

6.5 Semaphor (4)

V-Operation

Unterbrechungen sperren

$*s = *s + 1$

alle Prozess aus der Warteschlange
in den Zustand *bereit* versetzen

Unterbrechungen freigeben

Prozessumschalter ansteuern

- ◆ Prozesse probieren immer wieder, die P-Operation erfolgreich abzuschließen
- ◆ Schedulingstrategie entscheidet über Reihenfolge und Fairness
 - leichte Ineffizienz durch Aufwecken aller Prozesse
 - mit Einbezug der Schedulingstrategie effizientere Implementierungen möglich

6.5 Semaphor (5)

- ★ Vorteile einer Semaphor-Implementierung im Betriebssystem
 - ◆ Einbeziehen des Schedulers in die Semaphor-Operationen
 - ◆ kein aktives Warten; Ausnutzen der Blockierzeit durch andere Prozesse

■ Implementierung einer Synchronisierung

- ◆ zwei Prozesse P_1 und P_2
- ◆ Anweisung S_1 in P_1 soll vor Anweisung S_2 in P_2 stattfinden

```
int lock= 0;
```

```
...                               Prozess 1
S1;
V( &lock );
...
```

```
...                               Prozess 2
P( &lock );
S2;
...
```

★ Zählende Semaphore

6.5 Semaphor (6)

- Abstrakte Beschreibung von zählenden Semaphoren (PV System)
 - ◆ für jede Operation wird eine Bedingung angegeben
 - falls Bedingung nicht erfüllt, wird die Operation blockiert
 - ◆ für den Fall, dass die Bedingung erfüllt wird, wird eine Anweisung definiert, die ausgeführt wird
- Beispiel: zählende Semaphore

Operation	Bedingung	Anweisung
P(S)	$S > 0$	$S := S - 1$
V(S)	TRUE	$S := S + 1$

7 Klassische Koordinierungsprobleme

- Reihe von bedeutenden Koordinierungsproblemen
 - ◆ Gegenseitiger Ausschluss (*Mutual exclusion*)
 - nur ein Prozess darf bestimmte Anweisungen ausführen
 - ◆ Puffer fester Größe (*Bounded buffers*)
 - Blockieren der lesenden und schreibenden Prozesse, falls Puffer leer oder voll
 - ◆ Leser-Schreiber-Problem (*Reader-writer problem*)
 - Leser können nebenläufig arbeiten; Schreiber darf nur alleine zugreifen
 - ◆ Philosophenproblem (*Dining-philosopher problem*)
 - im Kreis sitzende Philosophen benötigen das Besteck der Nachbarn zum Essen
 - ◆ Schlafende Friseure (*Sleeping-barber problem*)
 - Friseure schlafen solange keine Kunden da sind

7.1 Gegenseitiger Ausschluss

■ Semaphore

- ◆ eigentlich reicht ein Semaphore mit zwei Zuständen: binärer Semaphore

```
void P( int *s )
{
    while( *s == 0 );
    *s= 0;
}
```

atomare Funktion

```
void V( int *s )
{
    *s= 1;
}
```

atomare Funktion

- ◆ zum Teil effizienter implementierbar

7.1 Gegenseitiger Ausschluss (2)

■ Abstrakte Beschreibung: binäre Semaphore

Operation	Bedingung	Anweisung
P(S)	$S \neq 0$	$S := 0$
V(S)	TRUE	$S := 1$

7.1 Gegenseitiger Ausschluss (3)

- ▲ Problem der Klammerung kritischer Abschnitte
 - ◆ Programmierer müssen Konvention der Klammerung einhalten
 - ◆ Fehler bei Klammerung sind fatal

```
P( &lock );
```

```
... /* critical sec. */
```

```
P( &lock );
```

führt zu Verklemmung (Deadlock)

```
V( &lock );
```

```
... /* critical sec. */
```

```
V( &lock );
```

führt zu unerwünschter Nebenläufigkeit

7.1 Gegenseitiger Ausschluss (3)

- Automatische Klammerung wünschenswert
 - ◆ Beispiel: Java

```
synchronized( lock ) {
```

```
... /* critical sec. */
```

```
}
```

