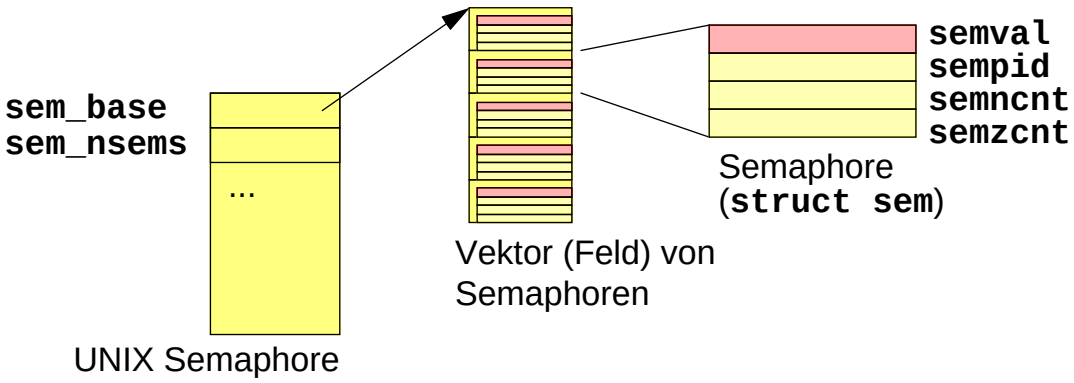
D.7 UNIX Semaphore

- Vektor von Semaphoren (erweitertes Vektoradditionssystem)



  - Gleichzeitige und atomare Operationen auf mehreren Semaphoren im Vektor möglich

1 Erzeugen einer UNIX Semaphore

- UNIX Semphore haben systemweit eindeutige Identifikation (Key)
  - Erzeugen und Aufnehmen der Verbindung zu einer Semaphore

```
int semget( key_t key, int nsems, int semflg );
```

Identifikation
– neue für Erzeugung
– bestehende für Verbindungsaufnahme

Anzahl d. Semaphoren im Vektor

Zugrifssrechte; Erzeugung oder Verbindungsaufnahme

  - Ergebnis ist eine Semaphore ID ähnlich wie ein Filedescriptor
    - Semaphore ID muß bei allen Operationen verwendet werden
  - Zugriffsrechte: Lesen, Verändern
    - einstellbar für Besitzer, Gruppe und alle anderen (ähnlich wie bei Dateien)

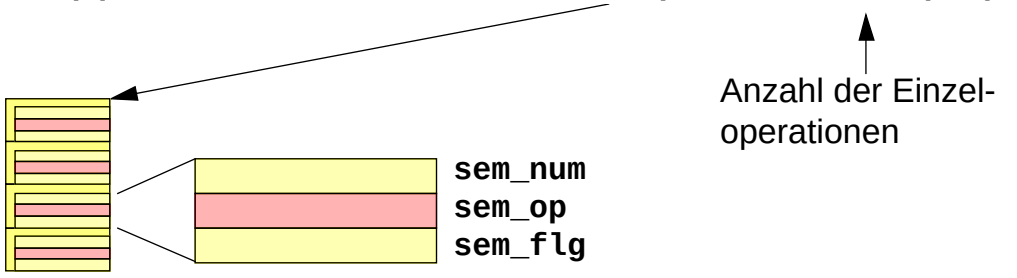
1 Erzeugen einer UNIX Semaphore (2)

- Verwendung des Keys
  - Alle Prozesse, die auf die Semaphore zugreifen wollen, müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann keine Semaphore mit gleichem Key erzeugt werden
  - Ist ein Key bekannt, kann auf die Semaphore zugegriffen werden
    - gesetzte Zugriffsberechtigungen werden allerdings beachtet

2 Operationen auf UNIX Semphoren

- Operationen auf mehreren der Semaphoren im Vektor

```
int semop( int semid, struct sembuf *sops, size_t nsops );
```



  - Operationen
    - **`sem_num`**: Nummer der Semaphore im Vektor
    - **`sem_op < 0`**: ähnlich P-Operation – Herunterzählen der Semaphore (blockierend oder mit Fehlerstatus, je nach **`sem_flg`**)
    - **`sem_op > 0`**: ähnlich V-Operation – Hochzählen der Semaphore
    - **`sem_op == 0`**: Test auf 0 (blockierend oder mit Fehlerstatus, je nach **`sem_flg`**)

2 Operationen auf UNIX Semaphoren (2)

- Kontrolloperationen

```
int semctl( int semid, int semnum, int cmd,
            [ union semun arg ] );
```

  - explizites Setzen von Werten (einen, alle)
  - Abfragen von Werten (einen, alle)
  - Abfragen von Zusatzinformationen
    - welcher Prozeß hat letzte Operation erfolgreich durchgeführt
    - wann wurde letzte Operation durchgeführt
    - Zugriffsrechte
    - Anzahl der blockierten Prozesse

3 Beispiel: Philosophenproblem

- Eine UNIX Semaphore mit fünf Elementen (entsprechen Gabeln)
  - Deklarationen

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int i;                      /* number of philosopher */
int j;
int semid;                          /* semaphore ID */
struct sembuf pbuf[2], vbuf[2];  /* operation buffer */

union semun {               /* UNION for semctl */
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;

...
```

3 Beispiel: Philosophenproblem (2)

◆ Erzeuge Semaphore

```
...

semid= semget( IPC_PRIVATE, 5, IPC_CREAT|SEM_A|SEM_R );
if( semid < 0 ) { ... /* error */ }

for( j= 0; j < 5; j++ ) { /* set all values to 1 */
    arg.val= 1;
    if( semctl( semid, j, SETVAL, arg ) < 0 ) {
        ... /* error */
    }
}

...
```


3 Beispiel: Philosophenproblem (3)

◆ Erzeugen der Prozesse

```
...

for( i=0; i<=3; i++ ) {   /* start children i= 0..3; */
    pid_t pid= fork();

    if( pid < (pid_t)0 ) { ... /* error */ }
    if( pid ==(pid_t)0 ) {
        /* child */

        break;
    }
}                         /* parent: i= 4; */

...
```

3 Beispiel: Philosophenproblem (4)

◆ Initialisierungen

```
...    /* we are philosopher i */

/* initialize buffer for P operation */

pbuf[0].sem_num= i; pbuf[1].sem_num= (i+1)%5;
pbuf[0].sem_op= pbuf[1].sem_op= -1;
pbuf[0].sem_flg= pbuf[1].sem_flg= 0;

/* initialize buffer for V operation */

vbuf[0].sem_num= i; vbuf[1].sem_num= (i+1)%5;
vbuf[0].sem_op= vbuf[1].sem_op= 1;
vbuf[0].sem_flg= vbuf[1].sem_flg= 0;

...
```


3 Beispiel: Philosophenproblem (5)

◆ Philosoph

```
...

