Betriebssysteme

Dr. Henrik Brosenne
Georg-August-Universität Göttingen
Institut für Informatik

Wintersemester 2013/14

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

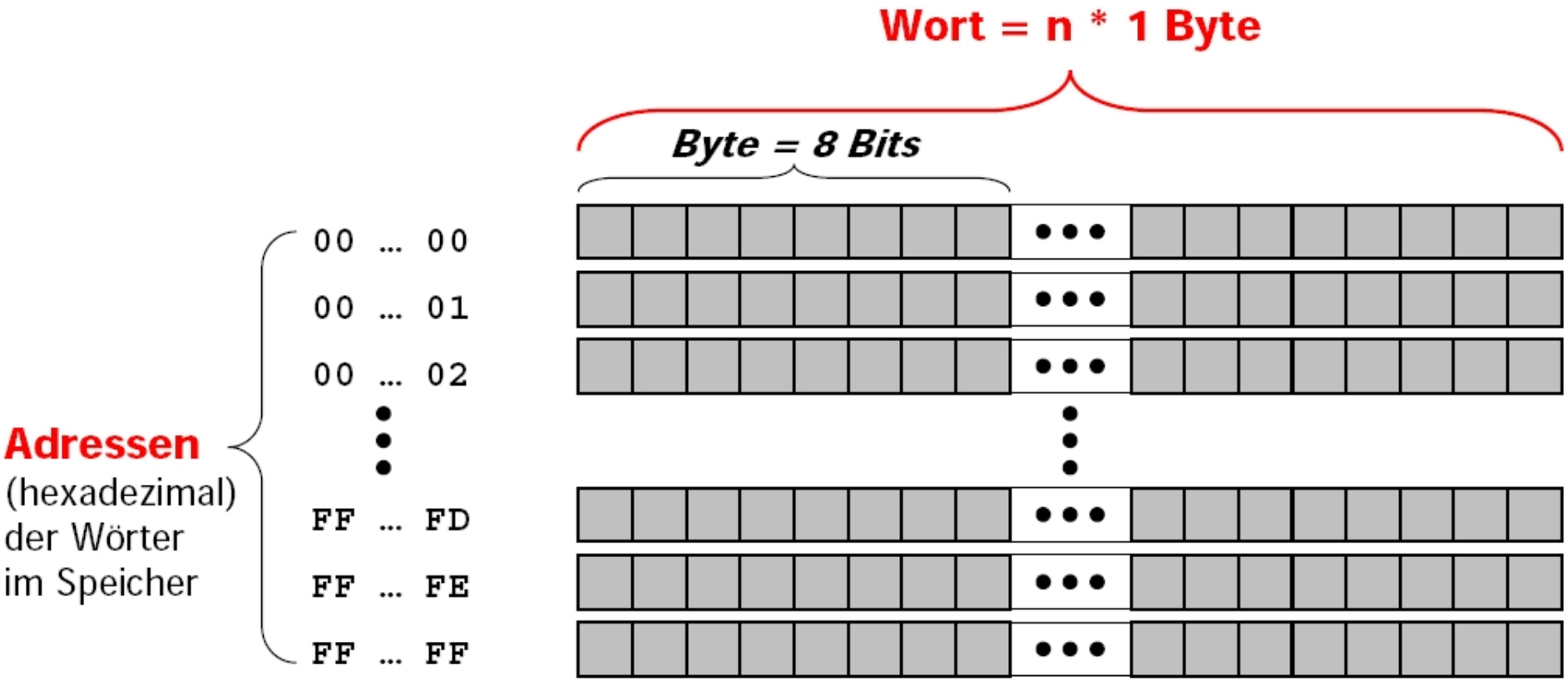
Ein-/Ausgabe

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Speicherorganisation



Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Monoprogrammierung (1/2)

Monoprogrammierung ist die einfachste Speicherverwaltungsstrategie.

Nur ein Prozess läuft, der Speicher wird zwischen Prozess und Betriebssystem aufgeteilt.

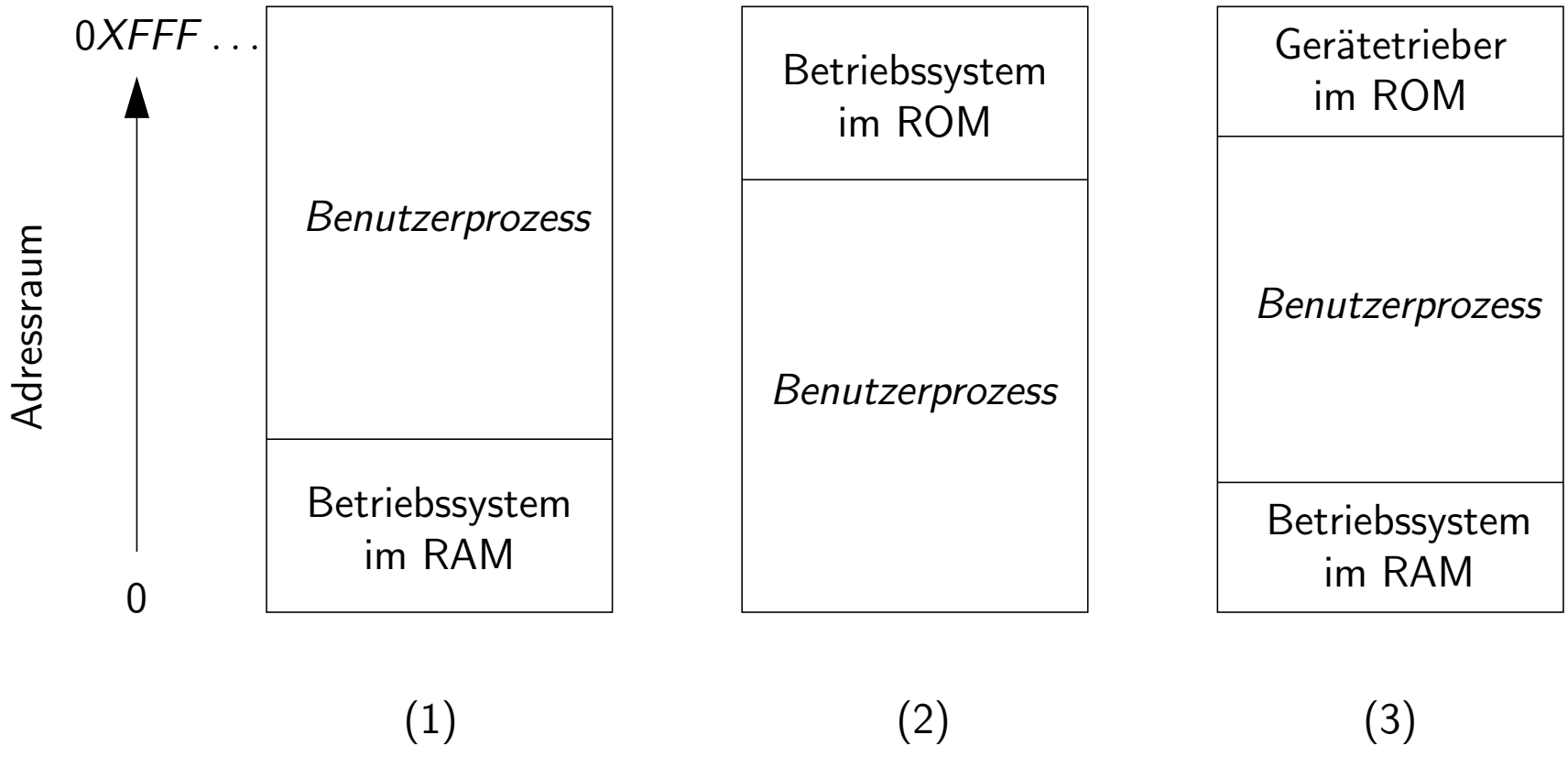
Verschiedene Möglichkeiten für die Lage des **Betriebssystems** im Speicher.

1. Nur im RAM (Random Access Memory).
   Beispiel. Frühe Mainframes und Mircocomputer.
2. Nur im ROM (Read Only Memory).
   Beispiel. Einige Palmtops und eingebettete Systeme.
3. Teil im ROM (Basic Input Output System, BIOS), Teil im RAM.
   Beispiel. MS-DOS.

Ein Prozess im freien RAM.

- Programme werden für vorgesehene feste Speicherstellen übersetzt (bei absoluten Adressierungsarten).
- Häufig kein Speicherschutz. Prozess kann Teile des Betriebssystems überschreiben und das Betriebssystem dadurch zum Absturz bringen.

Monoprogrammierung (2/2)



Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Multiprogrammierung

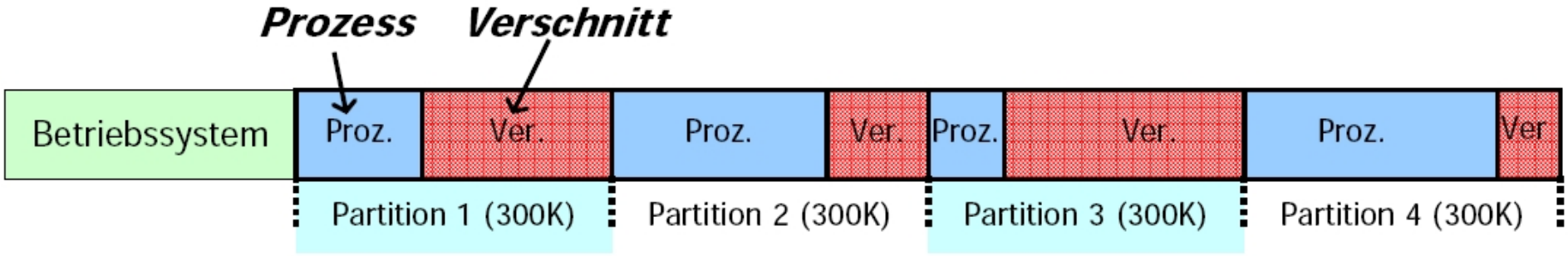
Bei (einfachem) **Multiprogrammierung** werden **mehrere** Prozesse **vollständig** im Hauptspeicher gehalten.

Der Hauptspeichers wird in **feste Partitionen** aufgeteilt.

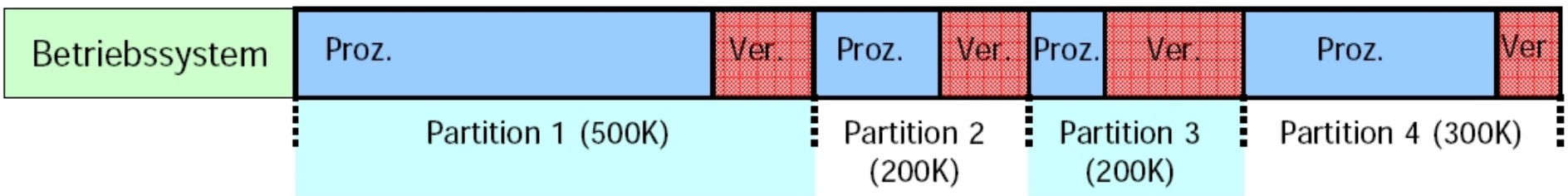
- gleicher Größe
- unterschiedlicher Größen

Feste Partitionen

Feste Partitionen gleicher Größe.



Feste Partitionen unterschiedlicher Größen.



Die Partitionen haben feste Größe, deshalb ist der ungenutzte Speicherplatz (Verschnitt) verloren, solange der, die Partition benutzende, Prozess aktiv ist.

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Speicherabbildung

Eine **programmierbare Hardwarekomponenten** ermöglicht die Abbildung von logischen auf physikalische Adressen.

- **Logische Adresse.** Adresse die ein Prozess, bzw. dessen Maschinensprachecode, benutzt.
- **Physikalische Adresse.** Tatsächliche Adresse im Speicher.

Damit entfällt die Notwendigkeit Programme für feste Adressen zu übersetzen und es ist auch **keine** feste Partitionierung des Speichers mehr nötig.

Adressraum

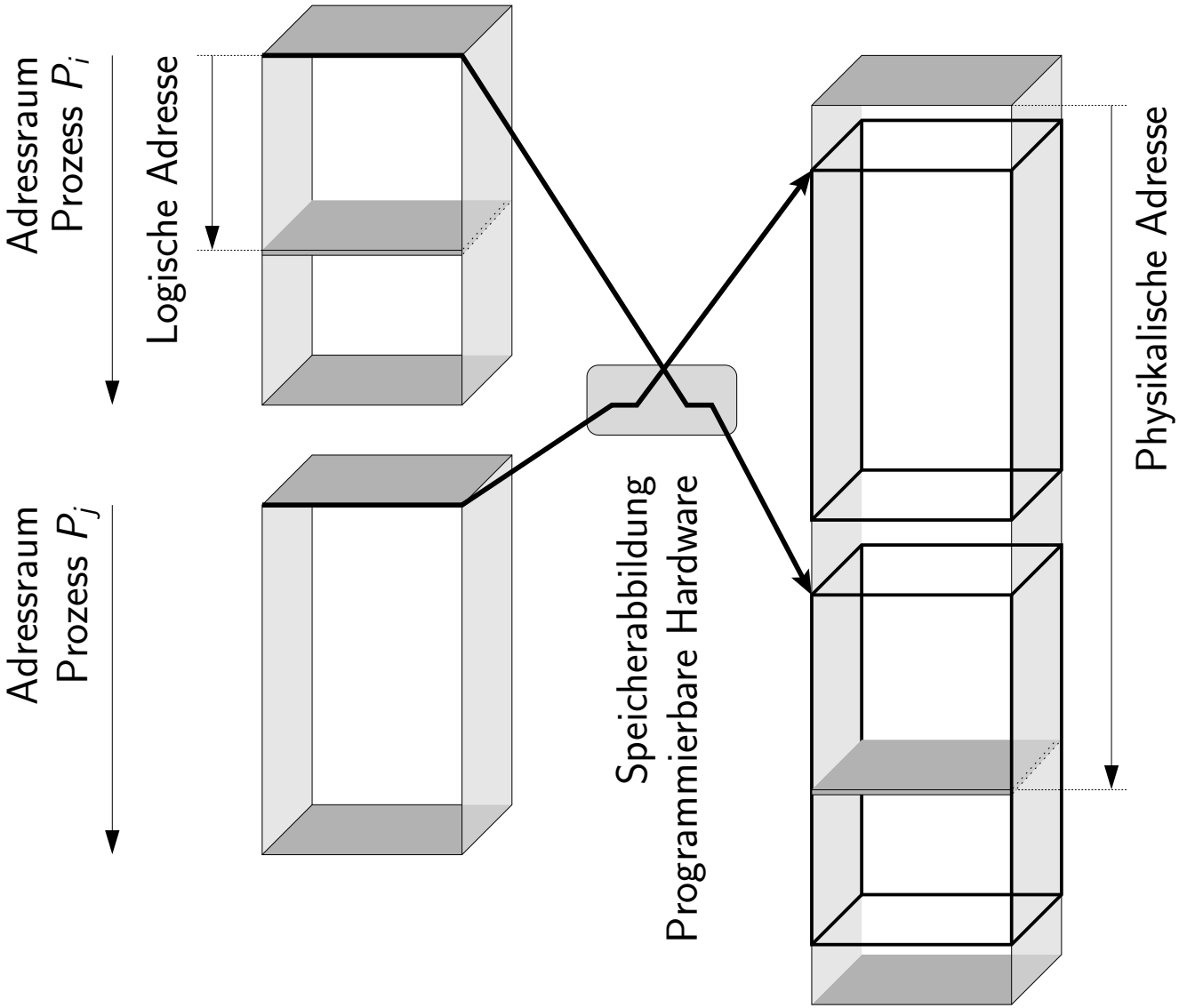
Der **Adressraums** (*adress space*) ist eine naheliegende Speicherabstraktion, wenn Speicherabbildung möglich ist.