7.2 Bounded Buffers

- Puffer fester Größe
 - ◆ mehrere Prozesse lesen und beschreiben den Puffer
 - ◆ beispielsweise Erzeuger und Verbraucher (Erzeuger-Verbraucher-Problem)
(z.B. Erzeuger liest einen Katalog; Verbraucher zählt Zeilen;
Gesamtanwendung zählt Einträge in einem Katalog)
 - ◆ UNIX-Pipe ist solch ein Puffer
- Problem
 - ◆ Koordinierung von Leser und Schreiber
 - gegenseitiger Ausschluss beim Pufferzugriff
 - Blockierung des Lesers bei leerem Puffer
 - Blockierung des Schreibers bei vollem Puffer

7.2 Bounded Buffers (2)

- Implementierung mit zählenden Semaphoren
 - ◆ zwei Funktionen zum Zugriff auf den Puffer
 - `put` stellt Zeichen in den Puffer
 - `get` liest ein Zeichen vom Puffer
 - ◆ Puffer wird durch ein Feld implementiert, das als Ringpuffer wirkt
 - zwei Integer-Variablen enthalten Feldindizes auf den Anfang und das Ende des Ringpuffers
 - ◆ ein Semaphor für den gegenseitigen Ausschluss
 - ◆ je einen Semaphor für das Blockieren an den Bedingungen „Puffer voll“ und „Puffer leer“
 - Semaphor `full` zählt wieviele Zeichen noch in den Puffer passen
 - Semaphor `empty` zählt wieviele Zeichen im Puffer sind

7.2 Bounded Buffers (3)

```
char buffer[N];
int inslot= 0, outslot= 0;
semaphor mutex= 1, empty= 0, full= N;
```

```
void put( char c )
{
    P( &full );
    P( &mutex );
    buffer[inslot]= c;
    if( ++inslot >= N )
        inslot= 0;
    V( &mutex );
    V( &empty );
}
```

```
char get( void )
{
    char c;

    P( &empty );
    P( &mutex );
    c= buffer[outslot];
    if( ++outslot >= N )
        outslot= 0;
    V( &mutex );
    V( &full );
    return c;
}
```

7.3 Erstes Leser-Schreiber-Problem

- Lesende und schreibende Prozesse
 - ◆ Leser können nebenläufig zugreifen (Leser ändern keine Daten)
 - ◆ Schreiber können nur exklusiv zugreifen (Daten sonst inkonsistent)
- Erstes Leser-Schreiber-Problem (nach *Courtois* et.al. 1971)
 - ◆ Kein Leser soll warten müssen, es sei denn ein Schreiber ist gerade aktiv
- Realisierung mit zählenden (binären) Semaphoren
 - ◆ Semaphor für gegenseitigen Ausschluss von Schreibern untereinander und von Schreiber gegen Leser: **write**
 - ◆ Zählen der nebenläufig tätigen Leser: Variable **readcount**
 - ◆ Semaphor für gegenseitigen Ausschluss beim Zugriff auf **readcount**: **mutex**

7.3 Erstes Leser-Schreiber-Problem (2)

```
semaphore mutex= 1, write= 1;  
int readcount= 0;
```

... Leser

```
P( &mutex );  
if( ++readcount == 1 )  
    P( &write );  
V( &mutex );  
  
... /* reading */  
  
P( &mutex );  
if( --readcount == 0 )  
    V( &write );  
V( &mutex );  
...
```

... Schreiber

```
P( &write );  
  
... /* writing */  
  
V( &write );  
...
```

7.3 Erstes Leser-Schreiber-Problem (3)

- Vereinfachung der Implementierung durch spezielle Semaphore?
 - ◆ PV-Chunk Semaphore:
 - führen quasi mehrere P- oder V-Operationen atomar aus
 - zweiter Parameter gibt Anzahl an
- Abstrakte Beschreibung für PV-Chunk Semaphore:

Operation	Bedingung	Anweisung
P(S, k)	$S \geq k$	$S := S - k$
V(S, k)	TRUE	$S := S + k$

