

# 1 Computersysteme

**Computersystem** besteht aus Hardware, System- und Anwendungssoftware.

**Systemsoftware:** Betriebssystem und systemnahe Software wie Compiler, Interpreter, Editoren und Anwendungssoftware wie Bankanwendungen, Browser, usw.

Anwendungssoftware (Browser, Bankanwendung, Buchhaltung,...)	
Systemsoftware	Systemnahe Software (Datenbanken, Compiler) Betriebssystem
Firmware und Hardware	Maschinsprache Mikroarchitektur Physikalische Gerte

**Aufgabe von Betriebssystemen** Anwender soll von Details der Rechnerhardware entlastet werden, Kapselung des Zugriffs auf die Betriebsmittel, Bereitstellung einer *virtuellen Maschine* ber der hardware

**Betriebsmittel** Unterscheidung: *entziehbare, nicht entziehbare*, sowie *exklusiv* oder *shared*

reale Betriebsmittel	Prozessoren, Speicher, Dateien, Gerte
virtuelle Betriebsmittel	virtueller Speicher, Prozessoren, Koprozessoren

**Betriebssystemkategorien** Smart Card, Embedded, Server, Desktop, Echtzeit

**Von-Neumann-Architektur** Leitwerk (Control Unit) holt Maschinenbefehle in den Speicher und fhrt sie aus. Rechenwerk (Processing Unit) fhrt logische und arithmetische Operationen aus. Maschinenbefehle und Daten liegen im selben Speicher. Ein-/Ausgabe (Input/Output) ist die Verbindung externer Gerte. Leitwerk + Rechenwerk = CPU

**Harvard-Architektur** Maschinenbefehle und Daten liegen in getrennten Speichern. Sie werden ber einen getrennten Bus mit der CPU verbunden. Effizient, da doppelt so viele Leitungen zur Verfgung stehen

**Caches** Schnelle Pufferspeicher in unterschiedlichen Speicherebenen. Speichert: Daten bzw. Code Teile → Zugriffsoptimierung. Level-n-Caches: Je kleiner n ist, desto schneller der Cache

**CPU-Register** CPU enthlt einen Registersatz um Daten schnell zu laden, speichern und zu rechnen

**Universalbetriebssysteme** Jede mglichen Anwendungen sind auf dem Betriebssystem ausfhrbar. Verfgt jedoch nicht ber Realtime-Eigenschaften

**Multicore-Prozessoren** Mikroprozessoren mit mehreren vollstndigen CPUs. Viele Ressourcen sind repliziert

**Hyperthreading-CPU**s mehrfhige Singlecore-Prozessoren mit mehreren Programmzhlern und Registerstzen sowie Interrupt-Controllern, die sich gegenber dem Betriebssystem aber als Multicore-Prozessoren darstellen

**Virtuelle Maschine** Konzept zur Abstraktion, zum Beispiel Dateien und Ordner anstatt Bits

# 2 Betriebssystemarchitekturen und Betriebsarten

## 2.1 Zugriffsschutz in Betriebssystemen

Betriebssystem von Anwendungen abgeschottet, nicht zulssig: Zugriff auf Hardware und Ressourcen durch Anwendungen und Beeintrchtigung einer Anwendung durch eine andere. Dedizierte, abgesicherte Schnittstellen ntig, wenn Anwendung Dienste des Betriebssystems bentigt

**Benutzermodus** (Usermodus) kein Zugriff auf kernelspezifische Code- und Datenbereiche mglich

**Kernelmodus** (privilegierter Modus) Programmteile des Betriebssystems werden ausgefhrt, die einem gewissen Schutz unterliegen

**Realisierung der Modi** Darstellung/Einstellung der Modi ber ein Steuer- und Kontrollregister des Prozessors. Maschinenbefehl, der einen kontrollierten bergang vom Benutzermodus in den privilegierten Modus ermglicht, damit ein Anwendungsprogramm eine Betriebssystemfunktion aufrufen kann

## 2.2 Betriebssystemarchitekturen

### 2.2.1 Klassische Architekturen

**Monolithischer Betriebssystemkern** alle Module werden im privilegierten Modus ausgefhrt. Zugriff auf den kompletten Kerneladressraum. Dienstverteiler, der die Systemaufrufe der Anwendungen entgegennimmt und an die einzelnen Kernelmodule weiterleitet

**Schichtorientierter Kernel** Kernel ist flexibler und berschaubarer. Hhere Schichten nutzen die Module der darunter liegenden Schicht. Die unterste Schicht dient meist dem Zugriff auf die Hardware. Abhngigkeit von der Hardware ist in dieser Schicht gekapselt

## 2.3 Klassische Grorechnerbetriebsarten

### 2.3.1 Multiprogramming, Multiprocessing und Multitasking

**Multiprogramming** gleichzeitige Ausfhrung von Programmen in einem Betriebssystem - Mehrprogrammbetrieb erfordert nicht unbedingt Mehrprozessorsysteme - heute ist die Anzahl der nebenlufigen Prozesse hher als die Anzahl der CPUs

**Einprogrammbetrieb** nur ein Prozess kann in den Speicher geladen und ausgefhrt werden

**Einprozessorsysteme** verwalten genau einen Prozessor

**Singletasking** nur ein Prozess ist aktiv, der smtliche Betriebsmittel des Systems nutzen kann

**Multitasking** mehrere Prozesse knnen nebenlufig ausgefhrt werden - Betriebsmittel werden nach verschiedenen Strategien (Prioritten, Zeitscheibeverfahren) zugeteilt - *Timesharing*: Zuordnung des Prozessors nach Zeitintervallen an die nebenlufigen Prozesse

### 2.3.2 Batchverarbeitung und interaktive Verarbeitung

Auftrge an das Betriebssystem werden zuerst in eine Warteschlange des Betriebssystems eingetragen und dann unter Bercksichtigung von Prioritten oder der Reihe nach abgearbeitet (Stapelbetrieb)

### 2.3.3 Teilnehmerbetrieb

Nutzung von Online-Systemen, die von Benutzern ber eine Dialogschnittstelle verwendet werden knnen - jeder Benutzer erhlt seinen eigenen Benutzerprozess sowie weitere Betriebsmittel, nach Anmeldung ber einen Login-Dialog - fr jeden Benutzer wird der Prozess einmal geffnet

### 2.3.4 Teilhaberbetrieb

Prozesse und Betriebsmittel werden ber einen *Transaktionsmonitor* zugeteilt - ideal fr dialogorientierte Programme mit vielen parallel arbeitenden Anwendern, die meistens kurze und schnelle Transaktionen ausfhren - Beispiel: Buchungssystem fr Flugbuchungen

**Transaktion** atomar auszufhrender Service, der entweder ganz oder gar nicht ausgefhrt wird. Sind mehrere Operationen, zum Beispiel auf einer Datenbank erforderlich, so darf dies nur in einem Stck erfolgen - Transaktionskonzept wichtig fr betriebliche Informationssysteme - untersttzt Entwicklung fehlertoleranter Anwendungssysteme

# 2.4 Terminalserver-Betrieb

**Terminalserver** verwaltet Terminals bzw. Client-Arbeitspltze

**Terminalserver-Betrieb** Vereinfachung der Administration verteilter Komponenten und bessere Kontrolle

**Serverlandschaft** bedient 'dumme' Client-Rechner (Thin Clients) - Anwendungsprogramme laufen vollstndig auf den Servern - Clientrechner dienen nur der Prsentation - Dienste: Verwaltung von Lizenzen, zentrale Benutzerverwaltung, Sicherheitsdienste

**Termindienst** Zentralisierung von Betriebsmitteln, um die beteiligten Systeme leichter bedienen zu knnen

**leistungsfhige 'Terminalserverfarm'** stellt Ressourcen wie Rechenleistung, Hauptspeicher, Plattenspeicher usw. bereit

## 2.5 Verteilte Verarbeitung

### 2.5.1 Echt verteilte Betriebssysteme

Transparente Verteilung eines Betriebssystems in einem Netzwerk, in dem die Betriebssystemdienste auf mehreren Rechnersystemen liegen. Anwendung weit nicht, wo sie genau welche Ressource benutzt - Kernel ist auf die Rechnersysteme verteilt - die verteilten Komponenten kooperieren fr die Anwendungsprozesse transparent ber ein Netzwerk

### 2.5.2 Client-/Server-Systeme

dedizierte softwaretechnische Rollen: *Client* und *Server* - Server stellt Client Dienste (Services) zur Verfgung - Server- und Clientkomponenten sind ber ein Netzwerk verteilt und kommunizieren ber einen Request-/Responsemechanismus - Komponenten laufen auf Rechnersystemen ab, die alle unabhngige Betriebssystemkerne enthalten

### 2.5.3 Peer-to-Peer-Systeme

keine dedizierten Rollen → gleichberechtigte Partner - Kommunikationspartner = Peers - Peer kann die Rolle eines Clients, Servers oder in einer Doppelrolle agieren - jeder Peer verfgt ber eigenen Betriebssystemkernel (unterschiedliche Betriebssysteme) - hybrid: mit zentralen Komponenten - Superpeers: spezielle Aufgaben → Verzeichnisdienste fr die Verwaltung der Peers - P2P bietet bessere Skalierungsmglichkeiten und hhere Verfgbarkeit

### 2.5.4 Kommunikations-Middleware

in der Regel im Benutzermodus betrieben - stellt dem Anwendungsprogramm Dienste zur Kommunikation mit den anderen Bausteinen zur Verfgung - jeder Rechnerknoten verfgt ber ein komplettes eigenes Betriebssystem (unterschiedliche Betriebssysteme) - Middleware in den kommunizierenden Anwendungsprozessen implementiert