Ein Adressraum ist eine Menge von Adressen, die ein Prozess zur Adressierung des Speichers nutzen kann.

Physikalisch ist der Adressraum, im einfachsten Fall, ein zusammenhängender Speicherbereich fester Größe.

Logisch (aus Sicht des Prozesses) ist der Adressraum der ganze zur Verfügung stehende Speicher, der in der Regel mit der Adresse 0 beginnt. Die Endadresse ist abhängig von der Größe des zugeteilen Speicherbereichs.

Speicher


Adressraum
Prozess P_i
Logische Adresse
Adressraum
Prozess P_j
Speicherabbildung
Programmierbare Hardware
Physikalische Adresse

Dynamische Relokation

Der Prozessor hat zwei zusätzliche Register, das **Basisregister** (`base`) und das **Limitregister** (`limit`). Diese Register gehören zum **Kontext** des Prozesses.

Beim Erzeugen eines Prozesses wird die dem Prozess zugeteilte physikalische Anfangsadresse in das Basisregister und die Größe des Speicherbereichs in das Limitregister geladen.

Wenn ein Prozess Speicher referenziert, um eine Befehl zu holen, einen Datenwort zu lesen oder zu schreiben, etc. werden automatisch Limit- und Basisregister benutzt.

- **Schutz**. Für jede Adresse wird geprüft ob der Wert kleiner oder gleich dem Wert im Limitregister ist, wenn nicht wird der Zugriff verweigert.
- **Speicherabbildung (Relokation)**. Zu jeder Adresse wird automatisch den Wert im Basisregister addiert und die berechnete Adresse wird auf den Adressbus geschrieben.

Bemerkung

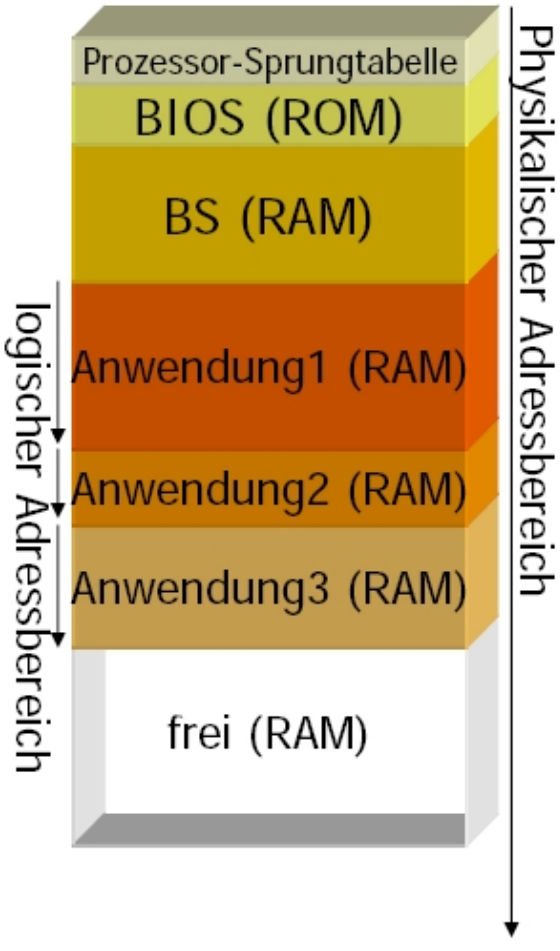
Dynamische Relokation wurde noch im Intel 8088 eingesetzt.

Beispiel UNIX

Jeder Prozesse hat eigenen Adressraum, d.h. einen eigenen logischen Adressbereich.

Bei jedem Kontextwechsel wird die Speicherabbildung (logischen auf physikalischen Adressbereich) für den aktuell laufenden Prozess eingestellt.

- Jedes Programm wird für einen gleichen/ähnlichen Adressraum kompiliert.
- Fremder Speicher ist gar nicht sichtbar.



Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Swapping, Prinzip

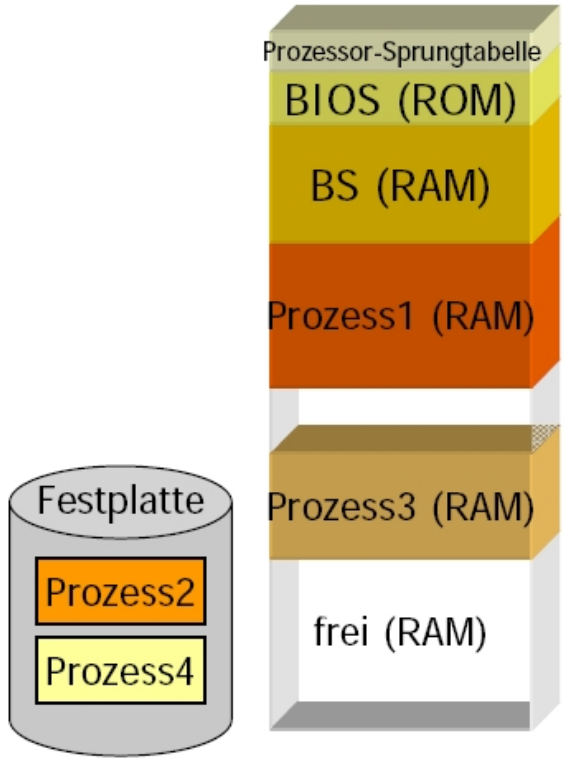
Swapping

- Auslagern von (blockierten) Prozessen auf die Festplatte.
- Einlagern eines wartenden Prozesses.
- Jeweils **vollständiger Prozess** als Ganzes!

Keine spezielle Hardware notwendig. Der Scheduler hat Überblick über wartende und blockierte Prozesse und kann Ein-/Auslagerung veranlassen.

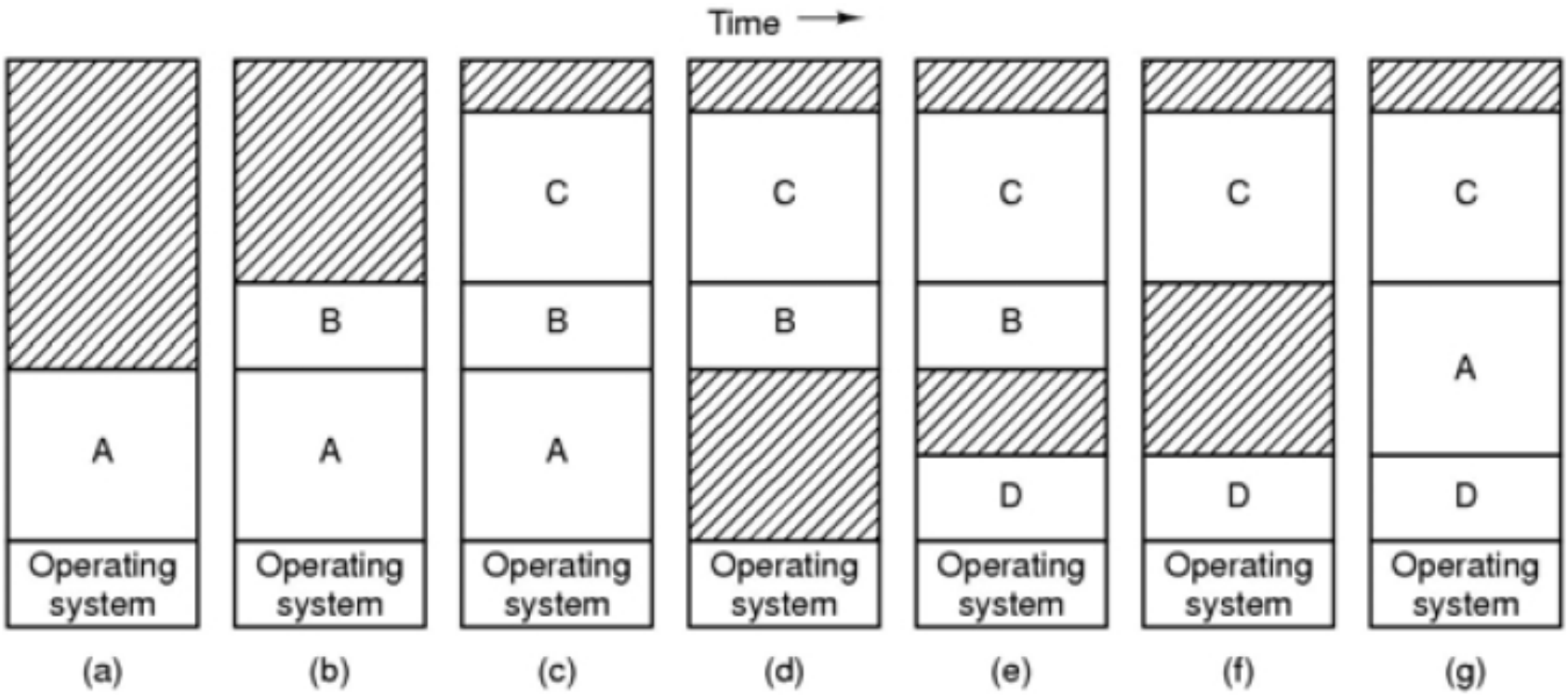
Ineffizient. Vollständiges Einlagern, obwohl bis zum nächsten Auslagern i.d.R. nur ein Teilbereich des Prozesses durchlaufen wird.

Konsequenz. Reines Swapping wird heutzutage nur noch selten genutzt.



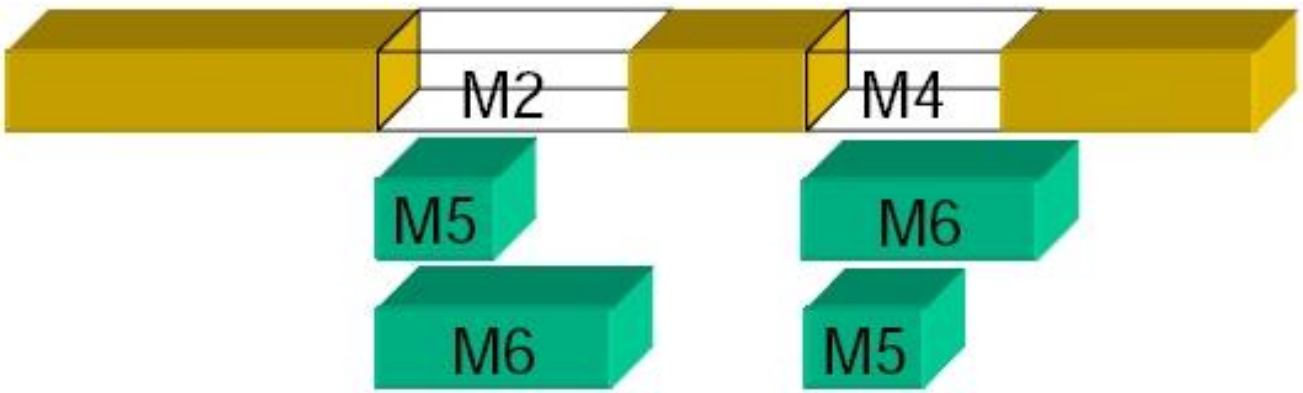
Swapping, Beispiel 1

Speicherzuteilung (physikalischer Adressbereich) für einzelne Prozesse ändert sich bei der Ein- und Auslagerung (freier Speicher = schraffierte Bereiche).



Swapping, Beispiel 2

M2, M4 werden freigegeben, wo M5, M6 platzieren?



Fazit.

Eine **Freispeicherverwaltung** ist notwendig.

Fragmentierung

Anforderung hat nur selten genau die Größe eines freien Speicherbereichs. Nach einiger Zeit entstehen viele kleine „Löcher“ im Speicher.

