `dup2()`.

#### ◇ Beispiel : Kommunikation Elternprozess → Kindprozess über `stdout` bzw `stdin` :

- Im **Elternprozess** : Duplizierung des Pipe-Deskriptors für Schreiben in den Deskriptor für `stdout`  
Schreiben nach `stdout` → **Schreiben in Pipe**
- Im **Kindprozess** : Duplizierung des Pipe-Deskriptors für Lesen in den Deskriptor für `stdin`  
Lesen aus `stdin` → **Lesen aus Pipe**

Der Kindprozess kann anschließend durch ein anderes Programm überlagert werden.

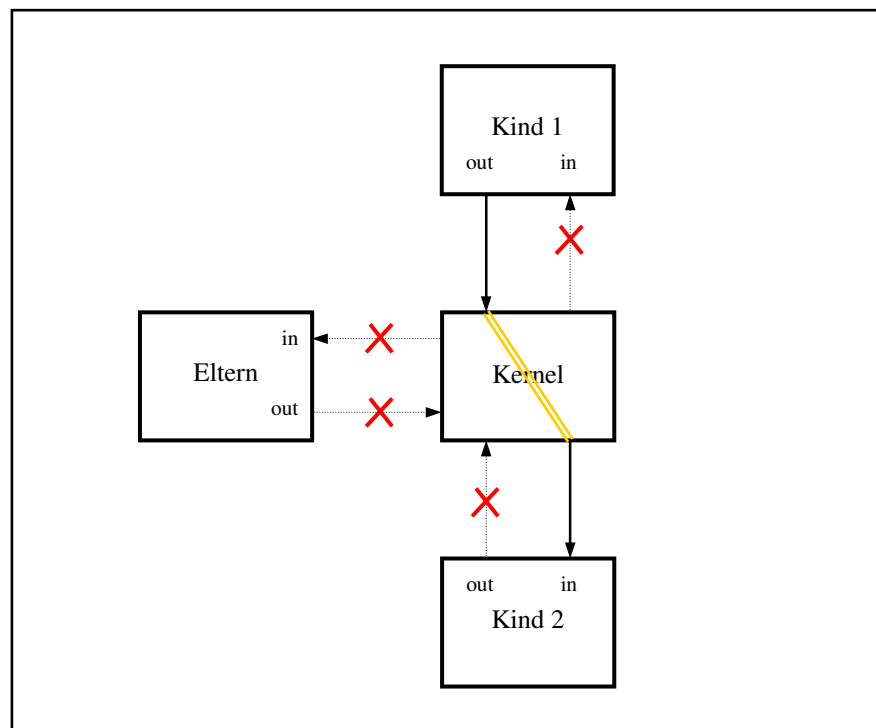
Dieses übernimmt die geöffneten File-Deskriptoren des Prozesses, sofern für sie nicht das *close-on-exec*-Flag gesetzt ist. Für die Standard-File-Deskriptoren ist dieses Flag nicht gesetzt, sie werden also übernommen.

→ Die Eingabe des Programms von `stdin` erfolgt dann über die Pipe.

- ◇ Nach dem Duplizieren können die ursprünglichen Pipe-Deskriptoren geschlossen werden. Sie werden nicht mehr benötigt.

### • Beispiel für eine Pipe zwischen "Geschwister"-Prozessen

- ◇ Kommunikation Kind-Prozess 1 → Kind-Prozess 2



### • Anmerkung

Es ist zulässig, dass **mehrere** Prozesse in eine Pipe **schreiben** und mehrere Prozesse aus der Pipe **lesen**.

Allerdings können in die Pipe geschriebene Daten immer nur von einem Prozess ausgelesen werden.

Das Schreiben erfolgt atomar, sofern die Anzahl der zu schreibenden Daten kleiner gleich der Größe des Kommunikations-Buffers (bei der Intel-Architektur `=4096` Bytes) ist.

### Beispiel zur Umleitung von stdout bzw stdin in eine Pipe

```
/* -----*/
/* Programm pipetest2 */
/* -----*/
/* Demonstrationsprogramm zur Umleitung von stdin bzw stdout in eine Pipe */
/* Erzeugen einer Pipe */
/* Erzeugen eines Kindprozesses */
/* Umleitung der Pipe-Ausgabe nach Stdout beim Elternprozess */
/* Umleitung der Pipe-Eingabe nach Stdin beim Kindprozess */
/* Ausfuehren eines Filterprogramms (tohigh) durch den Kindprozess */
/* Elternprozess liest Zeichen von Stdin und gibt diese ungefiltert nach */
/* Stdout aus */
/* Kindprozess liest ebenfalls von Stdin ein und gibt gefiltert nach Stdout aus*/
/* -----*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN_FD 0 /* Filedescriptor fuer Stdin */
#define STDOUT_FD 1 /* Filedescriptor fuer Stdout */

int main(int argc, char *argv[])
{
    int fd[2];
    pid_t iRetPID;
    int c;
    int stat;
    int iRet = EXIT_SUCCESS;

    if (pipe(fd)<0)
    {
        fprintf(stderr, "\nPipe kann nicht erzeugt werden\n");
        iRet = 1;
    }
    else
    {
        iRetPID=fork(); /* Erzeugung Kindprozess */
        switch(iRetPID)
        {
            case -1 : fprintf(stderr, "\nKindprozess kann nicht erzeugt werden\n");
                    iRet = 1;
                    break;

            case 0 : /* im Kindprozess */
                    close(fd[1]); /* Kindprozess schliesst Schreibseite der Pipe */
                    dup2(fd[0], STDIN_FD); /* Duplizieren fd[0] nach stdin */
                    close(fd[0]); /* fd[0] wird nicht mehr benoetigt */
                    if (execlp("tohigh", "tohigh", NULL)<0)
                    {
                        fprintf(stderr, "\nFehler bei exec von tohigh\n");
                        iRet = 1;
                    }
                    break;

            default : /* im Elternprozess */
                    fprintf(stderr, "\nIm Elternprozess\n");
                    close (fd[0]); /* Elternprozess schliesst Leseseite der Pipe */
                    dup2 (fd[1], STDOUT_FD); /* Duplizieren fd[1] nach stdout */
                    close (fd[1]); /* fd[1] wird nicht mehr benoetigt */
                    setbuf(stdout, NULL); /* Ausgabe nach stdout ungepuffert */
                    while ((c=getchar())!=EOF)
                        putchar(c);
                    close(STDOUT_FD); /* --> fuehrt zu EOF bei stdin des Kindproz. */
                    wait(&stat); /* Warten auf Ende des Kindprozesses */
                    fprintf(stderr, "\nEnde Elternprozess\n");
                    break;
        }
    }
    return iRet;
}
```

## Implementierung von Pipes in LINUX (1)

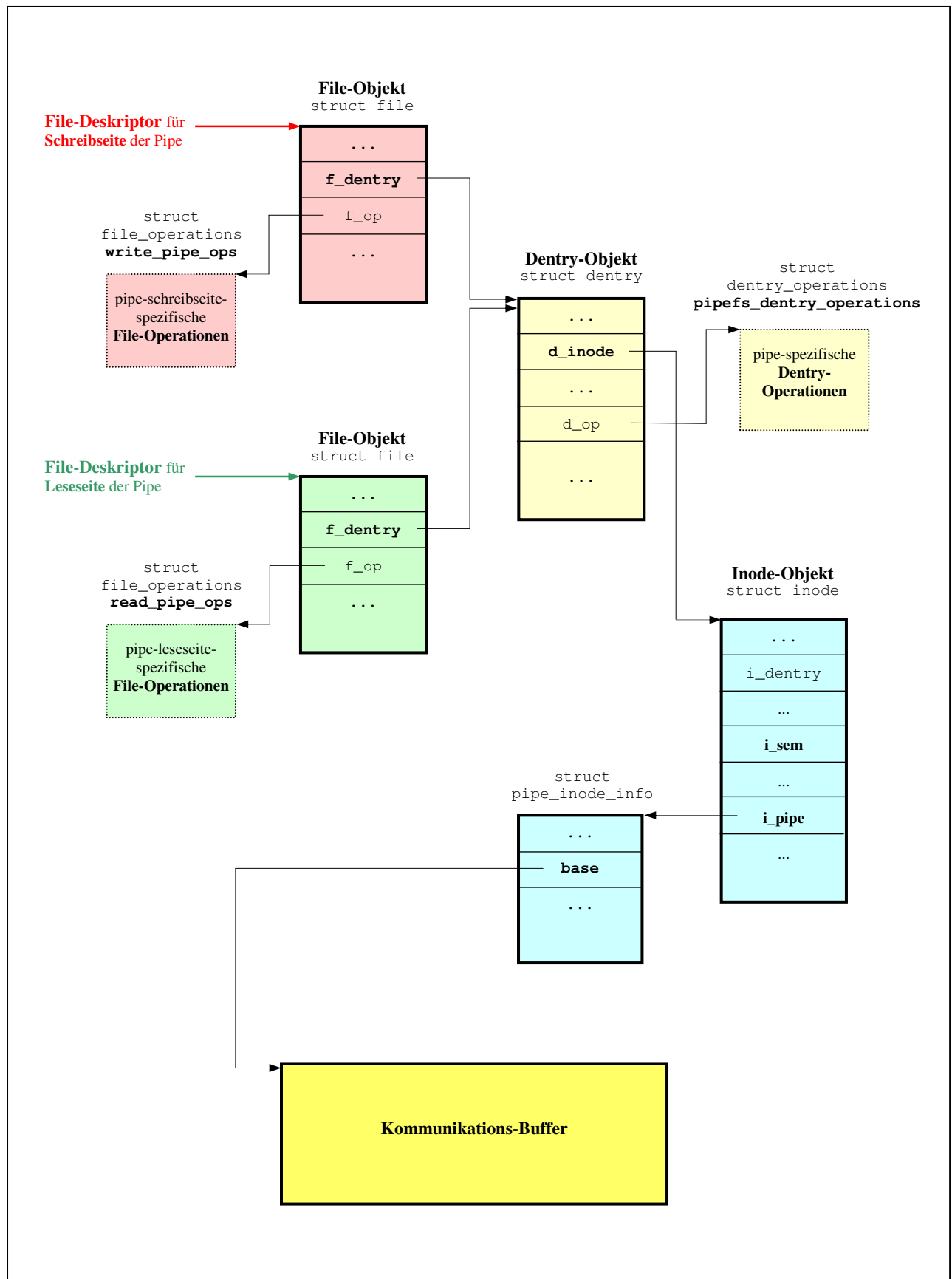
- **Wirkung des System Calls `pipe()`** (s. Funktion `do_pipe()` definiert in `fs/pipe.c`)
  - ◇ Pipes werden in das **virtuelle Dateisystem (VFS)** von Linux integriert. Dies erfordert die Anlage entsprechender VFS-Objekte. Formal werden Pipes einem eigenen "Pipe-Dateisystem" zugeordnet.
  - ◇ Erzeugen eines **Inode-Objekts** für die Pipe. Initialisierung der Komponente `i_pipe` dieses Objekts mit einem Pointer auf eine ebenfalls erzeugte **Pipe-Beschreibungs-Struktur**. Allokation des Speichers für den "Nachrichtenkanal" (**Kommunikations-Puffer**) der Pipe und Ablage dessen Adresse in der Pipe-Beschreibungs-Struktur. Die Größe dieses Puffers ist gleich der Seitengröße des Systems (bei der x86-Architektur 4096 Bytes).
  - ◇ Erzeugung eines das Pipe-Inode-Objekt referierenden **Dentry-Objekts**. Initialisierung der Komponente `d_op` dieses Objekts mit einem Pointer auf die pipe-spezifischen Dentry-Operationen `pipefs_dentry_operations`.
  - ◇ Erzeugung eines **File-Objektes** und eines dieses referierenden File-Deskriptors für die **Leseseite** der Pipe. Setzen der `f_flags`-Komponente des File-Objekts auf `O_RDONLY` und Initialisierung dessen `f_op`-Komponente mit einem Pointer auf die pipe-leseseite-spezifischen File-Operationen `read_pipe_fops`. Initialisierung der Komponente `f_dentry` mit einem Pointer auf das erzeugte Dentry-Objekt für die Pipe.
  - ◇ Erzeugung eines **File-Objektes** und eines dieses referierenden File-Deskriptors für die **Schreibseite** der Pipe. Setzen der `f_flags`-Komponente des File-Objekts auf `O_WRONLY` und Initialisierung dessen `f_op`-Komponente mit einem Pointer auf die pipe-schreibseite-spezifischen File-Operationen `write_pipe_fops`. Initialisierung der Komponente `f_dentry` mit einem Pointer auf das erzeugte Dentry-Objekt für die Pipe.
- **Pipe-Beschreibungs-Struktur `struct pipe_inode_info`**
  - ◇ Dieser Structure-Typ faßt pipe-spezifische Verwaltungsinformationen zusammen. Für jede Pipe wird eine Variable dieses Typs angelegt, die durch eine spezielle Komponente (`i_pipe`) des jeweiligen Pipe-Inodes referiert wird.
  - ◇ Definition (in `linux/pipe_fs_i.h`)

```
struct pipe_inode_info {
    wait_queue_head_t wait;
    char *base;
    unsigned int len;
    unsigned int start;
    unsigned int readers;
    unsigned int writers;
    unsigned int waiting_readers;
    unsigned int waiting_writers;
    unsigned int r_counter;
    unsigned int w_counter;
};
```

**wait** : Warteschlange der auf Schreiben bzw Lesen der Pipe wartenden Prozesse  
**base** : Adresse des **Kommunikations-Puffers**  
**len** : Anzahl der aktuell in der Pipe befindlichen Bytes  
**start** : Offset des **nächsten zu lesenden** Bytes  
**readers** : =1, wenn wenigstens 1 lesender Prozess existiert, sonst =0  
**writers** : =1, wenn wenigstens 1 schreibender Prozess existiert, sonst =0  
**waiting\_readers** : Anzahl der wartenden Lese-Prozesse  
**waiting\_writers** : Anzahl der wartenden Schreib-Prozesse  
**r\_counter** : für Fifos (Named Pipes) benötigt  
**w\_counter** : für Fifos (Named Pipes) benötigt
  - ◇ Der Offset des nächsten zu **schreibenden Bytes** ergibt sich zu :  $(start + len) \& (PIPE\_SIZE-1)$ . Dabei ist `PIPE_SIZE` die Größe des Kommunikations-Puffers in Bytes (== Seitengröße).
  - ◇ Die **Synchronisation des Zugriffs** zur Pipe (sowohl Lesen/Schreiben, als auch mehrere Leseprozesse bzw mehrere Schreibprozesse) erfolgt mittels des im Inode-Objekt vorhandenen **Semaphors** (Komponente `i_sem`).

## Implementierung von Pipes in LINUX (2)

### • Überblick



## FIFOs (Named Pipes) in Linux (1)

### • Eigenschaften

- ◇ **FIFOs** (Named Pipes) bilden einen ähnlichen Interprozess-Kommunikationsmechanismus wie Pipes.  
FIFO = *first in, first out*
- ◇ **Wesentlicher Unterschied** : FIFOs sind mit einem "Datei"-Namen im **Dateisystem** verankert.  
→ Sie belegen auf einem Plattenlaufwerk einen Directory-Eintrag und einen Inode.  
Allerdings existieren für sie keine Datenblöcke.
- ◇ Damit können FIFOs **unabhängig von den Prozessen**, die sie verwenden, **existieren**.  
Da prinzipiell jeder Prozess zu einem angelegten FIFO – wie zu einer Datei – zugreifen kann (unter Berücksichtigung der Zugriffsrechte), ermöglichen FIFOs eine **IPC zwischen beliebigen Prozessen**.  
Typische Anwendung : Kommunikation zwischen Client- und Serverprozessen
- ◇ Der Datenaustausch findet nicht über Plattenbereiche sondern wie bei "einfachen" Pipes über einen im Kernel angelegten **Kommunikations-Buffer** statt.
- ◇ Zu demselben FIFO können mehrere Prozesse schreibend und mehrere Prozesse lesend zugreifen.
- ◇ Zu einem FIFO kann **nur dann zugegriffen** werden, wenn er **sowohl für Lesen als auch für Schreiben geöffnet** worden ist.
- ◇ Der **Zugriff** zu einem FIFO erfolgt mittels der System Calls für die Dateibearbeitung (`open()`, `write()`, `read()`, `close()`). Dabei hängen die jeweils ausgeführten dateisystemspezifischen Bearbeitungsfunktionen nicht von dem Dateisystem ab, in dem der FIFO eingetragen ist. Vielmehr handelt es sich bei diesen – analog zu den "normalen" Pipes – um fifo-spezifische Funktionen, die großenteils mit den pipe-spezifischen Bearbeitungsfunktionen identisch sind.

### • Implementierung

- ◇ Die Implementierung von FIFOs entspricht weitgehend der Implementierung von "normalen" Pipes.  
Wie diese werden auch FIFOs – unter Verwendung der gleichen Objekte und Datenstrukturen – in das Virtuelle Dateisystem (VFS) von LINUX integriert.  
Für einen von einem Schreib- und einem Leseprozess geöffneten FIFO werden vom Kernel angelegt :
  - ▷ ein **Inode-Objekt**  
mit einem Pointer auf eine spezielle **Pipe-Beschreibungs-Struktur** (vom Typ `struct pipe_inode_info`)
  - ▷ ein **Dentry-Objekt**, dessen Komponente `d_name` den Dateipfad des FIFO-Eintrags im Dateisystem enthält (Unterschied zu "normalen" Pipes)
  - ▷ ein **File-Objekt** für die **Schreibseite** des FIFOs, sowie ein dieses referierender **File-Deskriptor**  
Die Komponente `f_op` zeigt auf die fifo-schreibeseite-spezifischen File-Operationen `write_fifo_fops` (definiert in `fs/pipe.c`).  
Bis auf eine (für `poll`) stimmen diese mit den entsprechenden pipe-spezifischen Funktionen überein.
  - ▷ ein **File-Objekt** für die **Leseseite** des FIFOs, sowie ein dieses referierender **File-Deskriptor**  
Die Komponente `f_op` zeigt auf die fifo-leseseite-spezifischen File-Operationen `read_fifo_fops` (definiert in `fs/pipe.c`).  
Bis auf eine (für `poll`) stimmen diese mit den entsprechenden pipe-spezifischen Funktionen überein.
- ◇ Der **wesentliche Unterschied** zwischen FIFOs und "normalen" Pipes liegt in ihrer Erzeugung, ihrer Existenz und dem Zeitpunkt ihres Öffnens :
  - ▷ Für "normale" Pipes wird gleichzeitig mit ihrer Erzeugung durch den System Call `pipe()` sowohl ihre Schreib- als auch ihre Leseseite geöffnet.  
Sie existieren nur durch die im Kernel angelegten Objekte.
  - ▷ FIFOs werden durch den System Call `mknod()` angelegt. Ihre Schreib- und Leseseite müssen getrennt davon jeweils durch den System Call `open()` geöffnet werden.  
Durch ihren Eintrag im Dateisystem existieren sie auch außerhalb des Kernels.



## FIFOs (Named Pipes) in Linux (2)

### • Erzeugung von FIFOs

- ◇ FIFOs werden mittels des System Calls `mknod()` erzeugt :

`mknod(<FIFO-Pfadname>, S_IFIFO | <Zugriffsrechte>, 0);`

Hierdurch wird im Dateisystem ein durch <FIFO-Pfadname> festgelegter Directory-Eintrag für eine "Datei" vom Typ "Named Pipe" angelegt.

- ◇ Alternativ kann zur Erzeugung die in der C-Bibliothek implementierte Funktion `mkfifo()` eingesetzt werden :

`mkfifo(<FIFO-Pfadname>, <Zugriffsrechte>);`

Die Funktion ruft `mknod()` mit den obigen Parametern auf.

In POSIX ist diese Funktion direkt als System Call vorgesehen.

### • Öffnen eines FIFOs

- ◇ Nach seiner Erzeugung kann ein FIFO von **jedem beliebigen Prozess** – unter Berücksichtigung der Zugriffsrechte – mit dem System Call `open()` geöffnet werden.

Die **fifo-spezifische Funktionalität** dieses System Calls wird durch die Funktion `fifo_open()` (definiert in `fs/fifo.c`) realisiert.

- ◇ Ein Prozess kann einen FIFO **nur zum Schreiben, nur zum Lesen** oder zum **Schreiben und Lesen** öffnen.
- ◇ Im **blockierenden Modus** (i.a. der Normalfall, Flag `O_NONBLOCK` nicht gesetzt) blockiert `open()` sowohl für Nur-Lesen als auch für Nur-Schreiben solange bis der FIFO auch für die jeweils entgegengesetzte Bearbeitungsrichtung geöffnet wird.
- ◇ Im **nicht-blockierenden Modus** (Flag `O_NONBLOCK` gesetzt) kehrt `open()` für **Nur-Lesen** auch dann **erfolgreich** zurück, wenn der FIFO noch nicht für Schreiben geöffnet wurde.  
Ein Aufruf von `open()` für **Nur-Schreiben** ohne dass der FIFO bereits für Lesen geöffnet ist, endet dagegen mit dem **Fehler ENXIO**.  
→ Ein Öffnen für nicht-blockierendes Schreiben ist nur möglich, wenn der FIFO zuvor für Lesen geöffnet wurde.
- ◇ Beim **Öffnen** eines FIFOs für **Lesen und Schreiben** kehrt `open()` sowohl im blockierenden als auch im nicht-blockierenden Modus **erfolgreich** zurück.  
Dies kann benutzt werden, um ein FIFO zum Schreiben zu öffnen, obwohl noch kein anderer Prozess den FIFO zum Lesen geöffnet hat.

### • Schreiben in und Lesen aus FIFOs

- ◇ In einen zum **Schreiben geöffneten** FIFO kann mittels des System Calls `write()` geschrieben werden.  
Aus einen zum **Lesen geöffneten** FIFO kann mittels des System Calls `read()` gelesen werden.  
Die in beiden System Calls aufgerufenen fifo-spezifischen File-Operationen sind mit den entsprechenden pipe-spezifischen File-Operationen identisch : `pipe_write()` bzw `pipe_read()`.  
→ **Lesen und Schreiben von FIFOs entspricht dem Schreiben und Lesen von "normalen" Pipes.**
- ◇ **Unterschied** zu "normalen" Pipes : Lesen und Schreiben kann über denselben File-Deskriptor erfolgen.
- ◇ Wenn mehrere Prozesse "gleichzeitig" in denselben FIFO schreiben, ist sichergestellt, dass die Daten nicht gemischt werden, solange mit einem `write()`-Aufruf nicht mehr als `PIPE_BUF` (=PIPE\_SIZE) Bytes (bei Intel-Architektur =4096 Bytes, def. für Anwender-Code in `linux/limits.h`) geschrieben werden.
- ◇ Beim Versuch in einen **FIFO zu schreiben**, dessen **Leseseite nicht geöffnet** ist, wird das Signal **SIGPIPE** vom Kernel an den schreibenden Prozess geschickt.  
Sowohl beim Ignorieren als auch beim Abfangen des Signals (prozesseigener Signalhandler) kehrt `write()` mit einem Fehler zurück und `errno` wird auf `EPIPE` gesetzt.

## C-Bibliotheks-Funktion `mkfifo()`

- **Funktionalität :** Erzeugung eines Directory-Eintrags für einen **FIFO** (*named pipe*).

- **Interface :**

```
int mkfifo(const char* path, mode_t mode);
```

- **Header-Datei :** `<sys/stat.h>`

- **Parameter :**
  - path* Pfadname (Zugriffspfad) des zu erzeugenden FIFOs
  - mode* Dateizugriffsrechte.  
Für diese ist eine bitweise ODER-Verknüpfung von einer oder mehreren der hierfür in `<sys/stat.h>` definierten Konstanten anzugeben.  
s. System Call `mkdir()` oder `creat()`.  
Die durch den Parameter angegebenen Zugriffsrechte werden mit der invertierten Dateikreierungsmaske `umask` des Prozesses bitweise UND-verknüpft.  
Die **tatsächlichen Zugriffsrechte** zum Eintrag ergeben sich somit zu :  
**`mode & ~umask`**

- **Rückgabewert :**
  - 0 bei **Erfolg**
  - -1 im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** Diese Funktion ruft den System Call Nr. **14** (`mknod`) auf :

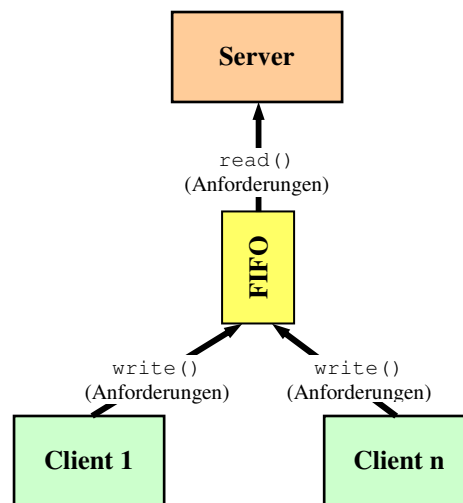
```
mknod(path, S_IFIFO | mode, 0);
```

- **Anmerkungen:**
  1. Der Datentyp `mode_t` ist – indirekt – in der Header-Datei `<sys/stat.h>` definiert als **`unsigned int`**.
  2. In POSIX ist diese Funktion direkt als System Call implementiert.
  3. Falls *path* bereits existiert, wird `errno` auf `EEXIST` gesetzt.

