

Minimieren der Verweilzeit → Shortest-Job-First

Die Anforderungen an einen Scheduler sind:

- **Fairness:** Da das Verfahren nicht präemptiv ist, können 3 sehr lange Prozesse dafür sorgen, daß alle anderen Prozesse sehr lange warten müssen. Ein einziger Prozess kann einen Prozessor beliebig lange blockieren.
- **Effizienz:** Wie man an den Lücken im Schedule sehen kann, sind nicht alle CPUs immer belegt, somit wird CPU-Zeit verschwendet.
- **Antwortzeit:** Es handelt sich um ein nicht-präemptives Scheduling. Fordert ein Prozess den Benutzer zu Eingaben auf, können diese sofort Verarbeitet werden, weil kein anderer Prozess in der Zwischenzeit den Prozessor belegt hat und erst verdrängt werden muss. Die Antwortzeit ist daher optimal.
- **Verweilzeit:** Die Verweilzeit der Prozesse ist nicht optimal (siehe nächste und übernächste Teilaufgabe).
- **Durchsatz:** Die 10 Prozesse benötigen insgesamt 11 Zeiteinheiten, bis sie komplett berechnet sind. Das lässt sich auf 10 Zeiteinheiten reduzieren (ohne Beweis).

Betriebssystem:

- Liefert eine abstrakte Schnittstelle zum Rechner
- Verwaltet und sichert Systemressourcen
- Verschiedene Varianten von Betriebssystemen existieren aufgrund verschiedener Anforderungen in unterschiedlichen Anwendungsgebieten

Ziele:

- Anpassung der Benutzerwelt an die Maschinenwelt
- Organisation und Koordination des Betriebsablaufs
- Steuerung und Protokollierung des Betriebsablaufs

Randbedingungen:

- ressourcenschonend
- robust
- sicher

Prozesse:

Ein Prozess ist ein Programm, dass sich in Ausführung befindet.

- Ein Prozess benötigt Betriebsmittel, das vom Betriebssystem verwaltet wird
- Prozesse sind gegeneinander abgeschottet, jeder besitzt (virtuell) seine eigenen Betriebsmittel wie etwa den Adressraum des Speichers.
- Prozesse konkurrieren um Rechenzeit □

Scheduling:

(Terminkalender für Prozesse)

Das Betriebssystem muss entscheiden, welcher Prozess wann wieviel Rechenzeit erhält □

- Ein Prozessorkern führt in jeder Zeiteinheit maximal **einen** Prozess aus

Multitasking:

Kooperatives Multitasking:

- Jedem Prozess ist es selbst überlassen, wann er die Kontrolle über den Prozess an den Kern wieder zurück gibt.
- **Vorteil:** Einfacheres Betriebssystem
- **Nachteil:** Auf gutmütige (kooperative) Programme angewiesen

Präemptives Multitasking:

- Neuzuteilung des Prozessors kann in regelmäßigen Zeitintervallen vom Betriebssystem erzwungen werden
- **Vorteil:** Steuerung durch das Betriebssystem
- **Nachteil:** Komplexeres Betriebssystem

Prozess-Scheduling

Anforderungen:

- **Fairness:** Jeder Prozess soll einen fairen Anteil an der CPU bekommen
- **Effizienz:** Die CPU sollte möglichst immer belegt sein
- **Antwortzeit:** Die Antwortzeit für interaktive Benutzer soll minimal sein
- **Verweilzeit:** Die Verweildauer von Programmen soll möglichst gering sein
- **Durchsatz:** Die Anzahl bearbeiteter Prozesse pro Zeitintervall soll maximiert werden

Deterministisches Scheduling: □

- Die Ausführungszeiten der Prozesse sind bekannt. □ Die Prozesse werden zeitlich so angeordnet, dass sich ein gewünschtes Systemverhalten ergibt (z.B. minimale Durchlaufzeit).

Probabilistisches Scheduling: □

- Lediglich die Erwartungswerte bzw. die Wahrscheinlichkeitsverteilung der Ankunfts- oder Bedienungszeiten sind bekannt. □
- Das Verhalten des jeweiligen Scheduling-Algorithmus wird unter der wahrscheinlichsten Last beschrieben. □

Scheduling verfahren:

Ursprung in Batch-Systemen

- First Come, First Served (**FCFS**)
- Shortest Job First (**SJF**)
- Shortest Remaining Time Next (**SRTN, SRT**) □

Interaktiv (heutige Desktopsysteme)

- Shortest Process Next (**SPN**)
- Round-Robin Scheduling (**RR**)
- Priority Scheduling (**PS**)
- Feedback und Multilevel Feedback Queues (**MLFQ**)

Scheduling-Verfahrensarten:

- **FCFS**: Einfach zu implementieren, aber u.U. längere Wartezeit
- **SJF**: Längere Prozesse verhungern, und vorhersage von Prozesslaufzeit ist fast nicht möglich
- **SRTN**: Ist die **präemptive** Version von SJF -> Re-Scheduling nur, wenn ein neuer Prozess eintrifft. Ist er kürzer als der derzeit laufende, so verdrängt er den aktuellen Prozess. **Nachteile** wie bei SJF.
- **ROUND ROBIN** (**wichtig!!**):
- Prozesse werden in einer Warteschlange eingereiht und in FIFO-Ordnung ausgewählt (wie FCFS-Scheduling).
 - Grundlegend präemptives Scheduling-Verfahren - Ein rechnender Prozess wird vom Betriebssystem nach dem Ablauf einer Zeitscheibe (time slice, quantum) unterbrochen und wieder hinten in die Warteschlange eingefügt (round robin).

Vorteile:

- Jeder Prozess bekommt Zeit zum rechnen -> kurze Antwortzeit!
- Die Prozessorzeit wird nahezu gleichmäßig auf die vorhandenen Prozesse aufgeteilt.
- Prozesse mit Ein-/Ausgabe geben die CPU ab, bevor ihre Zeitscheibe abgelaufen ist -> gut für Prozesse ohne IO!

- **Variation von RR mit mehreren Warteschlangen mit verschiedenen Prioritäten:**
 - Einreihung nach Priorität in Warteschlangen -> Prozessor arbeitet Warteschlange von hoher prio nach niedriger prio ab
 - **Vorteil:** Wichtige Prozesse bekommen genug rechenzeit (z.B. Audiowiedergabe)
- **Feedback-Sceduling:** Bezeichnet eine Scheduling Art, bei der die Priorität des Prozesses zur Laufzeit **erlernt** und verändert wird.
 - Beendet ein Prozess seine Zeitscheibe vollständig, wird er in die Warteschlange mit **nächstniedriger Priorität** eingereiht (**aging**).
 - Gibt der Prozess seine Zeitscheibe freiwillig auf (indem er **auf Ein-/ Ausgabe wartet**), so **behält er seine Priorität**.
 - **Vorteile:** Prozesse mit **viel IO** (wartezeit) haben **hohe prio** und **reagieren schnell** (shell, textverarbeitung)
- **Verbesserung:** Damit Hintergrundprozesse nicht aussterben: Erhöhe periodisch die Priorität aller Prozesse auf die höchste Priorität (**priority boost**)

Name des o.g. verfahrens: **Multilevel Feedback Queueing**, in Windoof genutzt.

Nebenläufigkeit & Interprozesskommunikation

Prozesse vs Threads

Elemente pro Prozess

- Adressraum
- Globale Variables
- Offene Dateien
- Kindprozesse
- Zu behandelnde Signale
- Signale und Signalroutinen
- Accountinginformationen

Elemente pro Thread

- Programmzähler
- Register
- Stack
- Zustand

Vorteile von Threads gegenüber Prozessen:

- Modellierung von Programmen mit mehreren gleichzeitigen Aktivitäten
Von denen einige von Zeit zu Zeit blockieren
- Ergebnis: **einfachere Programmierung**
- **Bessere Performance** bei Erzeugung und Zerstörung, da wenig Ressourcen mit Threads verbunden sind
- **Bessere Performance** bei Mischung von CPU- mit I/O-intensiven Aufgaben
- Vorteil bei reiner CPU-Nutzung mit Hyperthreading Prozessoren (**ein Prozessor kann 2 threads pseudo-parallel ausführen**, von der Hardware realisiert!)
- Kontextwechsel zwischen Threads **effizienter** als zwischen Prozessen
- **kein Wechsel des Adressraums**

Thread Typen:

Kernel Threads /Light-Weight Processes (LWP)

- Thread-Management komplett durch den Kernel
- Scheduling durch BS
- Kernel hält Kontext- Informationen über Prozesse und Threads (Stackverwaltung)

User-Space-Threads

- Scheduling durch Anwendung
- BS nur für Prozess-Scheduling verantwortlich
- Anwendung benutzt Thread- Bibliothek
- Thread-Bibliothek sorgt für Thread-Management

Vergleich:

Kernel-Level Threads

Vorteile:

- Mehrere Threads von einem Prozess können auf mehrere Prozessoren verteilt werden
- Wenn ein Thread blockiert, kann ein anderer Thread des gleichen Prozesses ausgeführt werden
- Kernel-Funktionen können multi-threaded implementiert werden

Nachteile:

- Thread-Wechsel erfordern Wechsel in Kernel-Mode (syscall)

User-Level Threads:

Vorteile:

- Thread-Wechsel benötigen keine Kernel-Mode- Funktionen
- Scheduling auf die Anwendung zugeschnitten (da von Anwendung realisiert!)
- Unterstützung für alle BS (keine Abhängigkeit)
- Schneller

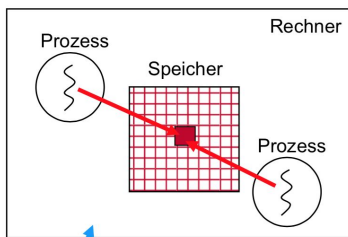
Nachteile:

- Wenn ein Thread blockiert, blockiert der ganze Prozess
- Kein Ausnutzen von Mehrprozessorsystemen

Interprozess-Kommunikation

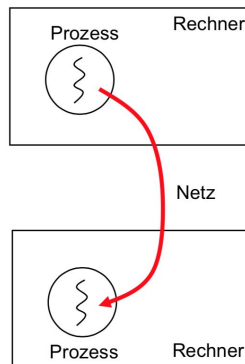
- **Prozesse und besonders Threads arbeiten nicht allein, sondern müssen Informationen austauschen**

Gemeinsame Variablen: vor allem in Ein-Prozessor und Multiprozessor-Systemen mit gemeinsamem physikalischen Speicher



Bei Threads sehr einfach!

■ **Nachrichtenaustausch:** vor allem bei verteilten Systemen, also Kommunikation über Rechengrenzen hinweg



Quelle: Bet
WS0910,
Prof. Dr. Be
TU Braunschweig

- **Problem:** Wie sicherstellen, dass nicht gleichzeitig auf gemeinsam genutzte Daten zugegriffen wird? Wie Reihenfolge sicherstellen?

