

**BSY - FS20**

*Pascal Brunner - brunnpa7*

# Inhaltsverzeichnis

# Kapitel 1

## Vorlesung 1 - Übersicht und Geschichte der Betriebssysteme

### 1.1 Sie können den Begriff Betriebssystem erklären

Das Betriebssystem soll die Komplexität mindern, also dem Programmierer einen Befehlssatz bereitstellen, der einfach zu beherrschen ist und interne Details vor dem Nutzer verbirgt.

#### Aufgaben

- Erweiterte (virtuelle) Maschine, die einfach zu programmieren ist
- Ressourcenmanager → verwaltet die vorhanden Hardware-Ressourcen (bspw. Speicher, CPU)
- Kontrollinstanz für jegliche Zugriffe

#### Bestandteile → Programme im Systemmodus (Kernel Mode)

- Gerätetreiber
- Prozessmanager
- Fenstermanager

Systemaufrufe sind Schnittstellen zwischen Anwendungsprogrammen und Betriebssystem

### 1.2 Sie können die Begriffe Batchbetriebssystem, Uniprogramming, Multiprogramming, Time sharing erklären und diskutieren

#### 1.2.1 Batchsysteme

- Anwender übergibt Job (Lochkarten oder Band) an den Operator
- Der Operator reiht mehrere Jobs sequentiell in einem Eingabegerät auf → Batch
- Ein spezielles Programm, der Monitor kontrolliert Ausführung der Jobs im Batch
- Hilfsprogramme werden nach Bedarf eingelesen

#### 1.2.2 Uniprogramming

- Es befindet sich nur ein Programm im Speicher.
- I/O Operationen sind sehr langsam im Vergleich zur Ausführung von Instruktionen → CPU oft idle
- Programme mit viel I/O Operationen haben eine schlechte CPU Auslastung und verbringen die meiste Zeit mit warten

### 1.2.3 Multiprogramming

- Mehrere Programme im Speicher, Programm wartet auf I/O Operation und der Monitor führt ein anderes Programm aus
- Multiprogramming / Multitasking

#### Hardwareunterstützung

- I/O Interrupts → Programmumschaltung möglich
- (wenn möglich) DMA
- Speicherverwaltung (Memory Management)

#### Softwareunterstützung

- Scheduling
- Steuerung und Kontrolle von Ressourcenzugriff

### 1.2.4 Time Sharing Systeme (TSS)

- Batch Multiprogramming → Keine Interaktion mit Anwender
- Erweiterung Multiprogramming → Concurrency; CPU-Zeit auf mehrere Benutzer verteilt;

#### Wieso funktioniert das?

Der Mensch hat mit einer langen Reaktionszeit, diese Zeit wird verwendet damit andere User das System brauchen können. Dies benötigt ca. 2 Sekunden Rechenzeit pro Minute → ca. 30 Anwender könnten ohne wahrnehmbare Verlängerung der Reaktionszeit ( $\leq 100\text{ms}$ ) das System gemeinsam nutzen.

## 1.3 Sie können den Begriff Monitor erklären

Der Monitor gilt als ein Minimalsystem, welches bei der Initialisierung die nicht gebrauchten Vektoren auf den Default Wert setzen. Es bietet einfache Schnittstellen, so dass die HW nicht beschädigt wird. Des Weiteren hat es eine einfache Kommunikation mit dem Anwender:

- setzen und lesen von Parametern, Speicher, etc.
- laden von Programmen
- debuggen von Programmen

Bei komplexeren Systemen (bspw. BIOS) werden noch weitere HW-Komponenten (Cache Controller oder Memory Management Unit) geladen; der Speicher wird auf die korrekte Funktionalität geprüft.

#### Batch-Systeme: Monitor

- Monitor liest Job um Job vom Eingabegerät
- Monitor legt Job im Anwendungsprogramm-Bereich ab

## 1.4 Sie können Problemstellungen bei frühen Betriebssystemen aufzählen

- Probleme mit nicht deterministischem Programmverhalten → Resultat der Berechnung abhängig von anderen aktiven Programmen
- Probleme mit gegenseitigem Ausschluss → mehr als ein Programm greift gleichzeitig auf Ressource zu
- Probleme mit der Synchronisation → Signale beim Warten verloren
- Probleme mit Deadlocks → Programme warten gegenseitig bspw. Freigabe

## **1.5 Sie können die grundlegende Konzepte von Betriebssystemen aufzählen und diskutieren**

1. Prozesse
  - Programm in Ausführung → Sandbox
2. Scheduling und Ressourcenverwaltung
  - Zuteilung von Rechenleistung
  - BS unterhält Queueus (Ready, Wait, Long Term Queue)
3. Datenmanagement (Speicherverwaltung, File-Systeme)
4. Schutz und Sicherheit
5. System-Architektur

## **1.6 Sie können die aktuelle Betriebssystemtypen aufzählen, erklären und diskutieren**

TBD

## **1.7 Sie können die Begriffe Multithreading und symmetrisches Multiprocessing erklären und diskutieren**

TBD

# Kapitel 2

## Vorlesung 2 - Computer Systeme

### 2.1 Sie können für das Betriebssysteme wichtige Rechnerkomponenten aufzählen und erklären

- Prozessor (CPU)  
Heute oft mehrere CPU's pro Prozessor, wobei man diese dann in unterschiedliche Threads aufteilt. Pro Core hat man dann eine ALU, dementsprechend verwenden die Threads dann die ALU zusammen. Die Grundidee dabei ist, dass man möglichst gut parallelisieren kann → wobei die Optimierung pro Core nicht linear ansteigt.
- Hauptspeicher → Daten und Code  
Diese Aufteilung kann man dann unterschiedlich aufteilen, wie man die Memory teilen möchte.
  - Uniform Memory Access (UMA) → alle Speicherzugriffe gleich lang
  - Non-Uniform Memory Access (NUMA) → ein Adressraum, der Speicher ist jedoch langsamer. Des Weiteren muss das Memory Management berücksichtigt werden
- I/O Module → Sekundärspeicher, Tastatur, Bildschirm etc.
- Bus → verbindet CPU - Speicher - I/O

### 2.2 Sie können Aufgaben und Funktionsweise der CPU aufzählen und erklären

#### Recheneinheiten

- ALU (Arithmetic and Logic Unit → Integer- und Bitoperationen)
- FPU (Floating Point Unit)

#### Adress-, Daten und Statusregister (PSW)

- für Benutzer sichtbar
- von System- und Anwenderprogramme benutzt

#### Kontroll- und Steuerregister

- viele nicht sichtbar
- z.T. von CPU genutzt

#### Instruction (Control) Unit

- Steuereinheit

### **Bus Interface Unit**

- Ansteuerung der externen Einheiten

*EVTL: Bild einfügen*

### **Programmier-Model**

- Beim Programmieren muss man sich mit den *CPU Registern* (Daten, Stack, PC) auseinandersetzen.
- Der *Instruktionssatz* kann sich je nach Chipmodel anpassen
- *Memory* ist ein linearer Addressraum, die Addressierung erfolgt in Bytes und Worte. Die kleinste Einheit ist ein Byte
- *I/O-Geräte* → memory mapped I/O (var1) oder dedicated I/O (var2). Diese Module dienen Daten vom und zum System-Bus zu liefern. Diese

## **2.3 Sie können Interrupts im Zusammenhang mit Betriebssystemen erklären und diskutieren**

Im Wesentlichen nutzt man Interrupts um hin- und her zu schalten. Es unterbricht die aktuelle Ausführung eines Prozesses.

*asynchrone Interrupts:*

- Timer → Aus Sicht des Programmes planbar
- I/O Devices → Aus Sicht des Programmes nicht vorhersehbar
- Hardwarefehler → Aus Sicht des Programmes nicht vorhersehbar

*synchrone Interrupts:*

- Program → bspw. div durch 0
- Trap, SWI

Die Kontrolle über diese Interrupts, wird durch den sogenannten Interrupt Handler (ISR) - → als Vektor-Tabelle dargestellt - dargestellt und kontrolliert. Wobei nur das Betriebssystem, die Vektortabelle setzen kann.

## **2.4 Sie können Kernel und User Mode erklären und diskutieren**

Die meisten Prozessoren arbeiten in zwei Modi

- Kernel Mode
- User Mode

### **Kernel Mode**

Ist ein privilegierter Mode → unbeschränkter Zugriff auf alle Instruktionen und Ressourcen.

Drei Wege um vom User Mode zum Kernel Mode zu wechseln

- Syscall
- Interrupt
- Exception

### **User Mode / System Mode**

Ist ein nicht privilegierter Mode mit beschränktem Zugang zu kritischen Instruktionen, kein Zugang zu System Timer, Interrupt Controller oder System Control Block. Des Weiteren besteht ein beschränkter Zugriff auf Speicher Peripherie.

- beschränkter Zugriff → 'Sicherer Modus'

## 2.5 Sie können die Funktionsweise von System Calls erklären

Grundsätzlich geht es darum eine Schnittstelle zwischen Anwendersoftware und Systemsoftware zu erstellen. Es schaltet zwischen Kernel und User Mode um.

### Ablauf

1. Anwenderprogramm stellt Parameter bereit (Register, Stack)
2. ruft Bibliotheksfunktion auf → normaler Prozedurauftruf (oft Assembler Programm und setzt System Call Number)
3. Syscall bzw. SWI → Kontrolle geht an Kernel ⇒ privilegierter Modus (wobei der Kernel den System Call Handler aufruft und der Handler ausgeführt wird)
4. evtl. Rückkehr zur Bibliotheksfunktion → Kann zu anderen Anwenderprogramme umschalten (falls System Call blockiert oder ein anderes Anwenderprogramm Rechenzeit benötigt.)
5. Rückkehrs ins Anwenderprogramm

TBD - Codebeispiel Vorlesung 2 Seite 17 einfügen

## 2.6 Sie können I/O Module und -Kommunikation beschreiben und diskutieren

Es wird zwischen drei Techniken der Kommunikation unterschieden:

### Programmed I/O oder synchroner I/O:

- benötigt keine Interrupts
- CPU wartet auf Beendigung jeder einzelnen I/O Operationen
- Unschönheit: busy wait → die CPU wartet bis alles beendet wurde ⇒ der Zeitfaktor ist hier bei ca.  $10^6$

### Interrupt Driven I/O oder asynchroner I/O:

- CPU führt während I/O Operation Code aus
- Wird unterbrochen, wenn I/O Operation beendet
- Die CPU kann andere Arbeiten ausführen, bis das Vorherige beendet wurde

### Direct Memory Access (kurz: DMA):

- Man schaltet die CPU direkt mit dem Memory
- Ein Speicherblock mit Daten wird vom/zum Speicher übertragen
- ohne Rechenleistung der CPU zu beanspruchen
- gibt einen Interrupt, wenn Transfer beendet
- CPU nur zu Beginn und zu Ende des Transfer benötigt
- CPU kann dazwischen andere Arbeiten ausführen

## 2.7 Sie können die Speicherhierarchie moderner Prozessorsysteme erklären und diskutieren und die mittlere Zugriffszeit berechnen

Zwischen Register und Hauptspeicher liegt ein Verhältnis von x100.

Zwischen Hauptspeicher und Disk liegt ein Verhältnis von x1'000'000 - im Vergleich ist dies 1s zu 12 Tagen.

Aus diesem Grund lohnt es sich ein entsprechendes Konzept aufzubereiten, damit man die häufigen Daten schnell zur Verfügung hat. Dabei unterscheidet man vor allem zwischen Cache Level 1 - 3

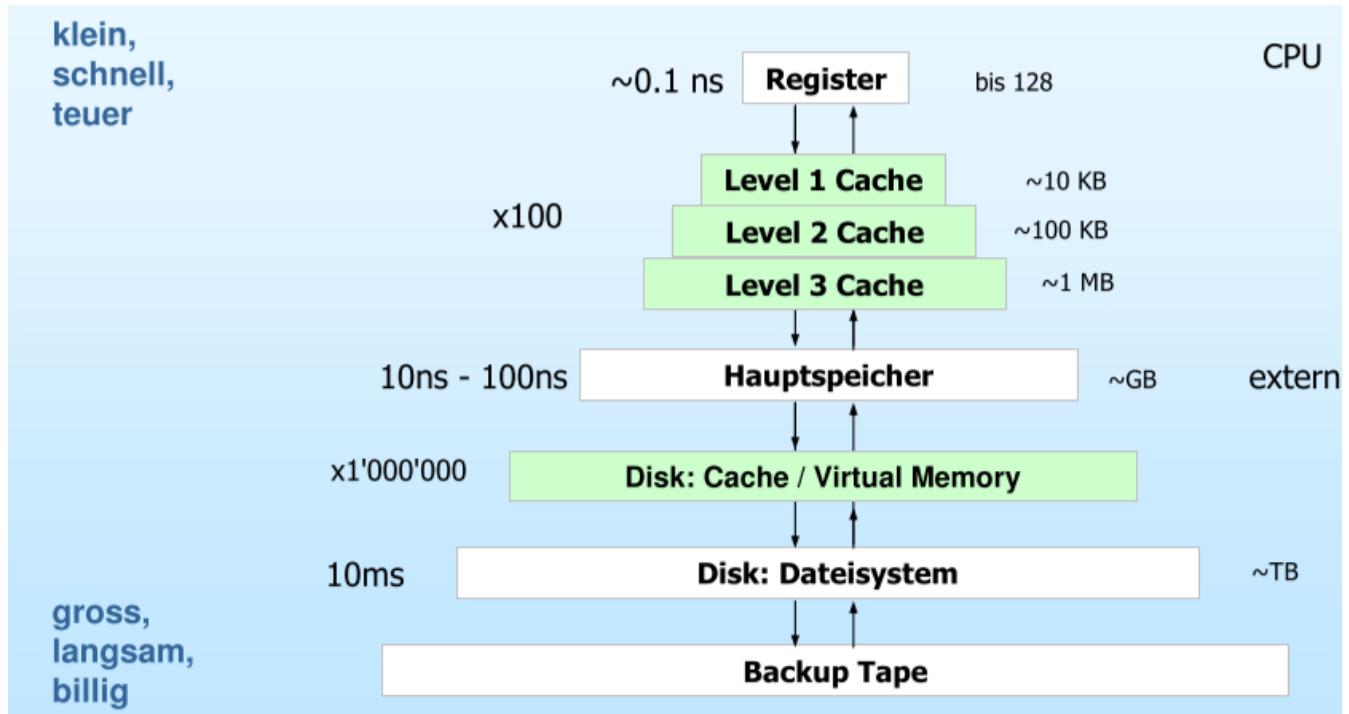


Abbildung 2.1: Abbildung der Speicherhierarchie

### 2.7.1 Cache

Beim Cache möchte man die wichtigen Daten möglichst nahe an der CPU halten. So dass man möglichst wenig Block Transfers zwischen Cache und Main Memory. Befindet sich das ein Wort im Cache, dann kann man dies direkt an die CPU übermitteln. Ist das Wort nicht im Cache, so muss man den kompletten Memory Blcok laden, in welchem sich das Wort befindet.

Das ganze ist transparent für den Benutzer, dies bedeutet, dass er dies (abgesehen von der Ladezeit) nicht wirklich merkt.

Wieso funktioniert das? → Lokalitätsprinzip

Dies funktioniert aufgrund des Lokalitätsprinzip. Man unterscheidet zwischen räumliche Lokalität und zeitliche Lokalität.

- räumliche Lokalität (spacial locality) → grosse Wsk, dass nächster Speicherzugriff, nahe liegende Daten stattfindet (Bspw. Array-Zugriff)
- zeitliche Lokalität (temporal locality) → grosse Wsk, dass Speicherzugriff auf gleiche Daten nochmals stattfindet (Bspw. for-Schleife)

Wie funktioniert es?

**Hitrate (hit ratio):**  $h$

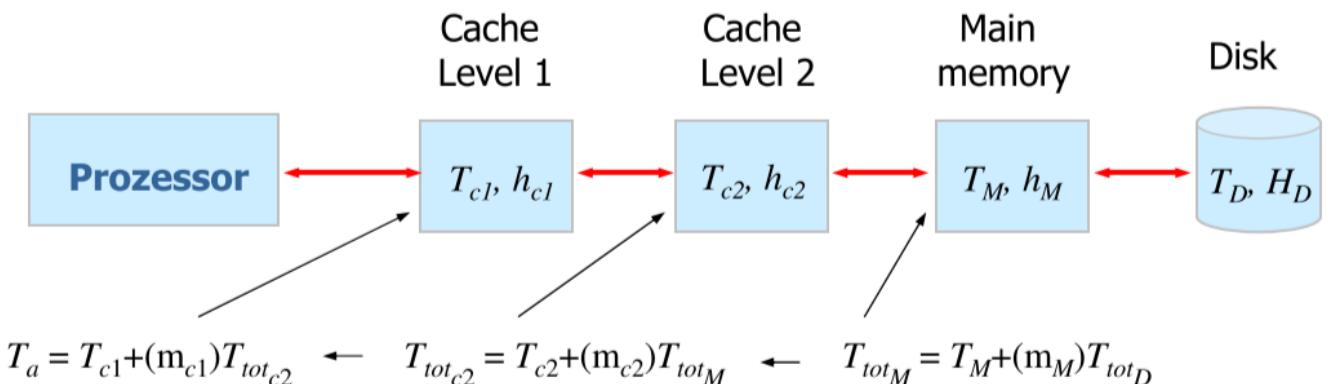
$h$ : Wahrscheinlichkeit, dass der Speicherzugriff die Daten im Cache findet

$m$ : Missrate (miss ratio):  $m = 1 - h$

**Mittlere Zugriffszeit**  $t_a$

- Zugriffszeit auf nächsten Speicher  $T_N$  gegeben
- hit ratio  $h$  gegeben
- einstufiger Cache gegeben

$$\begin{aligned} T_a &= T_C + (1 - h) * T_N \\ T_a &= T_C + m * T_N \end{aligned} \quad (2.1)$$



$$T_a = T_{c1} + (m_{c1}) \cdot (T_{c1} + m_{c2}) \cdot (T_M + (m_M) \cdot T_D)$$

$$T_a = T_{c1} + m_{c1} \cdot T_{c2} + m_{c1} \cdot m_{c2} \cdot T_M + m_{c1} \cdot m_{c2} \cdot m_M \cdot T_D$$

Beispiel:  $m = 10\%$        $\underbrace{0.1}_{\text{0.1}}$        $\underbrace{0.1 \cdot 0.1 = 0.01}_{\text{0.01}}$        $\underbrace{0.1 \cdot 0.1 \cdot 0.1 = 0.001}_{\text{0.001}}$

Abbildung 2.2: Berechnung der verschiedenen Cache Level 1-3

## 2.8 Sie können den Startvorgang einer Betriebssystems erklären

Das Betriebssystem startet in zwei wesentlichen Phasen

- hardwareabhängige Phase
- Start der eigentlichen Betriebssystems

### **Hardwareabhängige Phase:**

- Prozessor startet auf seiner Reset-Adresse
- Code aus Festwertspeicher (ROM, Flash) auf Platine ausgeführt → Hardwareüberprüfung, initialisiert Minimalzugriff auf Disk oder Netzwerk

### **Betriebsabhängige Phase:**

- Boot Code wird ausgeführt → steuert alle weiteren Schritte

### **Ablauf**

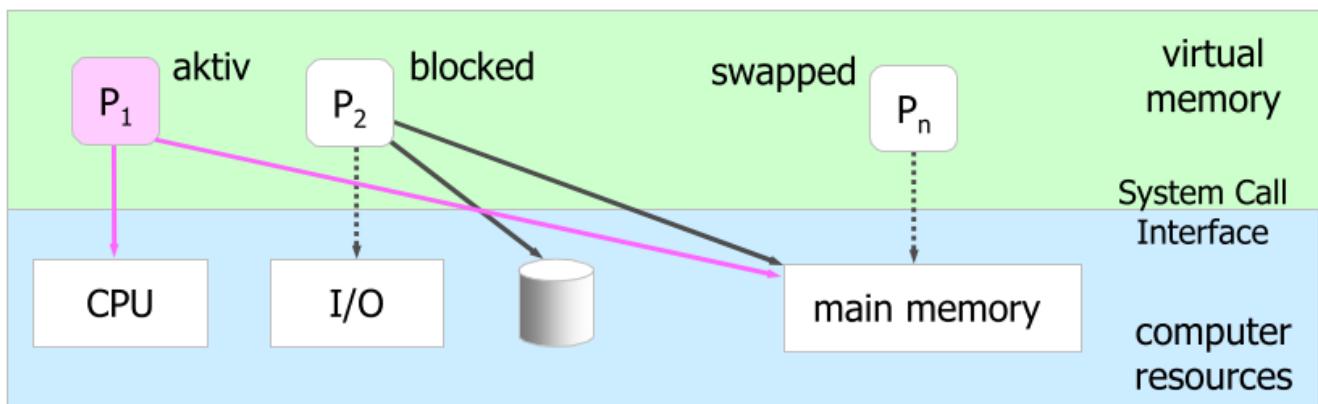
1. Der Prozessor startet das BIOS (Basic Input Output System)
2. BIOS testet das System (POST) und lädt Master Boot Record (MBR) vom ersten Sektor des ersten Disks
3. Der MBR-Code liest den Boot-Code von der ersten aktiven Partition und startet den Boot-Code (Loader)
4. Der Boot-Code lädt sich weitere Informationen (Files) zum Startet des Betriebssystems von der Partition

# Kapitel 3

## Vorlesung 3 - Prozesse und Threads

### 3.1 Sie können den Begriff Prozess erklären

- Ein Prozess ist eine Instanz eines Programmes in Ausführung
- Man schafft pro Prozess einen Kontext → Prozessenthält Kontextinformationen
- Prozess muss nicht zwingend aktiv sein → einer der wichtigen Fragestellungen, wie viele Prozesse darf ich gleichzeitig laufen lassen?
- Dadurch entsteht eine Kapselung → Sandbox-Konzept



$t = T_0$

Abbildung 3.1: Schnappschuss seiner Laufzeitumgebung im Prozess-Kontext

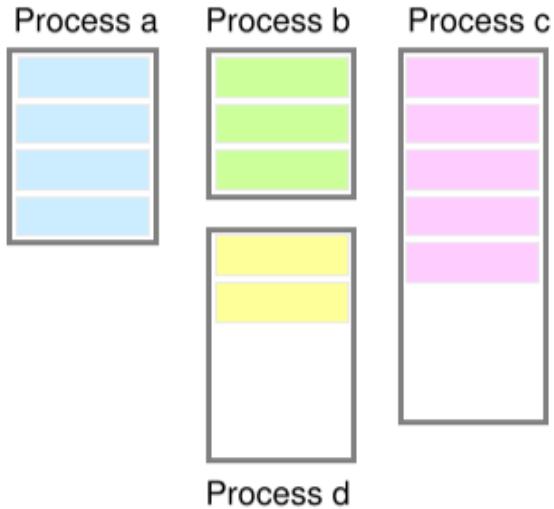
#### Zwei Sichtweise

- **Unit of Resource Ownership** → Eine Einheit, die Ressourcen besitzt; ein virtueller Adressraum, in dem das Prozess Image steht; Kontrolle über Ressourcen
- **Unit of Scheduling** → Eine Einheit, die schedulerbar ist; CPU-Scheduler weist der CPU einen Prozess zu (dispatch); zum Prozess gehören der Executino State (PC, SP, Register) und Ausführungsriorität

### 3.2 Sie können erklären, wie Prozesse auf einer CPU ausgeführt werden

- Prozesse werden gleichzeitig (concurrent) ausgeführt. Ihre Ausführung wird auf einer CPU verschränkt
- Prozesse können in der Ausführung unterbrochen und später weiterverarbeitet werden

## Konzeptmodell 4 Prozesse in Ausführung



## Prozess-Interleaving auf einer CPU

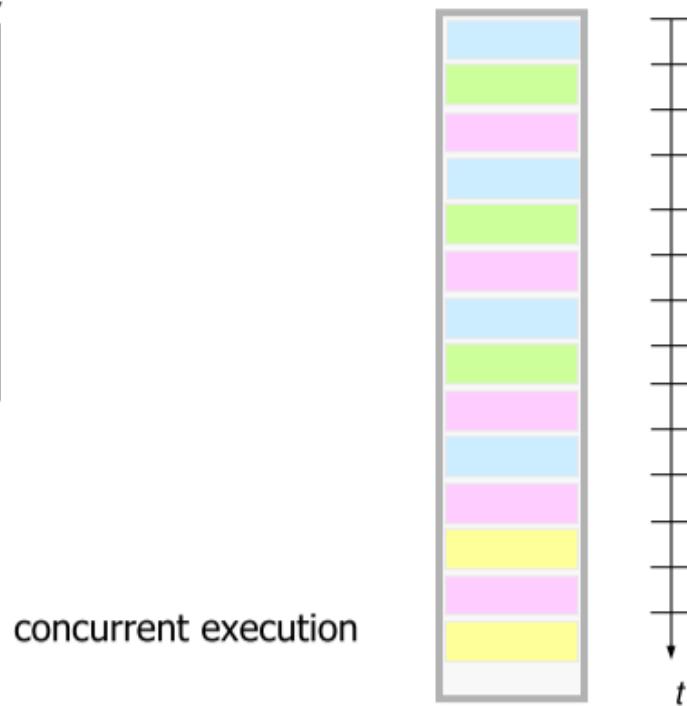


Abbildung 3.2: Prozessausführung einer CPU

**3.3 Sie können die Beschreibung der Prozessausführung erklären und skizzieren**

### 3.3.1 Zeitliches Verhalten

- Zustandsmodell
- Beschreibt den aktuellen Zustand eines Prozesses, z.B. Running oder Not-Running