## 2.6 Virtualisierung von Betriebs- und Laufzeitsystemen

Zweck: gemeinsame Hardware mehrfach zu nutzen - Beispiel: Server - auf der Basismaschine knnen konkrete Betriebssysteme wie Windows oder Linux aufsetzen - Basismaschine simuliert die Hardware und stellt den Gastbetriebssystemen eine Ablaufumgebung bereit - die Betriebssysteme laufen vllig isoliert voneinander

## 2.7 Cloud Computing

'Rechenwolke', die Dienste anbietet - eine IT-Infrastruktur wird ber ein Netzwerk zur Verfgung gestellt - Infrastruktur kann sein: Rechenkapazitt, ganze Betriebssysteme, Datenspeicher, Anwendungssoftware - man unterscheidet: private, ffentliche oder hybride Cloud-Infrastrukturen Varianten: *Infrastructure as a Service (IaaS)*: Zugang an virtualisierten Rechnern einschlielich ganzer Betriebssysteme und Speichersysteme *Platform as a Service (PaaS)*: Zugang zu ganzen Programmierungsumgebungen *Software as a Service (SaaS)*: Zugang zu Anwendungsprogrammen

3 Interruptverarbeitung

Wenn ein Gerat Signale oder Daten an die CPU ubertragen mochte, wird eine Unterbrechungsanforderung erzeugt - sie wird der richtigen Bearbeitungsroutine im Betriebssystem ubergeben - aktuell ablaufende Aktivitaten mussen unterbrochen werden - nach der Abarbeitung muss der alte Zustand wieder hergestellt werden. Ahnliches geschieht bei der Bearbeitung von Systemdiensten. Systemdienste werden uber sog. *Systemcalls* durch Programme aktiv initiiert. Unterbrechungsanforderungen (Interrupt-Anforderungen oder Interrupt-Request) und der zugehorigen Unterbrechungsbearbeitung (Interrupt-Bearbeitung)

3.1 Interrupts

3.1.1 Uberblick

**Polling** Nachfrage vom Prozessor, um deren Kommunikationsbereitschaft festzustellen bzw. um anliegende Ereignisse oder Kommunikationswunsche der Ereignisquelle abzufragen. **Nachteil:** CPU muss immer arbeiten → Effizienz beeintrachtigt

**Interrupt** Unterbrechung, die die CPU veranlasst, einen vordefinierten Code auszufuhren. Der Code liegt auerhalb der CPU - Ereignisquellen melden sich - konnen durch Hardware oder Software verursacht werden

**System Calls** auch: *Traps* oder *Faults*, siehe unten

**Synchrone Interrupts** auch: *Exceptions*, sie treten bei synchronen Ereignissen auf - treten immer, bei identischen Randbedingungen, bei der gleichen Programmstelle auf - vorhersehbar und wiederholbar - Prozessor alleine kann die Interrupts nicht losen und muss an das Anwendungsprogramm oder Betriebssystem melden

**Exceptions** von CPU fur das laufende Programm ausgelost. Beispiel: divide-by-zero-exception

**Traps** erst nach der Ausfuhrung vom Betriebssystem erkannt und an das Programm gemeldet

**Faults** vor der eigentlichen Ausfuhrung abgefangen und gemeldet. Beispiel: nicht zugewiesener Speicher wird adressiert

**Asynchrone Interrupts** nicht an laufendes Programm gebunden. Beispiel: Nachricht an den Netzwerkadapter - nicht vorhersehbar und nicht reproduzierbar

3.1.2 Interrupt-Bearbeitung

Interrupt-Bearbeitung wird die Steuerung an eine definierte Position im Kernel ubergeben, dabei Wechsel in den Kernelmodus, sofern noch nicht geschehen

**Maskierung** Ein- und Ausschalten von Interrupts bestimmter Gerate, kann uber ein *Interrupt-Maskenregister, IMR* erfolgen - Nutzung: Verhindern das ein Interrupt von einem anderen Interrupt unterbrochen wird

**Non Maskable Interrupt (NMI)** Steuerung fur maskierbare Interrupts. Wenn NMI eintritt entsteht eine Ausnahmesituation und somit eine Ausnahmebehandlung

**Interrupt-Service-Routine (ISR)** Bearbeitung von Interrupts. Fur jeden Interrupt-Typ gibt es einen passenden ISR

**Interrupt-Request-Bearbeitung (IRQ-Bearbeitung)** Hardwarebedingte Interrupts, die nicht direkt vom auslosenden Gerat an die CPU signalisiert werden, werden zuerst an einen *Interrupt-Controller* gemeldet. Der Interrupt-Controller erzeugt dann eine Unterbrechung der CPU, mit Hilfe passender ISR

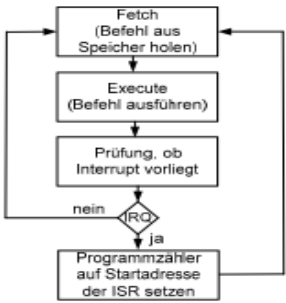
**Interrupt-Vektor-Tabelle** ISRs werden uber *Interrupt-Vektoren* adressiert, welche sich in der Interrupt-Vektor-Tabelle befinden

**Interrupt-Level** Prioritaten-Level, welcher dem Betriebssystem vorgibt, wie der Interrupt in die Gesamtverarbeitung eingebaut werden soll

**Erkennung einer Unterbrechungsanforderung** nach Ausfuhrung eines Maschinenbefehls wird uberpruft, ob ein Interrupt-Request anliegt - liegt einer vor, wird in spezielles Unterprogramm verzweigt (ISR oder Verteilungsroutine)

Prufung, ob Interrupt vorliegt

- 1. Interrupt unterbricht das aktuell laufende Programm
- 2. aktueller Prozessorstatus das laufenden Programms wird am Anfang der Interrupt-Bearbeitung gerettet
- 3. Interrupt-Bearbeitungsroutine wird in der Interrupt-Vektor-Tabelle gesucht und ausgefuhrt
- 4. Ende der Bearbeitung: ISR sendet dem Interrupt-Controller Bestatigung
- 5. alter Prozessstatus wird hergestellt und es wird an der abgebrochenen Stelle weitergearbeitet



3.2 Systemdienste und Systemcalls

**Dienste des Betriebssystems** Anwendungen nutzen die Dienste des Betriebssystems, die uber sogenannte Systemcalls aufgerufen werden - Wohldefinierte Einstiegspunkte ins Betriebssystem - Systemcalls werden im Kernelmodus ausgefuhrt - beim Aufruf wird durch den Prozessor vom User-modus in den Kernelmodus umgeschaltet

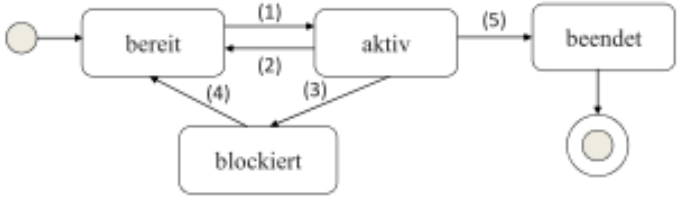
4 Prozesse und Threads

4.1 Prozesse und Lebenszyklus von Prozessoren

**Prozess** Ausfuhrung eines Programms auf einem Prozessor, dynamische Folge von Aktionen verbunden mit entsprechenden Zustandsanderungen, inklusive Daten und Registerinhalten und eigenem Adressraum

**Virtueller Prozessor** jedem Prozess im Multiprogramming wird ein virtueller Prozessor zugeordnet

**Konkurrenz zwischen Prozessen** Prozesse konkurrieren um die Betriebsmittel, laufen abwechselnd einige Millisekunden, werden mit Mitteln des Betriebssystems erzeugt



- (1) BS wahlt den Prozess aus (Aktivieren)
- (2) BS wahlt einen anderen Prozess aus (Deaktivieren, Vorrangunterbrechung)
- (3) Prozess wird blockiert (z.B. Warten auf Input, Betriebsmittel angefordert)
- (4) Blockierungsgrund aufgehoben (Betriebsmittel verfugbar)
- (5) Prozessbeendigung oder Fataler Fehler (Terminieren des Prozesses)

**Prozesstabelle und PCB** Eintrag in der Prozesstabelle wird als *Process Control Block (PCB)* bezeichnet - wichtige Informationen im PCB: Programmzahler, Prozesszustand, Prioritat, Prozess ID, etc.

4.1.1 C Fork()

Ein fork() erzeugt einen Kindprozess, der in der nachsten Zeile des Programms beginnt.

```
pid_d pid;
pid = fork();
if(pid == 0) {
    // Kindprozess
    printf("Kind-PID: %u", getpid());
    printf("Eltern-PID: %u", getppid());
    exit(0); // Prozess Beenden
} else {
    // Elternprozess
}
```

```
if(fork() == 0) {
    fork();
}
fork();
```

4.2 Threads

sind leichtgewichtige Prozesse die eine *nebenlaufige Ausfuhrungseinheit* innerhalb eines Prozesses darstellen. Alle Threads eines Prozesses teilen sich den gemeinsamen Adressraum dieses Prozesses. Threads haben eigenen Zustandsautomat sowie einen eigenen Programmzahler, einen eigenen Registersatz und einen eigenen Stack

**Implementierung auf Benutzerebene** Thread - Bibliothek ubernimmt das Scheduling und Dispatching fur Threads. Der Kernel merkt nicht von Threads

**Implementierung auf Kernelebene** Prozess ist nur noch Verwaltungseinheit fur Betriebsmittel. Scheduling-Einheit ist hier der Thread, nicht der Prozess

**Grunde fur Threads** Thread-Kontext-Wechsel gehen schneller als Prozess-Kontext-Wechsel, Parallelisierung der Prozessarbeit. Hort auf Netzwerkverbindungswunsche, fuhrt Berechnungen durch, kummert sich um das User-Interface

5 CPU-Scheduling

5.1 Grundlagen

- CPU-Vergabe in Abhangigkeit der Scheduling-Einheit → Prozess- oder Threadbasiert
- CPU-Vergabe in Abhangigkeit des Prozesstyps → Prozess A CPU-lastig, Prozess B Ein-/Ausgabelastig

**Scheduler** plant die Betriebsmittelzuteilung (teil des Prozessmanagers)

**Dispatcher** zustandig fur den Prozesswechsel (teil des Prozessmanagers)

**Zeitscheibenverfahren - (Timesharing)** CPU-Zuteilung durch das Betriebssystem. Wenn laufender Prozess auf ein Ereignis wartet oder eine bestimmte Zeit den Prozessor nutzte → nachster Prozess. Zeit abhangig vom Quantum, was wiederum abhangig ist von CPU und Betriebssystem

5.1.1 Scheduling - Kriterien und Ziele

Fairness - jeder Prozess erhalt eine garantierte Mindestzuteilung. Effizienz - Moglichst volle Auslastung der CPUs. Antwortzeit - soll moglichst minimiert werden. Verweilzeit (Durchlaufzeit) - Wartezeit von Prozessen soll moglichst klein sein. Durchsatz - Maximierung der Auftrage, die ein Betriebssystem in einem Zeitintervall durchfuhrt.