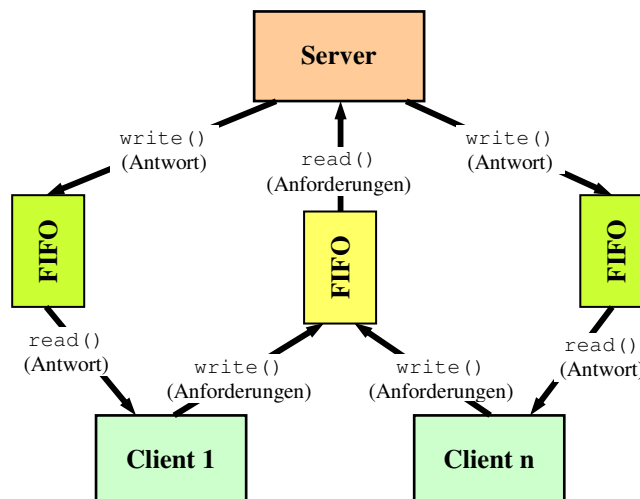
### FIFOs (Named Pipes) in Linux (3)

- **Anwendungsbeispiel : Verwendung von FIFOs in der Client-Server-Kommunikation**

- ◇ Der **Serverprozess erzeugt** ein **FIFO**, dessen Name im Dateisystem allen potentiellen Clients bekannt sein muß. Damit der Serverprozess auf Client-Anforderungen in einer Schleife warten kann, ohne durch den `open()`-System Call blockiert zu sein, muß er den FIFO sinnvollerweise für **Lesen und Schreiben öffnen**.
- ◇ Jeder **Clientprozess** schickt seine **Anforderungen** über diesen FIFO zum Serverprozess, nachdem er den FIFO für **Schreiben geöffnet** hat. Um ein Blockieren von `open()` bei nicht vorhandenem Server-Prozess zu vermeiden, sollte das Öffnen mit gesetztem `O_NONBLOCK`-Flag erfolgen  
Damit die einzelnen Client-Anforderungen nicht vermischt werden, darf jede nicht mehr als `PIPE_BUF` umfassen.



- ◇ Die **Antworten** des Serverprozesses an die verschiedenen Clientprozesse müssen jeweils über eine **eigene FIFO-Verbindung** geschickt werden. Hierfür erzeugt **jeder Client** seinen **eigenen FIFO** und **übermittelt** dem Serverprozess den FIFO-Zugriffspfad zusammen **mit seiner Anforderung**. Anschließend öffnet er diesen FIFO für Lesen. Der Serverprozess öffnet diesen FIFO für Schreiben und sendet seine Antwort über diesen an den anfordernden Client.
- ◇ Der **Serverprozess** sollte das **Signal SIGPIPE abfangen**, da der Fall auftreten kann, dass der Client nach Senden einer Anforderung an den Server vorzeitig – ohne eine Antwort abzuwarten – beendet und damit die Leseseite seines Empfangs-FIFOs geschlossen wird



## System V IPC - Allgemeines

### • Überblick

- ◇ **Unix System V** hat **drei IPC-Mechanismen** eingeführt :
  - ▷ **Semaphore**
  - ▷ **Message Queues**
  - ▷ **Shared Memory**
- ◇ Obwohl es sich hierbei um **unterschiedliche** Mechanismen handelt, sind sie weitgehend nach einem **gemeinsamen Konzept** implementiert und stellen **gleichartige Benutzerinterfaces** zur Verfügung.
- ◇ Bei allen drei Mechanismen werden vom Kernel auf Anforderung eines Prozesses jeweils spezifische **IPC-Objekte** angelegt und verwaltet, die prinzipiell von auch **nicht verwandten Prozessen** für eine gemeinsame **Kommunikation** genutzt werden können. Dabei besteht **keine Beschränkung** auf nur **zwei** Kommunikationspartner. Welche Prozesse ein angelegtes IPC-Objekt verwenden dürfen, wird durch **Zugriffsrechte** gesteuert, die beim Anlegen des Objekts gesetzt werden und gegebenenfalls später vom Objekt-Erzeuger oder von einem Root-Prozess geändert werden können.  
Die Zugriffsrechte entsprechen den Zugriffsrechten des Dateisystems. Allerdings hat `execute` keine Bedeutung. (nur `read`, `write` für `user`, `group`, `other`)
- ◇ Jedem angelegten IPC-Objekt wird vom Kernel eine eindeutige **Kennung** (Objekt-ID, *identifier*, nichtnegative ganze Zahl) zugeordnet. Über diese Kennung kann von den verschiedenen Prozessen zu dem Objekt **zugegriffen** werden.
- ◇ Mit jedem IPC-Objekt ist auch ein **Schlüssel** (*key*) verknüpft.  
Schlüssel sind vom Typ **key\_t**. Dieser Typ ist in **<sys/types.h>** (indirekt unter Verwendung von **<bits/types.h>** und **<bits/typesizes.h>**) definiert als **int**.  
Der Schlüssel muss beim Anlegen des Objekts vom erzeugenden Prozess angegeben werden. Andere Prozesse können über diesen Schlüssel die für den Zugriff benötigte Kennung eines bereits angelegten IPC-Objekts ermitteln.  
Der Versuch, mit einem Schlüssel ein neues Objekt anzulegen, für den ein IPC-Objekt gleichen Typs bereits existiert, schlägt fehl.  
Mit dem speziellen Schlüssel **IPC\_PRIVATE** (indirekt definiert in **<sys/ipc.h>**) lassen sich **schlüssellose** IPC-Objekte anlegen. Dieser Schlüssel bewirkt, dass immer ein neues IPC-Objekt angelegt wird.  
Damit andere Prozesse zu einem derartigen Objekt zugreifen können, muss ihnen die **Kennung** des Objekts vom erzeugenden Prozess übermittelt werden.

### • Verwendungsmöglichkeiten von IPC-Objekten durch nicht verwandte Prozesse

1. Die Prozesse, die über ein IPC-Objekt kommunizieren wollen (z.B. Server und Clients), **vereinbaren** einen **Schlüssel**. (z.B. in Headerdatei, Programmparameter, Programmeingabe).  
**Ein** Prozess (z.B. Server) **erzeugt** unter Verwendung dieses Schlüssels das **IPC-Objekt**. Die Erzeugungsfunktion liefert die **Kennung** des erzeugten Objekts zurück.  
Die **anderen Prozesse** (Clients) **ermitteln** unter Verwendung des Schlüssels die **Kennung** des bereits angelegten Objekts.  
Eventuelles **Problem** : Für den Schlüssel existiert bereits ein IPC-Objekt gleichen Typs.  
Der erzeugende Prozess muss dies erkennen, dieses Objekt vernichten und dann ein neues Objekt anlegen.
2. Ein Prozess (z.B. Server) erzeugt ein **schlüsselloses** IPC-Objekt. Die dabei erhaltene **Kennung** des erzeugten Objekts macht er den **anderen Prozessen** (z.B. Clients) **zugänglich**, z.B. durch Ablage in einer Datei oder mittels eines anderen IPC-Mechanismus.

### • Existenzdauer und Löschen von IPC-Objekten

- ◇ Ein angelegtes IPC-Objekt existiert solange, bis es **explizit gelöscht** wird oder das **System heruntergefahren** wird.
- ◇ Das Löschen kann durch **jeden** Benutzer, der **Schreibberechtigung** zum Objekt besitzt, sowie immer durch den Objekt-Erzeuger bzw –Besitzer und durch `root` erfolgen
  - ▷ innerhalb eines Prozesses mittels der hierfür geeigneten **System-V-API-Funktion**
  - ▷ auf der Betriebssystemebene mittels des **Kommandos `ipcrm`**

## System V IPC-API – Überblick

### • Betriebssystemfunktions-Gruppen

- ◇ Das System V IPC-API umfasst **drei Gruppen** von – jeweils **IPC-Mechanismusspezifischen** – Funktionen :
  - ▷ API-Funktionen zur **Erzeugung** und zur **Ermittlung** eines IPC-Objektes
  - ▷ API-Funktionen zur **Zustandsabfrage/-änderung** (einschliesslich **Löschung**) eines IPC-Objektes
  - ▷ API-Funktionen zur **Verwendung** eines IPC-Objektes
- ◇ Überblick :

Funktionsgruppe	Semaphore	Message Queue	Shared Memory
Erzeugung / Ermittlung	<code>semget ()</code>	<code>msgget ()</code>	<code>shmget ()</code>
Zustandsabfrage/-änderung, Löschung	<code>semctl ()</code>	<code>msgctl ()</code>	<code>shmctl ()</code>
Verwendung	<code>semop ()</code> <code>semtimedop ()</code>	<code>msgsend ()</code> <code>msgrcv ()</code>	<code>shmat ()</code> <code>shmdt ()</code>

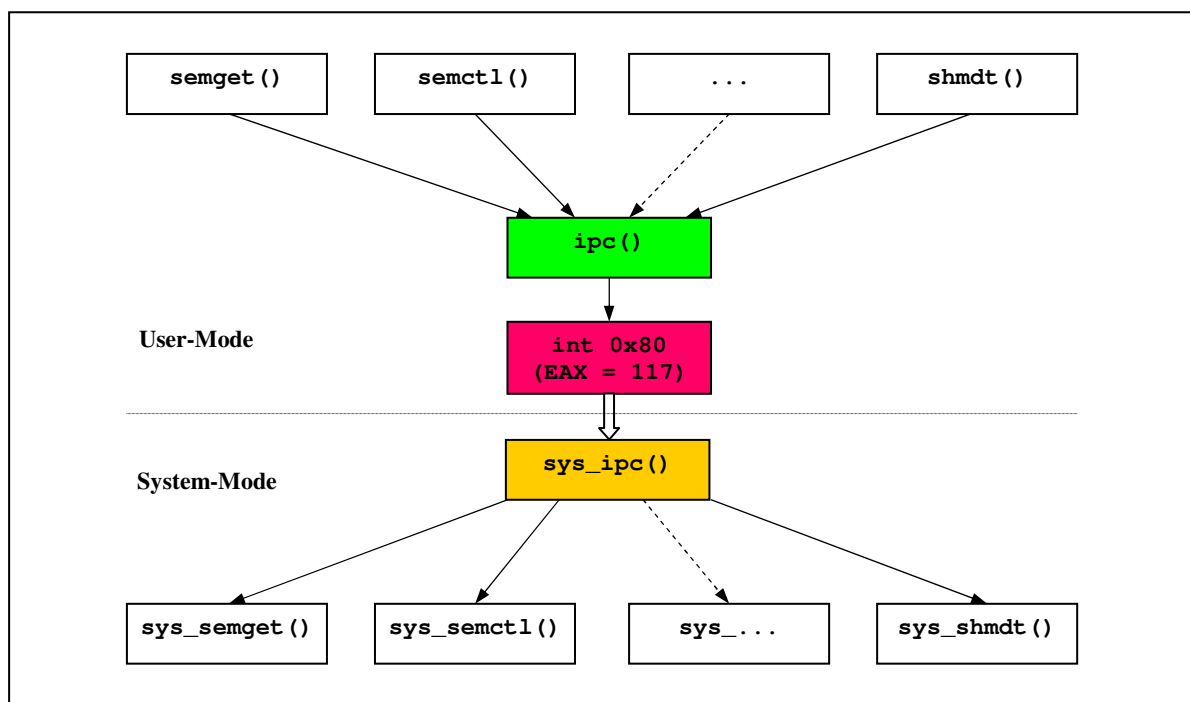
### • Prinzipielle Implementierung

Sämtliche System V IPC - Betriebssystemfunktionen (API-Funktionen) sind als **Unterfunktionen** eines **einzigen System Calls** implementiert : System Call **ipc ()** (Funktions-Nr. 117).

Die einzelnen Unterfunktionen sind durch eine **Unterfunktions-Nr.** gekennzeichnet, die im Register **EBX** (i386-Architektur) übergeben wird.

Die Unterfunktions-Nummern sind als **symbolische Konstanten** in **include/asm-generic/ipc.h** (Kernel) bzw **<asm-generic/ipc.h>** (GNU-C-Bibliothek, Anwendercode) definiert

Der Wrapper-Funktion des System Calls wird die Unterfunktions-Nr. als erster Parameter übergeben, die weiteren Parameter werden an die dadurch ausgewählte Betriebssystemfunktion weitergereicht.



Da für alle System V IPC - Betriebssystemfunktionen geeignete Wrapper-Funktionen existieren, werden i.a. diese und nicht die eigentliche System Call Wrapper-Funktion für ihren Aufruf eingesetzt.

## System V IPC – Gemeinsame Datenstrukturen des Kernels

- **Structure-Typ `struct ipc_ids`** (definiert in `ipc/util.h`)

- ◇ Grundlegender Datentyp für die Verwaltung der verschiedenen IPC-Objekte.

Für jeden der drei verschiedenen IPC-Objekt-Typen existiert im Kernel eine Variable dieses Typs :

- ▷ für **Semaphore** : `static struct ipc_ids init_sem_ids` (definiert in `ipc/sem.c`)
- ▷ für **Message Queues** : `static struct ipc_ids init_msg_ids` (definiert in `ipc/msg.c`)
- ▷ für **Shared Memory** : `static struct ipc_ids init_shm_ids` (definiert in `ipc/shm.c`)

◇

```
struct ipc_ids {  
    int in_use; /* Anzahl der aktuell verwendeten IPC-Objekte des jeweiligen Typs */  
    int max_id;  
    unsigned short seq;  
    unsigned short seq_max;  
    struct mutex mutex; /* zur Zugriffssynchronisation */  
    struct ipc_id_ary nullentry;  
    struct ipc_id_ary* entries; /* Pointer auf Array mit Pointern auf IPC-Objekte des jew. Typs */  
};
```

- **Structure-Typ `struct ipc_id_ary`** (definiert in `ipc/util.h`)

- ◇ Dieser Typ kapselt ein Array mit Pointern, die die im System existierenden IPC-Objekte des jeweiligen Typs referieren. Formal verweisen die im Array vorhandenen Pointer auf eine Struktur, die sich am Anfang aller IPC-Objekte – unabhängig von ihrem jeweiligen Typ – befindet.

Das Pointer-Array wird bei der Initialisierung des System V IPC-Subsystems dynamisch alloziert. Die dabei festgelegte tatsächliche Größe des Arrays wird in einer eigenen Komponente des Structure-Typs gespeichert.

Sie gibt an, wieviel IPC-Objekte des jeweiligen Typs maximal gleichzeitig existieren können.

Für jeden IPC-Objekt-Typ ist durch eine Kernel-Konstante (`SEMMNI`, `MSGMNI`, `SHMMNI`) ein entsprechender Maximalwert festgelegt.

◇

```
struct ipc_id_ary {  
    int size; /* Größe des Arrays */  
    struct kern_ipc_perm *p[0]; /* Array von Pointern auf IPC-Objekte */  
};
```

- **Structure-Typ `struct kern_ipc_perm`** (definiert in `include/linux/ipc.h`)

- ◇ Dieser Typ fasst für ein IPC-Objekt den Key, die Zugriffsrechte, Informationen über den Besitzer und den Erzeuger sowie Information zur Bildung/Ermittlung der Kennung zusammen

Bei den **Datentypen**, die die **verschiedenen IPC-Objekte beschreiben**, ist die **erste Komponente** immer von diesem Typ.

◇

```
struct kern_ipc_perm  
{  
    spinlock_t lock;  
    int deleted;  
    key_t key; /* Key des IPC-Objektes */  
    uid_t uid; /* effektive UID des Besitzers des IPC-Objekts */  
    gid_t gid; /* effektive GID des Besitzers des IPC-Objekts */  
    uid_t cuid; /* effektive UID des Erzeugers des IPC-Objekts */  
    gid_t cgid; /* effektive GID des Erzeugers des IPC-Objekts */  
    mode_t mode; /* Zugriffsrechte */  
    unsigned long seq; /* "Sequenznummer", verwendet zur Bildung der Kennung */  
    void* security;  
};
```

## System V IPC – Gemeinsame Datenstrukturen für den User-Code

- **Structure-Typ struct ipc\_perm** (definiert in `<bits/ipc.h>`, eingebunden durch `<sys/ipc.h>`)

- ◇ Dieser Typ ist die "User-Code-Version" des Kernel-Structure-Typs struct kern\_ipc\_perm. Er wird in den API-Funktionen zur Statusabfrage/-änderung der IPC-Objekte benötigt. Mit diesen Funktionen ist es u.a. möglich, die effektive UID und GUID des Besitzers sowie die Zugriffsrechte eines IPC-Objektes zu verändern.

◇

```
/* Data structure used to pass permission information to IPC operations. */
struct ipc_perm
{
    __key_t __key;           /* Key. */
    __uid_t uid;             /* Owner's user ID. */
    __gid_t gid;             /* Owner's group ID. */
    __uid_t cuid;            /* Creator's user ID. */
    __gid_t cgid;            /* Creator's group ID. */
    unsigned short int mode; /* Read/write permission. */
    unsigned short int __pad1;
    unsigned short int __seq; /* Sequence number. */
    unsigned short int __pad2;
    unsigned long int __unused1;
    unsigned long int __unused2;
};
```

**Anmerkung :** Im Zusammenspiel der Headerdateien `<sys/ipc.h>`, `<bits/ipc.h>`, `<bits/types.h>` und `<bits/typesizes.h>` sind definiert :

<code>__key_t</code>	als	<code>key_t</code>	als	<code>int</code>
<code>__uid_t</code>	als	<code>uid_t</code>	als	<code>unsigned int</code>
<code>__gid_t</code>	als	<code>gid_t</code>	als	<code>unsigned int</code>

- **Symbolische Konstanten** (definiert in `<bits/ipc.h>`, eingebunden durch `<sys/ipc.h>`)

- ◇ Zur Anwendung in den System V API-Funktionen sind in `<bits/ipc.h>` (eingebunden durch `<sys/ipc.h>`) die folgenden symbolischen Konstanten definiert :

```
/* Mode bits for `msgget', `semget', and `shmget'. */
#define IPC_CREAT 01000 /* Create key if key does not exist. */
#define IPC_EXCL 02000 /* Fail if key exists. */
#define IPC_NOWAIT 04000 /* Return error on wait. */

/* Control commands for `msgctl', `semctl', and `shmctl'. */
#define IPC_RMID 0 /* Remove identifier. */ /* → Löschen IPC-Objekt */
#define IPC_SET 1 /* Set `ipc_perm' options. */
#define IPC_STAT 2 /* Get `ipc_perm' options. */
#ifdef __USE_GNU
# define IPC_INFO 3 /* See ipcs. */ /* → Ermittlung von Info ueber IPC-Objekt */
#endif

/* Special key values. */
#define IPC_PRIVATE ((__key_t) 0) /* Private key. */
```

- **Betriebssystemkommando ipcs**

- ◇ Mit dem Betriebssystemkommando **ipcs** können **Informationen** (Typ, Key, Kennung, Besitzer, Zugriffsrechte) über einzelne, alle IPC-Objekte eines Typs oder alle aktuell existierenden **IPC-Objekte** ermittelt werden

## System V IPC – Semaphore (1)

### • Überblick

- ◇ Semaphore (Sperrvariable) ermöglichen die **Zugriffs-Synchronisation** von **mehreren Prozessen** zu nur exklusiv bzw eingeschränkt nutzbaren **Ressourcen**.
- ◇ Eine **System V Semaphore** (Semaphore-Objekt) besteht aus einem ganzen **Satz elementarer** (einfacher) **Semaphoren** (→ **Semaphoren-Menge**).  
Damit lassen sich gegebenenfalls Operationen auf mehrere elementare Semaphore zu einer einzigen – atomar erscheinenden – Operation zusammenfassen.  
Semaphoren-Mengen erlauben eine detaillierte Aufteilung von kritischen Abschnitten bzw Ressourcen bezüglich ihres Zugriffs-Schutzes.  
Die Anzahl der Semaphore in einer Semaphoren-Menge muß bei der Erzeugung eines Semaphor-Objektes angegeben werden.
- ◇ Durch **Kernel-Konstante** festgelegte **Maximalwerte** (definiert in `include/linux/sem.h`):  
(die angegebenen Werte beziehen sich auf den Kernel 2.6.22.17)
  - ▷ **SEMMNI** – Maximalzahl gleichzeitig existierender Semaphore-Objekte : **1024**
  - ▷ **SEMMSL** – Maximalzahl der in einem Semaphore-Objekt zusammenfassbaren Semaphore : **250**

### • Implementierung von Semaphore-Objekten

- ◇ System V Semaphore-Objekte werden im wesentlichen durch das Zusammenspiel der beiden folgenden Structure-Datentypen implementiert :
  - ▷ **struct sem**
  - ▷ **struct sem\_array**

### • Structure-Typ **struct sem** (definiert in `include/linux/sem.h`)

- ◇ Dieser Datentyp modelliert eine **elementare Semaphore**.  
Eine **Semaphoren-Menge** wird durch ein **Array** von Elementen dieses Typs gebildet.

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
    int semval;        /* current value */
    int sempid;        /* pid of last operation */
};
```

### • Structure-Typ **struct sem\_array** (definiert in `include/linux/sem.h`)

- ◇ Dieser Datentyp kann als **Semaphore-Objekt-Structure** bezeichnet werden. Er enthält u.a. einen Pointer auf das erste Element eines Arrays von **struct sem** Elementen → Semaphoren-Menge (Komponente **sem\_base**)

```
/* One sem_array data structure for each set of semaphores in the system. */
struct sem_array {
    struct kern_ipc_perm sem_perm;        /* permissions .. see ipc.h */
    int sem_id;                          /* Kennung (ID) des Semaphore-Objekts */
    time_t sem_otime;                    /* last semop time */
    time_t sem_ctime;                    /* last change time */
    struct sem* sem_base;                 /* ptr to first semaphore in array */
    struct sem_queue* sem_pending;        /* pending operations to be processed */
    struct sem_queue** sem_pending_last; /* last pending operation */
    struct sem_undo* undo;                /* undo requests on this array */
    unsigned long sem_nsems;              /* no. of semaphores in array */
};
```

- ◇ Die Komponenten **sem\_pending** und **sem\_pending\_last** referieren eine mit dem Semaphor-Objekt verknüpfte **Liste** von – **blockierten** – **Prozessen**, die auf Ausführung der von ihnen angeforderten **Semaphor-Operationen** warten



## System V IPC – Semaphore (2)

- **Structure-Typ `struct sem_queue`** (definiert in `include/linux/sem.h`)

- ◇ Dieser Datentyp dient zum Aufbau einer **doppelt verketteten Liste** von **Prozessen**, die **blockiert** sind und auf **Freigabe** einer **Semaphore** warten (**Warteschlange**), d.h. deren angeforderte Semaphore-Operation noch nicht ausgeführt werden konnte

**Pro Semaphor-Objekt** existiert genau **eine Warteschlange**

Neben dem jeweiligen wartenden **Prozess** (Komponente `sleeper`) referieren die einzelnen Listenelemente die vom Prozess angeforderten **Semaphor-Operationen** (Komponente `sops` vom Typ `struct sembuf`).

◇