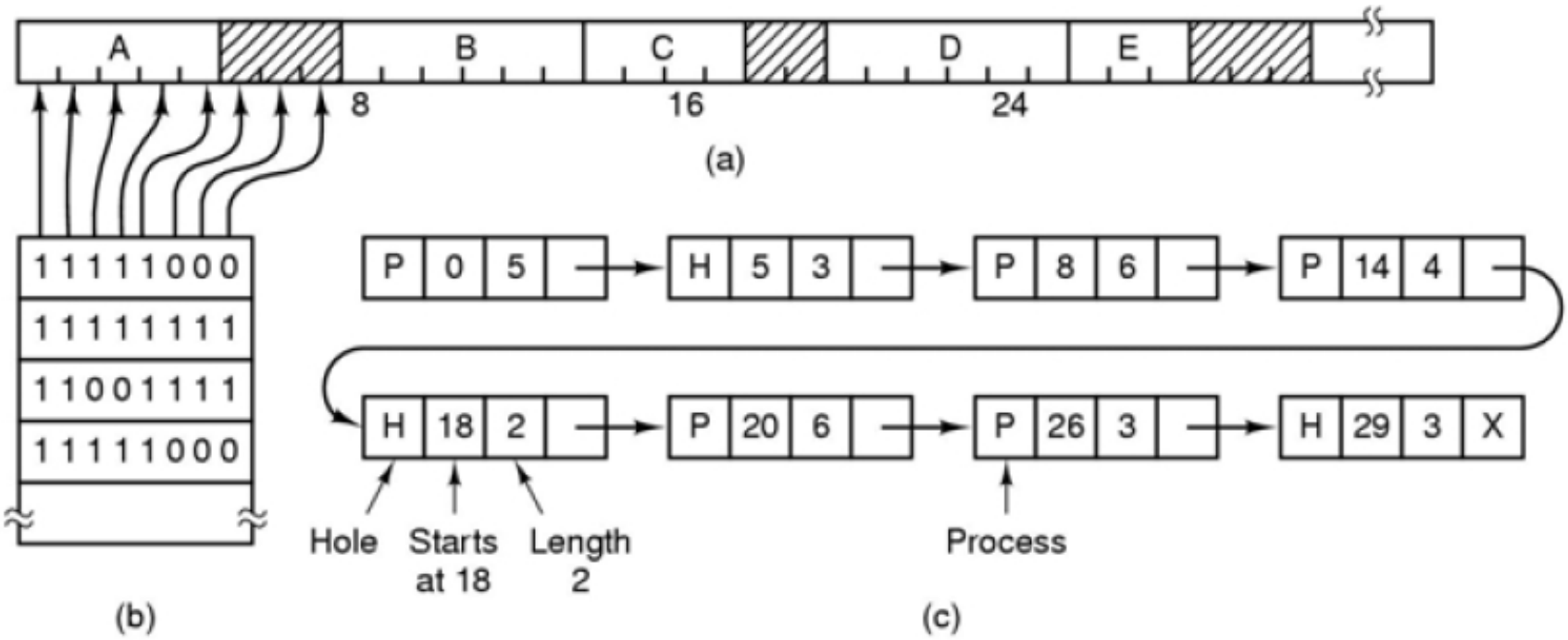
(Externe) Fragmentierung. Bezeichnet ungenutzten Speicher außerhalb des Adressraums der Prozesses.

- Z.B. bei variablen Partitionen: Summe der Löcher könnte dann z.B. ausreichen, um eine Speicheranforderung zu erfüllen, da Speicher aber zusammen-hängend benötigt wird, können Löcher nicht genutzt werden.

Freispeicherverwaltungsstrategien sollten Fragmentierung gering halten.

Freispeicherverwaltung (1/2)

Freispeicherverwaltung mit **Bitmaps** oder **verketteten Listen**.



Freispeicherverwaltung (2/2)

Freispeicherverwaltung mit Bitmaps.

- Einfacher Mechanismus.
- Bitmap muss komplett geladen und durchsucht werden, wenn ein Speicherbereich einer bestimmten Größe gesucht wird.

Freispeicherverwaltung mit Listen.

- Verwaltung und Suche sind einfacher (einfache Listenoperationen).

Speicherbelegungsstrategien

First Fit. Der erste ausreichend große freie Speicherbereich wird belegt.

Next Fit. Wie First Fit, aber Suche beginnt an der Stelle, wo zuletzt ein passendes Speicherbereich gefunden wurde. (D.h. wenn beim letzten Mal das Loch nicht vollständig geschlossen, sondern lediglich verkleinert wurde, wird bei diesem Loch gestartet.)

Best Fit. Es wird der kleinste freie Speicherbereich belegt, der die Anforderung noch erfüllen kann.

Worst Fit. Es wird der größte freie Speicherbereich belegt.

Quick Fit. Algorithmus hält Listen von Löchern mit gebräuchlichen Größen. Findet Löcher gebräuchlicher Größe sehr schnell, hat jedoch die gleichen Nachteile wie alle Algorithmen, die nach Lochgröße sortieren.

Bewertung.

Best Fit und Worst Fit sind in der Praxis schlechter als First Fit und Next Fit.

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Virtueller Speicher

Probleme

- Swapping lagert den vollständige Adressraum eines Prozess in den Speicher ein, obwohl bis zum nächsten Auslagern in der Regel nur ein Teilbereich benötigt wird.
- Adressraum eines Prozesses kann größer sein als der verfügbare Speicher.

Lösungen

- **Overlays,** manuelle (= historische) Lösung.
  - Der Programmierer zerlegt das Programm in Teile (*overlays*).
  - Wenn ein Teil fertig ist, ruft es das nächste auf, dadurch entstehen kleine Prozesse, die dynamisch ein- und ausgelagert werden.
- **Virtueller Speicher,** automatische Lösung.
  - Nimmt dem Programmierer das Aufteilens des Programms in einzelne Teile ab.
  - Programme benutzen **virtuelle Adressblöcke**, die auf **physikalische Adressblöcke** abgebildet werden.
  - Die am häufigsten verwendete Technik für virtuellen Speicher ist **Paging**.

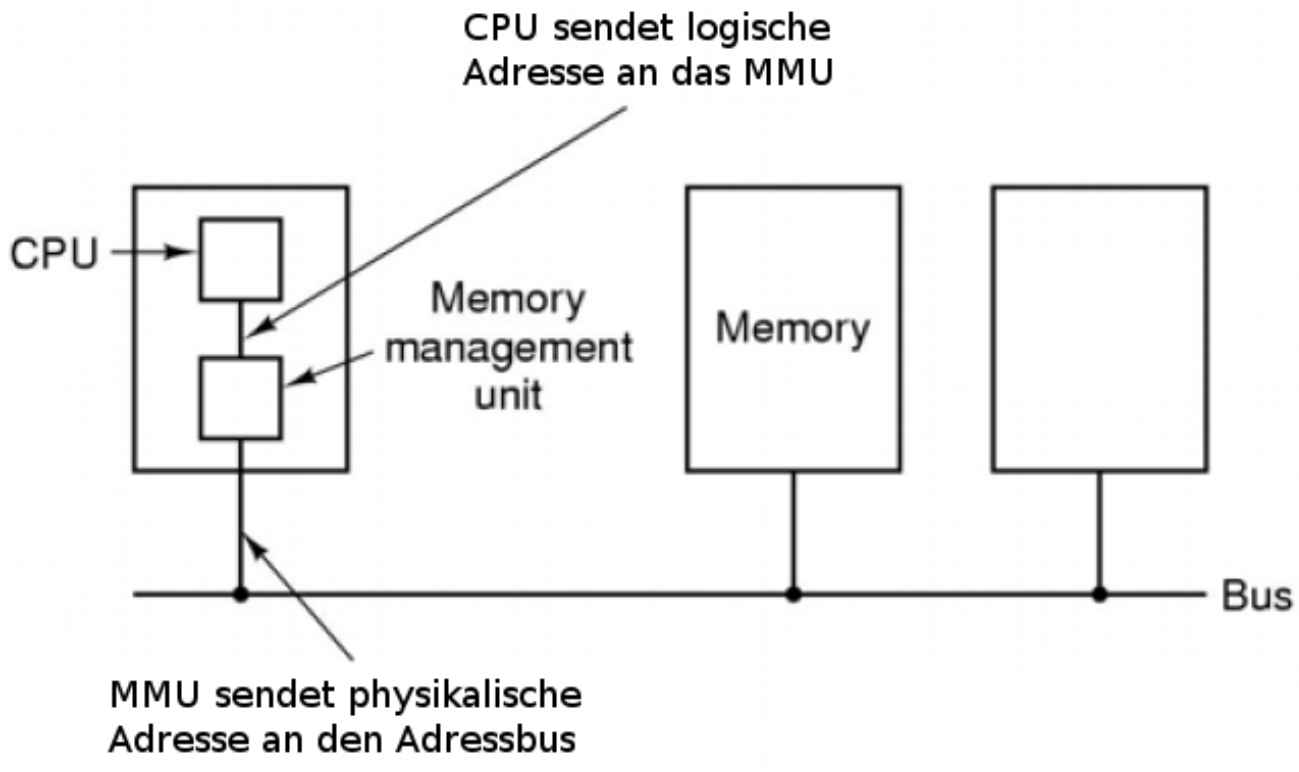
Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Memory Management Unit (MMU)

Eine programmierbare **Memory Management Unit (MMU)** ist zwischen Adressbus und Speicher geschaltet.

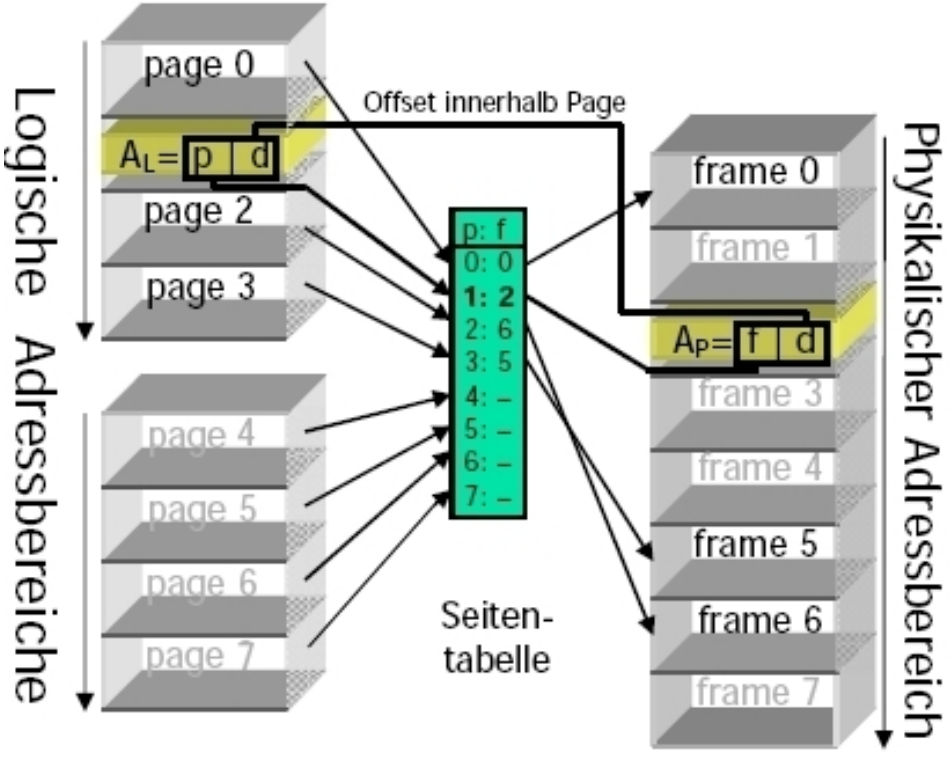


Die MMU bildet logische auf physikalische Adressen ab, indem die virtuelle auf physikalische Adressblöcke abbildet.

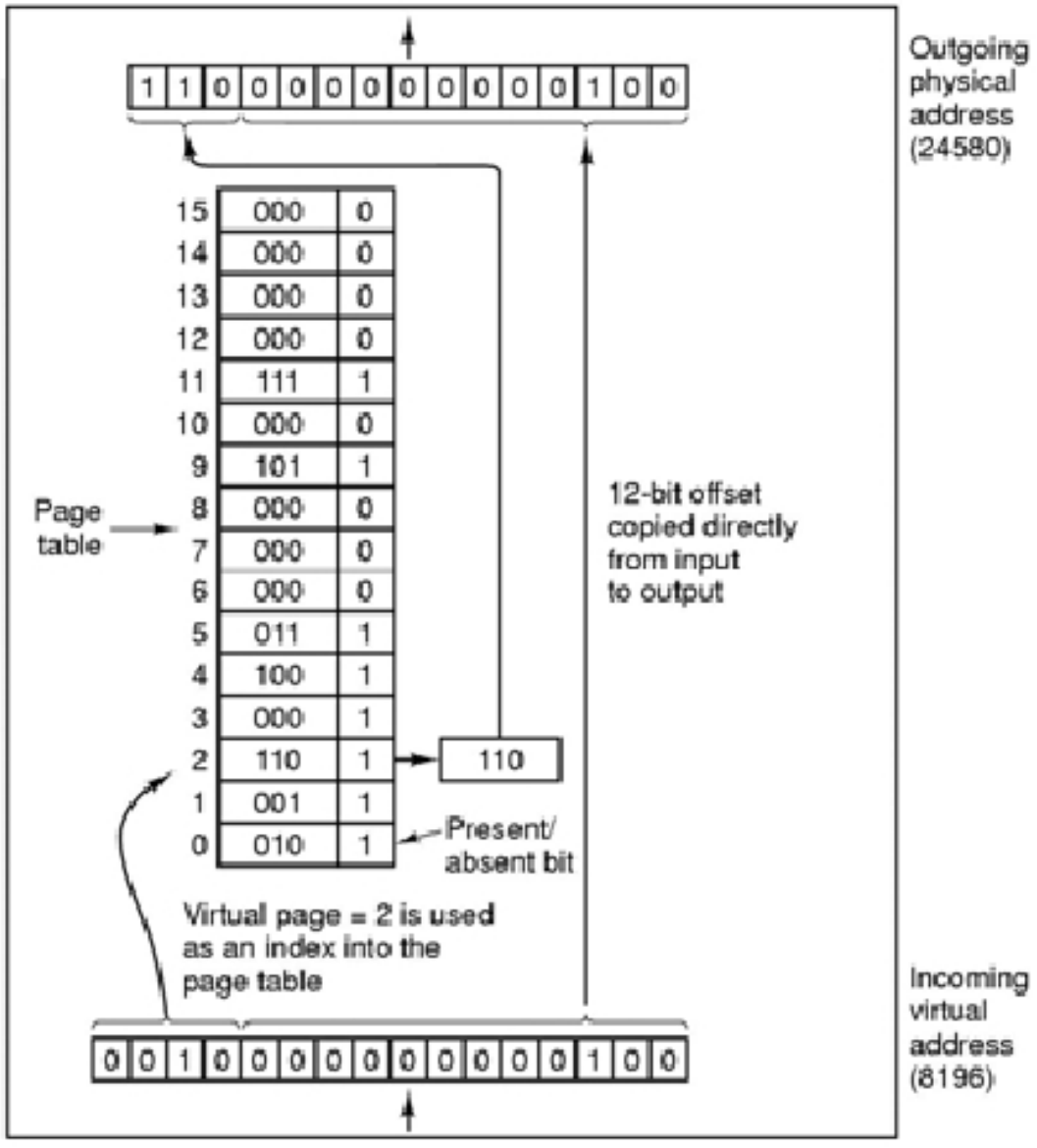
Prinzip

Abbildung einer logischen (A_L) auf eine physikalische Adresse (A_P) durch eine MMU.