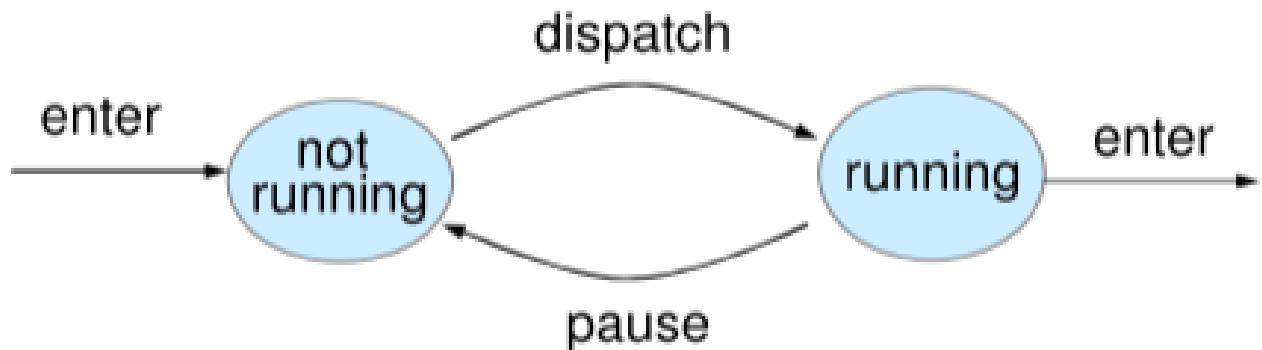


Abbildung 3.3: Prozessausführung einer CPU zeitliches Verhalten

⇒ In der Realität sieht dieses Zustandsdiagramm wesentlich komplexer aus:

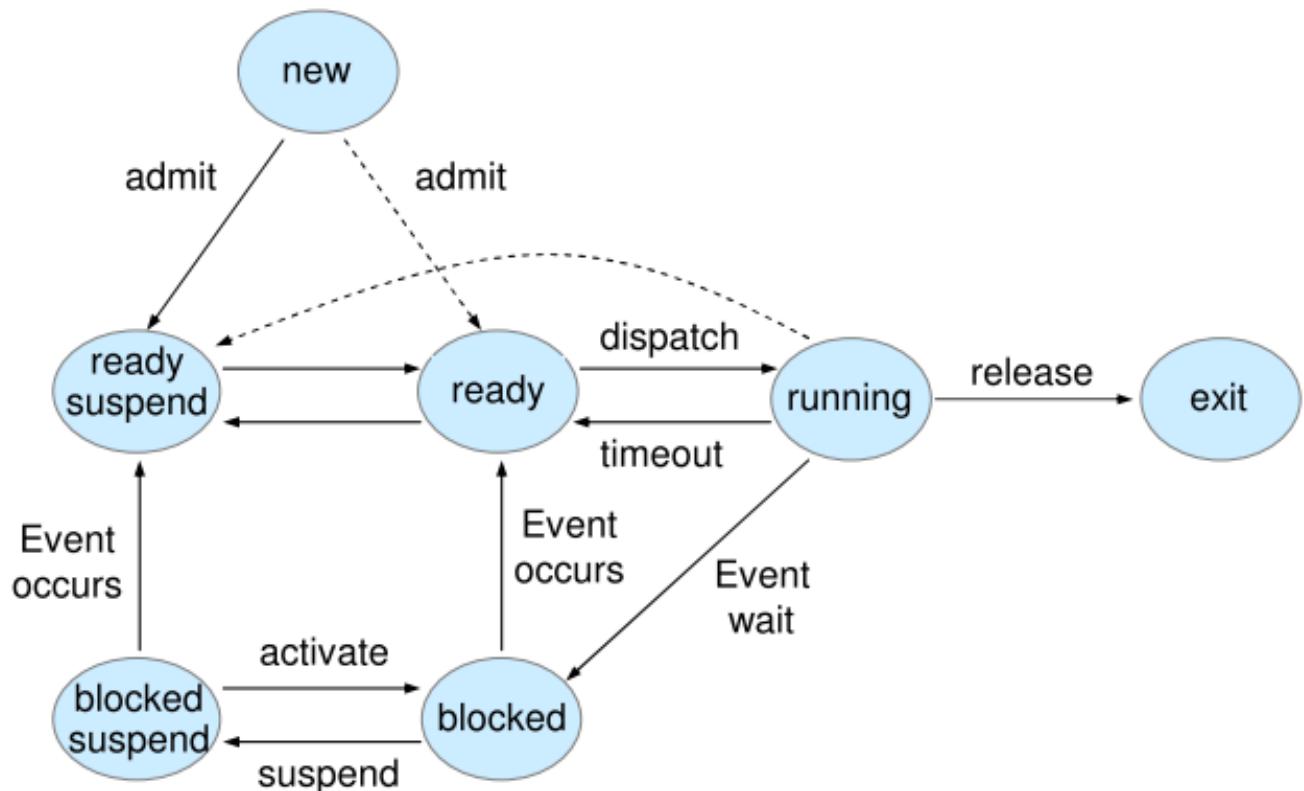


Abbildung 3.4: Prozessausführung einer CPU zeitliches Verhalten (Realität)

### 3.3.2 Örtliches Verhalten

- Queuing-Diagramm
- Beschreibt den Ort, wo sich Prozesse aufhalten, wenn Sie sich in einem bestimmten Zustand befinden
- Mehrere Prozesse können im gleichen Zustand befinden, z.B. Not-Running, in diesem Fall warten sie in einer Warteschlange

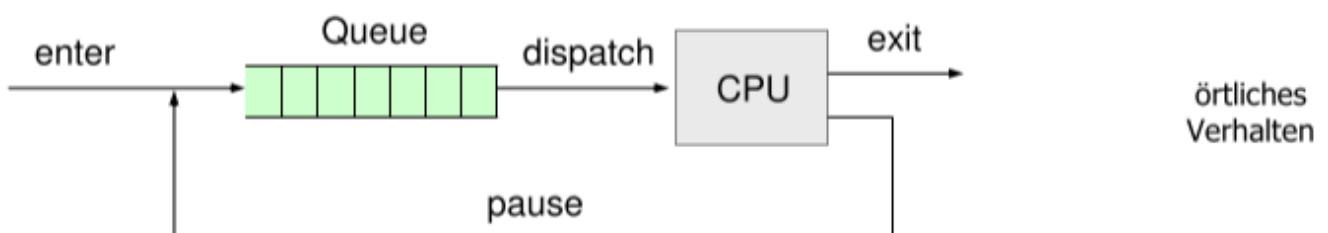


Abbildung 3.5: Prozessausführung Örtliches Verhalten

⇒ In der Realität sieht diese Queue wesentlich komplexer aus:

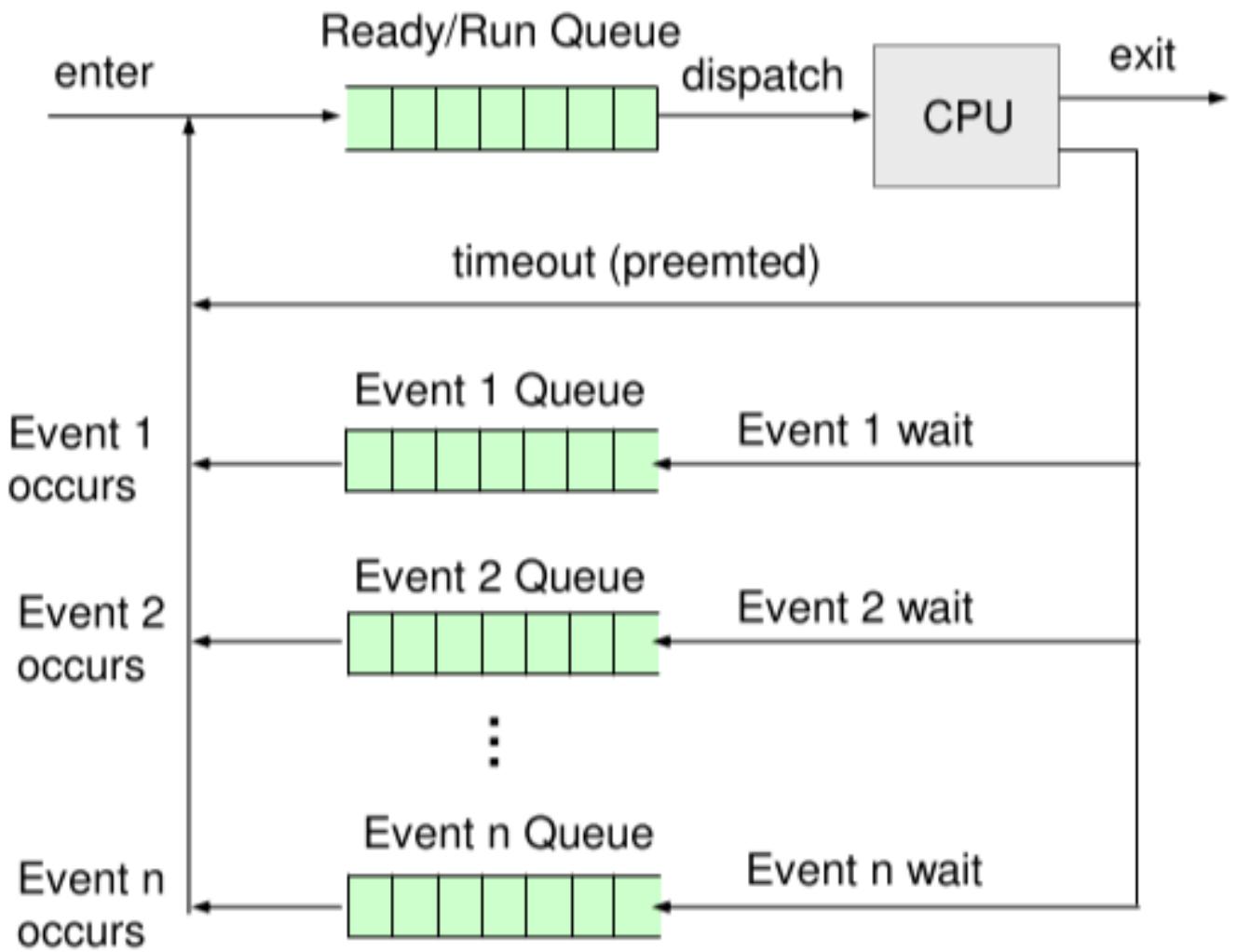


Abbildung 3.6: Prozessausführung Örtliches Verhalten (Realität)

### 3.4 Sie können den Prozesswechsel und die Ausführungsmodi erklären

Gründe für Prozesswechsel:

- Timer (clock) → Zeitintervall abgelaufen ⇒ Prozess geht in Zustand ready
- I/O Interrupt → auf Interrupt wartenden Prozess in den Zustand READY oder READY SUSPEND versetzen (Entscheid ob laufender Prozess unterbrochen wird oder nicht)
- Page fault (virtual memory)
- **TBD V03 - S.12**

Zwei Möglichkeiten für den Prozesswechsel:

1. **Mode Switch:** Kein Prozesswechsel, nur Unterbruch → kostet nicht so viel Zeit
2. **Context Switch:** Prozesswechsel, impliziert auch Mode Switch

### 3.4.1 Prozessausführung Modi

- **User Mode:** weniger privilegiert → Anwenderprogramme
- **System Mode:** Mehr privilegiert → Betriebssystemfunktion

## 3.5 Sie können erklären, wie Prozesse durch das BS verwaltet werden

Das Betriebssystem unterhält und verwaltet verschiedenste Tabellen für die Verwaltung von Prozessen und Ressourcen.

- Speicher → Memory Tabellen
- Geräte → I/O Tabellen
- Files → File Tabellen
- CPU → Prozess Tabellen

Die Informationen liegen in den verschiedenen Tabellen. Die oben erwähnten Tabellen werden konzeptionell von jedem Betriebssystem verwendet, jedoch unterscheidet sich dies in der Implementierung.

**Wichtig:** Die Tabellen müssen selbst wieder miteinander verknüpft sein, d.h. ein Prozess muss wissen welche Files er benutzt etc.

### Prozess Image

- Besteht aus Benutzerprogramm (Code bzw. Text), Daten, Stack, Heap
- Besteht Kontext im Prozesskontrollblock (PCB) gespeichert → PCB, eine Datenstruktur mit Zustandsinformationen zum Prozess
- Sind im virtuellen Speicher abgelegt

Das Betriebssystem hat nur Zugriff auf Prozessdaten, wenn das Prozessimage teilweise im physikalischen Speicher steht.

### Prozesskontrollblock PCB

Das PCB ist eine der wichtigsten Datenstrukturen im Betriebssystem → es speichert Prozesskontext und ist Teil des Prozessimages.

Es enthält alle notwendigen Informationen zum Prozess:

- Process Identification
- Process Control Information
- Process State Information

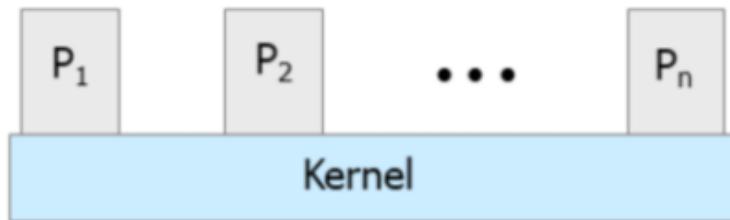
## 3.6 Sie können die Prozesserzeugung am Beispiel Unix / Linux erklären und anwenden (Praktikum)

nicht weiter relevant für mündlich Prüfung

### 3.7 Sie können erklären und diskutieren, wie das Betriebssystem ausgeführt wird

Das Betriebssystem arbeitet wie normale Computersoftware und wird als Programm vom Prozessor ausgeführt. Dabei gibt es verschiedene Implementationen:

#### 3.7.1 Kernel mit eigenem Kontext



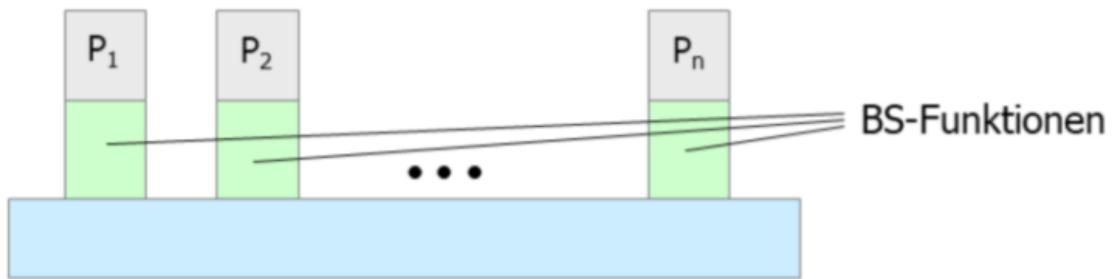
## ■ **Kernel mit eigenen Kontext**

- nur Benutzerprogramme sind Prozesse
- Betriebssystemcode
  - eigenständiges Programm mit eigenem Kontext
  - wird im System Mode ausgeführt
- innerhalb eines Benutzer-Prozesskontextes wird nie Betriebssystemcode ausgeführt

*Abbildung 3.7: Kernel eigener Kontext*

Hier kann der Kernel als eigenständiges Programm betrachtet werden. Es hat einen eigenen Kontext, also eigenen Daten- und Stackbereich.

### 3.7.2 BS-Funktionen im Kontext der Benutzerprozesse

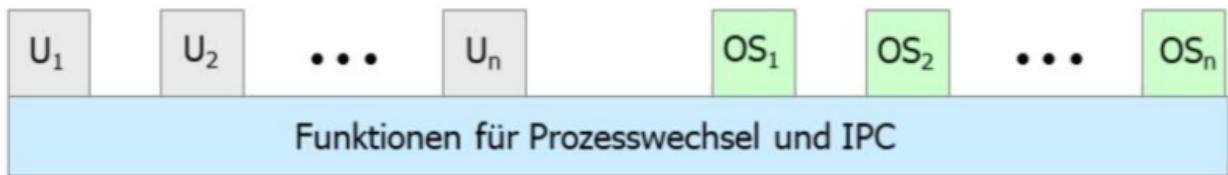


## ■ BS-Funktionen im Kontext der Benutzerprozesse

- BS stellt "**nur Funktionen**" zur Verfügung
  - Betriebssystemcode im Kontext des Benutzerprozesses
- System Calls, Interrupts, Traps
  - CPU schaltet in System Mode, Ausführung aber im Benutzerkontext
- Context Switching meist ausserhalb von Prozessen

Abbildung 3.8: Benutzerprozess

### 3.7.3 Prozessbasiertes Betriebssystem



## ■ Prozessbasierte Betriebssysteme

- kleine Anzahl von Systemfunktionen (Prozesswechsel, IPC) ausserhalb von Prozessen → Microkernel
- das BS ist eine Sammlung von Systemprozessen
- grössere Kernel Funktionen sind eigenständige Prozesse
- gut geeignet für Multiprozessorsysteme

Abbildung 3.9: prozessbasiert

3.8 Sie können den Unterschied zwischen Thread und Prozess erklären und diskutieren, sowie Anwendungen und Vorteile aufzeigen

## Thread vs. Prozess

---

### ■ Prozess: verwaltbare Einheit

- Prozess: unit of resource ownership
- Prozess: unit of scheduling

### ■ Moderne Betriebssysteme trennen Eigenschaften

- Prozess: unit of resource ownership
- Thread: unit of scheduling
- Multithreading → mehrere Threads pro Prozess

Abbildung 3.10: Thread vs Process

Prozesse sind *heavyweight processes*: Enthält virtuellen Adressraum, Programmcode, Daten, Betriebssystem-Ressourcen.  
Und sind single thread of execution.

Threads sind *lightweight process*: Sie sind an Prozesse gebunden, jedoch bestehen sie aus einer unabhängigem Stück Code. Teilt dabei mit anderen Thread den Adressraum, Daten und Kontext. Hat nur eigene lokale Variablen, Register, PC, Stack und Stackpointer.

*Vorteile von Threads:*

- Sind schnell erzeugt und beendet (brauchen nur Stack und Speicher für Register)
- Threadwechsel ist schnell (nur PC, SP und Register austauschen)
- Thread benötigen - wenig Ressourcen; - keinen neuen Adressraum, keine eigenen Datenbereich oder Programmcode; - keine zusätzlichen Betriebssystemressourcen

⇒ Man kann mittels Threads eine Effizienzsteigerung erreichen, dies dank Multithreading (schnelle Threadwechsel und bessere CPU Auslastung).

*Nachteile:*

- Bitten keinen Schutz zwischen Threads → keinen Schutz der Daten gegen unbeabsichtigten Zugriff

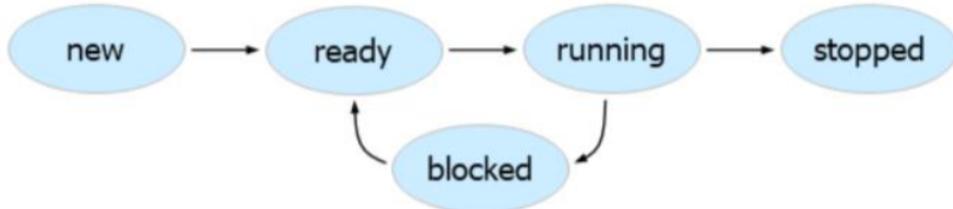
*Anwendungen:*

- Datenserver ≈ Webserver, der mehrere HTTP Requests gleichzeitig empfangen kann.
- Excel ⇒ Ein Thread für Anzeige, ein anderer führt im Hintergrund Befehle aus
- ⇒ weitere Beispiele: Slide 36

### 3.9 Sie können die Threadausführung erklären und diskutieren

#### ■ **Threads: 3 resp. 5 Zustände**

- Running, Ready und Blocked (New und Stopped)



- kein Zustand Suspend: alle Threads im gleichen Adressraum
- Swapping: Prozess mit allen Threads suspendiert
- Prozesses terminiert → alle Threads terminieren

#### ■ **Was geschieht wenn ein Thread blockiert ?**

- user level threads → der Prozess blockiert
- kernel level threads → der Thread blockiert

Abbildung 3.11: Threadausführung

### 3.10 Sie können Threadtypen aufzählen und diskutieren

#### 3.10.1 User Level Threads (ULT)

Kernel weiss nicht, dass es Threads gibt:

- Anwendung ist für Threadmanagement verantwortlich
- Threadwechsel benötigt keine Kernel Mode Privilegien

##### Vorteile

- Threadwechsel involviert Kernel nicht
- Läuft auf jedem BS
- Schedulingverfahren kann anwendungsspezifisch gewählt werden
- Schnell und einfach

##### Nachteile

- System Calls blockieren -> Prozess blockiert und damit auch andere Threads des Prozesses
- Nur Kernel kann Prozesse den Prozessoren zuweisen
- Page Fault eines Thread blockiert ganzen Prozess

### 3.10.2 Kernel Level Threads (KLT)

Thread Management durch Kernel

- Kernel verwaltet Kontextinformation von Prozessen und Threads
- Threadwechsel erfordert Intervention des Kernels
- Threads können auf mehreren Prozessoren verteilt werden

Vorteile

- Kernel kennt alle Thread (reagiert anders wenn Prozess 10 oder 1 Thread hat)
- Scheduling auf Thread Basis
- Kernelfunktionen können selbst Multithreaded sein
- geeignet für SMP (was das?)
- geeignet für Anwendungen die oft blockieren

Nachteile

- Threadwechsel innerhalb Prozesses kostet 2 Mode Switches
- spürbarer Overhead
- spürbar langsamer bei 1 CPU

## 3.11 Sie können das Solaris Prozess- und Threadkonzept erklären können

Annahme: Solaris ist ein Unix-Betriebssystem - wir schauen es uns als Case Study an.

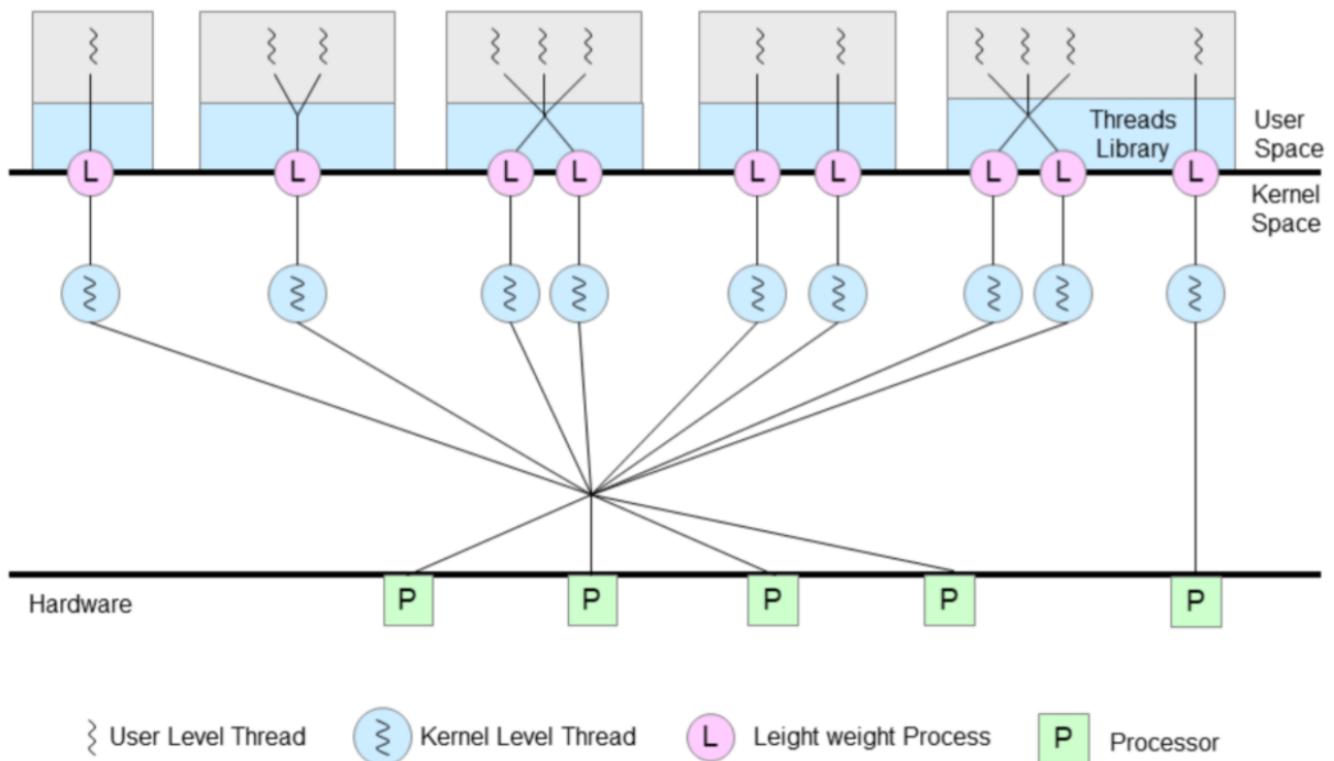


Abbildung 3.12: Solaris

Prozesse (blau/graue Boxen im User Space)

- Sind normale Unix Prozesse mit User Address Space, Stack und PCB
- Enthält einen oder mehrere Lightweight Prozesse

#### **User Level Thread** (Bibliotheksfunktionen)

- Sind unsichtbar für das Betriebssystem
- Schnittstelle zu Parallelität auf Applikationsebene

#### **Lightweight Process** (LWP)

- Bindeglied zwischen ULT und KLT
- Unterstützt mehrere ULTs, hat Verbindung zu einem KLT

#### **Kernel Level Threads**

- schedulierbare und dispatchable Einheiten

Weitere Bemerkungen:

- Man braucht mehrere User Level Threads, wenn eine logische Parallelität vorliegt und keine Hardware-Parallelität nötig ist => Bsp.: Mehrere Fenster auf dem Display(logische Parallelität), aber nur eins ist aktiv (keine Hardware-Parallelität).
- Vorteil von mehreren LWPs: Blockierende Threads können einzeln an einen LWP gebunden werden => Prozess blockiert nicht, wenn ein Thread blockiert.