```
/* One queue (element) for each sleeping process in the system. */
struct sem_queue {
    struct sem_queue* next;      /* next entry in the queue */
    struct sem_queue** prev;     /* previous entry in the queue, *(q->prev) == q */
    struct task_struct* sleeper; /* this process */
    struct sem_undo* undo;       /* undo structure */
    int pid;                    /* process id of requesting process */
    int status;                 /* completion status of operation */
    struct sem_array* sma;      /* semaphore array for operations */
    int id;                     /* internal sem id */
    struct sembuf* sops;        /* array of pending operations */
    int nsops;                  /* number of operations */
    int alter;                  /* does the operation alter the array? */
};
```

- ◇ Die Komponente `undo` (Typ `struct sem_undo`) verweist auf ein **Undo-Element**.  
Undo-Elemente sind zu einfach verketteten **Undo-Listen** zusammengefasst. Für jedes Semaphor-Objekt und jeden System V Semaphore verwendenden Prozess existiert eine Undo-Liste (Per-Semaphore-Listen und Per-Prozess-Listen). **Jedes Undo-Element** befindet sich sowohl in einer **Per-Prozess-Liste** als auch in einer **Per-Semaphor-Liste**.  
Die Komponente `sysvsem` des Prozess-Deskriptors verweist auf die Undo-Liste des Prozesses, die Komponente `undo` der Semaphore-Objekt-Structure (Typ `struct sem_array`) verweist auf den Anfang der Undo-Liste des Semaphor-Objekts  
Undo-Listen ermöglichen das **Rückgängigmachen** von **Semaphor-Operationen**, die durch einen abgestürzten oder sonstwie irregulär beendeten Prozess vorgenommen worden sind.  
Dadurch kann das Semaphor-Objekt wieder in einen konsistenten Zustand versetzt werden, wodurch gegebenenfalls Deadlocks verhindert werden können.  
Die Semaphor-Operationen, die gegebenenfalls rückgängig gemacht werden sollen, müssen bei der **Operationsanforderung** entsprechend **markiert** werden.

- **Structure-Typ `struct sembuf`** (definiert in `include/linux/sem.h`)

- ◇ Dieser Datentyp beschreibt eine **Operation** auf eine **elementare Semaphore**.  
Die – durch einen API-Funktionsaufruf angeforderten – **Operationen** auf ein **Semaphoren-Objekt** (Semaphoren-Menge !) werden in einem Array von Elementen dieses Typs zusammengefasst.

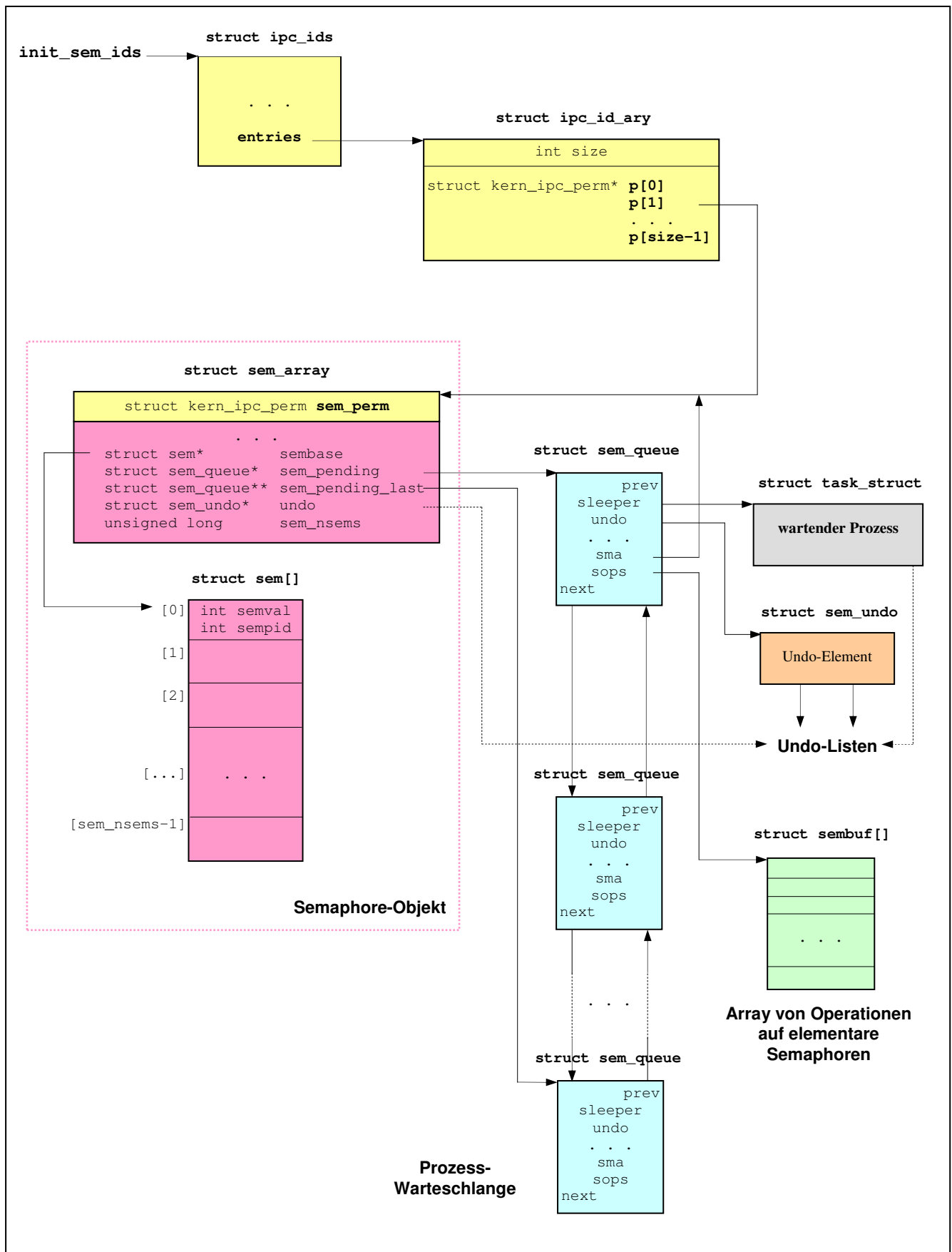
◇

```
/* semop system calls takes an array of these. */
struct sembuf {
    unsigned short sem_num;      /* semaphore index in array */
    short sem_op;               /* semaphore operation */
    short sem_flg;              /* operation flags */
};
```

- ◇ Der Datentyp wird **auch** im **User-Code**, als Parameter für die API-Funktionen `semop()` und `semtimedop()`, verwendet. Hierfür ist er in der Headerdatei `<sys/sem.h>` definiert.
- ◇ **Bedeutung** und mögliche **Werte** der einzelnen **Komponenten** werden im Kontext der Betriebssystemfunktion `semop()` erläutert.

## System V IPC – Semaphore (3)

### • Überblick über die Semaphore-bezogenen Datenstrukturen des Kernels



## System V IPC API-Funktion **semget**

- **Funktionalität :** **Erzeugung** eines neuen Semaphore-Objekts bzw **Ermittlung** der **Kennung** (identifizier) eines existierenden Semaphore-Objekts. Ein Semaphore-Objekt enthält eine **Semaphoren-Menge**. Ein neu angelegtes Semaphore-Objekt wird i.a. **nicht initialisiert**. (In Linux werden zwar alle elementaren Semaphoren im Objekt mit 0 initialisiert, dies ist aber kein portables Verhalten) Bei **Erfolg** Rückgabe der **Kennung** des – neu erzeugten bzw existierenden – Semaphore-Objekts.

- **Interface :**

```
int semget(key_t key, int nsems, int semflg)
```

- **Header-Dateien :** **<sys/ipc.h>** (symbolische Konstanten)  
**<sys/sem.h>** (Function Prototype)  
**<sys/stat.h>** (für Zugriffsberechtigungsflags)

- **Parameter :**

*key* **Schlüssel** (*key*) des Semaphore-Objekts.

spezieller Wert : **IPC\_PRIVATE**

(def. in *<bits/ipc.h>*, eingebunden durch *<sys/ipc.h>*)

→ bewirkt, dass immer ein neues Semaphore-Objekt angelegt wird

*nsems* **Anzahl** der **elementaren Semaphoren** im Semaphore-Objekt (muss **<= SEMMSL** sein).

Für die Ermittlung der Kennung eines existierenden Semaphore-Objekts kann 0 angegeben werden.

*semflg* **Steuerflags** und **Zugriffsberechtigungsflags** (bitweis-oder-verknüpft), bzw 0

Die zulässige **Steuerflags** sind :

(def. in *<bits/ipc.h>*, eingebunden durch *<sys/ipc.h>*).

**IPC\_CREAT** Erzeugung eines neuen Semaphore-Objekts, falls noch keines mit dem angegebenen Schlüssel existiert (Ausnahme : Schlüssel **IPC\_PRIVATE**), sonst Rückgabe der Kennung des bereits vorhandenen Objekts.

**IPC\_EXCL** Funktion endet fehlerhaft, falls ein neues Semaphore-Objekt erzeugt werden soll und bereits eines mit dem angegebenen Schlüssel existiert

→ diese Flag ist nur zusammen mit dem Flag **IPC\_CREAT** sinnvoll anwendbar

Die **Zugriffsberechtigungsflags** müssen nur angegeben werden, wenn ein neues Semaphore-Objekt erzeugt werden soll.

Es handelt sich um die gleichen Flags, die auch bei der Erzeugung neuer Dateien verwendet werden (def. in *<sys/stat.h>*, s.a. System Call *creat()*), allerdings haben die Flags für die Ausführungs-Berechtigung hier keine Bedeutung

- **Rückgabewert :** - **Kennung** (*identifizier*) des Semaphore-Objekts, bei Erfolg  
- **-1** im Fehlerfall, *errno* wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **2** (symb. Konst **SEMGET**)  
→ *sys\_ipc(...)* (in *arch/i386/kernel/sys\_386.c*)  
→ *sys\_semget(...)* (in *ipc/sem.c*)

- **Anmerkungen :** 1. Der Datentyp **key\_t** ist – indirekt – in den Header-Dateien *<sys/types.h>* und *<sys/ipc.h>* definiert als **int**.

2. **Anwendungs-Beispiele :**

```
#define SEMKEY 8421 /* symbolische Konstante für Objekt-Schlüssel */
#define SEMPERM 0666 /* Zugriffsberechtigungen: rw-rw-rw- */

/* Erzeugung eines neuen Semaphore-Objektes mit 1 elementarer Semaphore */
int id = semget(SEMKEY, 1, IPC_CREAT|IPC_EXCL|SEMPERM);

/* Ermittlung der Kennung eines existierenden Semaphore-Objekts */
int id = semget(SEMKEY, 1, 0);
```

## System V IPC – für die API-Funktion `semctl` verwendete User-Code-Datenstrukturen

- **Structure-Typ `struct semid_ds`** (definiert in `<bits/sem.h>`, eingebunden durch `<sys/sem.h>`)

◇ Dieser Datentyp dient zur Beschreibung eines **Semaphore-Objekts** im **User-Code**.

◇

```
/* Data structure describing a set of semaphores. */
struct semid_ds
{
    struct ipc_perm sem_perm;           /* operation permission struct */
    __time_t sem_otime;                 /* last semop() time */
    unsigned long int __unused1;
    __time_t sem_ctime;                 /* last time changed by semctl() */
    unsigned long int __unused2;
    unsigned long int sem_nsems;        /* number of semaphores in set */
    unsigned long int __unused3;
    unsigned long int __unused4;
};
```

**Anmerkung:** In der Headerdatei `<bits/sem.h>` (die ihrerseits durch `<sys/sem.h>` eingebunden wird) ist indirekt – durch das Zusammenspiel der eingebundenen Headerdateien `<sys/types.h>`, `<bits/types.h>` `<bits/typesizes.h>` und `<time.h>` – definiert:

`__time_t` als `time_t` als `long int`

◇ Der o.a. für die Komponente `sem_perm` verwendete Datentyp **`struct ipc_perm`** ist die User-Code-Version des Datentyps `struct kern_ipc_perm`.

Er fasst für ein IPC-Objekt den Key, die Zugriffsrechte, Informationen über den Besitzer und den Erzeuger sowie Information zur Bildung/Ermittlung der Kennung zusammen.

Seine Definition ist weiter oben (V-BS-A54-00) beschrieben.

- **Structure-Typ `struct seminfo`** (definiert in `<bits/sem.h>`, eingebunden durch `<sys/sem.h>`)

◇ Dieser –linux-spezifische – Datentyp dient zur Ablage einiger system-weiter **semaphore-bezogener Kernel-Parameter** und **Grenzwerte** im **User-Code**

◇

```
struct seminfo
{
    int semmap; /* wird derzeit nicht verwendet */
    int semmni; /* Maximalzahl gleichzeitig existierender Semaphore-Objekte */
    int semmns; /* Maximale Gesamtzahl aller elementaren Semaphore in allen Semaphore-Objekten */
    int semmnu; /* Maximalzahl aller Undo-Elemente im System, wird derzeit nicht verwendet */
    int semmsl; /* Maximalzahl der in einem Semaphor-Objekt zusammenfassbaren elementaren Semaphore */
    int semopm; /* Maximalzahl der Semaphore-Operationen pro Aufruf der API-Funktion semop() */
    int semume; /* Maximalzahl der Elemente in der Undo-Liste eines Prozesses, derzeit nicht verwendet */
    int semusz; /* Groesse eines Undo-Elementes (struct sem_undo) */
    int semvmx; /* Maximaler Wert, den eine elementare Semaphore annehmen kann */
    int semaem; /* Maximaler Korrektur-Wert, der für eine elementare Semaphore in einem Undo-Element */
}; /* gespeichert werden kann */
```

## System V IPC API-Funktion **semctl** (1)

- **Funktionalität :** Ausführung von **Kontroll-Operationen** (Zustandsabfrage/-änderung, Löschung) auf einem Semaphoren-Objekt.  
Die auszuführende Operation wird durch einen Kommando-Parameter festgelegt.  
Die Funktion benötigt – in Abhängigkeit vom Kommando-Parameter – drei oder vier Parameter

- **Interface :**

```
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */)

```

- **Header-Dateien :** **<sys/types.h>**

**<sys/ipc.h>** (symbolische Konstanten)

**<sys/sem.h>** (*Function Prototype* und symbolische Konstanten)

- **Parameter :**

*semid*     **Kennung** (*identifier*) des Semaphore-Objekts.  
bzw für Kommando **SEM\_STAT** : **Index** im kernel-internen Semaphore-Objekt-Pointer-Array

*semnum*   **Nummer** der vom Kommando betroffenen **elementaren Semaphore** im Semaphore-Objekt  
(== Index im `struct sem` -Array).  
Bei Kommandos, die sich auf das gesamte Semaphore-Objekt und nicht auf eine einzelne elementare Semaphore beziehen, wird dieser Parameter ignoriert

*cmd*        Auszuführendes **Kommando** (Kommando-Parameter),  
zulässige Kommandos sind :  
(def. in `<bits/ipc.h>` bzw `<bits/sem.h>`, eingebunden durch `<sys/sem.h>`)

IPC_STAT	GETVAL	}	s. gesonderte Beschreibung
IPC_SET	SETALL		
IPC_RMID	SETVAL		
GETALL	IPC_INFO (linux-spezifisch)		
GETNCNT	SEM_INFO (linux-spezifisch)		
GETPID	SEM_STAT (linux-spezifisch)		
GETZCNT			

*arg*        Kommando-abhängiger **Ein- bzw Ausgabeparameter**.  
Der hierfür verwendete Union-Typ **muss** in dem die API-Funktion aufrufenden User-Programm wie folgt **definiert** werden (in früheren Versionen war er in `<bits/sem.h>` definiert) :

```
union semun
{ int val; /* Wert für SETVAL */
  struct semid_ds* buf; /* Buffer für IPC_STAT u. IPC_SET */
  unsigned short* array; /* Array für GETALL u. SETALL */
  struct seminfo* __buf; /* Buffer für IPC_INFO (linux-spezifisch) */
};

```

- **Rückgabewert :** - bei Erfolg kommando-abhängiger Wert :

für GETNCNT u. GETZCNT : Anzahl der wartenden Prozesse

für GETPID : PID des Prozesses, der als letzter die Semaphore *semnum* geändert hat

für GETVAL : aktueller Wert der Semaphore *semnum*

für IPC\_INFO u. SEM\_INFO : Index der höchsten belegten Komponente des kernel-internen Pointer-Arrays auf Semaphore-Objekte

für SEM\_STAT : Kennung des Semaphore-Objekts, das durch Index *semid* referiert wird

für alle anderen Kommandos : 0

- -1 im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **3** (symb. Konst **SEMCTL**)  
     → `sys_ipc(...)` (in `arch/i386/kernel/sys_386.c`)  
     → `sys_semctl(...)` (in `ipc/sem.c`)

## System V IPC API-Funktion `semctl` (2)

- **Zulässige Kommandos** (zulässige Werte für den Parameter `cmd`) **1. Teil:**

- IPC\_STAT** Ermittlung von Informationen über das Semaphore-Objekt mit der Kennung `semid`.  
Ablage der Informationen in der durch `arg.buf` referierten `struct semid_ds`-Variablen.  
Der Parameter `semnum` wird ignoriert.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen
- IPC\_SET** Setzen einiger Komponenten der Kernel-Struktur (Typ `struct sem_array`), die das Semaphore-Objekt mit der Kennung `semid` beschreibt. Die zu setzenden Werte werden der durch `arg.buf` referierten `struct semid_ds`-Variablen entnommen.  
Gesetzt werden können folgende Komponenten: `sem_perm.uid`, `sem_perm.gid` sowie die 9 niederwertigen Bits von `sem_perm.mode`.  
Die effektive UID des aufrufenden Prozesses muss mit der UID des Objekt-Erzeugers oder Objekt-Besitzers übereinstimmen oder es muss ein `root`-Prozess sein.  
Der Parameter `semnum` wird ignoriert.
- IPC\_RMID** Löschung des Semaphore-Objekts mit der Kennung `semid` und Aufwecken aller am Semaphore-Objekt wartenden Prozesse.  
Die effektive UID des aufrufenden Prozesses muss mit der UID des Objekt-Erzeugers oder Objekt-Besitzers übereinstimmen oder es muss ein `root`-Prozess sein.  
Der Parameter `semnum` wird ignoriert, der Parameter `arg` wird nicht benötigt (Aufruf mit 3 Par.)
- GETALL** Ermittlung der aktuellen Werte aller elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid`. Ablage der Werte in dem durch `arg.array` referierten Array.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen.  
Der Parameter `semnum` wird ignoriert.
- GETNCNT** Ermittlung der Anzahl Prozesse, die auf eine Erhöhung des Wertes der durch `semnum` ausgewählten elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid` warten.  
Diese Anzahl wird als Funktionswert zurückgegeben.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen.  
Der Parameter `arg` wird nicht benötigt (Aufruf mit 3 Parametern)
- GETPID** Ermittlung der PID des Prozesses, der als letzter den Wert der durch `semnum` ausgewählten elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid` geändert hat.  
Die PID wird als Funktionswert zurückgegeben.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen.  
Der Parameter `arg` wird nicht benötigt (Aufruf mit 3 Parametern)
- GETVAL** Ermittlung des aktuellen Werts der durch `semnum` ausgewählten elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid`. Der Wert wird als Funktionswert zurückgegeben.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen.  
Der Parameter `arg` wird nicht benötigt (Aufruf mit 3 Parametern)
- GETZCNT** Ermittlung der Anzahl Prozesse, die darauf warten, dass der Wert der durch `semnum` ausgewählten elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid` gleich 0 wird.  
Diese Anzahl wird als Funktionswert zurückgegeben.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen.  
Der Parameter `arg` wird nicht benötigt (Aufruf mit 3 Parametern)
- SETALL** Setzen der Werte aller elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid`. Die zu setzenden Werte werden dem durch `arg.array` referierten Array entnommen.  
Der aufrufende Prozess muss Schreibrecht für das Semaphore-Objekt besitzen.  
Der Parameter `semnum` wird ignoriert.
- SETVAL** Setzen des Werts der durch `semnum` ausgewählten elementaren Semaphore des Semaphore-Objekts mit der Kennung `semid` auf den Wert `arg.val`.  
Der aufrufende Prozess muss Schreibrecht für das Semaphore-Objekt besitzen.

## System V IPC API-Funktion `semctl` (3)

- **Zulässige Kommandos** (zulässige Werte für den Parameter `cmd`), **2. Teil** :

**IPC\_INFO** (linux-spezifisch)

erforderlich für Verwendung : `#define __USE_GNU` (vor `#include <sys/ipc.h>`)  
Ermittlung einiger systemweiter semaphore-bezogener Kernel-Parameter und Grenzwerte.  
Ablage dieser Info in der durch `arg.__buf` referierten `struct seminfo`-Variablen.  
Der Parameter `semnum` wird ignoriert.

**SEM\_INFO** (linux-spezifisch)

Ermittlung derselben Info wie bei `IPC_INFO` mit folgenden Änderungen :  
(`arg.__buf`) -> `semusz` wird auf die Anzahl der aktuell existierenden Semaphore-Objekte,  
(`arg.__buf`) -> `semaem` wird auf die Anzahl der aktuell insgesamt (in allen Semaphore-Objekten) existierenden elementaren Semaphore gesetzt.  
Der Parameter `semnum` wird ignoriert.

**SEM\_STAT** (linux-spezifisch)

Ermittlung von Informationen über ein Semaphore-Objekt wie bei `IPC_STAT`.  
Der Parameter `semid` wird jedoch nicht als Kennung des Objekts, sondern als Index des kernel-internen Pointer-Arrays auf Semaphore-Objekte interpretiert.  
Der Parameter `semnum` wird ignoriert.  
Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen

- **Anmerkungen :**    **Anwendungs-Beispiele :**

```
int id = ... ;           /* Kennung eines Semaphore-Objekts */
union semun sun;

/* Ermittlung semaphore-bezogener Kernel-Parameter und Grenzwerte */
struct seminfo sinf;     /* Variable zur Aufnahme relevanter Kernel-Parameter */
sun.__buf = &sinf;
semctl(id, 0, SEM_INFO, sun);

/* Ermittlung von Informationen über ein Semaphore-Objekt */
struct semid_ds sds;     /* Variable zur Aufnahme von Info über Semaphore-Objekt */
sun.buf = &sds;
semctl(id, 0, IPC_STAT, sun);

/* Initialisierung einer elementaren Semaphore */
sun.val = 1;             /* Initialisierungswert */
semctl(id, 0, SETVAL, sun);

/* Löschen eines Semaphore-Objekts */
semctl(id, 0, IPC_RMID);
```

- Funktionalität : Durchführung von Operationen auf ausgewählte einfache Semaphore einer Semaphore-Menge (Semaphore-Objekt).**  
 Für jede beteiligte Semaphore muss die auszuführende Operation gesondert spezifiziert werden.  
 Die Funktion arbeitet **atomar** : Es werden entweder alle oder keine der angegebenen Operationen ausgeführt.

```
int semop(int semid, struct sembuf* sops, unsigned nsops)
```

- **Parameter :**

Der Element-Typ **struct sembuf** ist wie folgt definiert in `<sys/sem.h>` :

Als **Flags** (Komponente `sem_flg`) können angegeben werden (auch bitweise oder-verknüpft) :

- ▷ **SEM\_UNDO** bewirkt, dass die Operation in die Undo-Liste für das Semaphore-Objekt sowie für den aufrufenden Prozess aufgenommen wird und damit gegebenenfalls rückgängig gemacht werden kann
- ▷ **IPC\_NOWAIT** Wenn die spezifizierte Operation nicht ausgeführt werden kann, bewirkt dieses Flag, dass der aufrufende Prozess nicht in den Wartezustand versetzt wird, sondern die Funktion mit einem Fehler (**EAGAIN**) endet

Drei verschiedene Operationen sind möglich :

- ▷ `sem_op > 0`      **Erhöhung** des Semaphore-Wertes
- ▷ `sem_op < 0`      **Erniedrigung** des Semaphore-Wertes
- ▷ `sem_op == 0`      **Überprüfung** des akt. Semaphore-Wertes **auf 0**

Genauere **Beschreibung** der **Operationen** s. nächste Seite

*nsops*      Größe des Operations-Arrays (= **Anzahl** der auszuführenden **Operationen**)

- **Rückgabewert :**
  - 0 bei Erfolg
  - -1 im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung** : mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **1** (symb. Konst **SEMOP**)
  - `sys_ipc(...)` (in `arch/i386/kernel/sys_386.c`)
  - `sys_semop(...)` (in `ipc/sem.c`)
  - `sys_semtimedop(...)` (in `ipc/sem.c`)



## System V IPC API-Funktion `semop` (2)

### • Beschreibung der Semaphore-Operationen

Die für eine einfache Semaphore auszuführende Operation wird durch den Wert der Komponente `sem_op` des für diese Semaphore zuständigen Elements des Operations-Arrays (Typ `struct sembuf`) festgelegt.

- ▷ `sem_op > 0` Der aufrufende Prozess muss Schreibrecht für das Semaphore-Objekt besitzen.  
Der Wert wird zum akt. Semaphore-Wert (Komponente `semval`) **addiert**.  
Falls das Flag `SEM_UNDO` für diese Operation gesetzt ist, wird ein neues Undo-Element erzeugt bzw. ein bereits für diese Semaphore und diesen Prozess vorhandenes Undo-Element auf den neuesten Stand gebracht.  
Die Operation kann immer ausgeführt werden → kein Warten (Blockieren) des Prozesses
- ▷ `sem_op < 0` Der aufrufende Prozess muss Schreibrecht für das Semaphore-Objekt besitzen.  
Fallunterscheidung :
  - ▷ `abs(sem_op) <= akt. Semaphore-Wert (Komponente semval)`  
`abs(sem_op)` wird vom akt. Semaphore-Wert **subtrahiert**  
Falls das Flag `SEM_UNDO` für diese Operation gesetzt ist, wird ein neues Undo-Element erzeugt bzw. ein bereits für diese Semaphore und diesen Prozess vorhandenes Undo-Element auf den neuesten Stand gebracht.  
→ kein Warten (Blockieren) des Prozesses
  - ▷ `abs(sem_op) > akt. Semaphore-Wert (Komponente semval)`
    - Das Flag `IPC_NOWAIT` ist **nicht gesetzt**.  
Der Prozess wird – unabhängig davon, ob die übrigen spezifizierten Semaphore-Operationen ausgeführt werden können oder nicht – in den **Wartezustand** versetzt  
Wenn der akt. Semaphore-Wert `>= abs(sem_op)` geworden ist, wird der Prozess wieder aufgeweckt (Voraussetzung : kein Warten wegen anderer in der Funktion spezifizierten Op.) und die Funktion beendet.
    - Das Flag `IPC_NOWAIT` ist **gesetzt**.  
Die Funktion kehrt sofort mit dem Fehler **EAGAIN** zurück (Prozess blockiert nicht).  
Es wird keine der übrigen spezifizierten Operationen ausgeführt.
- ▷ `sem_op == 0` Der aufrufende Prozess muss Leserecht für das Semaphore-Objekt besitzen  
**Überprüfung** des akt. Semaphore-Wertes auf `0` ("*wait-for-zero*" operation)
  - ▷ akt. Semaphore-Wert `== 0` → kein Warten des Prozesses
  - ▷ akt. Semaphore-Wert `!= 0`
    - Das Flag `IPC_NOWAIT` ist **nicht gesetzt**.  
Der Prozess wird – unabhängig davon, ob die übrigen spezifizierten Semaphore-Operationen ausgeführt werden können oder nicht – in den **Wartezustand** versetzt.  
Wenn der Semaphore-Wert `== 0` geworden ist, wird der Prozess wieder aufgeweckt.
    - Das Flag `IPC_NOWAIT` ist **gesetzt**.  
Die Funktion kehrt sofort mit dem Fehler **EAGAIN** zurück (Prozess blockiert nicht).  
Es wird keine der übrigen spezifizierten Operationen ausgeführt.

### • Anmerkungen : 1. Anwendungs-Beispiele :