7.3 Erstes Leser-Schreiber-Problem (4)

- Implementierung mit PV-Chunk:
 - ◆ Annahme: es gibt maximal N Leser

```
PV_chunk_semaphore mutex= N;
```

Leser

```
...  
Pc( &mutex, 1 );  
  
... /* reading */  
  
Vc( &mutex, 1 );  
...
```

Schreiber

```
...  
Pc( &mutex, N );  
  
... /* writing */  
  
Vc( &mutex, N );  
...
```

7.4 Zweites Leser-Schreiber-Problem

- Wie das erste Problem aber: (nach Courtois et.al., 1971)
 - ◆ Schreiboperationen sollen so schnell wie möglich durchgeführt werden
- Implementierung mit zählenden Semaphoren
 - ◆ Zählen der nebenläufig tätigen Leser: Variable **readcount**
 - ◆ Zählen der anstehenden Schreiber: Variable **writcount**
 - ◆ Semaphor für gegenseitigen Ausschluss beim Zugriff auf **readcount**: **mutexR**
 - ◆ Semaphor für gegenseitigen Ausschluss beim Zugriff auf **writcount**: **mutexW**
 - ◆ Semaphor für gegenseitigen Ausschluss von Schreibern untereinander und von Schreibern gegen Leser: **write**
 - ◆ Semaphor für den Ausschluss von Lesern, falls Schreiber vorhanden: **read**
 - ◆ Semaphor zum Klammern des Leservorspanns: **mutex**

7.4 Zweites Leser-Schreiber-Problem (2)

```
semaphore mutexR= 1, mutexW= 1, mutex= 1;  
semaphore write= 1, read= 1;  
int readcount= 0, writecount= 0;
```

Bitte nicht
versuchen, dies
zu verstehen!!

Leser

```
...  
P( &mutex ); P( &read );  
P( &mutexR );  
if( ++readcount == 1 )  
    P( &write );  
V( &mutexR );  
V( &read ); V( &mutex );  
  
... /* reading */  
  
P( &mutexR );  
if( --readcount == 0 )  
    V( &write );  
V( &mutexR );  
...
```

Schreiber

```
...  
P( &mutexW );  
if( ++writecount == 1 )  
    P( &read );  
V( &mutexW );  
P( &write );  
  
... /* writing */  
  
V( &write );  
P( &mutexW );  
if( --writecount == 0 )  
    V( &read );  
V( &mutexW );  
...
```

7.4 Zweites Leser-Schreiber-Problem (3)

- Vereinfachung der Implementierung durch spezielle Semaphore?
 - ◆ Up-Down-Semaphore:
 - zwei Operationen *up* und *down*, die den Semaphor hoch- und runterzählen
 - Nichtblockierungsbedingung für beide Operationen, definiert auf einer Menge von Semaphoren

7.4 Zweites Leser-Schreiber-Problem (4)

- Abstrakte Beschreibung für Up-down-Semaphore

Operation	Bedingung	Anweisung
$\text{up}(S, \{S_i\})$	$\sum_i S_i \geq 0$	$S := S + 1$
$\text{down}(S, \{S_i\})$	$\sum_i S_i \geq 0$	$S := S - 1$

7.4 Zweites Leser-Schreiber-Problem (5)

- Implementierung mit Up-Down-Semaphoren:

```
up_down_semaphore mutexw= 0, reader= 0, writer= 0;
```

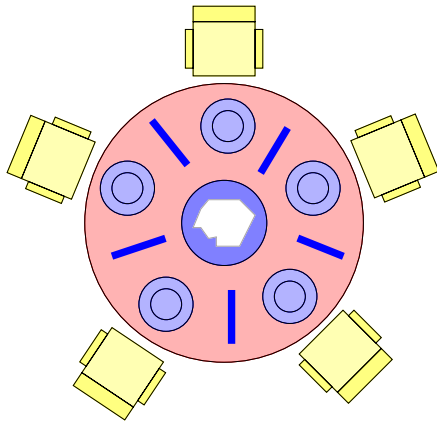
```
...                               Leser
down( &reader, 1, &writer );
... /* reading */
up( &reader, 0 );
...
```

```
...                               Schreiber
down( &writer, 0 );
down( &mutexw,
      2, &mutexw, &reader );
... /* writing */
up( &mutexw, 0 );
up( &writer, 0 );
...
```