# Kapitel 4

## Vorlesung 4 und 5 - Scheduling

- 4.1 Sie können den Begriff Scheduling erklären und diskutieren
- 4.2 Sie können die wichtigsten Scheduling Algorithmen aufzählen, erklären und diskutieren
- 4.3 Sie können die Scheduling Verfahren der wichtigsten Betriebssysteme erklären und diskutieren
- 4.4 Sie können Probleme beim Multiprozessor Scheduling aufzählen und erklären

- Task läuft lange auf CPU → viele Daten ins Cache geladen ⇒ beim CPU-Wechsel müssen die Daten neugeladen werden
- Task hält Spin-Lock  $q$  läuft ab →

### 4.4.1 Space Sharing

Verwandte Task bzw. mehrere Task (Threads) arbeiten zusammen. Scheduling mehrerer Tasks über mehrere CPU's.  
**einfachstes Verfahren:**

- non-preemptive → gestartet Tasks werden abgearbeitet
- CPU idle während Blocking
- Tasks nur starten wenn genügend CPU's verfügbar
- Vor-/Nachteil → kein Overhead wegen Kontextwechsel und eher schlechtes Load-Balancing

**Alternatives Verfahren:**

- zentrale Instanz (Server) überwacht Scheduling
- Anzahl Threads wird von Applikation dynamisch angepasst

### 4.4.2 Gang Scheduling

- man gruppierter verwandte Threads gemeinsam zu einer *Gang*
- Gang-Mitglieder sind gleichzeitig auf verschiedenen CPU's aktiv, welche gemeinsam starten und enden

## 4.5 Sie können den Begriff Real-Time Scheduling erklären und diskutieren

- Real-Time Ereignisse sind Systeme, welche auf die äussere Welt reagiert
- Ereignisse finden in der reellen Zeit statt → real-time
- intuitive Betrachtungsweise → was bedeutet das System muss schnell reagieren?

Real-Time Systeme verhalten sich korrekt, wenn:

- das logische Resultat einer Berechnung stimmt (Daten müssen beim Flugzeug real-time sein, beim Streaming ist es weniger schlimm)
- Das Resultat zum richtigen Zeitpunkt ausgeleifert wird (innerhalb der Deadline)
- Hard Real-Time Systeme ⇒ Deadline **MUSS** eingehalten
- Soft Real-Time Systeme ⇒ Deadline **Kann** eingehalten

### 4.5.1 Eigenschaften

- zeitkritische Task wiederholen sich in regelmässigen Abständen
- die Dauer und die benötigten Betriebsmittel sind bekannt

### 4.5.2 Drei Real-Time Task Klassen

1. kritische Tasks (periodische oder asynchrone bzw. sporadische Tasks)
2. nicht kritische, aber notwendige Tasks
3. nicht notwendige Tasks (nice to have)

## 4.6 Sie können die wichtigsten Real-Time Scheduling Algorithmen aufzählen, erklären und diskutieren

### 4.6.1 Rate Monotonic Scheduling (preemptive)

Statische Priorität eines Jobs proportional zu Repetitionsrate.

Das spannende dabei ist, dass die Grenze für erfolgreiches Scheduling berechnet werden kann  
**Diskussion:**

- eher konservativ, besserer Auslastung möglich
- Scheduling Algorithmus beliebig
- Grenzwert für n → inf:  $U_{tot} = \ln * 2 \approx 0.69$
- konkrete Anwendungen: Auslastung bis 90 Prozent realistisch

#### 4.6.2 Deadline Scheduling

Tasks zum richtigen Zeitpunkt starten resp. beendet  
**mögliche Deadlines:**

- Start Deadline → Zeitpunkt zu dem ein Task gestartet werden muss
- Completion Deadline → Zeitpunkt an dem der Task beendet sein muss

**Earliest Deadline Scheduling:**

- Priorität umgekehrt proportional zur Zeit bis Deadline
- Scheduling aufwendiger als bei RMS → Prioritäten müssen dynamisch angepasst werden
- Theoretische Auslastung bis zu 100 Prozent möglich

#### 4.6.3 Cyclic Executives

- statisches Scheduling
- non-preemptive
- Schedule für periodischen Hauptzyklus (100ms) → mehrere Nebenzyklen
- Realisierung: Aufruf von Prozeduren

# Kapitel 5

## Vorlesung 6 - Synchronisation

Die Begrifflichkeiten *Prozesse und Threads* werden hier als *Tasks* zusammengefasst

### 5.1 Sie können Problemstellungen um Nebenläufigkeit und Parallelverarbeitung aufzählen und diskutieren

- nebenläufige Tasks nutzen im Auftrag gemeinsam Daten und Ressourcen
- Nicht koordinierter Zugriff auf Daten → mind. ein Task kann inkonsistente Sicht der Daten haben
- Resultate nebenläufiger Berechnungen → Abhängig vom zeitlichen Ablauf der Tasks

Dabei kann eine sogenannte **Race Condition** auftreten:

- mind. zwei Tasks greifen auf gemeinsame Daten zu
- mind. ein Zugriff ist ein Schreibzugriff
- Das Resultat hängt von Ausführungsreihenfolgen der beteiligten Tasks ab

#### 5.1.1 Nebenläufigkeit

Das Betriebssystem unterstützt die gleichzeitige Ausführung von mehreren Programmen / Tasks. Eine Ausführung kann concurrent und/oder parallel erfolgen

Dabei unterscheidet man zwischen unabhängig und abhängig: **unabhängig**:

- keine gegenseitige Beeinflussung
- Nutzung von vers. Ressourcen
- keine Synchronisation

**abhängig**

- Programm kooperiert
- nutzen von Daten und Ressourcen sind gemeinsame
- Zugriff muss synchronisiert bzw. koordiniert werden

## 5.2 Sie können das Konzept des kritischen Abschnitts erklären und diskutieren

Als kritischer Abschnitt gilt, wenn ein Zugriff auf gemeinsame Daten / Ressourcen erfolgt. Wobei der Aufenthalt im kritischen Abschnitt gegenseitig auszuschliessen (Mutex) sein muss.

→ Zu jedem Zeitpunkt befindet sich höchstens ein Task im kritischen Abschnitt und der Zutritt zum kritischen Abschnitt muss genehmigt werden.

### 5.2.1 Struktur der Lösung

- entry section → Eintrittserlaubnis bzw. Eintrittssperre
- kritischer Abschnitt → Zugriff auf gemeinsame Daten und Ressourcen
- exit section → Eintritt freigeben
- gegenseitiger Ausschluss → mutual exclusion (Mutex)

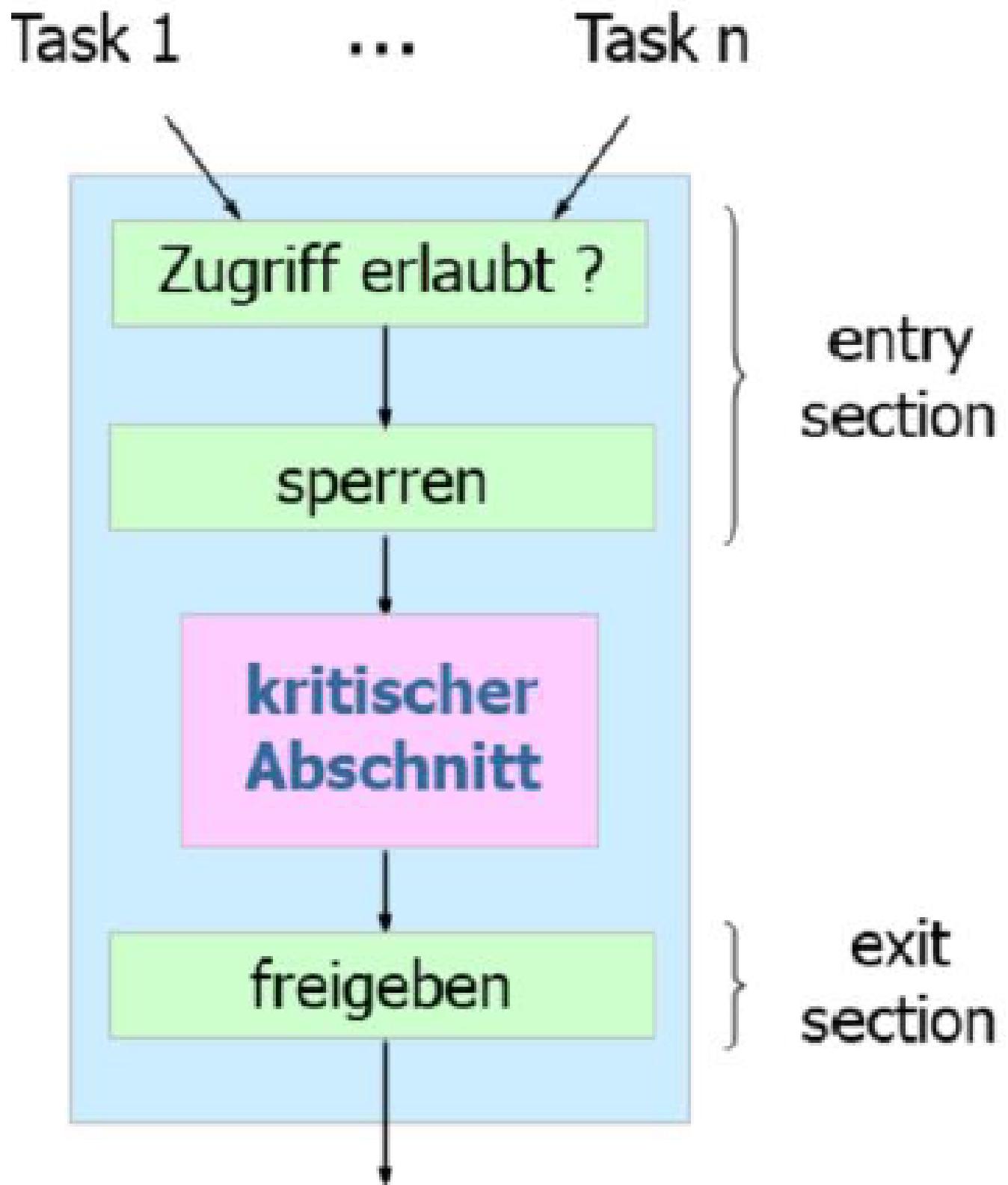


Abbildung 5.1: Struktur der Lösung eines kritischen Abschnittes

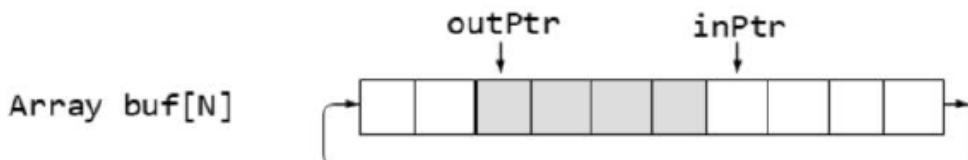
### 5.2.2 Anforderung an die Lösung

- 1 jeweils nur eine Aufgabe in ihren kritischen Abschnitt aufgenommen werden darf
- 2 es werden keine Annahmen über relative Aufgabengeschwindigkeiten oder die Anzahl der Prozessoren getroffen
- 3 ein Task, die in ihrem unkritischen Teil stehen bleibt, muss dies tun, ohne andere Aufgaben zu behindern
- 4 ein Task, die Zugang zu einem kritischen Abschnitt erfordern, dürfen nicht auf unbestimmte Zeit verzögert werden
- 5 kein Prozess befindet sich in einem kritischen Abschnitt: jede Aufgabe, die den Eintritt in die Krippenabteilung verlangt, muss ohne Verzögerung zugelassen werden
- 6 ein Task darf nur für eine begrenzte Zeit innerhalb ihres kritischen Abschnitts bleiben

## 5.3 Sie können das Consumer/Producer und Readers/Writers Problem erklären und diskutieren

### 5.3.1 Producer-Consumer Problem

Es geht um die koordinierte Datenausgabe → Mehrere Producer-Tasks produzieren Ausgabedaten und ein Consumer-Task übernimmt die Daten und gibt sie aus



Erzeuger (Anzahl: N)

```
while (true) {
    item = produceItem();
    while ((inPtr+1)%N == outPtr) {}
    buf[inPtr] = item;
    inPtr = (inPtr+1)%N;
}
```

Verbraucher (Anzahl: 1)

```
while (true) {
    while (outPtr == inPtr) {}
    item = buf[outPtr];
    outPtr = (outPtr+1)%N;
    consumeItem(item);
}
```

Hinweis: Code nicht synchronisiert

Abbildung 5.2: Beispiel einer Lösung für das Producer-Consumer-Problem

### 5.3.2 Readers-Writer-Problem

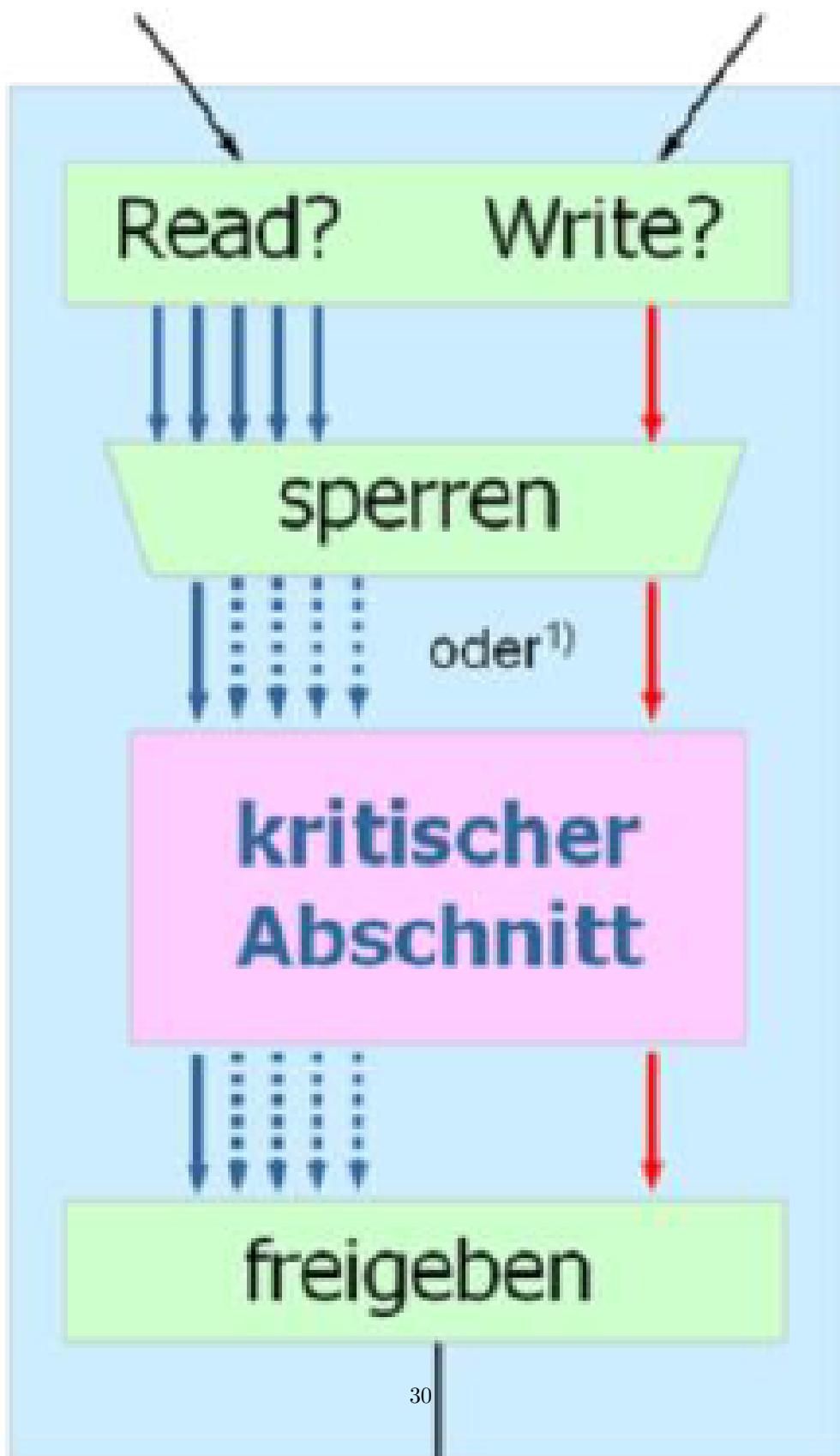
Bei gemeinsamen Daten wird häufiger gelesen, als geschrieben (bspw. Datenbanken, Hash-Tabellen etc.). Dabei gibt es einen erweiterten kritischen Abschnitt.

- mehrere Reader gleichzeitig im kritischen Abschnitt
- maximal ein Writer und kein Reader im kritischen Abschnitt

Task 1

...

Task n



1) oder: exklusiv

## 5.4 Sie können Mechanismen für die Implementation von kritischen Abschnitten und zur Lösung von Synchronisations-Problemen aufzählen und erklären (Hardwareunterstützung für Synchronisationsmechanismen; Softwarelösungen; Semaphore, Monitore, Transactional Memory)

Dies kann mit unterschiedlichen Ansätzen erfolgen:

- Reine Softwarelösungen → Software-Algorithmen, deren Korrektheit von keinen weiteren Bedingungen abhängt
- Hardwarelösungen → Unterstützung durch spezielle Maschineninstruktionen
- Lösungen mit Hilfe des Betriebssystems → Stellen dem Anwender Funktionen und Datenstrukturen zur Verfügung
- Lösungen zusammen mit Programmiersprachen → Monitore, z.B. Java

### 5.4.1 Softwarelösung

#### Busy-wait

Beim Busy-wait wartet der Task aktiv, d.h. er konsumiert Rechenzeit, bis der Scheduler einen Task-Switch durchführt  
→ auch Spinlock genannt.

- auf Uniprozessoren problematisch
- je nach Anwendung auf Multicore-Prozessoren sinnvoll
- gegenseitiger Ausschluss wird erreicht, aber die Tasks können nur abwechselnd den kritischen Abschnitt durchlaufen
- ⇒ Der Task hindert sich außerhalb des kritischen Abschnitts selbst, den kritischen Abschnitt zweimal hintereinander zu betreten (→ *Widerspruch der Forderung 3*)

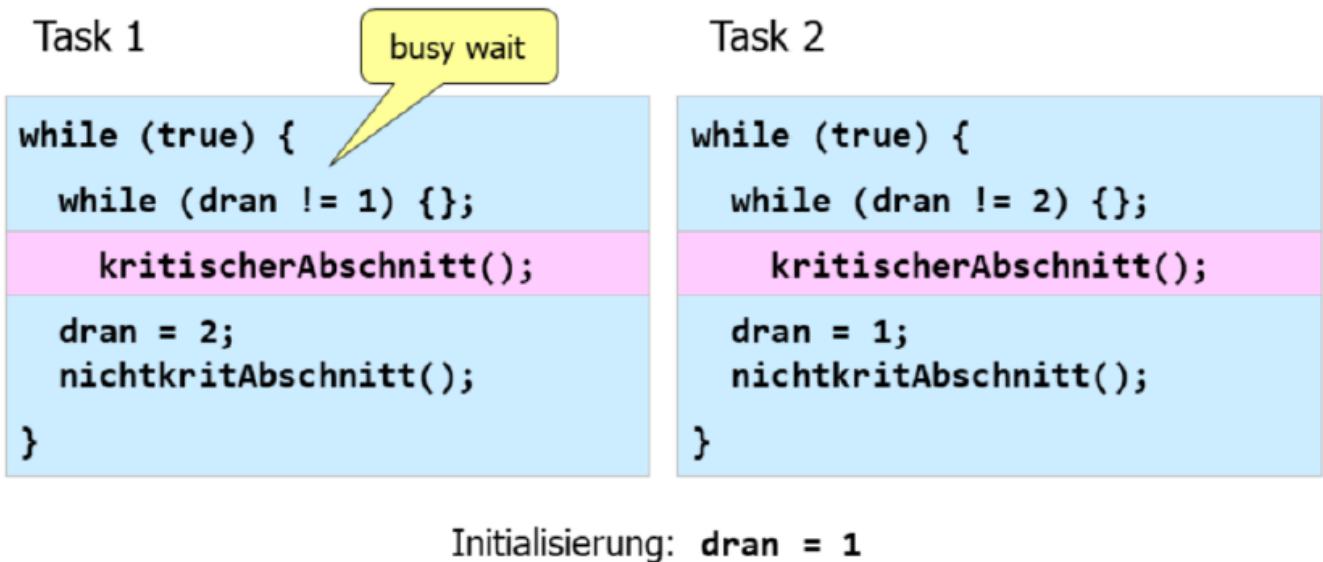


Abbildung 5.4: Lösungsansatz mit Busy-Wait

### Peterson 1981

- sinnvoll bei unkritischem zeitlichen Ablauf
- ist nur für zwei Tasks geeignet
- busy-wait → Prozess wartet in Rechenschleife auf Freigabe des kritischen Abschnitts
- **Wichtig!** Funktioniert nicht auf modernen Multicore-Prozessoren

#### Task 1

```
    . . .
interesse[1] = true;
dran = 2;
while ((dran == 2) &&
       (interesse[2] == true)) {};
kritischerAbschnitt();
interesse[1] = false;
    . . .
```

#### Task 2

```
    . . .
interesse[2] = true;
dran = 1;
while ((dran == 1) &&
       (interesse[1] == true)) {};
kritischerAbschnitt();
interesse[2] = false;
    . . .
```

Initialisierung: **dran = 1; Interesse[1] = Interesse[2] = false**

Abbildung 5.5: Lösungsansatz mit Peterson 1981

#### 5.4.2 Hardwarelösungen

verhindert, dass laufende Programme an beliebigen Stellen unterbrochen werden können. Dazu gibt es unterschiedliche Möglichkeiten:

- Interrupt ausschalten
- atomare Intrusktionen für TAS (Test And Set) und CAS (Compare And Swap)

→ man stösst auf ähnliche Probleme wie bei der Softwarelösungen ⇒ Hardware ist nur eine Unterstützung keine Lösung

To-Do: Müssen diese Variante noch weiter im Detail angeschaut werden?

# Kapitel 6

## Vorlesung 7 - Deadlocks

### 6.1 Sie können die Begriffe Deadlock, Lifelock und Starvation erklären und diskutieren

#### 6.1.1 Deadlock

Beim Deadlock (zu Deutsch: Verklemmung) warten Tasks auf die gegenseitige Freigabe von Ressourcen  $\Rightarrow$  Kein Task arbeitet weiter, es geht nicht mehr weiter

##### Voraussetzung

1. Mutual exclusion  $\rightarrow$  mind. eine Ressource ist exklusiv reserviert
2. Hold and wait  $\rightarrow$  mind. ein Task hat eine Ressource exklusiv reserviert und wartet auf weitere Ressourcen
3. No preemption  $\rightarrow$  reservierte Ressourcen können dem Task nicht entzogen werden
4. Circular wait  $\rightarrow$  geschlossene Kette von Tasks existiert, in der jeder Prozess mind. eine Ressource reserviert hat, die auch von einem Nachfolger in der Kette benötigt wird

$\Rightarrow$  Es müssen alle Bedingungen gegeben sein, damit der Deadlock eintritt

#### 6.1.2 Lifelock

Beim Lifelock arbeitet der Task zwar weiter, jedoch ist die Arbeit nicht produktiv  $\Rightarrow$  ungewollter busy-wait

#### 6.1.3 Starvation

Bei der Starvation (zu Deutsch: Verhungern) erhält der Prozess keinen Zugriff auf Ressourcen, dies kann beispielsweise bei einer unfairen Zuweisungspolicy unter anderem bei einem Stack (FILO-Prinzip)  $\Rightarrow$  Abhilfe kann durch eine faire Policy geschaffen werden, bspw. FIFO

### 6.2 Sie können die Möglichkeiten zur Verhinderung von Deadlocks aufzählen und diskutieren

Dabei gibt es drei Möglichkeiten

- Keine Deadlocks zulassen  $\rightarrow$  entweder verhindern oder vermeiden
- Deadlocks zulassen  $\rightarrow$  Wird gelöst wenn er eintritt
- Problem ignorieren  $\rightarrow$  Annahme dass keine Deadlocks auftreten

## Verhindern

Mind. eine der vier Bedingungen darf nicht eintreten  
⇒ ineffiziente Ressourcennutzung, serialisierte Verarbeitung

## Vermeiden

Ressourcen nicht zusprechen, wenn Deadlockgefahr besteht  
⇒ Alles ausser Circular Wait zulassen  
⇒ Ressourcenanforderung → überprüfen ob ein Deadlock eintreten könnte, evtl Task nicht starten  
⇒ System in sicherem Zustand → Ressourcenzuweisung führt nicht zu Deadlock

## Erkennen

Deadlocks zulassen, bei Auftreten die Situation lösen oder das System muss periodisch auf Deadlocks untersucht werden  
⇒ Betriebssystem überprüft ob ein Deadlock aufgetreten ist, falls ja Deadlock auflösen und lauffähigen Zustand wiederherstellen  
⇒ Die Überprüfung erfolgt wenn ein Task auf Ressourcen warten muss, zu einem spezifischen Intervall oder wenn die CPU Auslastung unter einen bestimmten Wert sinkt

*Recovery Strategies:*

1. alle beteiligten Tasks stoppen → Meist verwendete Strategie
2. alle beteiligten Tasks auf Checkpoint zurücksetzen → Rollback-Mechanismus notwendig, Risiko dass Deadlock wieder auftritt
3. beteiligte Tasks der Reihe nach stoppen, bis Deadlock gelöst (bspw. wenigsten CPU, wenigsten Output, wenigsten alloziert, kleinste Prio, längste geschätzte Rechenzeitn) → Kriterium zur Wahl, nach jedem Stopp Detektionsalgorithmus neustarten
4. beteiligten Tasks Ressourcen wegnehmen, bis Deadlock gelöst (bspw. wenigsten CPU, wenigsten Output, wenigsten alloziert, kleinste Prio, längste geschätzte Rechenzeitn) → Kriterium zur Wahl, Rollback zum Punkt vor Ressourcenallokation

## 6.3 Sie können Ressourcengrafen erklären, diskutieren und Ressourcengrafen für das Finden von Deadlocks einsetzen

Ist ein gerichteter Graf und stellt die Ressourcenzuordnung dar.