5.2 Scheduling Verfahren

**Non-preemptive Scheduling** (nicht verdrängend) Prozess wird nicht unterbrochen bis er fertig ist. Ungeeignet für Echtzeitverarbeitung oder Dialogbetrieb mit konkurrierenden Benutzern

**Preemptive Scheduling** (verdrängend) rechnende Prozesse können verdrängt werden. Geeignet für konkurrierende Benutzer, aber Zeitscheibentechnik erforderlich

5.2.1 Scheduling Algorithmen für Batch-Prozesse

**First Come First Serve (FCFS)** Nach der Reihenfolge des Eintreffens (*non-preemptive*)

**Shortest Job First (SJF)** Prozess mit kürzester Bedienzeit als erstes (*non-preemptive*)

**Shortest Remaining Time Next (SRTN)** Prozess mit der kürzesten verbleibenden Restzeit im System (*preemptive*)

5.2.2 Scheduling Algorithmen für interaktive Prozesse

**Round Robin (RR)** FCFS in Verbindung mit Zeitscheibe. **Warteschlange beachten!** Leistung hängt von der Länge der Zeitscheibe ab (kürzeres Quantum=rechenintensive Prozesse, längeres Quantum und/oder höhere Priorität=I/O-Intensive Prozesse). Verhältnis Arbeitzeit/Umlaufzeit darf nicht zu klein sein → Durchsatz schlecht (*preemptive*)

**Priority Scheduling (PS)** Prozesse mit höchster Priorität als nächstes. Dynamische und statische Prioritäten und Kombinationen sind möglich (*preemptive* oder *non-preemptive*)

**Lottery Scheduling** zufällige Vergabe von CPU-Zeit

5.2.3 Multi-Level-Scheduling

Mehrere Arten von Jobs unterschiedlicher Priorität

**Multi-Level-Feedback-Scheduling** Prozess kann in eine Warteschlange höherer oder niedriger Ordnung wechseln

5.2.4 Scheduling in Realtime-Systemen

**Hard-Realtime-Systeme** müssen unter allen Umständen rechtzeitig reagieren

**Soft-Realtime-Systeme** gelegentliche Verzögerungen erlaubt

**Statische Scheduling-Algorithmen** Scheduling-Entscheidung bereits vor dem Start des Systems bekannt

**Dynamische Scheduling-Algorithmen** Scheduling-Entscheidungen werden zur Laufzeit getroffen

5.3 Echtzeit-Scheduling

Berechnungen funktional korrekt und rechtzeitig, garantierte Einhaltung von Zeitschranken, Vorhersagbarkeit. Echtzeitsystem besteht aus einer Menge aus Tasks, die quasi parallel arbeiten. Task = Prozess, Thread, etc. → wird periodisch ausgeführt und hat eine bestimmte Priorität. Scheduler teilt den Tasks nach einem bestimmten Algorithmus jeweils die CPU zu

**Echtzeitsysteme** Unterstützt harte Echtzeitanforderungen (Garantien), ist auf den Worst-Case hin optimiert, 'Dringende' Prozesse verdrängen weniger wichtige, **Vorhersagbarkeit** ist wichtig. Sofortige Bearbeitung anstehender Aufträge (Ein-/Ausgabe)

**Universal-Betriebssystem** Unterstützt nur 'weiche' Echtzeitanforderungen (keine Garantien), Optimierung auf die Unterstützung verschiedenster Anwendungsfälle, Reaktionszeit nicht im Vordergrund. Alle Prozesse werden (weitgehend) gleich (fair) behandelt. **Durchsatz** und Antwortzeitverhalten wichtig, Blockweise Ein-/Ausgabe, Sammlung 'gleicher Aufträge'

5.3.1 Statisches Scheduling

**Rate Monotonic Scheduling (RMS)** Je kürzer die Periode T desto höher die Priorität. Bis zur Deadline D müssen alle Rechenzeiten des jeweiligen Prozesses erledigt sein. **Vorgehen:** Deadlines/Periodendauer einzeichnen. In jedem Prozess alle seine benötigten Rechenzeiten in jeder Periode einzeichnen, dies nach Priorität machen. Die Priorität wird in jedem Schritt neu überprüft. Dabei gegebenenfalls den Prozess unterbrechen und mit dem Prozess der höheren Priorität weitermachen

**Major cycle** (auch Hyper Period) nach einem major cycle wiederholt sich das Verhalten des Systems

**Critical Instant (CI)** alle Tasks wollen zur selben Zeit starten → stellt das **worst case** szenario dar. Wird dieser ohne Deadline-Überschreitung überstanden, so werden immer alle Deadlines eingehalten

**Response-Time Analyse (RTA)** maximale Antwortzeit  $R_i$  einer Task  $i$  →  $R_i$  = eigene Ausführungszeit  $C_i$  + 'Störungen' - System ist echtzeitfähig, wenn für alle Tasks  $i$  die Antwortzeit der Deadline ist →  $R_i D_i$

5.3.2 Dynamisches Scheduling

zur Laufzeit wird entschieden, welche Task die höchste Priorität bekommt

**Earliest-Deadline First (EDF)** Task mit der nächsten absoluten Deadline hat die höchste Priorität und die absolute Deadline ist abhängig vom Zeitpunkt. **Vorteil** ist hohe Prozessorauslastung, einfacher Schedulability Test. **Nachteil** ist, für den Scheduler entsteht ein erheblich größerer und im Aufwand schwankender Overhead als bei statischen Verfahren

6 Synchronisation und Kommunikation

6.1 Grundlegendes zur Synchronisation

6.1.1 Nebenläufigkeit, atomare Aktionen und Race Conditions

**Nebenläufigkeit** parallele bzw. quasi-parallele Ausführung von Befehlen auf einer oder mehreren CPUs. Prozesse bzw. Threads werden in Multiprogramming-Systemen nebenläufig ausgeführt - Programmierer kann die Ablaufreihenfolge nicht beeinflussen

**Atomare Aktion** Codebereiche des Betriebssystem (oder Anwendung), die in einem Stück ausgeführt werden müssen und nicht unterbrochen werden dürfen. Prozesse, die atomare Aktionen durchführen müssen exklusiv Betriebsmittel zugeteilt werden und dürfen dem Prozess nicht entzogen werden - Unterbrechung durch eine Scheduling-Entscheidung des Betriebssystem jederzeit möglich

**Beispiel** verkettete Listen werden vom Betriebssystem verwaltet → von mehreren Prozessen zugreifbar - Operation besteht aus zwei Kommandos - Prozess möchte ein neues Objekt an die Liste hängen, genau wie ein anderer Prozess und wird unterbrochen - Prozess B, welcher die selben Variablen wie Prozess A verwendet, könnte ein Problem verursachen

**Lost-Update-Problem** Inkonsistenzen bei gemeinsam genutztem Zähler, der von zwei Prozessen verändert wird

6.1.2 Kritische Abschnitte und wechselseitiger Ausschluss

**kritische Abschnitte** sind Programmteile, die nicht unterbrochen werden dürfen. Codeabschnitt, der zu einer Zeit nur durch einen Prozess durchlaufen wird - nebenläufige Prozesse dürfen nicht dazu treten - Prozess bzw. Thread, der den kritischen Abschnitt belegt, darf durch eine Scheduling-Entscheidung des Betriebssystems nicht beeinflusst werden

**wechselseitiger Ausschluss** schützt einen kritischen Abschnitt - Prozesse, die einen kritischen Abschnitt ausführen wollen, müssen warten bis dieser frei ist - Illusion einer atomaren Anweisungsfolge wird erschaffen - nur Illusion, da ein nebenläufiger Prozess zwischendurch die CPU erhalten kann

**Busy Waiting** Warten und ständige Abfragen eines Sperrkennzeichens am Eingang des kritischen Abschnitts, der freigegeben werden muss  
**Effizienter:** Prozess, der einen kritischen Abschnitt belegen will, schlafen legen und ihn wieder wecken wenn der Abschnitt frei ist. Zu einer Zeit ist immer nur ein Prozess im kritischen Abschnitt

**Kriterien nach Dijkstra** *Mutual Exclusion:* Zwei oder mehr Prozesse dürfen sich nicht gleichzeitig im gleichen kritischen Abschnitt befinden. Es dürfen keine Annahmen über die Abarbeitungsgeschwindigkeit und die Anzahl der Prozesse bzw. Prozessoren gemacht werden. Kein Prozess außerhalb eines kritischen Abschnitts darf einen anderen nebenläufigen Prozess **blockieren**. *Fairness Condition:* Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss ihn irgendwann betreten dürfen (**kein ewiges Warten**)

