Betriebssysteme
9. Speicherverwaltung

Tobias Lauer

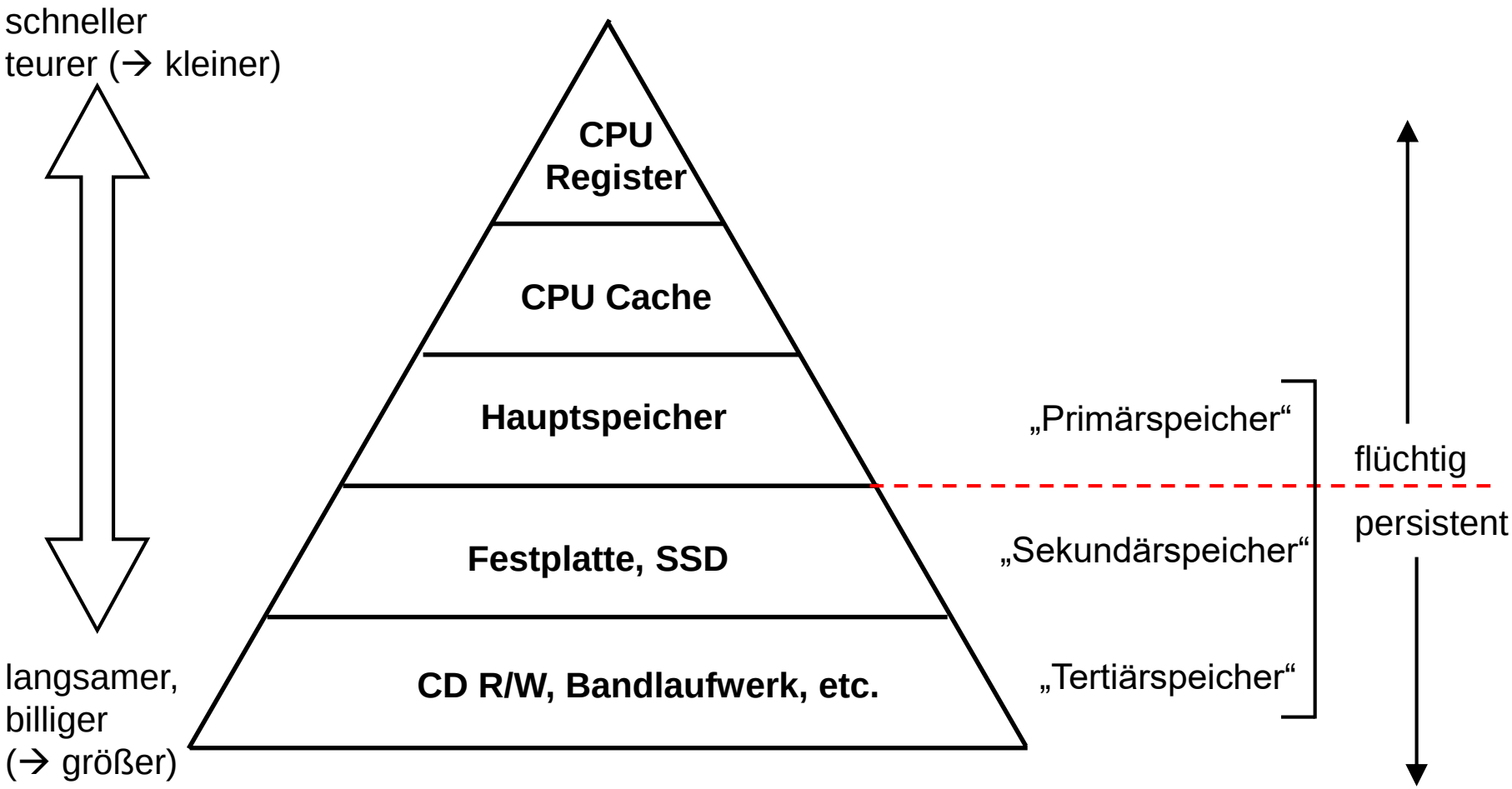
Speicherverwaltung

- Speicherverwaltung ist zentrale Komponente eines Rechners, realisiert durch

  Betriebssystem + Rechnerhardware

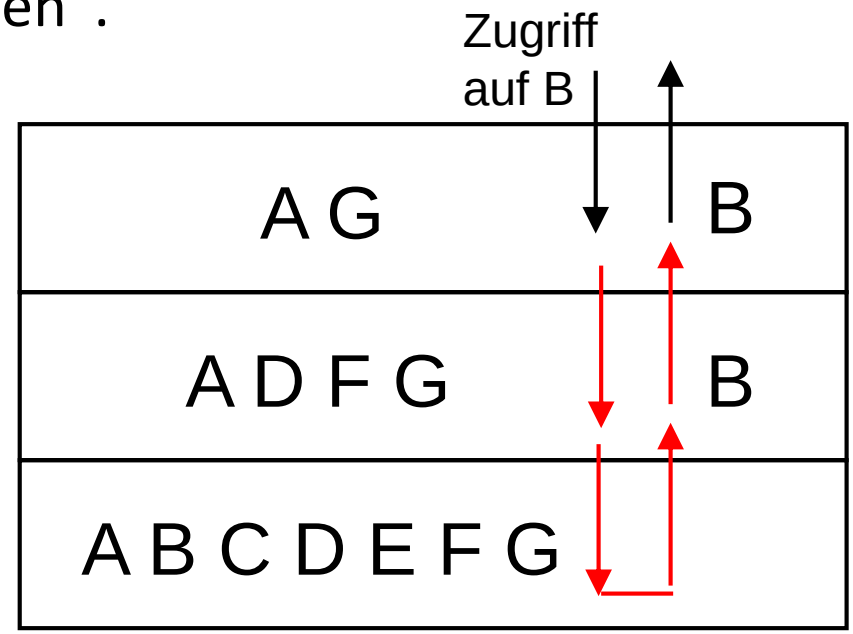
- Was sind die Aufgaben?
  - Anwendungsbezogen: Sicherstellen, dass Speicherreferenzen (in Programmen, in Daten) befriedigt werden
    - … damit Programme überhaupt laufen können
    - … damit alle Programme möglichst effizient laufen können
  - Systembezogen: Effiziente Gesamtverwaltung des physikalischen Speichers für ein Menge von Prozessen (was steht wo, und für wen?)
- Was sind die Randbedingungen?
  - Mehrere parallele bzw. nebenläufige Prozesse
  - Sich verändernde, nicht vorhersehbare Speicheranforderungen
  - Begrenztheit des physikalischen Speichers
- Was sind die Lösungen?
  - Dynamik in der Speicherzuordnung
  - Virtualisierung des Speichers

Wiederholung: Speicherhierarchie



Wie arbeitet die Speicherhierarchie?