⇒ Wenn kein Zyklus vorhanden ist, gibt es auch keinen Deadlock ⇒ Wenn ein Zyklus vorhanden ist, gibt es:

- eine Instanz pro Ressource → Deadlock
- mehrere Instanzen pro Ressource → Deadlock möglich

### 6.3.1 Elemente eines Ressourcengrafs

**Kreis:** stellt einen Task dar

**Rechtecke:** stellen die Ressourcen dar

**Punkte in Rechteck:** jede Instanz entspricht einem Punkt innerhalb des Rechtecks

**Pfeil von Ressource zu Task:** der Task hat Ressource alloziert

**Pfeil von Task zu Ressource:** der Task fordert Ressource an

#### **6.4 Sie können abschätzen, ob Deadlocks bzw. Starvation kritisch oder unkritisch sind**

Wo steht das? :)

# Kapitel 7

## Vorlesung 8 - Internprozess-Kommunikation

### 7.1 Sie können den Begriff Interprozesskommunikation erklären und erläutern

Prozesse arbeiten zusammen um gemeinsam eine Aufgabe zu lösen bzw. gemeinsame Ressourcen zu nutzen → Hierfür ist Datenaustausch notwendig

Gründe für die Zusammenarbeit:

- Parallelverarbeitung → Leistungssteigerung, Benutzerfreundlichkeit
- vereinfachte Strukturierung von Anwendungen → mehrere kooperierende, aber kleine Prozesse
- Daten- und Informationsaustausch → einfacher Zugriff auf gemeinsame Daten
- Echtzeitsysteme → Aufgaben mit unters. Repetitionsrate

Es können zwei mögliche Lösungsszenarien eintreten (wird anhand eines Druckauftrages aufgezeigt):

1. Synchronisation → Nur ein Prozess darf drucken, die anderen müssen warten
2. IPC → es gibt einen Druckprozess, welcher die Daten entgegen nimmt und anschliessend ausdruckt, wobei die Prozesse selber nicht drucken dürfen, darf weiterarbeiten können

### 7.2 Sie können Message Passing erklären und diskutieren

Hierbei geht es um den Austausch von Nachrichten innerhalb eines oder verteilten Systeme. Das Betriebssystem stellt die Zugriffsfunktion zur Verfügung und die Synchronisation wird durch diese Zugriffsfunktion definiert. Dabei ist der Anwender verantwortlich für das Verpacken der Daten in Nachrichten (bspw. Message Queues)

- Sehr häufiges Verfahren → zwischen Prozessen auf einem Rechner, verteilten Systemen, wobei die Synchronisation implizit ist
- Grundfunktion *send(dest, message)* und *receive(src, message)* ⇒ beide Funktionen können blockieren
- auch geeignet für reine Prozess-Synchronisation oder Implementation eines Mutex

## Prozess P<sub>1</sub>

```
do {  
    ...  
    send(dest,msg);  
    ...  
} while (!finished);
```

## Prozess P<sub>2</sub>

```
do {  
    ...  
    receive(src,msg);  
    ...  
} while (!finished);
```

Abbildung 7.1: Beispiel eines Message Passing

### 7.2.1 Verbindungsarten

Synchrone bzw. verbindungsorientierte Kommunikation → beide Prozesse müssen der Verbindung zustimmen

P1:

```
openConnection(adr)
    send(msg);
closeConnection(adr);
```

P2:

```
openConnection(adr)
    receive(msg);
closeConnection(adr);
```

Abbildung 7.2: Beispiel einer synchronen Message Passing

Asynchrone bzw. verbindungslose Kommunikation → Sender schickt Nachricht einfach los. Allenfalls ist dabei eine Quittierung notwendig (Sache der Applikation)

### asynchron

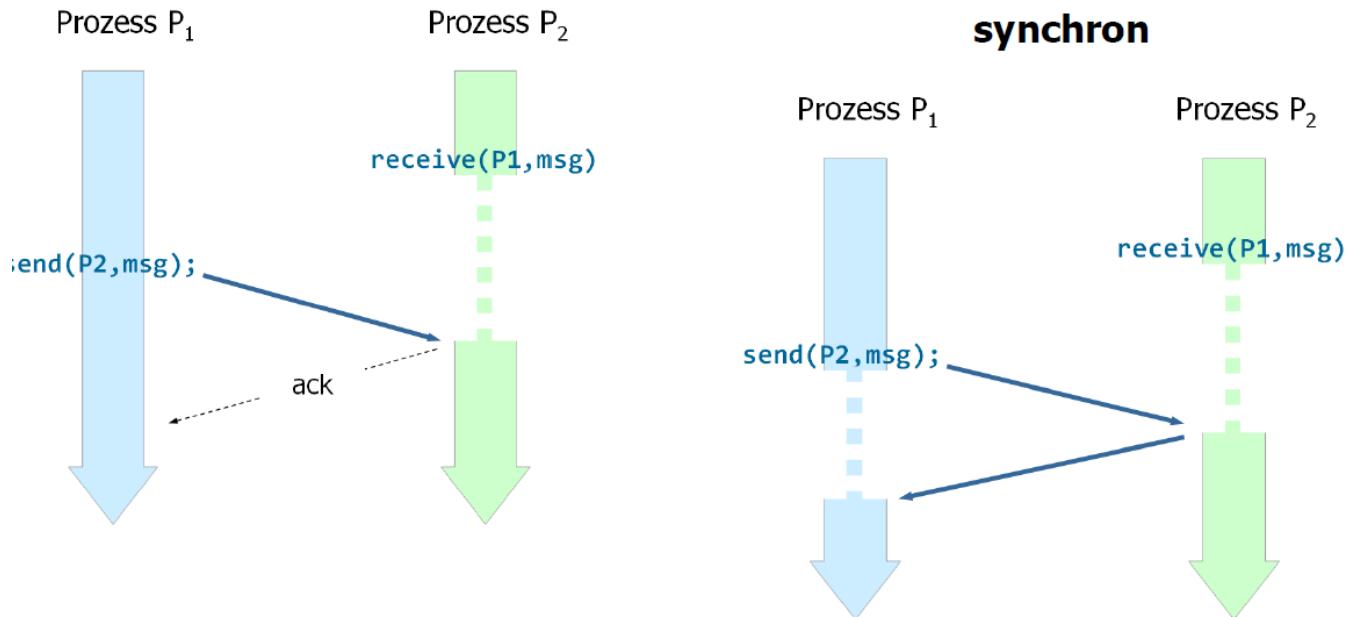


Abbildung 7.3: Unterschied synchron vs. asynchrones Message Passing

### 7.2.2 Adressierung

#### direkte Adressierung

- Nachricht wird direkt in den Adressraum (Speicher) des Empfänger kooperiert
- Adressierung kann *explizit* (Quelladresse wird beim Empfänger einer Nachricht angegeben) oder *implizit* (Quelladresse kann nicht angegeben werden) erfolgen
- Sender und Empfänger sind eng gekoppelt
- Vorteil: geschützter Datenverkehr



Abbildung 7.4: Beispiel einer direkten Adressierung

#### indirekte Adressierung

- Nachricht wird nicht direkt an den Empfänger gesendet
- Nachricht wird an eine Mailbox (bzw. Port) gesendet (Queue)
- Sender und Empfänger sind entkoppelt
- Vorteil: vers. Verbindungstopologien möglich

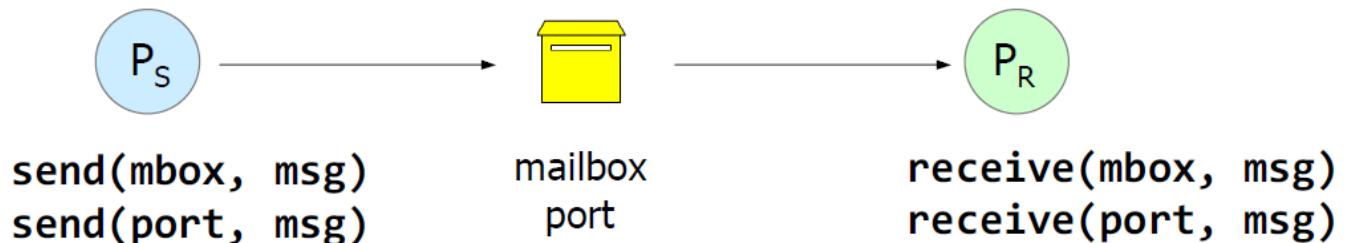


Abbildung 7.5: Beispiel einer indirekten Adressierung

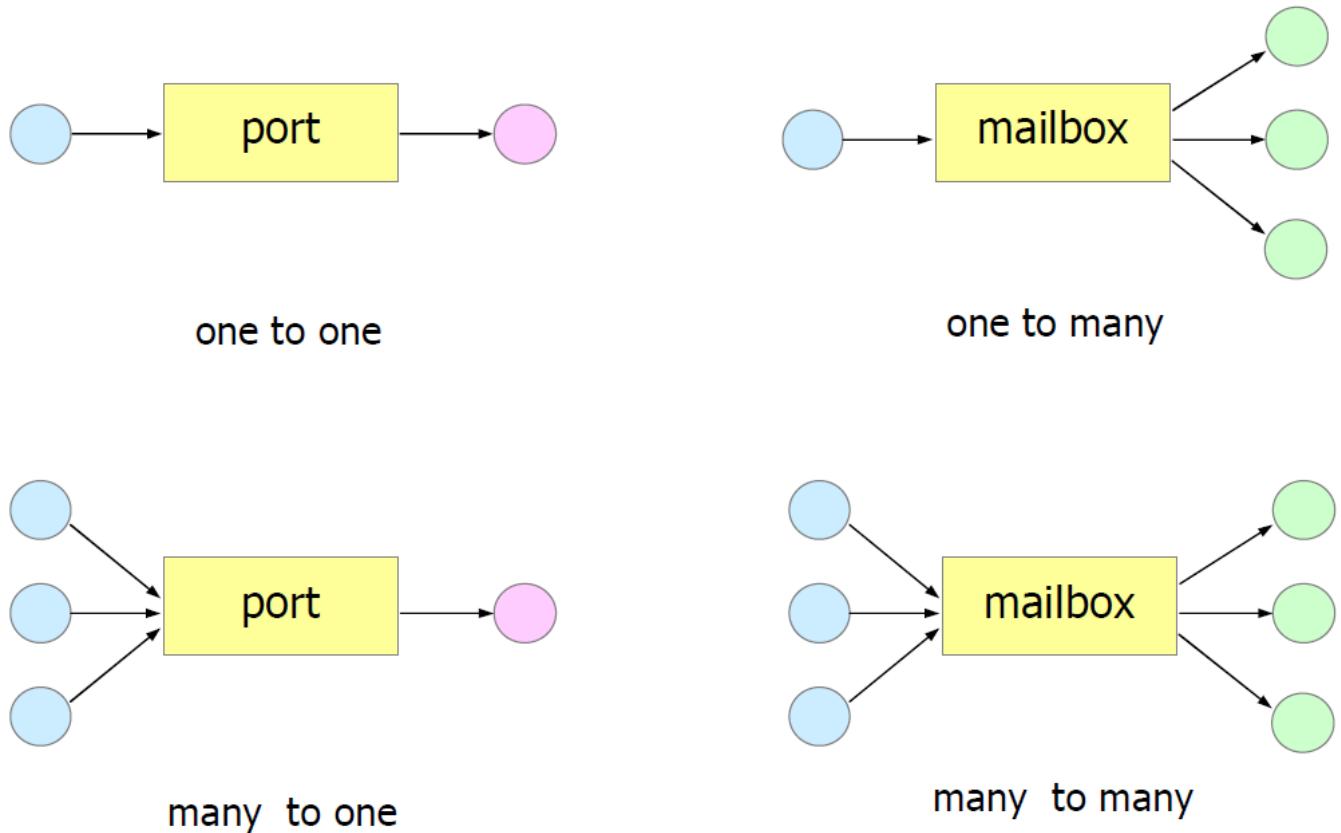


Abbildung 7.6: Vers. Verbindungsvarianten

Mailbox V.06 S. 12 relevant? Nachricht V.06 S. 13 relevant?

### 7.3 Sie können Shared Memory erklären und diskutieren

Hierbei geht es um einen gemeinsamen Speicher, welches meist innerhalb eines gemeinsamen Systems ist. Das Betriebssystem stellt den gemeinsamen Speicherbereich zu Verfügung.

Dabei ist der Anwender verantwortlich für Datenstruktur und Synchronisation.

Der Bereich eines physikalischen Speichers wird in den (virtuellen) Addressraum der Prozesse abgebildet. Der physikalische Bereich kann an vers. Orte im virtuellen Adressraum eingebunden werden

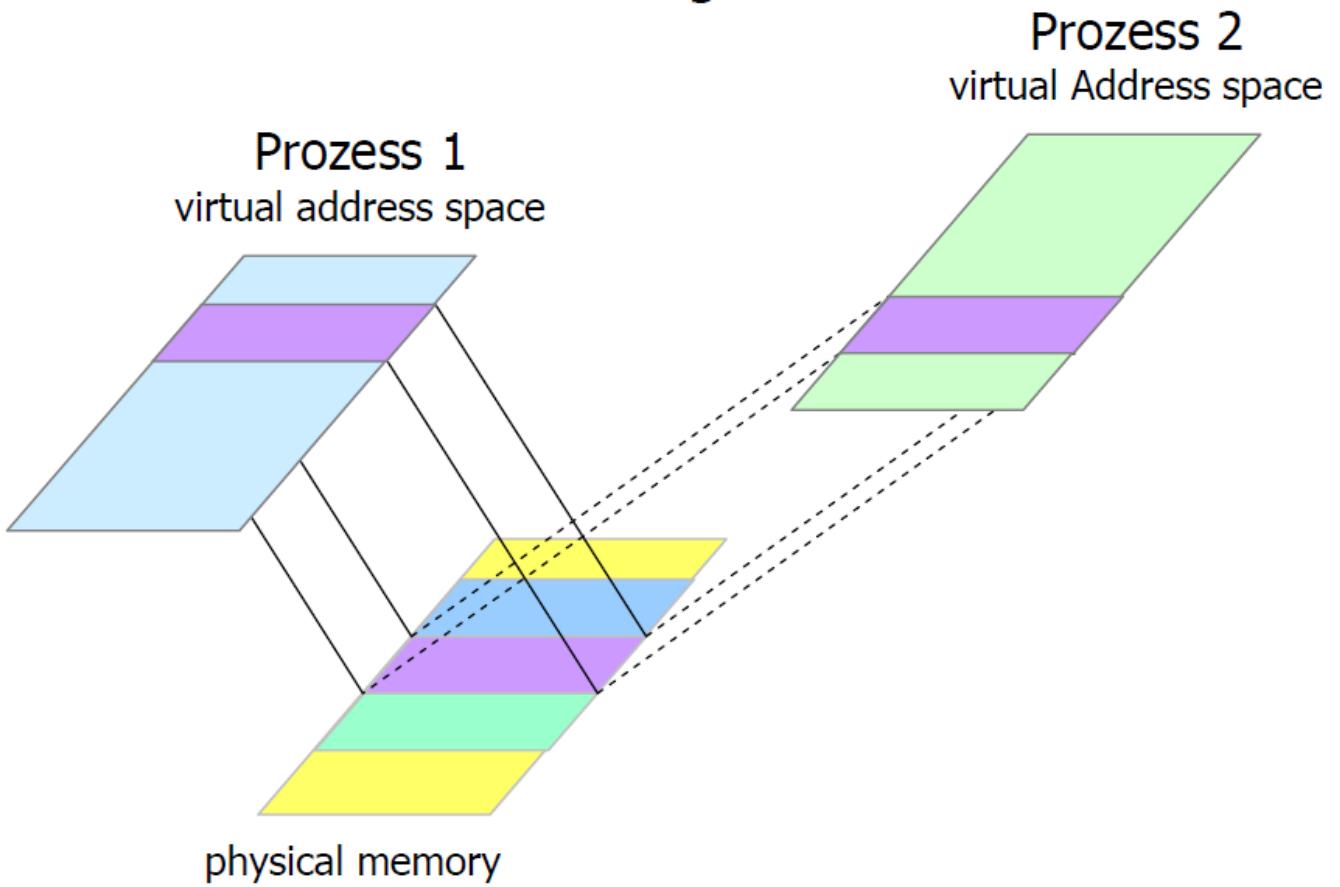


Abbildung 7.7: Abbildung shared Memory

### 7.3.1 Shared Memory vs. Message Passing

#### Shared Memory

- Muss bei Betriebssystem angefordert werden
- keine implizite Synchronisation
- Zugriff sehr schnell: Speicherzugriff
- nur in Shared Memory Systemen verfügbar

#### Message Passing

- Mailbox muss bei Betriebssystem angefordert werden
- implizite Synchronisation
- langsamer als Shared Memory
- auch in verteilten Umgebungen verfügbar

## 7.4 Sie können Unix/Linux IPC Mechanismen aufzählen, erklären und diskutieren

POSIX steht für *Portable Operating System Interface*  
**Interprozesskommunikation**

- IPC Ressourcen
- Shared Memory
- Message Queues
- Signale
- Pipes
- Sockets
- Shared Files / Memory Mapped Files

**Code Beispiel für die einzelnen Varianten  
Synchronisation**

- Semaphore
- Lock Files

## 7.5 Sie können Implementationsaspekte anhand der Unix/Linux IPC Mechanismen erklären und diskutieren

# Kapitel 8

## Vorlesung 9 - Memory Management, Virtual Memory

Das Memory Management ist für die Speicherverwaltung im Betriebssystem zuständig, da sich der Programmierer und Anwender nicht um den Speicher kümmern will.

**Uniprocessing:** Ein Prozess und das OS stehen im Speicher

**Multiprogramming:** Mehre Prozesse und das OS stehen im Speicher; div. Verwaltungsprozesse; bessere CPU-Nutzung

Dazu gehören folgende Einheiten

- Logical Organisation → logischer Adressraum, lineare Folge von Bytes (words) ⇒ Was sieht der Anwender
- Physical Organisation → physikalische Realisierung (Cache-, Haupt-, Sekundärspeicher) ⇒ Was sieht das Betriebssystem
- Protection → verhindern, dass sich Prozesse gegenseitig beeinflussen
- Sharing → gemeinsame Speicherbereiche zur Verfügung stellen
- Relocation → verschieben eines Prozesses an bel. Ort im Speicher

### 8.1 Sie können die Begriffe logische und physikalische Adresse erklären und diskutieren

**logische Adresse:** Ist eine Referenz auf Speicherplatz, unabhängig von Speicherorganisation

**physikalische Adresse:** Ist eine Referenz auf physikalischer Speicherplatz

Dabei erzeugt der Compiler Code mit relativen (logischen) Adressen

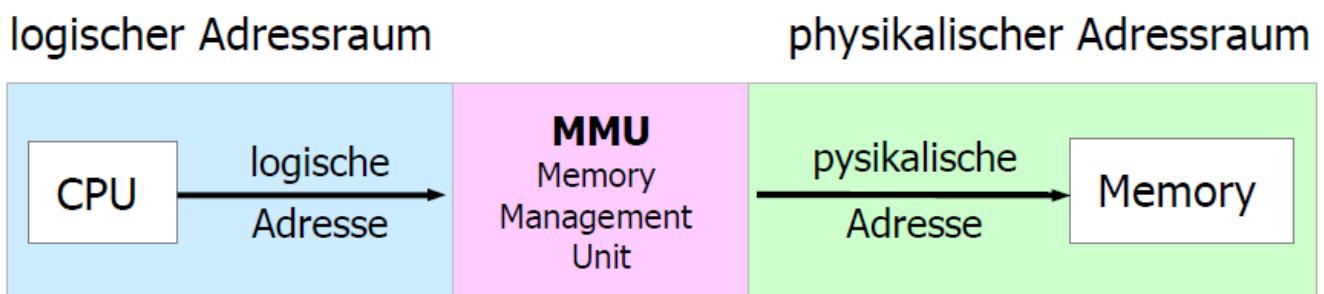


Abbildung 8.1: Abbildung logische vs. physikalische Adressen

(a) Adressübersetzung

- CPU erzeugt logische bzw. virtuelle Adresse
- logische Adressen sind relativ, meist bezogen auf Adresse 0
- muss schnell und transparent sein → Hardwareunterstützung notwendig

## 8.2 Sie können erklären, um was es beim Swapping geht

Die Problematik ist, dass oft nicht alle Prozesse im Speicher Platz haben, aus diesem Grund lagert man einen Prozess vom Speicher auf Disk bzw. umgekehrt aus. Dies gilt auch für Teile eines Prozesses (virtual Memory).  
 → ausgelagerter Prozess ist suspendiert ⇒ Heutzutage steht viel Speicher zur Verfügung, aus diesem Grund können auch ganze Prozesse im Speicher stehen.

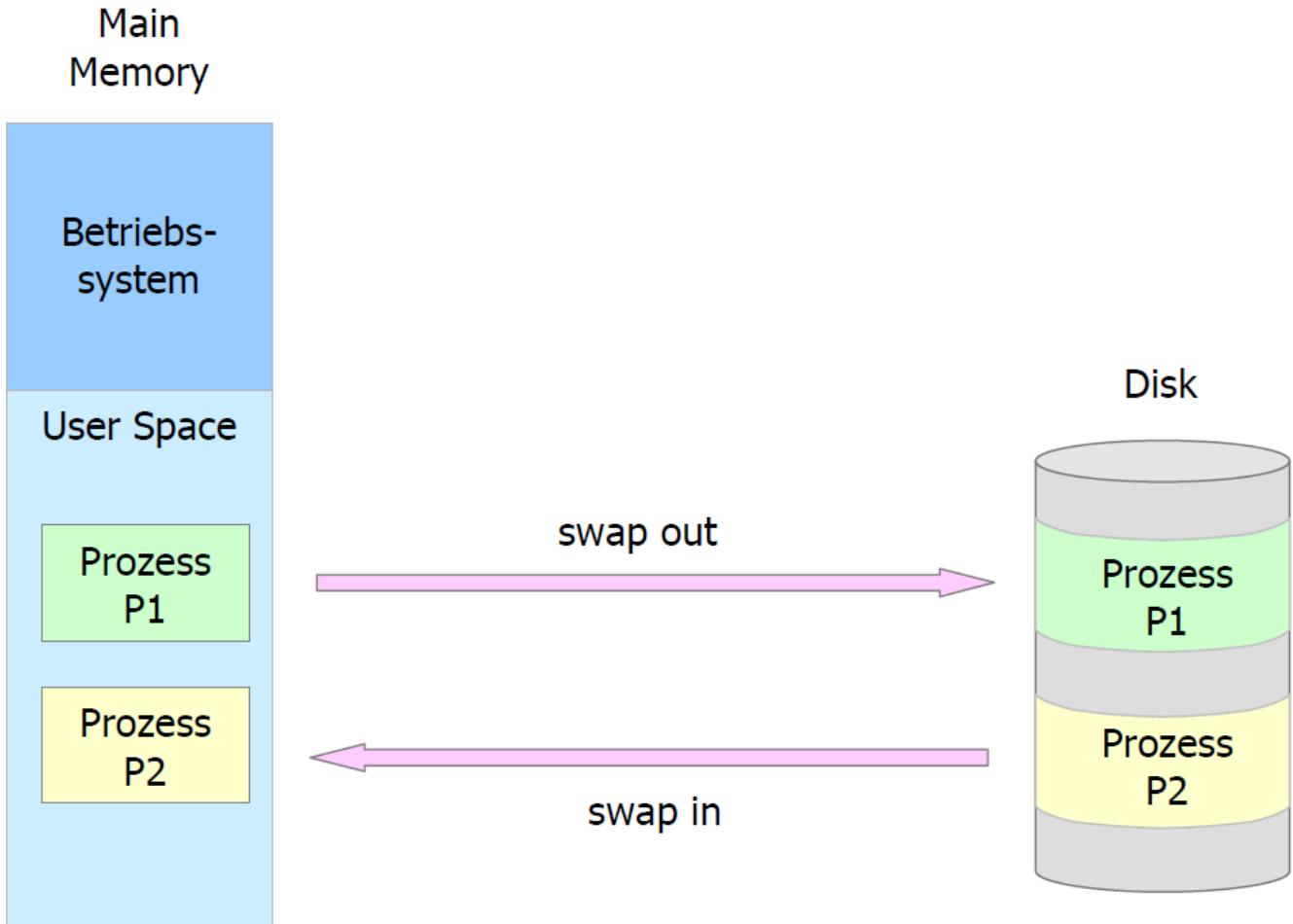


Abbildung 8.3: Abbildung Swapping

## 8.3 Sie können die grundlegenden Verfahren des Memory Managements aufzählen und diskutieren

### 8.3.1 Ansatz 1: einfaches Memory Management

- ganzer Prozess im Speicher
- traditioneller Ansatz

- einfaches Paging
- einfache Segmentation
- typische Anwendung: 'Kleine Systemen' → Embedded Systems, Real-Time Systeme

### Voraussetzung

- gesamter Prozess steht im Hauptspeicher
- unterstützt Multiprogramming
- für kleine / einfache Systeme

### mögliche Verfahren

- Addressraum zuteilen → fixed partitioning; dynamic partitioning; placement
- Addressraum aufteilen → paging; (Segmentation)

### fixed partitioning

- Aufteilung des Hauptspeichers in mehrere nicht überlappende Partitionen → OS belegt feste Partition; vorgegebene Anzahl Prozesse im Speicher; BS kann Prozesse auslagern
- Gleich grosse Partitionen → jedes Programm, egal wie gross, belegt eine Partition; *internal Fragmentation* wird eine nicht vollständig gefüllte Partition genannt; Nutzung des Hauptspeichers ist ineffizient
- verschiedene grosse Partitionen → reduzieren, aber lösen nicht das Problem; V1: jede Partition eine Prozess-Queue (interne Fragmentierung minimieren, Queues bleiben leer); V2: für alle Partitionen eine Prozess Queue (besseres Multiprogramming, erhöht interne Fragmentierung)