-> **Synchronisationsproblem.**

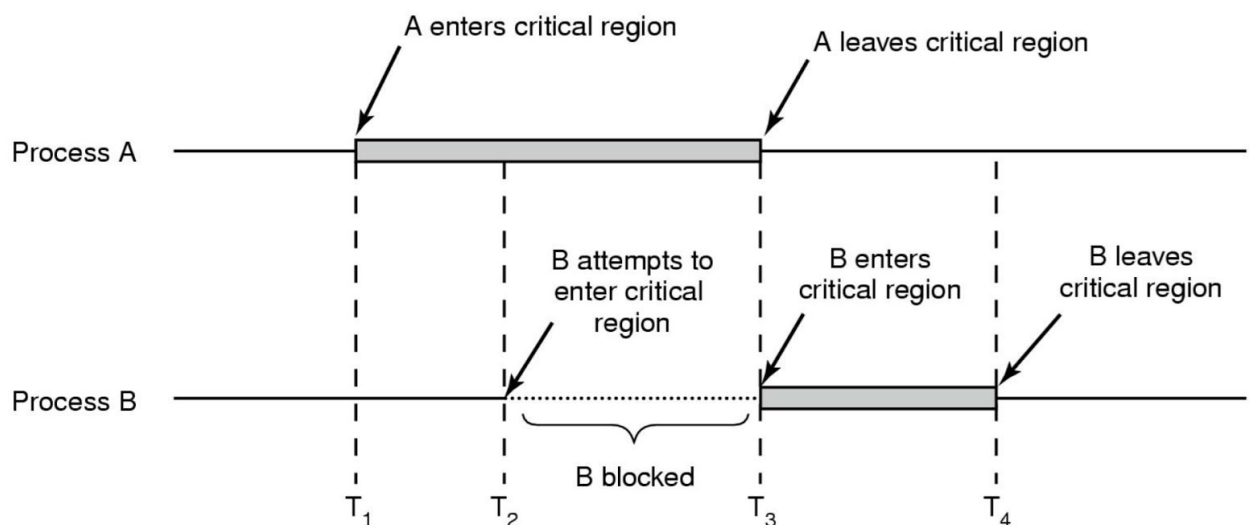
Vokabular:

- **Race Condition:**

Zwei Prozesse wollen zur gleichen Zeit denselben Speicher zugreifen - ergebniss hängt davon ab wer schneller ist

- **Kritischer Abschnitt (Critical Section):**

Menge von Instruktionen, in der das Ergebnis der Ausführung auf unvorhergesehene Weise variieren kann, wenn Variablen, die auch für andere parallel ablaufende Prozesse oder Threads zugreifbar sind, während der Ausführung verändert werden.



- #todo

Benötigt: Wechselseitiger Ausschluss von kritischen Abschnitten.

Anforderungen:

- Wechselseitiger Ausschluss von kritischen Abschnitten.
- Zwei Prozesse dürfen nicht gleichzeitig in ihrem kritischen Abschnitt sein (**safety**).
- Es dürfen keine Annahmen über die Bearbeitungsgeschwindigkeit von Prozessen gemacht werden.
- Kein Prozess, der außerhalb eines kritischen Bereichs ist, darf andere Prozesse beim Eintritt in den kritischen Abschnitt behindern.
- Kein Prozess darf ewig auf den Eintritt in den kritischen Abschnitt warten müssen (**fairness**).

- Möglichst **passives** statt **aktives** Warten, da aktives Warten einerseits Rechenzeit verschwendet und andererseits Blockierungen auftreten können, wenn auf Prozesse/Threads mit niedriger Priorität gewartet werden muss.

Praktische Realisierungen:

- Unterbrechungen durch den Scheduler ausschalten.
 - **Unattraktiv für PC**, Server-Betriebssysteme, **da Kontrolle dann beim Benutzerprozess**
 - **Schützvariable**
Aber: **Wer schützt den Schützer?**
- **TSL-Hardwarebefehl**
 - Ein **Test-and-Set-Lock** Hardwarebefehl wird vom Prozessor unterstützt
 - **Verhindere, dass der Prozess unterbrochen wird, nachdem** er seine **Variable gesetzt hat** und bevor er die des anderen Prozesses abfragen kann.
 - Maschineninstruktion **TSL (Test-and-Set-Lock)** führt genau diese Operation **in einem einzigen CPU-Befehl** durch.
 - **TSL register, addr** schreibt im Speicher an die Stelle **addr** eine **1** und gibt den vorherigen Inhalt der Speicherzelle in Register zurück.
 - Benötigt nur noch eine Variable für alle Prozesse, weil die eigene Schreiboperation das Ergebnis nicht beeinflusst
- **Petersons Lösung**
 - Benutze zusätzlich zur Variable einen **Interesse-Vektor** mit **einem Element pro Prozess**

```
// process0.py
while True:
    interested[0] = True
    while interested[1]:
        pass
    critical_section()
    interested[0] = False
    noncritical_section()
```

```
// process1.py
while True:
    interested [1] = True
    while interested[0]:
        pass
    critical_section()
    interested[1] = False
    noncritical_section()
```

Problem bei Variable und Hardwarebefehl:

Benötigt aktives Warten (busy waiting, polling), bis variable 0 ist.

→ Verschwendung von CPU Zeit.

Lösung:

Integriere **Mechanismen für wechselseitigen Ausschluss** direkt **ins Betriebssystem** - Prozesse können dann blockierend warten (genauso wie sie auf I/O warten).

→ **Semaphor** - **Mutex** - **Monitor**

Semaphor:

Ist eine **Integer- Variable**, auf die nur die unteilbaren (atomaren) Operationen **up** und **down** ausgeführt werden können

- **down()**:
Verringert Wert des Semaphores um 1. Ist dann der Wert < 0 , wird der aufrufende Prozess blockiert
- **up()**:
Erhöht den Wert des Semaphores um 1 Ist dann der Wert ≤ 0 , wird einer der wartenden Prozesse aufgeweckt

Oft wird die Fähigkeit zu zählen bei Semaphoren nicht benötigt, es genügt eine einfache binäre Aussage, ob ein kritischer Abschnitt frei ist oder nicht.->

Lösung: Mutex!!

Operationen:

- **lock()**: **blockiert** den Zugriff auf eine Ressource (wenn schon gelockt, wartet bis unlock())
- **unlock()**: **gibt den Zugriff wieder frei**

Monitor:

Monitore schützen ihre Datenstrukturen vor fehlerhaftem, unstrukturierten Zugriff. Es ist die Aufgabe des **Compilers**, Maschinenbefehle zu generieren, die den **wechselseitigen Ausschluss** im Monitor **garantieren**.

Mutex-Monitor:

```
class Account(object):
    def __init__(self):
        # ...
        self.lock = threading.Lock()



    def withdraw(amount):
        with lock:
            balance = balance - amount
            return balance
```

Deadlocks

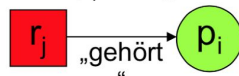
Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge auslösen kann.

Ressourcenbelegungen und Ressourcenanforderungen können durch einen gerichteten Belegungs-Anforderungs-Graphen dargestellt werden.

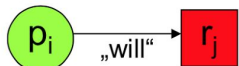
Es gibt 2 Arten von Knoten:

- Kreise repräsentieren Prozesse p_i : 
- Quadrate repräsentieren Ressourcen r_j : 

Eine Kante von einer Ressource r_j zu einem Prozess p_i bedeutet: Ressource r_j wird von Prozess p_i belegt.



Eine Kante von einem Prozess p_i zu einer Ressource r_j bedeutet: Prozess p_i hat Ressource r_j angefordert, aber noch nicht erhalten.



Zyklen im Belegungs-Anforderungsgraphen repräsentieren Deadlocks!

Folgende Voraussetzungen müssen erfüllt sein, damit ein Deadlock auftreten kann:

- **Wechselseitiger Ausschluss:**
Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.
- **Hold-and-wait-Bedingung:**
Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern.
- **Ununterbrechbarkeit:**

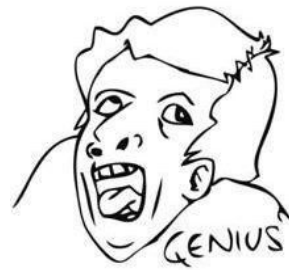
Ressourcen, die einem Prozess bewilligt wurden, können nicht gewaltsam wieder entzogen werden.

- **Zyklische Wartebedingung:**

Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört.

In Betriebssystemen diese Bedingungen üblicherweise erfüllt.

Grundsätzliche Strategien:



- Das Problem ignorieren

- **Deadlocks entdecken und auflösen nachdem sie eingetreten sind.** Z.B. durch Zurücksetzen (rollback) oder zwangsweises Beenden von Prozessen -> problematisch, erfordert Daten
- **Deadlocks vermeiden durch entsprechende Ressourcen- Allokation** (mit **Bankier** vorher checken)
- **Deadlocks verhindern, indem man nie alle Deadlock- Bedingungen erfüllt** (schwierig)

Grundidee des Bankieralgorithmus:

- Versuche möglichst viel „Pseudo-Parallelismus“ zu erreichen (keine streng sequentielle Abarbeitung der Prozesse)
- Riskiere dabei aber an keinem Punkt eine potentielle Deadlock-Situation!

Der Bankieralgorithmus:

- **Prüfe bei jeder Ressourcenanforderung** eines Prozesses p_i , ob da System bei Erfüllung der Anforderung in einen unsicheren Zustand kommt.
- **Falls ja:**
 - stelle Prozess p_i zurück und mache mit einem anderen Prozess weiter.
- **Andernfalls:**
 - Weise Ressource den Prozess p_i zu

Wie wird auf sicheren Zustand geprüft?

- Der Bankieralgorithmus überführt immer nur in sichere Zustände
 - Dadurch garantiert deadlockfreie Ausführung

Beispiel:

- Es gibt $V = 10$ Instanzen einer Ressource.

- 3 Prozesse p_1, p_2, p_3

- Maximale Anforderungen der Prozesse M_i :

	M_i
p_1	9
p_2	4
p_3	7

- Zustand zum Zeitpunkt t:

E_i : bereits erhaltene Ressourcen

A_i : noch anzufordernde Ressourcen

F: freie Ressourcen

	E_i	$A_i = M_i - E_i$	
p_1	3	$6 = 9 - 3$	
p_2	2	$2 = 4 - 2$	$F = 10 - 7 = 3$
p_3	2	$5 = 7 - 2$	
	7		

- Ist dies ein sicherer Zustand?

BURG

Um das zu prüfen, Algo zu ende ausführen. Wenn geht -> sicher. Beispiel für unsicher:

The diagram illustrates the execution of process p_2 and the resulting state of the semaphore. It consists of three tables representing the semaphore state at different points in time, connected by arrows indicating the progression of the process.

Initial State:

	E_i	A_i	M_i
p_1	4	5	9
p_2	2	2	4
p_3	2	5	7

$F = 10 - 8 = 2$

Process p_2 Execution: Führe Prozess p_2 bis zum Ende aus

State after p_2 execution:

	E_i	A_i	M_i
p_1	4	5	9
p_2	4	0	4
p_3	2	5	7

$F = 10 - 10 = 0$

Process p_2 Release: Freigabe durch Prozess p_2

Final State:

	E_i	A_i	M_i
p_1	4	5	9
p_2	0	-	-
p_3	2	5	7

$F = 10 - 6 = 4$

Mit den jetzt zur Verfügung stehenden 4 freien Ressourcen lassen sich weder Prozess p1 noch Prozess p3 ausführen, wenn sie ihre Ressourcenanforderungen sofort stellen und vor Prozessbeendigung nichts freigeben.

	Bandlaufwerke	Plotter	Scanner	CD-ROM	
V =	(4	2	3	1)	
M:	(2	0	1	1)	Prozess 1
	(3	0	1	1)	Prozess 2
	(2	2	2	0)	Prozess 3
E:	(0	0	1	0)	Prozess 1
	(2	0	0	1)	Prozess 2
	(0	1	2	0)	Prozess 3
A:	(2	0	0	1)	Prozess 1
	(1	0	1	0)	Prozess 2
	(2	1	0	0)	Prozess 3
F =	(2	1	0	0)	

Ist mit dem Bankieralgorithmus das Deadlock-Problem restlos gelöst? Leider **nein**, da - Prozesse können meist **nicht im Voraus eine verlässliche Obergrenze** für ihre Ressourcenanforderungen geben (zumindest nicht exakt genug, so dass das System nicht durch Überschätzung des Ressourcenbedarfs zu ineffizienter Ausführung gezwungen wird). - Garantierte Obergrenzen würden häufig sogar die Anzahl der verfügbaren Ressourcen übersteigen, aber durch ständiges Zuweisen und Freigeben von Ressourcen stellt dies trotzdem kein Problem dar.