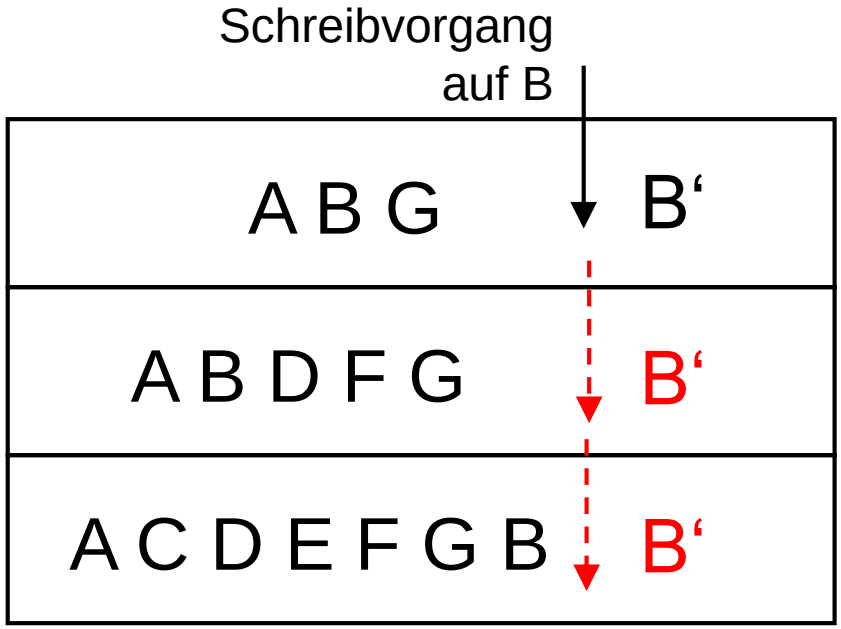
- Höhere Schichten halten Kopien von häufig verwendeten Datenelementen der darunter liegenden Schicht zur Optimierung eines schnellen Zugriffs
- Ist für einen Zugriff keine Kopie vorhanden, so wird diese „von unten nachgeladen“.



Schreibvorgänge in der Speicherhierarchie

- Kopien in höheren Schichten werden durch Schreibvorgänge modifiziert.
- Ohne weitere Vorsichtsmaßnahmen werden Kopien inkonsistent

  → (ggfs. verzögertes) Kopieren des geänderten Inhalts „nach unten“



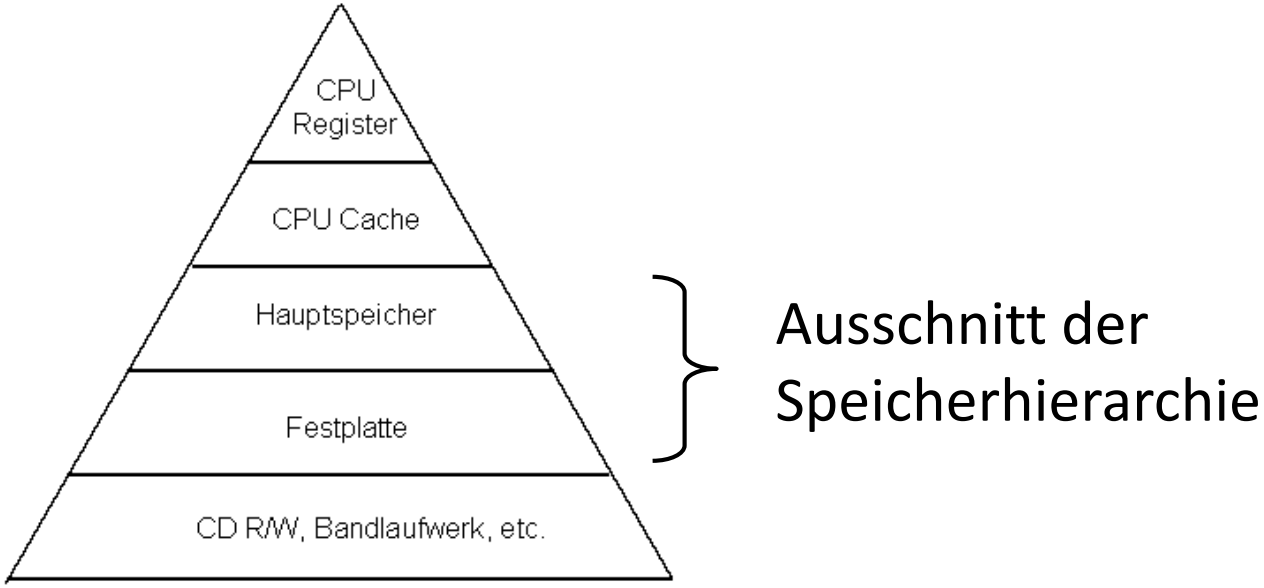
Welche typischen Entscheidungen sind zu treffen?

- Welche Daten werden wann nachgeladen (nur genau die, auf welche zugegriffen wird? Größerer Datenbereich? „Vorausschauendes“ Holen von Daten?)
- Falls der Speicher zu klein wird, um weitere Daten nachzuladen, welche schon gespeicherten Datenelemente werden wann verdrängt?
- Wann werden welche veränderten Daten zurückgeschrieben?