```
int id = ... ;           /* Kennung eines Semaphore-Objekts */  
  
/* P-Operation (Down-Operation) auf Semaphore Nr.0 */  
struct sembuf dn_op[1] = {0, -1, SEM_UNDO};  
semop(id, dn_op, 1);  
  
/* V-Operation (Up-Operation) auf Semaphore Nr.0 */  
struct sembuf up_op[1] = {0, 1, SEM_UNDO};  
semop(id, up_op, 1);
```

2. Die API-Funktion `semtimedop(...)` verhält sich wie die Funktion `semop(...)`, mit dem Unterschied, dass eine Zeit spezifiziert werden kann, nachdem der aufrufende Prozess einen eventuellen Wartezustand spätestens verlässt (mit dem Fehler `EAGAIN`)

## LINUX-Demonstrations-Programm zu System V IPC Semaphore

```
// C-Quelldatei semdemo_m.c
// Programm semdemo
// Demonstration von System V Semaphore

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 4321
#define SEMPERM S_IRWXU|S_IRWXG|S_IRWXO

void errorExit(char* msg)
{ fprintf(stderr, "(errno : %d ) ", errno);
  perror(msg);
  exit(1);
}

int initsem(key_t skey)
{ int semid = semget(skey, 1, SEMPERM);
  if (semid!=-1)
    printf("Semaphore (id = %d) existiert !\n", semid);
  else
  { printf("Semaphore wird angelegt !\n");
    semid = semget(skey, 1, IPC_CREAT|IPC_EXCL|SEMPERM);
    if (semid==-1)
      errorExit("Anlegen Semaphore fehlgeschlagen !\n");
    printf("Semaphore (id = %d) wird initialisiert !\n", semid);
    if (semctl(semid, 0, SETVAL, 1) == -1)
      errorExit("Initialisierung Semaphore fehlgeschlagen !\n");
  }
  return semid;
}

void down(int semid) /* P-Operation */
{ static struct sembuf dn_op[1] = {0, -1, SEM_UNDO};
  if (semop(semid, dn_op, 1)) errorExit("Fehler bei down-Op !\n");
}

void up(int semid) /* V-Operation */
{ static struct sembuf up_op[1] = {0, 1, SEM_UNDO};
  if (semop(semid, up_op, 1)) errorExit("Fehler bei up-Op !\n");
}

int main(int argc, char *argv[])
{
  int semid = -1;
  semid = initsem(SEMKEY);
  while (1)
  { printf("PID : %d vor Eintritt kritischer Abschnitt !\n", getpid());
    down(semid);
    printf("PID : %d im kritischen Abschnitt !\n");
    sleep(10);
    up(semid);
    printf("PID : %d Ende kritischer Abschnitt !\n", getpid());
    sleep(5);
  }
  return 0;
}
```

## System V IPC – Message Queues (1)

### • Überblick

- ◇ **Message Queues** ermöglichen das **Übermitteln** einer aus einer beliebigen Bytefolge bestehenden **Nachricht** von einem **Sendeprozess** zu einem **Empfangsprozess**.
- ◇ Der Sendeprozess stellt die zu übermittelnde Nachricht (*message*) in eine Nachrichten-Warteschlange (**Message Queue**), aus der sie ein anderer Prozess (Empfangsprozess) auslesen kann.  
Eine Nachricht kann immer **nur** von **einem Prozess** empfangen werden. **Nach dem Auslesen** wird sie aus der Warteschlange **entfernt**.
- ◇ Eine nicht ausgelesene Nachricht verbleibt – unabhängig von der Lebensdauer des Sendeprozesses – solange in der Message Queue bis das Message-Queue-Objekt gelöscht wird.  
→ Eine Nachricht kann auch nach Beendigung des Sendeprozesses noch empfangen werden
- ◇ Jeder Nachricht ist eine positive Kennzahl als **Message-Typ** zugeordnet.  
Dieser kann z.B. zur **Priorisierung** von Nachrichten oder zur **Festlegung** des **Empfängers**, für den eine Nachricht bestimmt ist, verwendet werden (ein Sender, mehrere Empfänger). Er ermöglicht auch die **bidirektionale Kommunikation** zweier Prozesse über die gleiche Message Queue.
- ◇ Eine **neue Nachricht** wird immer an das **Ende** der Warteschlange gestellt.  
Ein **Auslesen** von Nachrichten unabhängig vom Message-Typ erfolgt nach dem **FIFO-Prinzip**.  
Ansonsten gilt das FIFO-Prinzip jeweils für das Auslesen von Nachrichten des gleichen Message-Typs.
- ◇ **Prozesse** können bei der Kommunikation über Message Queues **blockiert** werden :
  - Sendeprozesse, wenn die Kapazität der Message Queue zur Aufnahme der neuen Nachricht nicht ausreicht,
  - Empfangsprozesse, wenn eine erwartete Nachricht noch nicht vorhanden ist.→ Mit einem Message-Queue-Objekt können Prozess-Warteschlangen assoziiert sein.
- ◇ Durch **Kernel-Konstante** festgelegte **Maximalwerte** (definiert in `include/linux/msg.h`) :  
(die angegebenen Werte beziehen sich auf den Kernel 2.6.22.17)
  - ▷ **MSGMNI** – Maximalzahl gleichzeitig existierender Message-Queue-Objekte : **16**
  - ▷ **MSGMAX** – maximale Länge einer Message in Bytes (einschliesslich Message Header) : **65536**
  - ▷ **MSGMNB** – maximale Gesamtlänge einer Message Queue in Bytes (einschliesslich Message Header) : **65536**

### • Implementierung von Message-Queue-Objekten

- ◇ System V Message-Queue-Objekte werden im wesentlichen durch das Zusammenspiel der beiden folgenden Structure-Datentypen implementiert :
  - ▷ **struct msg\_queue**
  - ▷ **struct msg\_msg**

### • Structure-Typ **struct msg\_msg** (definiert in `include/linux/msg.h`)

- ◇ Dieser Typ kapselt eine **Nachricht**. Er beschreibt ein Element einer Nachrichtenwarteschlange  
Zur Verkettung der einzelnen Warteschlangen-Elemente besitzt er eine Komponente des Typs `struct list_head`
- ◇ Der Typ beschreibt genaugenommen lediglich den **Header** einer Nachricht (im wesentlichen **Message-Typ** und **-Länge**).  
Die eigentliche Nachricht folgt unmittelbar nach dem Header. Für Header und Nachricht wird immer eine Speicherseite (i.a. 4K Bytes) alloziert. Grössere Nachrichten können auf mehrere Speicherseiten verteilt werden.

◇

```
/* one msg_msg structure for each message */
struct msg_msg {
    struct list_head m_list; /* Verkettung der Warteschlangen-Elemente (Nachrichten) */
    long m_type;             /* Message-Type */
    int m_ts;                /* message text size */
    struct msg_msgseg* next; /* Pointer auf Speicherseite, die die Fortsetzung der Nachricht enthält */
    void *security;
    /* the actual message follows immediately */ /* Rest der Speicherseite enthält die Nachricht */
};
```

## System V IPC – Message Queues (2)

- **Structure-Typ `struct msg_queue`** (definiert in `include/linux/msg.h`)

- ◇ Dieser Typ dient zur **Verwaltung** einer Message Queue.
- ◇ Er enthält u.a. jeweils die Listenköpfe (`struct list_head`) für
  - ▷ die Nachrichtenwarteschlange
  - ▷ die Liste der wartenden Empfangsprozesse
  - ▷ die Liste der wartenden Sendeprozesse

```
/* one msg_queue structure for each present queue on the system */
struct msg_queue {
    struct kern_ipc_perm q_perm;
    int q_id;
    time_t q_stime;           /* last msgsnd time */
    time_t q_rtime;          /* last msgrcv time */
    time_t q_ctime;          /* last change time */
    unsigned long q_cbytes;   /* current number of bytes on queue */
    unsigned long q_qnum;     /* number of messages in queue */
    unsigned long q_qbytes;   /* max number of bytes on queue */
    pid_t q_lspid;            /* pid of last msgsnd */
    pid_t q_lrpid;            /* last receive pid */
    struct list_head q_messages; /* Nachrichtenwarteschlange */
    struct list_head q_receivers; /* Liste der wartenden Empfangsprozesse */
    struct list_head q_senders; /* Liste der wartenden Sendeprozesse */
};
```

- ◇ Die beiden Längenangaben (`q_cbytes` und `q_qbytes`) umfassen auch die jeweiligen Nachrichten-Header

- **Structure-Typen `struct msg_sender` und `struct msg_receiver`** (definiert in `ipc/msg.c`)

- ◇ Der Datentyp **`struct msg_sender`** beschreibt ein Element der Liste der **wartenden Sendeprozesse**
- ◇ Der Datentyp **`struct msg_receiver`** beschreibt ein Element der Liste der **wartenden Empfangsprozesse**

```
/* one msg_sender for each sleeping sender */
struct msg_sender {
    struct list_head list; /* Listen-Verkettungspointer */
    struct task_struct* tsk; /* wartender Prozess */
};

/* one msg_receiver structure for each sleeping receiver */
struct msg_receiver {
    struct list_head r_list; /* Listen-Verkettungspointer */
    struct task_struct* r_tsk; /* wartender Prozess */
    int r_mode;
    long r_msgtype; /* Typ der Message, auf die gewartet wird */
    long r_maxsize; /* Groesse des Buffers zur Aufnahme der Nachricht */
    struct msg_msg* volatile r_msg; /* Pointer auf Buffer zur Aufnahme der Nachricht */
};
```

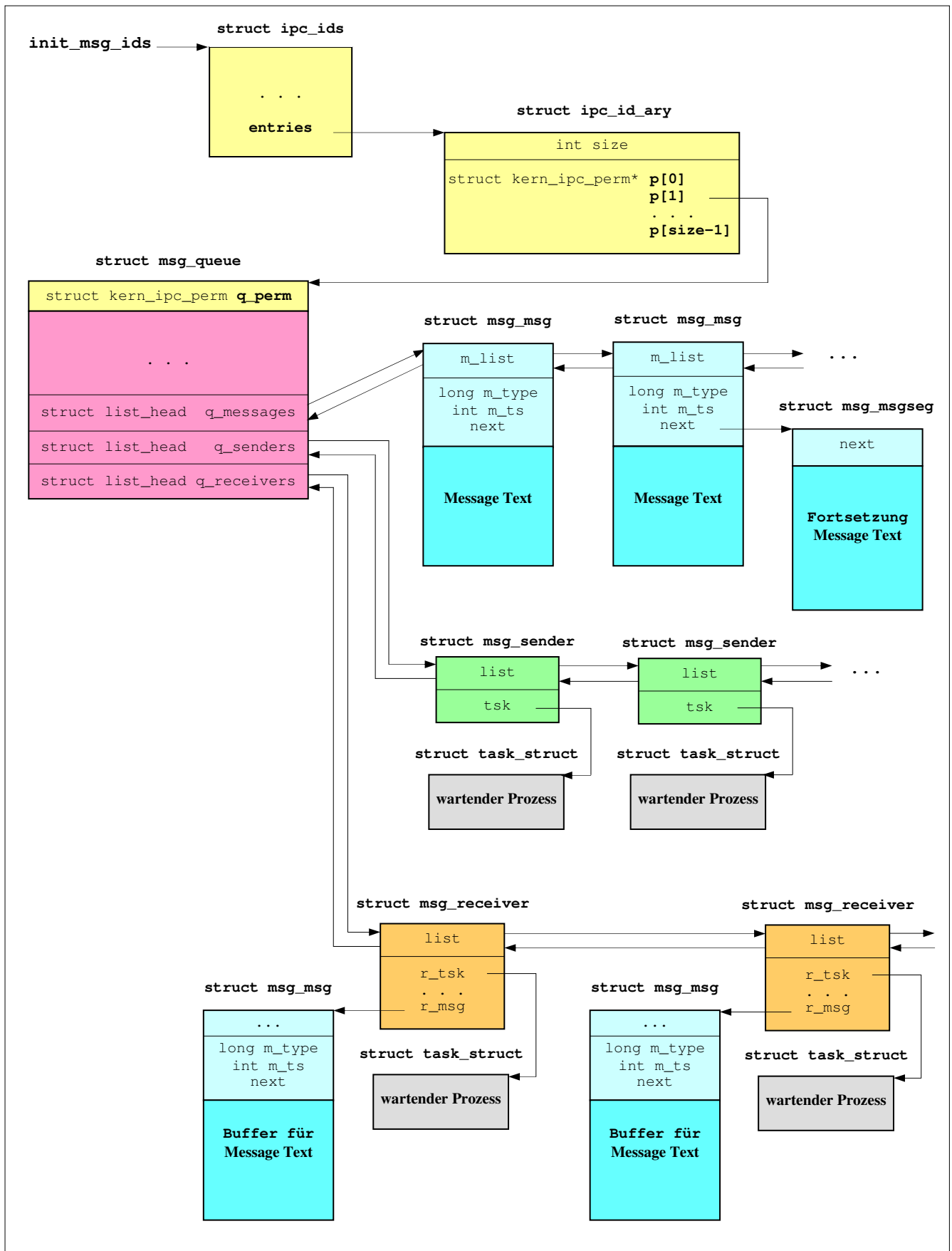
- **Structure-Typ `struct msg_msgseg`** (definiert in `ipc/msgutil.c`)

- ◇ Dieser Datentyp beschreibt den **Header** am Beginn einer **Speicherseite**, die die **Fortsetzung** einer Nachricht enthält.

```
struct msg_msgseg {
    struct msg_msgseg* next; /* Pointer auf nächste Fortsetzungsseite */
    /* the next part of the message follows immediately */
};
```

### System V IPC – Message Queues (3)

- Überblick über die Message-Queue-bezogenen Datenstrukturen des Kernels



## System V IPC API-Funktion `msgget`

- **Funktionalität :** Erzeugung einer neuen Message-Queue bzw Ermittlung der **Kennung** (identifizier) einer existierenden Message-Queue.  
Bei **Erfolg** Rückgabe der **Kennung** des – neu erzeugten bzw existierenden – Message-Queue-Objekts.

- **Interface :**

```
int msgget(key_t key, int msgflg)
```

- **Header-Dateien :** `<sys/ipc.h>` (symbolische Konstanten)  
`<sys/msg.h>` (Function Prototype)  
`<sys/stat.h>` (für Zugriffsberechtigungsflags)

- **Parameter :**

*key* **Schlüssel** (*key*) des Message-Queue-Objekts.

spezieller Wert : **IPC\_PRIVATE**

(def. in `<bits/ipc.h>`, eingebunden durch `<sys/ipc.h>`)

→ bewirkt, dass immer ein neues Message-Queue-Objekt erzeugt wird

*msgflg* **Steuerflags** und **Zugriffsberechtigungsflags** (bitweis-oder-verknüpft), bzw **0**

Die zulässigen **Steuerflags** sind :

(def. in `<bits/ipc.h>`, eingebunden durch `<sys/ipc.h>`).

**IPC\_CREAT** Erzeugung eines neuen Message-Queue-Objekts, falls noch keines mit dem angegebenen Schlüssel existiert (Ausnahme : Schlüssel `IPC_PRIVATE`), sonst Rückgabe der Kennung des bereits vorhandenen Objekts.

**IPC\_EXCL** Funktion endet fehlerhaft, falls ein neues Message-Queue-Objekt erzeugt werden soll und bereits eines mit dem angegebenen Schlüssel existiert

→ diese Flag ist nur zusammen mit dem Flag `IPC_CREAT` sinnvoll anwendbar

Die **Zugriffsberechtigungsflags** müssen nur angegeben werden, wenn ein neues Message-Queue-Objekt erzeugt werden soll.

Es handelt sich um die gleichen Flags, die auch bei der Erzeugung neuer Dateien verwendet werden (def. in `<sys/stat.h>`, s.a. System Call `creat()`), allerdings haben die Flags für die Ausführungs-Berechtigung hier keine Bedeutung

- **Rückgabewert :** - **Kennung** (*identifizier*) des Message-Queue-Objekts, bei Erfolg  
- **-1** im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117** (`ipc()`), Unterfunktion Nr. **13** (symb. Konst **MSGGET**)  
→ `sys_ipc(...)` (in `arch/i386/kernel/sys_386.c`)  
→ `sys_msgget(...)` (in `ipc/msg.c`)

- **Anmerkungen :** 1. Der Datentyp **key\_t** ist – indirekt – in den Header-Dateien `<sys/types.h>` und `<sys/ipc.h>` definiert als **int**.

2. **Anwendungs-Beispiele :**

```
#define MSGKEY 5432 /* symbolische Konstante für Objekt-Schlüssel */
#define MSGPERM 0666 /* Zugriffsberechtigungen : rw-rw-rw- */

/* Erzeugung einer neuen Message-Queue */
int id = msgget(MSGKEY, IPC_CREAT | IPC_EXCL | MSGPERM);

/* Ermittlung der Kennung einer existierenden Message-Queue */
int id = msgget(MSGKEY, 0);
```

## System V IPC – für die API-Funktion `msgctl` verwendete User-Code-Datenstrukturen

- **Structure-Typ `struct msqid_ds`** (definiert in `<bits/msq.h>`, eingebunden durch `<sys/msg.h>`)

◇ Dieser Datentyp dient zur Beschreibung eines **Message-Queue-Objekts** im User-Code.

◇

```
/* Types used in the structure definition. */
typedef unsigned long int msgqnum_t;
typedef unsigned long int msglen_t;

/* Structure of record for one message inside the kernel.
   The type `struct msg' is opaque. */
struct msqid_ds
{
    struct ipc_perm msg_perm;          /* structure describing operation permission */
    __time_t msg_stime;                /* time of last msgsnd command */
    unsigned long int __unused1;
    __time_t msg_rtime;                /* time of last msgrcv command */
    unsigned long int __unused2;
    __time_t msg_ctime;                /* time of last change */
    unsigned long int __unused3;
    unsigned long int __msg_cbytes;    /* current number of bytes on queue */
    msgqnum_t msg_qnum;                /* number of messages currently on queue */
    msglen_t msg_qbytes;                /* max number of bytes allowed on queue */
    __pid_t msg_lspid;                  /* pid of last msgsnd() */
    __pid_t msg_lrpid;                  /* pid of last msgrcv() */
    unsigned long int __unused4;
    unsigned long int __unused5;
};
```

◇ Der Datentyp `__time_t` ist definiert zu `long int`, der Typ `__pid_t` ist definiert zu `int` (durch das Zusammenspiel der Headerdateien `<bits/types.h>`, und `<bits/typesizes.h>`, diese werden durch die Headerdatei `<bits/msq.h>` eingebunden, diese wird ihrerseits durch `<sys/msg.h>` eingebunden)

◇ Der o.a. für die Komponente `msg_perm` verwendete Datentyp `struct ipc_perm` ist die User-Code-Version des Datentyps `struct kern_ipc_perm`.  
Er fasst für ein IPC-Objekt den Key, die Zugriffsrechte, Informationen über den Besitzer und den Erzeuger sowie Information zur Bildung/Ermittlung der Kennung zusammen.  
Seine Definition ist weiter oben (V-BS-A54-00) beschrieben.

- **Structure-Typ `struct msginfo`** (definiert in `<bits/msq.h>`, eingebunden durch `<sys/msg.h>`)

◇ Dieser –linux-spezifische – Datentyp dient zur Ablage einiger systemweiter **message-queue-bezogener Kernel-Parameter** und **Grenzwerte** im User-Code

◇

```
/* buffer for msgctl calls IPC_INFO, MSG_INFO */
struct msginfo
{
    int msgpool; /* bei IPC_INFO nicht verwendet, bei MSG_INFO : akt. Anz. von Message Queues */
    int msgmap; /* bei IPC_INFO nicht verwendet, bei MSG_INFO : akt. Anz. aller Messages in allen Queues */
    int msgmax; /* maximale Länge einer Message in Bytes (einschliesslich Message Header) */
    int msgmnb; /* maximale Anzahl Bytes in einer Message Queue (einschliesslich Message Header) */
    int msgmni; /* maximale Anzahl von Message Queues */
    int msgssz; /* nicht verwendet */
    int msgtql; /* bei IPC_INFO nicht verwendet */
    /* bei MSG_INFO : Gesamtzahl aller Bytes in allen Messages in allen Queues */
    unsigned short msgseg; /* nicht verwendet */
};
```

## System V IPC API-Funktion `msgctl` (1)

- **Funktionalität :** Ausführung von **Kontroll-Operationen** (Zustandsabfrage/-änderung, Löschung) auf einem Message-Queue-Objekt.  
Die auszuführende Operation wird durch einen Kommando-Parameter festgelegt.

- **Interface :**

```
int msgctl(int msqid, int cmd, struct msqid_ds* buf);
```

- **Header-Dateien :** `<sys/types.h>`

- `<sys/ipc.h>` (symbolische Konstanten)

- `<sys/msg.h>` (*Function Prototype*, Typdefinitionen und symbolische Konstanten)

- **Parameter :**

- msqid* **Kennung** (*identifier*) der Message Queue.

- bzw für Kommando **MSG\_STAT** : **Index** im kernel-internen Message-Queue-Obj-Pointer-Array

- cmd* Auszuführendes **Kommando** (Kommando-Parameter),

- zulässige Kommandos sind :

- (def. in `<bits/ipc.h>` bzw `<bits/msq.h>`, eingebunden durch `<sys/msg.h>`)

- IPC\_STAT

- IPC\_SET

- IPC\_RMID

- IPC\_INFO (linux-spezifisch)

- MSG\_INFO (linux-spezifisch)

- MSG\_STAT (linux-spezifisch)

- } s. gesonderte Beschreibung

- buf* Kommando-abhängiger **Ein-** bzw **Ausgabeparameter**.

- Der angegebene Typ **struct msqid\_ds\*** gilt nur für die Kommandos **IPC\_STAT**, **IPC\_SET** und **MSG\_STAT**.

- Für das Kommando **IPC\_RMID** ist der Parameter **bedeutungslos**, es kann der **NULL**-Pointer übergeben werden.

- Für die Kommandos **IPC\_INFO** und **MSG\_INFO** muss der Parameter vom Typ **struct msginfo\*** sein → **Type Cast** erforderlich

- **Rückgabewert :** - bei Erfolg kommando-abhängiger Wert :

- für IPC\_STAT, IPC\_SET und IPC\_RMID : **0**

- für IPC\_INFO u. MSG\_INFO : **Index** der höchsten belegten Komponente des kernel-internen Pointer-Arrays auf Message-Queue-Objekte

- für MSG\_STAT : **Kennung** des Message-Queue-Objekts, das durch Index *msqid* referiert wird

- **-1** im Fehlerfall, **errno** wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **14** (symb. Konst **MSGCTL**)

- `sys_ipc(...)` (in `arch/i386/kernel/sys_386.c`)

- `sys_msgctl(...)` (in `ipc/msg.c`)



## System V IPC API-Funktion `msgctl` (2)

- **Zulässige Kommandos** (zulässige Werte für den Parameter `cmd`) :

- IPC\_STAT** Ermittlung von Informationen über das Message-Queue-Objekt mit der Kennung `msqid`.  
Ablage der Informationen in der durch `buf` referierten `struct msqid_ds`-Variablen  
Der aufrufende Prozess muss Leserecht für das Message-Queue-Objekt besitzen
- IPC\_SET** Setzen einiger Komponenten der Kernel-Struktur (Typ `struct msg_queue`), die das Message-Queue-Objekt mit der Kennung `msqid` beschreibt. Die zu setzenden Werte werden der durch `buf` referierten `struct msqid_ds`-Variablen entnommen.  
Gesetzt werden können folgende Komponenten: `msg_qbytes`, `msg_perm.uid`, `msg_perm.gid` sowie die 9 niederwertigen Bits von `msg_perm.mode`.  
Die effektive UID des aufrufenden Prozesses muss mit der UID des Objekt-Erzeugers oder Objekt-Besitzers übereinstimmen oder es muss ein `root`-Prozess sein
- IPC\_RMID** Löschung der Message Queue mit der Kennung `msqid` und Aufwecken aller an der Message Queue-wartenden Empfangs- u. Sende-Prozesse.  
Die effektive UID des aufrufenden Prozesses muss mit der UID des Objekt-Erzeugers oder Objekt-Besitzers übereinstimmen oder es muss ein `root`-Prozess sein.  
Der Parameter `buf` wird ignoriert
- IPC\_INFO** (linux-spezifisch)  
erforderlich für Verwendung: `#define __USE_GNU` (vor `#include <sys/ipc.h>`)  
Ermittlung einiger systemweiter message-queue-bezogener Kernel-Parameter und Grenzwerte.  
Ablage dieser Info in der durch `buf` referierten `struct msginfo`-Variablen.  
Achtung: Type Cast für den Parameter `buf` erforderlich.
- MSG\_INFO** (linux-spezifisch),  
Ermittlung derselben Info wie bei `IPC_INFO` mit folgenden Änderungen :  
`buf->msgpool` wird auf die Anzahl der aktuell existierenden Message-Queue-Objekte,  
`buf->msgmap` wird auf die Anzahl der aktuell insgesamt (in allen Message Queues) vorhandenen Messages ,  
`buf->msgtql` wird auf die Gesamtzahl der Bytes in allen Messages in allen Queues gesetzt  
Achtung: Type Cast für den Parameter `buf` erforderlich.
- MSG\_STAT** (linux-spezifisch)  
Ermittlung von Informationen über ein Message-Queue-Objekt wie bei `IPC_STAT`.  
Der Parameter `msqid` wird jedoch nicht als Kennung des Objekts, sondern als Index des kernel-internen Pointer-Arrays auf Message-Queue-Objekte interpretiert.  
Der aufrufende Prozess muss Leserecht für das Message-Queue-Objekt besitzen

- **Anmerkungen :**    **Anwendungs-Beispiele :**