- ◆ Zähler für Leser: **reader** (zählt negativ)
- ◆ Zähler für anstehende Schreiber: **writer** (zählt negativ)
- ◆ Semaphore für gegenseitigen Ausschluss der Schreiber: **mutexw**

7.5 Philosophenproblem

■ Fünf Philosophen am runden Tisch



- ◆ Philosophen denken oder essen
"The life of a philosopher consists of an alternation of thinking and eating."
(Dijkstra, 1971)
- ◆ zum Essen benötigen sie zwei Gabeln, die jeweils zwischen zwei benachbarten Philosophen abgelegt sind

▲ Problem

- ◆ Gleichzeitiges Belegen mehrerer Betriebsmittel (hier Gabeln)
- ◆ Verklemmung und Aushungerung

7.5 Philosophenproblem (2)

■ Naive Implementierung

- ◆ eine Semaphore pro Gabel

```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

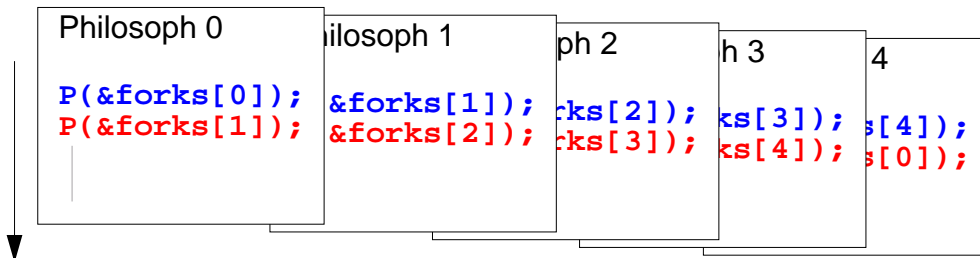
Philosoph i , $i \in [0,4]$

```
while( 1 ) {  
    ... /* think */  
  
    P( &forks[i] );  
    P( &forks[(i+1)%5] );  
  
    ... /* eat */  
  
    V( &forks[i] );  
    V( &forks[(i+1)%5] );  
}
```

7.5 Philosophenproblem (3)

■ Problem der Verklemmung

- ◆ alle Philosophen nehmen gleichzeitig die linke Gabel auf und versuchen dann die rechte Gabel aufzunehmen



- ◆ System ist verklemmt
 - Philosophen warten alle auf ihre Nachbarn

7.5 Philosophenproblem (4)

■ Lösung 1: gleichzeitiges Aufnehmen der Gabeln

- ◆ Implementierung mit binären oder zählenden Semaphoren ist nicht trivial
- ◆ Zusatzvariablen erforderlich
- ◆ unübersichtliche Lösung

★ Einsatz von speziellen Semaphoren: PV-multiple-Semaphore

- ◆ gleichzeitiges und atomares Belegen mehrerer Semaphoren
- ◆ Abstrakte Beschreibung:

Operation	Bedingung	Anweisung
$P(\{S_i\})$	$\forall i, S_i > 0$	$\forall i, S_i = S_i - 1$
$V(\{S_i\})$	TRUE	$\forall i, S_i = S_i + 1$

7.5 Philosophenproblem (5)

- ◆ Implementierung mit PV-multiple-Semaphoren

```
PV_mult_semaphore forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i , $i \in [0,4]$

```
while( 1 ) {  
    ... /* think */  
  
    Pm( 2, &forks[i], &forks[(i+1)%5] );  
  
    ... /* eat */  
  
    Vm( 2, &forks[i], &forks[(i+1)%5] );  
}
```