6.1.3 Eigenschaften nebenläufiger Prozesse

**Blockieren** Ein Prozess belegt Betriebsmittel, welche ein anderer Prozess benötigt

**Verhungern** Prozess erhält trotz Rechenbereitschaft keine CPU-Zeit zugeteilt

**Verklemmung** zwei oder mehrere Prozesse halten jeder für sich ein oder mehrere Betriebsmittel belegt und versuchen ein weiteres zu belegen, das aber von dem anderen Prozess belegt wird → kann zu **Dead Lock** führen

**Dead Lock** Verklemmung, es liegt eine Abhängigkeit vor. Kein Prozess gibt seine Betriebsmittel frei, und alle Prozesse warten daher ewig. Bedingungen: **Mutual Exclusion** für die benötigten Betriebsmittel Prozesse *belegen* Betriebsmittel und *fordern weitere* an **Kein Entzug** eines Betriebsmittels **möglich** Zwei oder mehrere Prozesse **warten** in einer Warteschleife auf **weitere Betriebsmittel**

**Sicherheit** es befinden sich nie zwei Prozesse in einem kritischen Abschnitt und es kommen nie Verklemmungen vor

**Lebendigkeit** ein Prozess kann nach einer evtl. Wartezeit in einen kritischen Abschnitt eintreten (wenn alle gewünschten Zustände eintreten)

6.2 Synchronisationskonzepte

6.2.1 Sperren

Eine einfache Lösung zur Realisierung von kritischen Abschnitten ist busy waiting (aktives Warten). Ein Prozess testet eine sog. Synchronisationsvariable solange, bis die Variable einen Wert hat, der den Zutritt erlaubt. Dieses Polling ist oft unwirtschaftlich, da es eine Verschwendung der CPU-Zeit bedeutet, aber: Spinlocks in Betriebssystemen sind oft anzutreffen (bei sehr kurzen Wartezeiten bei Multiprozessoren gut). Meist ist es besser, einen Prozess 'schlafen' zu legen und erst wieder wecken, wenn er in den kritischen Abschnitt darf.

**Hardwareunterstützung durch Maschinenbefehle** Problem: Zwei CPUs führen einen *TSL-Befehl* (Test and Set Lock) aus und die Teiloperationen überlappen sich. Ausführung eines TSL-Befehls in einem Speicherzyklus erledigt → funktioniert bei Multiprocessing

6.2.2 Semaphore

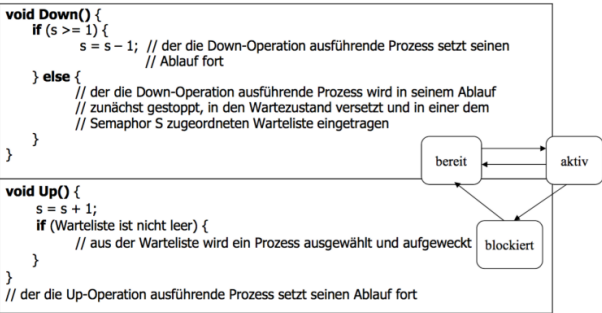
Dijkstra (1965) führte das Konzept der Semaphore zur Lösung des Mutual Exclusion Problems ein. Zwei elementare Operationen:

**P()** P-Operation, auch **Down**-Operation genannt. Aufruf bei Eintritt in den kritischen Abschnitt, Operation auf Semaphor. Aufrufender Prozess wird in den Wartezustand versetzt, wenn sich ein anderer Prozess im kritischen Abschnitt befindet → Warteschlange

**V()** V-Operation, auch **Up**-Operation genannt. Aufruf bei Verlassen des kritischen Abschnitts, evtl. wird einer der wartenden Prozesse aktiviert und darf den kritischen Abschnitt betreten

```
Nutzung des Semaphors könnte so aussehen:
Down(); // kritischer Abschnitt besetzt?
c=counter.read(); // kritischer Abschnitt
c++;
counter.write(c);
Up(); // Verlassen des kritischen Abschnitts
// Aufwecken eines wartenden Prozesses
```

- s ist der Semaphor-Zähler, Init: s >= 1



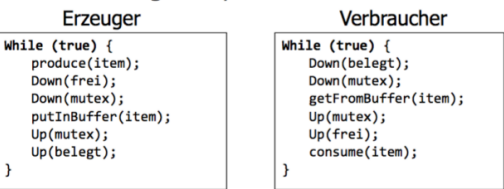
**Mutex** Wenn man auf den Zähler im Semaphor verzichten kann, kann eine einfachere Form angewendet werden. Ein Mutex ist leicht und effizient zu implementieren. Ein Mutex ist eine Variable, die nur zwei Zustände haben kann. *locked* und *unlocked*. Man braucht also nur 1 Bit zur Implementierung

6.3 Synchronisationsprobleme

6.3.1 Erzeuger-Verbraucher-Problem

Ein oder mehrere Erzeugerprozesse (producer) produzieren, ein oder mehrere Verbraucherprozesse (consumer) konsumieren. Flusskontrolle erforderlich. Erzeuger legt sich schlafen, wenn Puffer voll ist und wird vom Verbraucher aufgeweckt, wenn wieder Platz ist. Verbraucher legt sich schlafen, wenn Puffer leer ist und wird vom Erzeuger wieder aufgeweckt, wenn wieder was drinnen ist

- Lösung mit drei Semaphoren:
  - **mutex** für den gegenseitigen Ausschluss beim Pufferzugriff
  - **frei** und **belegt** zur Synchronisation



**Initialisierung:**  
belegt = 0; // Verbraucher muss erst mal warten, zählt die belegten Puffer  
frei = N (Puffergröße); // Am Anfang ist Puffer leer, zählt die leeren Puffer  
mutex = 1; // Mutual Exclusion Zugang nur für Pufferbearbeitung

→ am Anfang darf nur der Erzeuger etwas tun Nach Tanenbaum (2009)

6.4 Synchronisation in Programmiersprachen

6.4.1 Monitore

Abstrakter Datentyp, Erzeugung und Anordnung der Semaphor-Operationen dem Compiler überlassen. Menge von Prozeduren und Datenstrukturen - werden als Betriebsmittel betrachtet → für mehrere Prozesse zugänglich, kann nur von einem Prozess zu einer Zeit genutzt werden. Zusammengefasst: ein Objekt, das den Zugriff auf gemeinsame Daten über kritische Bereiche in Zugriffsmethoden realisiert, gemeinsam benutzte Daten werden durch Synchronisationsvariable geschützt

```
Monitor ProducerConsumer {
    final static int N = 5;
    static int count = 0;
    condition not_full;
    condition not_empty;
    int Puffer[N-1];

    void synchronized insert(item integer) {
        //Warten, bis Puffer wieder nicht mehr voll ist
        if (Count == N) {wait(not_full);}
        count++; // Item in Puffer einfüegen
        //Signalisieren, dass Puffer nicht mehr leer ist
        if (count == 1) {signal(not_empty);}
    }

    int synchronized remove() {
        // Warten, bis Puffer nicht mehr leer ist
        if (count == 0) {wait(not_empty);}
        count--; // Item aus Puffer entfernen
        // Signalisieren, dass Puffer nicht mehr voll ist
        if (count == (N-1)) {signal(not_full);}
        return (item);
    }
}

class UseMonitor {
    ProducerConsumer mon = new ProducerConsumer();

    void producer() {
        while (true) {
            // Item erzeugen
            mon.insert(item);
        }
    }

    void consumer() {
        while (true) {
            item = mon.remove();
            // Item verbrauchen
        }
    }
}
```

**synchronized** ermöglicht eine Zugriffsrealisierung. Methoden und Codeblöcke, die mit *synchronized* gekennzeichnet wurden, bilden gemeinsam einen kritischen Abschnitt. Bei Eintritt eines Threads in einem mit *synchronized* gekennzeichneten Bereich werden alle kritischen Bereiche des Objekts gesperrt

6.4.2 Methoden zur expliziten Synchronisierung

wait(): Thread wartet auf ein *notify* eines anderen Threads  
notify(): weckt mindestens einen wartenden Thread auf  
notifyAll(): alle Threads, die auf das gleiche Objekt warten, werden geweckt  
wait() und notify() dürfen nur innerhalb eines mit *synchronized* geschützten Abschnitt aufgerufen werden

6.4.3 C# Monitore

```
public sealed class Monitor {
    private Monitor();
    public static void Enter(object obj);
    public static void Exit(object obj);
    public static void Wait(object obj, int msec);
    public static void Pulse(object obj);
    public static void PulseAll(object obj);
    public static bool TryEnter(object obj, int msec);
    public static bool TryEnter(object obj);
    ...
}
```

6.5 Behandlung von Deadlocks

6.5.1 Ignorieren

Nur sinnvoll, wenn Deadlocks selten auftreten oder Kosten der Vermeidung sehr hoch sind. Unakzeptabel in Echtzeitsystemen oder Kosten eines Deadlocks sehr hoch sind

6.5.2 Erkennen und beheben

Deadlocks sind grundsätzlich zugelassen, werden zur Laufzeit erkannt. Deadlockerkennung z.B. über Betriebsmittelbelegungsgraphen. Beheben durch → **Unterbrechung** (Entziehung der Ressource eines Prozesses, nicht immer möglich), **Rollbacks** (Checkpoints regelmäßig setzen), **Prozessabbruch**, **Transaktionsabbruch**

6.5.3 Dynamisch Verhindern

Verhindernd, dass die Ressourcenspur in den gefährlichen Bereich (**unsicheren Zustand**) gerät. Die Ressourcen vorsichtig zuteilen