Üblicherweise werden Prozesse dynamisch neu erzeugt werden und sind nicht statisch vorhanden.

Ressourcen können auch plötzlich verschwinden (z.B. Bandlaufwerke fallen aus).

- **Deadlock-Verhinderung ist schwierig**, da dazu im Allgemeinen Informationen über zukünftige Ressourcen-Anforderungen nötig sind, die nicht bekannt sind!
- **Deadlock-Freiheit kann zwar prinzipiell erreicht werden**, häufig jedoch um den Preis **starker Effizienz-Verluste**.
- Häufig verzichtet man in der Praxis daher auf absolute Garantien für Deadlock-Freiheit.
- Warum geht bei solchen Systemen trotzdem meistens nichts schief?
 - **Weil sie nicht an ihrem Ressourcen-Limit betrieben werden**. Kritische (besonders knappe) Ressourcen werden auf unkritische Ressourcen abgebildet

Speicherverwaltung

- Speicher ist gemeinsam genutzte Ressource
 - Mehrere Prozesse gleichzeitig im Speicher
 - echter **gemeinsamer Zugriff** □ erfordert Speicherverwaltung
 - Aufteilung des verfügbaren Hauptspeichers
 - zwischen verschiedenen Prozessen
 - und dem Betriebssystem □
- **Aufgaben:**
- **Buchhaltung**
Vergabe und Rücknahme von Speicher

Konzepte zur Speicherverwaltung:

Partitionierung

Segmentierung

Paging

Partionen / Partitionierung:

- **Aufteilung des Speichers in Bereiche mit festen Grenzen**
- Fester, zusammenhängender Teil des Hauptspeichers für Betriebssystem
- Pro Prozess ein zusammenhängender Teil des Speichers
- Verschiedene Varianten:
 - **Statische Partitionierung:**
 - **Dynamische Partitionierung**
 - **Buddy-Verfahren**

Statische Partitionierung: □

Einteilung des Speichers in feste Anzahl von Partitionen,
pro Prozess eine Partition.

Probleme:

- Programm zu groß für Partition
- **Interne Fragmentierung:** Platzverschwendung, wenn Programm kleiner als Größe der zugeordneten Partition -> **In der Partition wird platz verschwendet**
- Fest vorgegebene Anzahl von Prozessen im Speicher (wegen fester Anzahl der Partitionen)

Dynamische Partitionierung:

- Einteilung des Speichers in Partitionen variabler Länge und variabler Anzahl

Probleme:

Ein- und Auslagern führt zu **externer Fragmentierung!**

BS, 8MB	BS, 8MB	BS, 8MB	BS, 8MB
56 MB	Prozess 1 20 MB	Prozess 1 20 MB	Prozess 1 20 MB
	36 MB	Prozess 2 14MB	Prozess 2 14MB
		22 MB	Prozess 3 18 MB
			4 MB

BS, 8MB	BS, 8MB	BS, 8MB	BS, 8MB
Prozess 1 20 MB	Prozess 1 20 MB	20 MB	Prozess 2 14 MB
14 MB	P. 4, 8MB	P. 4, 8 MB	6 MB
Prozess 3 18 MB	6 MB	6 MB	P. 4, 8MB
4 MB	Prozess 3 18 MB	Prozess 3 18 MB	6 MB
	4 MB	4 MB	Prozess 3 18 MB
			4 MB

Um beste Partition für Prozess zu finden, gibt es verschiedene Ansätze:

Best Fit: am schlechtesten!

- Suche **kleinsten** Block, der ausreicht.

First Fit: Das Beste!

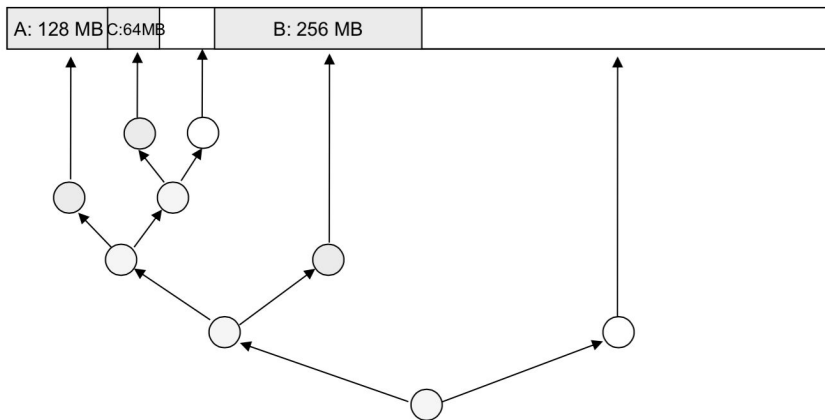
- Suche beginnend mit Speicheranfang bis ausreichender Block gefunden.
- Probleme/Nachteile: **Schnelle Fragmentierung des größten freien Speicherblocks am Ende des Speichers**

Next Fit:

- Suche beginnend mit der Stelle der letzten Blockzuweisung
 - Probleme/Nachteile: Am schlechtesten! Produziert schnell eine Reihe von sehr kleinen Fragmenten, die ohne Defragmentierung nie mehr benutzt werden können.
- langames Verfahren**

Buddy-System: (Halbierungsverfahren):

- Kompromiss zwischen statischer und dynamischer Partitionierung
- Anzahl nicht-ausgelagerter Prozesse dynamisch
- Interne Fragmentierung beschränkt
- Keine explizite Defragmentierung



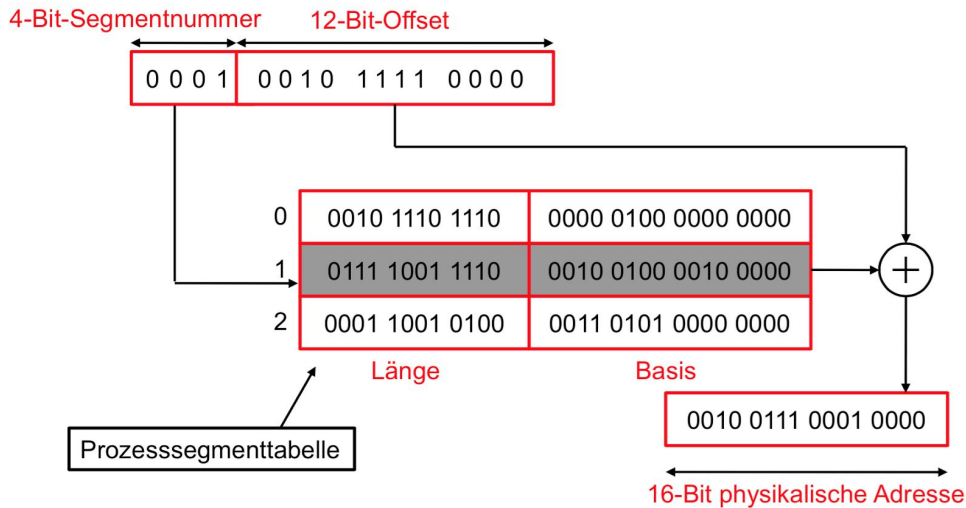
Generelle Probleme bei Multiprogramming:

- Viele Programme gleichzeitig im Speicher
- Ein Programm kann nicht immer an der gleichen Stelle landen
- Speicherreferenzen innerhalb des Programms müssen verschoben werden
- Absolute Sprungbefehle
- Datenzugriffsbefehle
- **Relokation!**
- Übersetzung der Speicherreferenzen im Programmcode in tatsächliche physikalische Speicheradressen durch Prozessorhardware oder/und Betriebssystemsoftware

Lösung?

Segmentierung:

Stelle jedem Programm **mehrere virtuelle Adressräume** zur Verfügung, die es für die jeweiligen Daten verwenden kann. Eine logische Adresse besteht dann aus Segment-Nummer und Adresse

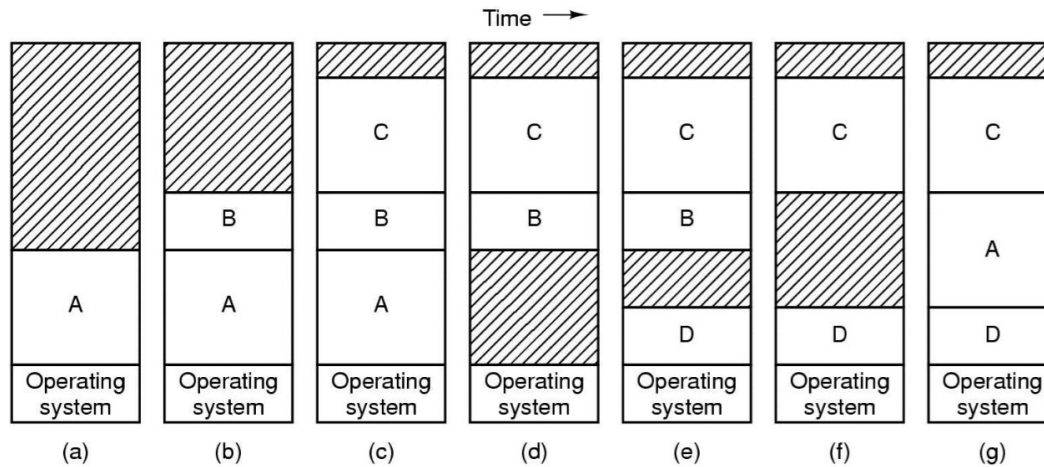


Vorteile der Segmentierung:

- **Relokation einfach** → Nur die Segmenttabelle muss angepasst werden
- **Segmente können wachsen, Management durch das BS** → Nachfolgende Segmente ändern ihre Basisadresse, wenn sie verschoben werden -Linken wird wegen der jeweiligen Startadresse innerhalb jedes Segments stark vereinfacht • Jede interne Tabelle beginnt direkt bei 0
- **Direkte Unterstützung durch CPU (MMU: Memory Management Unit)** → Arbeitet direkt mit logischen Adressen und schaut von alleine in Segmenttabellen nach
-

Probleme:

- Ein Segment muss immer noch vollständig im Hauptspeicher sein und ist dort **linear** abgelegt → Im Speicher muss ein entsprechend großer Bereich frei sein □
- Durch das Entfernen und Einfügen von Segmenten entstehen langfristig **kleine unbenutzte Speicherbereiche** (externe Fragmentierung).