```
int id = ... ;           /* Kennung eines Message-Queue-Objekts */

/* Ermittlung message-queue-bezogener Kernel-Parameter und Grenzwerte */
struct msginfo minf;     /* Variable zur Aufnahme relevanter Kernel-Parameter */
msgctl(id, MSG_INFO, (struct msqid_ds*)&minf);

/* Ermittlung von Informationen über ein Message-Queue-Objekt */
struct msqid_ds mds;     /* Variable zur Aufnahme von Info über Message-Queue-Objekt */
msgctl(id, IPC_STAT, &mds);

/* Löschen eines Message-Queue-Objekts */
msgctl(id, IPC_RMID, NULL);
```

- **Funktionalität : Senden** einer **Message** an eine Message Queue.  
Die gesendete Message wird an das Ende der Message Queue angehängt.  
Der aufrufende Prozess muss Schreibberechtigung für die Message Queue haben

```
int msgsnd(int msqid, const void* msgp, size_t msgsz, int msgflg)
```

- ```
int id = ... ;           /* Kennung eines Message-Queue-Objekts */
int msglen;              /* Länge des Message-Textes */
struct mymsgbuf mbuf;    /* Message-Buffer */
strcpy(mbuf.mtext; info); /* info ist ein Pointer auf einen C-String */
msglen = strlen(mbuf.mtext);
msgsnd(id, &mbuf, msglen, 0);
```

## System V IPC API-Funktion **msgrcv**

- **Funktionalität :** **Auslesen** und **Entfernen** einer **Message** aus einer Message Queue.  
Die ausgelesene Message hängt von dem anzugebenden Message-Typ ab.  
Der aufrufende Prozess muss Leseberechtigung für die Message Queue haben

- **Interface :**

```
int msgrcv(int msqid, void* msgp, size_t msgsz, long msgtyp, int msgflg)
```

- **Header-Dateien :** **<sys/types, h>**

**<sys/ipc.h>** (symbolische Konstante)

**<sys/msg.h>** (Function Prototype, symbolische Konstante)

- **Parameter :**

*msqid* **Kennung (identifier)** der Message Queue

*msgp* Pointer auf einen Buffer, der die auszulesende Message aufnimmt.  
Dieser Buffer muss eine Variable eines Structure-Typs sein, der prinzipiell wie folgt aufgebaut ist :

```
struct msgbuf
{ long mtype;          /* message type, muss >0 sein) */
  char mtext[SIZE];    /* message data, es muss sein : SIZE == msgsz*/
};
```

*msgsz* Maximal zulässige Länge des Message-Textes (== Größe des Message-Data-Buffers)  
Messages der Länge 0 sind zulässig.

*msgtyp* legt den Typ der auszulesenden Message in folgender Weise fest :

▷ **==0** die erste Message in der Queue wird ausgelesen

▷ **>0** Flag **MSG\_EXCEPT** nicht gesetzt : die erste Message des Typs *msgtyp* wird ausgelesen

Flag **MSG\_EXCEPT** gesetzt : die erste Message, die nicht vom Typ *msgtyp* ist, wird ausgelesen

▷ **<0** die erste Message mit dem kleinsten Typ  $\leq \text{abs}(\text{msgtyp})$  wird ausgelesen

*msgflg* Steuerflags, gegebenenfalls bitweis-oder-verknüpft (oder 0)

Zulässige **Flags** sind :

▷ **IPC\_NOWAIT** Wenn die Message Queue keine Message des angegeben Typs enthält, bewirkt dieses Flag, dass der aufrufende Prozess nicht in den Wartezustand versetzt wird (und die Funktion blockiert) , sondern die Funktion mit einem Fehler (**ENOMSG**) endet

▷ **MSG\_EXCEPT** Wirkung s. Beschreibung des Parameters *msgtyp*

▷ **MSG\_NOERROR** Wenn die Länge der auszulesenden Message größer als *msgsz* ist, bewirkt dieses Flag, dass die Message auf *msgsz* begrenzt wird und die Funktion nicht mit einem Fehler (E2BIG) endet.

Der abgeschnittene Teil der Message geht verloren.

- **Rückgabewert :** - die Länge des tatsächlich ausgelesenen Message-Textes, bei Erfolg  
- **-1** im Fehlerfall, *errno* wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117 (ipc())** , Unterfunktion Nr. **12** (symb. Konst **MSGRCV**)

→ *sys\_ipc(...)* (in *arch/i386/kernel/sys\_386.c*)

→ *sys\_msgrcv(...)* (in *ipc/msg.c*)

→ *do\_msgrcv(...)* (in *ipc/msg.c*)

- **Anmerkungen :** **Anwendungs-Beispiel :** analog zu dem Beispiel zur API-Funktion *msgsnd()*

## LINUX-Demonstrations-Programme zu System V IPC Message Queues (1)

### • Sendeprogramm msgreceiver

```
// C-Quelldatei msgreceiver_m.c
// Programm msgreceiver
// Demonstration der System V Message Queue
// hier : Empfaenger-Programm, erzeugt Message Queue und liest in einer Schleife aus
//        der Queue,
//        die gelesenen Messages werden in die Standardausgabe ausgegeben
// Wenn "shutdown" als Message empfangen wird, vernichtet das Programm die
// Message Queue und beendet sich dann

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <errno.h>

#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY 5432
#define MSGPERM S_IRWXU|S_IRWXG|S_IRWXO

#define BUFSIZE 256

struct mymsgbuf
{ long int mtype;
  char mtext[BUFSIZE];
};

int main(int argc, char *argv[])
{
    int iRet = 0;
    key_t mkey = MSGKEY;
    int msgid = -1;
    struct mymsgbuf mbuf;
    int msglen = 0;
    int shdn = 0;

    if ((msgid = msgget(mkey, IPC_CREAT|IPC_EXCL|MSGPERM)) == -1)
    { printf("\nErzeugung Message Queue fehlgeschlagen, errno = %d (%s)\n",
        errno, strerror(errno));
        iRet=1;
    }
    else
    { printf("\nMessage Queue erzeugt, ID = %d\n", msgid);
        while (!shdn && (msglen=msgrcv(msgid, &mbuf, BUFSIZE-1, 0, MSG_NOERROR)) >=0)
        { mbuf.mtext[msglen]='\0';
            if (strcmp(mbuf.mtext, "shutdown")==0) shdn = 1;
            printf ("\nempfangen : %s\n", mbuf.mtext);
        }
        msgctl(msgid, IPC_RMID, NULL);
    }
    return iRet;
}
```

## LINUX-Demonstrations-Programme zu System V IPC Message Queues (2)

- Sendeprogramm **msgsender**

```
// C-Quelldatei msgsender_m.c
// Programm msgsender
// Demonstration der System V Message Queue
// hier : Sender-Programm, ermittelt Message Queue Identifier
//        und schreibt in einer Schleife in die Queue,
//        die zu schreibenden Messages werden von der Standardeingabe eingelesen.

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <errno.h>

#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY  5432

#define BUFSIZE 256

struct mymsgbuf
{ long int mtype;
  char mtext[BUFSIZE];
};

int main(int argc, char *argv[])
{
    int iRet = 0;
    key_t mkey = MSGKEY;
    int msgid = -1;
    struct mymsgbuf mbuf;
    mbuf.mtype = getpid();
    int msglen = 0;

    if ((msgid = msgget(mkey, 0)) == -1)
    { printf("\nMessage Queue mit Key %d existiert nicht\n", mkey);
      iRet=1;
    }
    else
    { printf("\nMessage Queue vorhanden, ID = %d\n", msgid);
      while (putchar('?'), putchar(' '), fgets(mbuf.mtext, BUFSIZE, stdin) !=NULL)
      { msglen=strlen(mbuf.mtext);
        if (mbuf.mtext[msglen-1]=='\n') mbuf.mtext[--msglen]='\0';
        msgsnd(msgid, &mbuf, msglen, 0);
        printf ("Message gesendet : %s\n", mbuf.mtext);
      }
    }
    return iRet;
}
```

## System V IPC – Shared Memory (1)

### • Überblick

- ◇ **Shared Memory** stellt den schnellsten IPC-Mechanismus zur Verfügung.  
Zwei oder mehr Prozesse blenden ein und denselben Speicherbereich (=Shared Memory Segment) in ihren jeweiligen Adressraum ein und können damit gemeinsam – unter Berücksichtigung der jeweiligen Zugriffsberechtigungen – direkt zu diesem Speicherbereich zugreifen.  
Jeglicher Umweg über Funktionsaufrufe, Pufferbereiche und Kopieren von Daten entfällt.
- ◇ Ein **konkurrierender Zugriff** mehrerer Prozesse zum selben Speicherbereich muss **synchronisiert** werden.  
Hierfür werden meist Semaphore eingesetzt.
- ◇ Durch **Kernel-Konstante** festgelegte **Maximalwerte** (definiert in `include/linux/shm.h`):  
(die angegebenen Werte beziehen sich auf den Kernel 2.6.22.17)
  - ▷ **SHMMNI** – Maximalzahl gleichzeitig existierender Shared Memory Segmente : **4096**
  - ▷ **SHMMAX** – maximale Größe eines Shared Memory Segments (in Bytes) : **ULONG\_MAX** (==4294967295)
  - ▷ **SHMMIN** – minimale Größe eines Shared Memory Segments (in Bytes) : **1**
  - ▷ **SHMALL** – max. Gesamtgröße aller Shared Memory Segmente (in Pages) : **269435200** (==0xfffff00)
  - ▷ **SHMSEG** – Maximalzahl der Shared Memory Segmente pro Prozess : **4096**

### • Implementierung von Shared-Memory-Objekten

- ◇ System V Shared Memory Segmente werden unter Mitwirkung des **Virtuellen File Systems** (VFS, s. Externe Datenverwaltung) implementiert.  
Jedes Shared Memory Segment wird durch eine **Pseudo-Datei**, die in den physikalischen Arbeitsspeicher eingeblendet (*mapped*) ist, realisiert.
- ◇ Der Einbettung in das gemeinsame Konzept der System V IPC-Mechanismen dient der folgende Structure-Datentyp :
  - ▷ **struct shmid\_kernel**
- ◇ Zur Implementierung der Pseudo-Dateien werden im wesentlichen die **Datenstrukturen** des **Virtuellen File Systems** eingesetzt. Die wichtigsten dieser Datenstrukturen sind :
  - ▷ **struct file**
  - ▷ **struct dentry**
  - ▷ **struct inode**
  - ▷ **struct address\_space** (definiert in `include/linux/fs.h`,  
Adressraum-Objekt, benötigt für das Einblenden von Dateien in den Speicher-  
Adressraum, enthält u.a. die Informationen über die vom Shared Memory Segment  
belegten physikalischen Seiten)

### • Structure-Typ **struct shmid\_kernel** (definiert in `include/linux/shm.h`)

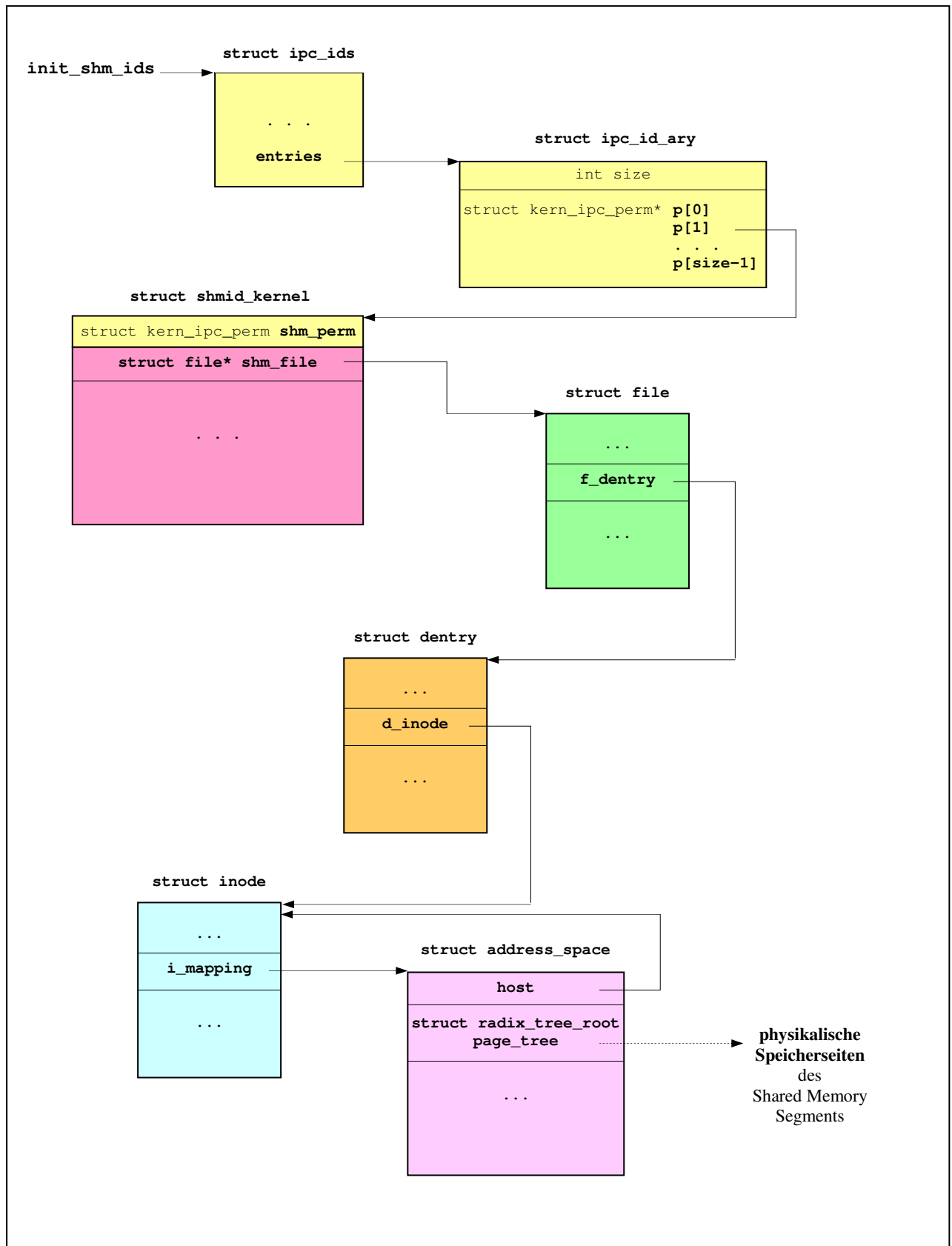
- ◇ Dieser Datentyp bildet den Ausgangspunkt für die Verwaltung von Shared Memory Segmenten.  
Diese Verwaltung erfolgt analog zur Verwaltung der anderen System V IPC Objekt-Typen.

◇

```
struct shmid_kernel /* private to the kernel */
{ struct kern_ipc_perm  shm_perm;
  struct file *         shm_file;
  int                   id;
  unsigned long         shm_nattch;
  unsigned long         shm_segsz;
  time_t                shm_atim;
  time_t                shm_dtim;
  time_t                shm_ctim;
  pid_t                 shm_cprid;
  pid_t                 shm_lprid;
  struct user_struct    *mlock_user;
};
```

## System V IPC – Shared Memory (2)

- Überblick über die Datenstrukturen des Kernels zur Implementierung von Shared Memory Segmenten



## System V IPC API-Funktion **shmget**

- **Funktionalität :** Erzeugung eines neuen Shared Memory Segments bzw Ermittlung der Kennung (identifizier) eines existierenden Shared Memory Segments.

Bei **Erfolg** Rückgabe der **Kennung** des – neu erzeugten bzw existierenden – Shared Memory Segments.

- **Interface :**

```
int shmget(key_t key, int size, int shmflg)
```

- **Header-Dateien :** **<sys/ipc.h>** (symbolische Konstanten)  
**<sys/shm.h>** (Function Prototype)  
**<sys/stat.h>** (für Zugriffsberechtigungsflags)

- **Parameter :**

*key* **Schlüssel** (*key*) des Shared-Memory-Objekts.

spezieller Wert : **IPC\_PRIVATE**

(def. in *<bits/ipc.h>*, eingebunden durch *<sys/ipc.h>*)

→ bewirkt, dass immer ein neues Shared-Memory-Objekt erzeugt wird

*size* **minimale Größe** des Shared Memory Segments, die tatsächliche Größe wird auf das nächste Vielfache der Seitengröße (==PAGE\_SIZE) aufgerundet.

Für die Ermittlung der Kennung eines existierenden Segments kann 0 angegeben werden.

*shmflg* **Steuerflags und Zugriffsberechtigungsflags** (bitweis-oder-verknüpft), bzw 0

Die zulässige **Steuerflags** sind :

(def. in *<bits/ipc.h>*, eingebunden durch *<sys/ipc.h>*).

**IPC\_CREAT** Erzeugung eines neuen Shared-Memory-Objekts, falls noch keines mit dem angegebenen Schlüssel existiert (Ausnahme : Schlüssel **IPC\_PRIVATE**), sonst Rückgabe der Kennung des bereits vorhandenen Objekts.

**IPC\_EXCL** Funktion endet fehlerhaft, falls ein neues Shared-Memory-Objekt erzeugt werden soll und bereits eines mit dem angegebenen Schlüssel existiert

→ diese Flag ist nur zusammen mit dem Flag **IPC\_CREAT** sinnvoll anwendbar

Die **Zugriffsberechtigungsflags** müssen nur angegeben werden, wenn ein neues Shared-Memory-Objekt erzeugt werden soll.

Es handelt sich um die gleichen Flags, die auch bei der Erzeugung neuer Dateien verwendet werden (def. in *<sys/stat.h>*, s.a. System Call **creat()**), allerdings haben die Flags für die Ausführungs-Berechtigung hier keine Bedeutung

- **Rückgabewert :** - **Kennung** (*identifizier*) des Shared-Memory-Objekts, bei Erfolg  
- **-1** im Fehlerfall, **errno** wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **23** (symb. Konst **SHMGET**)

→ **sys\_ipc(...)** (in *arch/i386/kernel/sys\_386.c*)

→ **sys\_shmget(...)** (in *ipc/shm.c*)

- **Anmerkungen :** 1. Der Datentyp **key\_t** ist – indirekt – in den Header-Dateien *<sys/types.h>* und *<sys/ipc.h>* definiert als **int**.

- 2. **Anwendungs-Beispiele :**

```
#define SHMKEY 7654 /* symbolische Konstante für Objekt-Schlüssel */
```

```
#define SHMPERM 0666 /* Zugriffsberechtigungen : rw-rw-rw- */
```

```
/* Erzeugung eines neuen Shared Memory Segments */
```

```
int shmsize = 4096;
```

```
int id = shmget(SHMKEY, shmsize, IPC_CREAT|IPC_EXCL|SHMPERM);
```

```
/* Ermittlung der Kennung eines existierenden Shared Memory Segments */
```

```
int id = shmget(SHMKEY, 0, 0);
```



## System V IPC – für die API-Funktion `shmctl` verwendete User-Code-Datenstrukturen

- **Structure-Typ `struct shmid_ds`** (definiert in `<bits/shm.h>`, eingebunden durch `<sys/shm.h>`)

◇ Dieser Datentyp dient zur Beschreibung eines **Shared-Memory-Objekts** im **User-Code**.

◇

```
/* Type to count number of attaches. */
typedef unsigned long int shmatt_t;

/* Data structure describing a set of semaphores. */ // richtig : shared memory segment
struct shmid_ds
{ struct ipc_perm shm_perm;          /* operation permission struct */
  size_t shm_segsz;                  /* size of segment in bytes */
  __time_t shm_atime;                 /* time of last shmat() */
  unsigned long int __unused1;
  __time_t shm_dtime;                 /* time of last shmdt() */
  unsigned long int __unused2;
  __time_t shm_ctime;                 /* time of last change by shmctl() */
  unsigned long int __unused3;
  __pid_t shm_cpid;                   /* pid of creator */
  __pid_t shm_lpid;                   /* pid of last shmop */ // shmat() bzw shmdt()
  shmatt_t shm_nattch;                /* number of current attaches */
  unsigned long int __unused4;
  unsigned long int __unused5;
};
```

- ◇ Der Datentyp `__time_t` ist definiert zu `long int`, der Typ `__pid_t` ist definiert zu `int` (durch das Zusammenspiel der Headerdateien `<bits/types.h>`, und `<bits/typesizes.h>`, diese werden durch die Headerdatei `<bits/shm.h>` eingebunden, diese wird ihrerseits durch `<sys/shm.h>` eingebunden)
- ◇ Der o.a. für die Komponente `sem_perm` verwendete Datentyp `struct ipc_perm` ist die User-Code-Version des Datentyps `struct kern_ipc_perm`. Er fasst für ein IPC-Objekt den Key, die Zugriffsrechte, Informationen über den Besitzer und den Erzeuger sowie Information zur Bildung/Ermittlung der Kennung zusammen. Seine Definition ist weiter oben (V-BS-A54-00) beschrieben.

- **Structure-Typ `struct shminfo`** (definiert in `<bits/shm.h>`, eingebunden durch `<sys/shm.h>`)

◇ Dieser –linux-spezifische – Datentyp dient zur Ablage einiger systemweiter **Shared-Memory-bezogener Kernel-Parameter** und **Grenzwerte** im **User-Code**. Er wird für das Kommando `IPC_INFO` verwendet

◇

```
struct shminfo
{ unsigned long int shmmax;           /* maximale Groesse eines Shared Memory Segments */
  unsigned long int shmmmin;          /* minimale Groesse eines Shared Memory Segments, immer == 1 */
  unsigned long int shmmni;           /* maximale Anzahl von Shared Memory Segmenten */
  unsigned long int shmseg;           /* max. Anzahl von SHM-Segmenten pro Prozess, nicht verwendet */
  unsigned long int shmall;           /* max. Gesamtgroesse aller Shared Memory Segmente (in Pages) */
  unsigned long int __unused[4];      /* tatsächlich definiert als 4 einzelne Komponenten */
};
```

- **Structure-Typ `struct shm_info`** (definiert in `<bits/shm.h>`, eingebunden durch `<sys/shm.h>`)

- ◇ Dieser –linux-spezifische – Datentyp dient zur Ablage einiger Informationen über durch Shared Memory verbrauchte System-Ressourcen. Er wird für das Kommando `SHM_INFO` verwendet.
- ◇ Die Definition dieses Datentyps ist bei der Beschreibung der API-Funktion `shmctl()` (Kommando `SHM_INFO`) angegeben (V-BS-A58-06)

## System V IPC API-Funktion **shmctl** (1)

- **Funktionalität :** Ausführung von **Kontroll-Operationen** (Zustandsabfrage/-änderung, Löschung) auf einem Shared Memory Segment.  
Die auszuführende Operation wird durch einen Kommando-Parameter festgelegt.

- **Interface :**