**Sichere und unsichere Zustände** Ein Zustand heißt sicher, wenn es mindestens eine Scheduling-Reihenfolge gibt, die nicht zum *Deadlock* führt  
Beispiel: Insgesamt sind 10 Instanzen einer Ressource verfügbar



Abbildung 6.9: Nachweis, dass der Zustand in (a) sicher ist

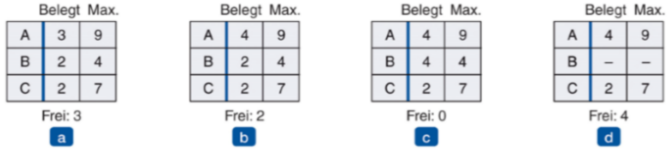


Abbildung 6.10: Nachweis, dass der Zustand in (b) unsicher ist

**Banker-Algorithmus** Der Banker räumt ein 'Kreditlimit' in und achtet darauf, dass sein verleihbares Kapital beim Ausleihen nicht überschritten wird → Ziel: unsichere Zustände verhindern

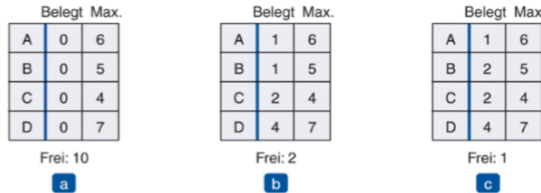


Abbildung 6.11: Drei Belegungszustände: (a) Sicher (b) Sicher (c) Unsicher

Nachteile: Bestimmung des Betriebsmittels-Bedarfs a-priori meist nicht möglich. Außerdem braucht der Algorithmus viel Rechenzeit und Speicherplatz. Das Verfahren garantiert zwar das jeder Prozess in endlicher Zeit seine Betriebsmittel bekommt, liefert aber keine Zeitschranken

6.5.4 Vermeiden

**Mutual exclusion** unvermeidbar, manchmal ist **Spooling** (Nur ein Prozess kann zugreifen und Ressource wird virtualisiert)

**Hold-and-Wait** Prozess muss alle benötigten Ressourcen auf einmal anfordern → danach muss er nicht mehr auf die Ressource warten. Probleme: Ressourcenbedarf nicht immer zu Beginn bekannt, Ressourcen sind länger wie nötig ist belegt. *Variante:* Bevor eine zusätzliche Ressource angefordert wird, müssen alle bisher belegten zuerst freigegeben werden

**No preemption** Ist keine Option, z.B. Drucker → riesige Wartezeit

**Circular Wait** Ressourcen aufsteigend nummeriert. Ressourcen nur in aufsteigender Reihenfolge anforderbar → Belegungsgraph bleibt immer zykl-frei. Probleme: Alle Prozesse müssen sich an die Vereinbarung halten, Ändert sich die Nummerierung müssen alle Prozesse angepasst werden

6.5.5 Echtzeitsysteme

Menge von periodischen Tasks mit festen Prioritäten. Vorhersagbar, d.h. Laufzeit und Ressourcenbedarf jeder Task bekannt

**Priority Ceiling Protokoll** Jede Ressource bekommt eine Ceiling Priority (maximale Priorität des Tasks die sie nutzen wollen). Belegt ein Task eine Ressource, bekommt sie die Ceiling Priority dieser Ressource → Unterlaufen der *Circular Wait* Bedingung

7 Kommunikation

Austausch von Informationen zwischen (Menschen), (Mensch und Maschine → Rechnersystem), (Maschinen → Prozesse/Threads)

**Verbindungsorientiert** Vor dem Datenaustausch wird eine logische Verbindung aufgebaut und danach wieder abgebaut. Kommunikationspartner kennen Verbindungspartner

**Verbindungslos** Daten ohne Verbindungsmanagement vom Sender zum Empfänger übertragen. Senderadresse in der Nachricht mit übertragen

**Speicherbasiert** Daten werden in einem gemeinsamen Speicher ausgetauscht oder 'geshared'

**Nachrichtenbasiert** Zwischen Prozessen/Threads werden Nachrichten ausgetauscht → einheitliches Protokoll

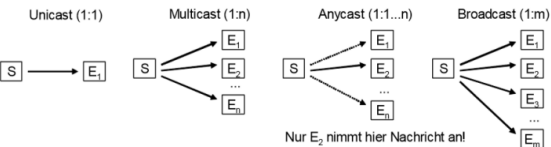
**synchrone Kommunikation** ist blockierend

**asynchrone Kommunikation** ist nicht blockierend

**Halbduplex** Nur einer der Partner sendet zu einer Zeit

**Vollduplex** beide Partner können unabhängig voneinander senden

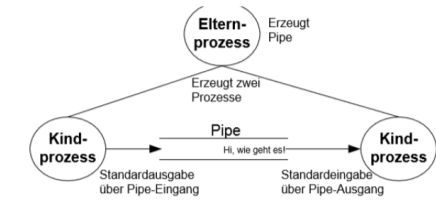
Varianten der Empfängeradressierung



**Pipes** Spezieller *unidirektionaler* Mechanismus - bidirektionale Kommunikation durch Nutzung mehrerer Pipes - bidirektionale Kommunikation kann sowohl halb- als auch vollduplex betrieben werden. Standardeingabe eines Prozesses kann mit der Standardeingabe eines weiteren Prozesses verbunden werden

Pipes - Programmierung

- Erzeugen - Unix pipe() oder popen()
- Schließen - Unix close() oder pclose()
- Elternprozess erzeugt Pipe und vererbt sie an den Kindprozess - blockierend und nicht blockierend einsetzbar (Pipe voll → Sendermodus blockiert, Pipe leer → Leseprozess blockiert)



**Verteilte Kommunikation zwischen Prozesse/Threads** Betriebssystem stellt Kommunikationssystem bereit, Netzzugang erforderlich

8 Speicherverwaltung

Reihenfolge, nach Zugriffsgeschwindigkeit: Register → Cache → Hauptspeicher → Magnetische Platten oder SSD → Magnetische Bandlaufwerke

Lokalitätsprinzip

- **Zeitlich:** Daten/Code-Bereiche, die gerade genutzt werden, sollten für die nächste Nutzung bereitgehalten werden
- **Örtlich:** Benachbarte Daten beim Zugriff auch gleich in schnelleren Speicher laden
- **Working Set:** Lokaler Code- bzw. Datenabschnitt, in dem ein Prozess vorwiegend arbeitet

**Adressraumbelegung** Durch den Adressraumbelegungsplan bestimmt - Festlegung im Betriebssystem - Ausrichtung auf Maschinenwörter → optimaler Zugriff - Compiler/Interpreter organisiert Bereich für Anwendungsprogramme

**Cache** Schneller Speicher mit kleiner Kapazität, wo Daten ein- und ausgelagert werden → möglichst viele Daten im Cache finden. **Cache hit** oder **Cache miss** → Rückgriff auf langsamere Ebene - Ziel auf allen Ebenen: Möglichst hohe Trefferrate. **Cache-Prinzip:** Je häufiger auf Daten zugegriffen wird, desto schneller sollte der Zugriff sein. **Einlagern:** Ganzer Datenblock aus Umgebung wird nach einem Cache miss geladen → Lokalitätsprinzip - Cache ist immer komplett gefüllt

**least recently uses (LRU)** Austausch des am längsten nicht mehr referierten Cache-Block. HW-Alterungszähler für jeden Cache-Block, bei Zugriff wird dieser auf 0 gesetzt, alle anderen um 1 erhöht. Block mit dem höchsten Zählerstand wird ausgelagert

**least frequently accesses (LFA)** Austausch des am wenigsten benutzten Cache-Blocks. HW-Referenzzähler auf den Cache-Block um 1 erhöht und durch das Betriebssystem nach einem Zeitintervall bzw. nach Austausch auf 0 gesetzt wird

**least recently loaded (LRL)** Austausch des am längsten geladenen Cache-Blocks. Festhalten und späterer Vergleich des Ladezeitpunktes → FIFO-Verfahren

**Zufallsauswahl** Auszutauschender Block wird zufällig gewählt → geringer Aufwand

8.1 Schreiboperationen

**Durchschreiben (write through)** Jede Schreiboperation auf einem Cache-Block wird sofort auch auf der unterliegenden Speicherebene durchgeführt

**Zurückschreiben (write back)** Schreiboperationen wirken zunächst nur auf den Cache-Block. Erst beim Austausch des Blocks werden Daten auch an der langsameren Ebene geändert

**Schreiben auf Anforderung (write on demand)** Schreiboperationen an der langsameren Ebene werden nur auf besondere Anforderung durchgeführt (bei Prozesswechsel)

**Trefferrate** hängt ab von der Cache-Größe, von der Ersetzungsstrategie, von der Programm- und Datenorganisation

8.2 Mechanismen der Speicherverwaltung

**Monoprogramming** einfachste Form der Speicherverwaltung → nur ein Programm läuft zu einer Zeit

**Mit festen Partitionen** Aufteilung eines Speichers in feste Teile (Partitionen) - Multiprogramming um Verbesserung der CPU-Auslastung möglich - Job wird in eine Queue eingetragen

8.2.1 Speicherverwaltung bei Multiprogramming mit Swapping

**Grundgedanke: Timesharing** Es passen nicht immer alle Prozesse in den Hauptspeicher - Prozess wird im Gesamte geladen - Prozess wird irgendwann auf einen Sekundärspeicher ausgelagert - Löcher können durch Kombination benachbarter Speicherbereiche eliminiert werden (aufwändig)

**Hauptunterschied zu festen Partitionen** Anzahl, Speicherplatz und Größe des für einen Prozess verwendeten Speicherbereichs variieren dynamisch - Prozess wird da geladen, wo Platz ist