Partitionen  
gleich gross

Partitionen  
verschieden gross

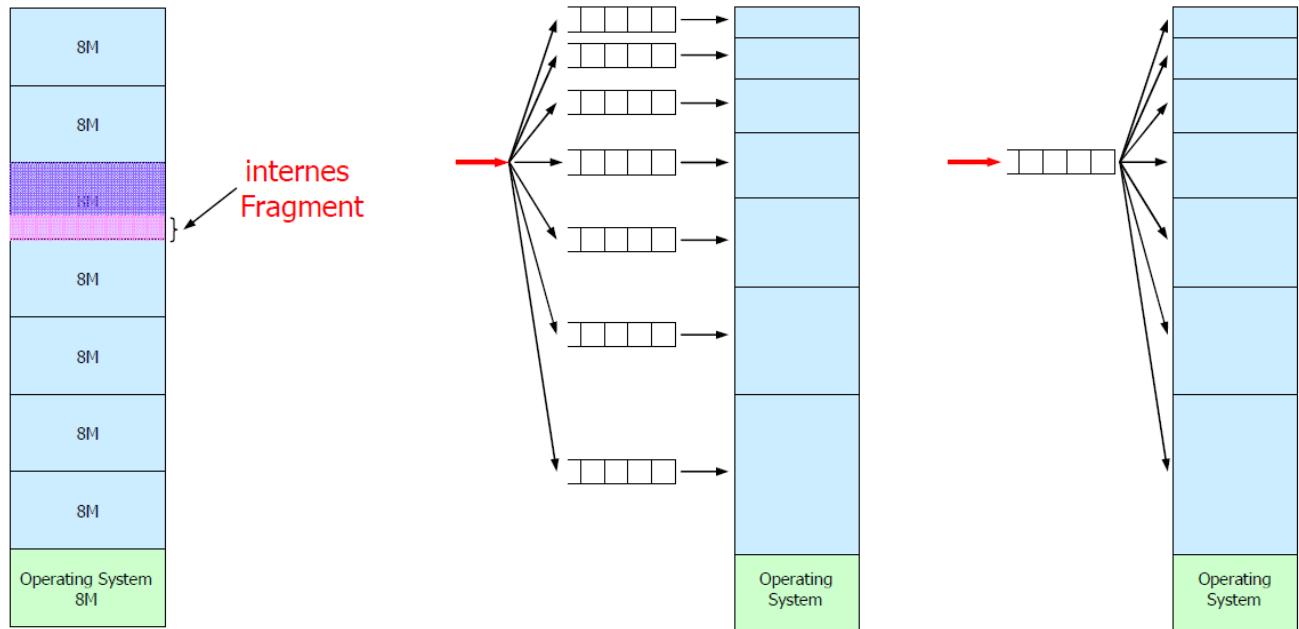


Abbildung 8.4: Abbildung fixed partitioning

## Dynamic Partitioning

- Größe und Anzahl der Partitionen ist variabel → jedem Prozess so viel Speicher zuweisen wie er benötigt
- Problem: externe Fragmentierung, da Prozesse ausgelagert werden und nicht immer durch gleich grosse Prozesse ersetzt werden
- ⇒ Compaction ist notwendig → Prozesse verschieben, bis Löcher geschlossen

Die Zuweisung kann durch verschiedene *Placement Algorithmen* erfolgen

- First Fit → einfacher und i.d.R. schnellster und bester Algorithmus ⇒ Tendenziell weniger Fragmentierung als bei Next-Fit
- Next Fit → alloziert oft freien Block am Schluss des Speichers, am Ende oft Blöcke am grössten ⇒ Compaction öfters notwendig als bei First Fit
- Best Fit → sucht kleinste, passende Blöcke, minimiert externe Fragmente zum nächsten Block, entstehen schnell viele kleine Fragmente ⇒ schlechterer Algorithmus (Compaction oft durchführen als bei First/Next Fit)

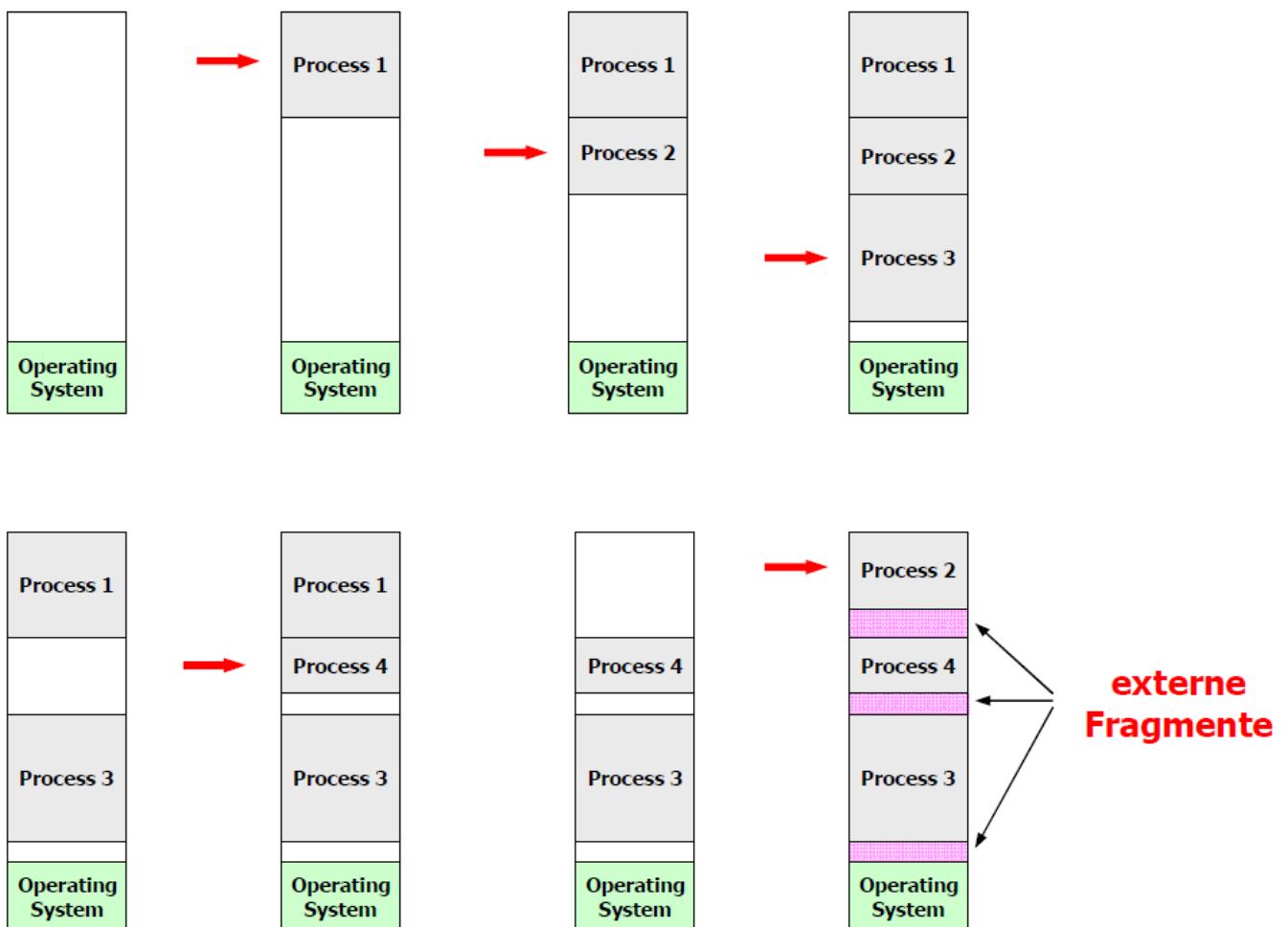


Abbildung 8.5: Abbildung dynamic partitioning

## Buddy System

- Kompromiss zwischen fixed und dynamic partitioning
- schneller Allokations- und Deallocationsalgorithmus
- Unix/Linux verwendet modifizierte Buddy Systeme

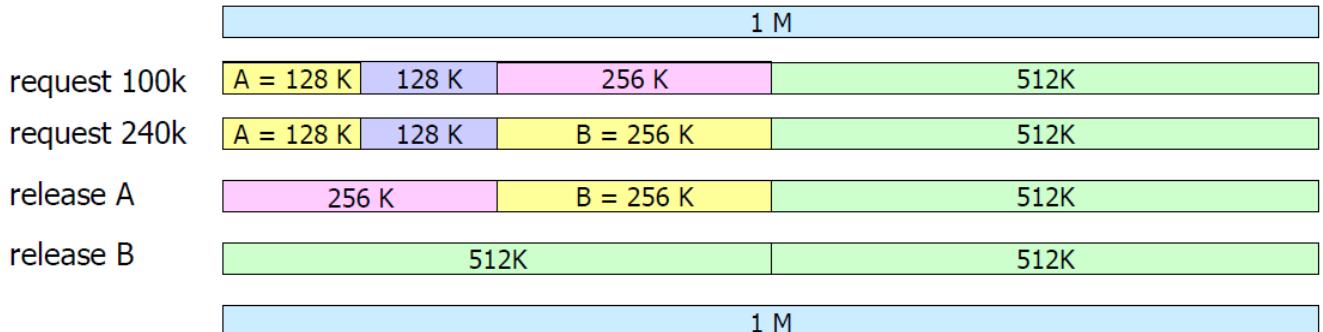


Abbildung 8.6: Abbildung Buddy System

### Algorithmus:

- es werden solange Blöcke halbiert, bis ein Block minimaler Grösse zur Verfügung steht
- Daten zum den freien Blöcken lassen sich mit einem Binärbaum darstellen
- effiziente Algorithmen verfügbar (bspw. Stallings)

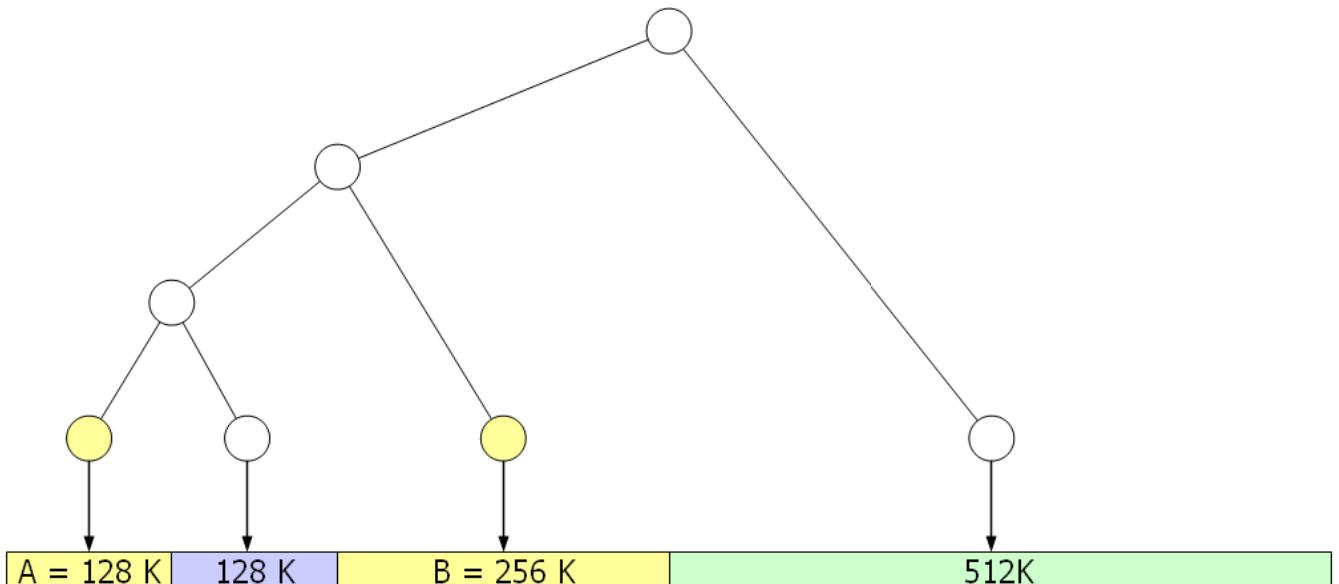


Abbildung 8.7: Abbildung eines Buddy System Binärbaums

### 8.3.2 Ansatz 2: Virutal Memory

- aktiver Teil des Prozesses im Speicher → Der Rest ist auf den Sekundärspeicher ausgelagert
- Paging
- virtualisiert den Speicher

- Hard- und Softwareunterstützung notwendig
- typische Anwendung in 'grossen Systeme' → Workstation, Server, PCs

## Sicht des Programmierers

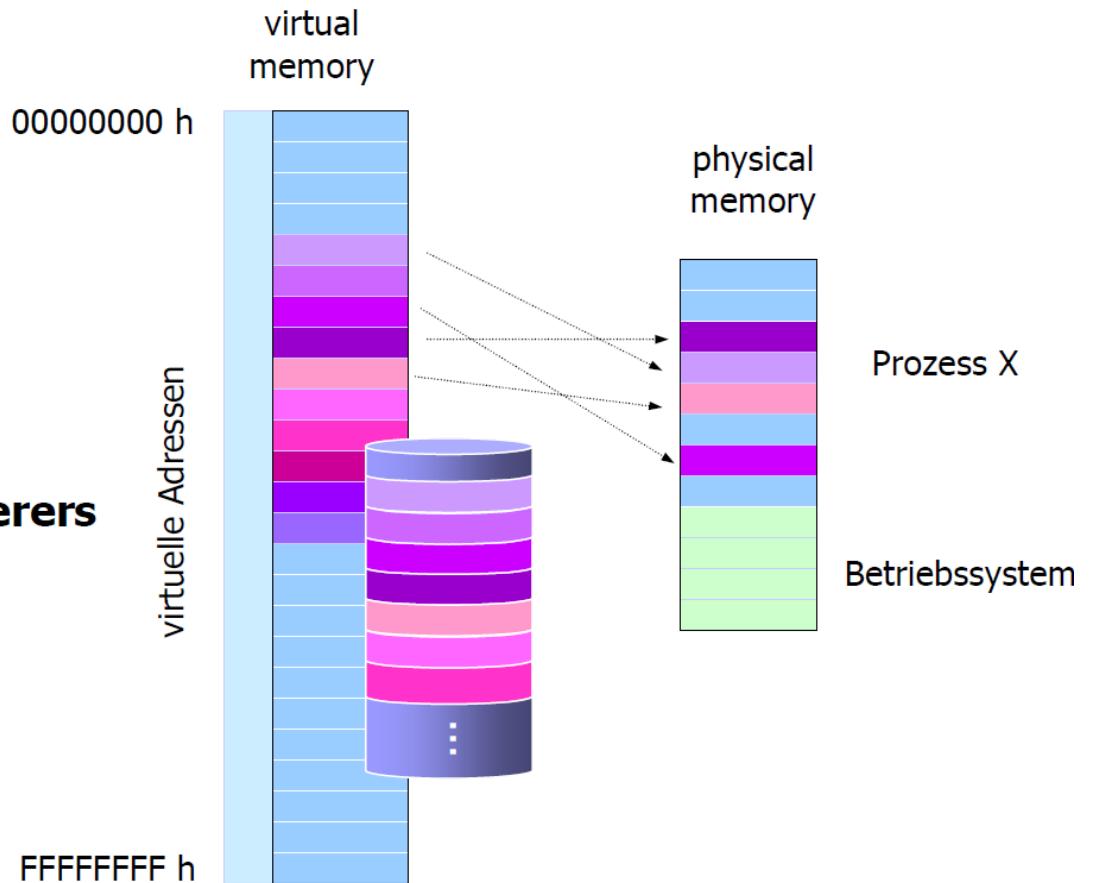


Abbildung 8.8: Abbildung von virtual memory

- logischer resp. virtueller Adressraum in PAgEs aufgeteilt
- physikalischer Addressraum entsprechend in Frames aufteilt
- nur aktuell benötigte Pages im Hauptspeicher
- Rest der Pages als Image auf Disk

**Wie funktioniert VM:** Präzessor referenziert virtuelle Adresse

**Wieso funktioniert VM:** Virtual Memory basiert auf dem **Lokalitätsprinzip** → Speicherzugriffe liegen nahe beieinander, gleiche Instruktionen werden mehrmals ausgeführt

**Welche Pages stehen im Speicher:** BS muss bei Page-Fault intelligente Entscheidung treffen → OS versucht zu raten, welche Pages als nächste gebraucht werden

### Page Table

Das Realisieren von Page Table ist aufwendiger und benötigt zusätzliche Einträge

- im virtuellen Memory kommen Page Tables zum Einsatz
- Realisierung aufwendiger, da zusätzliche Bits notwendig sind (siehe Abbildung)
- Adressübersetzung ist ähnlich wie beim Paging

- Page Tables werden gross
- Evtl. mehrere Speicherzugriffe notwendig → langsam
- Lösungen erfordern in der Regel Hardwareunterstützung
- Realisierung mittels Multilevel Organisation, Hashed Page Table oder Inverted Page Table

## Virtuelle Adresse



## Page Tabelleneintrag

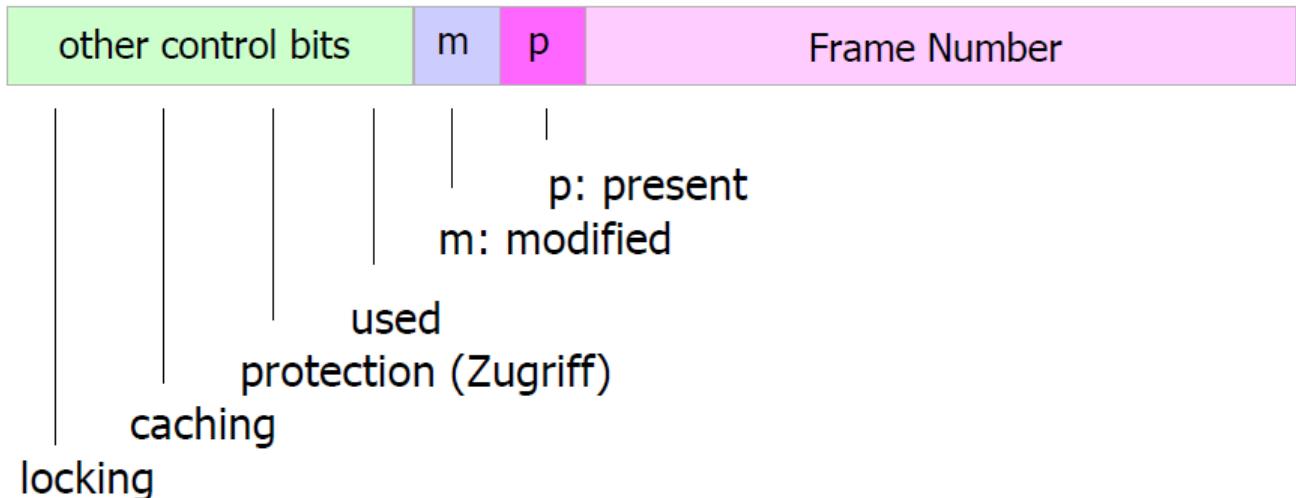


Abbildung 8.9: Abbildung einer Page Table

## Multilevel Organisation

Beispiel gemäss Abbildung

- mehrere Page Directories (Verzeichnisse) und mehrere Page Tables
- 1. Stufe wählt Verzeichnisse
- 2. Stufe wählt Page Table
- minimale Anzahl von Tabellen → ein Page Directory und eine Page Tabelle

## virtual address

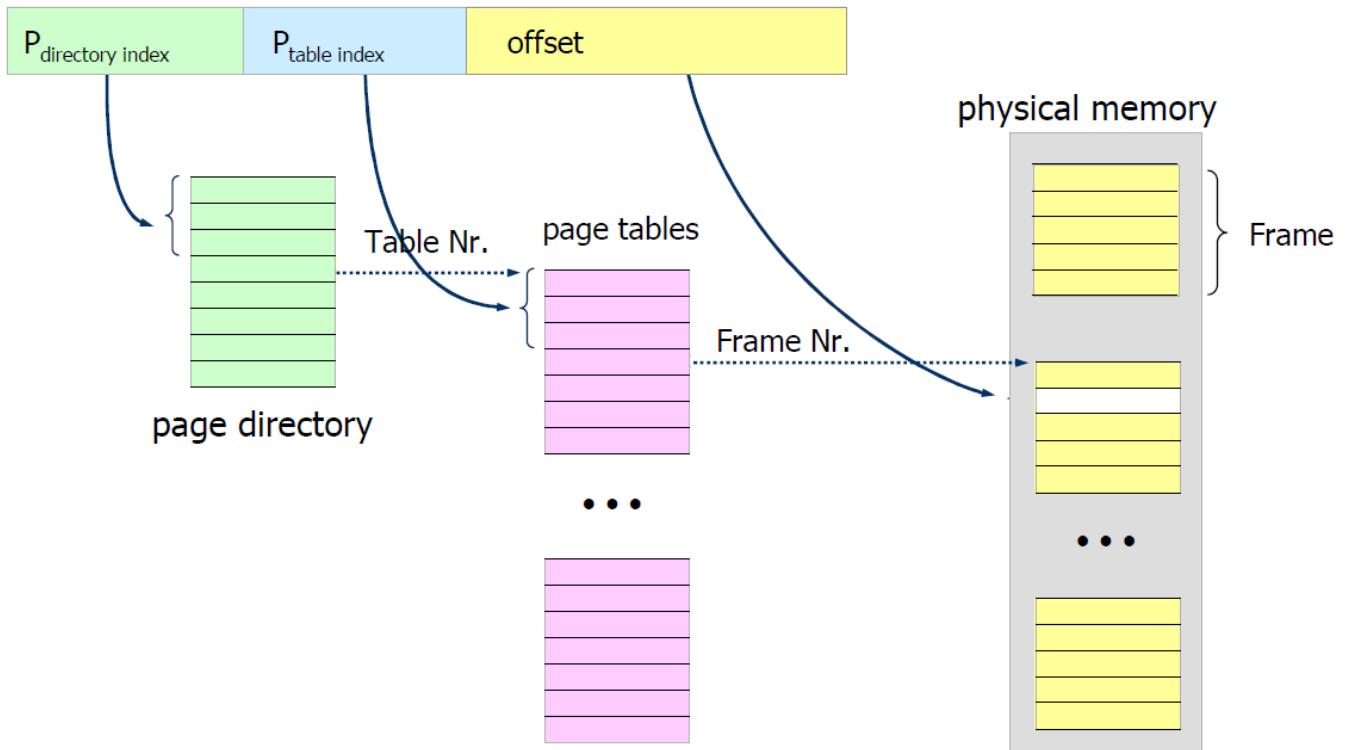


Abbildung 8.10: Abbildung einer Multilevel Organisation

### Hashed Page Table

- Page Nummer wird als Hash-Wert verwendet
- verschiedene Page Nummern erzeugen gleicher Hash-Wert
- Hash-Tabelle: jeder Eintrag zeigt auf eine Liste mit den Einträgen
- Hash-Tabelle: jeder Listeneintrag enthält Page Nummer und entsprechende Frame Nummer
- bei Zugriff muss jeweils Page Nummer verglichen werden (gleicher Hash Wert)
- → zusätzlicher Aufwand, da Suche in Hash-Liste
- geeignet für invertierte Page Tabellen
- geeignet für Adressräume grössser 32 Bit

virtual address

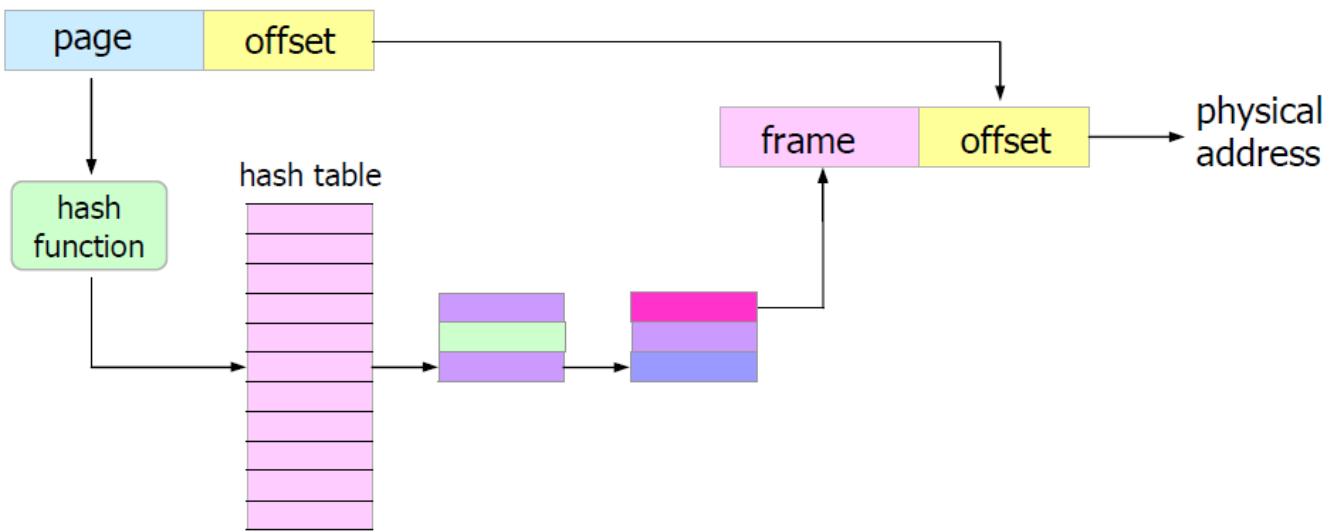


Abbildung 8.11: Abbildung einer Hashed Page Table

### Inverted Page Table

- pro Frame gibt es einen Eintrag
- Nummern des Eintrags  $j = \text{Frame-Nummer}$
- gleiche Page von verschiedenen Prozessen in verschiedenen Frames abgelegt bzw. Prozess ID muss in Tabelle gespeichert werden
- Problem: ganze Tabelle muss nach Page-Nummer und PID abgesucht werden
- Problem: ineffizient, wenn vollständig in Software
- Im Einsatz bei PowerPC, UltraSPARC, ITHANIUM