```
int shmctl(int shmid, int cmd, struct shmid_ds* buf);
```

- **Header-Dateien :** **<sys/types.h>**

**<sys/ipc.h>** (symbolische Konstanten)

**<sys/shm.h>** (*Function Prototype*, Typdefinitionen und symbolische Konstanten)

- **Parameter :**

*shmid* **Kennung** (*identifier*) des Shared Memory Segments.  
bzw für Kommando **SHM\_STAT** : **Index** im kernel-internen Shared-Memory-Obj-Pointer-Array

*cmd* Auszuführendes **Kommando** (Kommando-Parameter),  
zulässige Kommandos sind :  
(def. in *<bits/ipc.h>* bzw *<bits/shm.h>*, eingebunden durch *<sys/shm.h>*)

IPC\_STAT

IPC\_SET

IPC\_RMID

IPC\_INFO (linux-spezifisch)

SHM\_INFO (linux-spezifisch)

SHM\_STAT (linux-spezifisch)

SHM\_LOCK (linux-spezifisch)

SHM\_UNLOCK (linux-spezifisch)

} s. gesonderte Beschreibung

*buf* Kommando-abhängiger **Ein-** bzw **Ausgabeparameter**.

- Der angegebene Typ **struct shmid\_ds\*** gilt nur für die Kommandos **IPC\_STAT**, **IPC\_SET** und **SHM\_STAT**.
- Für die Kommandos **IPC\_RMID**, **SHM\_LOCK** und **SHM\_UNLOCK** ist der Parameter **bedeutungslos**, es kann der **NULL**-Pointer übergeben werden.
- Für das Kommando **IPC\_INFO** muss der Parameter vom Typ **struct shminfo\*** sein → **Type Cast** erforderlich
- Für das Kommandos **SHM\_INFO** muss der Parameter vom Typ **struct shm\_info\*** sein → **Type Cast** erforderlich

- **Rückgabewert :** - bei Erfolg kommando-abhängiger Wert :

für IPC\_STAT, IPC\_SET, IPC\_RMID, SHM\_LOCK und SHM\_UNLOCK : **0**

für IPC\_INFO u. SHM\_INFO : **Index** der höchsten belegten Komponente des kernel-internen Pointer-Arrays auf Shared Memory Segmente

für SHM\_STAT : **Kennung** des Shared Memory Segments, das durch den Index *shmid* referiert wird

- **-1** im Fehlerfall, **errno** wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **24** (symb. Konst **SHMCTL**)  
→ **sys\_ipc(...)** (in *arch/i386/kernel/sys\_386.c*)  
→ **sys\_shmctl(...)** (in *ipc/shm.c*)

## System V IPC API-Funktion `shmctl` (2)

- **Zulässige Kommandos** (zulässige Werte für den Parameter `cmd`) :

- IPC\_STAT** Ermittlung von Informationen über das Shared Memory Objekt mit der Kennung `shmid`.  
Ablage der Informationen in der durch `buf` referierten `struct shmid_ds`-Variablen  
Der aufrufende Prozess muss Leserecht für das Shared Memory Objekt besitzen
- IPC\_SET** Setzen einiger Komponenten der Kernel-Struktur (Typ `struct shmid_kernel`), die das Shared Memory Objekt mit der Kennung `shmid` beschreibt. Die zu setzenden Werte werden der durch `buf` referierten `struct shmid_ds`-Variablen entnommen.  
Gesetzt werden können folgende Komponenten: `shm_perm.uid`, `shm_perm.gid` sowie die 9 niederwertigen Bits von `shm_perm.mode`.  
Die effektive UID des aufrufenden Prozesses muss mit der UID des Erzeugers oder des Besitzers des Shared Memory Segments übereinstimmen oder es muss ein `root`-Prozess sein
- IPC\_RMID** Markieren des Shared Memory Segments mit der Kennung `shmid` zum Löschen.  
Das Segment wird erst dann tatsächlich gelöscht, wenn der letzte Prozess es aus seinem Adressraum ausgeblendet hat  
Die effektive UID des aufrufenden Prozesses muss mit der UID des Erzeugers oder des Besitzers des Shared Memory Segments übereinstimmen oder es muss ein `root`-Prozess sein  
Der Parameter `buf` wird ignoriert
- IPC\_INFO** (linux-spezifisch)  
erforderlich für Verwendung: `#define __USE_GNU` (vor `#include <sys/ipc.h>`)  
Ermittlung einiger systemweiter shared-memory-bezogener Kernel-Parameter und Grenzwerte.  
Ablage dieser Info in der durch `buf` referierten `struct shminfo`-Variablen.  
Achtung : Type Cast für den Parameter `buf` erforderlich.
- SHM\_INFO** (linux-spezifisch),  
Ermittlung von Informationen über von Shared Memory verbrauchte System-Ressourcen :  
Ablage dieser Info in der durch `buf` referierten `struct shm_info`-Variablen  
Achtung : Type Cast für den Parameter `buf` erforderlich.  
Der Datentyp `struct shm_info` ist wie folgt definiert :
- ```
struct shm_info
{ int used_ids; /* Anzahl der aktuell existierenden Shared Memory Segmente */
  unsigned long int shm_tot; /* Gesamtzahl der Speicherseiten für Shared Memory */
  unsigned long int shm_rss; /* Anzahl der residenten Shared Memory Speicherseiten */
  unsigned long int shm_swp; /* Anzahl der ausgelagerten Shared Memory Speicherseiten */
  unsigned long int swap_attempts; /* seit Linux 2.4 nicht verwendet */
  unsigned long int swap_successes; /* seit Linux 2.4 nicht verwendet */
};
```
- SHM\_STAT** (linux-spezifisch)  
Ermittlung von Informationen über ein Shared Memory Segment wie bei `IPC_STAT`.  
Der Parameter `shmid` wird jedoch nicht als Kennung des Segments, sondern als Index des kernel-internen Pointer-Arrays auf Shared-Memory-Objekte interpretiert.  
Der aufrufende Prozess muss Leserecht für das Shared-Memory-Objekt besitzen
- SHM\_LOCK** (linux-spezifisch)  
Blockieren des Swappings des Shared Memory Segments mit der Kennung `shmid`  
Der Parameter `buf` wird ignoriert
- SHM\_UNLOCK** (linux-spezifisch)  
Freigabe des Swappings des Shared Memory Segments mit der Kennung `shmid`  
Der Parameter `buf` wird ignoriert

- **Anmerkungen :**    **Anwendungs-Beispiele :**  
                                 analog zu den Beispielen zur API-Funktion `msgctl()`

- **Funktionalität : Einblenden** (*attach*) eines **Shared Memory Segments** in den **Speicheradressraum** eines Prozesses.  
Ein Einblenden ist nur an Seitengrenzen möglich.

```
void* shmat(int shmid, const void* shmaddr, int shmflg)
```

- ▷ **SHM\_RND** Bei Angabe einer Nicht-Seitenadresse wird die Einblendadresse auf die nächst niedrigere Seitenadresse abgerundet
- ▷ **SHM\_RDONLY** Das Shared Memory Segment wird **read-only** eingeblendet. Der aufrufende Prozess muss Leserecht für das Segment besitzen. Wird das Flag nicht gesetzt, erfolgt das Einblenden für Lesen und Schreiben. In diesem Fall muss der aufrufende Prozess Lese- und Schreibrecht besitzen.
- ▷ **SHM\_REMAP** (linux-spezifisch) Das Einblenden an einer vom Prozess bereits verwendeten Adresse ist zulässig. Der vorher an dieser Adresse eingeblendet gewesene Speicherbereich wird durch das neu eingeblendete Shared Memory Segment ersetzt.

- **Implementierung** : mittels System Call Nr. **117** (**ipc()**), Unterfunktion Nr. **21** (symb. Konst. **SHMAT**)
  - `sys_ipc(...)` (in `arch/i386/kernel/sys_386.c`)
  - `sys_shmat(...)` (in `ipc/shm.c`)
  - `do_shmat(...)` (in `ipc/shm.c`)

- ```
int id = ... ; /* Kennung eines Shared Memory Segments */
void* pshm;

/* Einblenden zum Lesen und Schreiben, Einblend-Adresse vom System festgelegt */
if ((pshm = shmat(shmid, NULL, 0)) != (void*)-1)
    // ..
```

## System V IPC API-Funktion **shmdt**

- **Funktionalität :** Ausblenden (*detach*) eines Shared Memory Segments aus dem Speicheradressraum eines Prozesses.  
Wenn das auszublendende Segment zum Löschen markiert war (API-Funktion `sysctl(...)` und durch keinen anderen Prozess mehr eingeblendet ist, wird das Segment auch gelöscht.
- **Interface :**

```
void* shmdt(const void* shmaddr)
```

  - **Header-Dateien :** `<sys/types,h>`  
`<sys/shm.h>` (*Function Prototype, symbolische Konstante*)
  - **Parameter :**  
*shmaddr* Adresse, ab der das auszublendende Shared Memory Segment bisher eingeblendet war.  
Der angegebene Wert muss als Funktionswert von `shmat(...)` zurückgegeben worden sein
  - **Rückgabewert :** - `0` , bei Erfolg  
- `(void*)-1` im Fehlerfall, `errno` wird entsprechend gesetzt
- **Implementierung :** mittels System Call Nr. **117** (`ipc()`), Unterfunktion Nr. **22** (symb. Konst **SHMDT**)
  - `sys_ipc(...)` (in `arch/i386/kernel/sys_386.c`)
  - `sys_shmdt(...)` (in `ipc/shm.c`)
- **Anmerkungen :**
  1. Nach einem `execv()` System Call und infolge eines `_exit()` System Calls werden alle Shared Memory Segmente des Prozesses ausgeblendet.
  2. **Anwendungs-Beispiel :**

```
void* pshm = ... ;      /* Einblend-Adresse eines Shared Memory Segments */  
  
/* Ausblenden des Shared Memory Segments */  
shmdt (pshm);
```

## LINUX-Demonstrations-Programme zu System V IPC Shared Memory (1)

### • Programm shmreader

```
// C-Quelldatei shmreader_m.c
// Programm shmreader
// Demonstration des System V Shared Memory
// hier : Reader-Programm, erzeugt Shared Memory und liest in einer Schleife aus
//        diesem, die gelesenen Messages werden in die Standardausgabe ausgegeben.
//        Wenn "shutdown" gelesen wird, vernichtet das Programm das Shared Memory
//        und beendet sich dann
// Zur Synchronisation zwischen Reader- u. Writer-Programm dient das 1. Byte
// im Shared Memory

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <errno.h>

#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 7654
#define SHMPERM S_IRWXU|S_IRWXG|S_IRWXO
#define SHMSIZE 1024
#define BUFSIZE 256

struct shmtextbuf          // Typ der Datenstruktur im Shared Memory
{ volatile int cflag;      // Change-Flag (fuer Synchronisation)
  volatile char mtext[BUFSIZE];
};

int main(int argc, char *argv[])
{ int iRet = 0;
  key_t smkey = SHMKEY;
  int shmsize = SHMSIZE;
  int shmid = -1;
  void* pshm;
  struct shmtextbuf* ptb = NULL;
  int txtlen = 0;
  int shdn = 0;
  if ((shmid = shmget(smkey, shmsize, IPC_CREAT|IPC_EXCL|SHMPERM))== -1)
  { printf("\nErzeugung Shared Memory fehlgeschlagen, errno = %d (%s)\n",
    errno, strerror(errno));
    iRet=1;
  }
  else
  { printf("\nShared Memory erzeugt, ID = %d\n", shmid);
    if ((pshm = shmat(shmid, NULL, 0))!=(void*)(-1))
    { printf("Shared Memory eingeblendet, Adr = %lx\n", (unsigned long)pshm);
      memset(pshm, '\0', shmsize);
      ptb = (struct shmtextbuf*)pshm;
      while (!shdn)
      { while (!ptb->cflag) ;
        if (strcmp(ptb->mtext, "shutdown")==0) shdn = 1;
        printf ("ngelesen : %s\n", ptb->mtext);
        ptb->cflag = 0;
      }
      shmdt (pshm);
    }
    shmctl(shmid, IPC_RMID, NULL);
  }
  return iRet;
}
```

## LINUX-Demonstrations-Programme zu System V IPC Shared Memory (2)

### • Programm shmwriter

```
// C-Quelldatei shmwriter_m.c
// Programm shmwriter
// Demonstration des System V Shared Memory
// hier : Sender-Programm, ermittelt Shared Memory Identifier,
//        blendet das Shared Memory in seinen Adressraum ein
//        und schreibt in das Shared Memory
// Zur Synchronisation zwischen Reader- u. Writer-Programm dient das 1. Byte
// im Shared Memory

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <errno.h>

#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 7654
#define SHMSIZE 1024
#define BUFSIZE 256

struct shmtextbuf          // Typ der Datenstruktur im Shared Memory
{ volatile int cflag;      // Change-Flag (fuer Synchronisation)
  volatile char mtext[BUFSIZE];
};

int main(int argc, char *argv[])
{ int iRet = 0;
  key_t smkey = SHMKEY;
  int shmsize = SHMSIZE;
  int shmid = -1;
  void* pshm;
  struct shmtextbuf* ptb = NULL;
  int txtlen = 0;
  int shdn = 0;
  if ((shmid = shmget(smkey, shmsize, 0))== -1)
  { printf("\nShared Memory mit Key %d existiert nicht\n", smkey);
    iRet=1;
  }
  else
  { printf("\nShared Memory vorhanden, ID = %d\n", shmid);
    if ((pshm = shmat(shmid, NULL, 0))!=(void*)(-1))
    { printf("Shared Memory eingeblendet, Adr = %lx\n", pshm);
      ptb = (struct shmtextbuf*)pshm;
      while(ptb->cflag);
      while (!shdn &&(putchar('?'), putchar(' '),
                          fgets(ptb->mtext, BUFSIZE, stdin) !=NULL))
      { txtlen=strlen(ptb->mtext);
        if (ptb->mtext[txtlen-1]=='\n') ptb->mtext[--txtlen]='\0';
        if (strcmp(ptb->mtext, "shutdown")==0) shdn = 1;
        ptb->cflag=1;
        printf ("geschrieben : %s\n", ptb->mtext);
        while(ptb->cflag);
      }
      shmdt (pshm) ;
    }
  }
  return iRet;
}
```

# **Betriebssysteme**

## **Kapitel 11**

### **11. Socket-Schnittstelle in LINUX**

11.1. Überblick über das Socket-API

11.2. UNIX Domain Sockets

11.3. INET Sockets



## Sockets in LINUX – Grundlagen (1)

### • Allgemeines

- ◇ Linux stellt auch das **BSD Socket API** zur Verfügung.  
Dieses kann als die heutige **Standard-Schnittstelle** für die **Programmierung** von **Netzwerkverbindungen** betrachtet werden. ⇒ Sockets ermöglichen eine **Interprozesskommunikation (IPC)** **über Rechengrenzen hinweg**. Sie lassen sich aber **auch** als **lokaler IPC-Mechanismus** innerhalb eines Rechners einsetzen.
- ◇ Das Socket API realisiert eine **einheitliche Schnittstelle** für **unterschiedliche Netzwerkprotokolle**. Die Schnittstelle ist so konzipiert, dass jederzeit neue Protokolle ohne Änderung der Schnittstelle hinzugefügt werden können. Netzwerkprotokolle werden in Abhängigkeit von ihren Kommunikationseigenschaften in verschiedene **Protokoll-Typen** unterteilt (z.B. Datagramm-Protokolle, Stream-Protokolle usw.) und in **Protokollfamilien** (Kommunikations-Domänen, *communication domains*) zusammengefasst (z.B. TCP/IP-Protokoll-Familie, Novell IPX-Protokoll usw.)
- ◇ Sockets ermöglichen eine **bidirektionale Kommunikation**.  
Ein **Socket** bildet einen **Kommunikations-Endpunkt** → Jeder der beiden an einer Kommunikation beteiligten Prozesse benötigt einen Socket.
- ◇ Ein Socket wird – wie eine Datei – über einen **File Deskriptor** referiert  
→ für jeden Socket existiert eine **struct file**-Struktur (File-Objekt).  
Damit können für eine Socket-Kommunikation **Dateibearbeitungs-Funktionen** (System Calls) eingesetzt werden.

### • Realisierung von Sockets

- ◇ Sockets werden durch **spezielle Datenstrukturen** und durch **spezielle Funktionen** realisiert.  
Im virtuellen Dateisystem (*Virtual File System*) werden sie durch **spezielle Inodes** (VFS-Inodes) repräsentiert. Diese besitzen eine **socket-spezifische Komponente** vom Typ **struct socket**
- ◇ Datenstruktur **struct socket** (definiert in "*linux/include/linux/net.h*")

```
typedef enum {
    SS_FREE = 0,           /* not allocated          */
    SS_UNCONNECTED,       /* unconnected to any socket */
    SS_CONNECTING,        /* in process of connecting */
    SS_CONNECTED,         /* connected to socket      */
    SS_DISCONNECTING      /* in process of disconnecting */
} socket_state;