while( 1 ) {
    ...    /* thinking */

    if( semop( semid, pbuf, 2 ) < 0 ) { ... /* error */ }

    ...    /* eating */

    if( semop( semid, vbuf, 2 ) < 0 ) { ... /* error */ }
}
```

D.8 Zusammenfassung

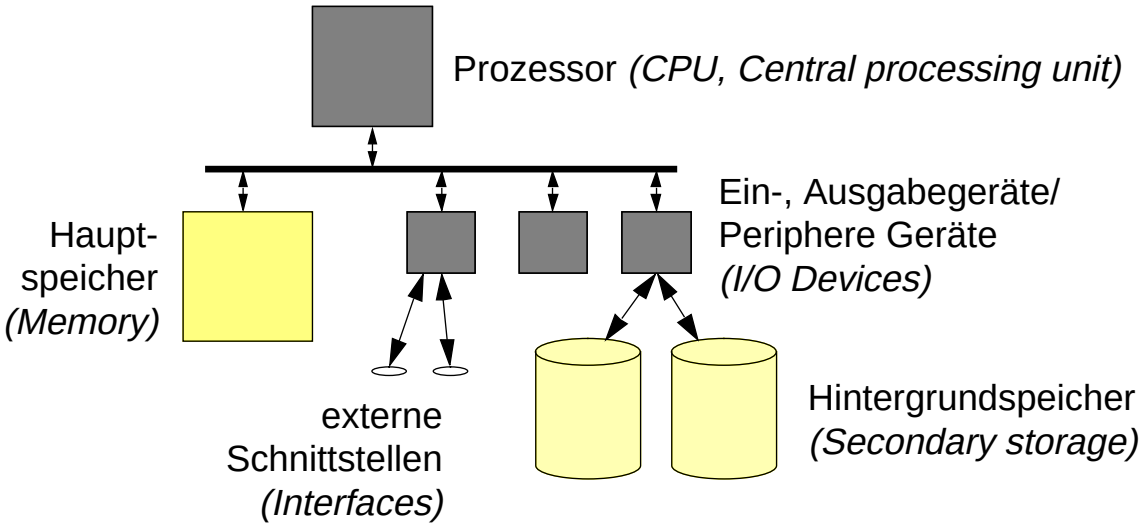
- Programmiermodell: Prozeß
  - ◆ Zerlegung von Anwendungen in Prozesse oder Threads
  - ◆ Ausnutzen von Wartezeiten; Time sharing–Betrieb
  - ◆ Prozeß hat verschiedene Zustände: laufend, bereit, wartend etc.
- Auswahlstrategien für Prozesse
  - ◆ FCFS, SJF, PSJF, RR, MLFB
- Prozeßkommunikation
  - ◆ Pipes, Queues, Signals, Sockets, Shared memory, RPC
- Koordinierung von Prozessen
  - ◆ Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern

D.8 Zusammenfassung (2)

- Gegenseitiger Ausschluß mit Spinlocks
- Klassische Koordinierungsprobleme und deren Lösung mit Semaphoren
  - ◆ Gegenseitiger Ausschluß
  - ◆ Bounded buffers
  - ◆ Leser-Schreiber-Probleme
  - ◆ Philosophenproblem
  - ◆ Schlafende Friseure
- UNIX Systemaufrufe
  - ◆ fork, exec, wait, nice
  - ◆ pipe
  - ◆ msgget, msgsnd, msgrcv
  - ◆ signal, kill
  - ◆ semget, semop, semctl

E Speicherverwaltung

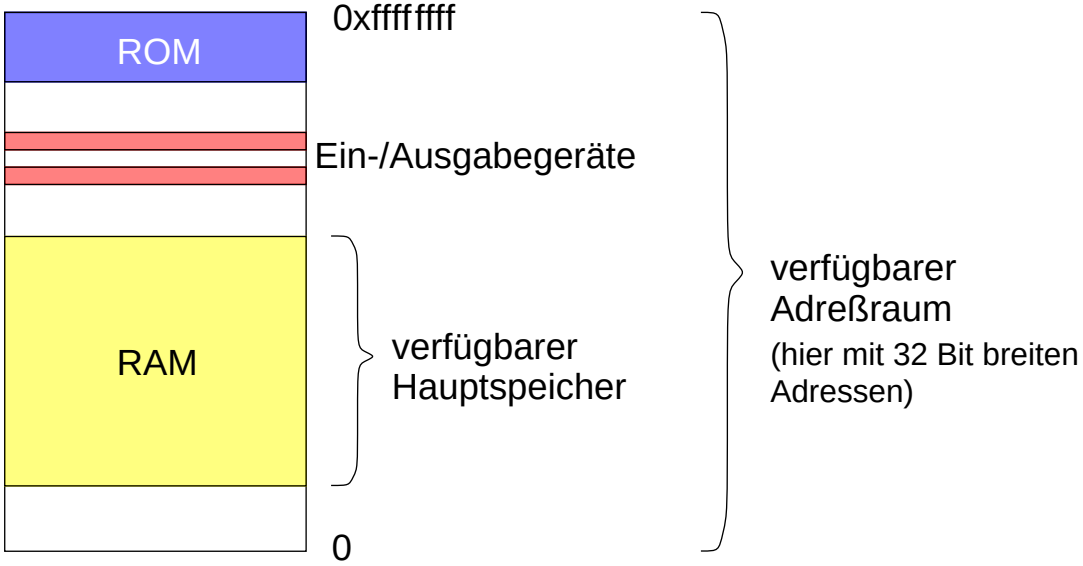
- Betriebsmittel



E.1 Speichervergabe

1 Problemstellung

- Verfügbarer Speicher



1 Problemstellung (2)

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

2 Statische Speicherzuteilung

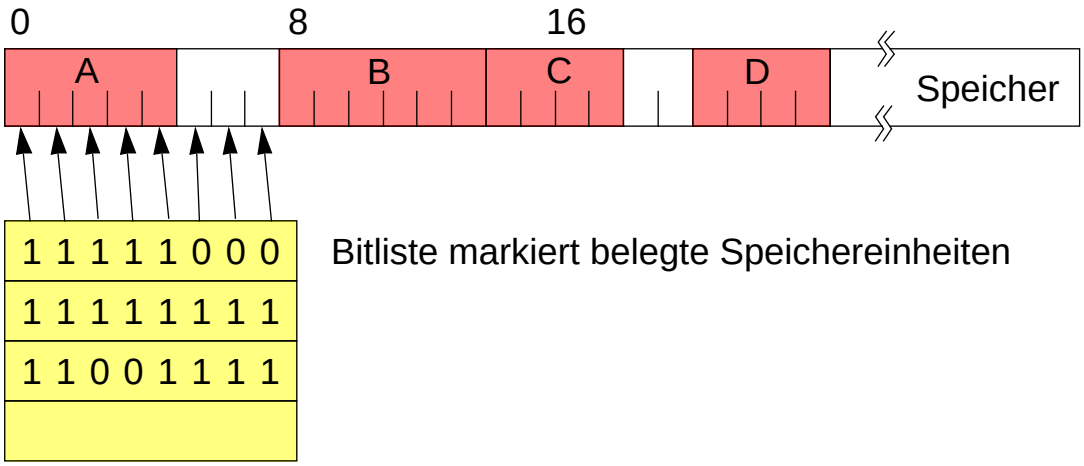
- Feste Bereiche für Betriebssystem und Benutzerprogramm
- ▲ Probleme:
  - Begrenzung anderer Ressourcen
    (z.B. Bandbreite bei Ein-/Ausgabe wg. zu kleiner Systempuffer)
  - Ungenutzter Speicher des Betriebssystems kann von Anwendungsprogramm nicht genutzt werden und umgekehrt
- ★ Dynamische Speicherzuteilung einsetzen

3 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
- ★ Speicherzuteilungsstrategien nötig

4 Freispeicherverwaltung

- Freie (evtl. auch belegte) Segmente des Speichers müssen repräsentiert werden
- Bitlisten

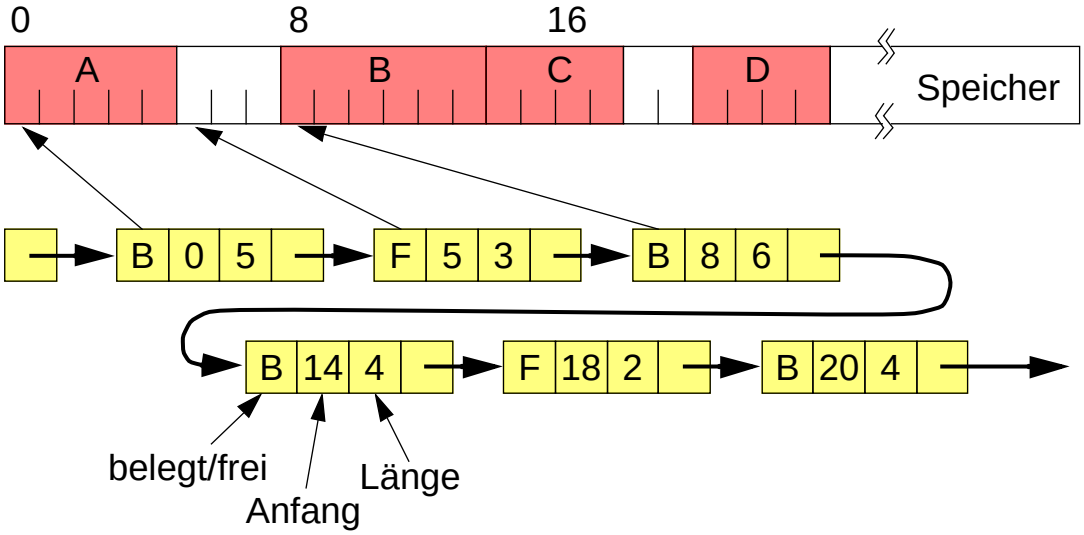


Bitliste markiert belegte Speichereinheiten

Speichereinheiten gleicher Größe (z.B. 1 Byte, 64 Byte, 1024 Byte)

4 Freispeicherverwaltung (2)

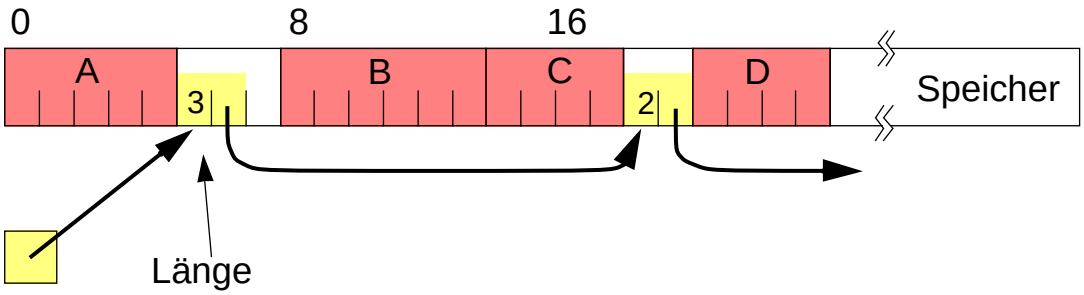