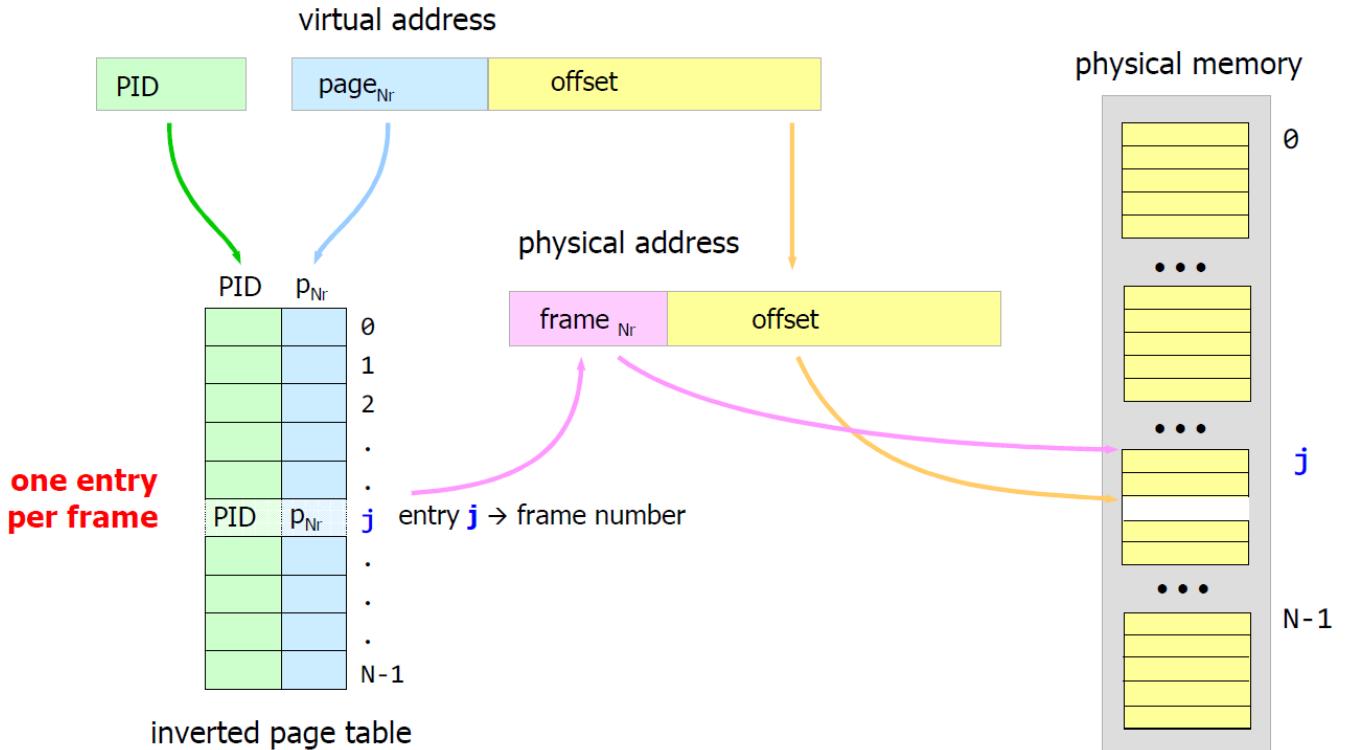


Abbildung 8.12: Abbildung einer Inverted Page Table

## 8.4 Sie können Paging erklären und diskutieren

- logischer Addressraum der Prozesse wird in Pages aufgeteilt
- physikalischer Addressraum wird in Frames (Page Frames) aufgeteilt
- Pagegrösse = Framegrösse

Aktuelle Problematik:

- Fixed Partitioning erzeugt interne Fragmentierung
- Dynamic Partitioning erzeugt externe Fragmentierung und erfordert Compaction

Mit Paging werden

- Prozesse in Blöcke gleicher Länge aufteilt → **Pages**
- Speicher in Blöcke gleicher Länge aufteilt → **Frames**
- Die Zuweisung von Pages zu Frames ist beliebig → Prozess müssen nicht zusammenhängend im physikalischen Speicher stehen

### 8.4.1 Page Replacement

Im Page Replacement werden folgende Fragen beantwortet:

- welche Page ersetzen, wenn Speicher voll ist?
- welche Pages bzw. Frames kommen für Ersatz in Frage?
- wie viele Frames sollen einem Prozess zugewiesen werden?

→ Wenn der Speicher voll: welche Page ersetzen?  
 Dabei kommen vier Replacement Policies ins Spiel:

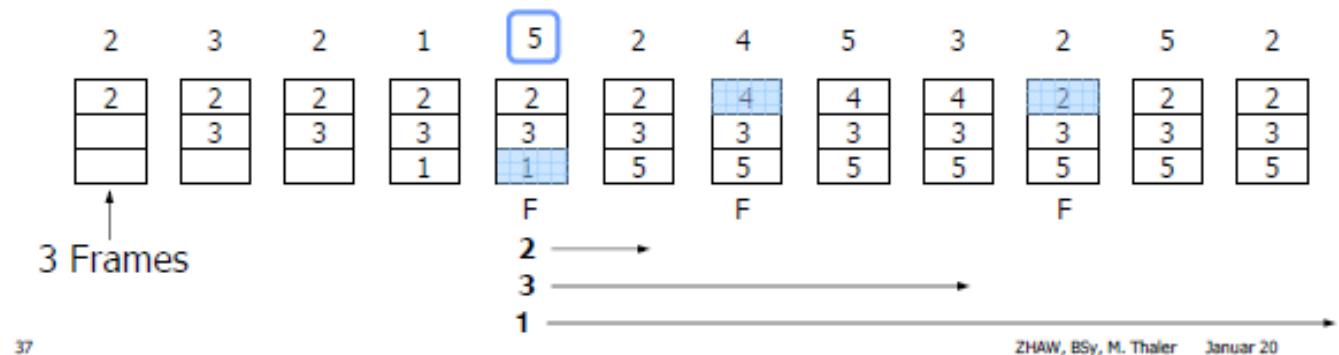
- optimal
- least recently used
- fifo
- clock

Wobei das Bewertungskriterium die minimale Anzahl von Page Faults ist

#### Replacement Policy: optimal

Ersetzt Page, die am spätesten referenziert wird → nicht implementierbar, aber beweisbar, dass minimale Anzahl Page Faults erreicht wurde  
 → gut geeignet für Vergleiche

- **Page Referenzen:** 2 3 2 1 5 2 4 5 3 2 5 2



37

ZHAW, BSy, M. Thaler Januar 20

Abbildung 8.13: Abbildung der ReplacementPolicy: optimal

#### Replacement Policy: least recently used

Ersetzt die am längsten nicht referenzierte Page, welches fast so gut wie optimal funktioniert. Jedoch ist die Implementation aufwendig (TimeStamp oder UsageCounter notwendig)  
 → Funktioniert nach dem Lokalitätsprinzip, wird Wahrscheinlich nicht mehr referenziert

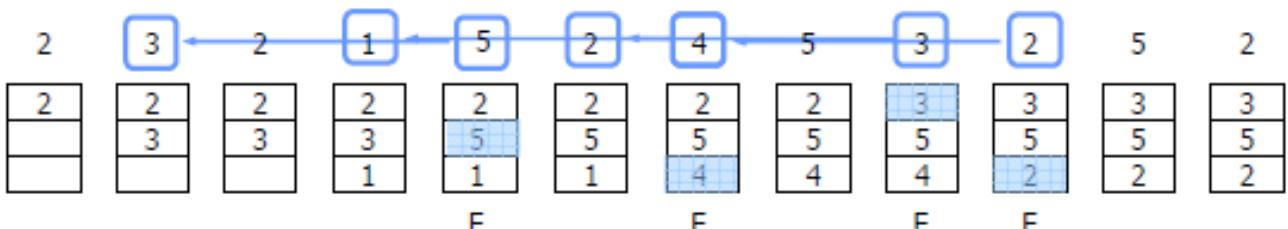


Abbildung 8.14: Abbildung der ReplacementPolicy: least recently used

## Replacement Policy: FIFO

Pages Frames in zirkulärem Buffer angeordnet → älteste Page wird ersetzt, was sehr einfach zu implementieren ist - liefer jedoch schlechte Resultate

→ Problematik, dass auch oft referenzierte Pages ausgelagert werden

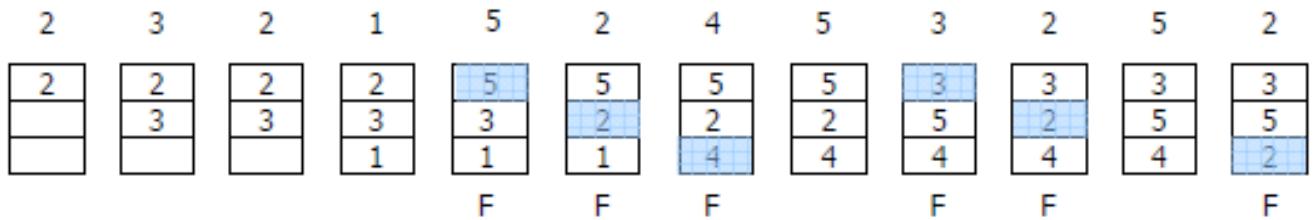
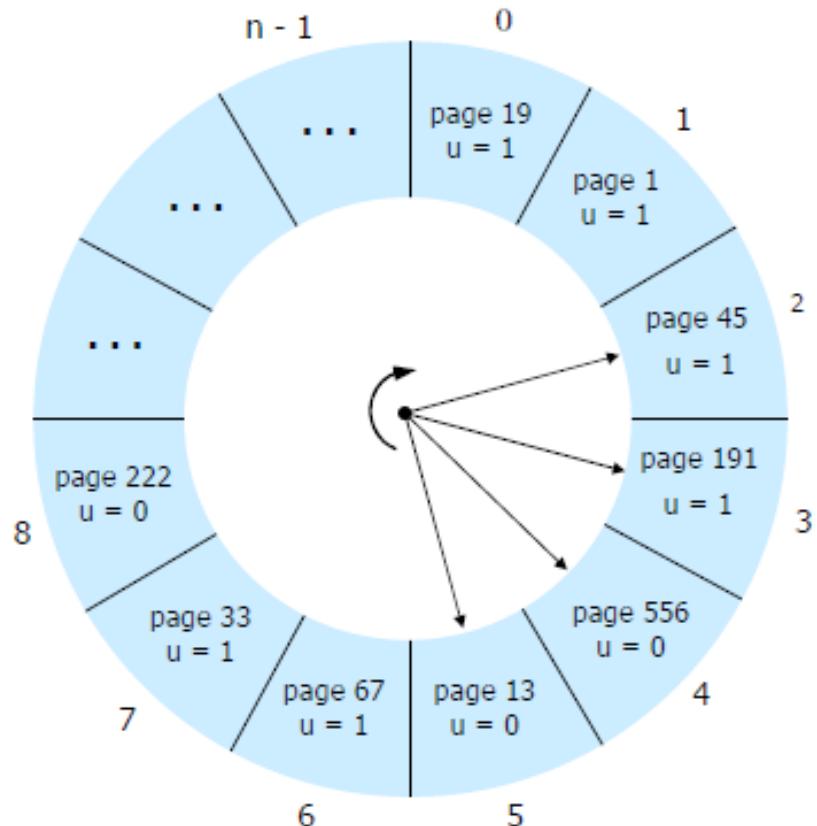


Abbildung 8.15: Abbildung der ReplacementPolicy: FIFO

## Replacement Policy: Clock

- Anordnung in zirkulärem Buffer
- Ersetzt eine Page ausgehend von der Zeigerposition mit UseBit = 0
- UseBit = 1 beim Laden
- schützt häufig referenzierte Seiten
- Kombination von LRU / FIFO

Für Page 727 einen Frame suchen



ZHAW, BSy, M. Thaler Januar 20

Abbildung 8.16: Abbildung der ReplacementPolicy: Clock

## Kapitel 9

# Vorlesung 10 - Input / Output Management

Innerhalb eines PCs gibt es verschiedenste Peripheriebusse

- AGP accelerated graphics port → stellt Grafik schnellen Zugriff auf Hauptspeicher zur Verfügung
- PCI peripheral component interconnect bus → universeller Peripherie Bus
- uvm

In der grossen Gerätevielfalt gibt es nur wenige Konzept wie Geräte an ein Computer angeschlossen werden und wie die Software die Hardware steuert.

Der Anschluss erfolgt über

- Ports → ein Gerät, Datenstrom, z.B. serielle Schnittstelle
- Busse → mehrere Geräte, mehrere Drähte und ein Protokoll

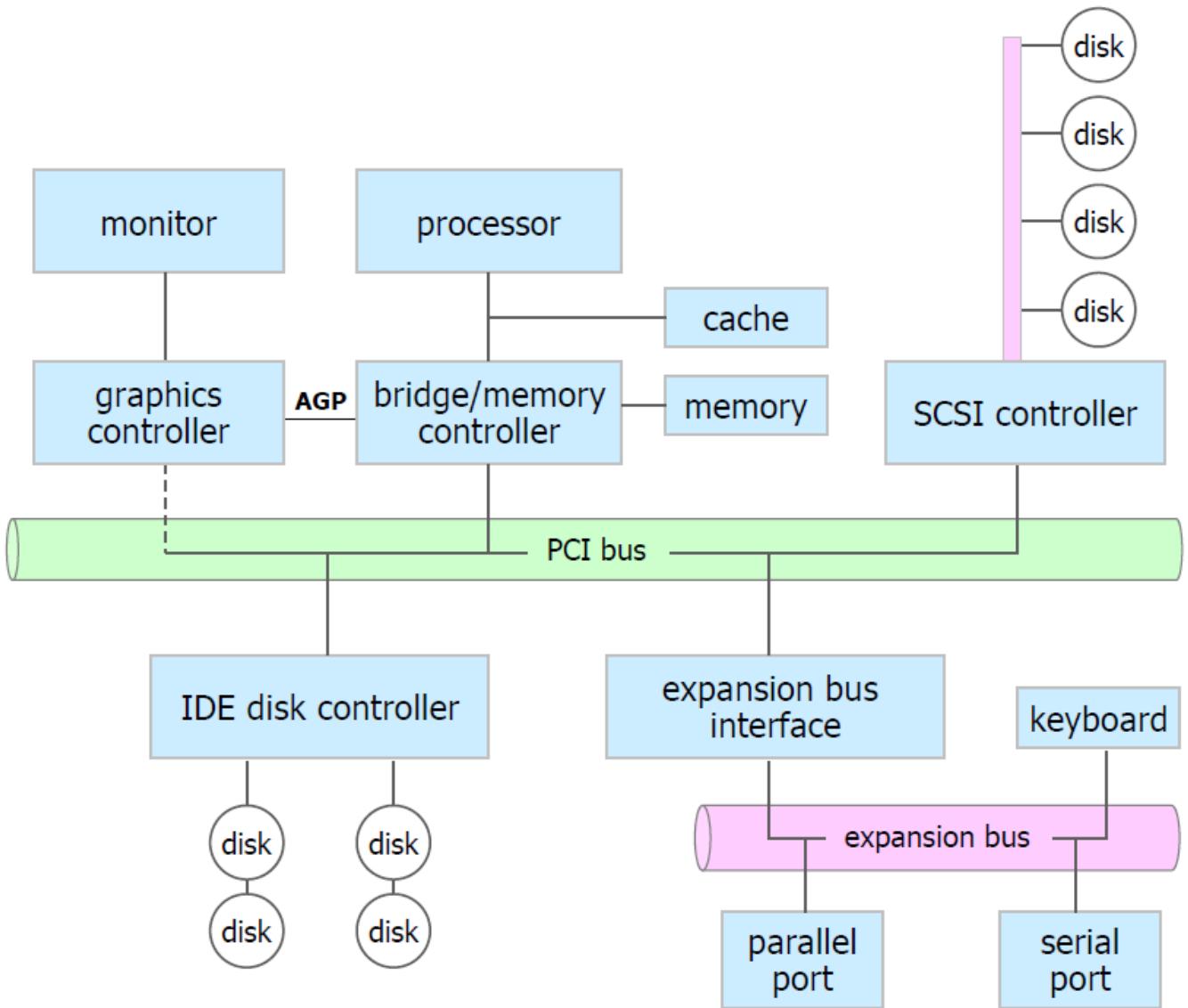


Abbildung 9.1: Abbildung eines PC Architektur

## 9.1 Sie können die grundlegende I/O-Architektur erklären und diskutieren

- Schichtenmodell
- wichtigste Komponente: Device Drivers → standardisierte Schnittstelle zum Kernel I/O-Subsystem und ist intern an Gerätedetails angepasst
- Keine vereinheitlichten Schnittstellen für Device Driver in Betriebssystemen → jedes Betriebssystem benötigt andere Treiber
- Logical I/O → Das Gerät wird als logische Resource betrachtet
- Device I/O → angeforderte Operationen in entsprechende Sequenzen von I/O Instruktionen umwandeln, Buffering

- Scheduling und Control → steuert den Ablauf der Interaktion zwischen Gerät und Software, Interrupt-Handling, IO Status etc.
- API → bei den meisten OS und Geräten

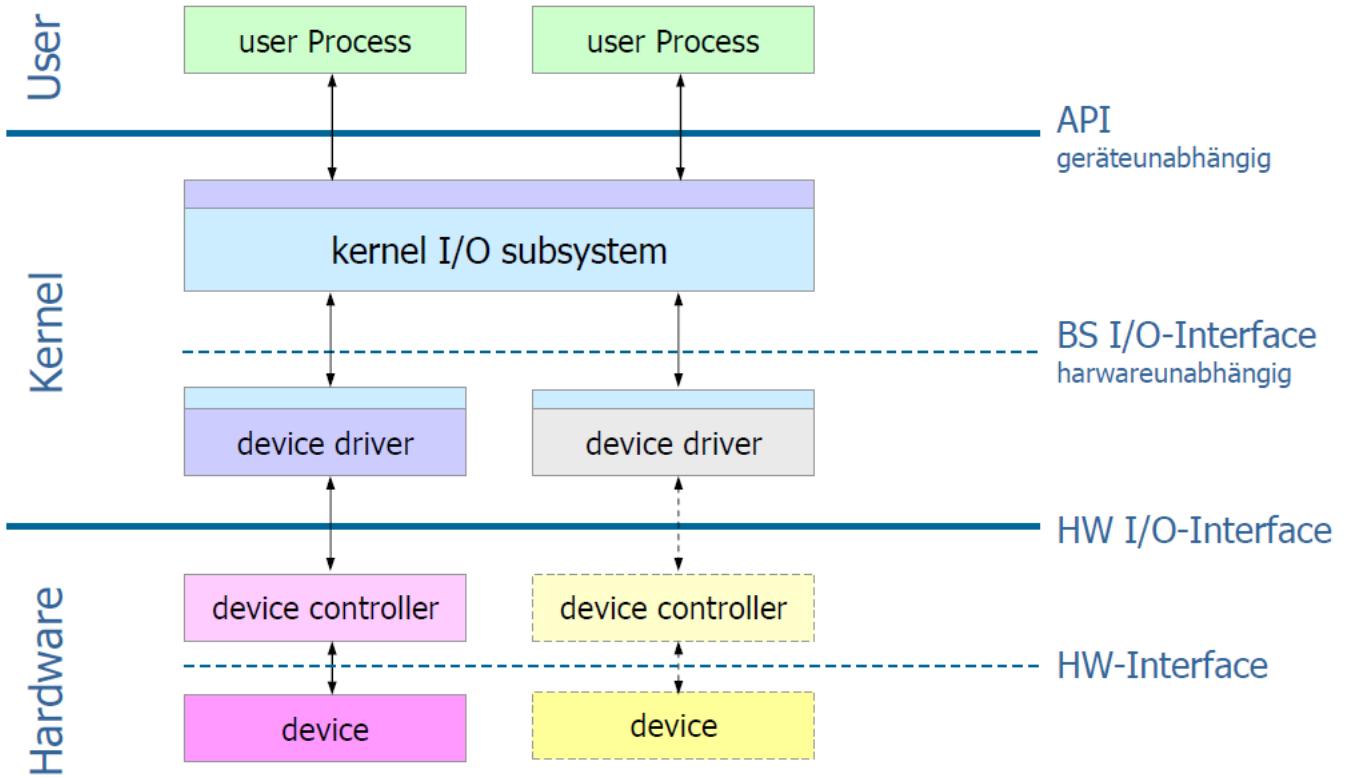


Abbildung 9.2: Abbildung der IO Architektur

### 9.1.1 Kernel I/O Subsystem

- I/O Scheduling → I/O-Anfragen von Anwendungen selten in optimaler Reihenfolge, OS ordnet Anfragen nach Optimierungskriterien (Performance, Fairness, Wartezeit)
- Buffering → Temporärer Speicher für Datenaustausch, da die Übertragungsgeschwindigkeit und Blockgrößen unterschiedlich sind
- Caching → hält Kopie der Daten für schnelleren Zugriff
- Spooling → Speicher mit Daten für Geräte die keine überlappenden Datenströme erlauben bspw. Drucker (oft durch Daemon)
- Reservation → Zugriffsreservation für nur exklusiv allozierbare Geräte

## 9.2 Sie können die Konzepte des Linux Kernel I/O aufzeigen und diskutieren

- Gerätetypen → Block-, Stream- oder Netzwerkgeräte
- Schnittstelle zu Geräten: Filesystem → Geräte werden wie Files angesprochen

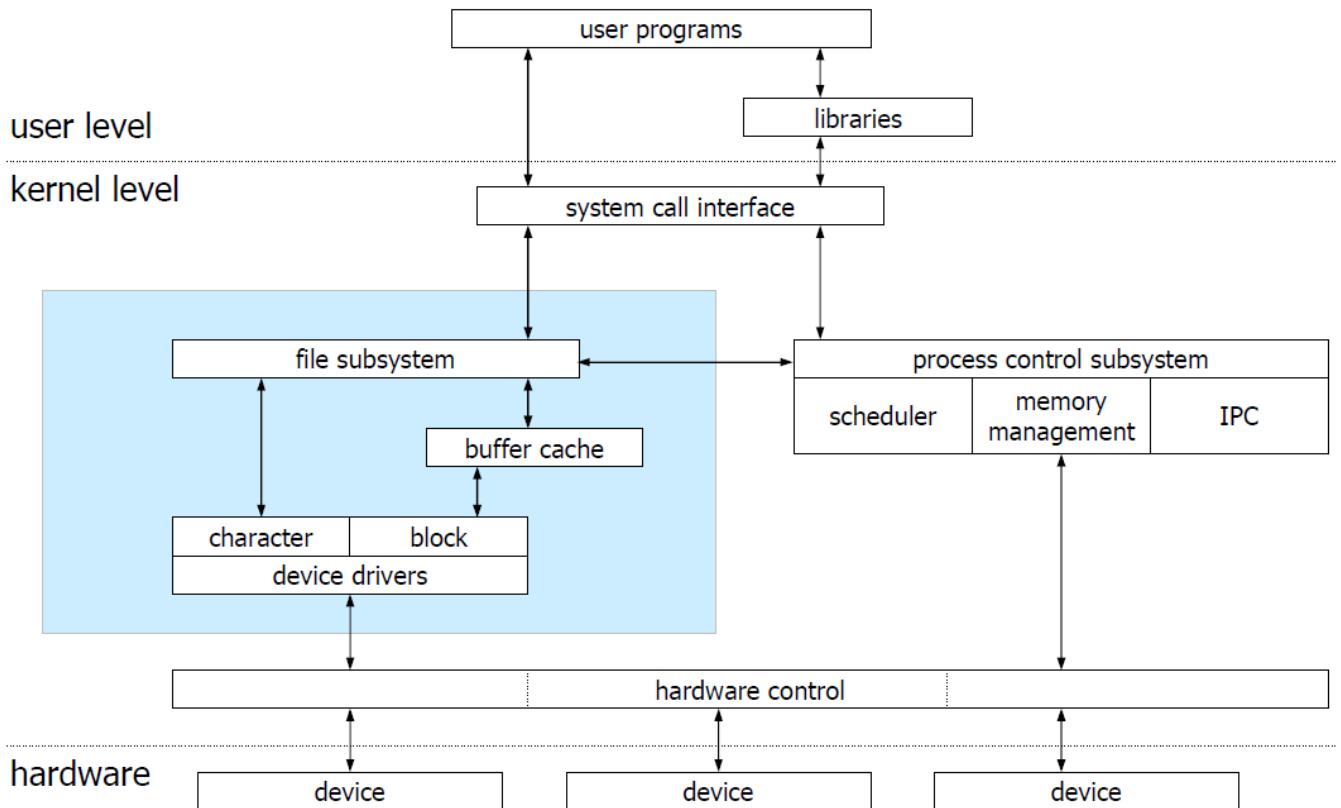


Abbildung 9.3: Abbildung der Linux IO Architektur

### 9.3 Sie können Charakteristiken von I/O Geräten aufzeigen und diskutieren

- Stream → Datenübertragung Byte um Byte bspw. serielle Schnittstelle → charakteristisches Verhalten: spontan erzeugter Inputs
- Block → Datenübertragung als Block von Bytes (bspw. Disk) → charakteristisches Verhalten: random access
- Netzwerkgeräte erhalten Zugriff meist über Sockets

Eigenschaft	Möglichkeiten	Beispiel
Transfer-Modus	character (stream) block network	Terminal Disk Netzwerk (sockets)
Zugriffsmethode	sequential random	Modem Disk
Ablauf	synchron asynchron	Tape Keyboard
Sharing	dedicated sharable	Tape Keyboard
Richtung	read write readonly writeonly	Disk CD-ROM (GPU)
Geschwindigkeit	latency transfer rate delay	Reaktionszeit Datenmenge Verzögerung