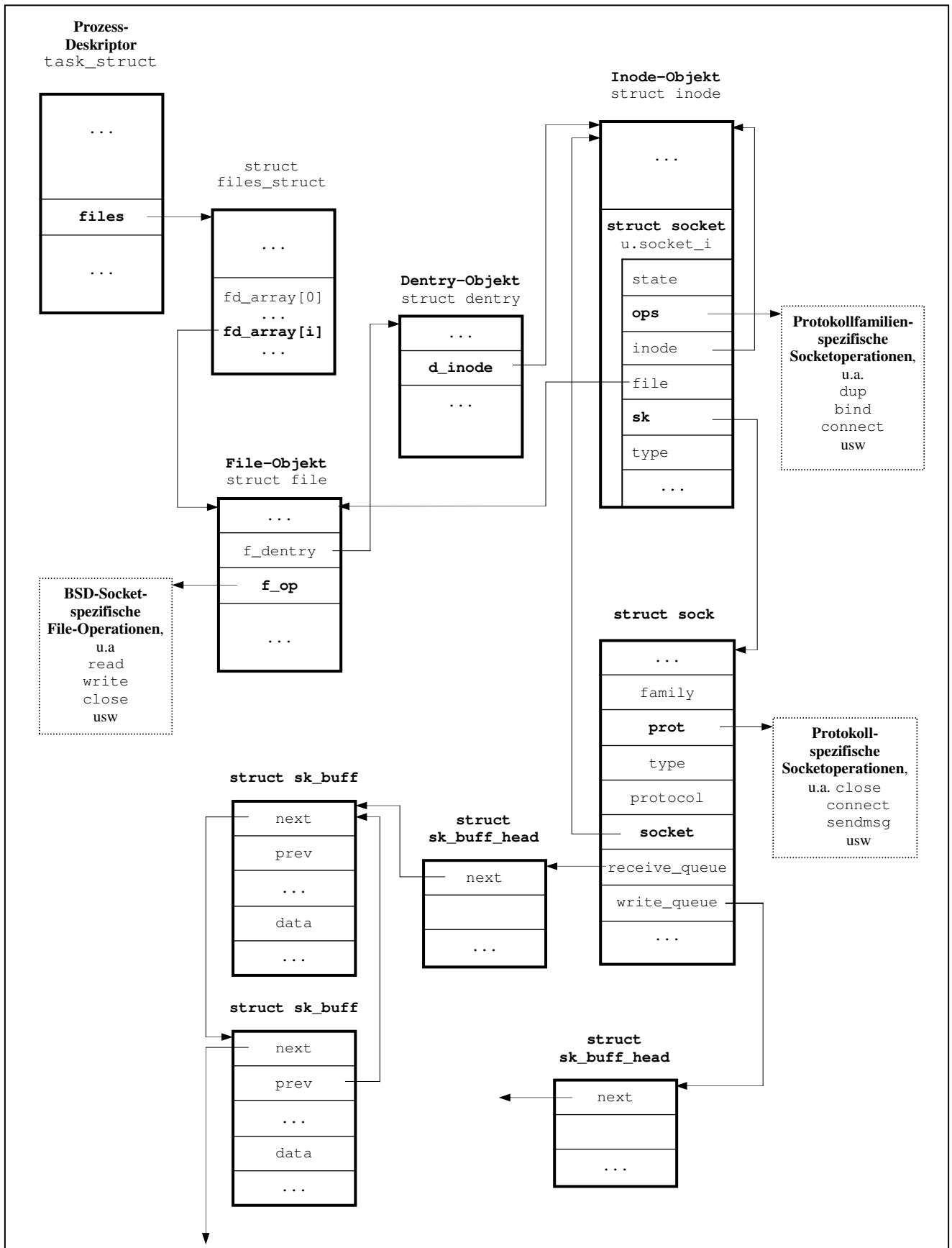
struct socket
{
    socket_state      state;
    unsigned long     flags;
    struct proto_ops* ops;
    struct inode*     inode;
    struct fasync_struct* fasync_list; /* Asynchronous wake up list */
    struct file*      file;           /* File back pointer for gc */
    struct sock*      sk;
    struct wait_queue* wait;
    short             type;
    unsigned char     passcred;
    unsigned char     tli;
};
```

#### Bedeutung einiger Komponenten :

ops     Pointer auf Struktur mit Pointern auf Funktionen für protokollfamilienspezifische Socket-Operationen  
inode   Pointer auf Beginn des Inodes zu dem die struct socket Struktur gehört  
sk      Pointer auf eine Unterstruktur mit protokollspezifischen Informationen  
type    Socket-Typ (entsprechend dem Protokoll-Typ)

## Socketrelevante Verwaltungsstrukturen in LINUX

### • Überblick :



## Prinzip der Socket-Kommunikation (1)

### • Kommunikationsprinzip

Die Kommunikation über Sockets erfolgt nach dem **Client-Server-Modell** :

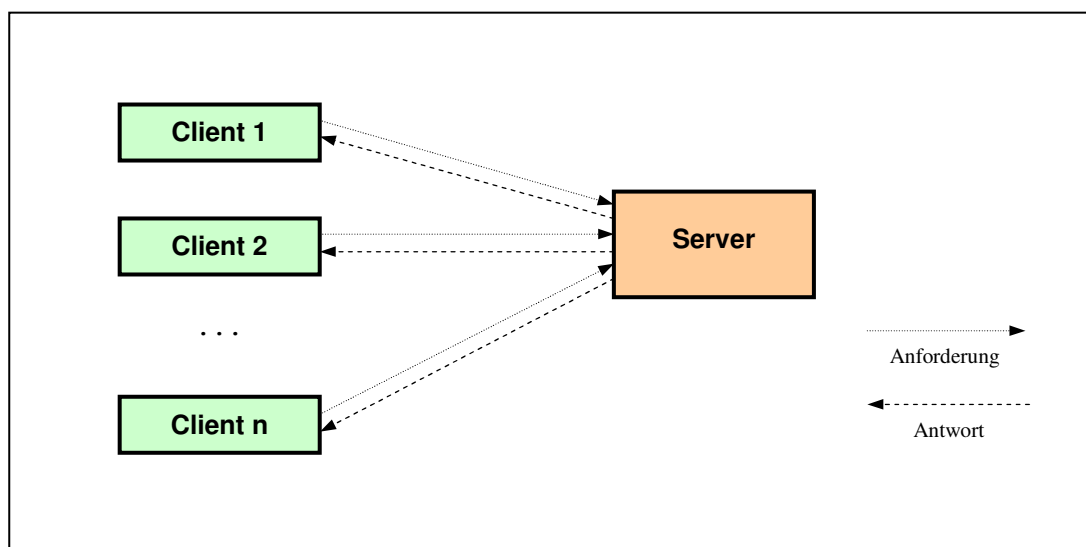
Ein **Server-Prozess** stellt **Dienstleistungen** über eine definierte Schnittstelle zur Verfügung, die von **Client-Prozessen** genutzt werden können.

I.a. findet zwischen Client und Server eine **bidirektionale Kommunikation** statt.

Diese erfolgt **asymmetrisch** auf der Basis eines festgelegten **Protokolls** nach dem **Request-Response-Prinzip** :

Der **Client** richtet zu einem beliebigen Zeitpunkt, also **asynchron**, **Anforderungen** an den **Server**. Dieser **antwortet** i.a. innerhalb einer bestimmten Zeit, also **synchron**.

Ein **Server** kann häufig zu **mehreren Clients** eine Kommunikationsverbindung unterhalten.



### • Kommunikationsverbindungsarten

Die Semantik einer Socket-Kommunikation wird durch den mit jedem Socket assoziierten **Protokoll-Typ** festgelegt. Neben weiteren Kommunikationseigenschaften bestimmt dieser die **Verbindungsart** der Kommunikation.

Man unterscheidet :

#### ◆ verbindungsorientierte Kommunikation

Zwischen Client und Server wird für die Dauer der Kommunikation eine **ständige** – virtuelle – Verbindung hergestellt.

Diese Verbindung muß aufgebaut werden, bevor die eigentliche Kommunikation stattfinden kann

→ Die einzelnen Sendeoperationen benötigen keine expliziten Angaben über die Empfängeradresse.

Andere Prozesse können in eine derartige Verbindung nicht eindringen.

Wichtigster Protokoll-Typ mit dieser Verbindungart : **SOCK\_STREAM**.

#### ◆ verbindungslose Kommunikation

Zwischen Client und Server wird **keine ständige** Kommunikationsverbindung hergestellt.

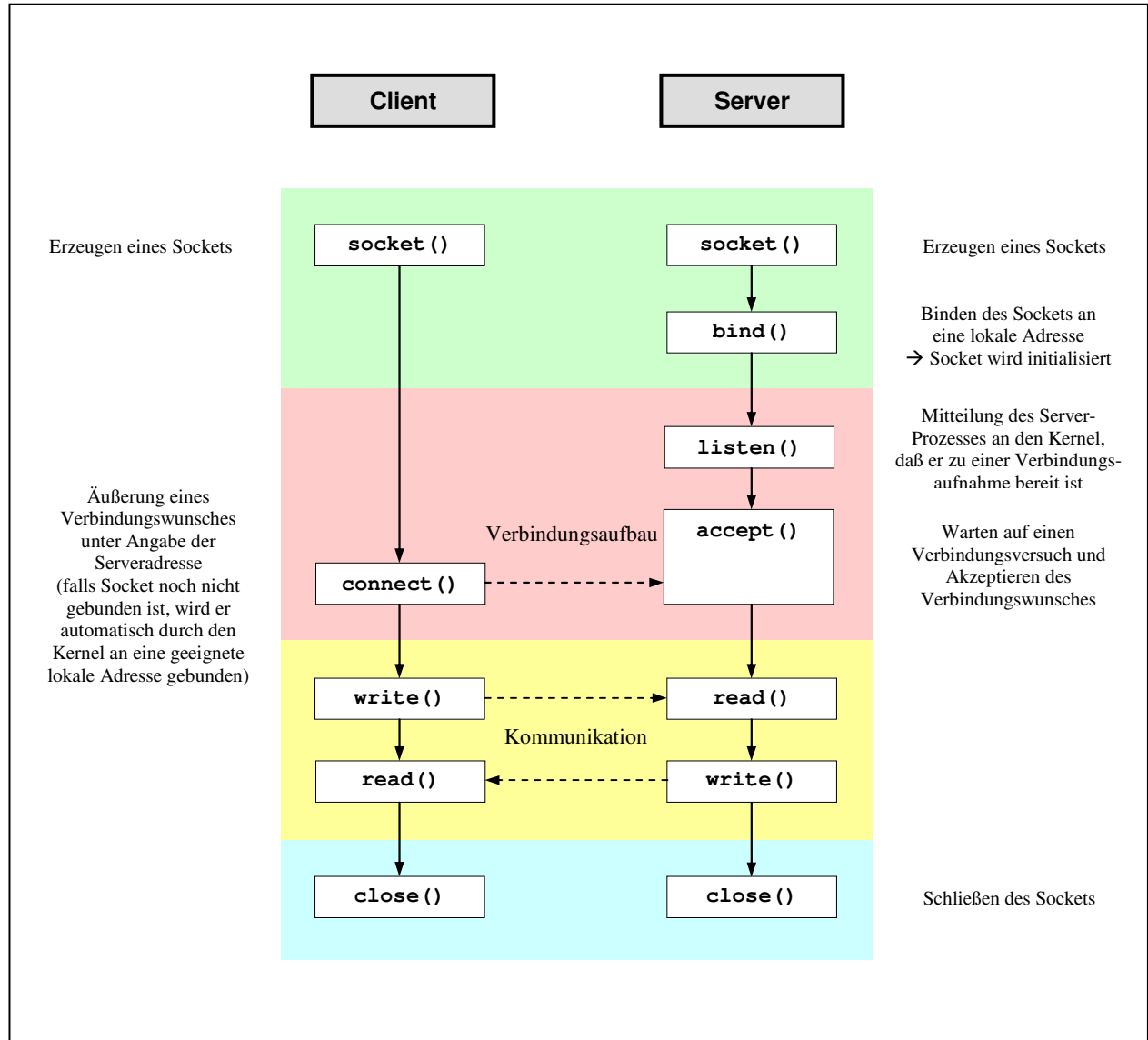
Mit jedem übertragenen Nachrichtenpaket wird die Verbindung zwischen Sender und Empfänger erneut aufgebaut.

→ Jeder Sendeoperation muß die Adresse des Empfängers explizit mitgegeben werden.

Wichtigster Protokoll-Typ mit dieser Verbindungart : **SOCK\_DGRAM**.

## Prinzip der Socket-Kommunikation (2)

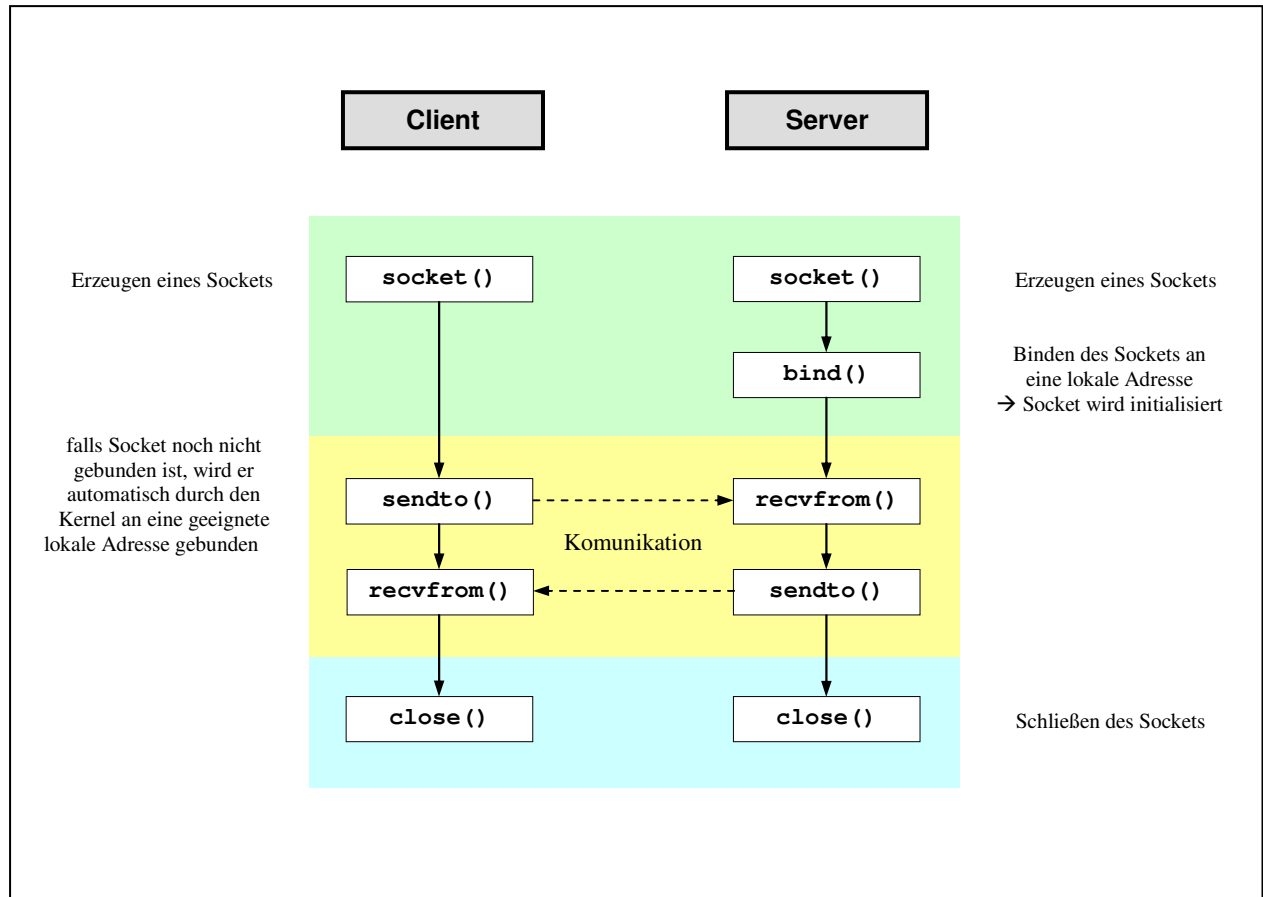
- **Prinzipieller Ablauf einer verbindungsorientierten Socket-Kommunikation**



- ◇ Bei der Socket-Erzeugung (Socket-API-Funktion `socket()`) wird der Protokoll-Typ und damit die Verbindungsart festgelegt.
- ◇ Der **Aufbau der Socket-Verbindung** wird durch ein **symmetrisches Rendezvous** zwischen den Funktionen `accept()` und `connect()` hergestellt:  
`accept()` blockiert bis ein Aufruf von `connect()` das Rendezvous herbeiführt.  
Andererseits blockiert `connect()` bis `accept()` das Rendezvous herbeiführt.  
Allerdings darf der Client `connect()` erst nach dem Aufruf von `listen()` durch den Server ausführen.  
Ein Aufruf von `connect()` vor `listen()` endet mit einer Fehlermeldung.
- ◇ Die **Kommunikation** zwischen Client und Server kann mittels der System Calls `write()` und `read()` oder alternativ mit den Socket-API-Funktionen `send()` und `recv()` erfolgen
- ◇ Nach Beendigung der Kommunikation sollte der Socket wieder geschlossen werden (System Call `close()`)

### Prinzip der Socket-Kommunikation (3)

- **Prinzipieller Ablauf einer verbindungslosen Socket-Kommunikation**



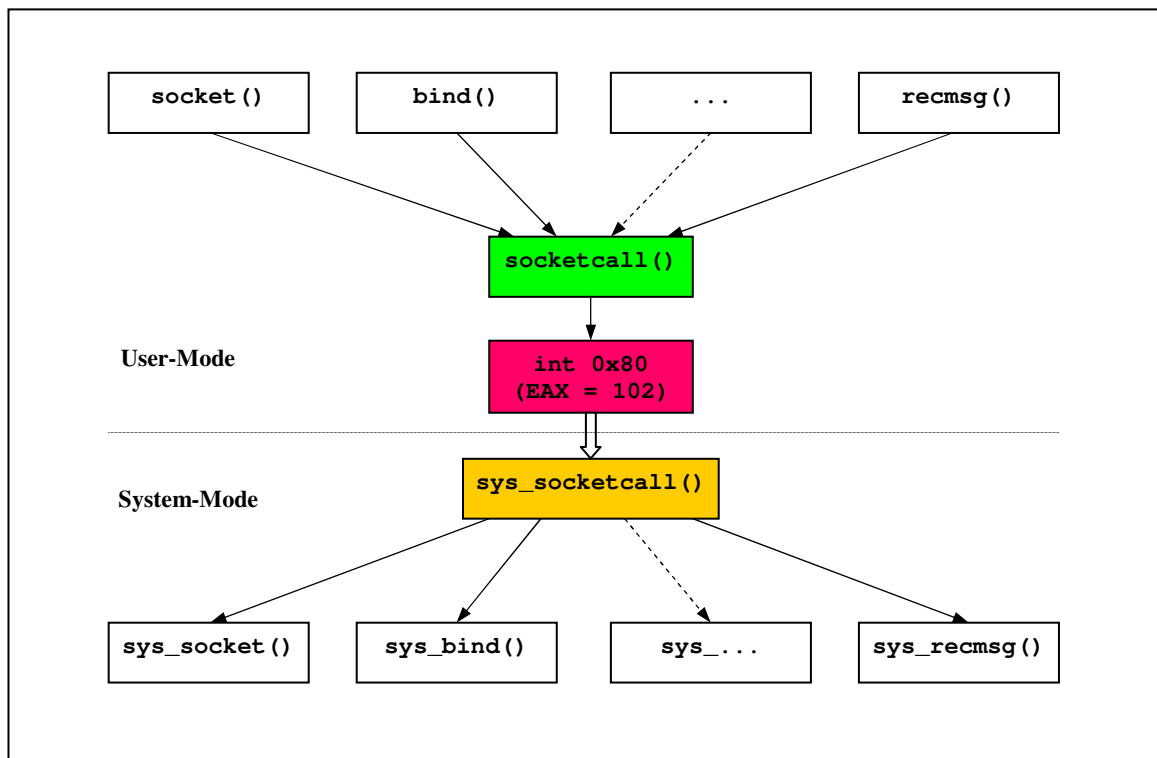
- ◇ In jedem `sendto()`-Aufruf ist die Adresse des Empfängers anzugeben. Durch den Aufruf wird – neben den eigentlichen Daten – auch die Adresse des jeweiligen Absenders übertragen. Der Client muß die Adresse des Servers kennen. Mit jedem Aufruf von `sendto()` wird seine eigene Adresse an den Server übertragen. Dieser erhält die übermittelten Daten und die Adresse des Clients durch den Aufruf von `recvfrom()`. Damit ist er in der Lage seinerseits mittels `sendto()` Informationen (Antwort) an den Client zu übertragen.

## Socket-API in LINUX – Überblick (1)

### • Prinzipielle Realisierung

Das **Socket-API** besteht aus einer **Reihe von Betriebssystemfunktionen**, die alle als **Unterfunktionen** des System Calls **socketcall()** (Funktions-Nr. 102) realisiert sind.

Die einzelnen API-Funktionen sind durch eine **Unterfunktions-Nr.** gekennzeichnet, die im Register **EBX** übergeben wird.



### • Socket-API-Funktionen und Unterfunktions-Nummer

Die Unterfunktions-Nummern für die Socket-API-Funktionen sind in `<linux/net.h>` definiert.

```
#define SYS_SOCKET      1      /* sys_socket(2)          */
#define SYS_BIND        2      /* sys_bind(2)            */
#define SYS_CONNECT     3      /* sys_connect(2)         */
#define SYS_LISTEN      4      /* sys_listen(2)          */
#define SYS_ACCEPT      5      /* sys_accept(2)          */
#define SYS_GETSOCKNAME  6      /* sys_getsockname(2)     */
#define SYS_GETPEERNAME  7      /* sys_getpeername(2)     */
#define SYS_SOCKETPAIR   8      /* sys_socketpair(2)      */
#define SYS_SEND         9      /* sys_send(2)            */
#define SYS_RECV        10     /* sys_recv(2)            */
#define SYS_SENDFROM    11     /* sys_sendto(2)          */
#define SYS_RECVFROM    12     /* sys_recvfrom(2)        */
#define SYS_SHUTDOWN    13     /* sys_shutdown(2)        */
#define SYS_SETSOCKOPT   14     /* sys_setsockopt(2)      */
#define SYS_GETSOCKOPT   15     /* sys_getsockopt(2)      */
#define SYS_SENDMSG     16     /* sys_sendmsg(2)         */
#define SYS_RECVMSG     17     /* sys_recvmsg(2)         */
```

## Socket-API in LINUX – Überblick (2)

### • Adress-Familien

Sockets werden an eine **Adresse** gebunden (→ Socket-Adresse). Diese Adresse referiert den jeweiligen Kommunikations-  
teilnehmer eindeutig (z.B. im Netzwerk : Netzwerk-Adresse).

Der genaue Aufbau einer derartigen Adresse hängt von der Protokoll-Familie, für den der Socket eingesetzt wird, ab.

→ Jeder **Protokoll-Familie** entspricht eine **Adress-Familie**.

Protokoll-Familie und Adress-Familie werden durch die gleiche ganzzahlige Konstante (Familien-Nr) referiert.  
Ihnen sind **symbolische Namen** zugeordnet, die in der durch **<sys/socket.h>** eingebundenen Headerdatei  
<bits/socket.h> definiert sind.

Die **wichtigsten** von LINUX unterstützten **Protokoll- und Adress-Familien** sind :

| Familien-Nr | Protokoll-Familie | Adress-Familie    | "Kommunikations-Domäne"                             |
|-------------|-------------------|-------------------|-----------------------------------------------------|
| 1           | PF_UNIX, PF_LOCAL | AF_UNIX, AF_LOCAL | UNIX-Domain-Sockets (lokale Kommunikation)          |
| 2           | PF_INET           | AF_INET           | IPv4 Internet-Protokoll-Familie (TCP/IP, Version 4) |
| 3           | PF_AX25           | AF_AX25           | Amateur-Radio AX.25 Protokoll                       |
| 4           | PF_IPX            | AF_IPX            | Novell Internet-Protokolle (IPX)                    |
| 5           | PF_APPLETALK      | AF_APPLETALK      | Appletalk DDP                                       |
| 9           | PF_X25            | AF_X25            | ITU-T X.25 / ISO-8208 Protokoll                     |
| 10          | PF_INET6          | AF_INET6          | IPv6 Internet-Protokoll-Familie (TCP/IP, Version 6) |
| 16          | PF_NETLINK        | AF_NETLINK        | "Kernel User Interface Device"                      |
| 17          | PF_PACKET         | AF_PACKET         | "Low Level Packet Interface"                        |
| 23          | PF_IRDA           | AF_IRDA           | IRDA Sockets                                        |
| 31          | PF_BLUETOOTH      | AF_BLUETOOTH      | Bluetooth Sockets                                   |

### • Generischer Socket-Adress-Typ

Das Socket-API **abstrahiert** die unterschiedlichen Adressen durch einen **generischen Socket-Adress-Typ**.

In diversen API-Funktionen wird ein Pointer auf diesen Typ als Parameter verwendet.

Beim Aufruf dieser Funktionen muss der Pointer auf den tatsächlich verwendeten protokollfamilien-spezifischen Adress-  
typ in einen Pointer auf den generischen Adresstyp gecastet werden.

Zusätzlich ist in einem gesonderten Parameter die Länge des tatsächlich verwendeten Adresstyps zu übergeben.

Der generische Socket-Adress-Typ **struct sockaddr** ist in der durch **<sys/socket.h>** eingebundenen  
Headerdatei <bits/socket.h> wie folgt definiert :

```
struct sockaddr {
    unsigned short  sa_family;    /* Address family, AF_XXX */
    char            sa_data[14]; /* Address Data */
};
```

## LINUX Socket-API-Funktion `socket`

- **Funktionalität :** **Erzeugung** eines **Sockets**.  
Rückgabe eines den Socket referierenden **File Deskriptors**.

- **Interface :**

```
int socket(int domain, int type, int protocol);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :** *domain*      **Protokoll-Familie.**

Die wichtigsten zulässigen Werte sind :

**PF\_UNIX, PF\_LOCAL**    UNIX-Domain-Sockets (lokale Kommunikation)

**PF\_INET**              IPv4 Internet-Protokoll-Familie (TCP/IP, Version 4)

**PF\_INET6**             IPv6 Internet-Protokoll-Familie (TCP/IP, Version 6)

**PF\_IPX**                Novell Internet-Protokolle (IPX)

**PF\_NETLINK**           "Kernel User Interface Device"

**PF\_X25**                ITU-T X.25 / ISO-8208 Protokoll

**PF\_AX25**               Amateur-Radio AX.25 Protokoll

**PF\_APPLETALK**        Appletalk DDP

**PF\_PACKET**            "Low Level Packet Interface"

**PF\_IRDA**               Protokoll für IRDA

**PF\_BLUETOOTH**        Protokoll für Bluetooth

- type*      **Protokoll-Typ (Socket-Typ)**

Zulässig sind prinzipiell die folgenden Werte :

**SOCK\_STREAM**        verbindungsorientiertes Protokoll, streambasiert, mit Einhaltung der Datenreihenfolge (Sequencing), mit Fehlerkontrolle

**SOCK\_DGRAM**        verbindungsloses Protokoll, paketbasiert, ohne Sequencing und ohne Fehlerkontrolle, feste max. Paketgröße

**SOCK\_SEQPACKET**    verbindungsorientiertes Protokoll, paketbasiert, mit Sequencing und Fehlerkontrolle, feste max. Paketgröße

**SOCK\_RDM**           verbindungsorientiertes Protokoll, paketbasiert, ohne Sequencing, mit Fehlerkontrolle

**SOCK\_RAW**           direkter Zugriff zur Netzwerkschicht (z.B. IP) ermöglicht die Implementierung neuer Transportschichtprotokolle im User-Bereich

(nur für Prozesse mit `EUID==0`)

- protocol*    zu verwendendes **Protokoll**.

Meist kann der Wert **0** übergeben werden → der Kernel wählt das **Standard-Protokoll** des angegebenen Typs der Protokoll-Familie aus.

(Normalerweise existiert innerhalb einer Protokoll-Familie nur **ein Protokoll** eines **bestimmten Typs**.)

- **Rückgabewert :** - **File Deskriptor**, über den der Socket referiert wird, bei **Erfolg**  
- **-1** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (`socketcall()`), Unterfunktion Nr. **1**  
→ `sys_socketcall(...)` (in `net/socket.c`)  
→ `sys_socket(...)` (in `net/socket.c`)

- **Anmerkung :** Der erzeugte Socket ist **nicht initialisiert**, d.h. an keine Resource (Adresse) gebunden.  
→ Seine Verwendung (Lesen und Schreiben) ist noch nicht möglich.



## LINUX Socket-API-Funktion **bind**

- **Funktionalität :** Binden eines Sockets an eine lokale Adresse.  
("Zuordnung eines Namens" zum Socket)

- **Interface :**

```
int bind(int sockfd, struct sockaddr* addr, socklen_t addrlen);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :**
  - `sockfd` File-Deskriptor , der Socket referiert
  - `addr` Pointer auf lokale Adresse, an die der Socket gebunden werden soll.  
Art und Aufbau der Adresse sind abhängig von der Protokoll-Familie (Adress-Familie) des Sockets.  
Der Pointer auf die tatsächliche Adresse muss in `struct sockaddr*` gecastet werden.
  - `addrlen` Länge der durch `addr` referierten Adresse in Bytes

- **Rückgabewert :**
  - `0` bei Erfolg
  - `-1` im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. 102 (`socketcall()`), Unterfunktion Nr. 2
  - `sys_socketcall(...)` (in `net/socket.c`)
  - `sys_bind(...)` (in `net/socket.c`)

- **Anmerkungen:**
  1. Der Datentyp `socklen_t` ist in der Headerdatei `<sys/socket.h>` definiert als `unsigned int`.
  2. Üblicherweise verzichtet der Client-Prozess auf ein explizites Binden des Sockets an eine lokale Adresse.  
Stattdessen überlässt er es dem Kernel, den Socket – bei der Äußerung eines Verbindungswunsches bzw beim erstmaligen Schreiben (verbindungslose Kommunikation) – an eine geeignete Adresse zu binden.

## LINUX Socket-API-Funktion **listen**

- **Funktionalität :** **Mitteilung** des (Server-)Prozesses an den Kernel, daß er für eine **Verbindungsaufnahme** über den Socket durch andere Prozesse (Client-Prozesse) **bereit** ist.  
Anwendung nur für **verbindungsorientierte** Protokolle

- **Interface :**

```
int listen(int sockfd, int backlog);
```

- **Header-Datei :** **<sys/socket.h>**

- **Parameter :**     *sockfd*   File-Deskriptor , der Socket referiert

*backlog*   Festlegung der maximal erlaubten Anzahl wartender Verbindungswünsche ("pending connections") in der *listen queue*.  
Verbindungswunsch : Aufruf von `connect()` durch einen Client-Prozess.  
Stehen bereits *backlog* Verbindungswünsche an, schlagen weitere Aufrufe von `connect()` fehl.  
Die Implementierung kann den angegebenen Wert auf einen von ihr vorgesehenen Maximalwert begrenzen.  
Grundsätzlich soll jede Implementierung *backlog*-Werte bis zu `SOMAXCONN` (definiert in `<bits/socket.h>`, eingebunden durch `<sys/socket.h>`) unterstützen  
Im Kernel 2.6 ist `SOMAXCONN` definiert zu `128`.

- **Rückgabewert :**   - `0`   bei **Erfolg**  
                      - `-1` im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (**`socketcall()`**), Unterfunktion Nr. **4**
  - `sys_socketcall(...)` (in `net/socket.c`)
  - `sys_listen(...)`   (in `net/socket.c`)

## LINUX Socket-API-Funktion `accept`

- **Funktionalität :** **Akzeptieren** eines **Verbindungswunsches** zu einem Socket durch einen Serverprozess  
Anwendung nur für **verbindungsorientierte** Protokolle.  
Die Funktion wandelt den nächsten anstehenden Verbindungswunsch (aus der gegebenenfalls vorhandenen Warteschlange) in eine akzeptierte Verbindung um.  
Hierfür wird ein **neuer Socket** erzeugt, über den eine sich anschließende Kommunikation abgewickelt werden kann. Der ursprüngliche Socket wird nicht verändert. Er steht für eine Kommunikation nicht zur Verfügung, sondern nur für das Warten auf – weitere – Verbindungswünsche.  
Der neu erzeugte Socket hat die gleichen Eigenschaften wie der ursprüngliche Socket.  
**Rückgabe** eines den neu erzeugten Socket referierenden **File-Deskriptors**.  
Steht kein Verbindungswunsch an, **blockiert** die Funktion solange bis einer auftritt. Es sei denn der Socket ist mit `fcntl()` als `NON_BLOCKING` markiert worden. In diesem Fall kehrt die Funktion sofort mit dem Fehler `EAGAIN` zurück.

- **Interface :**

```
int accept(int sockfd, struct sockaddr* addr, socklen_t* paddrlen);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :**
    - sockfd*      File-Deskriptor , der Socket referiert, zu dem der Verbindungswunsch ansteht.
    - addr*          Pointer auf Adress-Struktur, in die der Kernel die Socket-Adresse des Prozesses, der den Verbindungswunsch äußert (Client-Prozeß), ablegt  
Art und Aufbau dieser Struktur sind abhängig von der Protokoll-Familie (Adress-Familie) des Sockets.  
Der Pointer auf die tatsächliche Adress-Struktur muss gecastet werden in `struct sockaddr*`  
Wird der `NULL`-Pointer übergeben, erfolgt keine Eintragung durch den Kernel.
    - paddrlen*      Pointer auf Variable, in die der Kernel die Adresslänge des Prozesses, der den Verbindungswunsch geäußert hat, ablegt.  
Beim Aufruf sollte `*paddrlen` auf die Größe der durch *addr* referierten Adress-Struktur gesetzt werden.

- **Rückgabewert :**
    - **File Deskriptor**, über den der neu erzeugte Socket referiert wird, bei **Erfolg**
    - **-1** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (`socketcall()`), Unterfunktion Nr. **5**
  - `sys_socketcall(...)` (in `net/socket.c`)
  - `sys_accept(...)` (in `net/socket.c`)

- **Anmerkungen :**
  1. Der Datentyp `socklen_t` ist in der Headerdatei `<sys/socket.h>` definiert als **unsigned int**.
  2. Die – bidirektional mögliche – Kommunikation über den neu erzeugten Socket-File-Deskriptor kann mittels der System Calls `read()` und `write()` bzw `send()` und `recv()` abgewickelt werden.  
Nach Beendigung der Kommunikation ist der Socket-File-Deskriptor mittels `close()` zu schließen.

## LINUX Socket-API-Funktion `connect`

- **Funktionalität :** **Initiierung** eines **Verbindungswunsches** zu einem Socket durch einen Client.  
Wird die Funktion mit einem **ungebundenen Socket** aufgerufen, findet ein **implizites Binden** des Sockets an eine geeignete lokale Adresse statt.  
Die Adresse, an die der Socket gebunden ist, wird zum Serverprozess übertragen.  
Wenn der Verbindungswunsch durch den Serverprozess akzeptiert wird, kehrt die Funktion erfolgreich zurück.  
Anwendung nur für **verbindungsorientierte** Protokolle.