- Verkettete Liste



Repräsentation auch von belegten Segmenten

4 Freispeicherverwaltung (3)

- Verkettete Liste in dem freien Speicher

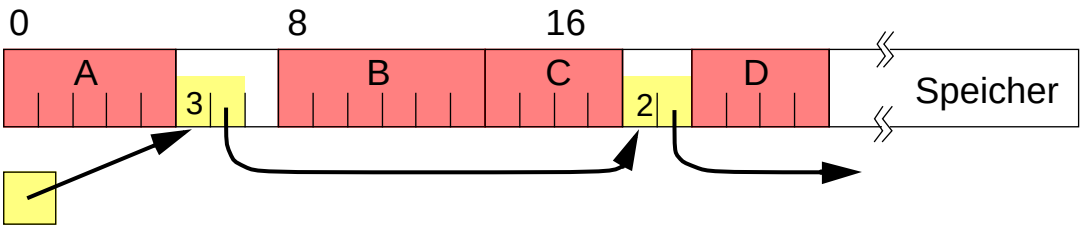


Mindestlückengröße muß garantiert werden

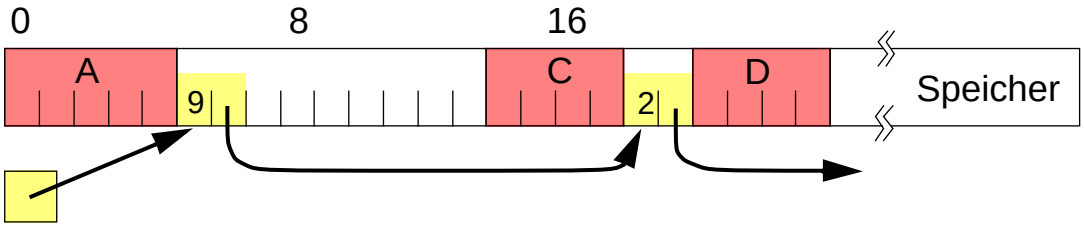
- Zur Effizienzsteigerung eventuell Rückwärtsverkettung nötig
- Repräsentation letztlich auch von der Vergabestrategie abhängig

5 Speicherfreigabe

- Verschmelzung von Lücken



nach Freigabe von B:

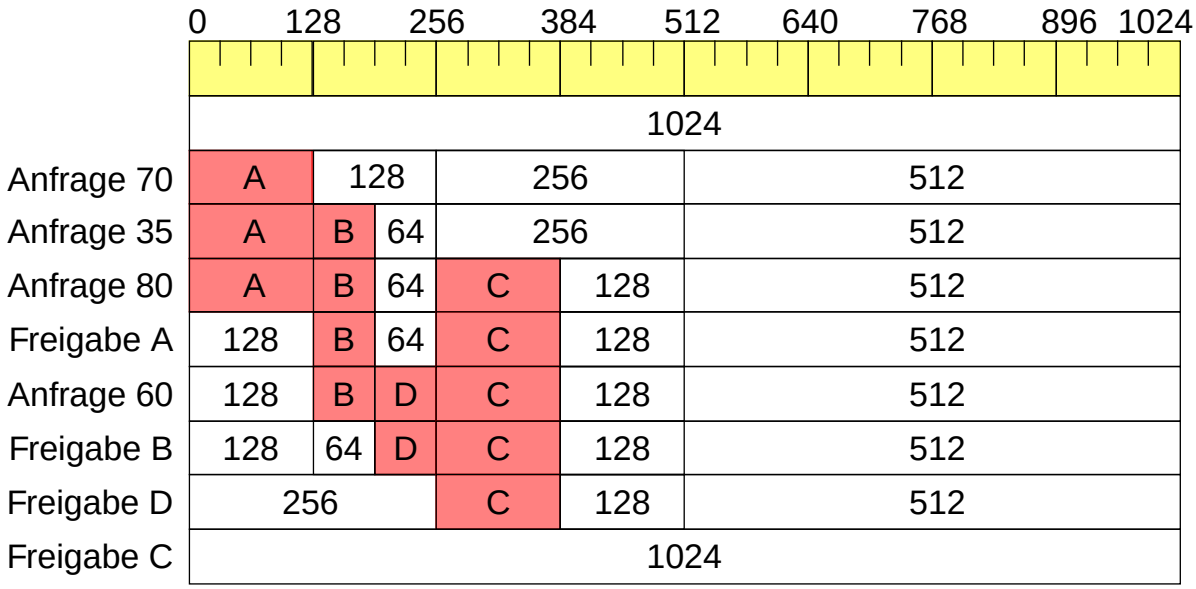


6 Vergabestrategien

- 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
- Worst Fit
  - ◆ größte passende Lücke wird gesucht
- ▲ Probleme:
  - ◆ Speicherverschnitt
  - ◆ zu kleine Lücken

7 Buddy Systeme

- Einteilung in dynamische Bereiche der Größe 2^n



Effiziente Repräsentation der Lücken und effiziente Algorithmen

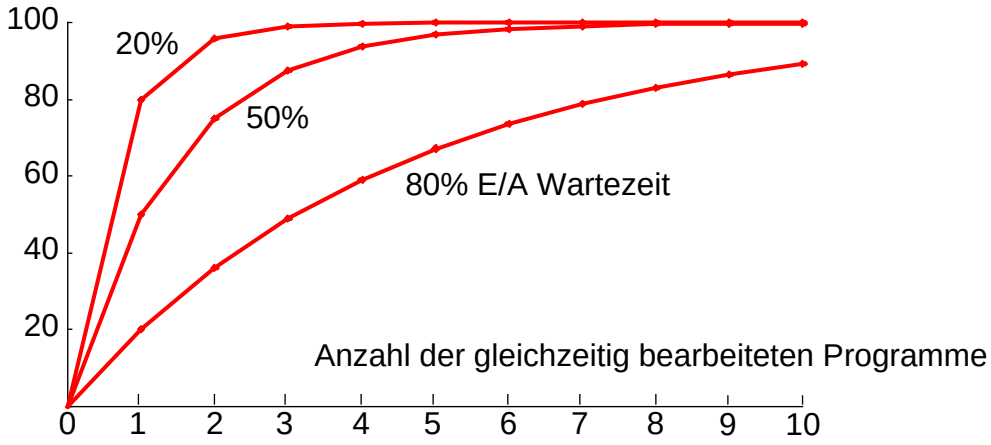
8 Einsatz der Verfahren

- Einsatz im Betriebssystem
  - Verwaltung des Systemspeichers
  - Zuteilung von Speicher an Prozesse und Betriebssystem
- Einsatz innerhalb eines Prozesses
  - Verwaltung des Haldenspeichers *(Heap)*
  - erlaubt dynamische Allokation von Speicherbereichen durch den Prozeß (**`malloc`** und **`free`**)
- Einsatz für Bereiche des Sekundärspeichers
  - Verwaltung bestimmter Abschnitte des Sekundärspeichers z.B. Speicherbereich für Prozeßauslagerungen *(Swap space)*

E.2 Mehrprogrammbetrieb

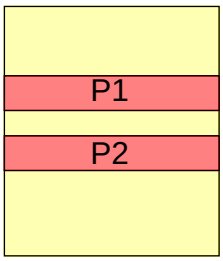
1 Problemstellung

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



1 Problemstellung (2)

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



zwei Prozesse und deren Codesegmente im Speicher

- Relokation von Programmbefehlen (Binaries)
- Ein- und Auslagern von Prozessen
- Hardwareunterstützung

2 Relokation

- 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
  - Programm kann nur an bestimmter Speicherstelle korrekt ablaufen
- Statisches Binden (*Load time*)
  - Beim Laden (Starten) des Programms werden die absoluten Adressen angepaßt (reloziert)
  - Relokationsinformation nötig, die vom Compiler oder Assembler geliefert wird

2 Relokation (2)

- Compilevorgang (Erzeugung der Relokationsinformation)

test.c

C-Programm