Abbildung 9.4: Abbildung der IO Charakteristik

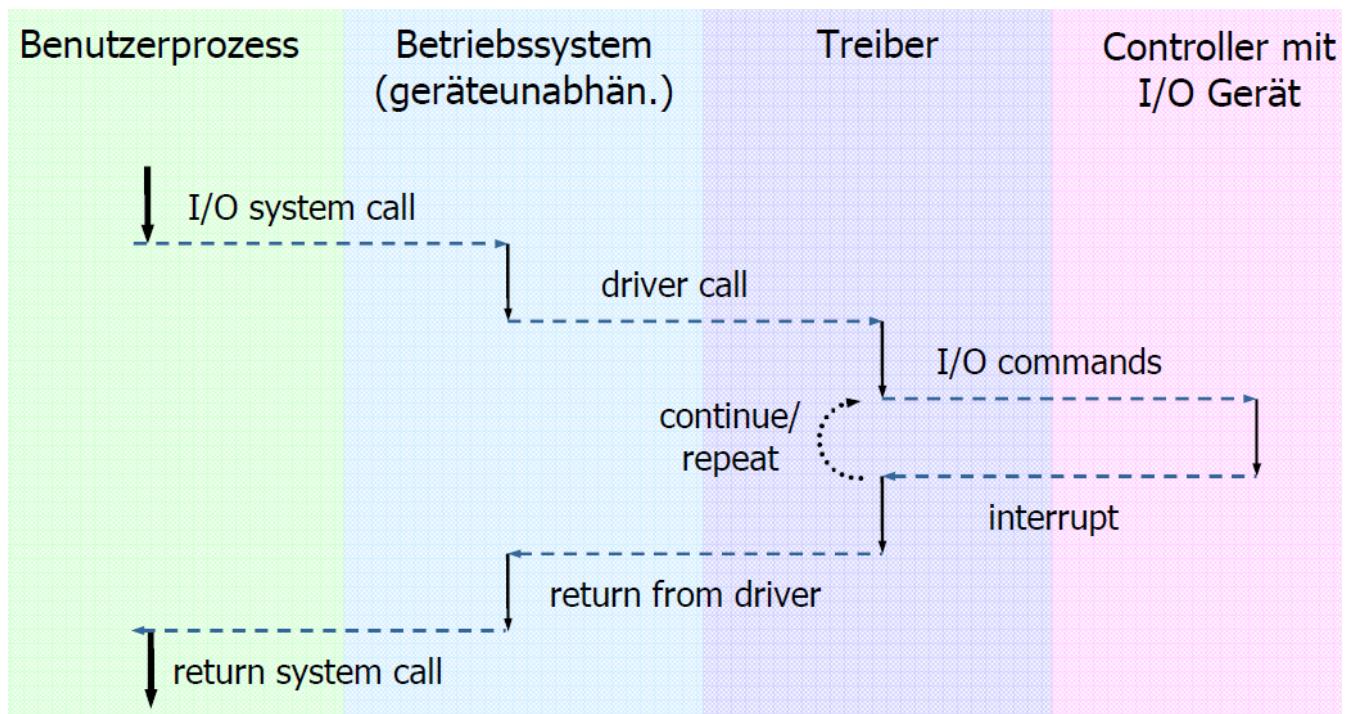


Abbildung 9.5: Abbildung der Linux Gerätezugriff

## 9.4 Sie können das Linux Modul/Treiber-System erklären

- geladenes Modul ⇒ Teil des Kernels, realisiert Kernel-Code
- Treiber ⇒ gehören zum Kernel, werden als Module realisiert

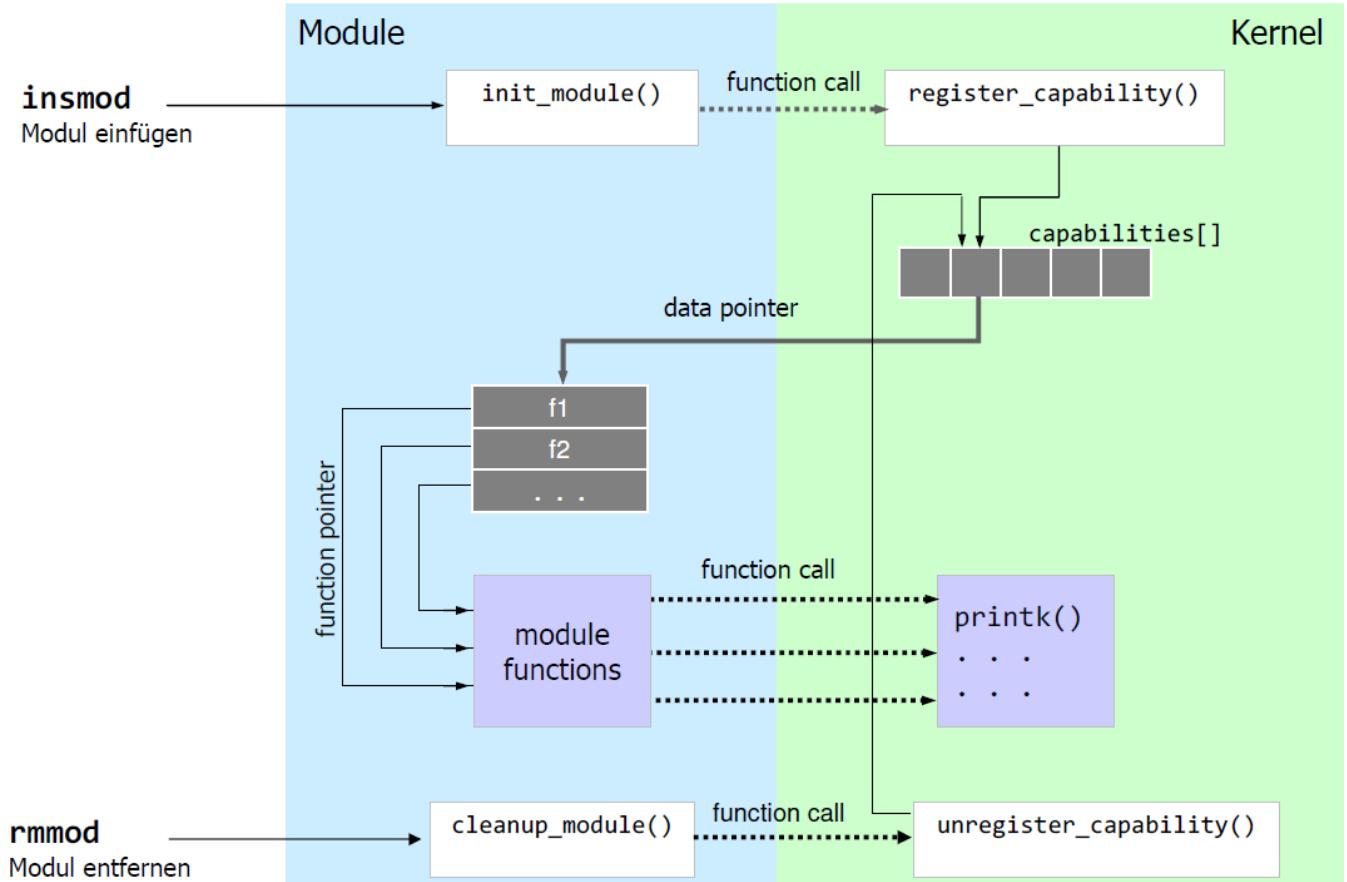


Abbildung 9.6: Abbildung des Treiber vs. Modul in Linux

## 9.5 Sie können das Windows I/O System erklären

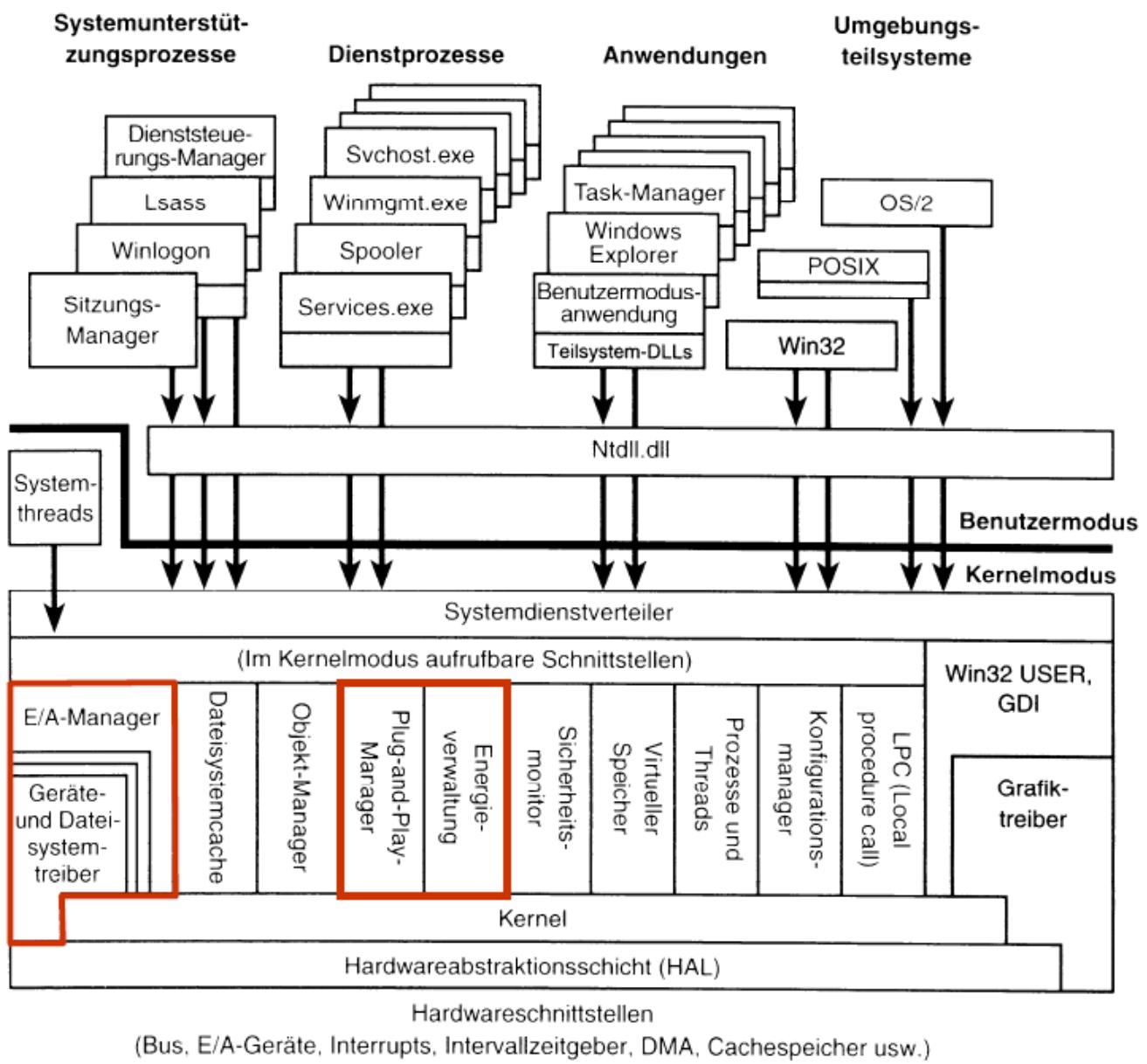


Abbildung 9.7: Abbildung des NT Architektur in Windows

# Kapitel 10

## Vorlesung 11 - I/O Geräte

### 10.1 Sie können Problemstellungen bezüglich Disks aufzeigen und diskutieren

Wie geht das Betriebssystem mit dem wichtigen Hardwarekomponente Disk um?

#### 10.1.1 Aufbau Disk

- magnetische Disk
- Sekundärspeicher
- sehr günstig
- in Platten organisiert
- Adressierung → 1D-Array von logischen Blöcken, ein Block kleinste transferierbare Einheit

#### 10.1.2 Diskzugriff

1. Anfrage durch Prozess (wartet bis Device bereit  $t_w$ ; wartet auf Kanal  $t_k$ ; Zugriff:  $t_A$ )
2. Kopf auf Track positionieren → seek time
3. warten bis Sektor unter Kopf → (rotational) latency
4. Datentransfer

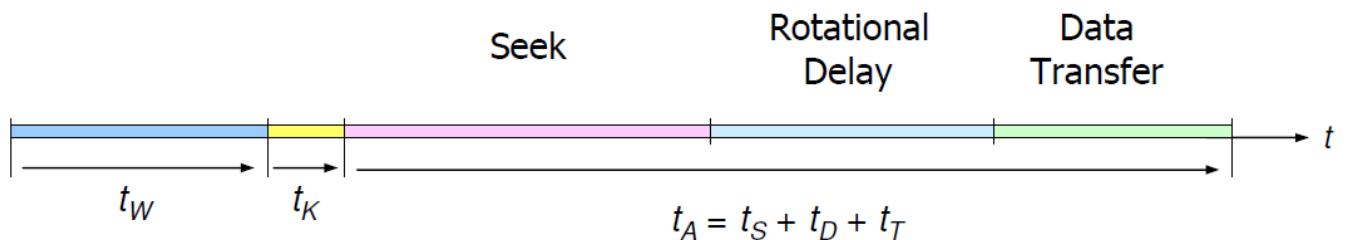


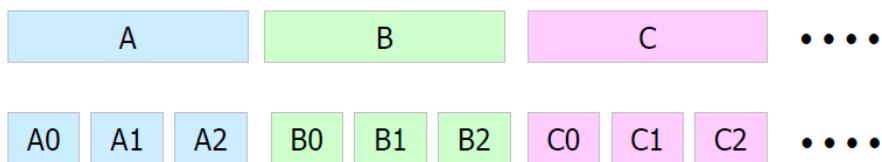
Abbildung 10.1: Abbildung des Diskzugriffs

## 10.2 Sie können Redundant Array of Independent Disks (RAIDs) erklären und diskutieren

Grundidee: mit mehreren Disks einen zuverlässigen und schnellen Massenspeicher realisieren → Ausfall einer Disk darf nicht zum Systemausfall führen

- verbessert Ausfallsicherheit des Massenspeichers
- mit Hot Spares: Reparaturzeit nahezu Null
- schützt nicht vor Viren, Löschen, Softwarefehler etc.
- Stromversorgung, Kabel etc. beeinflusst Verfügbarkeit
- muss an Anwendung angepasst sein
- Leseleistung  $\geq$  Schreibleistung

### Rechner sendet Daten in Blöcken an Disk



### Raid Controller stückelt Daten

- verteilt Stücke auf Disks
- fügt Schutzinformation bei

### Striping-Einheiten

- Bit, Byte, Wort, Block
- etc.

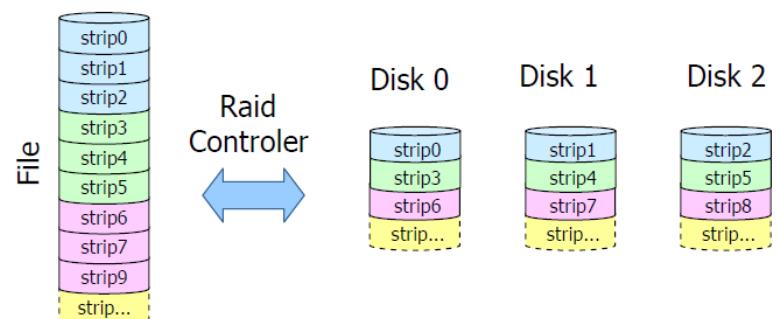


Abbildung 10.2: Abbildung des Grundprinzipes von RAID

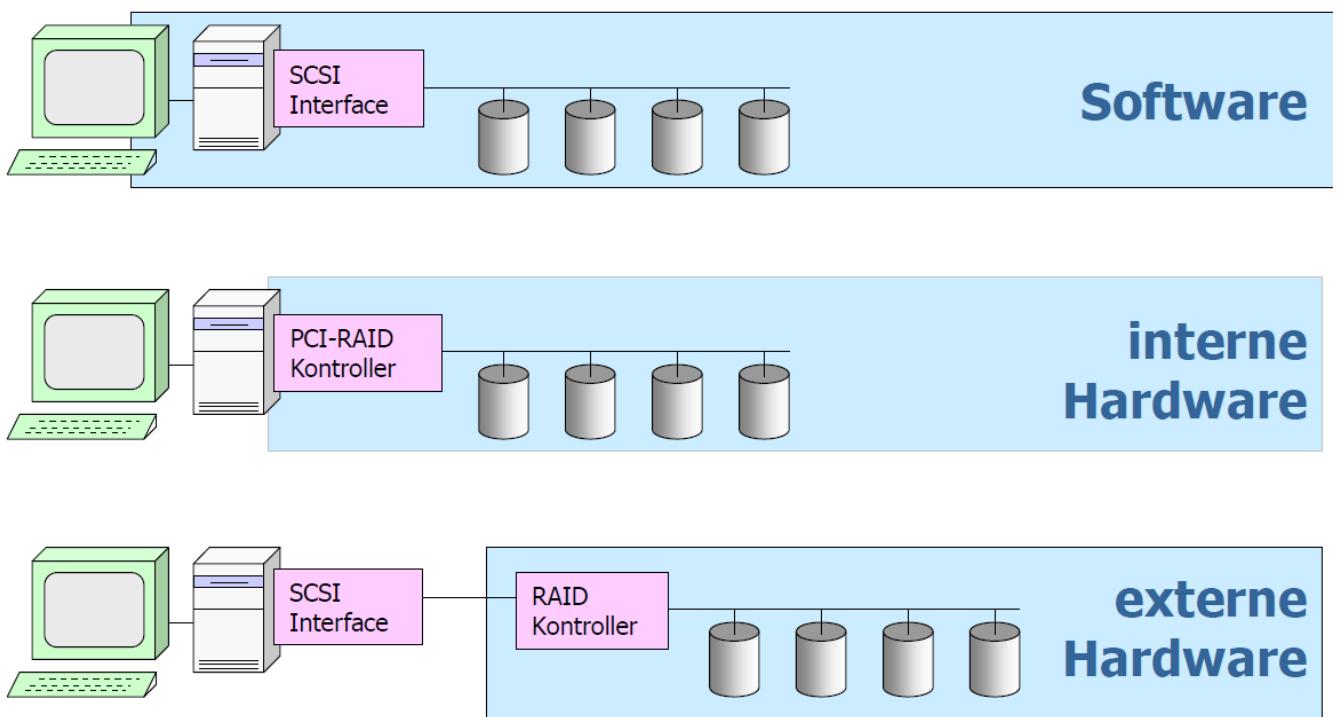


Abbildung 10.3: Abbildung Realisierungsformen von RAID

# Kapitel 11

## Vorlesung 12 - Filesystem

### 11.1 Sie können den Begriff File erklären und diskutieren

Aus Anwendersicht ist das File die kleinste logische Speichereinheit von Sekundärspeicherplatz, unabhängig vom Sekundärspeichermedium (Magnetische/Elektronische/Optische Disks, Memory Sticks, Ram Disks, Magnetische Tapes, ...). Es abstrahiert von physikalischen Eigenschaften des Mediums und ist ein Objekt mit Namen. Daten können nur in Form von Files auf Sekundärspeicher abgelegt werden, es werden Methoden für Zugriff und Verwaltung angeboten.

#### 11.1.1 Field

- einzelner Datenwert, kleinster speicherbare Einheit
- Byte
- String: z.B. name
- feste Länge
- bei variabler Länge → Feld Terminator notwenig z.B. \0

#### 11.1.2 Record

- Sammlung von zusammengehörigen Feldern
- Feste oder variable Länge, feste oder variable Anzahl Felder
- Variable Anzahl Felder → Feldname notwendig

#### 11.1.3 File

- Sammlung gleicher oder ähnlicher Records
- Files
  - ⇒ haben Namen, können erzeugt und gelöscht werden
  - ⇒ i.A. Zugriffskontrolle auf Fileebene
- Viele Sekundärspeichermedien verfügbar ⇒ Memory Sticks ⇒ Magnetische / Elektronische und optische Disks

## 11.2 Sie können Anforderungen an das File System aufzählen und diskutieren

### 11.2.1 Aufgabe: Bereitstellen von Dienstleistungen

- Persistenz ⇒ das File bleibt erhalten, trotz Crashes und An- / Abschaltzyklen
- Einfache Benutzbarkeit ⇒ Files sind einfach: zu finden, zu lesen, zu modifizieren, etc.
- Effizienz ⇒ Disk Space wird gut ausgenutzt
- Geschwindigkeit ⇒ Daten stehen schnell zur Verfügung
- Schutz ⇒ andere Benutzer haben keinen beabsichtigten oder unbeabsichtigten Zugriff auf meine Files, ... außer ich will es

### 11.2.2 Funktionen

- Files erzeugen, löschen und ändern
- über symbolische Namen auf Files zugreifen
- Zugriffsart auf Files festlegen
- kontrollierter Zugang zu Files anderer Benutzer
- Filestruktur an die Problemstellung anpassen können
- Daten zwischen Files kopieren / verschieben
- Files sichern, bei Schäden wiederherstellen

## 11.3 Sie können Operationen auf Files erklären und diskutieren

### 11.3.1 Funktionen

- erzeugen / öffnen *create()*/*open()*
- löschen / schliessen *delete()*/*close()*
- lesen / schreiben *read()*/*write()*
- positionieren (Record) *seek()*

## 11.4 Sie können den File Zugriff erklären und diskutieren

## 11.5 Sie können die File System Architektur aufzeigen und erklären

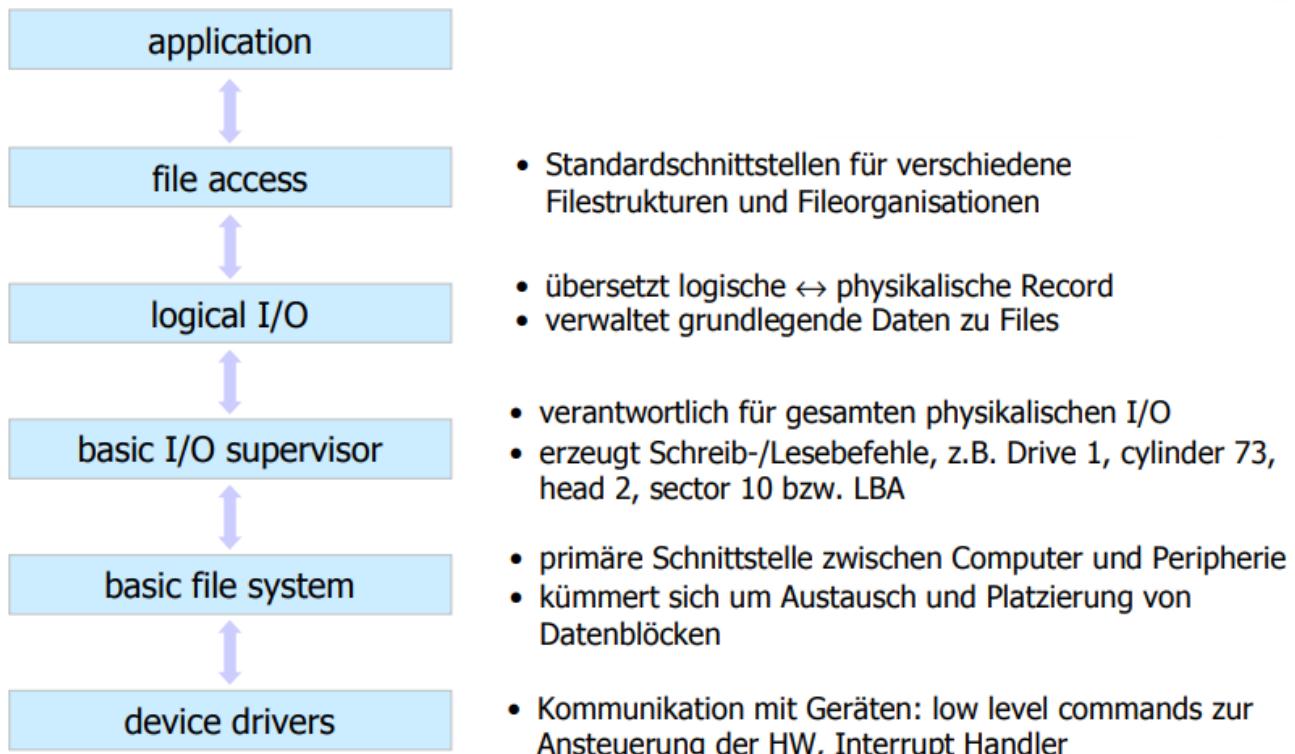


Abbildung 11.1: File System Architektur

## 11.6 Sie können File Allocation und Free Space Management erklären und diskutieren

### 11.6.1 File Allocation

Wie werden Disk Blöcke alloziert? → Einfluss auf Zugriff und Ausnutzung des Disk Platzes. Auf einer einzigen Disk können sich viele Files mit verschiedensten Größen befinden. Die Allokierung von Diskspace für diese Files muss zwei Anforderungen erfüllen: Diskspace muss effizient genutzt werden und der Filezugriff muss effizient sein. Es gibt drei wesentliche Lösungsmöglichkeiten:

#### Contiguous allocation

Blöcke werden zusammenhängend hintereinander gespeichert. Unterstützt sowohl sequential als auch random access. Für das Finden von freiem Speicherplatz werden die gleichen Lösungsstrategien wie bei Memory Management angewendet, am häufigsten werden First-Fit und Best-Fit eingesetzt (→ externe Fragmentierung). Ein weiteres Problem ist jedoch die Frage, wie viel Platz alloziert werden soll. Beim Kopieren eines Files ist die Größe bekannt, aber was, wenn das File vergrößert werden soll? Entweder muss File kopiert werden (kein Platz durch Best-Fit) oder durch die vorherige Allocation entsteht interne Fragmentierung.