- Der logische Adressbereich und der physikalische Speicher werden in virtuellen Adressblöcke gleicher Größe unterteilt.
- Ein virtueller Adressblöcke im logischen Adressbereich wird als **Seite** (page) und im physikalischen Speicher als **Rahmen** (frame) bezeichnet.
- Die Zuordnung von Seiten zu Rahmen erfolgt durch eine **Seitentabelle**.



Innenleben einer MMU

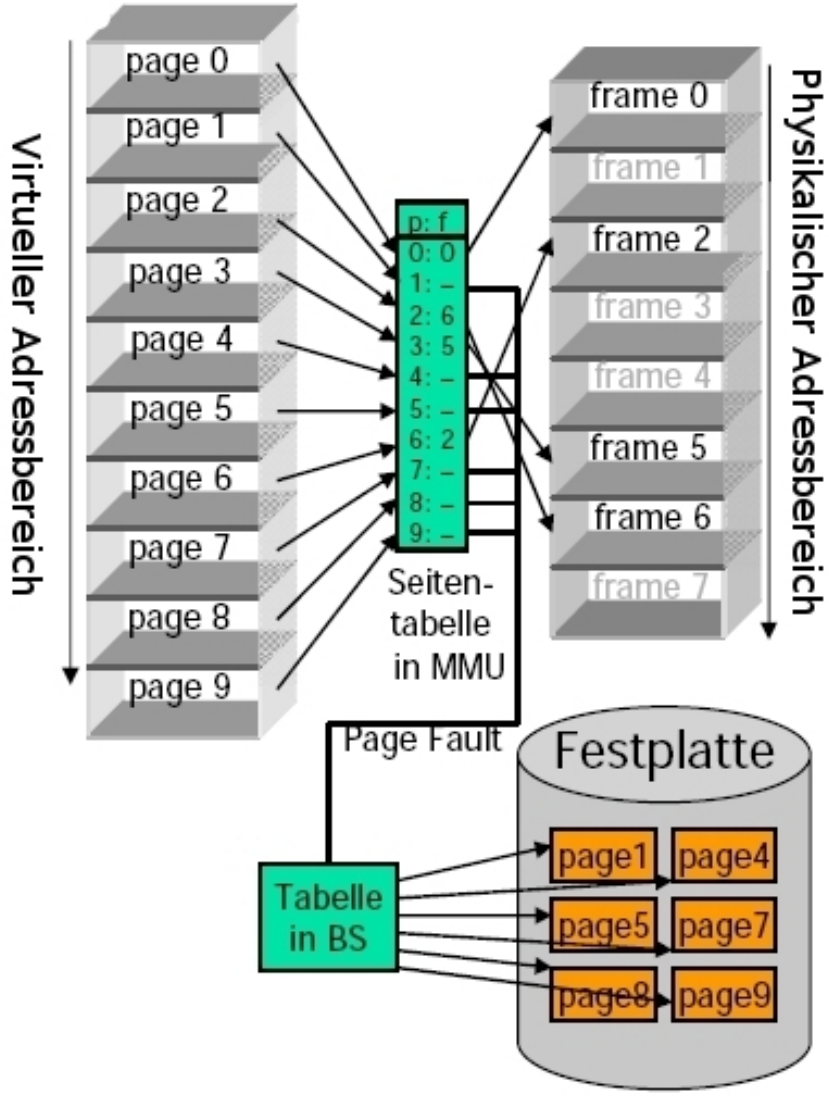


Paging

Ein-/Auslagern einzelner **Seiten** (*pages*) aus dem Hintergrundspeicher (Festplatte) in **Rahmen** (*frames*) im Hauptspeicher, unabhängig von der Prozesszugehörigkeit.

Einlagern in einen Rahmen erst, wenn auf eine Adresse in der ausgelagten Seite zugegriffen werden soll. Löst Seitenfehler (*page fault*) aus.

Auslagern aus einem Rahmen erst, wenn Hauptspeicher für eine neue oder einzulagernde Seite benötigt wird.

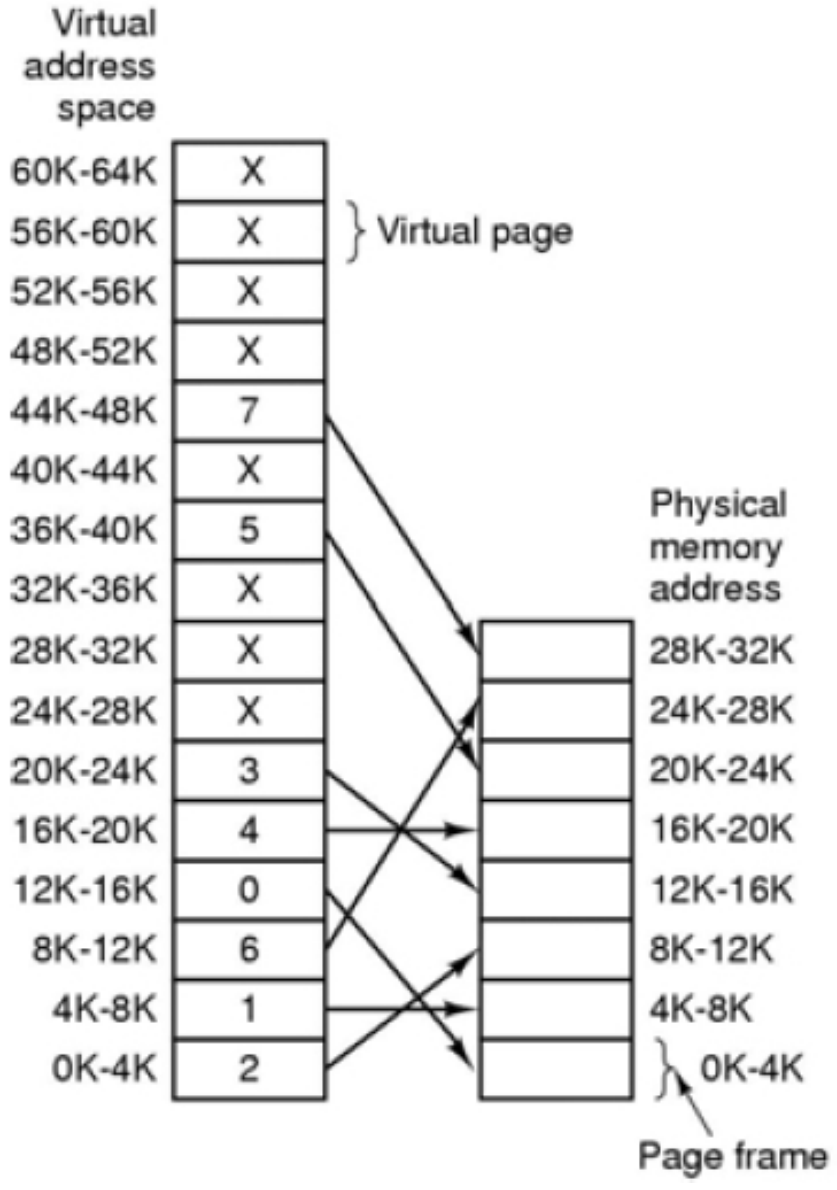


Beispiel

Rechner mit 16 Bit-Adressen, d.h. maximal 64KB adressierbar.

Rechner ist jedoch nur mit 32KB Speicher ausgebaut.

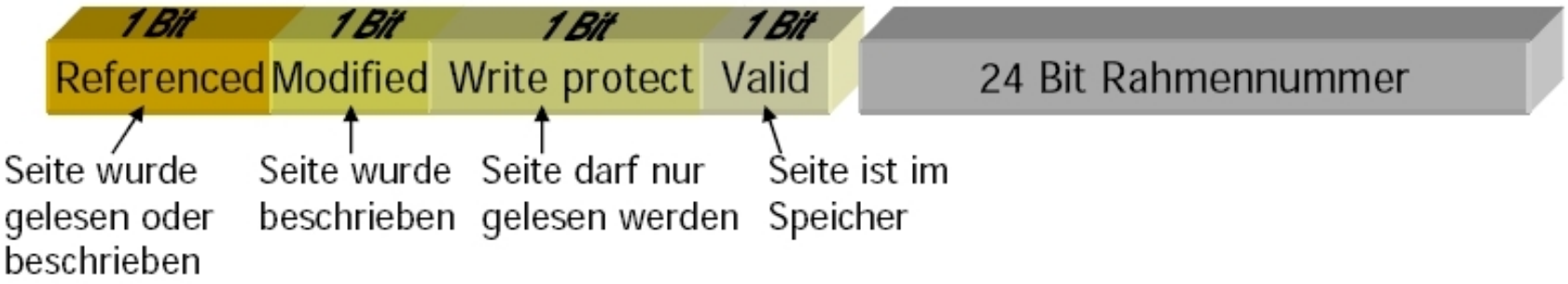
Nur einzelne Seiten des virtuellen Adressraums werden in Rahmen im physikalischen Speicher gehalten.



Seitentabelleneintrag

Heutige MMUs erlauben Seitengrößen von 256B bis 16MB.

Aufbau eines Seitentabelleneintrags (bei 32 Bit Adressraum und 256 Byte Seitengröße)



Zugriffszeit verdoppelt sich durch Zugriff auf Seitentabelle:

1. Zugriff auf Seitentabelle im RAM,
2. Zugriff auf eigentliche Speicheradresse.

Lösung. Ein **Translation Lookaside Buffer** (Seitentabellen-Cache) in MMU.

Beispiel

Beispiel. Größe der Seitentabelle

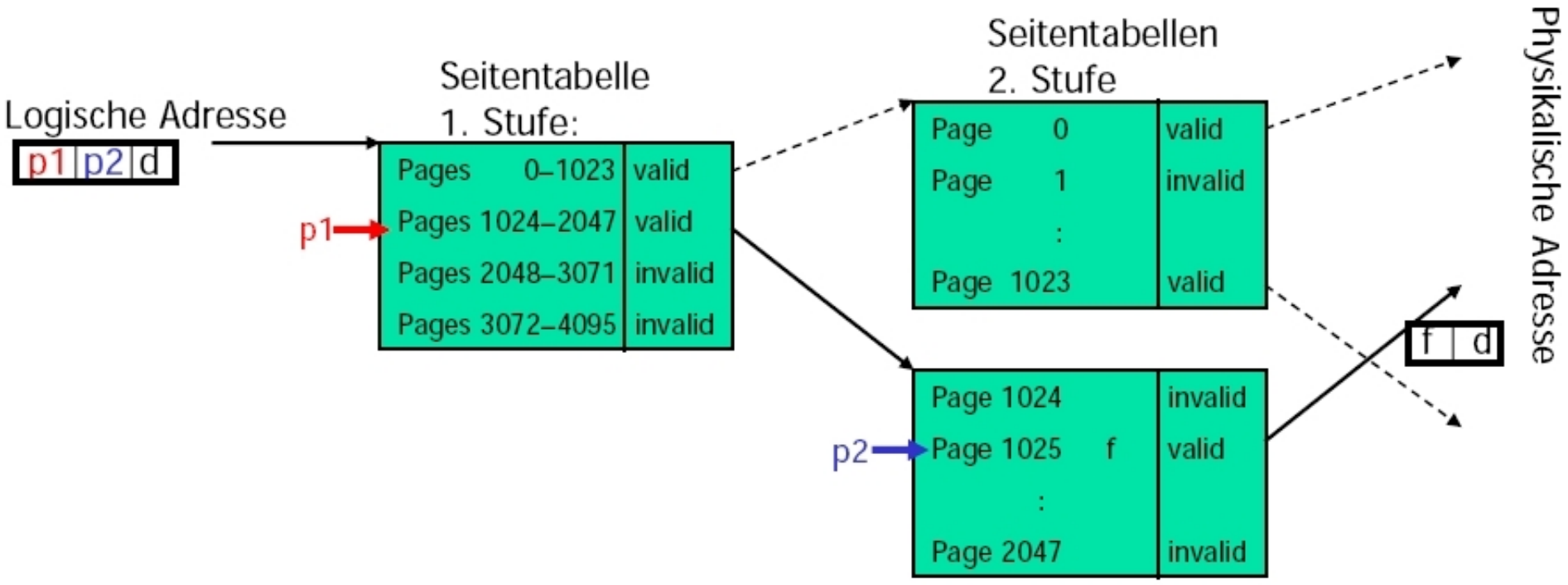
- 4KB Seitengröße, 32 Bit logischer Adressraum.
- Seitentabelle umfasst $2^{32}/2^{12} = 2^{20} \approx 1$ Million Seiten/Einträge.
- Pro Seitentabelleneintrag 4 Byte, d.h. ≈ 4 MB für die Seitentabelle.

Derart große Seitentabellen können nicht vollständig in der MMU gecached werden.

Der logischer Adressraum wird jedoch selten vollständig genutzt, sondern enthält viele ungenutzte Bereiche, für deren Seiten allesamt das Valid-Bit auf invalid gesetzt ist.

Mehrstufige Seitentabellen realisieren das Zusammenfassen von Seitentabellenabschnitten.

Mehrstufige Seitentabellen



Seiten werden zu **Blöcken** zusammengefasst.

Bis 4-stufige Seitentabellen werden in MMUs implementiert.

Befindet sich ein Seitentabelleneintrag nicht im Translation Lookaside Buffer, erhöht sich die Speicherzugriffszeit, um die Anzahl der Stufen, die durchlaufen werden müssen.

Translation Lookaside Buffer lohnt sich wegen der **Lokalität** der Zugriffe.

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Seitenersetzung

Beim Zugriff auf eine Seite, die nicht im Hauptspeicher ist tritt ein **Seitenfehler** (*page fault*) auf, der vom Betriebssystem behandelt wird.

Fall 1

Im Hauptspeicher ist ein freier Rahmen vorhanden.

- Die benötigte Seite wird in den freien Rahmen einlagert.
- Die Seitentabelle wird aktualisiert.

Fall 2

Alle Rahmen im Hauptspeicher sind belegt.

- Es wird ein Rahmen zur Ersetzung ausgesucht.
- Wurde der Inhalt des ausgesuchte Rahmen geändert (*modified bit*), wird er ausgelagt.
- Die angeforderte Seite wird in den ausgesuchten Rahmen einlagert.
- Die Seitentabelle wird aktualisiert.