Verantwortlich: Speicherverwaltung (transparent für den Benutzer)

Hauptspeicherverwaltung

- Realisiert eine dynamische Abbildung zwischen
  - logischem Adressraum (Adressraum des Benutzers, Programmadressraum)
  - physikalischem Adressraum (Adressraum des Rechners, realer Speicher)
- Bezieht die Festplatte bzw. SSD als zweite Speicherstufe mit ein



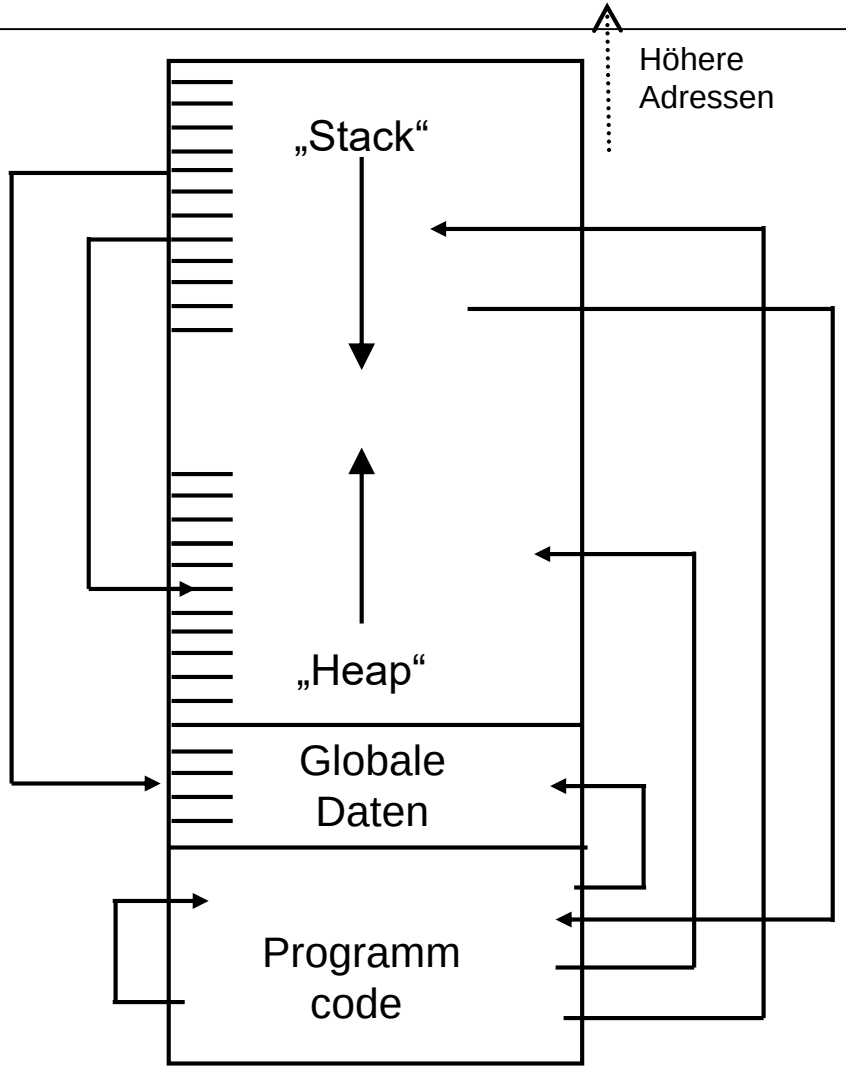
Adressraum eines Programms

- Adressraum = Menge von (Hauptspeicher-)Adressen, welche alle für die Ausführung eines Programms benötigten Code- und Dateninhalte beinhalten.
- Beispiel:
  - Startadresse...Endadresse einer Prozedur/Programms
  - Startadresse..Endadresse eines statisch allokierten Arrays
  - Adresse einer lokalen Variable (liegt auf dem Programmstack)
  - Adressen eines dynamisch allokierten „Shared Memory“-Segments
  - etc.
- Adressraum eines Programms hat i.d.R. Lücken (Lediglich eine Untermenge der numerisch möglichen Adressen eines Programms sind auch gültige Adressen)
- Programm und Daten haben Vielzahl von internen Querverweisen/-referenzen (Bsp: verkettete Liste)

Adressraum eines Programms

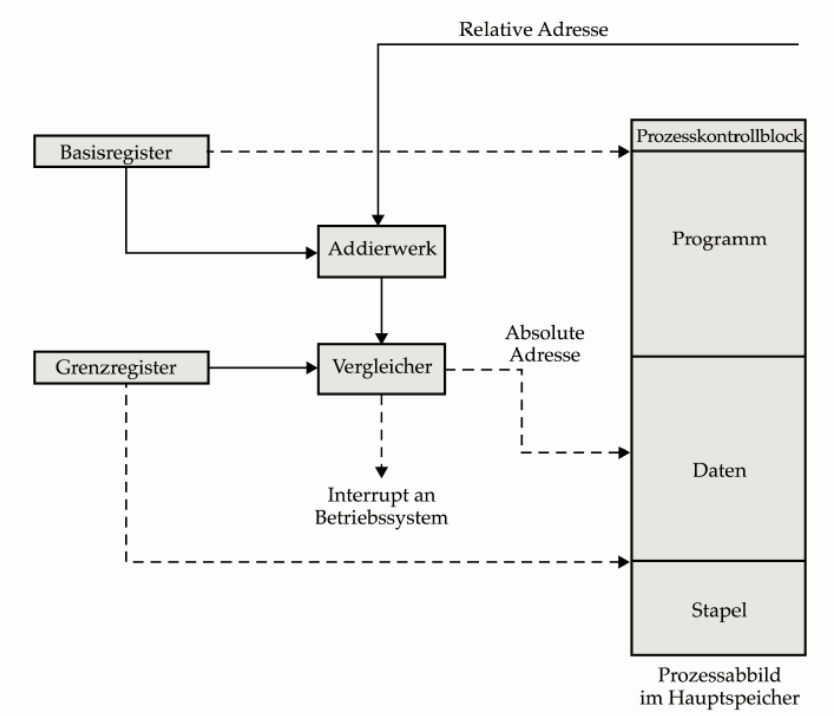
```
char my_buf[20];


void sort (char *inbufptr,
           char *outbufptr)
{
    .... sortieren...
}
main ()
{
  int i, j;
  char *new_buf;
  ...
  for (i=0; i<20; i++)
      my_buf[i] = rand();
  ....
  new_buf = malloc (sizeof(my_buf));
  ....
  sort (my_buf, new_buf);
}
```