8.3 Grundprinzipien des virtuellen Speichers

Speichergröße eines Programms darf den vorhandenen physikalischen Hauptspeicher überschreiten - Prozess kann ablaufen, wenn er nur teilweise im Hauptspeicher ist - Programmierer soll sich am besten nur mit einem linearen Adressraum befassen - Betriebssystem hält die gerade benutzten Teile im Hauptspeicher und den Rest auf der Festplatte

**Virtueller Adressraum** Wird vom Prozess gesehen, ist in *Seiten* aufgeteilt

**Realer Adressraum** RAM, normalerweise kleiner als virtueller Adressraum, ist in *Seitenrahmen* aufgeteilt

**Seite** Teile eines virtuellen Adressraums. Mehrere Speicherseiten bilden Speicherabbild eines Prozesses - nur die benötigten Speicherseiten müssen im Arbeitsspeicher geladen sein, während der Prozess läuft - Größe meist gering, Anzahl beliebig groß

**Seitenrahmen** Teile eines realen Adressraums. *Seiten* und *Seitenrahmen* müssen zwingend die gleiche Größe haben

8.3.1 Strategien zur Verwaltung von virtuellem Speicher

**Paging** Umlagerung zwischen Hauptspeicher und Festplatte. Jeder Prozess darf alle Adressen verwenden, welche die HW-Architektur des Rechners vorgibt - bei Systemen mit 32-Bit-Adressen kann jeder Prozess einen Adressraum von 4GiB verwenden

**Memory Management Unit (MMU)** CPU sendet virtuelle Adressen an die MMU, welche daraus die reale Adresse ermittelt und sie an den Hauptspeicher weiterleitet

**Adressumsetzung** Virtuelle Adresse ist in virtueller *Seitennummer* und einem *Offset* geteilt - virtuelle Seitennummer ist ein Index auf die *Seitentabelle* - im Eintrag steht die Rahmen-Nummer, falls die Seite einem Rahmen zugeordnet ist. Befindet sich eine angesprochene Adresse nicht im Hauptspeicher → MMU verursacht bei der MMU einen **Trap** - Beispiel: MOVE R1, 8196 → Adresse befindet sich in Page 2 des virtuellen Adressraumes → MMU transformiert 8196 → 4

Mapping muss schnell sein - in großen Adressräumen sind sehr große Seitentabellen möglich (32 Bit Adressraum → 1 Mio. Einträge, Seitengröße von 4 KiB)(Bei 4 Byte pro Eintrag → 4MiB Hauptspeicher notwendig)

8.4 Optimierung der virtuellen Speichertechnik

**Virtuelle Adressierung aufwändig, da...** viele, umfangreiche Tabellen benötigt werden - Teil der Festplatte wird als Paging-Area verwendet -



laufend muss untersucht werden, ob Seiten im Hauptspeicher bleiben oder auf die Festplatte auszulagern sind

**Optimierung durch Adressumsetzpuffer...** größere Seiten (64-Bit-Prozessoren)

**Translation Lookaside Buffer (TLB)** Adressumsetzpuffer ist ein schnelles Speicherabbild - Zuordnung von virtuellen auf reale Adressen für die aktuell am *häufigsten* benötigten Adressen - bei der Adressumsetzung wird zuerst in dem TLB geschaut → **Hit**: Kein Zugriff auf Seitentabelle nötig - Einsparung von Hauptspeicherezugriffen. Dadurch ist eine beträchtliche Leistungsoptimierung möglich. Der TLB ist Bestandteil der MMU. Ein TLB-Eintrag enthält: Virtuelle Seitennummer, Verweis auf den Seitenrahmen im Hauptspeicher, Tag zur Adressraum-Identifikation → Tagged TLB

8.4.1 Optimierung durch invertierte Seitentabellen

angelegte Tabelle, in der man reale Adressen auf virtuelle abbildet - ein Eintrag pro Frame in einer invertierten Seitentabelle - **Vorteil**: weniger Tabelleneinträge → nur noch so viele wie Seitenrahmen im Hauptspeicher zu Verfügung stehen - **Nachteil**: In der Seitentabelle ist keine Ordnung nach virtuellen Adressen → Suche aufwändiger

**Virtuelle Speichertechnik Vorteile**: Prozesse müssen nicht komplett speicherresident sein - lineare Speicheradressierung - beim Prozesswechsel behält ein Prozess seine hauptspeicherresidenten Seiten - Anwendungsprogramme können vollen virtuellen Adressraum nutzen

8.5 Speicherbelegung und Vergabe

8.5.1 Speicherbelegungsstrategien

**Speicherbelegungstabellen** Verwaltet die Belegung des Hauptspeichers

**Bitmap** Bits(0 = frei, 1 = belegt) werden einem Rahmen zugeordnet. Nebeneinander liegende Nullen deuten einen freien Hauptspeicherbereich an

8.5.2 Vergabestrategien

**Sequentielle Suche** First-Fit: erster geeigneter Bereich wird vergeben

**Optimale Suche** Best-Fit: passender Bereich, um Fragmentierung zu vermeiden, wird vergeben

**Buddy Technik** Stetige Halbierung des Speichers → Such nach dem kleinsten Bereich. Bereiche immer in 2<sup>n</sup> aufteilen. Bei Hauptspeicherfreigabe werden Rahmen wieder zusammengefasst, die vorher getrennt wurden

8.6 Entladen (Cleaning)

Legt den Zeitpunkt fest, wann eine modifizierte Seite auf die Paging-Area geschrieben wird

**Demanding-Cleaning** Bei Bedarf! Seite lange im Hauptspeicher, aber Verzeigerung bei Seitenwechsel

**Precleaning** Präventives Zurückschreiben, wenn Zeit ist! Frames in der Regel verfügbar

**Page-Buffering** Listen verwalten! Modified List: Zwischenpuffer, Unmodified List: Für Entladen freigeben

8.6.1 Lokale vs Globale Seitenersetzung

**Lokale Strategie** Jeder Prozess hat eine feste Anzahl an Seitenrahmen. Beim *Working Set* Algorithmus sinnvoll

**Globale Strategie** Meist besser: *FIF*, *LRU*, ...; Prozess entscheidet dynamisch, wie viel Speicher dem Prozess zugeteilt wird

9 Dateien & Geräte

9.1 Dateien und Verzeichnisse

persistente Speicherung von Informationen - von mehreren Prozessen gleichzeitig zugreifbar sein. *Dateien* = abstrakter Mechanismus zur Spe-

icherung und zum Wiederauffinden von Informationen

**Pseudodateien** z.B. Liste der defekten Blöcke eines Datenträgers

**Verzeichnisse** Verwaltung der Struktur des Dateisystems

**Metadaten** Informationen über die Datei selbst

9.1.1 Dateizugriff

**Sequentieller Zugriff** Bytes werden nacheinander gelesen. Vor- und Zurückspringen nicht möglich

**Wahlfreier Zugriff** Bytes können in beliebiger Reihenfolge gelesen werden (Datenbanksysteme)

**Master Boot Record (MBR)** Ist auf Sektor 0, dient zum Booten des Rechnersystems. Enthält die *Partitionstabelle*

**Partition Table** Liegt am Ende der MBR. Festplatte kann in mehrere Partitionen eingeteilt werden

**Bootblock** Wird beim Hochfahren gelesen und ausgeführt

**Superblock** Enthält Verwaltungsinformationen zum Dateisystem

**Free Blocks** Gibt die freien Blöcke des Dateisystems an

**Rootverzeichnis** Enthält den Inhalt des Dateisystems

**I-Nodes** Einträge im Inhaltsverzeichnis des Dateisystems

9.1.2 Implementierung von Dateien

Belegung durch verkettete Liste → Sequentieller Zugriff unkompliziert → Wahlfreier Zugriff extrem langsam. Verkettete Liste mit Tabelle im Arbeitsspeicher → *File Allocation Table* (FAT)

9.1.3 File Allocation Table (FAT)

Speichert Datei als verkettete Liste von Plattenblöcken, hält die Information über Verkettung der Blöcke im Hauptspeicher. Vorteil: Wahlfreier Zugriff schneller, Verfolgen der Kette passiert im RAM. Nachteil: nicht praktikabel bei großen Festplatten

9.1.4 Implementierung von Verzeichnissen

**Verzeichniseintrag** Enthält Informationen zum Auffinden der zugehörigen Plattenblöcke (Nr. des 1. Blocks | Nr. des I-Nodes). Üblicherweise nicht sortiert. Dateiattribute können im Verzeichniseintrag gespeichert werden. Dateiattribute können im I-Node gespeichert werden (nur Verweis auf I-Node)

**Hard Link** Verzeichniseinträge verweisen auf den selben I-Node

**Symbolischer Link** Spezielle Datei vom Typ *LINK*, welcher Pfad zur verlinkten Datei enthält

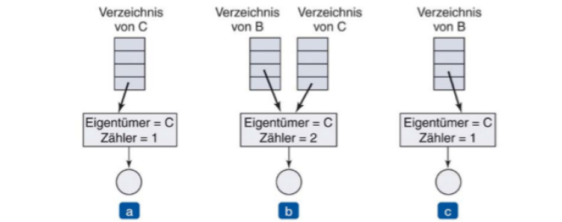


Abbildung 4.17: (a) Situation vor der Verlinkung (b) Situation nach der Erzeugung des Links (c) Situation, nachdem der Eigentümer die Datei entfernt hat

9.1.5 Virtuelle Dateisysteme

**Beispiele:** NTFS, ext, FATx für Festplatten - CD/DVD Dateisysteme - NFS (Network File System)

Systemaufrufe (open, read, etc) gehen an die obere Schnittstelle des VFS - konkretes Dateisystem muss die untere Schnittstelle implementieren

9.1.6 Dateisystem Verwaltung

**Datenrate vs. Platteneffizienz** Je mehr Daten mit einem einzigen Plattenzugriff geholt werden können, desto besser → Datenrate steigt mit der Blockgröße