7.5 Philosophenproblem (6)

- Lösung 2: einer der Philosophen muss erst die andere Gabel aufnehmen

```
semaphore forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i , $i \in [0,3]$

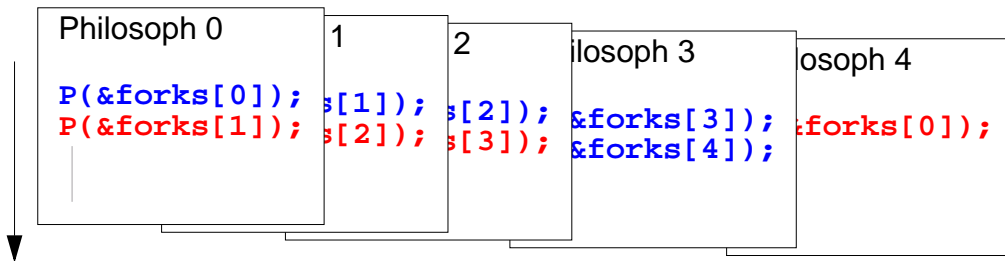
```
while( 1 ) {  
    ... /* think */  
  
    P( &forks[i] );  
    P( &forks[(i+1)%5] );  
  
    ... /* eat */  
  
    V( &forks[i] );  
    V( &forks[(i+1)%5] );  
}
```

Philosoph 4

```
while( 1 ) {  
    ... /* think */  
  
    P( &forks[0] );  
    P( &forks[4] );  
  
    ... /* eat */  
  
    V( &forks[0] );  
    V( &forks[4] );  
}
```

7.5 Philosophenproblem (7)

- ◆ Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- ◆ System verklemmt sich nicht

7.6 Schlafende Friseure

- Friseurladen mit N freien Wartestühlen
 - ◆ Friseure schlafen solange kein Kunde da ist
 - ◆ eintretende Kunden warten bis ein Friseur frei ist; gegebenenfalls wird einer der Friseure von einem Kunden aufgeweckt
 - ◆ sind keine Wartestühle mehr frei, verlassen die Kunden den Laden
- Problem:
 - ◆ Mehrere Bearbeitungsstationen sollen exklusive Bearbeitungen durchführen
- Implementierung mit zählenden Semaphoren
 - ◆ Semaphore zum Schutz der Variablen zum Zählen der Kunden: **mutex**
 - ◆ Semaphore zum Zählen der Friseure: **barbers**
 - ◆ Semaphore zum Zählen der Kunden: **customers**

7.6 Schlafende Friseure (2)

- Implementierung mit zählenden Semaphoren (PV System)

```
semaphor customers= 0, barbers= 0, mutex= 1;  
int waiting= 0;
```

Barber

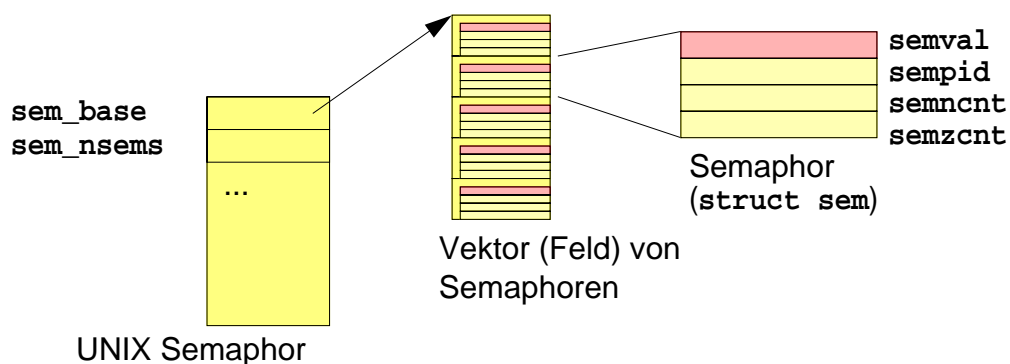
```
while( 1 ) {  
    P( &customers );  
    P( &mutex );  
    waiting--;  
    V( &barbers );  
    V( &mutex );  
  
    ... /* cut hair */  
}
```

Customer

```
P( &mutex );  
if( waiting < N ) {  
    waiting++;  
    V( &customers );  
    V( &mutex );  
    P( &barbers );  
  