Relative Adressierung

- Problem: Wie schafft man es, dass ein Programm unabhängig von der konkreten Platzierung im Hauptspeicher richtige Adressen verwendet?
- Lösung: Relative Adressierung + Hardwareunterstützung (Memory Management Unit, MMU)

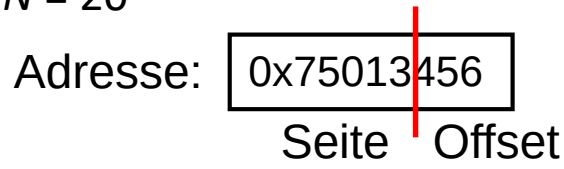


Paging

- Früher: Speicherverwaltungsverfahren allokieren ein zusammenhängendes Stück Hauptspeicher („Partitionierung“).
  → Nachteil: Mangelnde Flexibilität, „Fragmentierung“, Ineffizienz
- Besserer Ansatz: Aufteilen
  1. des Adressraums eines Prozesses — *Page, Seite*
  +
  2. des physikalischen Speichers — *Frame, Rahmen, Kachel*

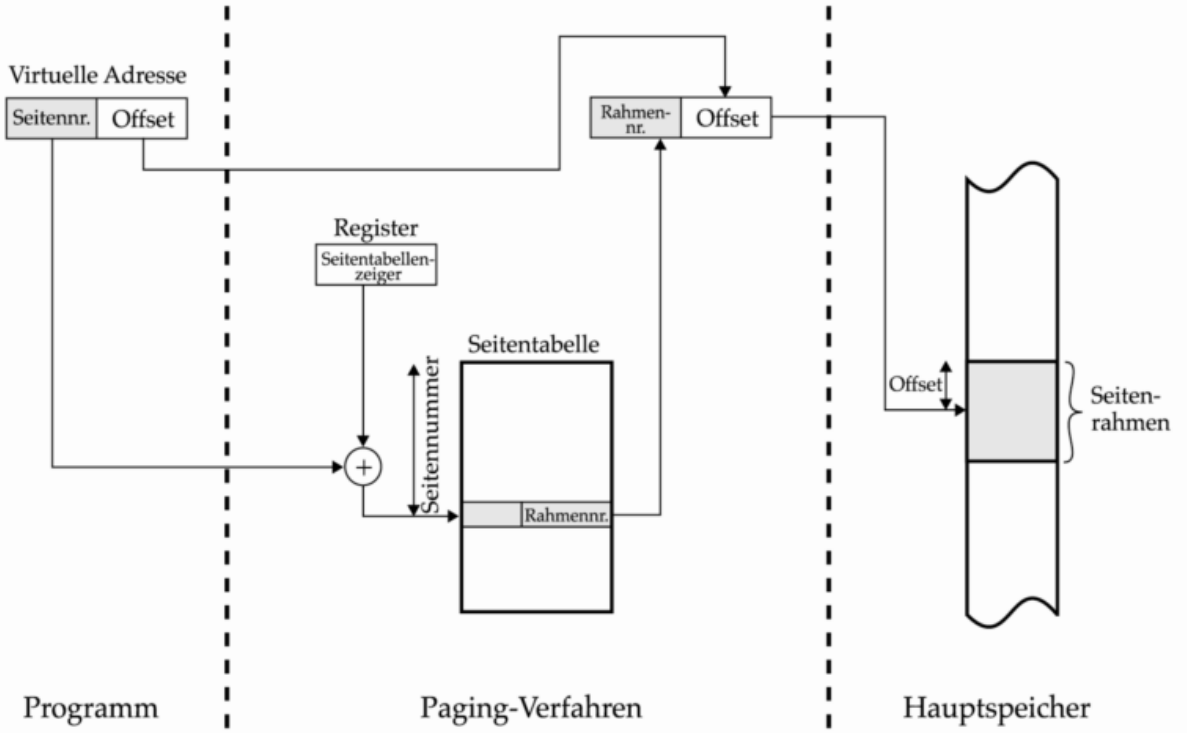
  in kleine Abschnitte fester Größe
- Dynamische Zuordnung zwischen diesen („Page X“ belegt „Rahmen Y“)
  Verwaltet in → „Seitentabellen“
- Unterstützung durch Hardware beim Zuordnen der Adressen
- Vom Programm verwendete logische Adressen werden unterteilt:
  - die *N* ersten Bits werden als Seitennummer interpretiert
  - die restlichen Bits definieren den Offset innerhalb der Seite