9.1.7 Freie Blöcke verwalten

**Verkettete Liste** 1KB-Block kann bis zu 255 Blocknummern (a 32 Bit) aufnehmen

**Bitmap** 1 Bit pro Block (0 = frei | 1 = belegt)

9.1.8 Konsistenz

2 Tabellen mit Zählern für jeden Block. Tabelle 1 zählt, wie oft jeder Block in einer Datei vorkommt. Tabelle 2 zählt, wie oft jeder Block in der Freibereichsliste vorkommt. Jeder Block sollte genau einmal in Tabelle 1 oder in Tabelle 2 auftauchen

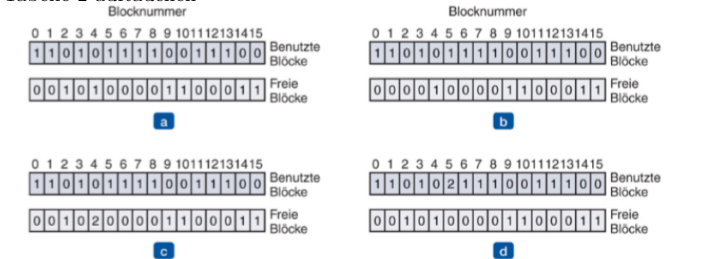


Abbildung 4.27: Zustände von Dateisystemen (a) Konsistent (b) Fehlender Block (c) Doppelt vorhandener Block in der Freibereichsliste (d) Doppelt vorhandener Datenblock

**Gleicher Block** Lösung: in Freibereichsliste einfügen/löschen

**Doppelt vorhandener Block in Tabelle 2 (freie Blöcke)** nicht möglich, wenn freie Blöcke in einer Bitmap verwaltet werden → Freibereichsliste korrigieren

**Doppelt vorhandener Block in Tabelle 1 (benutzte Blöcke)** falls Datei gelöscht → Block sowohl benutzt als auch frei. Lösung: Block kopieren und Kopie einer der beiden Dateien zuweisen oder Benutzer informieren das Datei defekt ist

**Prüfung auf Dateiebene** hard Links → zählen wie oft eine Datei in Verzeichnissen auftaucht - Ergebnis mit Link-Zähler im I-Node der Datei vergleichen. **Link-Zähler zu hoch**: Nach dem Löschen der Datei in allen Verzeichnissen, wäre der Link-Zähler immer noch > 0 → **Lösung**: Link-Zähler korrigieren - **Link-Zähler zu klein**: Datei wird im Verzeichnis gelöscht → Link-Zähler = 0 → I-Node und die Blöcke werden freigegeben, obwohl sie benutzt werden - **Lösung**: Link-Zähler korrigieren

**Konsistenz - Journaling - Dateisysteme**

**Beispiel: Löschen einer Datei** Löschen einer Datei aus dem Verzeichnis - Freigeben des I-Node in den Pool der freien I-Nodes - Freigeben der Plattenblöcke in den Pool der freien Blöcke

**Zugehöriger Log-Eintrag** Listet die anstehenden Aktionen auf. *Log-Eintrag* wird auf die Festplatte geschrieben - die Aktionen werden ausgeführt - Log-Eintrag wird gelöscht

9.1.9 Performanz

**Caching** Block-Cache | Puffer-Cache  
Menge von Blöcken im Speicher - Zugriff auf einen Block. Es wird geprüft ob er schon im Cache liegt. Falls nicht → wird in den Cache geladen - Suche mit Hashfunktion. **Problem**: Blöcke sollten nicht zu lange verändert im Cache liegen - selten wird mehrmals innerhalb einer kurzen Zeitspanne zugegriffen.

**LRU-Schema anpassen:** Wenn Block bald wieder gebraucht wird → hinten in die LRU-Liste einfügen - wenn Block wichtig für die Konsistenz des Dateisystems → sofort auf die Platte schreiben

**UNIX/Linux - sync Systemaufruf** Bewirkt, dass alle Blöcke sofort auf die Festplatte geschrieben werden (dauert ca. 30s)

**Write-Through-Cache** Sobald Block verändert wird → sofort auf Festplatte schreiben (sicher, aber langsam)

**Bewegungen des Plattenarms reduzieren** Blöcke, auf die der Reihe nach zugegriffen werden → sollten nahe beieinander auf der Festplatte liegen. *Bitmap* → zusammenhängende Blöcke können leicht gefunden werden. Alle I-Nodes liegen am Rand der Festplatte → durchschnittliche Entfernung I-Node und dem ersten Block der Datei = die Hälfte des Zylinderdurchmessers der Festplatte. **Besser:** I-Nodes in der Mitte platzieren | Platte in Zylindergruppen einteilen

**Festplattenspeicher defragmentieren** Fragmentierung nimmt mit dem 'Vollsein' der Festplatte zu - Zusammengehörige Blöcke sind dann auf der Platte verteilt → Festplatte regelmäßig defragmentieren (Gilt nicht für SSD Festplatten → *Wear out*)

9.1.10 Dateisysteme

**CD-ROM** Einfach strukturiert → Medium kann nur einmal beschrieben werden - keine Verwaltung der freien Blöcke nötig. **CD-R** → Dateien werden nie überschrieben, sondern am Ende der CD-R angehängt

**MS-DOS (FAT32, vFAT)** USB-Sticks, Digitalkameras, MP3-Player usw. → benutzt Datei-Allokationstabelle *FAT*  
**Beispiel:** FAT-12 mit 512-Byte-Blöcken →  $12 \cdot 512 \text{ Byte} = 2 \text{ MiB}$  Partition-sgröße. FAT mit 4096 Einträgen a 2 Byte → 8KiB Hauptspeicher

**UNIX V7** Jeder I-Node (2 Byte für die I-Node Nummer → max. 65536 Dateien) hat eine feste Position auf der Festplatte

**Linux** *Small is beautiful* → einfache Mechanismen und wenige Systemaufrufe (Verzeichnisbaum, Links, Virtuelles Filesystem / mounting)

**Prozess - Dateisystem proc** Enthält Informationen zum System und zu den laufenden Prozessen. Enthält *Pseudodateien* (existieren nicht wirklich auf der Festplatte) - für jeden laufenden Prozess existiert ein Verzeichnis. **Beim Lesen:** System ermittelt Informationen durch Systemaufrufe. **Beim Schreiben:** Änderung der Systemparameter

**ext2 - Dateisystem** Partition in Gruppen von Blöcken aufgeteilt. I-Nodes und zugehörige Datenblöcke in der selben Gruppe am besten nah beieinander auf der Festplatte halten. Blöcke werden im Voraus für eine Datei reserviert - Verzeichniseinträge nicht sortiert - System speichert die kürzlich benutzten Verzeichnisse im Cache

**Superblock** Informationen über das Dateisystem → Layout, Anzahl I-Nodes, Anzahl Festplattenblöcke, etc.

**Gruppendeskriptor** Position der *Bitmaps*, Anzahl freier Blöcke und I-Nodes, Anzahl der Verzeichnisse in der Gruppe

**Bitmaps** Verwaltung der freien I-Nodes und Blöcke. Jede Bitmap ist so groß wie ein Block

**Network File System (NFS)** Clients und Server nutzen Dateisystem gemeinsam. NFS-Server exportiert ein oder mehrere Verzeichnisse für den Zugriff von entfernten Clients - Jeder Rechner kann Client und Server sein

**NT Filesystem (NTFS)** Dateikompression, Journaling, alternative Datenströme, 64 Bit-Adressen für Blöcke → wichtigste Datenstruktur: *Masterdateitabelle (MFT)*

**Master File Table (MFT)** Einträge sind ein 1KiB groß und beschreiben Datein oder Verzeichnisse. Freie MFT werden mit *Bitmap* verwaltet. MFT ist eine Datei - Datenblöcke einer Datei werden in Form von Serien aufeinanderfolgender Blöcke gespeichert → Je mehr Fragmentierung, desto mehr MFT-Datensätze werden benötigt

**Datenkompression** Gruppen von 16 logischen Blöcken werden komprimiert gespeichert, wenn sie dann nur noch 15 oder weniger Blöcke belegen → Zugriff schwieriger, da evtl. zuerst dekomprimiert werden muss

**Alternative Datenströme** (Zusätzliche Datenattribute) Wird benutzt für: Schadcode oder getrennte Audio- und Videostreams in einer Datei

9.2 Geräteverwaltung

9.2.1 Grundlagen

Verwaltung von externen Geräten - durch Abstraktion → leicht zu bedienende Schnittstelle. **Aufgaben:** Kommandos an Geräte senden, Daten empfangen und weiterleiten, Interrupts behandeln, Fehlerbehandlung

**Geräte-Datei** Eine spezielle (*Pseudo*-) Datei unter Linux, die ein Gerät repräsentiert (z.B. Drucker). Durch Lese-/Schreiboperationen auf die Datei wird auf das entsprechende Gerät zugegriffen

**Blockorientierte Geräte** Speichern Daten in Blöcken fester Größe, welche einzeln adressierbar sind

**Zeichenorientierte Geräte** Erzeugen bzw. lesen Zeichenströme. Nicht adressierbar

**Controller** Gerät besteht aus 2 Komponenten. Mechanische und elektronische (Controller/Steueinheit) - ein Controller kann auch mehrere Geräte verwalten. **Aufgaben:** Seriellen Bit-Stream in Byte-Blöcke konvertieren, Fehlerkorrektur, Daten in den Arbeitsspeicher kopieren

**Controller-Schnittstelle** Schnittstelle welche für was jedes Gerät anders ist. Controller besitzt eine Menge von Kontrollregistern zur Ansteuerung. **Schreiben** in ein Register → Befehle erteilen, Aktionen auslösen. **Lesen** eines Registers → Statusinformationen. **Datenpuffer**, die lesen/geschrieben werden können. **Zugriff auf die Kontrollregister:** Der Zugriff kann auf zwei Arten erfolgen. Register liegen in einem eigenen, getrennten Adressraum  
Register werden in den Arbeitsspeicher eingeblendet → **Memory-Mapped I/O**