Lösung? **Paging!**

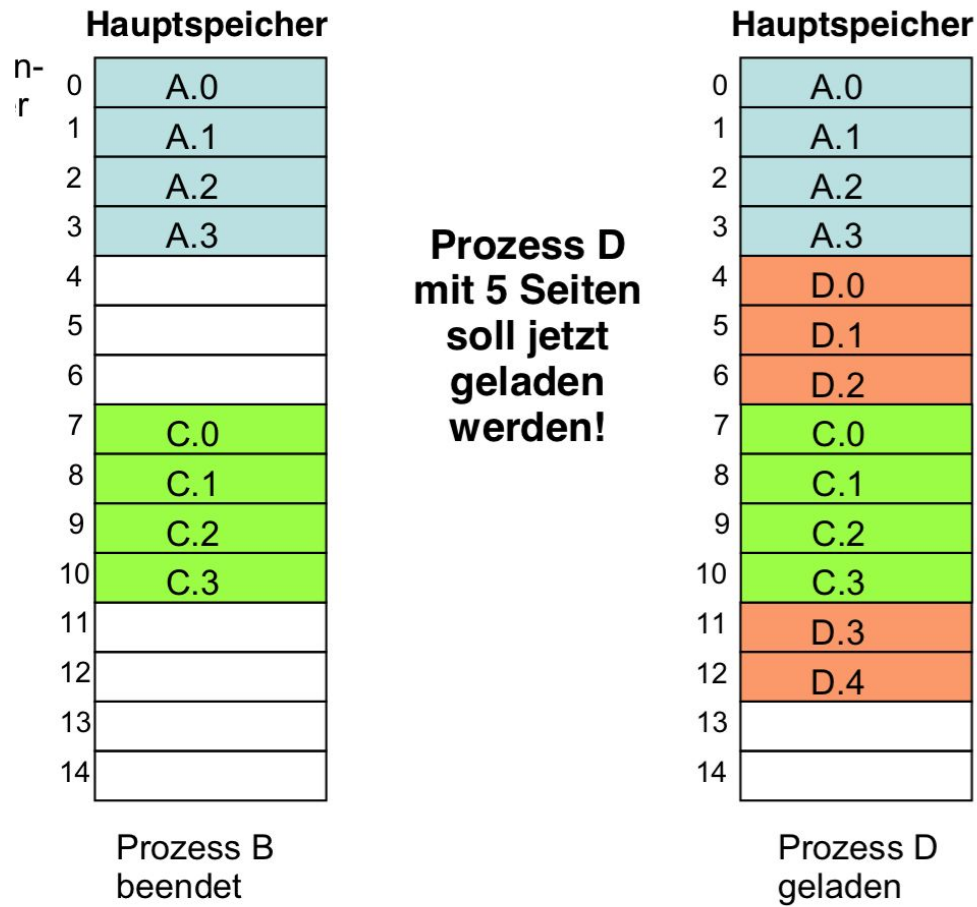
Probleme bei Partitionierung und Segmentierung

- Programme (oder Segmente) müssen zusammenhängend im Speicher sein
- Fragmentierung □

Paging:

- Im Gegensatz zu Partitionierung werden Prozessen **nicht** notwendigerweise **zusammenhängende** physikalische Speicherbereiche zugeordnet.
- **Hauptspeicher** aufgeteilt in **gleichgroße Seitenrahmen**
- **Logischer Adressraum** eines Prozesses aufgeteilt in **gleichgroße Seiten** (pages)
- **Umsetzungstabelle**, die **Seitentabelle** (page table) bildet die Seiten auf die verfügbaren Kacheln ab.
 - **Pro Prozess eine Seitentabelle**
 - Die Seiten eines Adressraums können beliebig auf die verfügbaren Kacheln verteilt sein.

Beispiel:

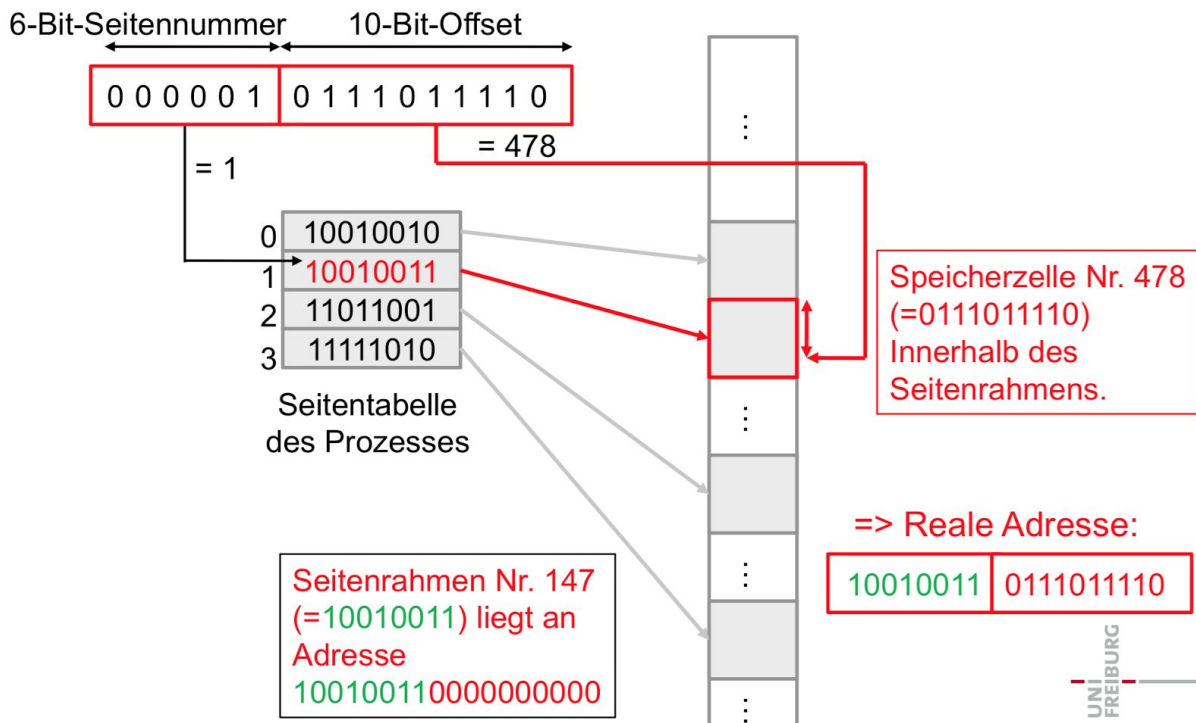




Besonders einfach, wenn Rahmengröße = 2^k

- Basisadresse eines Rahmens ist dann Rahmengröße \ll Rahmennummer (Linksshift)
- Offset von 0 bis $2^{\text{Rahmengröße}} - 1$

IRG



Seitengröße?

Die Seitengröße wird vom BS festgelegt.

Vorteile kleiner Seiten:

- Im Durchschnitt ist die letzte Seite zur Hälfte leer (interne Fragmentierung).
- Große Seiten führen zu Verschwendung von Hauptspeicher. □

Nachteile kleiner Seiten:

- Programme brauchen viele Seiten.
- Seitentabellen werden größer.

Segmentierung VS Paging:

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Vorteile Paging:

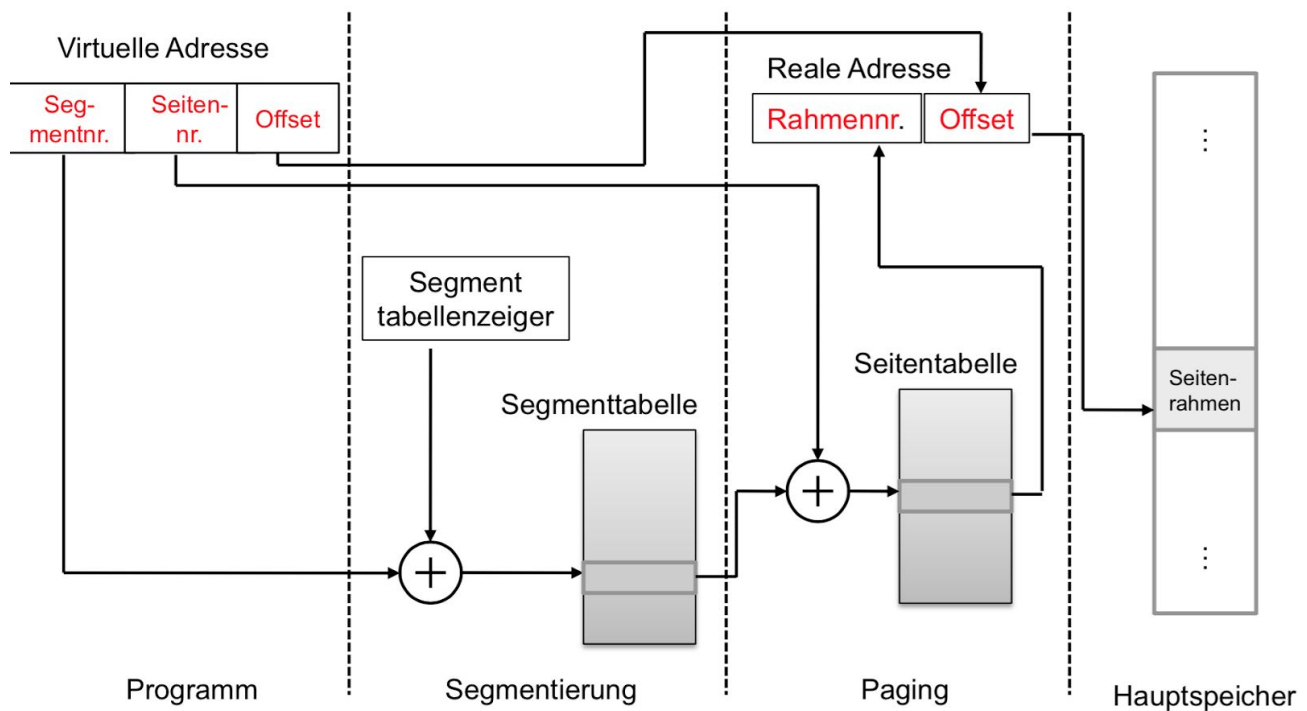
Für den Nutzer transparent → Als gehöre ihm der Speicher alleine

- Feste Seitengrößen → Vereinfacht Speicherverwaltung
- Keine externe Fragmentierung! □

Vorteile Segmentierung:

- Anpassung an dynamischen Speicherbedarf von Prozessen

→ Daher **Kombination** von Segmentierung mit Paging



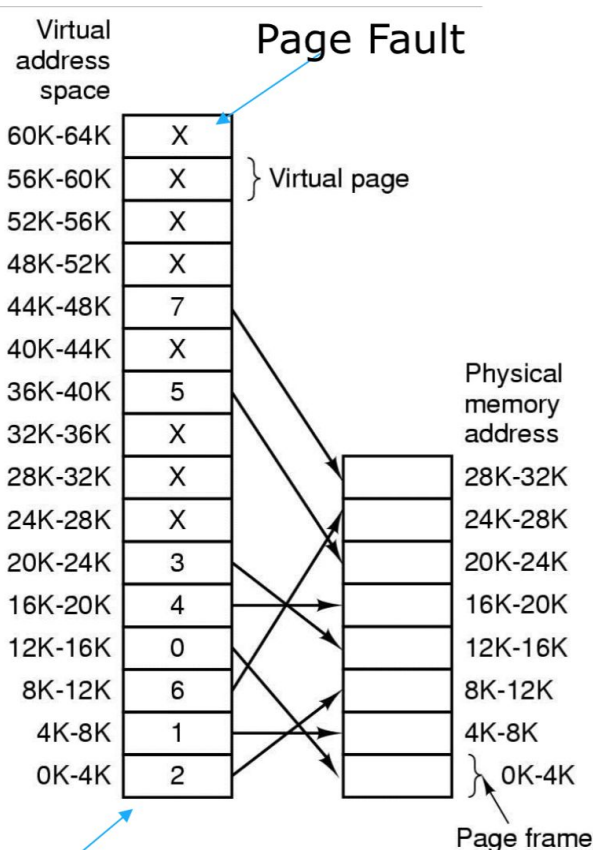
Virtueller Speicher

In Multiprogramming-Systemen werden Prozesse ohnehin **pseudo-parallel** ausgeführt
→ **Prozesse, die gerade nicht rechnen, müssen eigentlich auch nicht im Speicher sein!**

- Das Betriebssystem schreibt den kompletten Speicherinhalt eines Prozesses auf die Festplatte (Auslagern, swapping) □
- Wird ein Prozess wieder aktiviert, wird er vom Betriebssystem eingelagert
- Die Reihenfolge beim Zurückschreiben der Seiten ist egal
- Das Betriebssystem aktualisiert entsprechend die Seitentabelle, alle Adresse im Programm bleiben erhalten

Verbesserung:

- Lagere **nicht ganze Prozesse** aus, **sondern** immer jeweils **einzelne Seiten!**
- Wird auf Informationen zugegriffen, die ausgelagert wurden, so muss das Betriebssystem diese nachladen □
- Die Seitentabelle gibt zu jeder Seite den entsprechenden Rahmen an
- Ein spezielles Bit (present bit) gibt an, ob die Seite überhaupt im Page Fault Speicher ist



Bei Zugriff auf eine Seite, die sich nicht im Hauptspeicher befindet

- Hardware (MMU) stellt anhand des present bits fest, daß angefragte Seite nicht im Hauptspeicher ist (=> „Seitenfehler“ bzw. „page fault“).

