# Grundlagen der Informatik für Ingenieure I

## Background

## 6. Speicherverwaltungssystem


*6.1 Speichervergabe*
*6.1.1 Problemstellung*
*6.1.2. Dynamische Speicherzuteilung*
*6.1.2.1 Vergabestrategien*
*6.1.3 Mehrprogrammbetrieb*
*6.1.3.1 Problemstellung*
*6.1.3.2 Relokation und Binden*
*6.1.4 Segmentierung*
*6.1.5 Seitenadressierung (Paging)*
*6.1.5.1 MMU mit Seiten-Kacheltabelle*
*6.1.6 Segmentierung und Paging*
*6.1.7 Virtueller Speicher*
*6.1.7.1 Demand Paging*
*6.1.8 Seitenersetzung*
*6.1.8.1 Optimale Ersetzungsstrategie*
*6.1.9 Seitenanforderung*

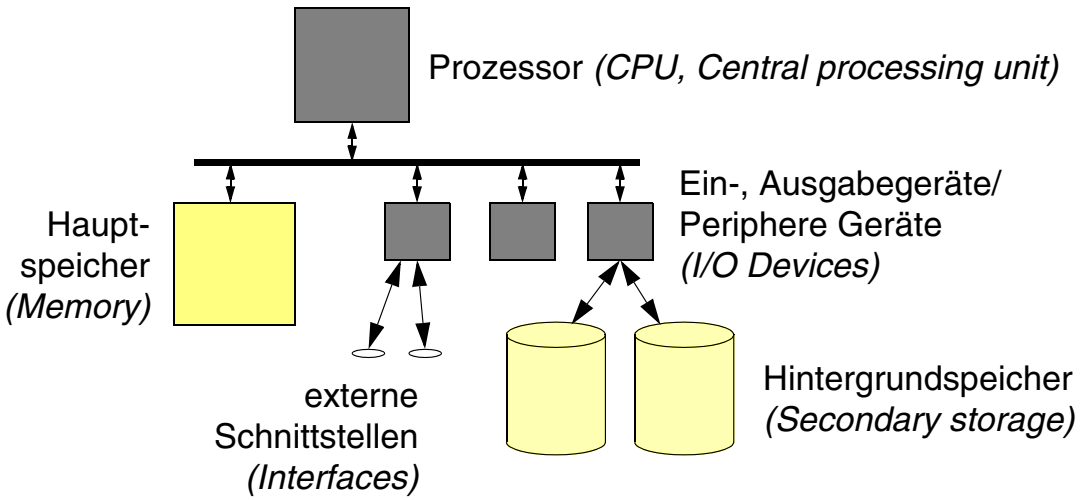






# 6 Speicherverwaltungssystem

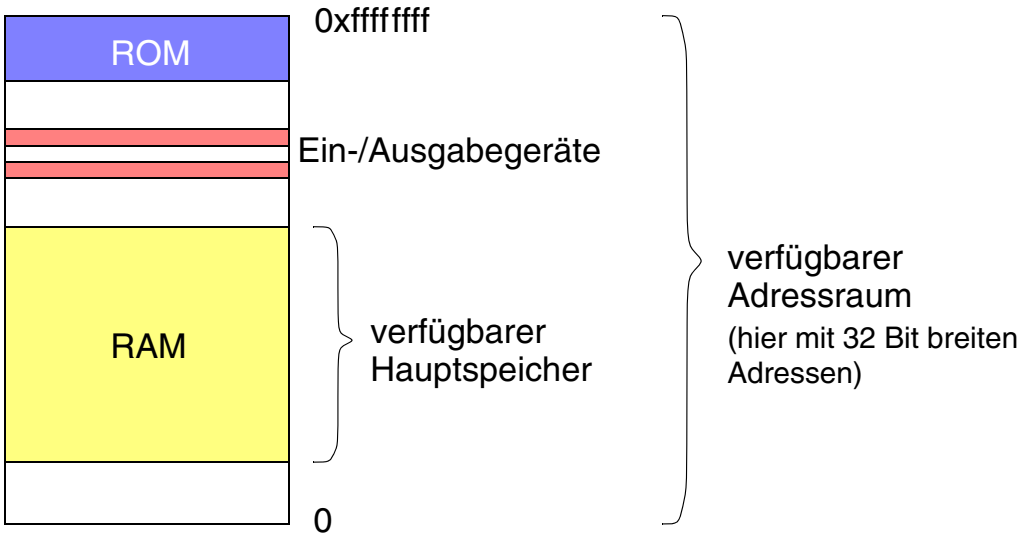
- Betriebsmittel



# 6.1 Speichervergabe

## 1 Problemstellung

- Verfügbarer Speicher









## 1 Problemstellung

- Belegung des verfügbaren Hauptspeichers durch
  - Benutzerprogramme
    - Programmbefehle (Code, Binary)
    - Programmdaten
  - Betriebssystem
    - Betriebssystemcode
    - Puffer
    - Systemvariablen
- ... also Zuteilung des Speichers nötig