**Memory-Mapped I/O** Zugriff auf getrennten Bereich erfolgt über Assembler-Befehle. Die Kontrollregister liegen am Rand des Adressraumes. Jedes Gerät hat eine eindeutige Adresse

**Memory-Mapped I/O - Vorteile** Kann in C/C++ programmiert werden, einfach Zugriffskontrolle

**Memory-Mapped I/O - Nachteile** Caching muss für den Seitenrahmen ausgeschaltet werden - bei jedem Speicherzugriff muss unterschieden werden, ob es sich um ein Kontrollregister handelt oder nicht

**Direct Memory Access(DMA)** Ablauf ohne Direct Memory Access → Lesen von einer Festplatte. Plattencontroller liest einen Block bytewise von der Festplatte - speichert Block in seinen internen Puffer - Festplattencontroller berechnet Prüfsumme und führt ggf. Fehlerkorrektur durch. Plattencontroller erzeugt einen Interrupt. Betriebssystem kopiert den Pufferinhalt bytewise in den Arbeitsspeicher → viel Aufwand für das Betriebssystem  
Der DMA-Controller lohnt sich nur, wenn der Prozessor tatsächlich etwas anderes zu tun hat

9.2.2 Ein-/Ausgabe Software, Treiber

**Geräteunabhängigkeit** Programme können Geräte ansprechen ohne es genau zu kennen

**Benennungsschema** Name unabhängig von der Art des Gerätes

**Fehlerbehandlung** Sollte nah an der Hardware stattfinden

**Synchronisation** Synchron oder Asynchron

**Puffern** Daten können meist nicht direkt im Speicherziel abgelegt werden

**Nutzung** Exklusiv oder Gemeinsam

**Programmierte Ein-/Ausgabe** Das Betriebssystem übernimmt die komplette Übertragung zum Gerät → Prozessor ist während der gesamten Ein-/Ausgabe belegt

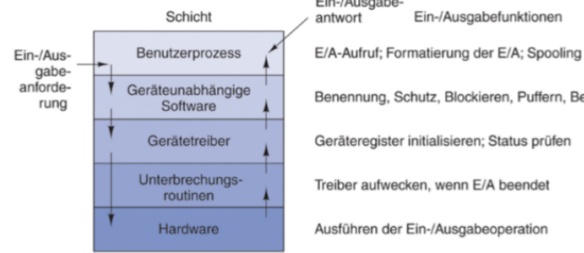


Abbildung 5.17: Schichten des Ein-/Ausgabesystems und die Hauptfunktion jeder Ebene

**Treiber** Ansteuern des Controllers ist die Aufgabe des Treibers (geräte-spezifischer Programmteil). Meist durch den Gerätehersteller in eigenen Modulen realisiert. **Aufgaben:** Initialisierung und Bekanntgabe des Gerätes, Datenübertragung von und zu einem Gerät, Logisches Programmiermodell auf gerätespezifische Anforderungen übersetzen, Pufferung von Daten, Interruptverarbeitung, Koordination der nebenläufigen Zugriffe. Treiber sind im Betriebssystem integriert → Treiber kann das gesamte System zum Absturz bringen

Integration von Treibern

**Statisch** Beim Compilieren des Kerns werden alle Treiber eingebunden

**Dynamisch** Treiber werden ins laufende System nachgeladen

**Schnittstellen für Treiber** Das Betriebssystem definiert eine Standardschnittstelle und - Architektur für die Installation von Treibern. Standardschnittstelle für blockorientierte Gerätetreiber. Standardschnittstelle für zeichenorientierte Gerätetreiber

10 Betriebssystemvirtualisierung

10.1 Grundlagen, Begriffe

**Definition Virtualisierung** Nachbildung eines Hard- oder Software Objekts durch ein ähnliches Objekt mit Hilfe einer Softwareschicht

**Virtuelle Maschine (VM)** Nachbildung eines kompletten Rechners durch einen ähnlichen oder ganz anderen Rechner (Softwareschicht: Hypervisor oder Virtual Machine Monitor)

**Vorteile von VMs** Mehrere Betriebssysteme auf dem Rechner - bessere Nutzung leistungsfähiger Hardware durch mehrere virtuelle Server auf nur einem realen Rechner - einfaches Kopieren eiber VM

**Nachteile von VMs** VMs müssen sich die real vorhandenen Ressourcen auf einem Rechner teilen - Ausfall des realen Rechners → Ausfall aller VMs. Virtualisierung (VMM) benötigt selbst Ressourcen

**Emulation** Jeder einzelne Maschinenbefehl des nachgebildeten Rechners wird durch Software ausgeführt → langsam, Geschwindigkeitsverlust Faktor 5 bis 10

**Virtualisierung** Die meisten Maschinenbefehle des nachgebildeten Rechners werden auf dem nachgebildeten Rechner nativ, also ohne Software-Eingriff direkt durch dessen Hardware ausgeführt

10.1.1 Virtual Machine Monitor (VMM)

**Aufgabe** Bereitstellen von isolierten Ablaufumgebungen in Form von Duplikaten der realen Hardware

**Implementierung** VMM fängt an bestimmte Maschinenbefehle ab und ersetzt diese durch andere, VM-spezifische Aktionen

10.2 Voraussetzungen

**Mindestvoraussetzung** Unterscheidung zwischen Kernel- und Anwendungsmodus des Prozessors. Vorhandensein einer MMU zur Abschottung von Adressräumen der VMs

**Problem mit den sensitiven Befehlen** Es gibt privilegierte und nicht privilegierte Befehle im Befehlssatz von Prozessoren. Privilegierte Befehle lösen Sprünge ins Betriebssystem aus. Sensitive Befehle sind zustandsverändernd und dürfen nur im Kernelmodus ausgeführt werden

**Virtualisierbarkeit nach Popek und Goldberg Theorem** Prozess ist virtualisierbar, wenn alle privilegierten Maschinenbefehle eine Unterbrechung erzeugen, wenn die in einem unprivilegierten Prozessormodus ausgeführt werden. Alle sensitiven Befehle sind auch privilegierte Befehle

**Kritische Befehle** Lösung: Dynamische Übersetzung kritischer Befehle zur Laufzeit durch direkte Aufrufe an den VMM. Code des Gast-Betriebssystems vor Abarbeitung in einen Puffer laden. Nach den kritischen Befehlen durchsuchen → durch direkte Aufrufe an den VMM ersetzen

10.2.1 Arten der Virtualisierung

**(VMM=) Typ-1-Hypervisor** Direkt über Hardware als kleines Minibetriebssystem - benötigt HW-Unterstützung für Virtualisierung im Prozessor. VMM **muss** alle Treiber zur Verfügung stellen

**Typ-2-Hypervisor** VMM läuft als Prozess unter einem Wirtsbetriebssystem → kann die Treiber des Wirtsbetriebssystems mitnutzen - Das Wirtsbetriebssystem ist auch direkt zugänglich

**Arten der Virtualisierung** Paravirtualisierung mit Modifikationen des Gast-Betriebssystems

**Übersetzungszeit** Ersetzung kritischer Maschinenbefehle im Gast-Betriebssystem durch direkte Aufrufe an den VMM

**Vorteil** Unabhängigkeit von den oben genannten Voraussetzungen, Durchsatzsteigerung

**Nachteil** Gast-Betriebssystem muss im Quellcode zugänglich sein

10.2.2 Speicher- und Gerätevirtualisierung

**Schattentabellen** Für jede VM eine Schattentabelle - übersetzt guestphysische Adresse in physische Adresse

**Geräteemulation** VMM betreibt das Gerät via eigenem Treiber oder Treiber des Wirtsbetriebssystems. VMM bietet dem Gastbetriebssystem eine Schnittstelle zu einem Standard-Gerät. Gastbetriebssystem kann seine eigenen Treiber verwenden

**Direktzuweisung eines Gerätes** Gerät wird einem Gastbetriebssystem exklusiv zugewiesen. Das Gastbetriebssystem interagiert via seinem eigenen Treiber direkt, also unter Umgebung des VMM, mit dem Gerät

11 Storage Systems

11.1 RAID Systeme

RAID: *Redundant Array of Inexpensive Disks* multiple Plattenspeicher - mehrere kleine Platten werden als eine große virtuelle Platte verwaltet. Für das Betriebssystem transparent

**RAID - 0** Stripes werden über mehrere Platten eines Arrays verteilt. Verteilung übernimmt das Betriebssystem oder eigener RAID-Controller → hoher I/O-Durchsatz, aber nicht ausfallsicher

**RAID - 1** wie RAID - 0, aber mit Redundanz. **Schreiben:** Jedes Stripe wird doppelt geschrieben. **Lesen:** Eine der beiden Kopien kann genutzt werden

**RAIN - 2** Platten laufen synchron - jedes Byte wird in 2 Halbbytes aufgeteilt. Zu jedem Halbbyte wird ein Hamming-Code hinzugefügt

**RAID - 3** 1-Bit-Fehlerkorrektur → Bei Ausfall einer Platte kann das fehlende Bit wiederhergestellt werden

**RAID - 4, RAID - 5** Stripe-Parität: XOR-Operation auf alle Stripes → bei Ausfall kann die fehlende Platte wiederhergestellt werden

**NAS - Network Attached Storage** Massenspeichereinheiten, die an ein lokales Netzwerk (LAN) angeschlossen sind

**SAN - Storage Area Network** Eigenes Netzwerk zwischen Servern und den Speicherressourcen. Speicher kann virtuell wie eine einzige Festplatte behandelt werden