Seitenersetzungsstrategien

Welche Rahmen ersetzen?

- Seitenersetzungsstrategien sind nötig, die mit möglichst wenig Ersetzungen (Festplattenzugriffen) auskommen.

Ähnliche Probleme existieren auch in anderen Bereichen der Informatik.

- Aktualisierung von Cache-Speichern.
- Caching von WWW-Seiten von Web-Servern.

Seitenersetzungstrategien lassen sich daher auf verschiedene Probleme übertragen

Optimale Seitenersetzung

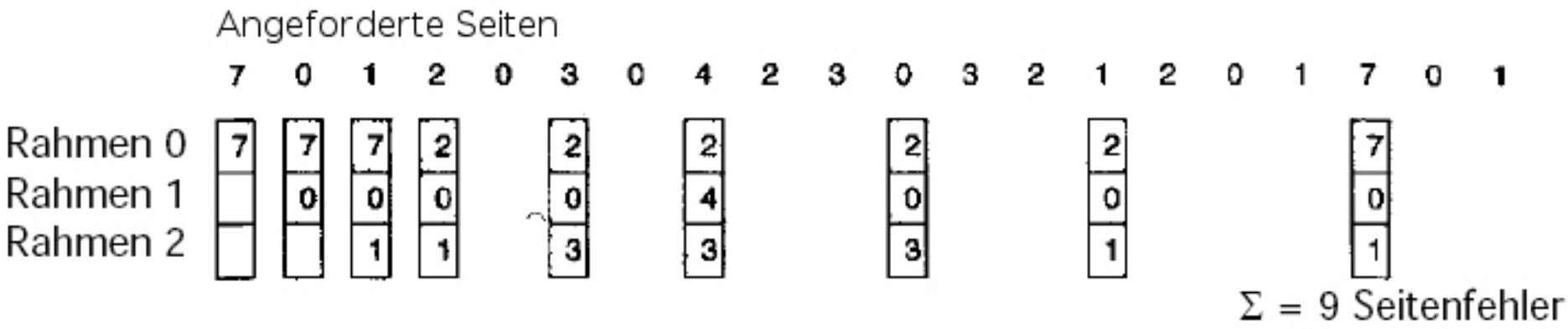
Die **Optimale Seitenersetzung** ersetzt die Seite, die in Zukunft am **längsten nicht benutzt** wird.

Diese Strategie kommt mit den wenigsten Ersetzungen (= Seitenfehlern) aus.

Optimale Seitenersetzung ist nicht implementierbar, da man in die Zukunft schauen muss. Aber ein guter Maßstab zum Vergleich von anderen Strategien.

Vergleich funktioniert nur mit genau einem Programm mit vorgegebenen Eingabedaten.

Optimale Seitenersetzung, Beispiel



Fist in, First out (FIFO)

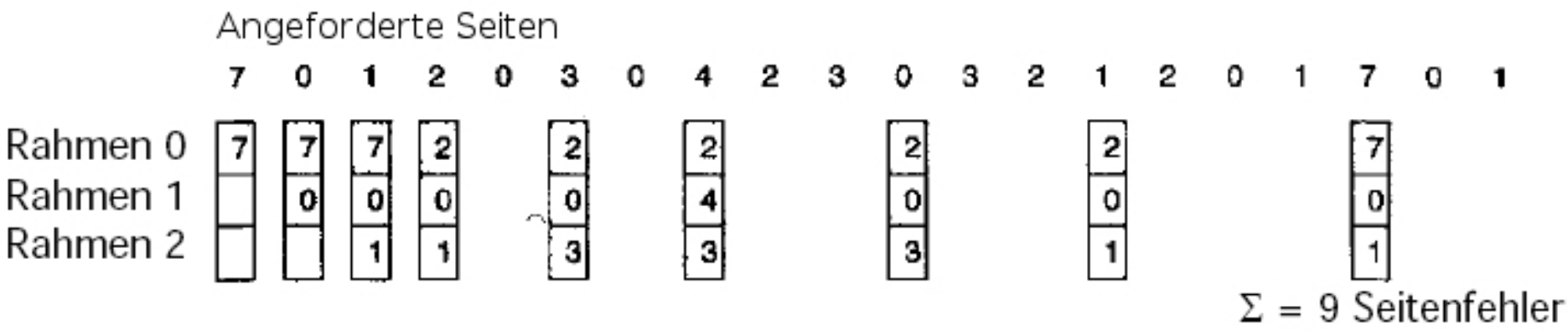
First-In, First-Out verdrängt immer die älteste Seite.

Vorteil. Einfache Implementierung durch FIFO- Warteschlange (Länge = Anzahl Rahmen). Seite an Spitze der Warteschlange wird verdrängt.

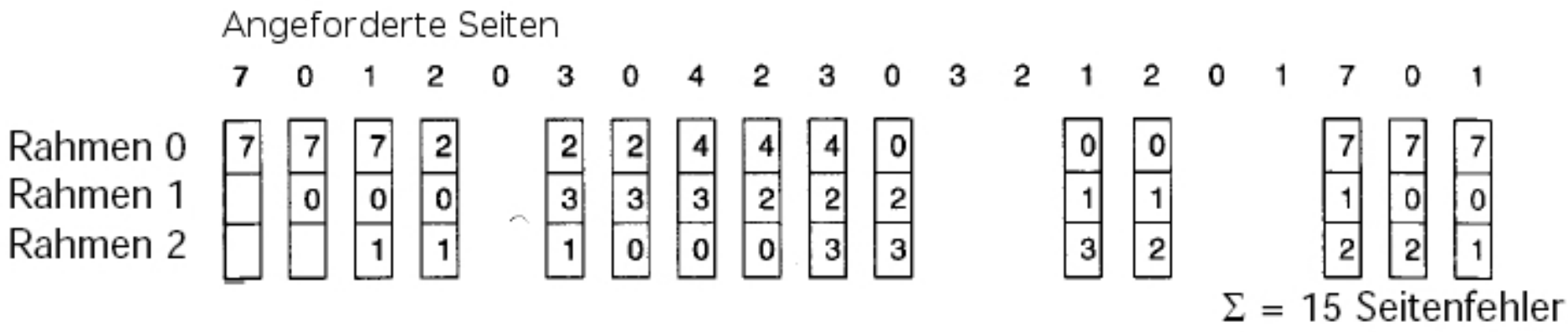
Nachteil. Die älteste Seite kann diejenige sein, die am häufigsten benötigt wird.

FIFO, Beispiel

Optimal



FIFO



Not Recently Used (NRU)

Bei **NRU** werden Seiten **gewichtet**, danach ob sie verändert, gelesen oder nicht benutzt wurden (*referenced/modified bit*).

Verfahren

- Wenn Seite ersetzt werden soll, klassifiziere Seiten
  1. nicht referenziert, nicht modifiziert
  2. nicht referenziert, modifiziert
  3. referenziert, nicht modifiziert
  4. referenziert, modifiziert.
- Verdränge zufällig eine Seite aus der kleinsten nicht-leeren Klasse.

Bemerkung

- Das *referenced bit* wird periodisches durch Timerunterbrechungen gelöscht.
- Das *modified bit* darf nicht gelöscht werden, da es benötigt wird, um zu Entscheiden, ob eine Seite vor der Ersetzung auf Platte geschrieben werden muss.

Second Chance

Second Chance ist eine Variante von FIFO um zu verhindern, das häufig benutzte Seiten ausgelagert werden.

Verfahren überprüft das *referenced bit* der ältesten Seite.

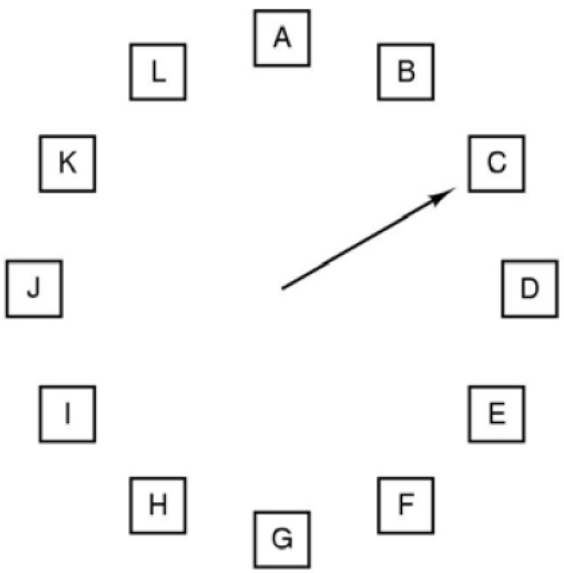
- *Referenced bit* **nicht** gesetzt, dann wird die Seite ausgelagert (Seite alt und unbenutzt).
- *Referenced bit* gesetzt, dann wird es zurückgesetzt und die Seite wird an den Anfang der FIFO-Liste eingefügt (Seite wird wie neu eingelagert behandelt).

Algorithmus degeneriert zu FIFO, wenn die *referenced bits* für alle Seiten gesetzt sind, terminiert aber, da schrittweise alle *referenced bits* auf 0 gesetzt werden, bis wieder die zuerst betrachtete Seite am Ende der Liste steht und dann ausgelagert wird.

Clock-Algorithmus

Der **Clock-Algorithmus** unterscheidet sich nur in der Implementierung von Second Chance.

Verfahren
Tritt ein Seitenfehler auf wird die Seite am Zeiger untersucht



- Wurde die Seite benutzt (*referenced bit*), wird der Status der Seite auf unbenutzt gesetzt, der Zeiger weiterbewegt und die nächste Seite wird untersucht.

  Wurde die Seite verändert (*modified bit*), bleibt dieser Status erhalten.
- Wurde die Seite (seit dem letzten Durchlauf) nicht benutzt wird dieser Rahmen für die einzulagernde Seite benutzt.

  Anhängig vom *modified bit* wird die Seite ausgelagert oder einfach überschrieben.

Least Recently Used (LRU)

LRU ist eine gute Annäherung an Optimal.

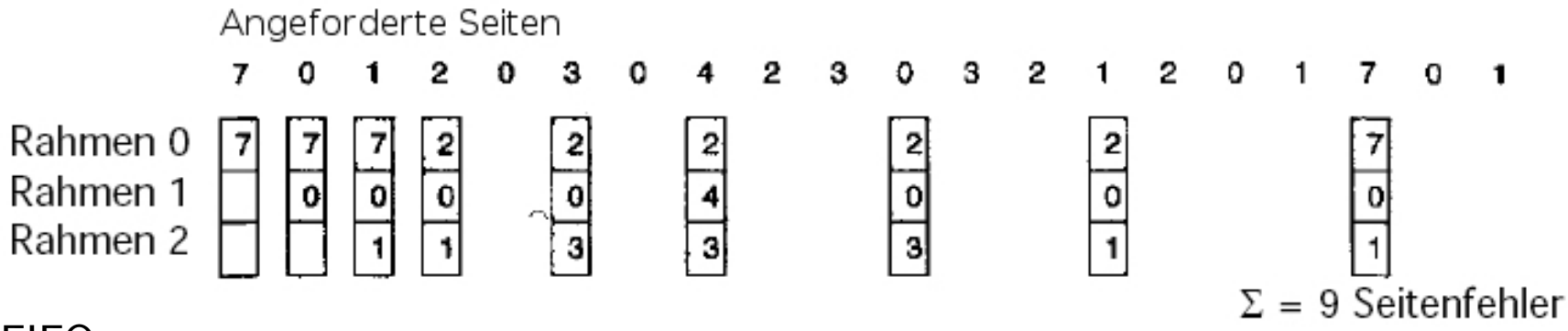
Annahme
Seiten, die zuletzt häufig benutzt wurden, werden auch in Zukunft häufig benutzt. (**Lokalitätsprinzip**).

Verfahren
Bei einem Seitenfehler wird die Seite entfernt, die am längsten unbenutzt ist.

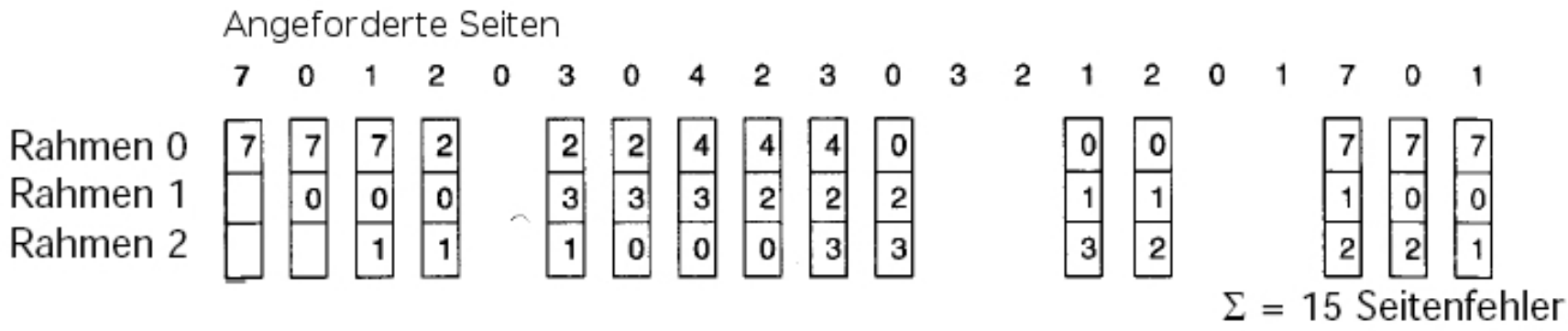
Problem
Die Implementierung ist schwierig und ineffizient, da die Liste der Seiten nach Zugriffszeitpunkt sortiert und die ganze Liste nach jedem Speicherzugriff aktualisiert werden muss.