# 2 Dynamische Speicherzuteilung

- Segmente
  - zusammenhängender Speicherbereich
    (Bereich mit aufeinanderfolgenden Adressen)
- Allokation (Anforderung) und Freigabe von Segmenten
- Ein Anwendungsprogramm besitzt üblicherweise folgende Segmente:
  - Codesegment
  - Datensegment
  - Stacksegment (für Verwaltungsinformationen, z.B. bei Funktionsaufrufen)
- Suche nach geeigneten Speicherbereichen zur Zuteilung
- ...also Speicherzuteilungsstrategien nötig



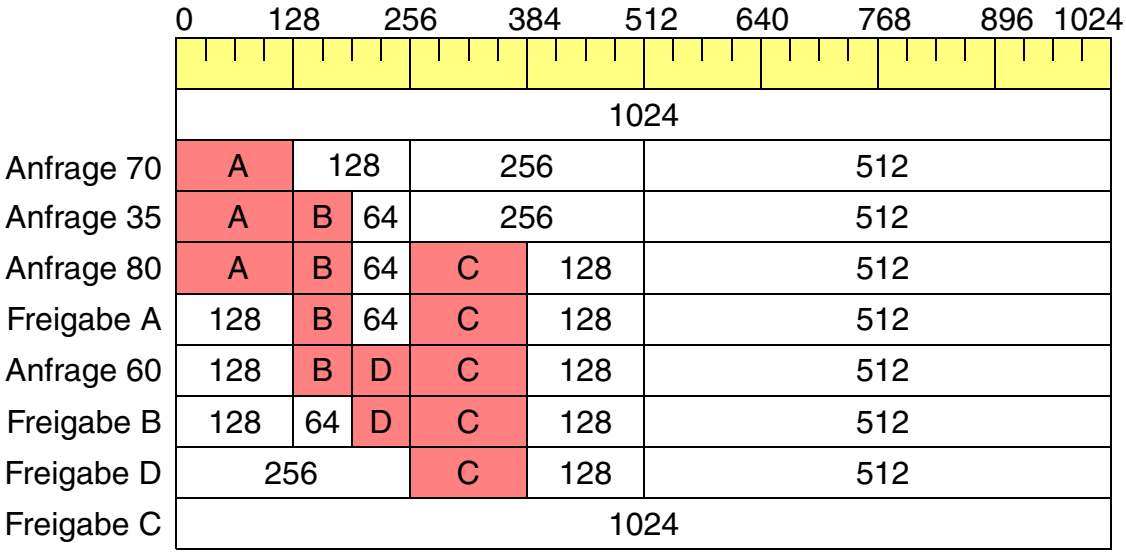



## 6.1.2.1 Vergabestrategien

- First Fit
  - erste passende Lücke wird verwendet
- Rotating First Fit / Next Fit
  - wie First Fit aber Start bei der zuletzt zugewiesenen Lücke
- Best Fit
  - kleinste passende Lücke wird gesucht
- auftretende Probleme:
  - Speicherverschnitt
  - zu kleine Lücken

## 6.1.2.2 Buddy Systeme

- Einteilung in dynamische Bereiche der Größe $2^n$




Effiziente Repräsentation der Lücken und effiziente Algorithmen






## 6.1.2.3 Einsatz der Verfahren

- Einsatz im Betriebssystem
  - Verwaltung des Systemspeichers
  - Zuteilung von Speicher an Prozesse und Betriebssystemteile; ein Teil des Betriebssystems ist immer statisch platziert.
- Einsatz innerhalb eines Prozesses
  - Verwaltung des Haldenspeichers *(Heap)*
  - erlaubt dynamische Allokation von Speicherbereichen durch den Prozess:
    Genaueres: Siehe man(3)!!! malloc(3C)

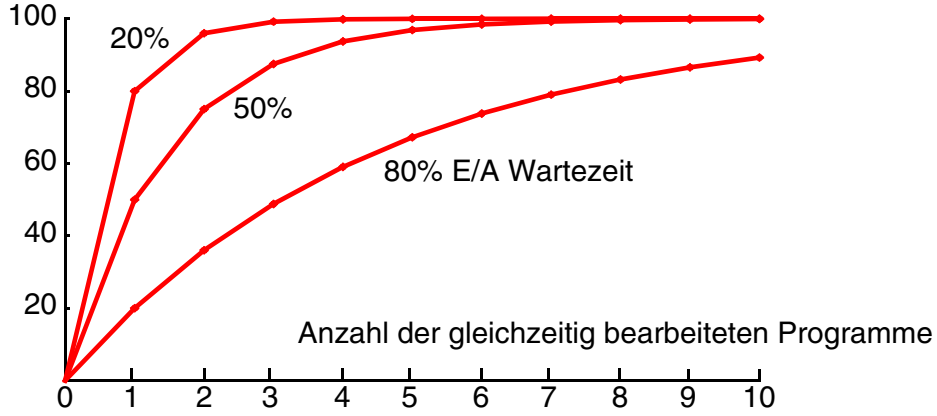
```
char *malloc (unsigned size)
char *realloc (char *ptr, unsigned size)
void free(char *ptr))
```

- Einsatz für Bereiche des Sekundärspeichers
  - Verwaltung bestimmter Abschnitte des Sekundärspeichers
    z.B. Speicherbereich für Prozessauslagerungen *(Swap space)*