- Auslösen einer Unterbrechung („Interrupt“) durch die Hardware

- Behandlung des Interrupts:
 - Sichern des aktuellen Programmzustandes durch Hardware (Stand des Programmzählers)

- Routine zur Interruptbehandlung wird aufgerufen:
- 1. Ermittelt die Adresse der fehlenden Seite
- Finde einen freien Seitenrahmen im Hauptspeicher
- Wenn kein Seitenrahmen frei ist, wird zunächst eine andere Seite aus dem Speicher ausgelagert
- Die fehlende Seite wird nun von der Festplatte in den freien Seitenrahmen kopiert
- Die Rahmennummer wird in die Seitentabelle des Prozesses eingetragen
-

→ Programmzustand wird wiederhergestellt und der Prozess wird fortgesetzt

Vorteile von Paging mit virtuellem Speicher:

- Mehr aktive Prozesse im System (**Pseudoparallelismus**)
- Adressraum eines Prozesses kann jetzt größer sein als verfügbarer Hauptspeicher.

Nachteil:

- Bei Zugriff auf Code/Daten, die nicht im Hauptspeicher vorhanden sind, muss das Betriebssystem die entsprechenden Seiten nachladen. Dabei müssen evtl. andere Seiten ausgelagert werden, um Platz zu schaffen.

Mehrfaches Paging:

Seitentabelle muss groß genug sein für alle Einträge die ein Prozess haben kann.

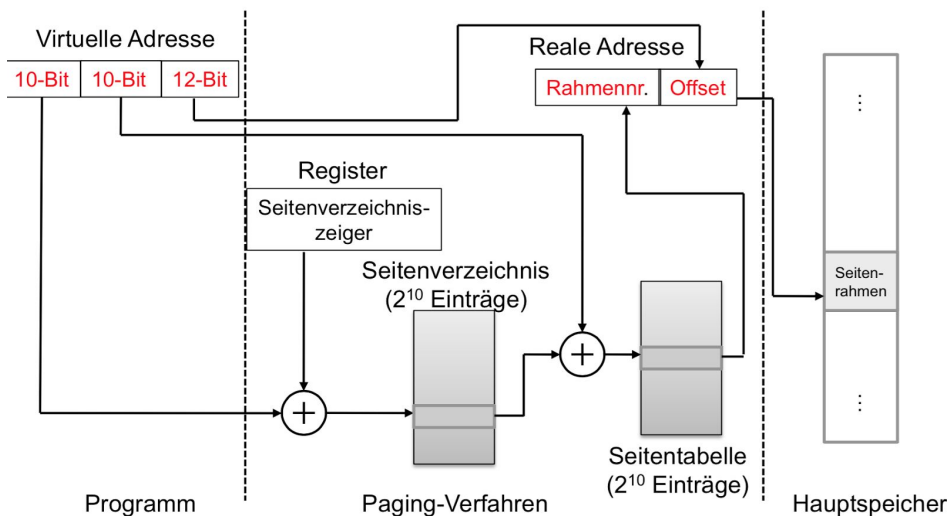
→ bei 32 Bit Adressraum sind das 4MB pro Prozess! bei 64 Bit noch schlimmer..

Lösung: Hierarchische Tabellen

- Wende den gleichen Mechanismus wie beim Paging auf die Seitentabelle selbst an

IA32:

- Die 4 MB große Seitentabelle wird unterteilt in 1024 Einzeltabellen von 4 kB Größe (wie alle anderen Seiten auch)
- Für diese 1024 Seiten wird wiederum eine Tabelle benötigt, das Seitentabellenverzeichnis (kurz: Seitenverzeichnis, page directory (table))
- Dieses Verzeichnis liegt immer im Speicher
 - Diesmal nur $1024 \cdot 4\text{Byte} = 4\text{kB}$ groß! □
- Die logische Adresse wird nun weiter unterteilt:
 - Erste 10 Bit: Index im Verzeichnis ($2^{10} = 1024$)
 - Nächste 10 Bit: Index in der Seitentabelle
- Restliche 12 Bit wie üblich als Offset innerhalb der Seite
- -> Jetzt können große Teile der Seitentabelle ausgelagert werden



Auch mehrstufig realisierbar!

Seitenersetzungsstrategien

Frage: Wenn seiten ausgelagert werden müssen, welche dan?

Optimaler Algo:

- Angenommen zum aktuellen Zeitpunkt befinden sich k Seiten $\{p_0 \dots p_{k-1}\}$ eines Programms im Hauptspeicher.
- Markiere jede Seite p_i mit der Anzahl der Befehle, die noch ausgeführt werden müssen, bis ein Zugriff auf diese notwendig ist.
- Muss eine Seite ausgelagert werden, wähle die mit der aktuell höchsten Anzahl noch auszuführenden Befehle.

Problem: **Nicht realisierbar, da nicht bekannt**

Grundidee:

- **Lokalität:**
Adressen, auf die zugegriffen wurde \rightarrow wird in naher Zukunft nochmal zugegriffen
- Lagere zufällig eine Seite aus, auf die länger kein Zugriff stattfand.
- Jede Seite erhält Statusbits: •
 - **R** – auf die Seite wird/wurde lesend zugegriffen.
 - **M** – auf die Seite wird/wurde schreibend zugegriffen.
 - Bei Prozessbeginn alle Statusbits 0
- Alle Seiten werden in Klassen eingeteilt:
 - Klasse 0: nicht referenziert, nichtmodifiziert
 - Klasse 1: nicht referenziert, modifiziert
 - Klasse 2: referenziert, nicht modifiziert
 - Klasse 3: referenziert und modifiziert

Algorithmen:

NRO:

- Muss eine Seite ausgelagert werden, entferne eine zufällige Seite aus der niedrigsten Klasse, in der Seiten existieren. □

- **Vorteil:**
 - Einfach zu implementieren
 - Leicht verständlich □
- **Nachteil:**
 - Es ist keine optimale Strategie, eine zufällige Seite zu entfernen.

FIFO replacement:

- **Lagere die Seiten aus, die am längsten im Hauptspeicher ist. □**
- **Nachteil:**
 - Können nur wenige Seiten verwaltet werden, dann können Seiten mit hoher Zugriffsrate voreilig ausgelagert werden.

Second-Chance:

Überprüfe R-Bit der Seite am Listenende:

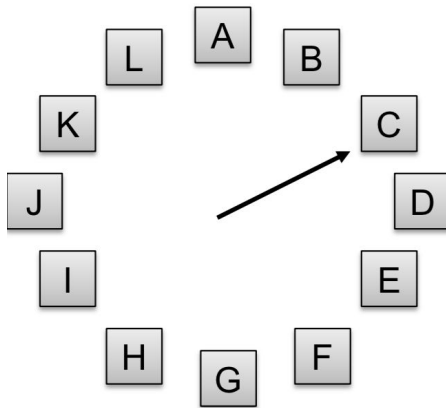
- **R gesetzt:** Seite wird festgehalten, sie gelangt an den Listenbeginn. Das R-Bit wird gelöscht und die Seite am Listenende wird überprüft.
- **R nicht gesetzt:** kein Zugriff, Seite wird entfernt.

Nachteil:

- **Wird auf jede Seite zugegriffen, degeneriert der Second-Chance-Algorithmus zum FIFO-Algorithmus.**

Clock Replacement

- Vereinfachung von Second Chance
- Vermeiden des Kopierens vollständiger Seiten durch Ringstruktur: Um eine Seite vom Ende der Liste an den Anfang zu kopieren, kann man in der Ringstruktur den Anfangszeiger auf den Vorgänger des Anfangselementes setzen.
- Wiederhole die Wahl des Vorgängers, bis Seite mit $R=0$ gefunden. Diese Seite wird aus Ringstruktur gelöscht und durch neue ersetzt
- Clock ist nur eine spezielle Implementierung von Second- Chance.



Sobald ein Seitenfehler auftritt, wird die Seite referenziert, auf die der Pfeil zeigt. Die Folgeaktion hängt dann vom R-Bit ab:

R = 0: Seite hinauswerfen.

R = 1: Lösche R und wandere mit dem Pfeil weiter.

Least Recently Used (LRU):

Annäherung an den optimalen Algorithmus

- **Entferne bei Seitenfehler die am längsten nicht benutzte Seite.** □

Probleme:

- Vollständige Implementierung benötigt vollständige Liste aller Seiten sortiert nach Zugriffszeit (Seite mit letztem Zugriff vorne).
- Liste muss nach jedem Zugriff aktualisiert werden.

Realisierung mit Matrix (Hardware):

	0	1	2	3	
0	0	0	0	0	= 0
1	1	0	1	1	= 11
2	1	0	0	1	= 9
3	1	0	0	0	= 8

	0	1	2	3	
0	0	1	1	1	= 7
1	0	0	1	1	= 3
2	0	0	0	1	= 1
3	0	0	0	0	= 0

➤ Ältester Seitenrahmen ist jetzt Seitenrahmen 3.

Ladestrategien:

Seiten müssen nicht nur ausgelagert werden sondern auch wieder eingelagert.

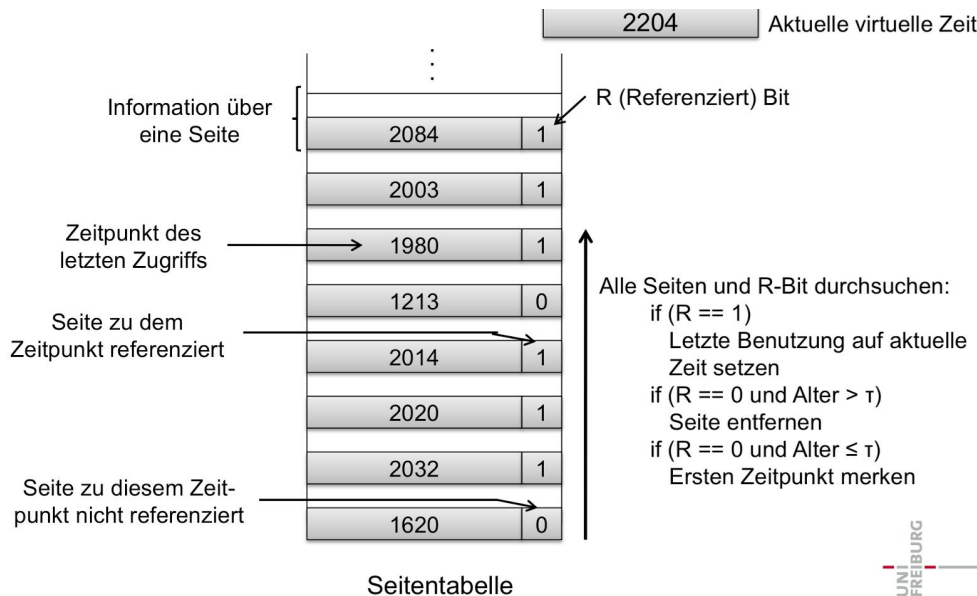
- **Swapping:**
Übertragung eines ganzen Adressraums mit einem einzigen Zugriff auf den Hauptspeicher.
- **Demand Paging:**
Die benötigten Seiten werden genau dann in den Speicher geladen, wenn auf sie zugegriffen wird.
- **Prepaging:**
Es werden Seiten geladen, auf die in der Zukunft ein Zugriff erwartet wird. Erfordert Kenntnisse über typische Zugriffsmuster.
- **Page Clustering:**
Gemeinsame Übertragung von mehreren zusammengehörigen Seiten. Ermöglicht die Nutzung großer Seiten auf Hardware, die nur geringe Seitengrößen unterstützt.