Beispiel: $N = 20$



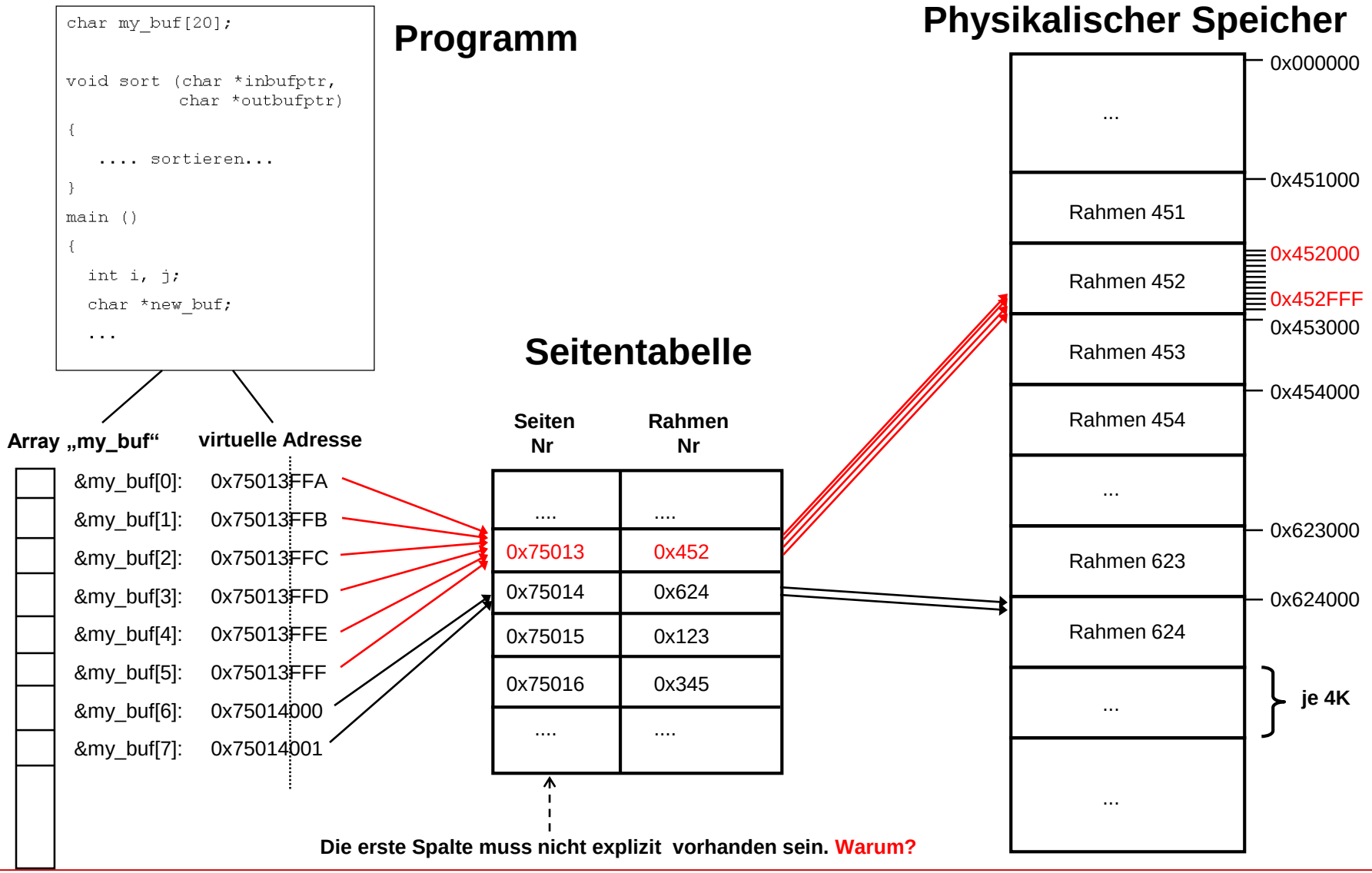
Paging – Seitennummer und Offset

Adressumsetzung beim Paging-Verfahren



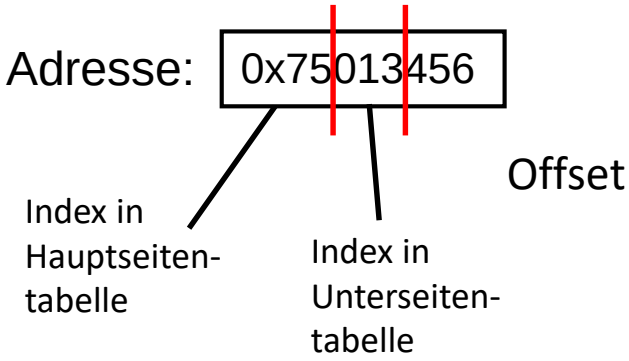
- 1 Seitentabelle pro Prozess
- Verweis auf Seitentabelle z.B. im Prozesskontrollblock
- Prozesswechsel → andere Seitentabelle wird verwendet

Beispiel – Paging

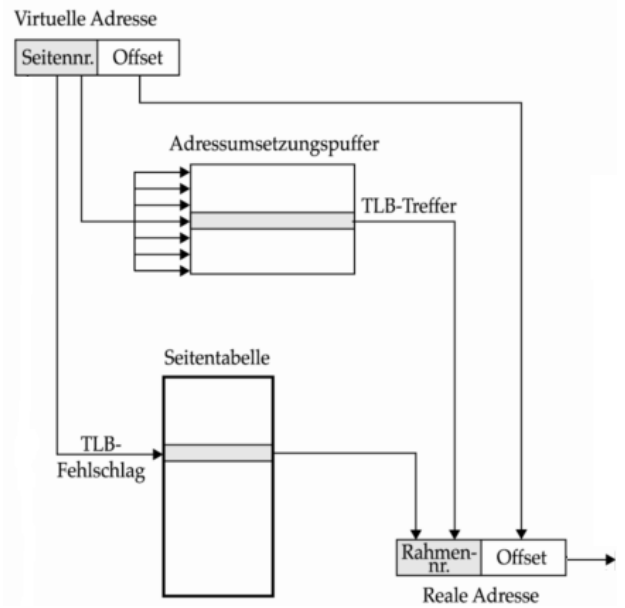


Varianten beim Paging

- Mehrstufiges Paging
  - Seitentabellen können zu groß werden
  - Hauptseitentabelle & Unterseitentabelle
  - Mehrfache indirekte Referenzierung