# 3 Mehrprogrammbetrieb

## 6.1.3.1 Problemstellung

- Mehrere Prozesse laufen (quasi) gleichzeitig
  - Wartezeiten von Ein-/Ausgabeoperationen ausnutzen
  - CPU Auslastung verbessern
- CPU-Nutzung in Prozent, abhängig von der Anzahl der Prozesse



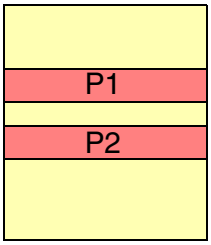






## 6.1.3.1 Problemstellung

- Mehrere Prozesse benötigen Hauptspeicher
  - Prozesse liegen an verschiedenen Stellen im Hauptspeicher
  - Speicher reicht eventuell nicht für alle Prozesse
  - Schutzbedürfnis des Betriebssystems und der Prozesse untereinander




zwei Prozesse und deren Codesegmente im Speicher

- Relokierbare (verschiebbare) Ladeobjekte
- Ein- und Auslagern von Prozessen
- Hardwareunterstützung

# 6.1.3.2 Relokation und Binden *(Relocation and Linking)*

- Festlegung absoluter Adressen in den Programmbefehlen:
  - z.B. ein Sprungbefehl in ein Unterprogramm oder
  - ein Ladebefehl für eine Variable aus dem Datensegment

◆ Absolutes Binden (*Compile time*):
  - Adressen stehen fest --> **Absolute Adressierung**
  - Programm kann nur an bestimmter Speicherstelle korrekt ablaufen

◆ Statisches Binden (*Load time*):
  - Beim Laden (Starten) des Programms werden die relativen Adressen angepasst (reloziert) --> **Relative Adressierung**
  - Relokationsinformation nötig, die vom Compiler oder Assembler geliefert wird




6.1 Speichervergabe

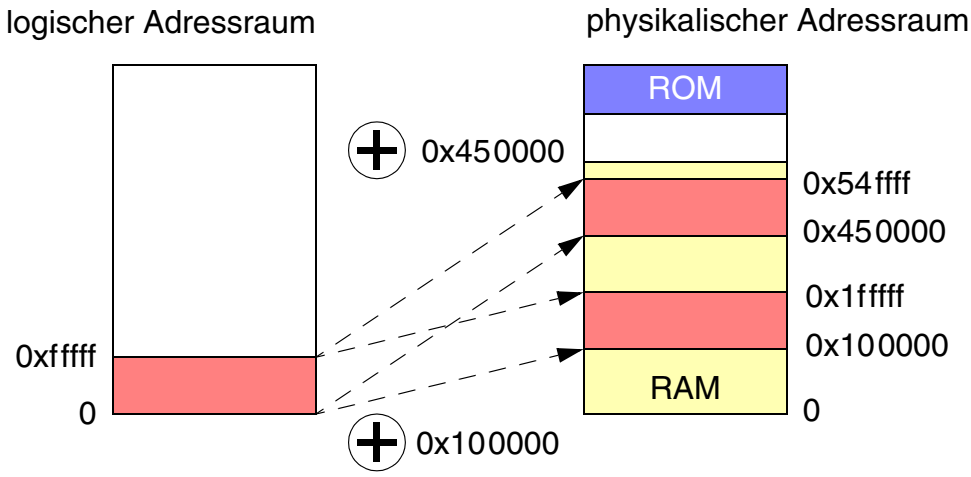
6.1.3.2 Relokation und Binden *(Relocation and Linking)*

◆ Dynamisches Binden *(Run time):*

- Bibliotheken werden **einmal** getrennt von den sie benutzende Programmsystemen geladen.
- Die Referenzen werden zur Laufzeit abgesättigt.
- Spart erheblich Speicherplatz, da der Code gemeinsam genutzt werden kann. Code ist *sharable!*

# 4 Segmentierung

- **Hardwareunterstützung**: Umsetzung logischer in physikalische Adressen
  - Prozesse erhalten einen logischen Adressraum:



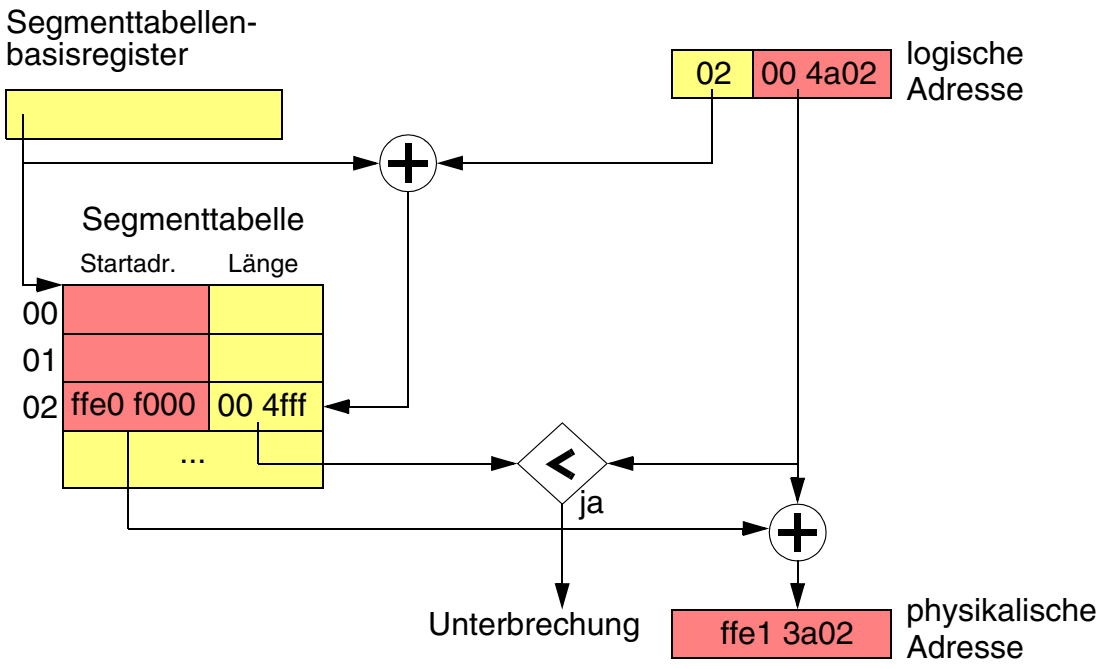

Das Segment im logischen Adressraum kann an pot. jeder beliebige Stelle im physikalischen Adressraum liegen, bei gewisser Rasterung!






# 4 Segmentierung

- Realisierung mit Übersetzungstabelle:



# 4 Segmentierung

- Hardware wird MMU *(Memory management unit)* genannt
- Schutz vor Segmentübertretung
  - Unterbrechung zeigt Speicherverletzung an
  - Programme und Betriebssystem voreinander geschützt
- Prozessumschaltung durch Austausch der Segmentbasis
  - jeder Prozess hat eigene Übersetzungstabelle
- Ein- und Auslagerung vereinfacht
  - nach Einlagerung an pot. beliebige Stelle muss lediglich die Übersetzungstabelle angepasst werden
- Gemeinsame Segmente möglich
  - Befehlssegmente
  - Datensegmente (*Shared memory*)



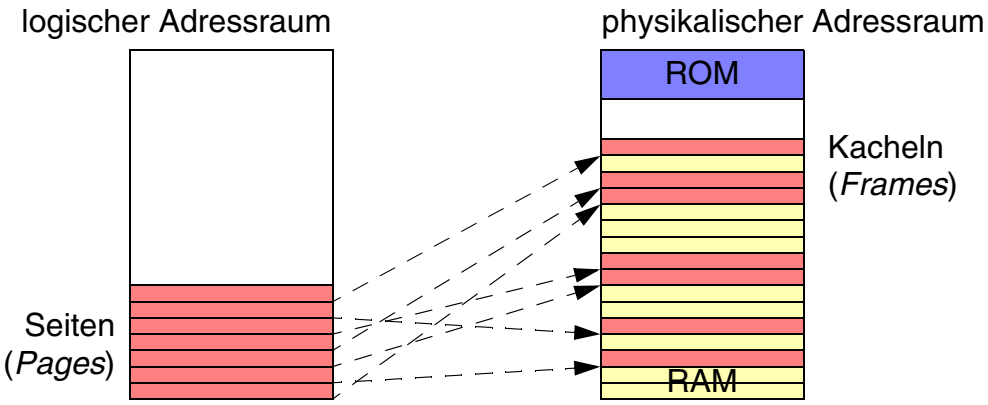



# 4 Segmentierung

- Zugriffsschutz einfach integrierbar
  - z.B. Rechte zum Lesen, Schreiben und Ausführen von Befehlen, die von der MMU geprüft werden
- Fragmentierung des Speichers durch häufiges Ein- und Auslagern
  - es entstehen kleine, nicht nutzbare Lücken
- Kompaktifizieren
  - Segmente werden verschoben, um Lücken zu schließen; Segmenttabelle wird jeweils angepasst
- lange E/A Zeiten für Ein- und Auslagerung
  - nicht alle Teile eines Segments werden gleich häufig genutzt

# 5 Seitenadressierung (*Paging*)

- Einteilung des logischen Adressraum in gleichgroße Seiten, die an beliebigen Stellen im physikalischen Adressraum liegen können
  - Lösung des Fragmentierungsproblem
  - keine Kompaktifizierung mehr nötig
  - Vereinfacht Speicherbelegung