Working-Set-Replacement

- Identifiziere die wenigen, für einen Prozess relevanten Seiten (working set, Arbeitsbereich). Lagere stets prozessfremde Seiten aus.
- Wird ein **Prozess fortgesetzt**, dann lagere immer das **gesamte working-set** auf einmal ein.

Implementierung eines Working-set-Algorithmus

- Praktikabler Ansatz mit Timer-Unterbrechung
- Seitentabelle enthält pro Seite virtuelle Zeit des letzten Zugriffs und R-Bit
- Auslagerungsentscheidung auf Basis von virtueller Zeit und R-Bit
- **Nachteil: Die Seitentabelle muss komplett durchlaufen werden.**



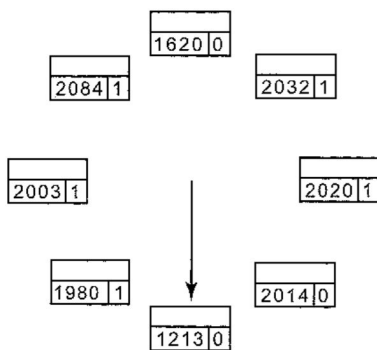
WS-Clock- Seitenersetzungsalgorithmus

Kombination Working-set-Algorithmus mit Clock-Algorithmus

- Ringförmige Liste von Seitenrahmen
- Rahmen mit R- und M-Bit sowie virtueller Zugriffszeit

Vorgehensweise:

- Seite wurde in Periode τ benutzt, lösche R-Bit und rücke Zeiger vor.
- R-Bit ist 0, lagere neue Seite mit aktuellem Zeitstempel ein und rücke Zeiger vor.



Algorithmus	Bewertung
Optimal	Unrealisierbar, gedacht für Vergleiche z.B. in Beweisen
NRU (Not Recently Used)	Sehr primitiv
FIFO (First-In, First-Out)	Wichtige Seiten können entfernt werden.
Second Chance	Enorme Verbesserung gegenüber FIFO
Clock	Realistisch
LRU (Least Recently Used)	Exzellent, schwierig zu implementieren
NFU (Not Frequently Used)	Unoptimiert mit LRU vergleichbar
Aging	Optimierter Algorithmus, der fast LRU erreicht
Working set	Etwas aufwändig in der Implementierung
WSClock	Guter und effizienter Algorithmus

In der Realität wird insbesondere WSClock eingesetzt.

Zusammenfassung:

- Speicherverwaltung ist sehr wichtig für die Performance des Gesamtsystems
- In der theoretischen Betrachtung ähnliche Probleme wie beim Scheduling
- Optimale Lösung erfordert Blick in die Zukunft
- Jeder Fehler kostet Zeit □
- Praktisch Ausnutzen von bestimmten Prinzipien (Lokalität, Working Sets)
- Virtuelle Adressräume
 - Paging ermöglicht das effiziente Ausnutzen des Speichers
 - Auslagern ermöglicht eine Überbelegung, fängt somit Lastspitzen ab
 - Lokalität Kernkonzept für das erfolgreiche Gelingen
 - Implementierung oft direkt durch Hardware unterstützt

Kapitel 5: Eingabe/Ausgabe (IO)

Geräte (Drucker, Festplatten) sind blackboxes, welche eine Elektronik beinhalten, die das Gerät kontrolliert (=Device Controller).

Wie bekomme ich Daten von und zu diesen Blackboxes?

- Device Controller
- Memory Mapped I/O
- Interrupts

IO Geräte Kategorien:

- Block device (Blockgerät)
 - Gerät, das Informationen in Blöcken fester Größe speichert.
 - Jeder Block hat eine eigene Adresse.
 - Jeder Block kann unabhängig von anderen gelesen und geschrieben werden.
 - Typisches Beispiel: Festplatten
- Character device (Zeichengerät)
 - Gerät, das einen Strom von Zeichen ohne Blockstruktur empfängt oder liefert.
 - Eine Adressierung bestimmter Bereiche ist nicht möglich.
 - Typische Beispiele: Drucker, Netzwerkkarte, Maus

Arten der Kommunikation mit Geräten:

Port Mapped IO

- Jeder Device Controller verfügt über einige Statusregister
- Beispielsweise die Position des Lesekopfes der Festplatte
- Diese Register sind über einen I/O-Bus mit dem Prozessor verbunden
 - Verschiedene Portnummern identifizieren unterschiedliche Register in den einzelnen Hardwarekomponenten
 - Portnummern sind entweder fest vorgegeben (einige standardisierte Geräte) oder werden dynamisch ausgehandelt (plug and play)
- Jedem Gerät (genauer: Register) ist also eine Portnummer zugewiesen (port mapping)
- Port Mapped I/O (PMIO)

- Im Grunde ist das sehr ähnlich zu Speicherzugriff
- Portnummer entspricht einer Speicheradresse
- OUT x, y: „Schreibe Wert y an Portnummer x“ (Befehl)
- Getrennte „Adressräume“ für I/O und Speicher

Memory Mapped IO

- Basierend auf der Idee des virtuellen Speichers:
- Die Kontrollregister werden direkt in den normalen Speicheradressraum eingeblendet (memory mapping)
- Zugriff dann mit den gleichen Instruktionen wie Speicherzugriff
- Alle Adressen oberhalb von z.B. 0xC0000000 sind I/O, alles darunter ist Hauptspeicher

- **Vorteile:**
 - Im Gegensatz zu speziellen Maschinenbefehlen ist Speicherzugriff auch in Hochsprachen (C, Pascal, C++) sehr **einfach** möglich
 - Eine Abbildung in verschiedene Adressräume erlaubt sehr feine Zugriffskontrolle robuster

-

In der Praxis meist ein **Hybridsystem** aus beiden I/O- Arten

IO aus Softwaresicht:

Schematischer Ablauf:

- CPU programmiert Operation im Gerätecontroller
- Controller steuert Hardware, CPU wartet auf Ergebnis
- CPU liest Ergebnis aus Controller aus
- CPU programmiert nächste Operation

Wie sieht das Warten für den Prozessor aus?

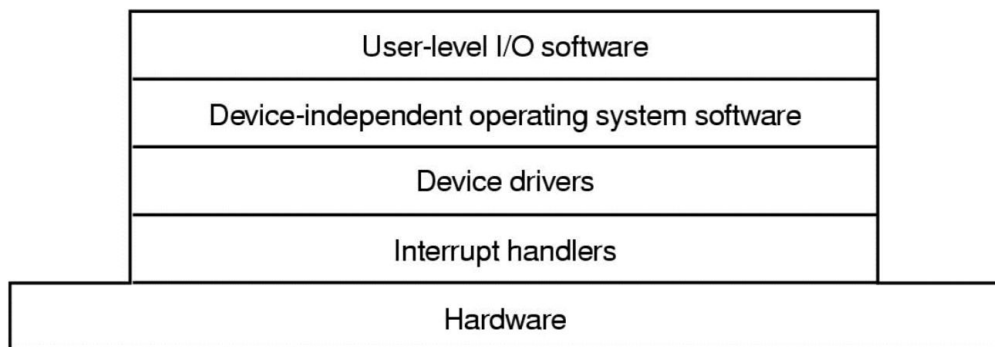
- Die CPU wartet nach dem Absenden eines Steuerbefehls solange, bis ein Ready-Bit im Controller gesetzt wurde
(schlecht! polling, busy waiting -> CPU kann in der Zwischenzeit keine anderen Aufgaben übernehmen)
- **Interrupt:**
 - Das Gerät teilt der CPU von außen mit, dass es mit der Operation fertig ist
 - CPU kann andere Aufgaben übernehmen bis sie unterbrochen wird
 - Der Interrupt-Handler schließt dann die I/O-Operation ab, indem er die Daten aus dem Controller in den Hauptspeicher kopiert

Direct Memory Access (DMA)

- Den Prozessor nicht mit I/O belasten, stattdessen direkt von Gerät in den Speicher schreiben
- Ein neuer Controller, der DMA-Controller, übernimmt die Steuerung des Kopiervorgangs -
- Vergib einen Auftrag an DMA-Controller, erledige bis zum Ende der Bearbeitung andere Dinge

I/O-Software – Grundlagen

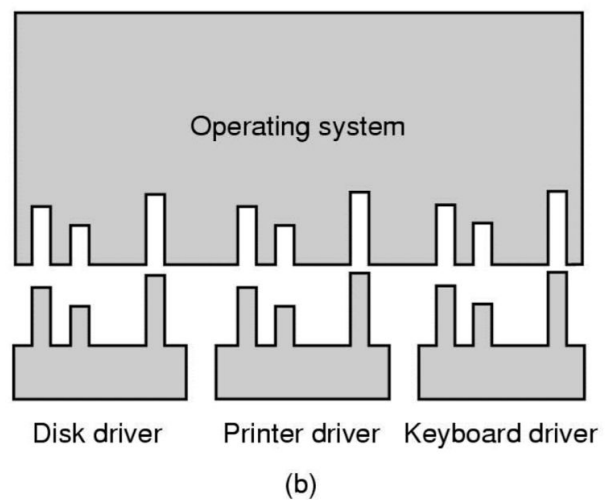
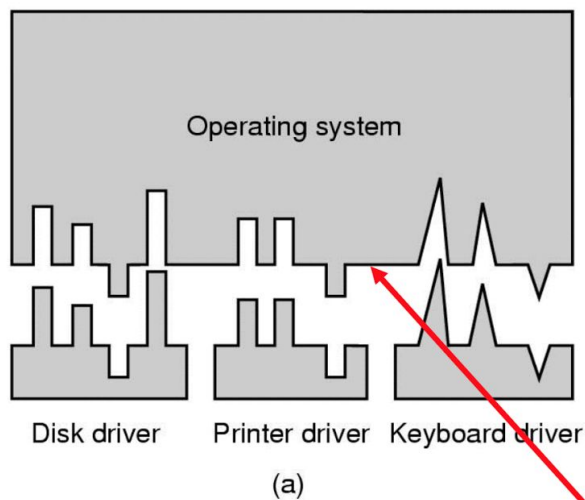
- Die Abstraktion dieser Komplexität ist eine weitere Kernaufgabe von Betriebssystemen
- Wichtige Prinzipien daher:
 - **Geräteunabhängigkeit:** ein „write“ funktioniert auf Festplatte genauso wie auf das Netzwerk -
 - **Einheitliche Namensverwendung** (alle Geräte sind bspw. über Pfade im Dateisystem erreichbar) -
 - **Fehlerbehandlung** so nah wie möglich an der Quelle (Hardware)
 - **Synchrone** (Blockieren) und **asynchrone Zugriffe** (Interrupt- gesteuert) **erlauben**
 - Abstraktion der Zugriffssemantik
 - Abstraktion der Benennung
 - Z.B. egal ob Festplatte oder CD bei Dateizugriff
 - Abstraktion von Zugriffsdetails und Fehlerbehandlung
 - Abstraktion vom Zeitverlauf der E/A Aktion
 - Asynchrones Verhalten und Puffern



Gerätetreiber:

- Aufgabe: Verbergen der Komplexität der Registerbelegungen des Gerätecontrollers
- Jedes Gerät benötigt üblicherweise seinen eigenen Treiber
- Treiber sind praktisch immer Teil des Betriebssystems mit privilegierten Rechten
- Sie stellen für darüberliegende Schichten eine Programmierschnittstelle (API) bereit
- Driver sollten mehrfach wieder aufgerufen werden können (z.B. mehrere Festplatten), also reentrant sein.