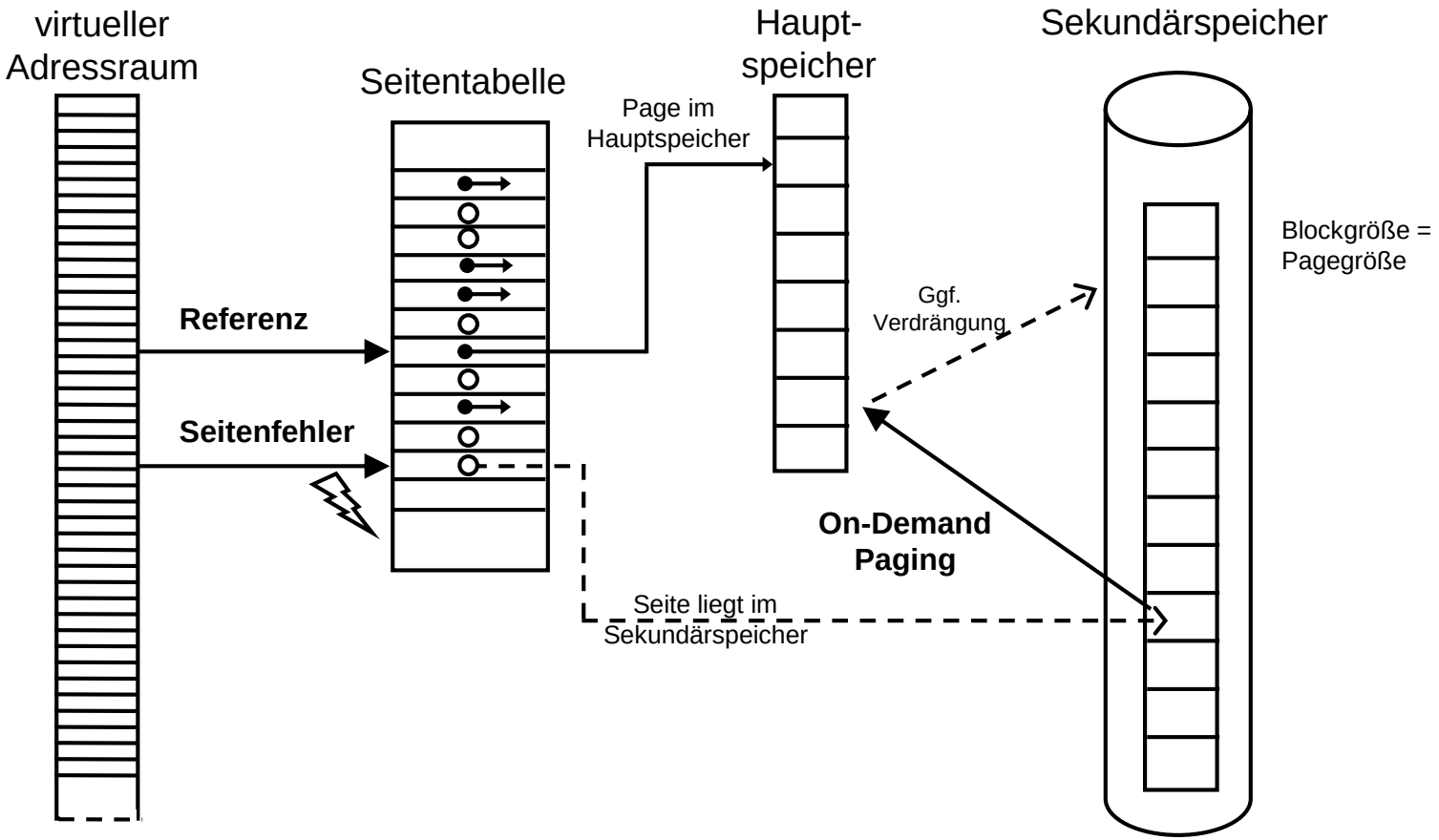


- Schneller Cache für die Seitentabellen
  - „Translation Look-Aside Buffer“ (TLB)
  - die letzten verwendeten Adressen werden gespeichert
  - Verwendung von „Assoziativspeicher“ (Hardware): gleichzeitiger Vergleich der Seitennummer mit ALLEN Einträgen.
  - Treffer im Assoziativspeicher: schneller Zugriff auf Seite
  - Fehltreffer: normaler Zugriff via Seitentabelle (langsamer)
  - Prozesswechsel: Alle Inhalte werden invalidiert



Virtueller Speicher

- Verfahren, um einem Anwenderprozess einen beliebig großen Adressraum zu gestatten und diesen durch ein Paging-Verfahren mit Rückgriff auf Sekundärspeicher zu realisieren.



Virtueller Speicher

- Seitentabelle beinhaltet (wie zuvor) Einträge für den gesamten logischen Adressraum des Prozesses, aber:
  - Einige Einträge zeigen an, dass die zugehörige Seite auf Sekundärspeicher ausgelagert ist.
  - Zugriff auf einen solchen Eintrag erzeugt einen „Seitenfehler“ („Page Fault“)

- Abfolge bei Seitenfehler
  1. Seitenfehler löst einen Interrupt (Kernel Trap) aus
  2. Betriebssystem-Scheduler bringt Prozess in Zustand „Blocked“
  3. Speicherverwaltung kopiert Block mit gewünschter Page vom Sekundärspeicher in einen Rahmen des Hauptspeichers
  4. Seitentabelle wird aktualisiert (geladene Page erhält Referenz auf Hauptspeicher)
  5. Falls bei „3“ kein Platz mehr im Hauptspeicher:
     i. Es wird eine Page ausgewählt und auf Sekundärspeicher verdrängt
     ii. Die Seitentabelle für diese Page erhält einen „Not Present“-Eintrag
  6. Betriebssystem Scheduler bringt Prozess in Zustand „Ready“

Lokalitätsprinzip

- Voraussetzung für effizienten Einsatz des On-Demand Paging-Verfahrens ist eine häufige, zeitnahe Benutzung der gleichen Hauptspeicherseite (sonst dauernde Verdrängung)
- Dies ist in der Praxis auch gegeben (empirisch bestätigt):
  - Schleifen, die über längere Zeit verwendet werden
  - Datenstrukturen, die häufig nahe beieinanderliegen (auf 1 oder 2 Pages)
  - Häufig: 80% der Zeit werden mit 20% des Codes/der Daten verbracht