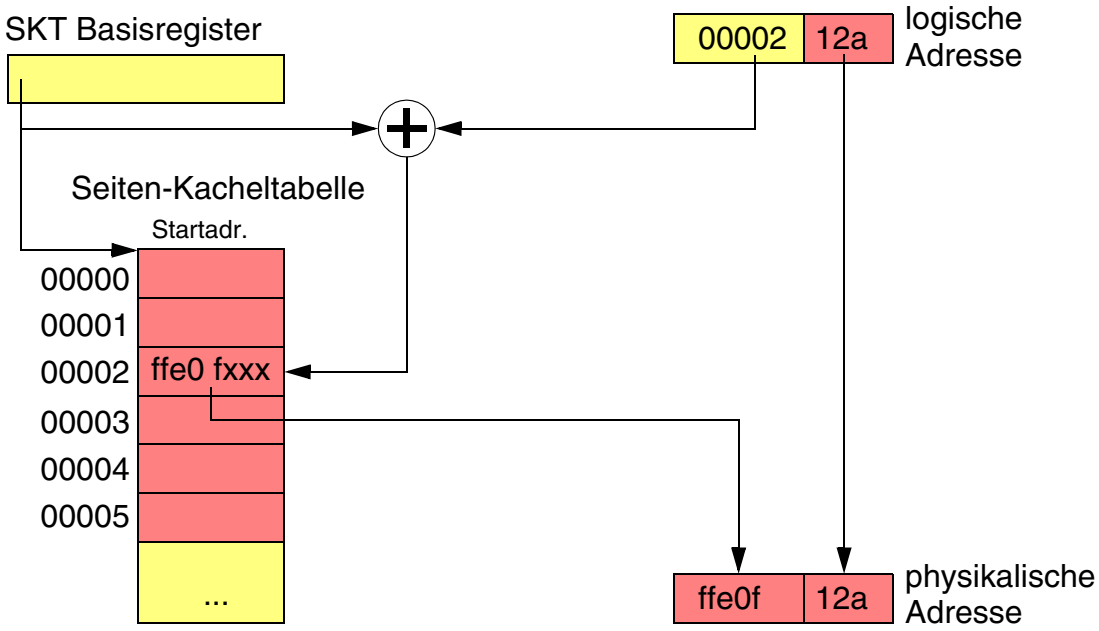






## 6.1.5.1 MMU mit Seiten-Kacheltabelle

- Tabelle setzt Seiten in Kacheln um



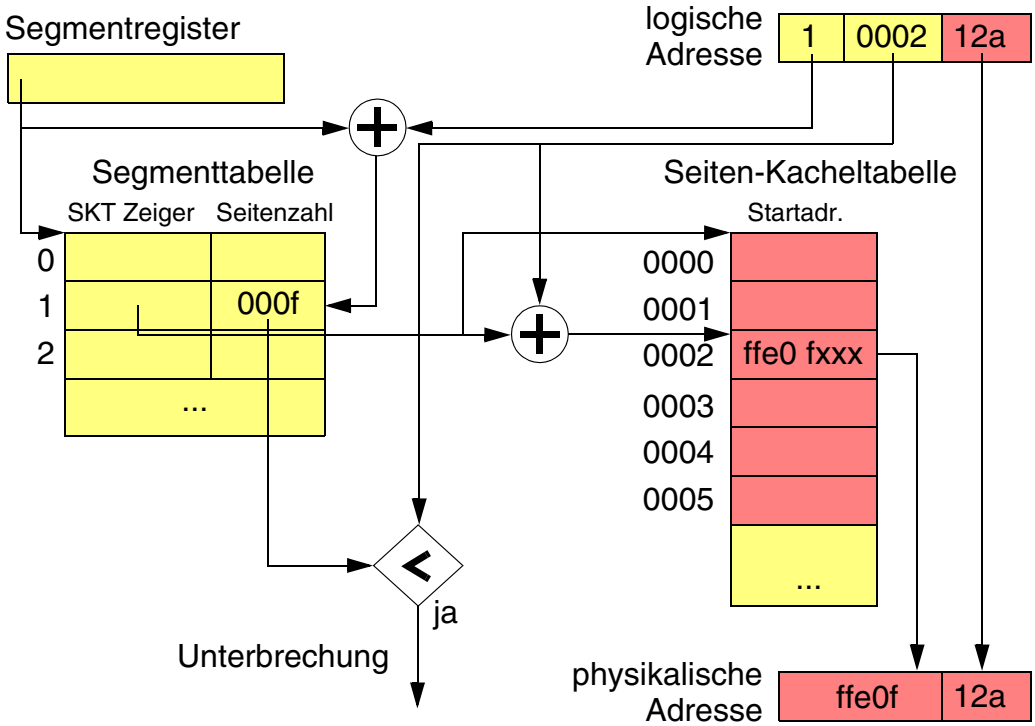
# 6.1.5.1 MMU mit Seiten-Kacheltabelle

- Seitenadressierung erzeugt internen Verschnitt
  - letzte Seite eventuell nicht vollständig genutzt
- Seitengröße
  - kleine Seiten verringern internen Verschnitt, vergrößern aber die Seiten-Kacheltabelle (und umgekehrt)
  - übliche Größen: 512 Bytes — 8192 Bytes
- große Tabelle, die im Speicher gehalten werden muss
- viele implizite Speicherzugriffe nötig
- Kombination mit Segmentierung sinnvoll






# 6 Segmentierung und Seitenadressierung



# 7 Virtueller Speicher

- Entkoppelung des Speicherbedarfs vom verfügbaren Hauptspeicher
  - Prozesse benötigen nicht alle Speicherstellen gleich häufig
    - bestimmte Befehle werden selten oder gar nicht benutzt (z.B. Fehlerbehandlungen)
    - bestimmte Datenstrukturen werden nicht voll belegt
  - Prozesse benötigen evtl. mehr Speicher als Hauptspeicher vorhanden