LRU, Beispiel

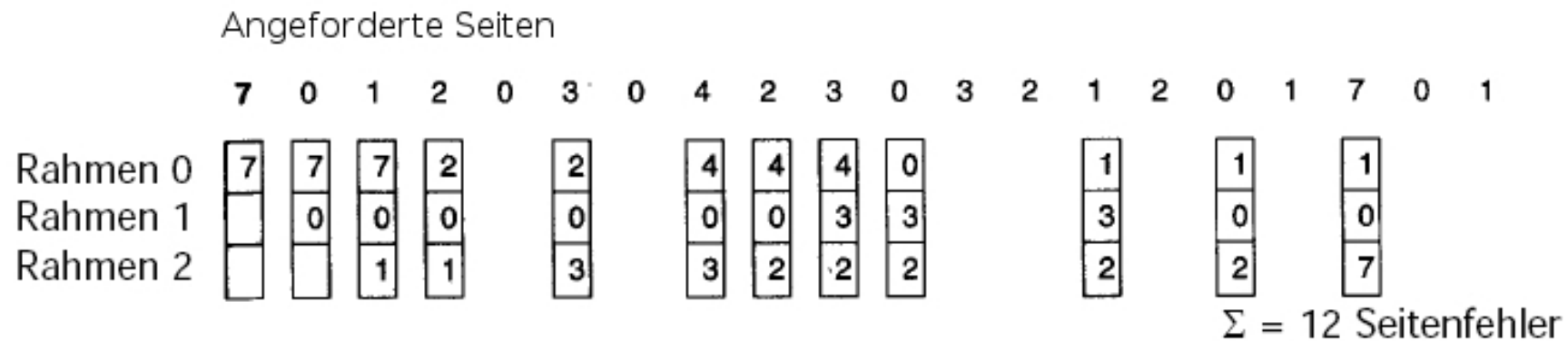
Optimal



FIFO



LRU



Not Frequently Used (NFU)

NFU ist eine effiziente (Software-) Annäherung an LRU.

- Jeder Seite bekommt einen **Zähler**.
- **Zähle** periodisch (z.B. durch Timerunterbrechungen oder bei Kontextwechsel) **Zugriffe** für jede Seite, indem der Inhalt des jeweiligen *referenced bits* auf dessen Zähler addiert wird.
- Setze *referenced bits* anschließend zurück.
- Verdränge Seite mit niedrigstem Zählerstand.

Problem
Zeitliche Änderungen des Zugriffsmusters werden schlecht berücksichtigt, da Zähler nicht **altern**.

Aging NFU, stärkere Gewichtung des aktuellen *referenced-bit* bei jeder Periode.

- Dividiere Zähler durch 2.
- Ersetze höchstwertigstes Bit des Zählers (=0) durch das aktuelle *referenced-bit*.

Aging NFU, Beispiel

Referenced-Bits $r_0 \ldots r_5$

	Periode 1	Periode 2	Periode 3	Periode 4	Periode 5
	101011	110010	110101	100010	011000

Zähler

Seite	Periode 1	Periode 2	Periode 3	Periode 4	Periode 5
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

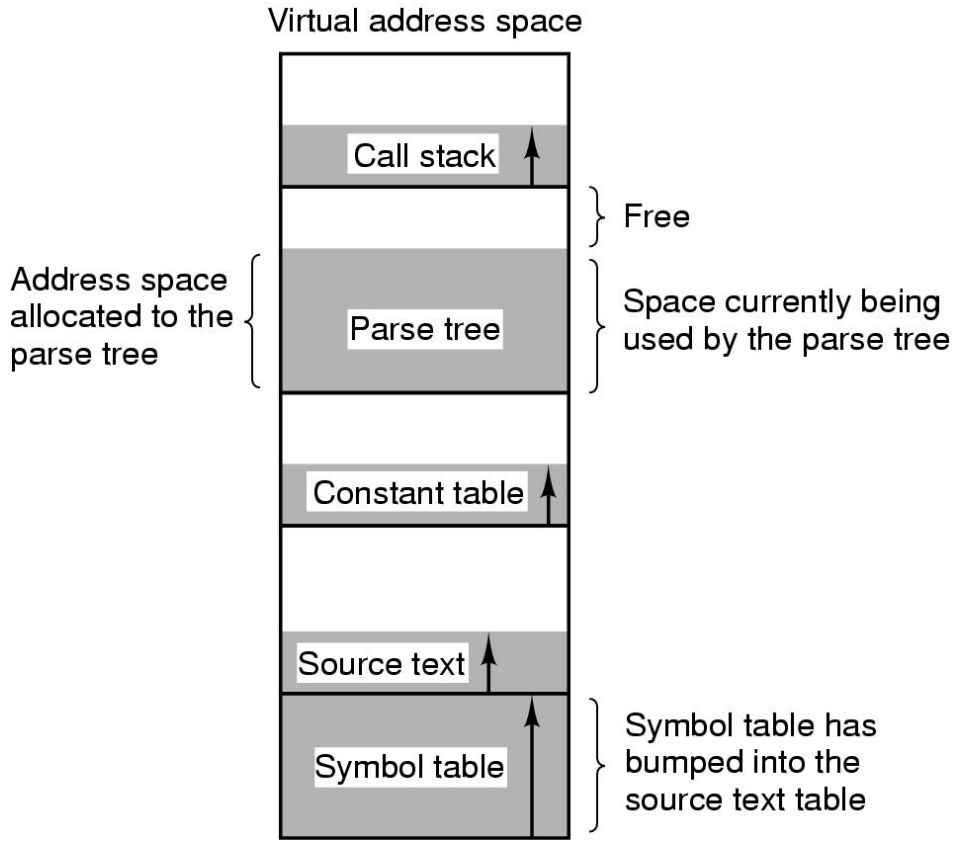
Einführung

Bisherige Betrachtung

Eindimensionaler Adressraum, in dem alle Programmtext, Daten, Stack im selben Adressraum abgelegt werden.

Problem

Was passiert, wenn die Daten, z.B. Tabellen, nach dem Anlegen und der fortschreitenden Übersetzung eines Programms zu groß werden und in einen anderen Bereich *hineinwachsen*?

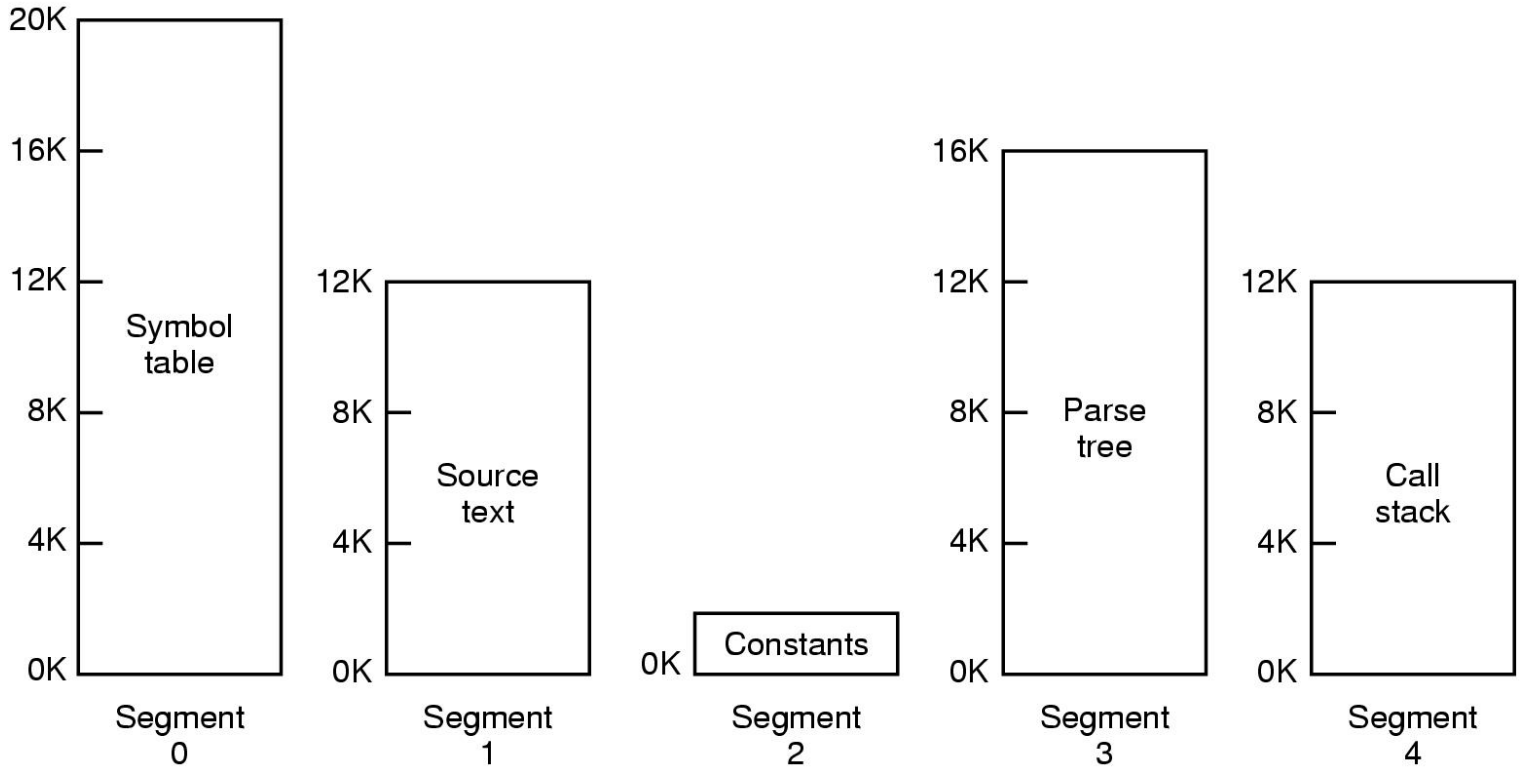


Lösung

Keinen eindimensionalen Adressraum benutzen, denn dann können die Probleme nur durch Modifikationen am Compiler gelöst werde und dass sollte man vermeiden.

Segmentierung

Durch **Segmentierung des Adressraums** werden jedem Programm mehrere Adressräume (Segmente) zur Verfügung gestellt, die es für die jeweils für eine Klasse von Informationen (Progammtext, Daten, Stack, etc.) verwenden werden.



Eine Adresse besteht dann aus Segment-Nummer und Adresse.

Vorteile

Vorteile

- Segmente können wachsen, Management durch das Betriebssystem.
- Der Zugriff auf Informationen wird wegen der jeweiligen Startadresse *0x00* stark vereinfacht
- Speicherschutz kann leicht realisiert werden.

Segmentierung und Paging

Segmentierung und Paging können gleichzeitig eingesetzt werden, um die Vorteile beider Ansätze zu kombinieren.

Zum Beispiel wird es sinnvoll sein, Segmente aufzuteilen, wenn sie nicht komplett in den Hauptspeicher passen.

Moderne Prozessorfamilien unterstützen meist beide Modelle, um für beliebige Betriebssysteme offen zu sein.

Beispiel

Ein Pentium besitzt insgesamt $16 \cdot 1024$ Segmente, wobei jedes bis zu 10^9 32-Bit-Worte speichern kann. Diese Segmentgröße ist relativ groß.

Inhalt

Speicherverwaltung
- Einführung
- Monoprogrammierung
- Multiprogrammierung
- Logische/Physikalische Adresse
- Swapping
- Virtueller Speicher
- Memory Management Unit (MMU)
- Seitenersetzung
  - Optimale Seitenersetzung
  - Fist in, First out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance
  - Clock-Algorithmus
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- Segmentierung
- Abschlussbemerkungen

Ein-/Ausgabe

Abschlussbemerkungen

In der Vorlesung nicht behandelt

- Analyse von Seitenersetzungsstrategien,
- Konkrete Implementierungsbeispiele.

Informationen dazu finden Sie in der einschlägigen Literatur (siehe Literaturliste).

Inhalt

Speicherverwaltung

Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Inhalt

Speicherverwaltung

Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Geräte/Schnittstellen

Es gibt eine Vielfalt von Ein-/Ausgabe-Geräte und Hardware-Schnittstellen.

Ein-/Ausgabe-Geräte

- Drucker
- Scanner
- Tastatur
- Maus
- Festplatte
- Solid State Drive
- CD/DVD/Bluray–Laufwerk/Brenner
- Streamer (Bandlaufwerk)
- Grafikkarte
- Netzwerk
- Modem
- ...

Hardware-Schnittstellen

- USB
- Parallelport
- Serieller Port
- IDE/SATA
- SCSI
- LAN/WLAN
- Bluetooth
- ...

Aufgaben der Betriebsystems

Verwaltung der Ein-/Ausgabe-Geräte durch Bereitstellung von **Gerätetreibern**.

- Prozessen soll einfacher, aber kontrolierten Zugriff auf die Ein-/Ausgabe-Geräte ermöglicht werden.
- Bei Multiprogramming, soll ein möglichst fairer Zugriff auf Ein-/Ausgabe-Geräte, die von mehreren Prozessen nebenläufig genutzt werden können, gewährleistet werden.
- Der Ein-/Ausgabe-Flaschenhals muss durch geignete Verfahren minimiert werden.

Beherschung der Vielfalt von Ein-/Ausgabe-Geräten und Hardware-Schnittstellen.

- Anwendungen sollen dich bei der Ein-/Ausgabe nicht mit der Geräte-Vielfalt auseinandersetzen müssen.
  Deshalb werden Geräte mit **ähnlicher Charakteristik** jeweils unter einer **einheitlichen Programmierschnittstelle** zusammengefasst.

Inhalt

Speicherverwaltung

Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Gerätecharakteristik

Zusammenfassen von Ein-/Ausgabe-Geräten mit ähnlicher Charakteristik.

Ein-/Ausgabe-Geräte sind **zeichenorientiert** oder **blockorientiert**.

- Ein Tastatur versendet einzelen Zeichen.
- Auf eine Festplatte kann mit einer Operation nur mindestens ein ganzer Sektor (= Block, z.B. 512 KByte) geschrieben werden.

Beispiel

Zeichenorientiert. Tastatur, Drucker, Modem, ...

Blockorientiert. Festplatte, CD/DVD, Bandlaufwerk, ...

Blockorientierte Ein-/Ausgabe-Geräte gestatten **wahlfreien Zugriff** oder **sequentiellen Zugriff**.

- Eine Festplatte kann Ihren Schreib-/Lesekopf über einem beliebigen Sektor positionieren.
- Beim einem Bandlaufwerk wird erst an eine feste Stelle vor- oder zurückgespult und dann müssen alle zwischen Ausgangs- und Zielposition befindlichen Blöcke aufgesucht werden.