```
int main()
{
    exit(0);
}
```

test.s

Assembler

```
main:
 pushl %ebp
 movl %esp,%ebp
 pushl $0
 call exit
 addl $4,%esp
 movl  %ebp,%esp
 popl %ebp
 ret
```

test.o

Bindemodul

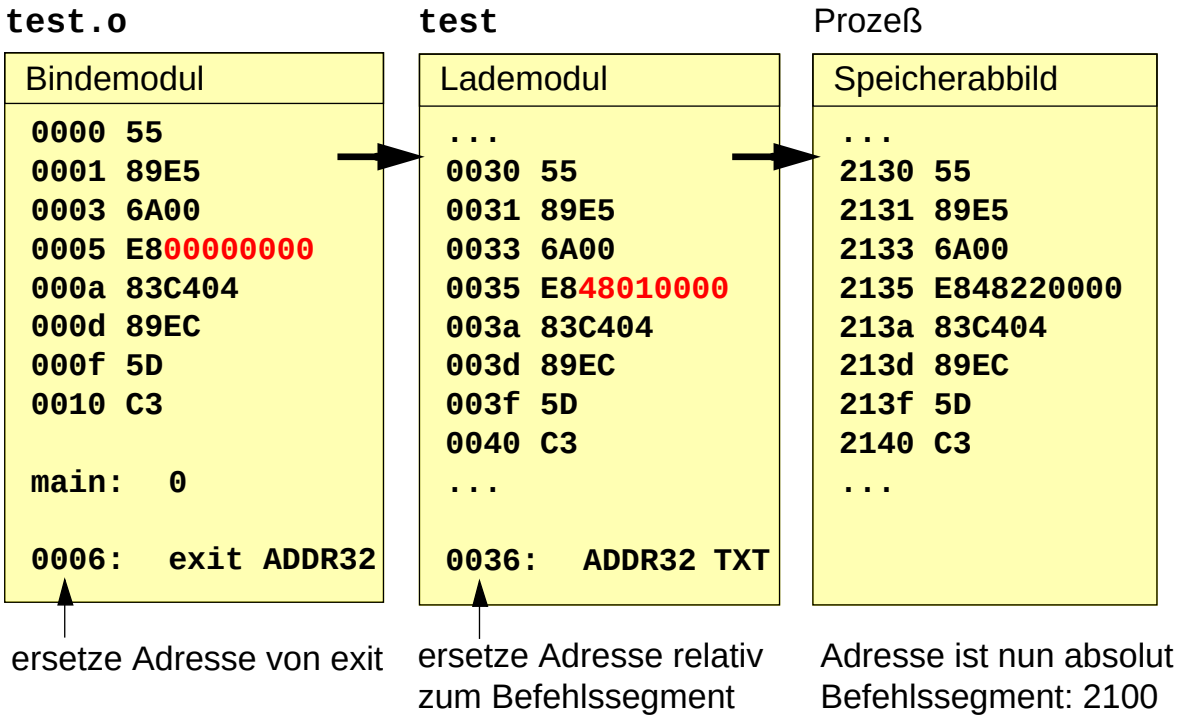
```
0000 55
0001 89E5
0003 6A00
0005 E800000000
000a 83C404
000d 89EC
000f 5D
0010 C3