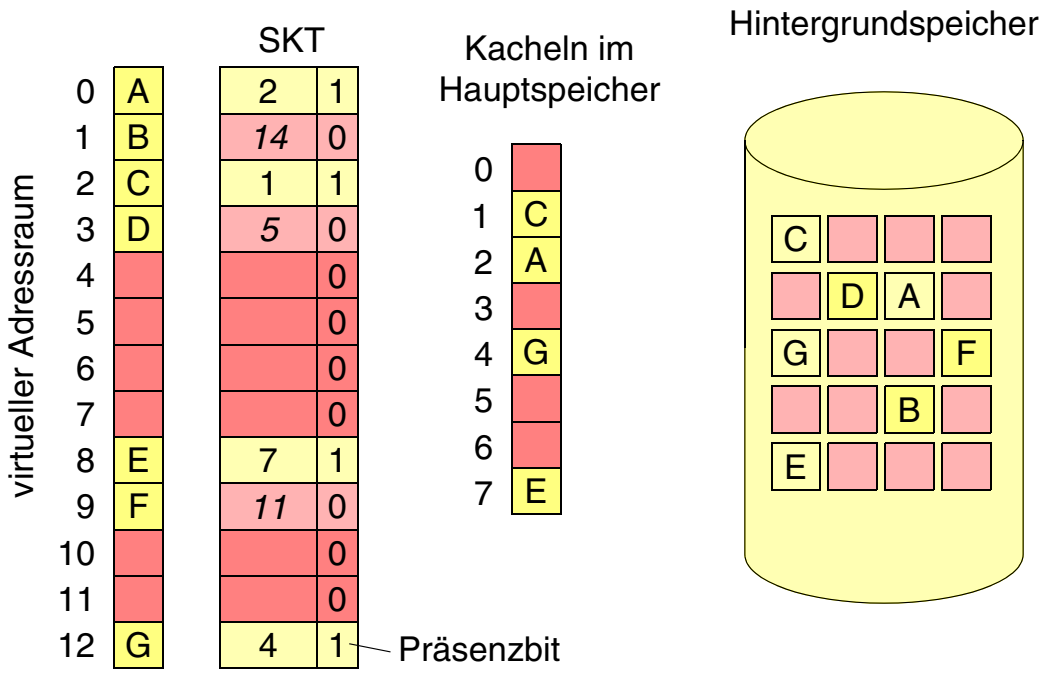
- Idee
  - Vortäuschen eines großen Hauptspeichers
  - Einlagern benötigter Speicherbereiche
  - Abfangen von Zugriffen auf nicht eingelagerte Bereiche
  - Bereitstellen der benötigen Bereiche