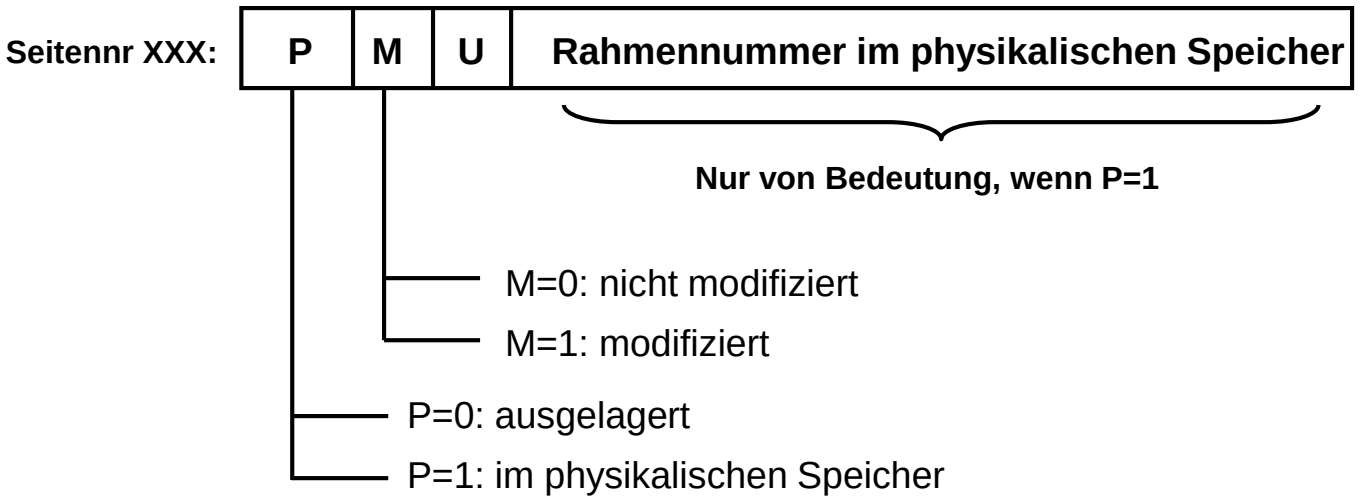
  => „Lokalitätsprinzip“ (räumliche und zeitliche Lokalität)

- Was ist außerdem wichtig:
  - gute Verdrängungsstrategie (siehe unten)
  - Bestimmte Anzahl an eingelagerten Seiten für jeden Prozess („Working Set“)

Lokalitäts-prinzip + **Gute Verdrängungs-strategie** + **Richtige Größe des Working Sets** = **Wenige Seitenfehler**

Speicherorganisation

- Page-Größe = Rahmengröße = Blockgröße auf Sekundärspeicher
- Seitentabelle enthält zusätzliche Information:
  - Präsenz (P-) Bit: zeigt an, ob die Seite im Hauptspeicher (d.h. nicht ausgelagert) ist
  - Modify (M-) Bit: zeigt an, ob eine Seite seit dem letzten Laden modifiziert wurde
  - Weitere Bits für spezielle Seitenersetzungsstrategie (Use-Bit, s.u.)



Seitenaustauschstrategien

- Welche Seite wird verdrängt, wenn ein Seitenfehler auftritt und alle Rahmen des Hauptspeichers belegt sind?

- Zwei große Alternativen:
  - Verdrängung einer Page desjenigen Prozesses, der den Seitenfehler verursacht hat
  - Verdrängung einer Page eines beliebigen Prozesses

- Beispiele für Strategien
  - FIFO (First-In-First-Out)
  - LRU (Least-Recently-Used)
  - Second Chance Algorithmen (verwenden Used-Bit in Seitentabelle)

- Anforderungen
  - effektiv (d.h. die “richtigen” Seiten verdrängen)
  - effizient (d.h. möglichst einfach und schnell)

Was wäre die optimale Strategie?

- Ersetze diejenige Seite, die am längsten nicht mehr gebraucht werden wird.
- Beispiel: Hauptspeicher mit 3 Rahmen

  Programm benutzt nacheinander Seiten 2,3,2,1,5,2,4,5,3,2,5,2

2	3	2	1	5	2	4	5	3	2	5	2			
2	2	2	2	2	2	4	4	4	2	2	2			
-	3	3	3	3	3	3	3	3	3	3	3			
-	-	-	1	5	5	5	5	5	5	5	5			
F	F		F	F		F			F					

- Nicht praktikabel, da in der Regel keine Voraussicht möglich ist.
- Aber: verwendbar, um andere Strategien zu bewerten

FIFO – Strategie (First-In-First-Out)

- Ersetze diejenige Seite, die am längsten im Speicher ist.

- Gleiches Beispiel:

2	3	2	1	5	2	4	5	3	2	5	2			
2	2	2	2	5	5	5	5	3	3	3	3			
-	3	3	3	3	2	2	2	2	2	5	5			
-	-	-	1	1	1	4	4	4	4	4	2			
F	F		F	F	F	F		F		F	F			

- Einfach implementierbar
  - Doppelt verkettete Liste
  - neu geladene Seiten werden vorne eingefügt
  - jeweils letztes Element der Liste wird verdrängt

- Nicht immer effektiv, Anomalien möglich

LRU – Strategie (Least Recently Used)

- Ersetze diejenige Seite, auf die schon am längsten nicht mehr zugegriffen wurde.

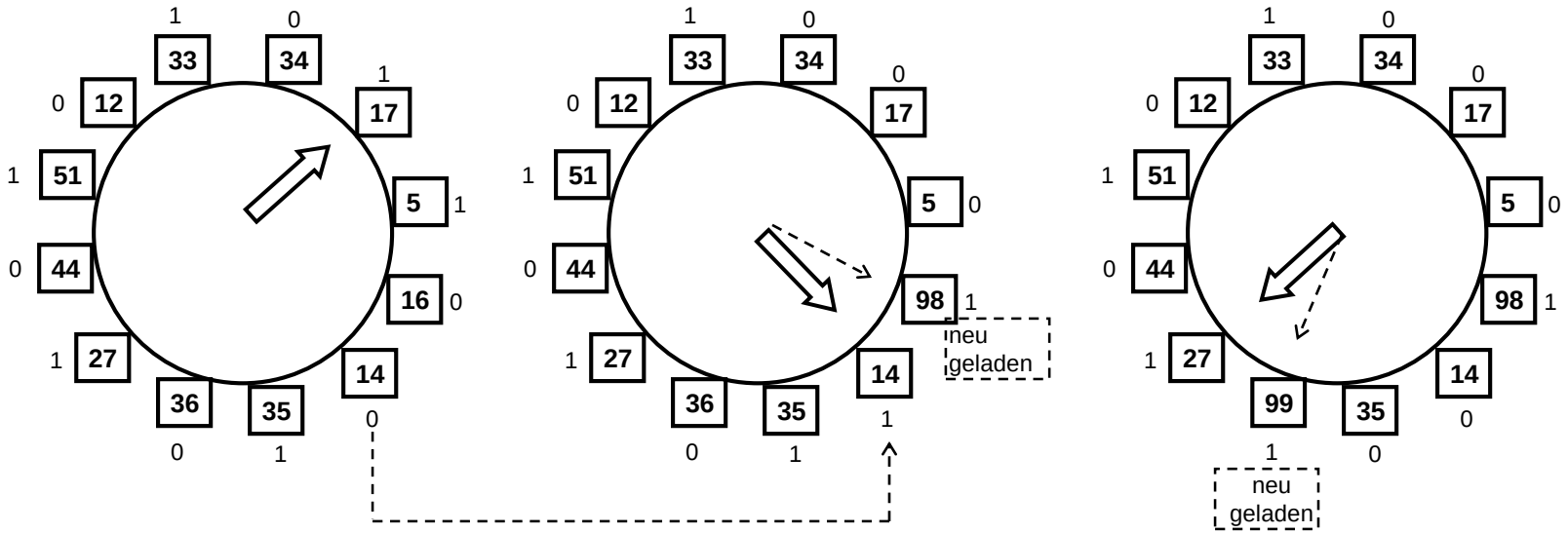
- Gleiches Beispiel:

2	3	2	1	5	2	4	5	3	2	5	2			
2	2	2	2	2	2	2	2	3	3	3	3			
-	3	3	3	5	5	5	5	5	5	5	5			
-	-	-	1	1	1	4	4	4	2	2	2			
F	F		F	F		F		F	F					

- Etwas komplexere Implementierung
  - z.B. doppelt verkettete Liste (in der Reihenfolge der letzten Benutzung)
  - neu geladene Seiten werden vorne eingefügt
  - wiederverwendete Seiten werden gesucht und nach vorne verschoben
  - jeweils letztes Element der Liste wird verdrängt
- I.d.R. besser als FIFO; optimierte LRU-Strategien in UNIX und Windows

Second-Chance Strategie (auch „Clock“-Strategie)

- Alle Rahmen in einem virtuellen Ring angeordnet, jede Seite hat „Use“-Bit
- Zeiger „kreist“ und bestimmt, welche Seite als nächste verdrängt wird:
  Der Zeiger wählt immer die jeweils nächste Seiten mit Use-Bit = 0
  - Use-Bit wird auf „1“ gesetzt, wenn ein Prozess auf die Seite zugreift
  - Gleichzeitig setzt der Zeiger bei allen Seiten bei denen er vorbeikommt die Use-Bits von „1“ auf „0“ (Aging)



- sehr effizienter Algorithmus

Second-Chance Strategie – Implementierung

- Implementierung der „Uhr“ als zirkuläre verkettete Liste
  - Es gibt einen Zeiger auf das aktuelle Element („Uhrzeiger“)

- Seitenanforderung → Nachschlagen in Seitentabelle
  - Fall 1: Seite schon im Hauptspeicher
    → Use-Bit auf 1 setzen (sonst nichts)

  - Fall 2: Seite ausgelagert auf Sekundärspeicher → Clock-Mechanismus:
    1. Für Seite an aktueller Zeigerposition schauen (in Seitentabelle), ob Use-Bit = 1
    2a. Falls ja, …
        - Use-Bit = 0 setzen
        - Zeiger weiterbewegen
        - zu Punkt 1 gehen
    2b. Falls nein, …
        - Seite an Zeigerposition verdrängen (falls vorhanden)
        - angeforderte Seite laden und dort einfügen
        - Seitentabelle aktualisieren (inkl. Use-Bit = 1 setzen)
        - Zeiger weiterbewegen
        - Ende

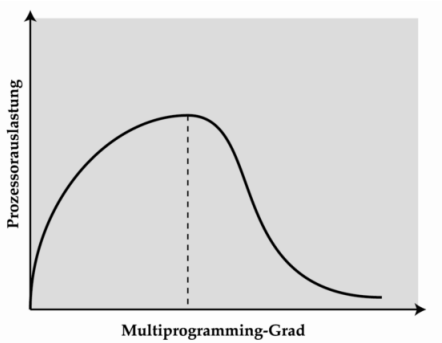
Clock-Algorithmus

-
-

2	3	1	5	5	2	4	5	3	2	5	2			
2	2	2												
-	3	3												
-	-	1												
F	F	F												

Thrashing

- Thrashing (manchmal auch umgangssprachlich als „Swapping“ bezeichnet)
  = übergroße Anzahl von Seitenfehlern / Zeiteinheit, die dazu führt, dass das System nur noch mit Schreib-/Lese-vorgängen auf dem Sekundärspeicher beschäftigt ist und **kein Arbeitsfortschritt** mehr stattfindet.
- Grund (meist):
  - zuviele Prozesse
  - zuviel gleichzeitiger Speicherbedarf
- Für die am Thrashing beteiligten Prozesse gilt i.d.R:
  - Anzahl eingelagerter Seiten < Anzahl Seiten in deren Working Set
- Konsequenz:
  - Jeder „aktiv“ werdende Prozess verdrängt wichtige Seiten von „bereiten“ oder „blockierten“ Prozessen.
  - Sobald diese wieder aktiv werden, erzeugen sie viele Seitenfehler und verdrängen ihrerseits die Seiten der vorher aktiven Prozesse, usw.



- Abhilfen:
  - Neue Prozesse nur zulassen, wenn deren Working Set befriedigbar ist.
  - Ggf. Prozesse komplett suspendieren (→ Swapping; Auslagerung des gesamten Speicherbereichs eines Prozesses)

Weitere Techniken

- Segmentierung (i.d.R. kombiniert mit Paging)
  - Programm hat mehrere „kleine Adressräume“ = Segmente
    (z.B. Segment für Code / für Daten / für Stack ..)
  - Segmente können variable Größe haben
  - Prozess hat eine Segmenttabelle und pro Segment jeweils eine Seitentabelle (jede Zeile in der Segmenttabelle verweist auf eine separate Seitentabelle)
  - Jede Adresse ist dreigeteilt:

  - Vorteil: zusätzliche Schutzmechanismen
  - Sharing von Speicher (Shared Memory) wird i.d.R. über Segmente realisiert