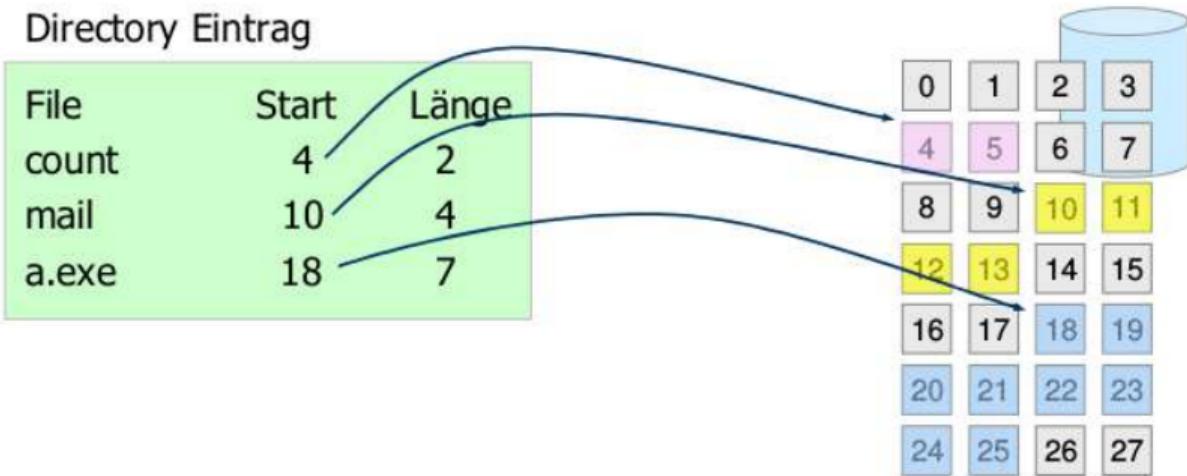


Abbildung 11.2: Contiguous Allocation

### Linked Allocation

Die Blöcke eines Files bilden eine verkettete Liste, was alle Probleme von Contiguous Allocation löst. Jedoch hat diese Methode eigene Probleme: Sie ist ineffizient für Random Access (Lösungsstrategien: First-Fit, Best-Fit), ausserdem brauchen die Zeiger Disk Space (4 Byte pro Pointer, 512 Byte Blöcke  $\rightarrow$  0.78 % des Disk Space). Lösung: Clustering, Cluster von Blöcken anfordern, damit Pointer im Verhältnis weniger Platz brauchen). Ein weiteres Problem ist die Zuverlässigkeit: Was, wenn ein Zeiger verloren geht? Teilweiser Lösungsansatz ist double linked list, was jedoch auch mehr Overhead bedeutet.

FAT16 und FAT32 verwenden Varianten von Linked Allocation.

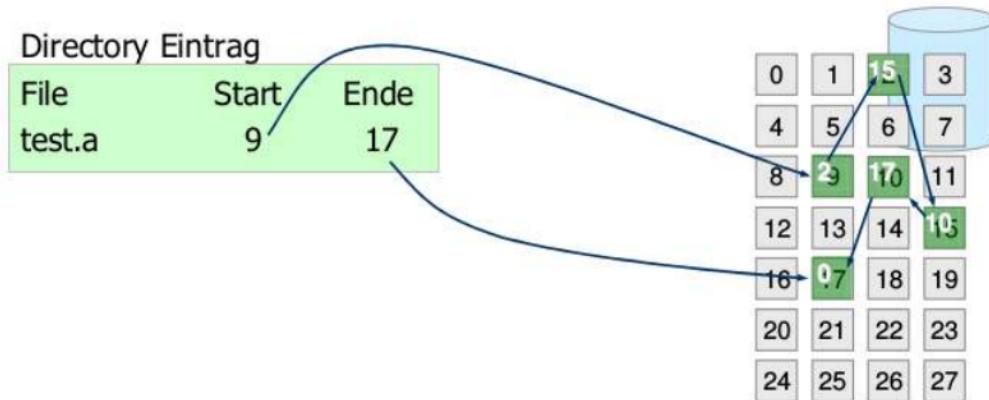


Abbildung 11.3: Linked Allocation

### Indexed Allocation

Bei der Indexed Allocation werden die Blockzeiger in einem Index Block gespeichert. Vorteile dieser Methode sind keine externe Fragmentation und effizienter Random Access. Ein Problem ist natürlich, dass die Zeigerblöcke Speicherplatz brauchen.

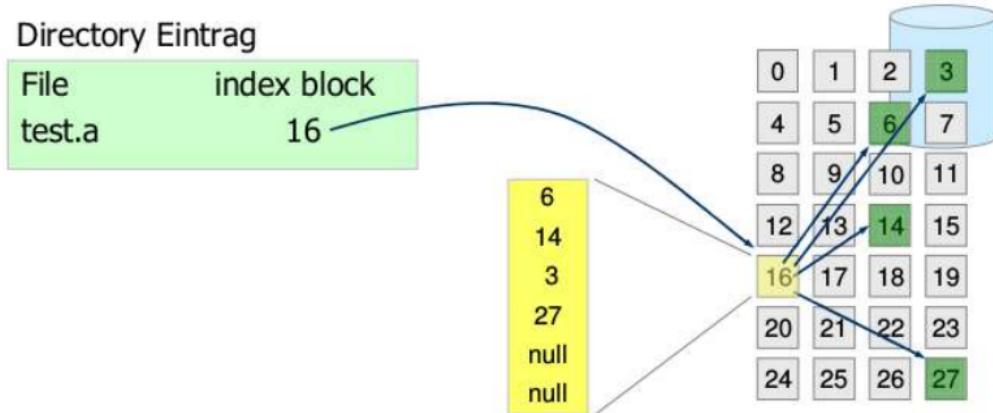


Abbildung 11.4: Indexed Allocation

### 11.6.2 Free Space Management

Freie Blöcke auf der Disk müssen verwaltet werden, deswegen führt das BS eine Liste mit freien Blöcken (Free Space List genannt, jedoch nicht zwingend eine Liste).

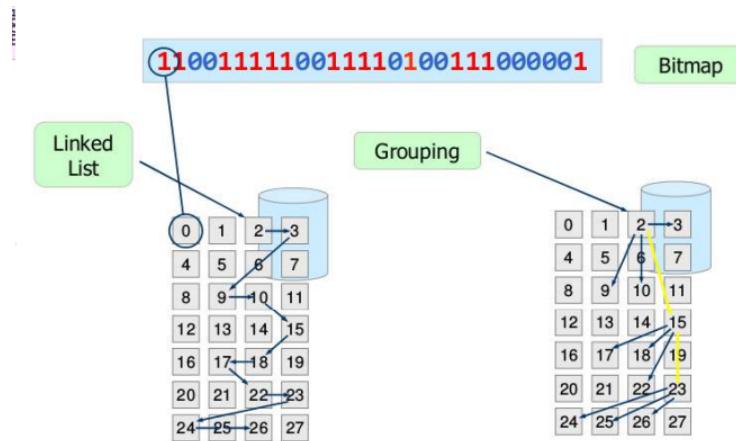


Abbildung 11.5: Free Space Management

#### Bit Vector / Bit Map

Jeder Block entspricht einem Bit in der Liste. Effizient wenn im Speicher, aber Speicheraufwendig (bei 6 GB Disk mit 512 Byte Blöcken 1.5 MB).

#### Linked List

Freie Blöcke als Linked List, Zeiger auf ersten Block im Speicher. Absuchen der Free List ist aufwendig, jedoch nicht häufig notwendig.

#### Grouping

Im ersten freien Block werden die Adressen von N freien Blöcken gespeichert. Der letzte Block der Adresse zeigt auf den nächsten Block mit freien Blöcken.

## 11.7 Sie können File Management erklären und diskutieren

### 11.7.1 Record Locking

Da in Multiuser-Umgebungen Mehrfachzugriffe möglich sind, ist Mutex auf Files notwendig (z.B. bei Datenbanksystemen, Daemonen, ...). Das sperren einer ganzen Datei ist ineffizient, deswegen werden durch Record-Locking nur Teile des Files gesperrt. Dafür ist jedoch BetriebssystemUnterstützung notwendig, bei Linux/Unix ist das fcntl().

### 11.7.2 Journaling

Jorunaling verhindert Inkonsistenzen des File Systems bei Crashes, sämtliche Änderungen am File System geschieht über Transaktionen. Moderne Betriebssysteme unterstützen Journaling (Linux: ext3, ext4, reiserfs – Windows: NTFS). Aktuelle Jorunaling Filesysteme verwenden Journaling meist nur für Metadaten.

Ext3 verfügt über drei Journaling-Modi:

- Ordered: Nur Metadaten (Info zu Files, nicht zu Inhalt) werden im Journal gespeichert.
- Writeback: Nur Metadaten, ordered, aber nicht synchron.
- Jorunal: Auch Daten werden im Journal eingetragen. Wegen zweifachem Speichern deutlich langsamer.

#### Wie funktionierts?

Beispiel File löschen unter Unix: Geschieht in drei Schritten:

1. File im Verzeichnis löschen
2. I-Node löschen, in den Pool zurückgeben
3. Alle Disk Blöcke freigeben, in den Pool zurückgeben

Vorgehen:

- Log auf Disk schreiben und verifizieren
- Lösch-Operationen ausführen
- Wenn OK → Log löschen
- Wenn Crash → Nach Reboot Log überprüfen, falls notwendig Schritte wiederholen

### 11.7.3 Memory Mapped Files

Memory Mapped Files werden in den Virtual Memory des Prozesses abgebildet und mit demand paging in den physischen Speicher gelesen: Filezugriff = Speicherzugriff. Bei Memory Mapped Files und File Sharing ist die Konsistenzsemantik zu beachten (Wann sind Daten ins File geschrieben und zum Lesen verfügbar?) → Ev. Synchronisation durch Anwender nötig

### 11.7.4 Quotas

Legen Grenzen für Disk-Usage (Partitions) fest, im Allgemeinen pro Anwender.

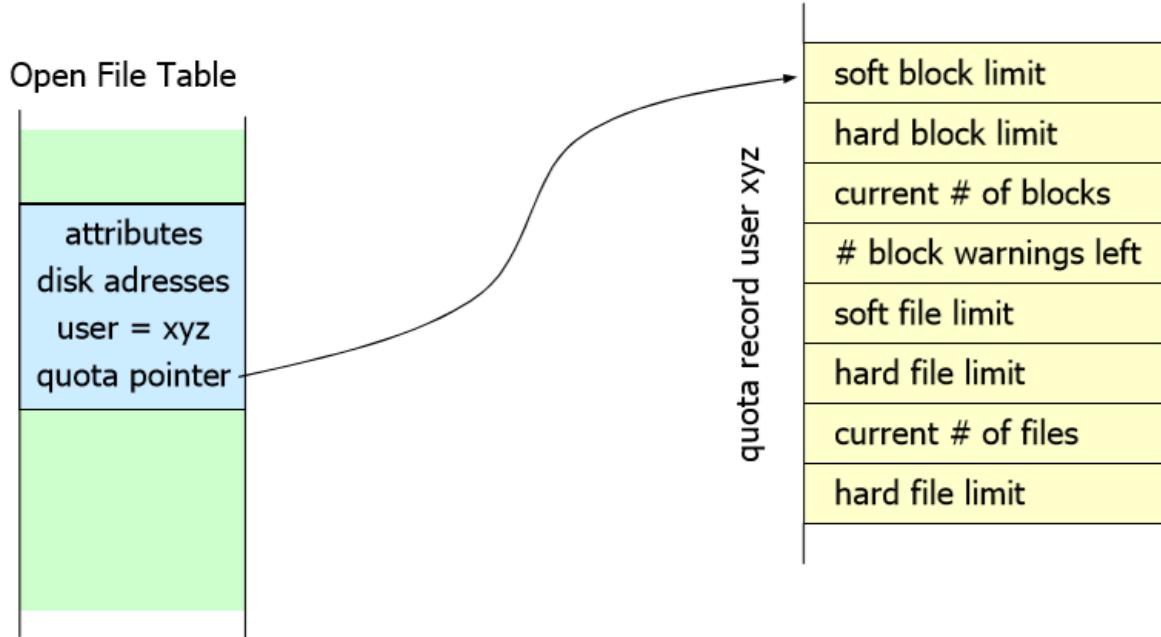


Abbildung 11.6: File Management Quotas

- 11.8 Sie können das Unix File System erklären und diskutieren, Filegrößen bestimmen

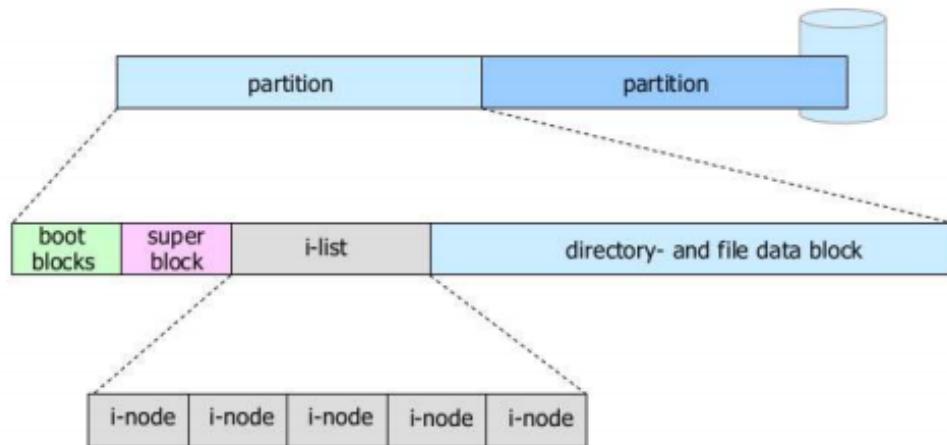


Abbildung 11.7: Übersicht

### 11.8.1 Bootblock

Bootstrap-Code zum starten des BS, meistens erster Sektor.

### 11.8.2 Superblock

Beschreibt Aufbau des Filesystems. Grösse, Anzahl Dateien, freier Platz, Inodeinformation, Blockinformation, ...

### 11.8.3 Inode List

Inodes enthalten Informationen zu Files (z.B. Diskblöcke, Metadaten wie Datum, Zugriffsrechte, Open Count). Grösse beim Konfigurieren des Filesystems festgelegt.

Für kleine Files enthalten I-Nodes direkte Zeiger auf Datenblöcke. Für Grosse Files Zeiger auf Blöcke die weitere Zeiger enthalten (oder auch direkt auf Datenblöcke).

Die Maximale Filegrösse ergibt sich folgendermassen:

Blockgrösse 1 Kbyte und 32 Byte Zeiger (Standard Unix).

10 (Linux: 12) direkte Blöcke à 1 Kbyte:  $\Rightarrow 10 \text{ KB}$

1 indirekter Block mit 256 direkten Blöcken:  $\Rightarrow 256 \text{ KB}$

1 doppelt indirekter Block mit 256 indirekten Blöcken:  $\Rightarrow 64 \text{ MB}$

1 dreifach indirekter Block mit 256 doppelt indirekten Blöcken:  $\Rightarrow 16 \text{ GB}$

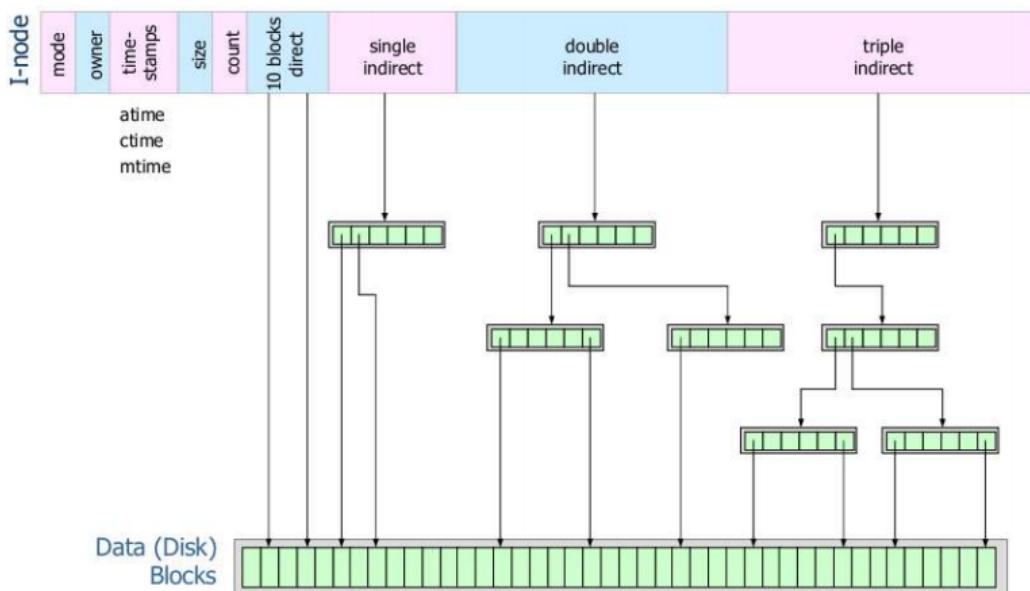


Abbildung 11.8: Inodes

### 11.8.4 Datenblöcke

Beginnen nach der Inode Liste und enthalten echte Daten bei Datenfiles, Verwaltungsinformation für Directories. Ein Datenblock kann nur zu einem einzigen File gehören.

Directory Blöcke gehören zu Directory Files und enthalten die I-Node Nummer, den Filenamen, Länge des Eintrags und des Filenamens. Directoryeinträge haben eine variable Länge. I-Nodes enthalten neben den direkten Zeiger auf Datenblöcke auch Zeiger auf Blöcke, die weitere Zeiger enthalten.

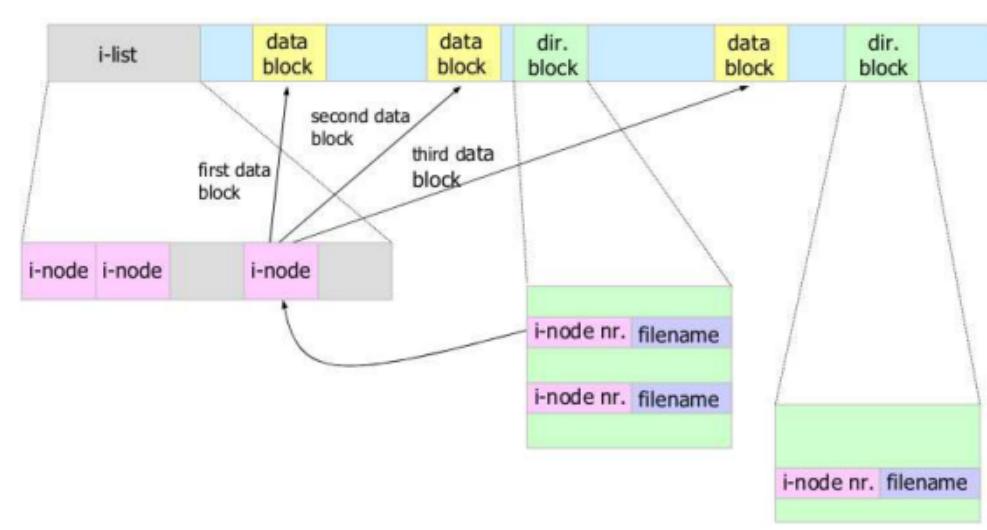


Abbildung 11.9: Datenblöcke

## 11.9 Sie können das Windows File System erklären und diskutieren

### 11.9.1 File Allocation Table (FAT) File System

- File System von DOS, Win95b, Win98, (WinXP)
- verkettete Liste
- File: Folge von Clustern
- am Anfang des Disks gespeichert
- meist mehrfach (Redundanz)

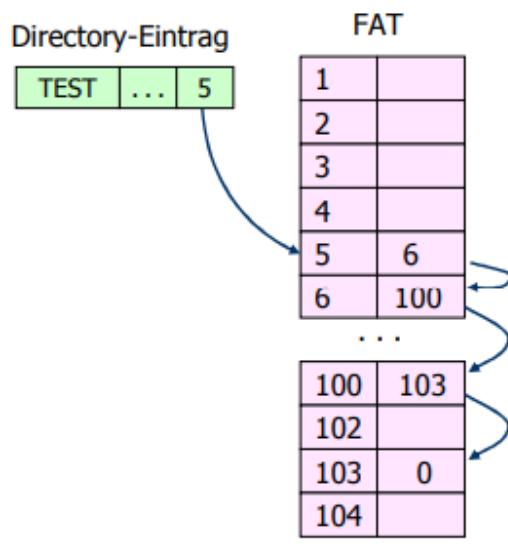


Abbildung 11.10: File Allocation Table Eintrag

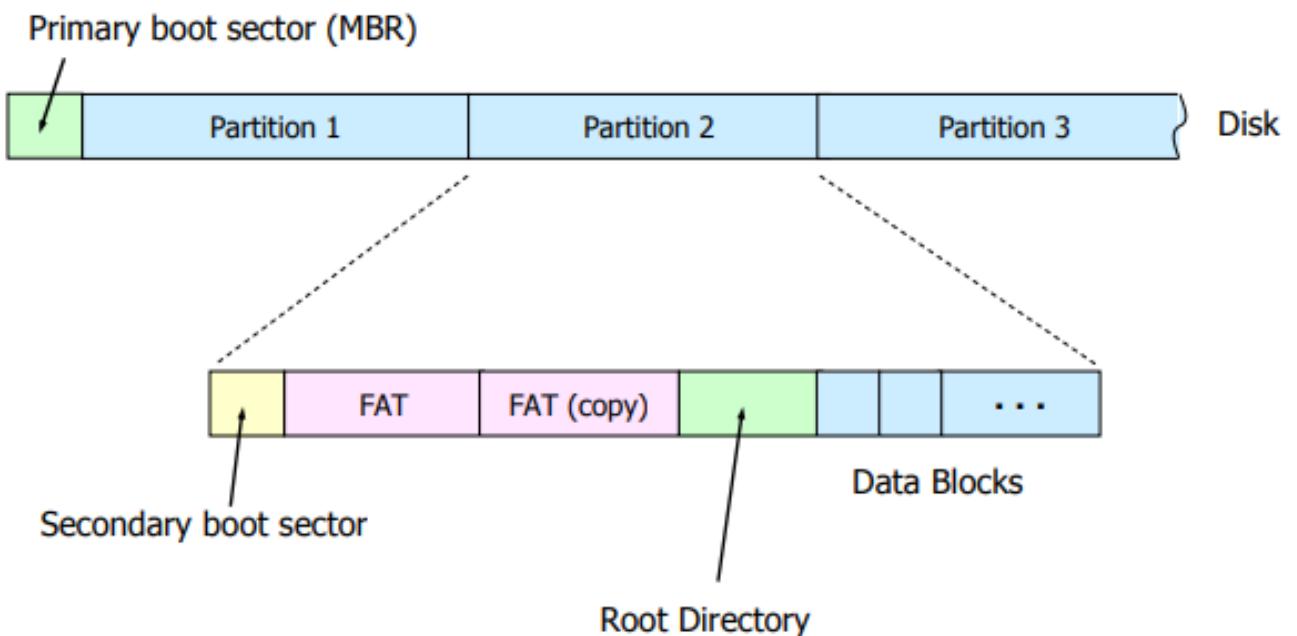


Abbildung 11.11: File Allocation Table

### 11.9.2 NT File System (NTFS)

- Wiederherstellbares File System (Journaled File System)
- Daten in Clustern gespeichert
- unterstützt mehrere Datenströme (wie MAC data/resource fork)
- Cluster-Remapping  
⇒ defekte Sektoren im Betrieb erkannt und markiert
- sichert Bootsektor
- Dateien in Cluster gespeichert  
⇒ mehrere Sektoren = 1 Cluster  
⇒ Sektor 512 Bytes ... Trend 4KB  
⇒ Clustergrößen: 512B, 1KB, 2KB, 4KB

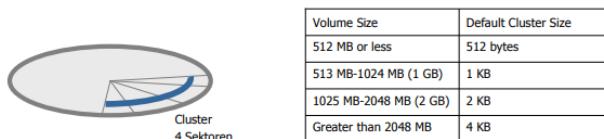


Abbildung 11.12: NTFS Cluster Size

### MFT-Record

- Sammlung von Attribut/Wert Paaren  
⇒ Standard Information  
⇒ Filename

⇒ Data  
⇒ ...

- Filename bis 255 Zeichen
- Daten: kleine Files  
⇒ Große Files: indexed allocation von Clustern

## Directories

- Directories sind Files
- Sammlung von Attribut/Wert Paaren  
⇒ Standard Information  
⇒ Directory Name (Filename)  
⇒ Index root
  - Zeiger auf Files in Directory
  - MFT-Index
  - Filename / Länge des Namens
  - etc.

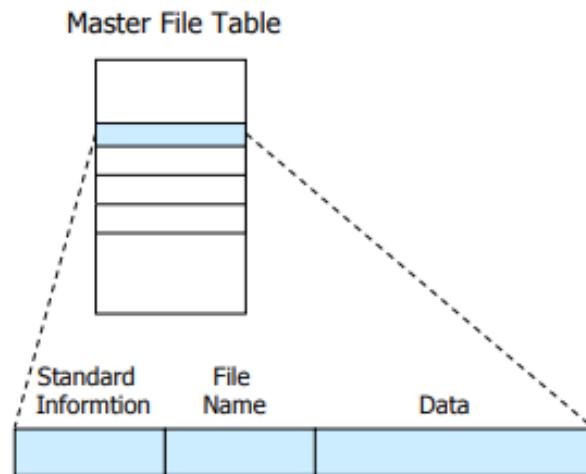


Abbildung 11.13: MFT Record

## Security

- Jedes File hat eine ACL (access control list)  
⇒ detaillierte, differenzierte Berechtigungen  
⇒ vererbbarer Berechtigungen  
⇒ Entzug hat Vorrang
- Kopieren einer Datei  
⇒ Berechtigungen werden vom Ziel-Directory übernommen
- Verschieben einer Datei ⇒ Berechtigungen werden beibehalten