main:   0

0006:   exit ADDR32
```

Erzeugung beim Compilevorgang

ersetze Adresse von exit

2 Relokation (3)

- Binde- und Ladevorgang

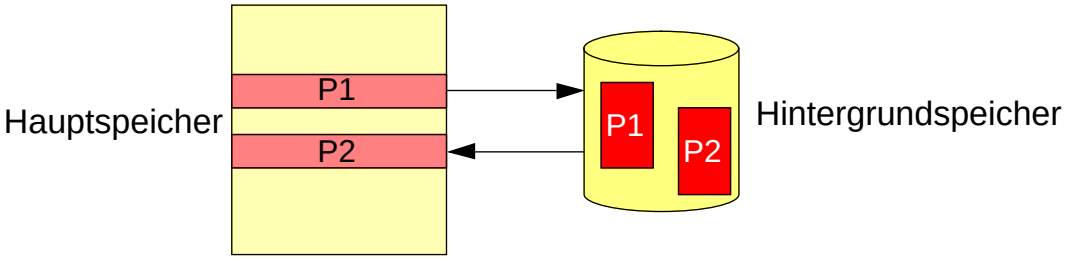


2 Relokation (4)

- Relokationsinformation im Bindemodul
  - ◆ erlaubt das Binden von Modulen in beliebige Programme
- Relokatinsinformation im Lademodul
  - ◆ erlaubt das Laden des Programms an beliebige Speicherstellen
  - ◆ absolute Adressen werden erst beim Laden generiert
- Alternative
  - ◆ Programm benutzt keine absoluten Adressen und kann daher immer an beliebige Speicherstellen geladen werden

3 Ein-, Auslagerung (*Swapping*)

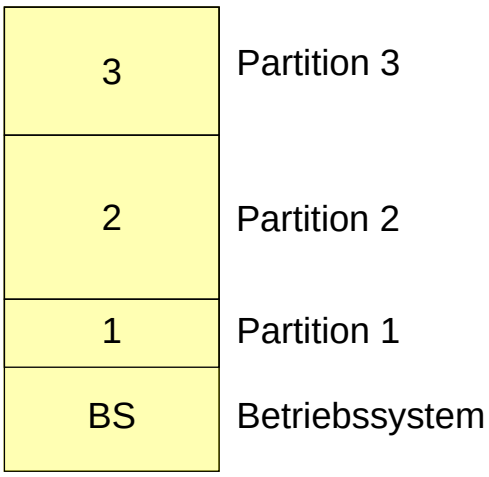
- Segmente eines Prozesses werden auf Hintergrundspeicher ausgelagert und im Hauptspeicher freigegeben
  - z.B. zur Überbrückung von Wartezeiten bei E/A oder Round-Robin Schedulingstrategie
- Einlagern der Segmente in den Hauptspeicher am Ende der Wartezeit



- Aus-, Einlagerzeit ist hoch
  - Latenzzeit der Festplatte
  - Übertragunszeit

3 Ein-, Auslagerung (2)

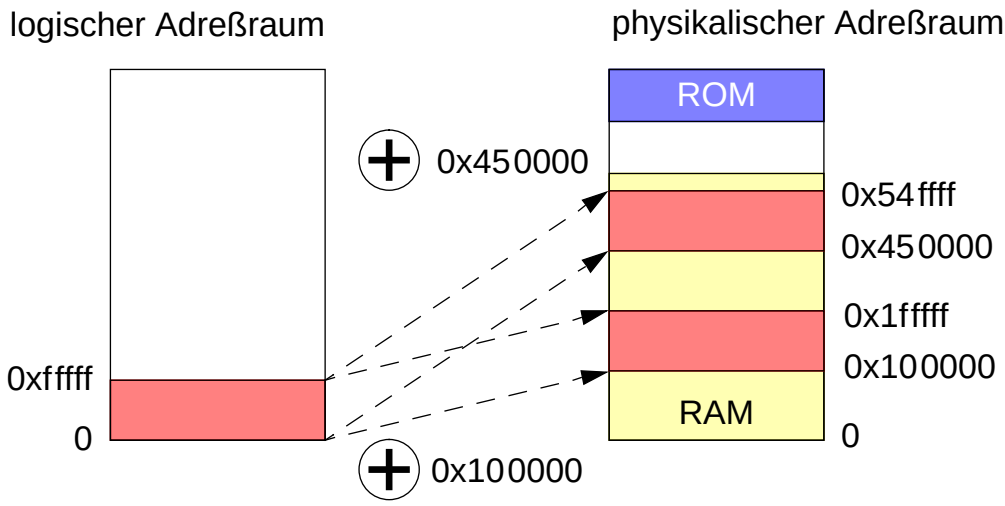
- Prozeß ist statisch gebunden
  - kann nur an gleiche Stelle im Hauptspeicher wieder eingelagert werden
  - Kollisionen mit eventuell neu im Hauptspeicher befindlichen Segmenten
- Mögliche Lösung: Partitionierung des Hauptspeichers



  - In jeder Partition läuft nur ein Prozeß
  - Einlagerung erfolgt wieder in die gleiche Partition
  - Speicher kann nicht optimal genutzt werden

4 Segmentierung

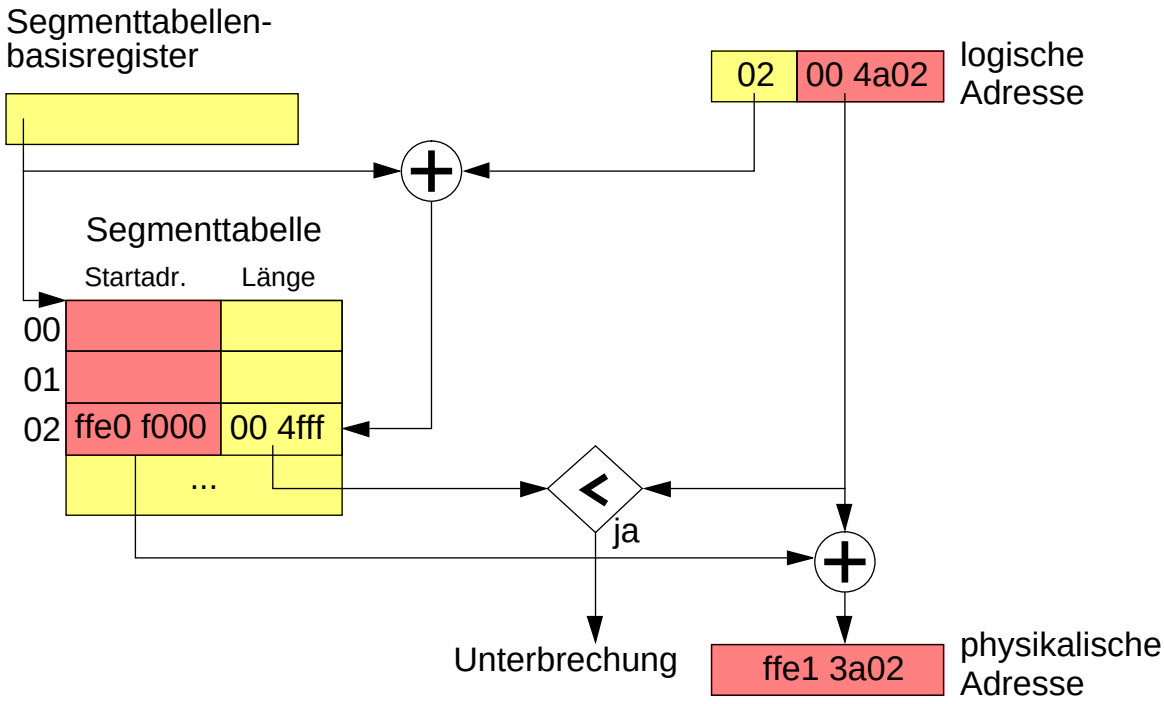
- Hardwareunterstützung: Umsetzung logischer in physikalische Adressen
  - ◆ Prozesse erhalten einen logischen Adreßraum



Das Segment im logischen Adreßraum kann an jeder beliebige Stelle im physikalischen Adressraum liegen.

4 Segmentierung (2)

- Realisierung mit Übersetzungstabelle



4 Segmentierung (3)

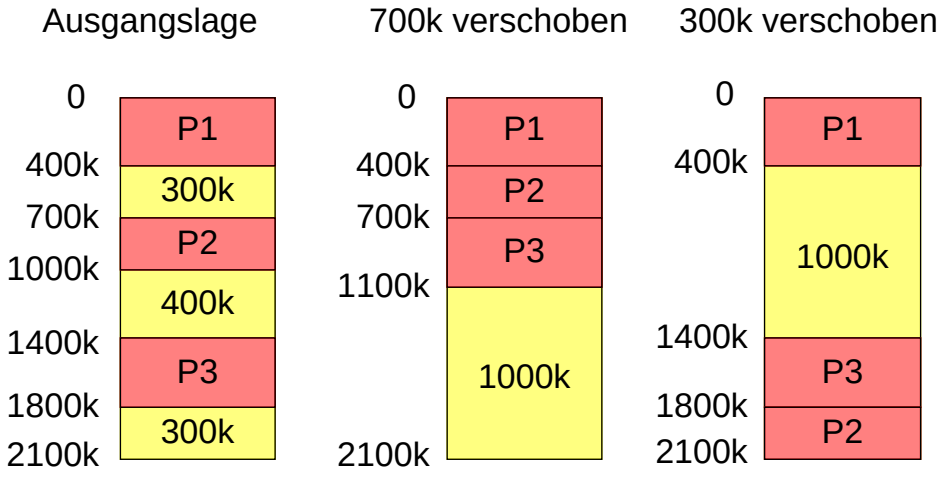
- Hardware wird MMU *(Memory management unit)* genannt
- Schutz vor Segmentübertretung
  - ◆ Unterbrechung zeigt Speicherverletzung an
  - ◆ Programme und Betriebssystem voreinander geschützt
- Prozeßumschaltung durch Austausch der Segmentbasis