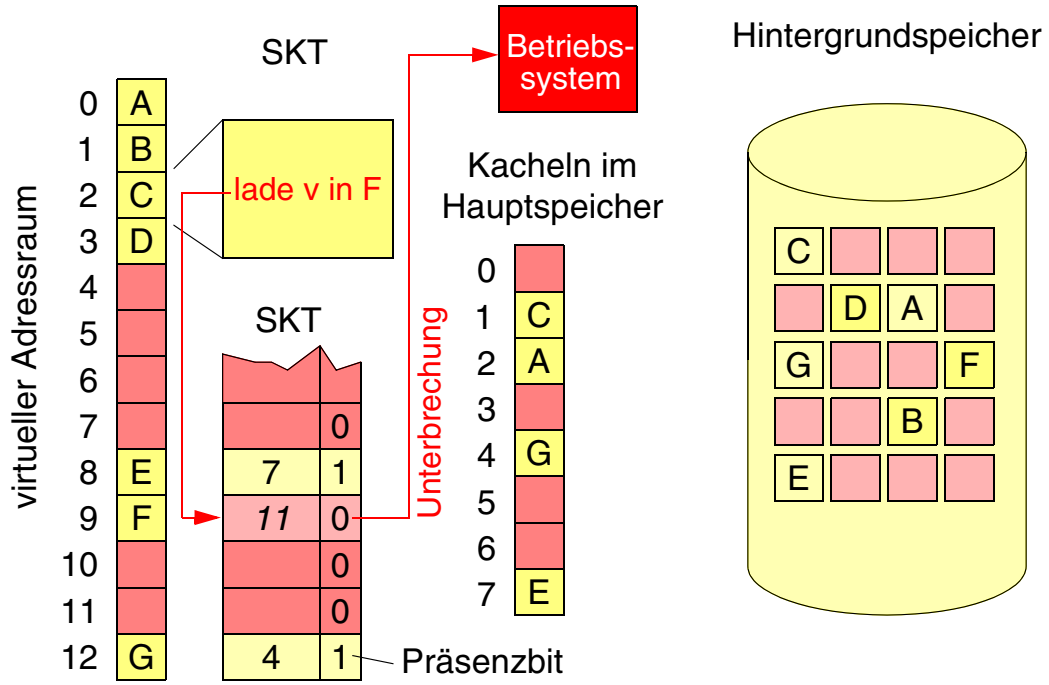

## 6.1.7.1 Demand Paging

- **Bereitstellen von Seiten auf Anforderung**



# 6.1.7.1 Demand Paging

- **Reaktion auf Seitenfehler**



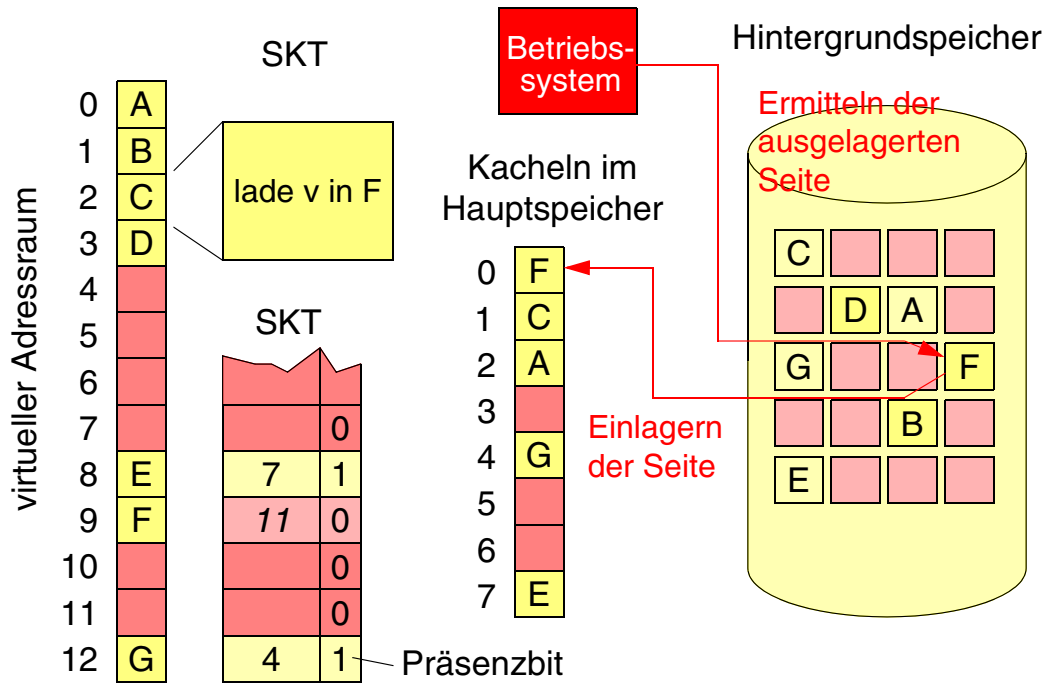






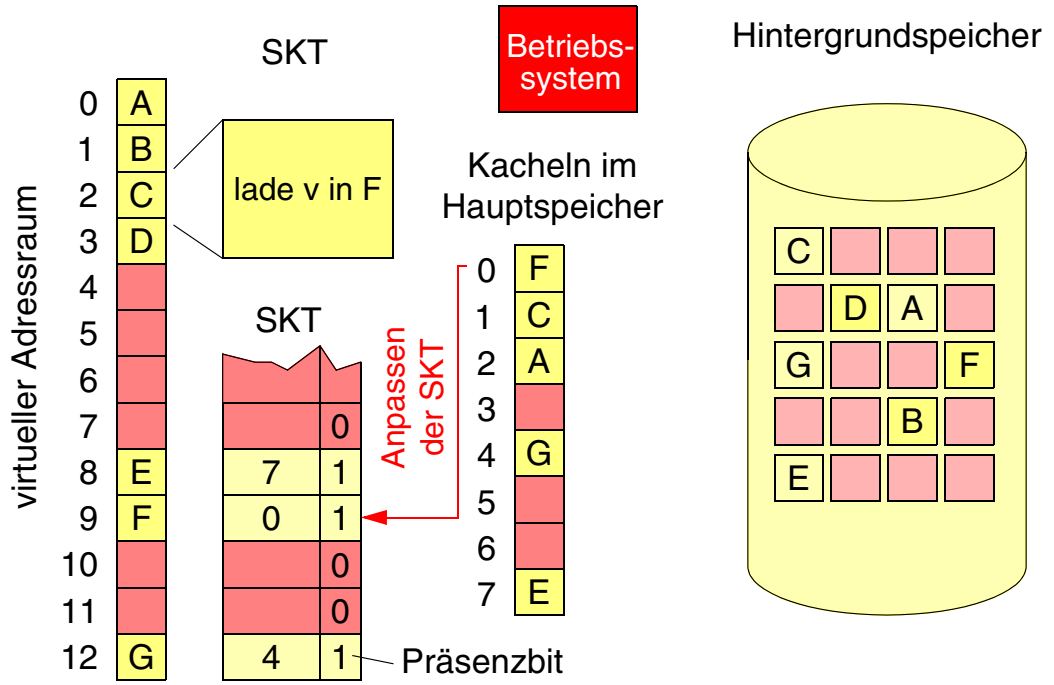
# 6.1.7.1 Demand Paging

- **Reaktion auf Seitenfehler**



# 6.1.7.1 Demand Paging

- **Reaktion auf Seitenfehler**



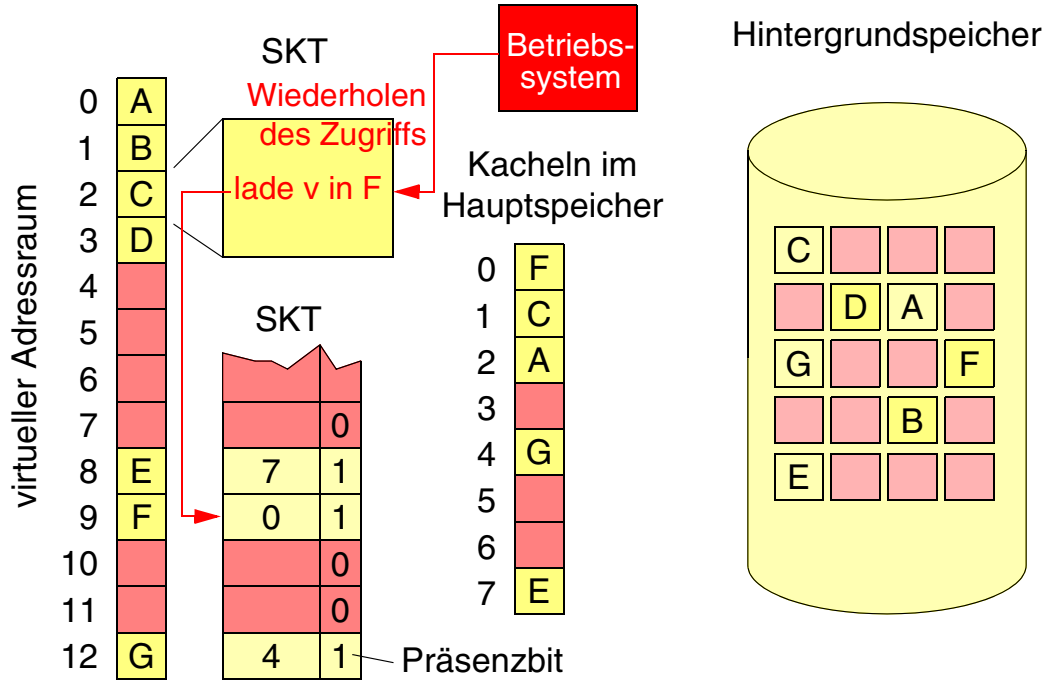




6.1 Speichervergabe

6.1.7.1 Demand Paging

- Reaktion auf Seitenfehler



# 8 Seitenersetzung

- Was tun, wenn **keine freie Kachel** vorhanden?
  - Eine Seite muss verdrängt werden, um Platz für neue Seite zu schaffen!
  - Auswahl von Seiten, die nicht geändert wurden (*Dirty bit* in der SKT)
  - Verdrängung erfordert Auslagerung, falls Seite geändert wurde

- Vorgang:
  - Seitenfehler *(Page fault)*: Unterbrechung
  - Auslagern einer Seite, falls keine freie Kachel verfügbar
  - Einlagern der benötigten Seite
  - Wiederholung des Zugriffs

- Problem:
  - **Welche Seite** soll ausgewählt werden?






## 6.1.8.1 Optimale Ersetzungsstrategie

- Es gibt eine **optimale Ersetzungsstrategie $B_0$**:
  - Die tatsächliche Referenzfolge
    - Seite mit dem größten “Vorwärtsabstand” wird ersetzt.
    - d.h. Seite die am längsten in der Zukunft nicht mehr benötigt wird
  - Problem: Referenzfolge ist vorher nicht bekannt.

- Suche nach Strategien, die möglichst nahe an $B_0$ kommen:
  - z.B. ***Least recently used*** (LRU):
    - Wähle die Seite zum Auslagern aus, die am längsten nicht mehr referenziert wurde