Also: Geräte UNABHÄNGIGE Schnittstellen:



Kapitel 7: Sicherheit

Sicherheitsziele:

- **Vertraulichkeit** (data confidentiality):
Geheime Daten sollen geheim bleiben.
- **Datenintegrität** (data integrity):
Unautorisierte Benutzer dürfen ohne Erlaubnis des Besitzers Daten nicht modifizieren.
- **Systemverfügbarkeit** (system availability):
Niemand soll das System so stören können, dass es dadurch unbenutzbar wird.
- **Datenschutz** (privacy):
Schutz von Personen vor dem Missbrauch ihrer persönlichen Daten.

Kryptographie

Kryptographie \Leftrightarrow Kryptoanalyse

Kerckhoffs'sche Prinzip (moderne Fassung):

Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Verfahrens abhängen.

- Ver- und Entschlüsselungsalgorithmen sollten immer öffentlich sein.
- Sicherheit durch Verschleierung der Funktionsweise (Security through Obscurity) ist eine falsche Sicherheit!

Verschlüsselungsarten:

Symmetrische Kryptographie:

- Ein symmetrisches Kryptosystem ist ein Kryptosystem, bei welchem im Gegensatz zu einem asymmetrischen Kryptosystem beide Teilnehmer denselben Schlüssel verwenden.

Beispiele:

- **Caesar-Verschlüsselung:** Verschieben um xx Buchstaben -> unsicher!
(Monoalphabetisch)
- Geheimalphabet -> Wahlfreies Vertauschen von Buchstaben statt einfaches Weiterschieben -> Besser, aber auch knackbar -> Statische Eigenschaften von Sprachen

Verbesserung: Polyalphabetische Kryptographie: Polyalphabetische Ersetzungsschiffren bezeichnen in der Kryptographie Formen der Textverschlüsselung, bei der einem Buchstaben/Zeichen jeweils ein anderer Buchstabe/Zeichen zugeordnet wird. Im Gegensatz zur monoalphabetischen Substitution werden für die Zeichen des Klartextes mehrere Geheimtextalphabete verwendet.

Beispiel: **Enigma**

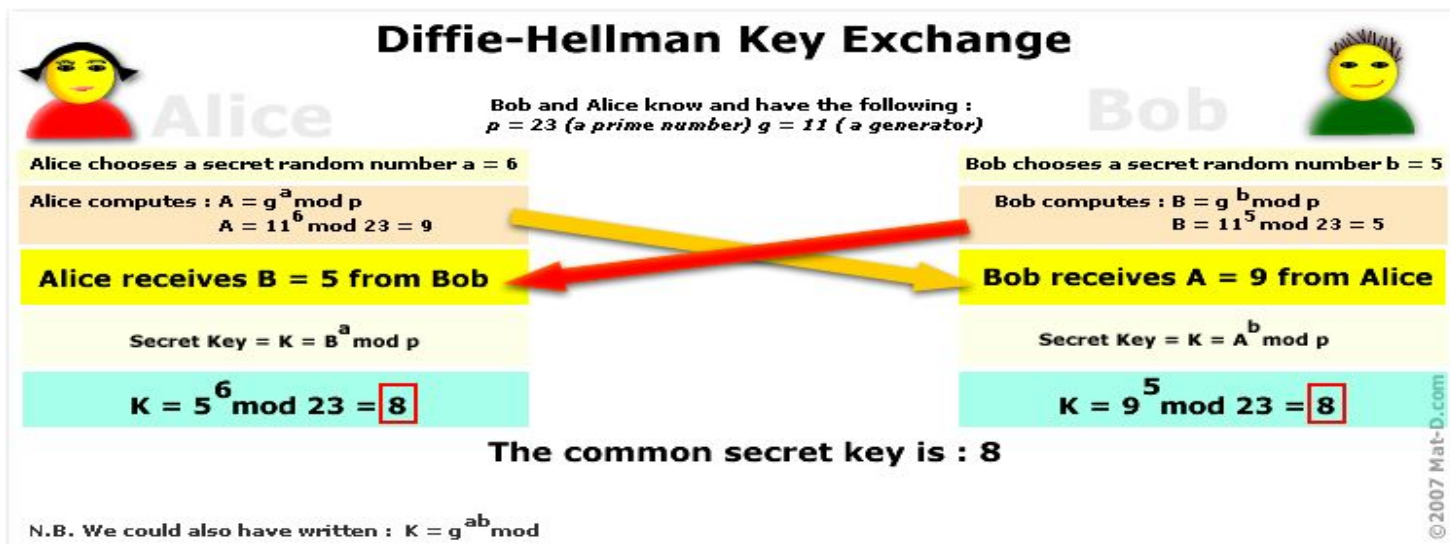
Symmetrische Verschlüsselung:

- **Monoalphabetische** Verschlüsselung recht **einfach** zu entschlüsseln
- **Polyalphabetische** Verschlüsselungen Anfang des 20. Jahrhunderts **sehr sicher**
- Heutige symmetrische Verschlüsselungen basieren auf komplexen algebraischen Berechnungen
 - Erschweren Häufigkeitsanalysen
 - Bei den besseren Algorithmen nur durch Brute Force knackbar, also ausprobieren aller Schlüssel -

- Schnelle Computer können das sehr schnell •
 - Sichere Schlüssellängen ab 256-Bit-Schlüssel $\Rightarrow 2^{256} \approx 10^{77}$
- **Problem** bleibt: **Austausch des geheimen Schlüssels**

Lösungen:

Diffie-Hellmann-Schlüsselaustausch



- > Es werden nur Informationen ausgetauscht, aus denen der Schlüssel selbst nicht zurückgerechnet werden kann
- ABER: **Anfällig für Man-in-the-Middle-Angriffe**

Public-Key-Kryptographie

Ein Public-Key-Verschlüsselungsverfahren ist ein **asymmetrisches** Verschlüsselungsverfahren, um mit einem öffentlichen Schlüssel einen Klartext in einen Geheimtext umzuwandeln, aus dem der Klartext mit einem geheimen Schlüssel wieder gewonnen werden kann. Der **geheime Schlüssel muss geheim gehalten werden**, und es muss praktisch unmöglich sein, ihn aus dem **öffentlichen Schlüssel** zu berechnen. Der **öffentliche Schlüssel muss jedem zugänglich sein**, der eine verschlüsselte Nachricht an den Besitzer des geheimen Schlüssels senden will.

Also:

- Alice kennt öffentlichen Schlüssel von Bob $K_{\text{öffentlich}}$ und verschlüsselt damit den Klartext P .
- Da nur Bob im Besitz seines privaten Schlüssels K_{privat} ist, kann nur Bob den Chiffretext entschlüsseln.

Bleibendes **Problem**: **Woher weiß ich, dass der öffentliche Schlüssel von Bob der echte ist?**
(MITM immernoch möglich)

Passwörter

- Unix speichert die Passwörter verschlüsselt (hashed) in einer Datei
- Bei Login wird das eingegebene Passwort ebenfalls verschlüsselt und mit dem Eintrag in der Datei verglichen
- **Problem:**
 - Ein Angreifer könnte eine Liste von wahrscheinlichen Passwörtern (Rainbow Table) mit dem selben Verfahren verschlüsseln.
 - Anschließend muss der Angreifer die öffentlich einsehbare Passwortdatei durchlaufen und die verschlüsselten Passwörter vergleichen.

Lösung:

- Kombination des Passwortes mit einer n-Bit-Zufallszahl (=Salt)
- Salt wird unverschlüsselt in die Passwortdatei gespeichert.
- Falls als Passwort „hallo“ vermutet wird, muss der Angreifer anschließend 2n Zeichenketten verschlüsseln („hallo0000“, „hallo0001“, ...).

Angriffsmethoden:

- **Login Spoofing** - fake login screen präsentieren
- **Ausnutzen von programmierfehlern**
- **Pufferüberlauf**angriffe
- **Formatstring**-Angriffe
- **Return-to-libc**-Angriffe
- Angriff durch **Code-Injektion**
- **Privilege-Escalation**-Angriff

Malware:

- Trojanische Pferde
- Viren
- Würmer
- Spyware
- Rootkits

Dateisysteme

Wird über Dateinamen "angesteuert", hat noch weitere Attribute wie Änderungsdatum, Ersteller etc

Dateisystem: Hierarchische Strukturierung des externen Speichers -

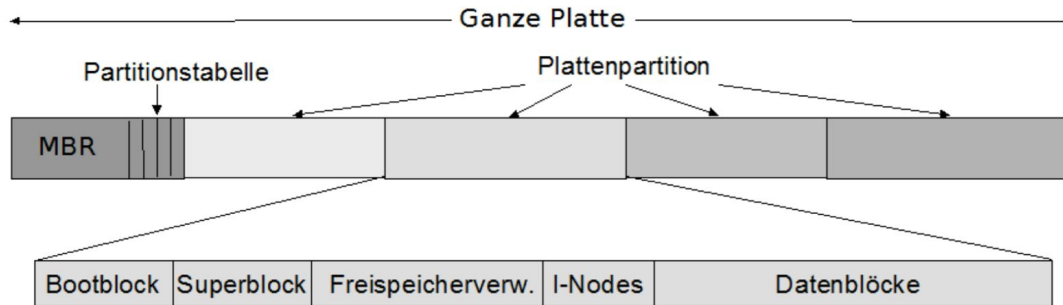
- Ausgehend vom „Wurzelverzeichnis“ (engl: root directory, Unix: /)
- Mehrere Dateien sind in einem Verzeichnis zusammengefasst
- Daten befinden sich in den Dateien an den „Blättern“ der Hierarchie
- Absolute Namen durch Aneinanderreihung der Verzeichnisnamen und des Dateinamens (mit Trennsymbol je nach Betriebssystem /, \, >, :)

Implementierung:

Auf einer Festplatte sind Daten in einzelnen Blöcken fester Größe gespeichert (Sektoren)

- Wie jedes System hat nun das Dateisystem zur Aufgabe
 - Diese Blöcke effizient zu verwalten -
 - Sie bestimmten Nutzern (= Dateien) zuzuordnen
- Somit wird eine Abstraktion geschaffen, damit der Rest des Betriebssystems bequem auf Daten (in Dateien organisiert) zuzugreifen, ohne die Details zu kennen -
 - Gleiche Schnittstelle für Dateien auf Festplatte, SSD, CD, USB-Stick

Layout von Speichermedien:

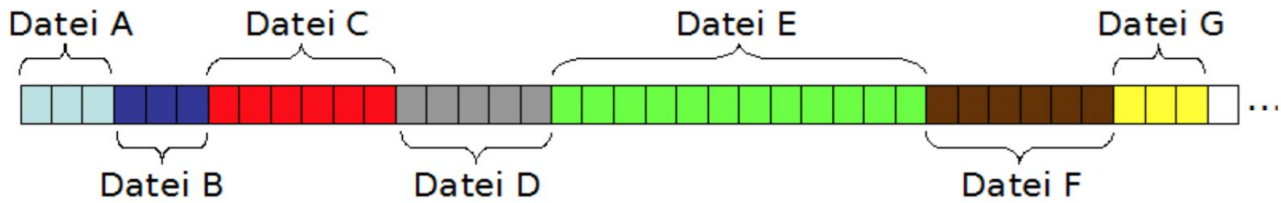


- Superblock des Dateisystems enthält wichtige Parameter über das Dateisystem
 - z.B. Schreibschutzmarkierung, Dateisystemtyp, ...
- Freispeicherverwaltung: Informationen über freie Blöcke im Dateisystem
- Inodes: Informationen über die eigentlichen Dateien
- Datenblöcke: Eigentliche Inhalte der Dateien