  - ◆ jeder Prozeß hat eigene Übersetzungstabelle
- Ein-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

4 Segmentierung (4)

- 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

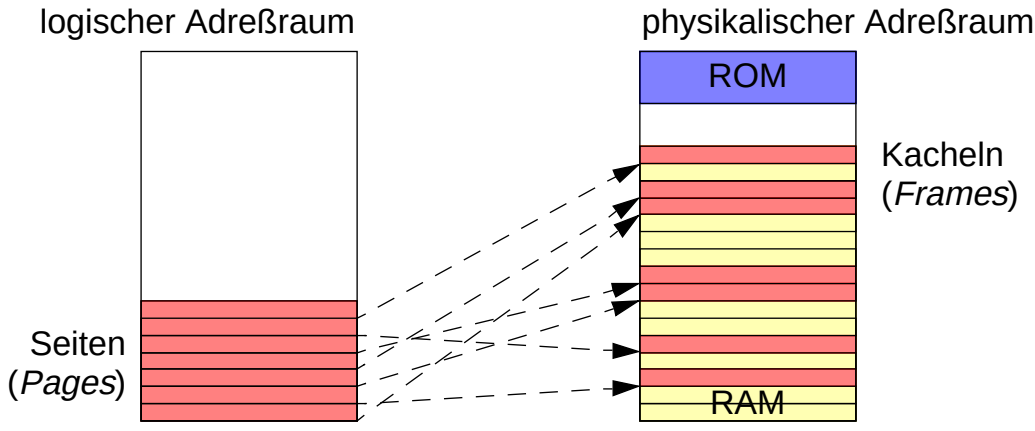
5 Kompaktifizieren

- Verschieben von Segmenten
  - Erzeugen von weniger aber größeren Lücken
  - Verringern des Verschnitts
  - aufwendige Operation, abhängig von der Größe der verschobenen Segmente



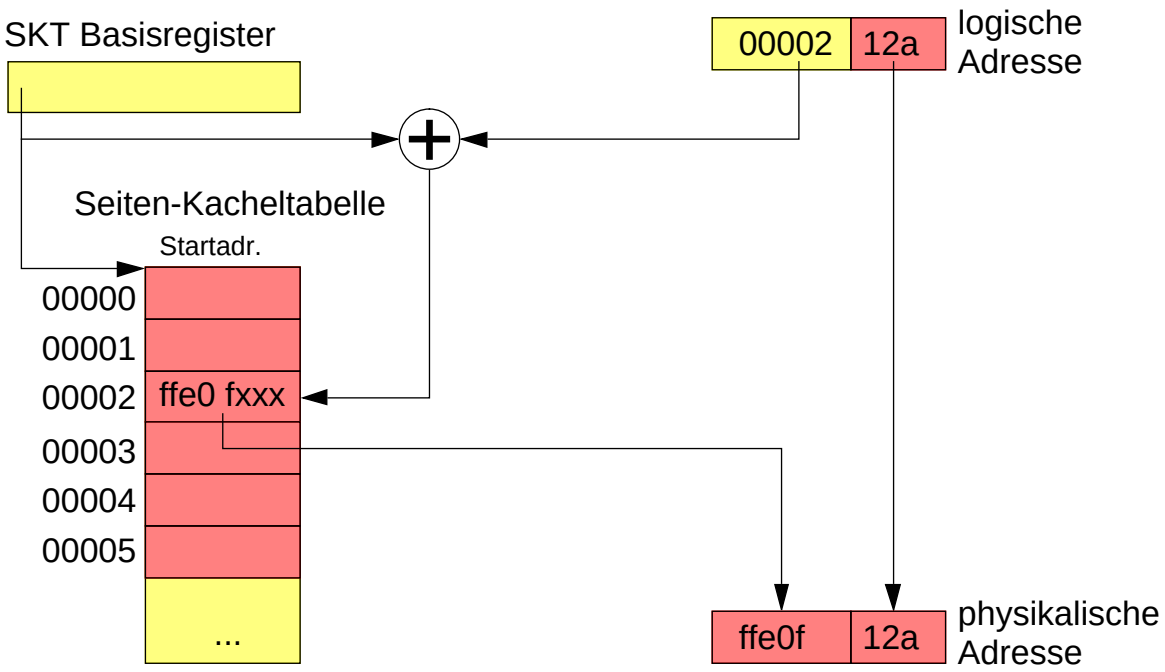
E.3 Seitenadressierung (*Paging*)

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



1 MMU mit Seiten-Kacheltabelle

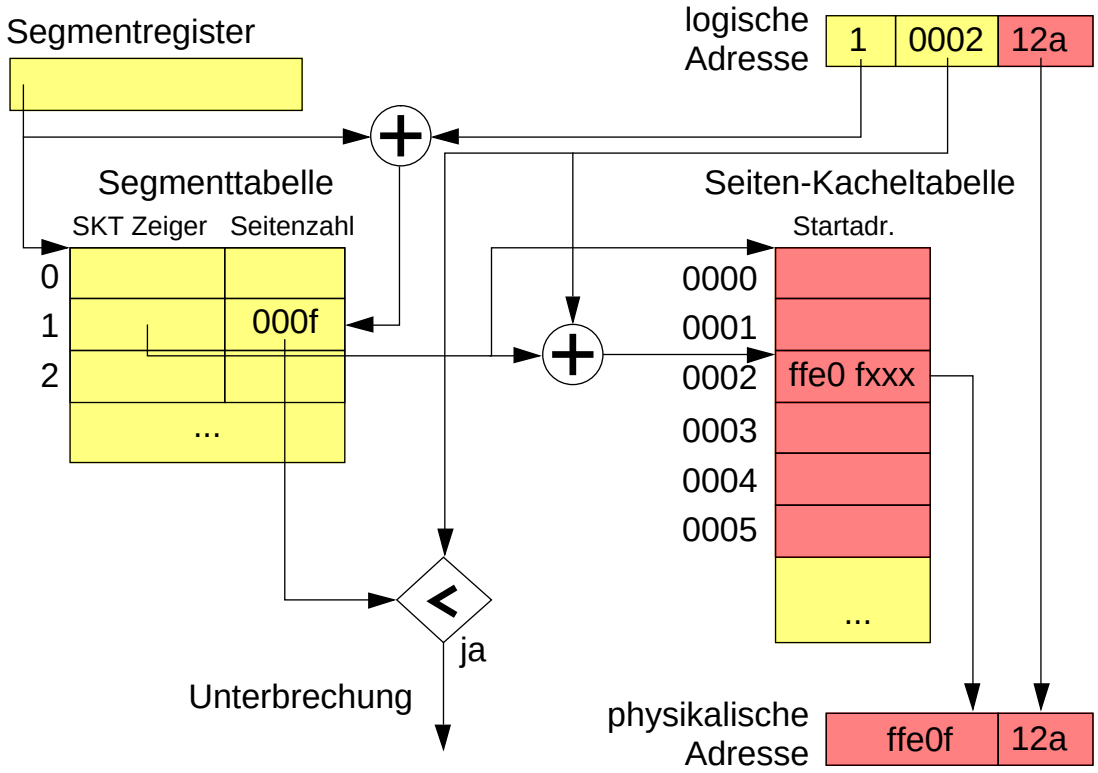
- Tabelle setzt Seiten in Kacheln um



1 MMU mit Seiten-Kacheltabelle (2)

- ▲ 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
- ▲ nur ein „Segment“ pro Kontext
- ★ Kombination mit Segmentierung

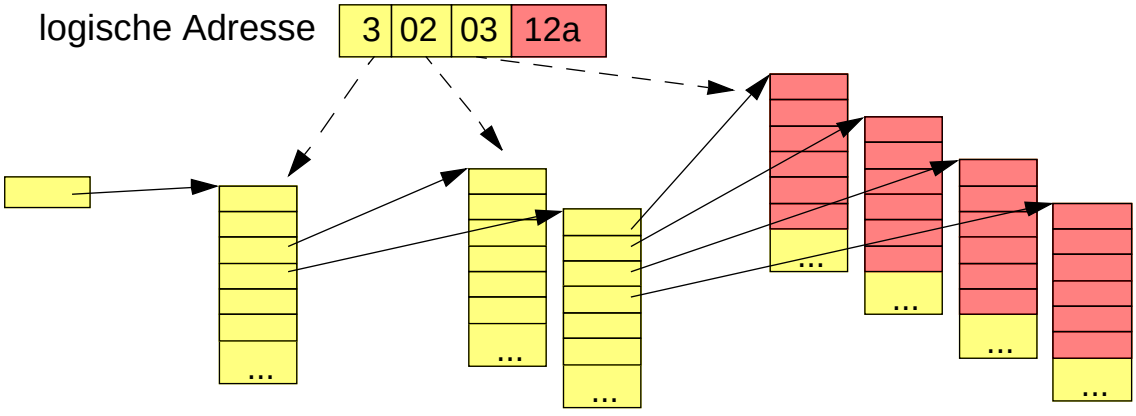
2 Segmentierung und Seitenadressierung



2 Segmentierung und Seitenadressierung (2)

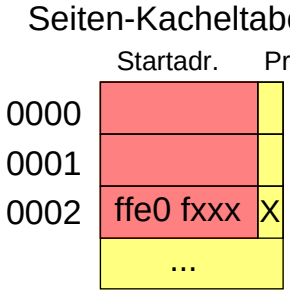
- ▲ noch mehr implizite Speicherzugriffe
- ▲ große Tabellen im Speicher
- ★ Mehrstufige Seitenadressierung mit Ein- und Auslagerung

3 Mehrstufige Seitenadressierung



3 Mehrstufige Seitenadressierung (2)

- Ein- und Auslagerung von Seiten



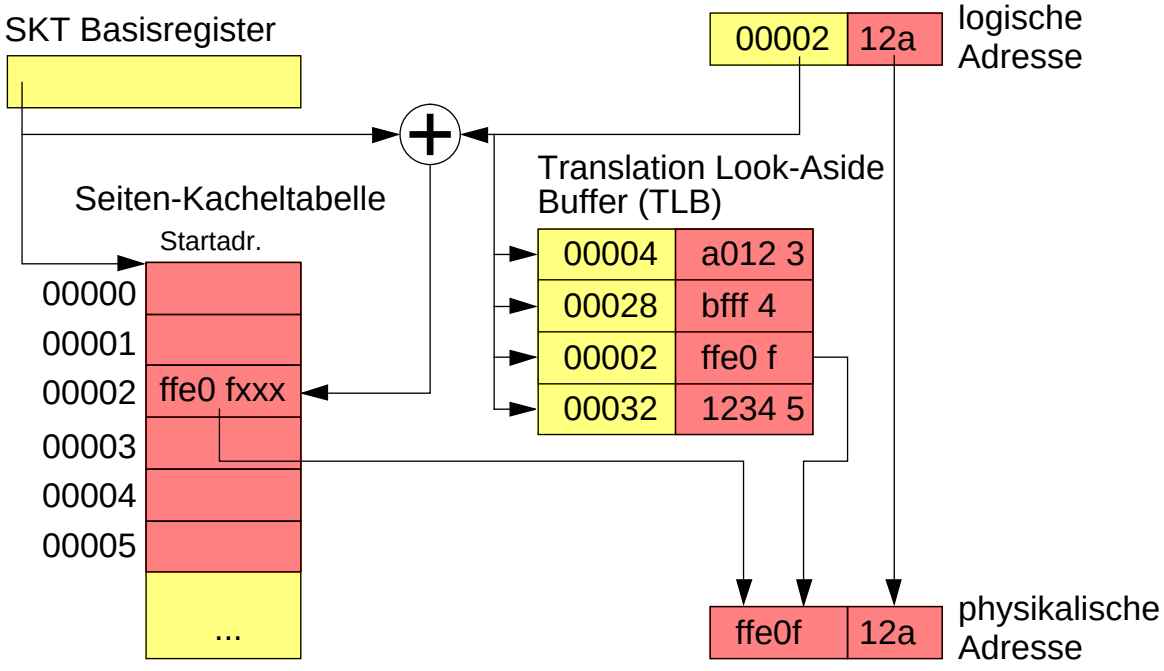
  - Ist das Präsenzbit gesetzt, bleibt alles wie bisher.
  - Ist das Präsenzbit gelöscht, wird eine Unterbrechung ausgelöst (*Page fault*).
  - Die Unterbrechungsbehandlung kann nun für das Laden der Seite vom Hintergrundspeicher sorgen und den Speicherzugriff danach wiederholen (benötigt HW Support in der CPU).

- Präsenzbit auch für jeden Eintrag in den höheren Stufen
  - Tabellen sind aus- und einlagerbar

- Noch mehr implizite Speicherzugriffe

4 Translation Look-Aside Buffer

- Schneller Registersatz wird konsultiert bevor auf die SKT zugegriffen wird

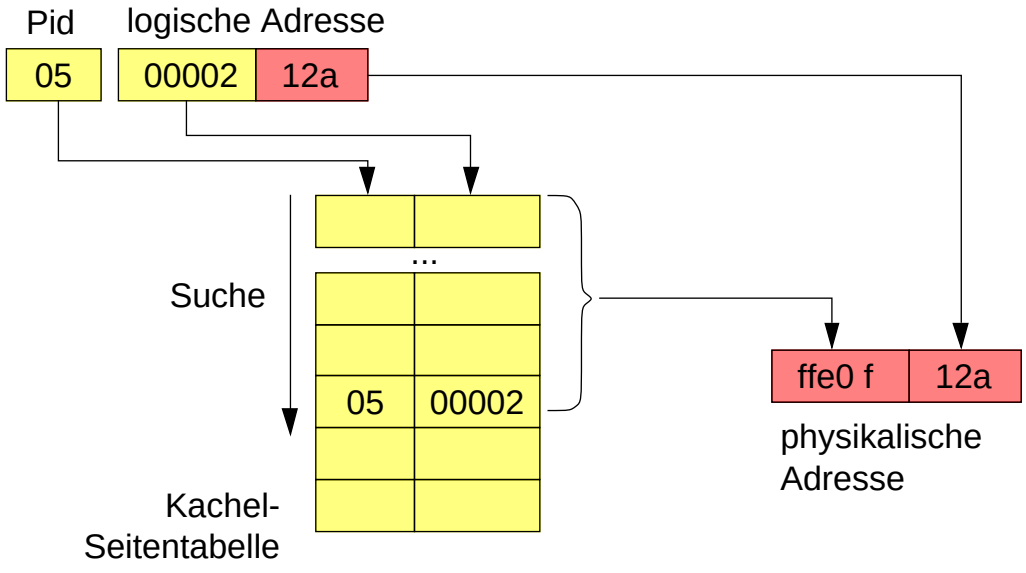


4 Translation Look-Aside Buffer (2)

- Schneller Zugriff auf Seitenabbildung, falls Information im voll-assoziativen Speicher des TLB
  - ◆ keine impliziten Speicherzugriffe nötig