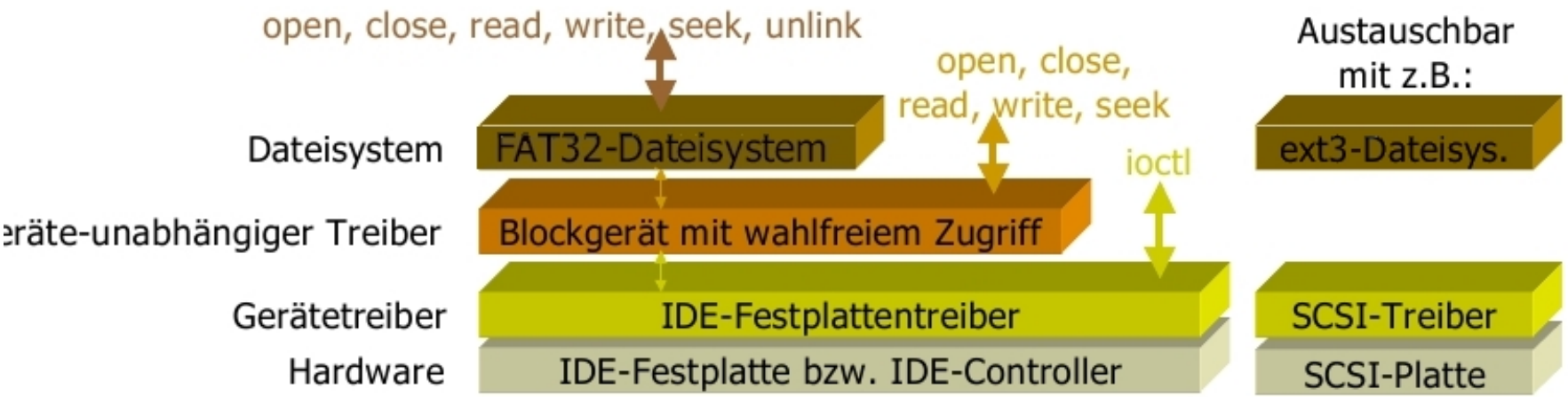
Treiberschichtung (1/2)

Schichtung von Abstraktionsebenen

- Geräte-unabhängige Treiber entsprechen allgemeiner Geräte-Charakteristik.
- Nur der geräte-abhängige Teil muss individuell implementiert werden.

Treiber der gleichen Schicht sind **austauschbar**, solange die Charakteristik der angrenzenden Schichten passt.

Beispiel. FAT32-Dateisystem braucht Blockgeräte mit wahlfreiem Zugriff.



Treiberschichtung (2/2)

Geräte-unabhängige Treiber, stellen eine allgemeine **hardware unabhängige** Infrastruktur zur Verfügung.

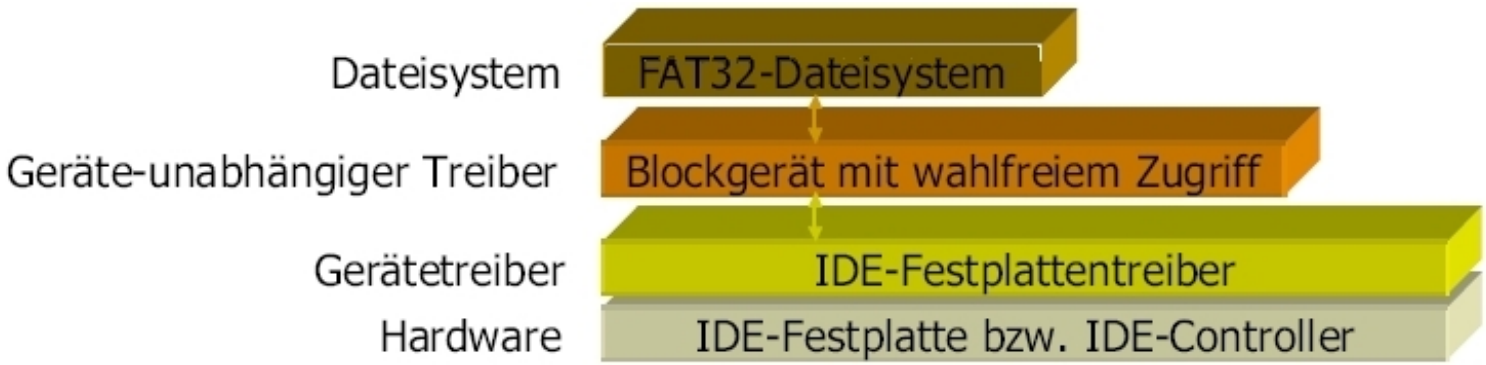
Beispiel, Blockgeräte mit wahlfreiem Zugriff.

- Puffern von Blöcken um auch zeichenweisen Zugriff zu ermöglichen.
- Verwaltung der aktuellen Schreib-/Lese-Position.

Geräte-(abhängige)-treiber, sind eine konkrete **gerätespezifische** Implementierung, die nicht (im darüberliegenden Treiber) hardware unabhängig realisiert werden kann.

Beispiel, IDE-Treiber.

- Erzeugen von IDE-Kommandos zum Lesen-/Schreiben von Blöcken, die vom Blockgerätetreiber verwendet werden.



Inhalt

Speicherverwaltung

Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Hardwareanbindung

Wie kommunizieren die Gerätetreiber (Software) mit den Ein-/Ausgabe-Geräten bzw. den zugehörigen Schnittstellen-Controllern (Hardware)?

Das Betriebssystem muss dafür sorgen, dass Daten aus dem Adressraum eines Prozesses zum Ein-/Ausgabe-Gerät werden und Daten vom Ein-/Ausgabe-Gerät im Adressraum eines Prozess hinterlegt werden.

Es gibt mehrere Möglichkeiten zur Anbindung des Ein-/Ausgabe-Geräts (Controllers) an den Prozessor bzw. den Hauptspeicher.

- **Programmed I/O (PIO)**
- **Memory-Mapped I/O**
- **Direct Memory Access (DMA)**

Ziel. Durch geeichnetes Scheduling wird der Ein-/Ausgabeflaschenhals minimiert.

Optimal. Arbeitet das Ein-/Ausgabe-Gerät (Prozess ist blockiert) kann der Prozessor eine anderen (bereiten) Prozess abarbeiten.

Programmed I/O (PIO) (1/2)

Der Prozessor (CPU) ist direkt an ein Register des Ein-/Ausgabe-Controllers angebunden.

Das Betriebssystem bzw. der Prozessor holt (liefert) die Daten wort-/byteweise vom (zum) Ein-/Ausgabe-Controller.

Hinterher schreibt (Vorher liest) der Prozessor die Daten in (aus) den Hauptspeicher.

- *Polling*. Der Gerätetreiber fragt regelmäßig beim Ein-/Ausgabe-Controller nach ob Daten übertragen werden müssen (Nachteil *busy waiting*).
- *Interrupts*. Der Ein-/Ausgabe-Controller generiert einen Interrupt, wenn ein neues Datum vorliegt. Erst dann holt der Geräteteiber das Datum vom Controller ab.

Vorteil. Die Umsetzung, insbesondere mit Polling, benötigt eine minimale Unterstützung von Seiten der Hardware.

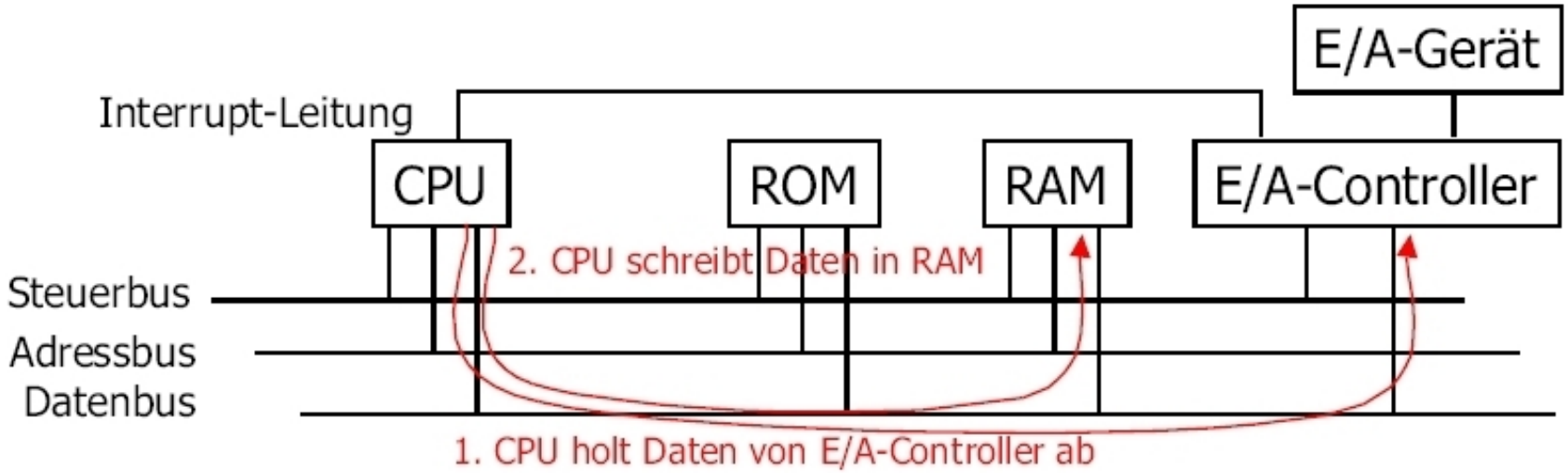
Nachteil. Für Multiprogramming ungünstig, da das Übertragung der Daten Rechenzeit benötigt, d.h. der Prozessor ist belegt und kann während der Datenübertragung keinen anderen Prozess abarbeiten.

Beispiel. Parallele Schnittstelle (IEEE-1284, Drucker-Parallelport)

Programmed I/O (PIO) (2/2)

Vereinfachtes Bespiel

`read`-Kommando



Memory-Mapped I/O (1/3)

Der Ein-/Ausgabe-Controller hat **eigenen Speicher**, der in den Adressbereich des Prozessors **eingeblendet** wird.

Beispiel. Grafikkarte

Vorteil.

- Wenig Rechenzeit intensiv. Prozessor kann arbeiten während der Controller selbstständig auf die Daten im (eigenen) Speicher zugreift.
- Ein Prozess kann auf den (eingeblendeten) Speicher, d.h. auf das Ein-/Ausgabegerät, mit den üblichen Speicherzugriffsroutinen zugreifen. Es werden keine besonderen Befehle benötigt.

Nachteil. Speicher der von zwei Seiten gleichzeitig genutzt wird (Prozessor und Controller) ist entweder **teurer** (dual-ported-RAM) oder **langsamer** (shared memory) als herkömmlicher Speicher.

Memory-Mapped I/O (2/3)

Memory-Mapped I/O nutzt aus, dass in der Regel weniger physikalischer Speicher vorhanden ist, als mit den vom Betriebssystem für die Adressierung vorgesehenen Adressengröße (Breite der Adressregister/des Adressbus) adressiert werden kann.

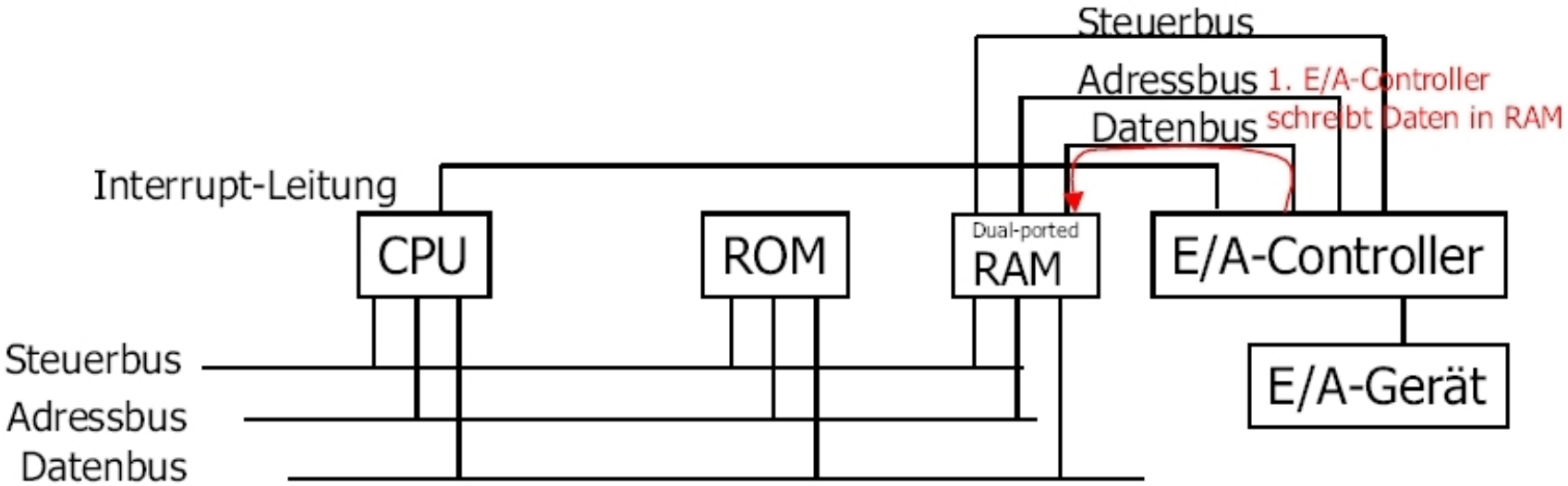
Beispiel.

Ein 32-Bit Betriebssystem benutzt 32-Bit Adressen, d.h. es können 2^{32} Byte (Speicherstellen), also 4 GByte, adressiert werden. Das war lange unvorstellbar viel Speicher und konnte auf PC-Mainboards auch nicht verbaut werden.
Die freien Adressen werden benutzt um mit Memory-Mapped I/O z.B. die Grafikkarte anzusprechen. Deshalb sind heute 4 GB mit einem 32-Bit Betriebssystem nicht vollständig nutzbar.

Memory-Mapped I/O (3/3)

Vereinfachtes Bespiel

`read`-Kommando



Direct Memory Access (DMA) (1/2)

Bei DMA kann der Ein-/Ausgabe-Controller selbstständig Daten an Adressen im Hauptspeicher schreiben bzw. daraus lesen.

Beispiel. Festplattencontroller

Vorteil. Wenig Rechenzeit intensiv. Prozessor kann arbeiten während der Controller selbstständig auf die Daten im Speicher zugreift.