3 Ansätze zur Realisierung von Dateien:

- **Zusammenhängende Belegung**
Eine Datei wird immer in aufeinanderfolgende Blöcke geschrieben -
- **Belegung durch verkettete Listen**
Von einem Block kann zum nächsten gesprungen werden -
- **Inodes**

Zusammenhängende Belegung:



Vorteile:

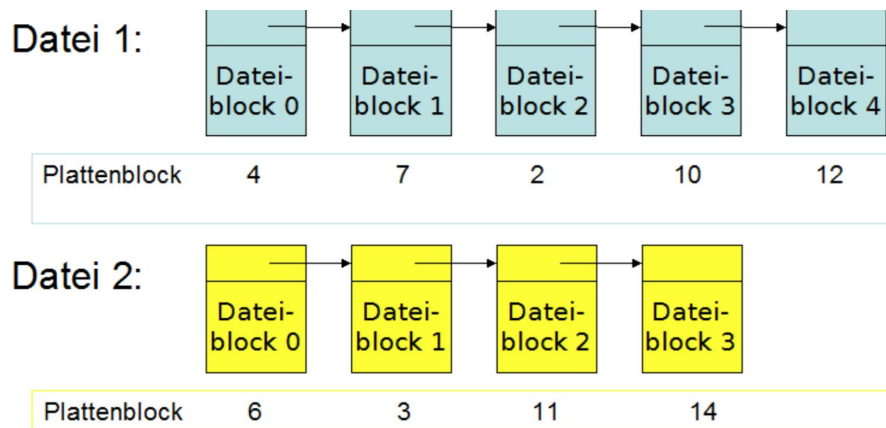
- **Schnelles** sequentielles **Lesen**
- **Einfache** Implementierung
- Start- und Endblock reicht zur Verwaltung aus

Problem:

- Löschen von Dateien D und F erzeugt externe Fragmentierung!

→ also keine gute Idee, wird nur bei 1x Beschreibbaren Datenträgern (CDRom usw) benutzt.

Verkettete Listen:



Vorteile:

- Keine Fragmentierung
- Sequentielles Lesen

Nachteile: Overhead

Verschiedene Dateisysteme:

FAT - File Allocation Table

Basiert auf verketteter Liste von Blöcken für jede Datei mit Verbesserung:

- **Trenne** die **Nutzdaten** und **Informationen** über die verkettete Liste
- Information über Verkettung als gesonderte **Allokationstabelle** (FAT, File Allocation Table)
- Halte **FAT** komplett im **Hauptspeicher**
 - Bei wahlfreiem Zugriff nur **schnelle Hauptspeicherzugriffe**

Bootblock/ Superblock	FAT	FAT (Kopie)	Wurzel- verzeichnis	Daten
--------------------------	-----	----------------	------------------------	-------

- In der FAT sind die belegten und freien Blöcke im Dateisystem erfasst
- Im **Wurzelverzeichnis** ist für jede Datei (und jedes Unterverzeichnis) **ein Eintrag** vorhanden:
 - Bei **FAT12** und **FAT16** eine **feste Größe**, daher begrenzte Anzahl an Dateien im Wurzelverzeichnis
 - Bei **FAT32** eine **variable Größe** (bestimmt durch Superblock)
 - Unterverzeichnisse werden als Eintrag mit gesetztem „Verzeichnis“-Bit gespeichert, die als Daten dann selbst wieder Verzeichniseinträge enthalten

Einträge in der FAT haben folgende Struktur:

- Für jeden Datenblock einen Eintrag -
- **0x0000** bedeutet, der Block ist **frei**
- **0xFFFF7** bedeutet, der Block ist **defekt**
- **0xFFFF8** bedeutet, dieser Block ist das **Ende** einer Kette-
- Alles andere gibt den nächsten Block der Datei an

Vorteil:

- Da FAT im Hauptspeicher abgelegt, muss bei wahlfreiem Zugriff auf Block n nur eine Kette von Verweisen im Hauptspeicher verfolgt werden

- Sehr **einfache** Datenstrukturen, einfach zu Verwalten und zu implementieren

Nachteil:

- **Größe der FAT im Speicher**
- Anzahl der Einträge = Gesamtzahl der Plattenblöcke ist begrenzt
 - Häufig erweitert (von 12 auf 16 auf 32 Bit für Adressen)

Beobachtung/Optimierungsansatz: Halte nur eine Struktur im Speicher, welche die aktuell bearbeiteten Dateien und ihre Blöcke referenziert

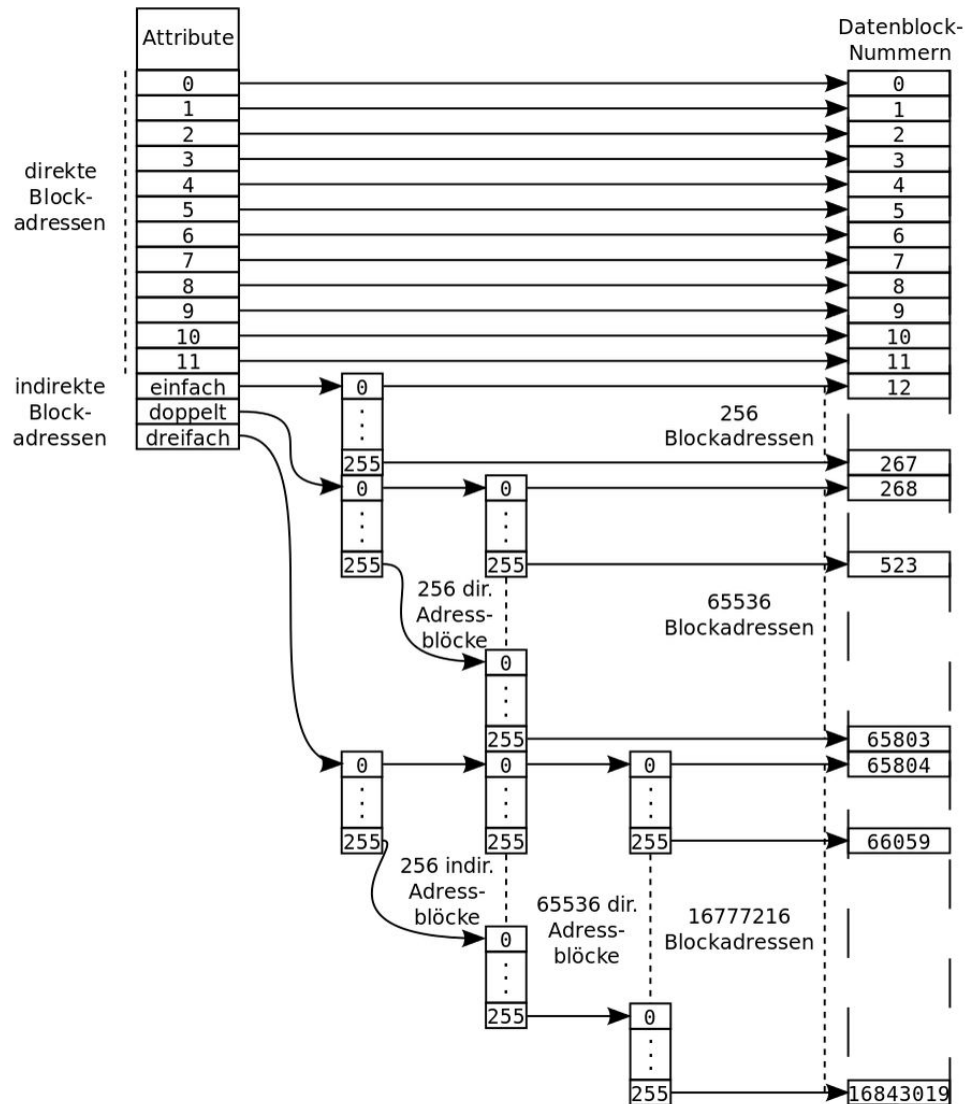
Introducing Inodes!

Ein Inode ist die grundlegende Datenstruktur zur Verwaltung von Dateisystemen mit **unix**artigen Betriebssystemen. Jeder Inode wird innerhalb einer Partition eindeutig durch seine Inode-Nummer identifiziert. Jeder Namenseintrag in einem Verzeichnis verweist so auf genau einen Inode. Dieser wiederum verweist auf genau eine Datei oder ein Verzeichnis.

- **eine iNode pro Datei**
- die Inode ermöglicht Zugriff auf alle Blöcke der Datei
- die Inode muss nur dann im Speicher sein, wenn eine Datei offen ist
- Alle Inodes sind durchnummeriert

Struktur:

- Alle Attribute der Datei
- m Adressen von Plattenblöcken
- UNIX System V: $m = 10$
- Weitere Referenzen auf Plattenblöcke verwendet falls Datei größer als $m * \text{Blockgröße}$



Da alle Inodes lediglich eine Nummer haben, benötigt man einen „**Einstiegspunkt**“ in eine Datei
 -> Verzeichnisse

- Verzeichnisse sind ebenfalls „Dateien“ (mit gesetztem speziellen Verzeichnis-Bit)
- Sie liefern eine Abbildung von Datei- bzw. Verzeichnis- namen auf Inode-Nummern
- Jeder Verzeichniseintrag ist ein Paar aus Name und Inode- Nummer
- Über Inodes kommt man dann zu den Dateiinhalten

NTFS - “New Technology File System”

Im Vergleich zum Dateisystem FAT bietet NTFS unter anderem einen gezielten Zugriffsschutz auf Dateiebene sowie größere Datensicherheit durch Journaling. Allerdings ist keine so breite Kompatibilität gegeben wie bei FAT. Ein weiterer Vorteil von NTFS ist, dass die Dateigröße nicht wie bei FAT auf 4 GiB beschränkt ist.

Journaling:

- Inodes und MFT (NTFS version von Inodes) sind zwei sehr komplexe Datenstrukturen in Dateisystemen
- Um eine Datei zu ändern muss an vielen Stellen irgendetwas verändert werden
- Wenn eine Datenstruktur bereits aktualisiert wurde, während die nächste Datenstruktur noch im alten Zustand ist, ist dieser Zustand im Dateisystem nicht vorgesehen •
 - Das Dateisystem ist „korrupt“ (eigentlich: verdorben)
- Dieser Zustand wird normalerweise wenige Millisekunden später behoben, wenn der Dateisystemtreiber die nächsten Datenstrukturen aktualisiert hat
- Die Schreiboperationen in komplexen Dateisystemen bestehen aus vielen Einzelschritten
- Idee:
 - Schreibe diese Einzelschritte vorher in ein „Tagebuch“ oder Journal -
 - Währenddessen finden noch keine Änderungen am Dateisystem selbst statt -
 - Anschließend werden die Schritte aus dem Tagebuch abgearbeitet
 - Wird dieser Vorgang unterbrochen, kann anhand des Tagesbuchs der vorherige Zustand (manchmal sogar der eigentliche Zielzustand) wieder hergestellt werden

Zusammenfassung Dateisystem:

- Festplatten werden hardwareseitig in einzelnen Datenblöcken organisiert
- Programme und Anwender erwarten in der Regel eine hierarchische Dateistruktur mit Verzeichnissen
- Das Dateisystem verwaltet daher Dateien und Verzeichnisse in eigenen Datenstrukturen
- Verschiedene Implementierungsvarianten für Dateisysteme
- Verkettete Listen (FAT) sind einfach, aber oft ineffizient. Inodes sind komplexer, dafür Ressourcenschonender und effizienter
- Journaling erhöht die Zuverlässigkeit auf Kosten der Geschwindigkeit
- Geeignete Wahl des Dateisystems bestimmt durch Eigenschaften des physikalischen Speichermediums und Anforderungen an den Datenzugriff