- Bei Kontextwechseln muß TLB gelöscht werden (*Flush*)
- Bei Zugriffen auf eine nicht im TLB enthaltene Seite wird die entsprechende Zugriffsinformation in den TLB eingetragen
  - ◆ Ein alter Eintrag muß zur Ersetzung ausgesucht werden
- TLB Größe
  - ◆ Pentium: Daten TLB = 64, Code TLB = 32, Seitengröße 4K
  - ◆ Sparc V9: Daten TLB = 64, Code TLB = 64, Seitengröße 8K
  - ◆ Größere TLBs bei den üblichen Taktraten zur Zeit nicht möglich

5 Invertierte Seiten-Kacheltabelle

- Zum Umsetzen der Adressen nur Abbildung der belegten Kacheln nötig
  - ◆ eine Tabelle, die zu jeder Kachel die Seitenabbildung hält



5 Invertierte Seiten-Kacheltabelle (2)

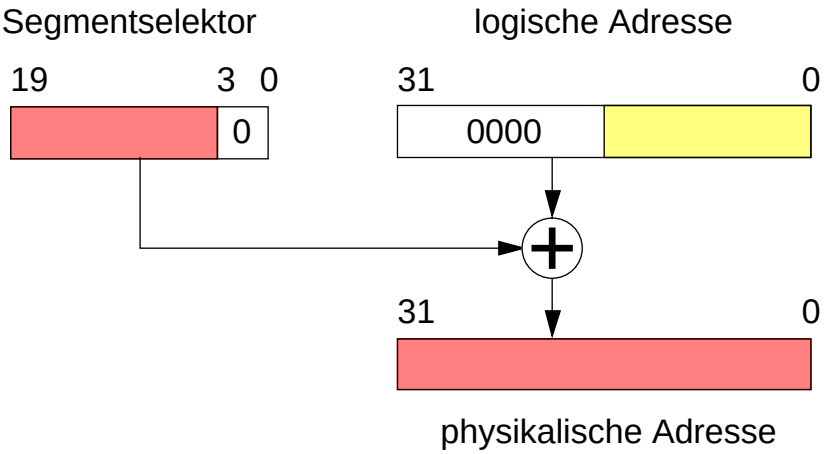
- Vorteile
  - ◆ wenig Platz zur Speicherung der Abbildung notwendig
  - ◆ Tabelle kann immer im Hauptspeicher gehalten werden
- Nachteile
  - ◆ prozeßlokale SKT zusätzlich nötig für Seiten, die ausgelagert sind
    - diese können aber ausgelagert werden
  - ◆ Suche in der KST ist aufwendig
    - Einsatz von Assoziativspeichern und Hashfunktionen

E.4 Fallstudie: Pentium

- Physikalische Adresse
  - ◆ 32 bit breit
- Segmente
  - ◆ CS – Codesegment: enthält Instruktionen
  - ◆ DS – Datensegment
  - ◆ SS – Stacksegment
  - ◆ ES, FS, GS – zusätzliche Segmente
  - ◆ Befehle beziehen sich auf eines oder mehrere der Segmente
- Segmentadressierung
  - ◆ Segmentselektor zur Auswahl eines Segments:
    16 bit bezeichnen das Segment

1 Real Mode Adressierung

- Adreßgenerierung im Real Mode
  - 16 bit breiter Segmentselektor wird als 20 bit breite Adresse interpretiert und auf die logische Adresse addiert



2 Protected Mode Adressierung

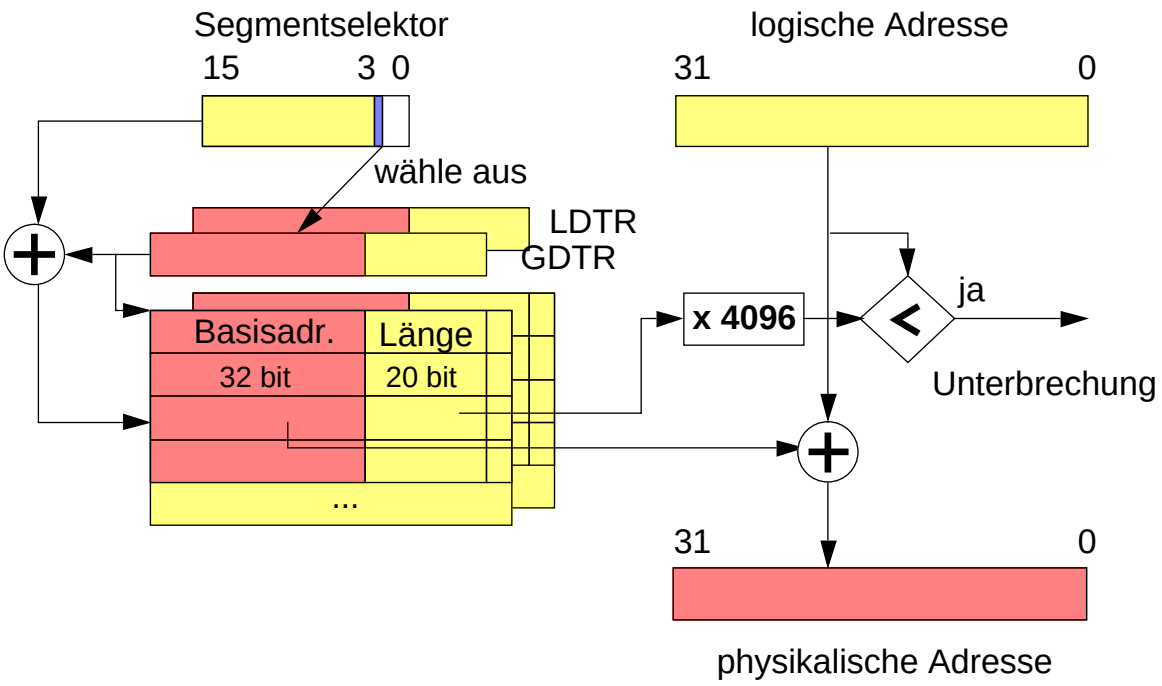
- Vier Betriebsmodi (Stufen von Privilegien)
  - Stufe 0: höchste Privilegien (privilegierte Befehle, etc.): BS Kern
  - Stufe 1: BS Treiber
  - Stufe 2: BS Erweiterungen
  - Stufe 3: Benutzerprogramme
  - Speicherverwaltung kann nur in Stufe 0 konfiguriert werden
- Segmentselektoren enthalten Privlegierungsstufe
  - Stufe des Codesegments entscheidet über Zugriffserlaubnis
- Segmentselektoren werden als Indizes interpretiert
  - Tabellen von Segmentdeskriptoren