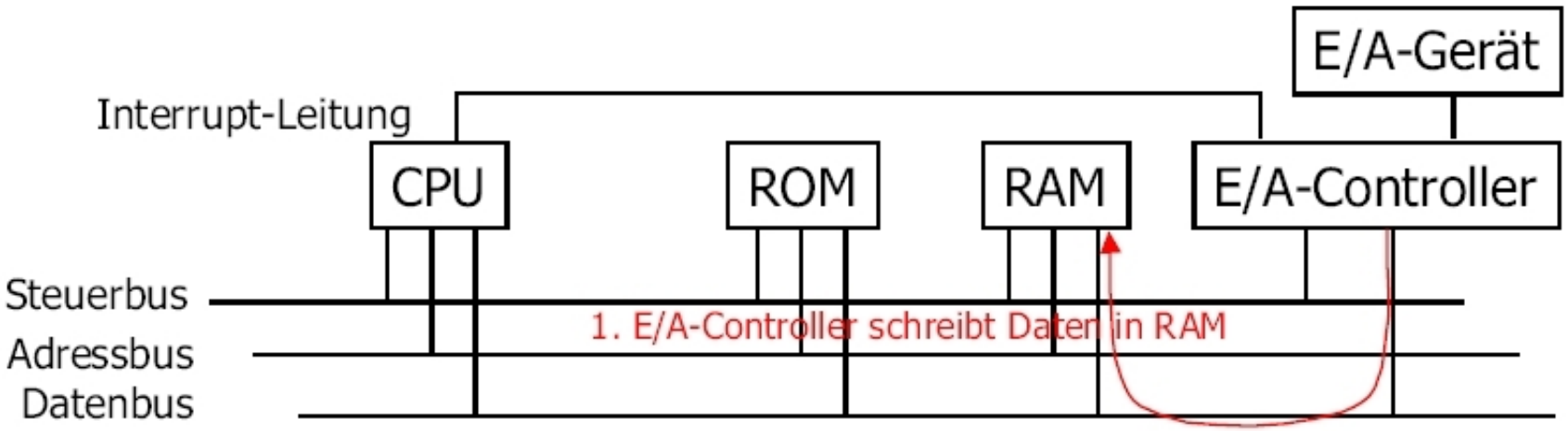
Nachteil.

- DMA bremst Zugriff des Prozessors auf den Speicher aus (vergleiche shared memory).
- Zugehörige Speicherbereiche müssen vor anderer Nutzung oder Auslagerung (z.B. durch Swapping oder Paging) geschützt werden.

Direct Memory Access (DMA) (2/2)

Vereinfachtes Bespiel

`read`-Kommando



Inhalt

Speicherverwaltung

Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Ein-/Ausgabe-Flaschenhals

Zugriffe auf Ein-/Ausgabe-Geräte stellen im Vergleich zur Prozessor- und Speichergeschwindigkeit einen **Flaschenhals** dar, d.h. Prozesse müssen (lange) warten, bis das Betriebssystem die Ein-/Ausgabe erledigt hat.

Lösungsansätze zur Flaschenhalsminimierung

- **Prozess-Scheduling.** Auf Ein-/Ausgabe wartende Prozesse sind blockiert, währenddessen kann der Prozessor andere Prozesse bearbeiten.

  Beispiel. Funktioniert bei DMA-basierter Ein-/Ausgabe.
- **Pufferung** (Caching). Der Prozess kann weiterarbeiten ohne auf das **Ausgabe**-Gerät zu warten.

  Beispiel. Drucker-Warteschlange

Bemerkung.
Das wichtigste Ein-/Ausgabe-Geräte und damit der wichtigste Flaschenhals ist der Hintergrundspeicher (Festplatte).

Pufferung zur Flaschenhalsminimierung

Der Festplatten-Controller haben eigenen Speicher zur Pufferung (Festplattencache).

Das Betriebssystem verwaltet davon unabhängig einen Puffer im Hauptspeichers. Dieser Speicher steht den Prozessen nicht zur Verfügung. Wird das z.B. durch virtuellen Speicher und Paging ausgeglichen steigen die Zugriffe auf den Hintergrundspeicher, d.h. ein vernünftiges Verhältnis muss eingehalten werden.

Puffern für Lesezugriffe

- Puffert Sektoren, die vorher schon einmal gelesen oder beschrieben wurden.

Puffern von Schreibzugriffen (verzögert schreiben)

- Mehrfaches Schreiben an dieselbe Position zusammenpassen.
- Prozesse (auch des Betriebssystems) können weiterarbeiten ohne darauf zu warten, dass das Ausgabe-Gerät Daten entgegennimmt.
- Bei Stromausfall/Systemabsturz kann es zu Verlust/Inkonsitenz von Daten kommen.

Kopf-Scheduling zur Flaschenhalsminimierung

Das Ziel ist aufwendige **Kopfbewegungen** zu **minimieren**.

Beobachtung
In einer Flaschenhalssituation stauen sich Festplattenzugriffe beim Treiber auf.

Idee. Die aufgestauten Zugriffe (d.h. Kopfpositionierungen) werden **umsortiert**, sodass zu deren Abarbeitung möglichst wenig Kopfbewegung nötig ist.

Problem. Das Umsortieren ist nur möglich, wenn es **keine Abhängigkeiten** zwischen den zu übertragenden Daten gibt.

Beispiel
Sei der Wert in `sektor42` ungleich dem Wert von `B`.

- `A=read(sektor42); write(sektor42, B); C=read(sektor42);` daraus folgt A≠B=C
- `A=read(sektor42); C=read(sektor42); write(sektor42, B);` daraus folgt A=C≠B

Lösung. Zugriffe auf **gleichen** Sektor untereinander nicht umsortieren.

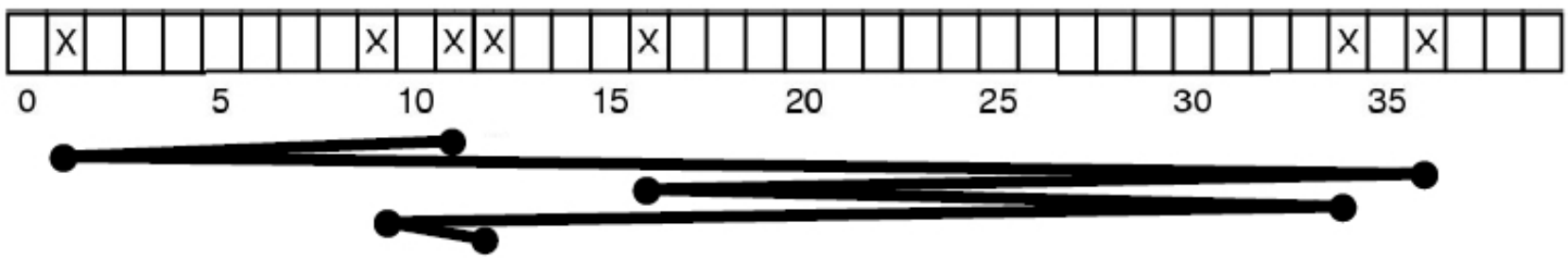
Bemerkung. Diese Problematik ist in der Praxis nicht existent, da das Betriebssystem die Festplattenzugriffe puffert.

First Come, First Served (FCFS)

Zugriffe werden in der Reihenfolge des Eintreffens bearbeitet, d.h. kein Umsortieren.

Beispiel

Eintreffende Zugriffe: 11, 1, 36, 16, 34, 9, 12



Zurückgelegte Wegstrecke des Kopfes: 10+35+20+18+25+3=111

Vorteil. Fair und einfach zu implementieren.

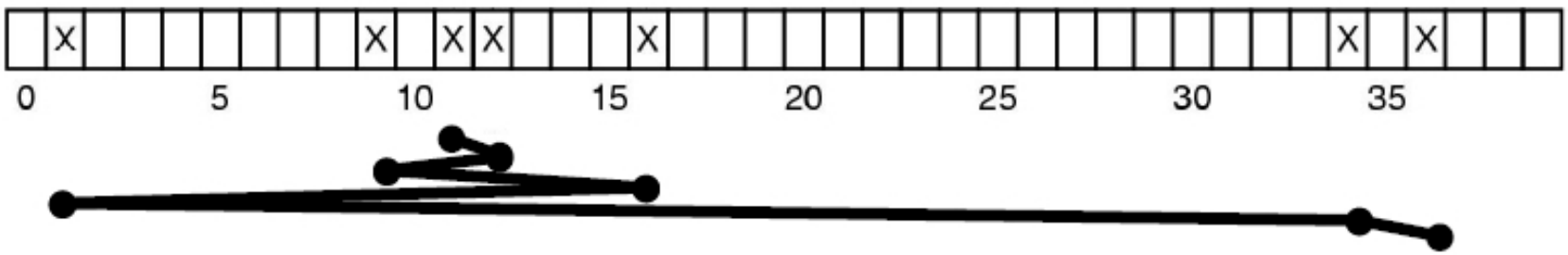
Nachteil. Keine Minimierung der Kopfbewegungen.

Shortest Seek First (SSF)

Unter den aufgestauten Zugriffen wird derjenige ausgewählt, der sich am nächsten an der aktuellen Kopfposition befindet.

Beispiel

Eintreffende Zugriffe: 11, 1, 36, 16, 34, 9, 12



Zurückgelegte Wegstrecke des Kopfes: 1+3+7+15+33+2=61

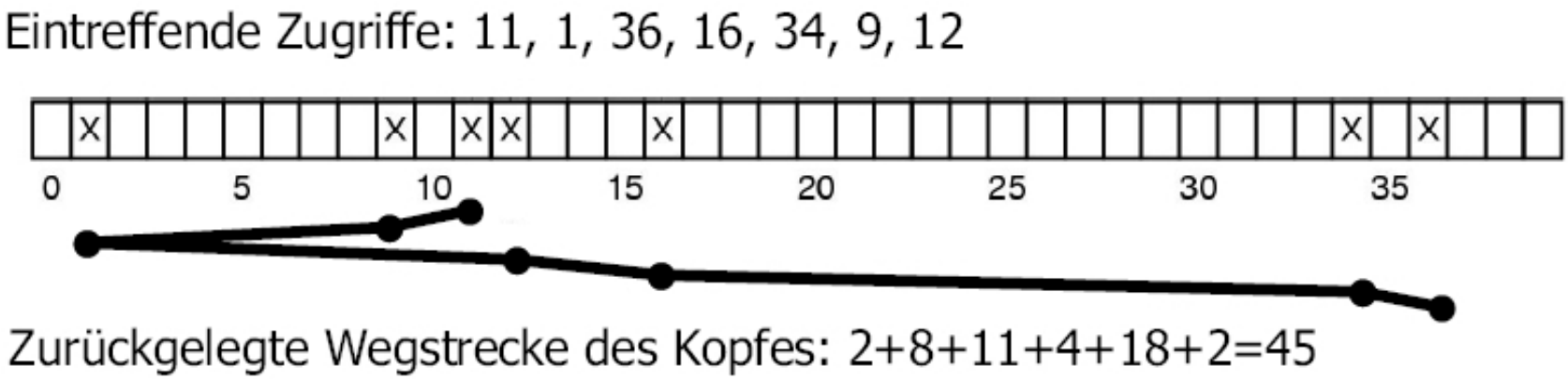
Vorteil. Weniger Kopfbewegung als FCFS.

Nachteil. Nicht fair.

Elevator Scheduling

Der Kopf wird zwischen der größten und der kleinsten Kopfposition aus den aufgestauten Zugriffen hin und her bewegt, dabei werden die auf dem Weg liegenden Zugriffe abgearbeitet.

Beispiel



Vorteil. Weniger Kopfbewegung als SSF.

Nachteil. Fair, vorausgesetzt neu eintreffende Zugriffe auf die aktuelle Kopfposition werden erst auf dem Rückweg bearbeitet.

Inhalt

Speicherverwaltung

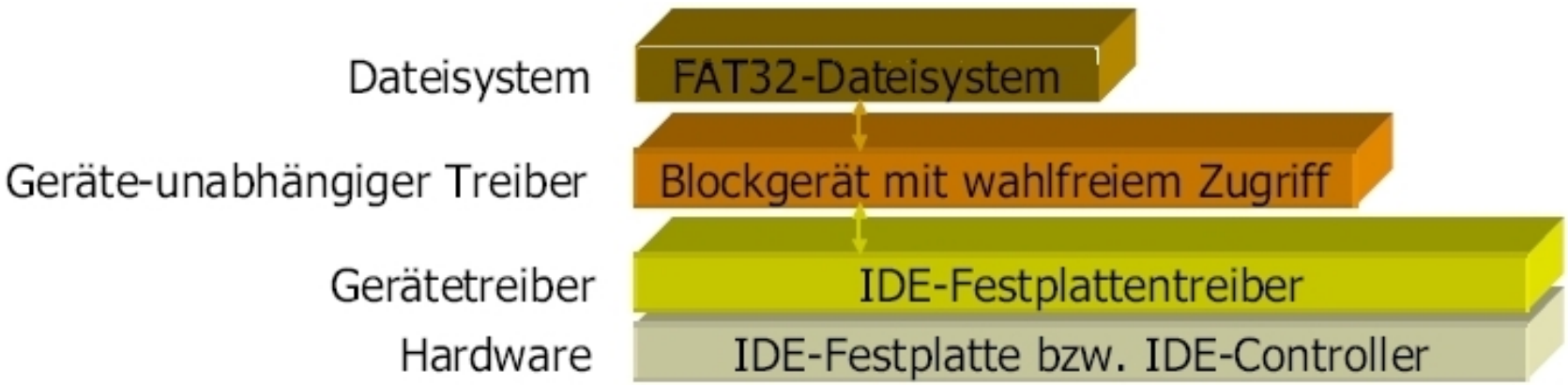
Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Zusammenfassung und Ausblick

Die vorgestellten Konzepte zur Flachenhalsminimierung sind in den unteren Treibeschichten implementiert.

- Kopfscheduling in IDE-Festplatten-Treiber.
- Pufferung im geräteunabhängigen Treiber für wahlfreien Zugriff.

Aufbauend auf diesen Schichten können weitere Dienste realisiert werden, z.B. das **Dateissystem**.



Inhalt

Speicherverwaltung

Ein-/Ausgabe
- Einführung
- Gerätecharakteristiken/Treiberschicht
- Ein-/Ausgabe-Hardwareanbindung
  - Programmed I/O (PIO)
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Ein-/Ausgabe-Flaschenhals
  - First Come, First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Scheduling
- Zusammenfassung und Ausblick
- Ausblick

Ausblick

Aufbauend auf der Treiberschichtung können weitere Dienste realisiert werden, z.B. das **Dateissystem**.