- **Interface :**

```
int connect(int sockfd, struct sockaddr* servaddr, socklen_t addrlen);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :**
  - sockfd* File-Deskriptor , der den Socket des Client-Prozesses referiert
  - servaddr* Pointer auf die Adresse, an die der Socket des Ziel-Serverprozesses ( zu dem eine Verbindung aufgebaut werden soll) gebunden ist  
Art und Aufbau der Adresse sind abhängig von der Protokoll-Familie (Adress-Familie) des Sockets.  
Der Pointer auf die tatsächliche Adresse muss in `struct sockaddr*` gecastet werden.
  - addrlen* Länge der durch *servaddr* referierten Adresse in Bytes

- **Rückgabewert :**
  - **0** bei **Erfolg**
  - **-1** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (`socketcall()`), Unterfunktions Nr. **3**
  - `sys_socketcall(...)` (in `net/socket.c`)
  - `sys_connect(...)` (in `net/socket.c`)

- **Anmerkung :** Der Datentyp `socklen_t` ist in der Headerdatei `<sys/socket.h>` definiert als `unsigned int`.

## LINUX Socket-API-Funktion `setsockopt`

- **Funktionalität :** Setzen von Optionen für Sockets

- **Interface :**

```
int setsockopt(int sock, int level, int option, const void* val,  
               socklen_t vallen);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :**
  - sock*      File-Deskriptor des Sockets
  - level*     Typ der Option  
Beispiele : - **SOL\_SOCKET**    generische Socket-Option  
                                  (auf Socket-Ebene interpretiert)  
                                  (definiert in `<asm/socket.h>`, - indirekt –  
                                  eingebunden durch `<sys/socket.h>`)  
                  - Protokoll-Nummer des die Option interpretierenden Protokolls  
                  (Protokoll-Nummern sind u.a. in `/etc/protocols` enthalten)
  - option*    zu setzende Option, Angabe durch symbolischen Namen  
                  (mögliche Werte sind definiert in der durch `<sys/socket.h>` - indirekt –  
                  eingebundenen Headerdatei `<asm/socket.h>`)  
                  Beispiel : - **SO\_REUSEADDR**    Adresse, an die Socket gebunden ist, kann in  
                                  sehr kurzer Zeit wieder verwendet werden (sonst i.a. erst nach  
                                  längerer Wartezeit, z.B. bei TCP-Ports 2 min)
  - val*        Pointer auf Speicherbereich (Variable), der den Wert der zu setzenden Option  
                  enthält.  
                  Die meisten generischen Socket-Optionen verwenden hierfür eine `int`-Variable  
                  (!=0 : Setzen der Option, ==0 : Rücksetzen der Option)
  - vallen*    Grösse des Speicherbereichs (in Bytes), auf den *val* zeigt.
- **Rückgabewert :**
  - **0**    bei Erfolg
  - **-1**    im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (`socketcall()`), Unterfunktion Nr. **14**
  - `sys_socketcall(...)`    (in `net/socket.c`)
  - `sys_setsockopt(...)` (in `net/socket.c`)

- **Beispiel für Anwendung :**

Bei INET-Socket-Kommunikation : Aufhebung der Beschränkung, dass der Serverprozess sein lokales Port erst nach einer längeren Wartezeit erneut binden kann → Unmittelbar nach Aufheben einer Bindung kann der Socket an dieselbe Adresse erneut gebunden werden.

```
int fdSock;                    /*File Deskriptor des Sockets */  
...  
int iOpt=1;  
setsockopt(fdSock, SOL_SOCKET, SO_REUSEADDR, &iOpt, sizeof(iOpt));
```

## LINUX Socket-API-Funktion `sendto`

- **Funktionalität :** **Senden** von Daten von einem Quell-Socket zu einem Ziel-Socket.  
Da der Funktion u.a. auch ein Pointer auf die Adresse des Ziel-Sockets übergeben werden muss, lässt sie sich vor allem für eine **verbindungslose** (aber auch eine **verbindungsorientierte**) Kommunikation einsetzen.  
Die Funktion überträgt mit den Daten auch die lokale Adresse des Quell-Sockets zum Ziel-Socket.  
Ist der Quell-Socket zum Zeitpunkt des Funktionsaufrufs noch nicht an eine lokale Adresse gebunden, so führt die Funktion ein implizites Binden durch.

- **Interface :**

```
int sendto(int sockfd, const void* msg, size_t len, int flags,  
            const struct sockaddr* to, socklen_t tolen);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :**

|               |                                                                                                                                                                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>sockfd</i> | File-Deskriptor , der Quell-Socket referiert                                                                                                                                                                                                    |
| <i>msg</i>    | Pointer auf Buffer mit den zu sendenden Daten                                                                                                                                                                                                   |
| <i>len</i>    | Anzahl der zu sendenden Bytes                                                                                                                                                                                                                   |
| <i>flags</i>  | bitweise Oder-Verknüpfung mehrerer möglicher Flags zur Beeinflussung der Arbeitsweise der Funktion.<br>für den normalen Datenverkehr üblicherweise auf 0 gesetzt                                                                                |
| <i>to</i>     | Pointer auf die Adresse des Ziel-Sockets<br>Art und Aufbau der Adresse sind abhängig von der Protokoll-Familie (Adress-Familie) des Sockets.<br>Der Pointer auf die tatsächliche Adresse muss in <code>struct sockaddr*</code> gecastet werden. |
| <i>tolen</i>  | Länge der durch <i>to</i> referierten Adresse in Bytes                                                                                                                                                                                          |

- **Rückgabewert :**
  - **Anzahl der gesendeten Bytes** bei Erfolg
  - **-1** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (`socketcall()`), Unterfunktion Nr. **11**
  - `sys_socketcall(...)` (in `net/socket.c`)
  - `sys_sendto(...)` (in `net/socket.c`)

- **Anmerkungen:**
  1. Der Datentyp `socklen_t` ist in der Headerdatei `<sys/socket.h>` definiert als **unsigned int**.
  2. Der Datentyp `size_t` ist in der von `<sys/socket.h>` eingebundenen ANSI-C-Header-Datei `<stddef.h>` definiert als **unsigned int**.

## LINUX Socket-API-Funktion `recvfrom`

- **Funktionalität :** **Empfangen** von Daten aus einem Ziel-Socket.  
Die Funktion lässt sich für eine **verbindungslose** (aber auch eine **verbindungsorientierte**) Kommunikation einsetzen.  
Bei verbindungsloser Kommunikation gibt die Funktion die Adresse des Quell-(Sende-)Sockets über einen Parameter an den aufrufenden Prozess zurück.

- **Interface :**

```
int recvfrom(int sockfd, void* msg, size_t len, int flags,  
             struct sockaddr* from, socklen_t* fromlen);
```

- **Header-Datei :** `<sys/socket.h>`

- **Parameter :**
  - sockfd*      File-Deskriptor , der den Ziel-Socket referiert
  - msg*          Pointer auf Buffer, in dem die Empfangs-Daten abgelegt werden
  - len*          Größe des Buffers für die Empfangs-Daten
  - flags*        bitweise Oder-Verknüpfung mehrerer möglicher Flags zur Beeinflussung der Arbeitsweise der Funktion.  
für den normalen Datenverkehr i.a. auf 0 gesetzt.  
u.a. ist das folgende Flag definiert :  
**MSG\_PEEK**    die empfangenen Daten werden nicht aus dem Socket entfernt,  
ein erneuter Aufruf von `recvfrom()` liefert dieselben Daten noch einmal
  - from*        Pointer auf Struktur, in der – falls `!=NULL` – bei verbindungsloser Kommunikation die Adresse des Quell-(Sende-)Sockets abgelegt wird  
Art und Aufbau der tatsächlichen Adresse sind abhängig von der Protokoll-Familie (Adress-Familie) des Sockets.
  - fromlen*     Pointer auf Variable, die beim Aufruf mit der Größe der durch *from* referierten Adressstruktur zu initialisieren ist.  
Nach Rückkehr der Funktion enthält sie die tatsächliche Größe der unter *from* abgelegten Adresse (in Bytes).

- **Rückgabewert :**
  - **Anzahl der empfangenen Bytes** bei Erfolg
  - **-1** im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **102** (`socketcall()`), Unterfunktion Nr. **12**
  - `sys_socketcall(...)` (in `net/socket.c`)
  - `sys_recvfrom(...)` (in `net/socket.c`)

- **Anmerkungen:**
  1. Der Datentyp `socklen_t` ist in der Headerdatei `<sys/socket.h>` definiert als **unsigned int**.
  2. Der Datentyp `size_t` ist in der von `<sys/socket.h>` eingebundenen ANSI-C-Header-Datei `<stddef.h>` definiert als **unsigned int**.

## UNIX-Domain-Sockets in LINUX

### • Allgemeines

- ◇ Die Protokollfamilie **PF\_UNIX** (==**PF\_LOCAL**) ist die einfachste Protokollfamilie innerhalb des Socket-APIs. Sie stellt die **UNIX-Domain-Sockets** zur Verfügung. Diese Sockets ermöglichen keine echte Netzwerkcommunication sondern eine Kommunikation zwischen **beliebigen Prozessen** des gleichen Rechners (**lokale Interprozess-Kommunikation**). Die miteinander kommunizierenden Prozesse müssen weder miteinander verwandt sein noch die gleiche EUID besitzen.
- ◇ UNIX-Domain-Sockets sind – im Unterschied zu Named Pipes - immer **verbindungsorientiert**. Zwischen zwei Prozessen wird ein privater Kommunikationskanal eingerichtet, in den kein dritter Prozess eindringen kann. Ein Server-Prozess, der gleichzeitig Verbindungen zu mehreren Clients unterhält, verwendet für jede Verbindung einen eigenen Socket-Deskriptor.
- ◇ Gültige **Protokoll-Typen** sind : **SOCK\_STREAM** und **SOCK\_DGRAM**. Dabei arbeitet **SOCK\_DGRAM** ebenfalls verbindungsorientiert und gewährleistet Sequencing und eine Fehlerkontrolle (zuverlässige Verbindung).

### • UNIX-Domain-Adressen

- ◇ Die Adressen von UNIX-Domain-Sockets sind **Dateipfade** im Dateisystem (Adress-Familie **AF\_UNIX** bzw **AF\_LOCAL**)  
Hierfür existiert ein eigener Dateityp "Socket".  
Durch das Binden einer Adresse an einen Socket (mittels `bind()`) wird ein **neuer Eintrag im Dateisystem** vom Typ "Socket" erzeugt.  
Falls bereits ein Eintrag des angegebenen Pfadnamens existiert (egal welchen Dateityps), endet `bind()` mit dem Fehler `EADDRINUSE`.
- ◇ Damit ein (Client-)Prozess eine Verbindung zu einem existierenden (Server-)Socket aufnehmen kann ( mittels API-Funktion `connect()` ) muss er Lese- und Schreibrechte zu der entsprechenden "Socket-Datei" besitzen.
- ◇ Nach **Beendigung der Verwendung** eines Sockets sollte dieser wieder **aus dem Dateisystem entfernt** werden (→ System Call `unlink()` ).

### • Structure-Datentyp `struct sockaddr_un`

- ◇ Dieser in `<sys/un.h>` definierte Datentyp dient zur Darstellung von **UNIX-Domain-Adressen**.

```
struct sockaddr_un {  
    unsigned short  sun_family;      /* Address family, hier AF_UNIX */  
    char            sun_path[108];   /* Address Data : Dateipfad      */  
};
```

- ◇ Die für diesen Adresstyp anzugebende **Länge** (als Parameter in API-Funktionen) ergibt sich als Summe der Länge der Adress-Familien-Komponente (`sun_family`) und der tatsächlichen Länge des Dateipfades.
- ◇ **Beispiel** : 

```
struct sockaddr_un my_addr;  
socklen_t addrlen;  
...  
addrlen = sizeof(my_addr->sun_family) + strlen(my_addr->sun_path);
```
- ◇ **Achtung** : Bei der Übergabe eines Pointers auf diese Adress-Struktur in den API-Funktionen ist ein Cast in einen Pointer auf den generischen Socket-Adresstyp `struct sockaddr*` notwendig.



### Beispiel zu UNIX-Domain-Sockets (1) : Serverprozess

```
/* ----- */
/* C-Quelldatei locserver_m.c */
/* Modul mit main()-Funktion fuer das Programm locserver */
/* ----- */
/* Beispiel fuer Unix Domain Sockets */
/* Einfacher Serverprozess, */
/* - der auf einen Verbindungswunsch durch einen Clientprozess wartet, */
/* - einen Verbindungswunsch akzeptiert */
/* - und die ihm vom Clientprozess geschickten Daten an die Standardausgabe ausgibt */
/* Wenn der Clientprozess die Kommunikation durch Senden von EOF (=CTRL-D) beendet, */
/* wartet er auf den naechsten Verbindungswunsch */
/* ----- */

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>

#define SOCKET_PATH "/home/thomas/mysocket"
#define FD_STDOUT 1

void errorExit(char* msg);
void copyData(int iDest, int iSrc);

int main(int argc, char *argv[])
{
    struct sockaddr_un strAddr;
    socklen_t lenAddr;
    int fdSock;
    int fdConn;

    if ((fdSock=socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
        errorExit("socket");

    unlink(SOCKET_PATH); /* Sicherstellung, dass SOCKET_PATH nicht existiert */
    strAddr.sun_family=AF_UNIX; /* Unix Domain */
    strcpy(strAddr.sun_path, SOCKET_PATH);
    lenAddr=sizeof(strAddr.sun_family)+strlen(strAddr.sun_path);

    if (bind(fdSock, (struct sockaddr*)&strAddr, lenAddr) != 0)
        errorExit("bind");

    if (listen(fdSock, 5) != 0)
        errorExit("listen");

    while ((fdConn=accept(fdSock, (struct sockaddr*)&strAddr, &lenAddr)) >= 0)
    {
        printf("\nConnection !!! receiving data ...\n");
        copyData(FD_STDOUT, fdConn);
        printf("\n... finished\n");
        close (fdConn);
    }

    close(fdSock);
    unlink(SOCKET_PATH);

    return 0;
}
```

### Beispiel zu UNIX-Domain-Sockets (2) : Clientprozess

```
/* ----- */
/* C-Quelldatei locclient_m.c */
/* Modul mit main()-Funktion fuer das Programm locclient */
/* ----- */
/* Beispiel fuer Unix Domain Sockets */
/* Einfacher Clientprozess, */
/* - der versucht, eine Socket-Verbindung zu einem Serverprozess herzustellen */
/* - und bei Erfolg fortlaufend Zeichen von der Standardeingabe einliest */
/* - und diese über den Socket an den Serverprozess sendet. */
/* Der Prozess endet, wenn EOF (==CTRL-D) eingegeben wird */
/* ----- */

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>

#define SOCKET_PATH "/home/thomas/mysocket"
#define FD_STDIN 0

void errorExit(char* msg);
void copyData(int iDest, int iSrc);

int main(int argc, char *argv[])
{
    struct sockaddr_un strAddr;
    socklen_t lenAddr;
    int fdSock;

    if ((fdSock=socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
        errorExit("socket");

    strAddr.sun_family=AF_UNIX; /* Unix domain */
    strcpy(strAddr.sun_path, SOCKET_PATH);
    lenAddr=sizeof(strAddr.sun_family)+strlen(strAddr.sun_path);

    if (connect(fdSock, (struct sockaddr*)&strAddr, lenAddr) !=0 )
        errorExit("connect");

    printf("\nConnected to Server ... sending data ...\n");
    copyData(fdSock, FD_STDIN);
    printf("\nready !\n");
    close(fdSock);

    return 0;
}
```

### Beispiel zu UNIX-Domain-Sockets (3) : Hilfsfunktionen für Server und Client

```
/* ----- */
/* C-Quell-Modul com_util.c */
/* ----- */
/* Hilfsfunktionen, die in den Programmen locserver und locclient */
/* eingesetzt werden */
/* ----- */

#include <unistd.h>
#include <stdio.h>

/* ----- */

/* Ausgabe einer Meldung msg, gefolgt von der Fehlermeldung, die dem in errno */
/* gespeicherten Fehlercode entspricht */
/* anschließende Beendigung des Programms */

void errorExit(char* msg)
{
    perror(msg);
    exit(1);
}

/* ----- */

/* Kopieren von Bytefolgen von einem Quell-File-Deskriptor zu einem Ziel-File- */
/* Deskriptor */
/* Das Kopieren endet, wenn keine Bytes mehr vom Ziel-Deskriptor gelesen werden */

#define BUFFLEN 1024

void copyData(int iDest, int iSrc)
{
    static char acBuff[BUFFLEN];
    int iCount;

    while ((iCount=read(iSrc, acBuff, sizeof(acBuff)))>0)
    {
        if (write(iDest, acBuff, iCount)!=iCount)
            errorExit("write");
    }

    if (iCount<0)
        errorExit("read");
    return;
}

/* ----- */
```

## INET - Sockets in LINUX (1)

### • Allgemeines

- ◇ Sockets der Protokoll-Familie **PF\_INET** (INET-Sockets) ermöglichen eine Netzwerkkommunikation unter Verwendung der im **Internet** eingesetzten TCP/IP-Protokolle (Internet-Protokoll Version 4, das neue Internet-Protokoll Version 6 wird durch Sockets der Protokoll-Familie **PF\_INET6** realisiert).
- ◇ Gültige Angaben für den Protokoll-Typ (Parameter *type* der API-Funktion `socket()`) sind :
  - **SOCK\_STREAM** (→ TCP-Socket)
  - **SOCK\_DGRAM** (→ UDP-Socket)
  - **SOCK\_RAW** (→ Raw-Socket, IP-Protokoll)
- ◇ Zulässige Werte für den Parameter *protocol* der API-Funktion `socket()` (definiert in `<netinet/in.h>`) sind :
  - TCP-Sockets : **0** oder **IPPROTO\_TCP** (def. in `<netinet/in.h>`)
  - UDP-Sockets : **0** oder **IPPROTO\_UDP** (def. in `<netinet/in.h>`)
  - Raw-Sockets : eine der in RFC1700 den definierten IANA IP-Protokollen zugeordnete Protokoll-Nr.
- ◇ Eine lokale Adresse, die an einen TCP-Socket gebunden war, ist nach dem Schließen des Socket-Deskriptors für eine gewisse Zeit für eine erneute Bindung nicht verfügbar (es sei denn die Option `SA_REUSEADDR` ist gesetzt gewesen)

### • INET-Adressen (IP-Socket-Adressen)

- ◇ Die Adressen, an die INET-Sockets gebunden werden (Adress-Familie **AF\_INET**), bestehen aus
  - einer **IP-Adresse** (Adresse des Rechners im Netzwerk)
  - einer **Port-Nummer** (eindeutige Identifikation einer auf dem Rechner laufenden TCP/IP-Applikation)
- ◇ **IP-Adressen** sind – im gesamten Netzwerk eindeutige – **32-Bit-Werte** (4 Bytes).  
Üblicherweise werden die 4 Bytes durch je einen Punkt getrennt in Dezimaldarstellung angegeben (*Dotted Decimal Notation*) : **aaa.bbb.ccc.ddd**  
Beispiel : 192.168.206.52
- ◇ **Portnummern** liegen im Bereich **0 ... 65535 (16-Bit-Werte)**.  
Die Portnummern **0 ... 1023** (*well known ports*) sind **reserviert** für Prozesse mit Root-Rechten. Hierzu gehören i.a. die "Internet-Standard-Dienste", wie z.B. `ftp` (Nr. 21), `http` (Nr. 80) usw. (s. Datei `/etc/services`).
- ◇ Eine **TCP/IP-Verbindung** wird durch **zwei Endpunkte** festgelegt, die **jeweils** durch eine **IP-Adresse** und eine **Port-Nummer** gekennzeichnet sind.

### • Structure-Datentyp `struct sockaddr_in`

- ◇ Dieser in `<netinet/in.h>` definierte Datentyp dient zur Darstellung von **INET-Adressen**.

```
struct in_addr {                /* Typ zur Darstellung von IP-Adressen */
    unsigned int  s_addr;       /* in Network Byte Order          */
}

struct sockaddr_in {
    unsigned short sin_family;   /* Address family, hier AF_INET    */
    unsigned short sin_port;    /* Port-Nr. (in Network Byte Order) */
    struct in_addr sin_addr;    /* IP-Adresse (in Network Byte Order) */
};
```

- ◇ Für **Server-Prozesse** ist i.a. die **lokale IP-Adresse uninteressant** (das bedeutet, sie nehmen Verbindungswünsche auf jeder IP-Adresse, die der Rechner hat, entgegen)  
→ für die Komponente **sin\_addr** können dann (bei `bind()`) **00-Bytes** angegeben werden ("*don't care*"),  
in `<netinet/in.h>` ist hierfür die symbolische Konstante **INADDR\_ANY** definiert
- ◇ Die Angabe von **0** für **sin\_port** (Port-Nr. 0) bewirkt, dass der Server auf **jedem Port** auf Verbindungswünsche "lauscht".

## INET - Sockets in LINUX (2)

- **Darstellungsformate von IP-Adressen und Port-Nummern**

IP-Socket-Adressen (IP-Adressen und Port-Nummern) werden über das Netzwerk übertragen. Da Sende- und Empfangs-Rechner unterschiedliche Darstellungsformate für Mehrbyte-Werte haben können (*Little Endian*, *Big Endian*), hat man ein **einheitliches Übertragungsformat** für Mehrbyte-Protokollinformationen festgelegt : **Network Byte Order** (*Big Endian*). (*Big Endian* : das höchstwertigste Byte wird an der niedrigsten Adresse abgelegt bzw als erstes übertragen).

→ Sowohl IP-Adressen als auch Port-Nummern müssen gegebenenfalls zwischen der vom jeweiligen Rechner verwendeten Darstellungsart (**Host Byte Order**) und der **Network Byte Order umgewandelt** werden.  
Um dies transparent – ohne Kenntnis der jeweiligen *Host Byte Order* – durchführen zu können, stehen in der **C-Bibliothek geeignete Funktionen** zur Verfügung.

- **C-Bibliotheksfunktionen zur Konvertierung zwischen Network Byte Order und Host Byte Order**

◇ Alle Funktionen sind in der Headerdatei `<netinet/in.h>` deklariert.

◇ **Umwandlung eines 32-Bit-Wertes** (in LINUX `int == long`) **von Host Byte Order in Network Byte Order**

```
unsigned int htonl(unsigned int hostlong);
```

Parameter : *hostlong* umzuwandelnder 32-Bit-Wert in *Host Byte Order*

Funktionswert : Umgewandelter 32-Bit-Wert in *Network Byte Order*

◇ **Umwandlung eines 16-Bit-Wertes** (in LINUX `short`) **von Host Byte Order in Network Byte Order**

```
unsigned short htons(unsigned short hostshort);
```

Parameter : *hostshort* umzuwandelnder 16-Bit-Wert in *Host Byte Order*

Funktionswert : Umgewandelter 16-Bit-Wert in *Network Byte Order*

◇ **Umwandlung eines 32-Bit-Wertes** (in LINUX `int == long`) **von Network Byte Order in Host Byte Order**

```
unsigned int ntohl(unsigned int netlong);
```

Parameter : *netlong* umzuwandelnder 32-Bit-Wert in *Network Byte Order*

Funktionswert : Umgewandelter 32-Bit-Wert in *Host Byte Order*

◇ **Umwandlung eines 16-Bit-Wertes** (in LINUX `short`) **von Network Byte Order in Host Byte Order**

```
unsigned short ntohs(unsigned short netshort);
```

Parameter : *netshort* umzuwandelnder 16-Bit-Wert in *Network Byte Order*

Funktionswert : Umgewandelter 16-Bit-Wert in *Host Byte Order*

## INET - Sockets in LINUX (3)

### • C-Bibliotheks-Funktionen zur Umwandlung der Darstellungsart von IP-Adressen

- ◇ IP-Adressen werden häufig als Strings in *Dotted Decimal Notation* angegeben. Andererseits werden sie zur Verwendung für die Socket-API-Funktionen in der `struct in_addr` in **interner Binärdarstellung** (32-Bit-Wert) benötigt. Zur Umwandlung zwischen diesen beiden Darstellungsarten stellt die **C-Bibliothek** geeignete Umwandlungsfunktionen zur Verfügung.
- ◇ Diese Umwandlungsfunktionen sind in der Headerdatei `<arpa/inet.h>` deklariert.
- ◇ **Umwandlung einer IP-Adresse aus der String-Darstellung (*Dotted Decimal Notation*) in die 32-Bit-Binärdarstellung**

```
int inet_aton(const char* cp, struct in_addr* inp);
```

Parameter : `cp` Pointer auf IP-Adresse in String-Darstellung  
          `inp` Pointer auf Structure-Variable, in der die umgewandelte Binärdarstellung (in *Network Byte Order*) abgelegt wird.

Funktionswert : **!= 0**, wenn durch `cp` eine gültige IP-Adresse referiert wird  
              **0**, wenn keine gültige IP-Adresse referiert wird

- ◇ **Umwandlung einer IP-Adresse aus der 32-Bit-Binärdarstellung in die String-Darstellung (*Dotted Decimal Notation*)**

```
char* inet_ntoa(struct in_addr in);
```

Parameter : `in` IP-Adresse in Binärdarstellung (*Network Byte Order*)

Funktionswert : Pointer auf einen String, der die IP-Adresse in *Dotted Decimal Notation* enthält.

Anmerkung : Der zurückgegebene String befindet sich in einem statisch allozierten Buffer, der beim nächsten Aufruf der Funktion überschrieben wird.