    - Globale Deskriptor Tabelle
    - Lokale Deskriptor Tabelle

2 Protected Mode Adressierung (2)

- Deskriptortabelle
  - ◆ enthält bis zu 8192 Segmentdeskriptoren
    - physikalische Basisadresse
    - Längenangabe
    - Granularität (Angaben für Bytes oder Seiten)
    - Präsenzbit
    - Privilegierungsstufe
  - ◆ globale Deskriptortabelle für alle Prozesse zugänglich (Register GDTR)
  - ◆ lokale Deskriptortabelle pro Prozeß möglich
    (Register LDTR gehört zum Prozeßkontext)

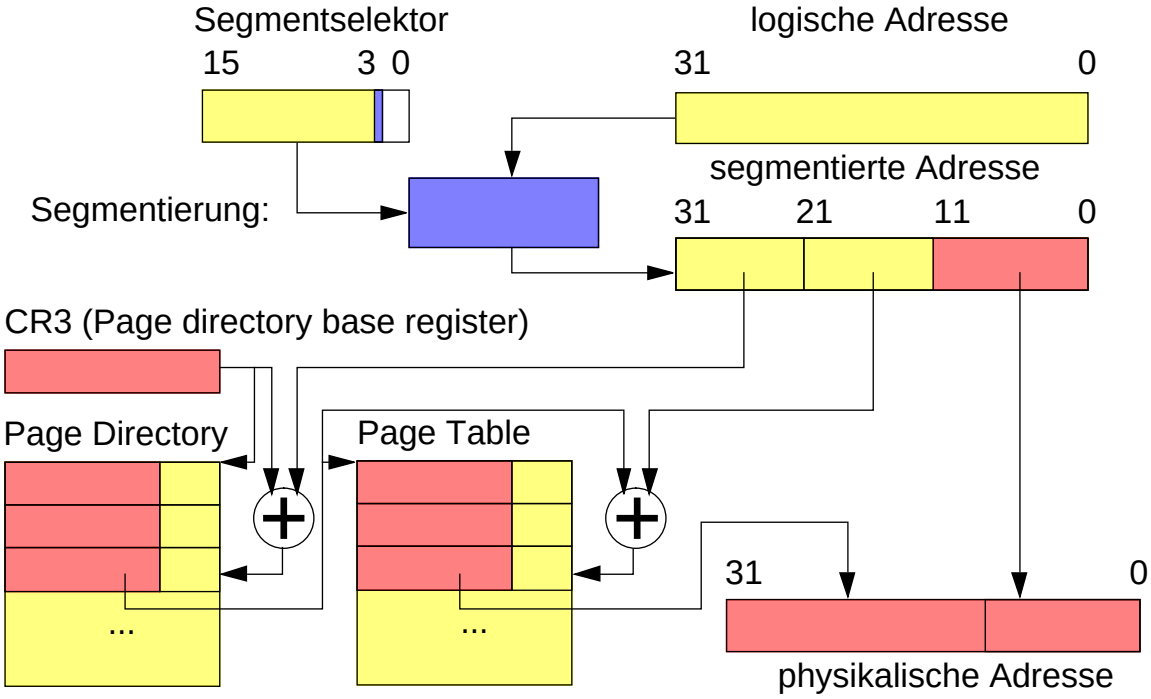
3 Adreßberechnung bei Segmentierung

- Verwendung der Protected mode Adressierung



4 Adreßberechnung bei Paging

- Seitenadressierung wird der Segmentierung nachgeschaltet



4 Adreßberechnung bei Paging

- Zweistufige Seitenadressierung
  - Directory — Page table
  - Seitengröße fest auf 4096 Bytes
- Inhalt des Seitendeskriptor
  - Kacheladresse
  - Dirty Bit: Seite wurde beschrieben
  - Access Bit: Seite wurde gelesen oder geschrieben
  - Schreibschutz: Seite nur lesbar
  - Präsenzbit: Seite ausgelagert (31 Bits für BS-Informationen nutzbar)
  - Kontrolle des Prozessorcaches
- Getrennte TLBs für Codesegment und Datensegmente
  - 64 Einträge für Datenseiten; 32 Einträge für Codeseiten

E.5 Gemeinsamer Speicher (*Shared Memory*)

- Speicher, der mehreren Prozessen zur Verfügung steht
  - gemeinsame Segmente (gleiche Einträge in verschiedenen Segmenttabellen)
  - gemeinsame Seiten (gleiche Einträge in verschiedenen SKTs)
  - gemeinsame Seitenbereiche (gemeinsames Nutzen einer SKT bei mehrstufigen Tabellen)
- Gemeinsamer Speicher wird beispielsweise benutzt für
  - Kommunikation zwischen Prozessen
  - gemeinsame Befehlssegmente

E.5 Gemeinsamer Speicher (2)

- Systemaufrufe unter Solaris 2.5
  - Erzeugen bzw. Holen eines gemeinsamen Speichersegments

```
int shmget( key_t key, int size, int shmflg );
```

  - Einblenden und Ausblenden des Segments in den Speicher

```
void *shmat( int shmid, void *shmaddr, int shmflg );
int shmdt( void *shmaddr );
```

  - Kontrolloperation

```
int shmctl( int shmid, int cmd, struct shmid_ds *buf );
```