    - also die Seite mit dem größten "Rückwärtsabstand".

# 6.1.8.1 Optimale Ersetzungsstrategie

- Statt nur eine Seite zu ersetzen wird vorausschauend der Transfer auch der nächsten Seite angestoßen (read ahead)
- Ein Hintergrundprozess sorgt durch "Auslagerung auf Verdacht", dass immer genügend freie Seiten für eine Einlagerung zur Verfügung stehen
- Ziel der angewandten Strategien ist es, die *Page Fault Raten* zu minimisieren.






# 8 Seitenanforderung

- Mögliche Zuordnungen (Anzahl der Kacheln pro Prozess):
  - Anzahl der Prozesse bestimmt die Kachelmenge, die ein Prozess bekommt
  - "Größe" des Programms fließt in die zugeteilte Kachelmenge ein
- Sind zuviele Prozesse aktiv, werden welche deaktiviert
  - ◆ Kacheln teilen sich auf weniger Prozesse auf
    - Verbindung mit dem Scheduling nötig
      - Verhindern von Aushungerung
      - Erzielen kurzer Reaktionszeiten
      - Vermeidung von "**Seitenflattern**" (Thrashing = ausgelagerte Seite wird wieder angefordert)
  - ◆ guter Kandidat: Prozess mit wenigen Seiten im Hauptspeicher
    - geringe Latenz bei Wiedereinlagerung bzw. wenige Seitenfehler bei Aktivierung und Demand paging

# 8 Seitenanforderung

- **Working Set Model** (Arbeitsmengenmodell):
  - Aus der Beobachtung der Vergangenheit eines Prozessverhaltens wird mit (wahrscheinlichkeitstheoretischen) Annahmen auf das Verhalten der Zukunft geschlossen.
  - Ziel ist es, dem Prozess die Seiten zur Verfügung zu stellen, die die Anzahl der Seitenanforderungen (*Page Faults*) minimiert.