    ... /* get hair cut */  
}  
else {  
    V( &mutex );  
}
```

8 UNIX-Semaphor

- Ein UNIX-Semaphor entspricht einem Vektor von Einzelsemaphoren (erweitertes Vektoradditionssystem)



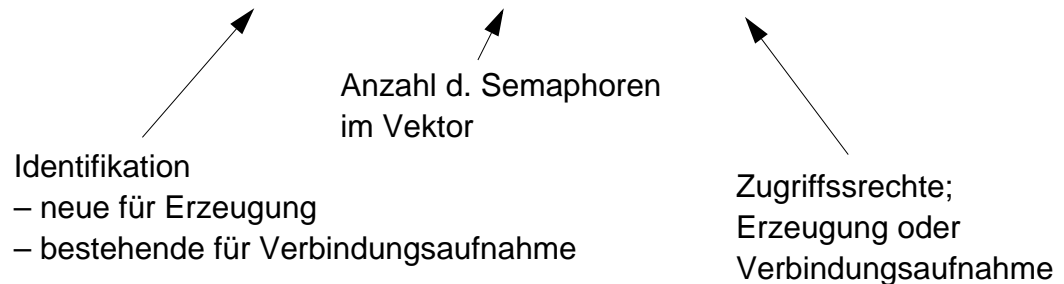
- ◆ Gleichzeitige und atomare Operationen auf mehreren Semaphoren im Vektor möglich

8.1 Erzeugen einer UNIX-Semaphore

■ UNIX-Semaphore haben systemweit eindeutige Identifikation (Key)

◆ Erzeugen und Aufnehmen der Verbindung zu einer Semaphore

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



◆ 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)

8.1 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

◆ Private Semaphore (ohne Key) können erzeugt werden

■ Semaphore sind persistent

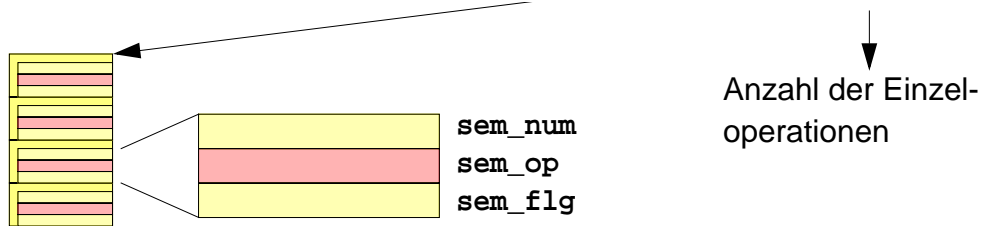
◆ Explizites Löschen notwendig

```
ipcrm -S <key>
```


8.2 Operationen auf UNIX-Semaphoren

■ Operationen auf mehreren der Semaphoren im Vektor

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



◆ Operationen

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

8.2 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
- ◆ Löschen des Semaphor

8.3 Beispiel: Philosophenproblem

- Ein UNIX-Semaphor 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;

...
```

8.3 Beispiel: Philosophenproblem (2)

- ◆ Erzeuge Semaphor

```
...

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 */
    }
}

...
```

8.3 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; */  
  
...
```

8.3 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;  
  
...
```

8.3 Beispiel: Philosophenproblem (5)

◆ Philosoph

```
...  
while( 1 ) {  
    ...      /* thinking */  
  
    if( semop( semid, pbuf, 2 ) < 0 ) { ... /* error */ }  
  
    ...      /* eating */  
  
    if( semop( semid, vbuf, 2 ) < 0 ) { ... /* error */ }  
}
```

9 Zusammenfassung

- Programmiermodell: Prozess
 - ◆ Zerlegung von Anwendungen in Prozesse oder Threads
 - ◆ Ausnutzen von Wartezeiten; Time sharing-Betrieb
 - ◆ Prozess hat verschiedene Zustände: laufend, bereit, blockiert 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

9 Zusammenfassung (2)

- Gegenseitiger Ausschluss mit Spinlocks
- Klassische Koordinierungsprobleme und deren Lösung mit Semaphoren
 - ◆ Gegenseitiger Ausschluss
 - ◆ Bounded buffers
 - ◆ Leser-Schreiber-Probleme
 - ◆ Philosophenproblem
 - ◆ Schlafende Friseure

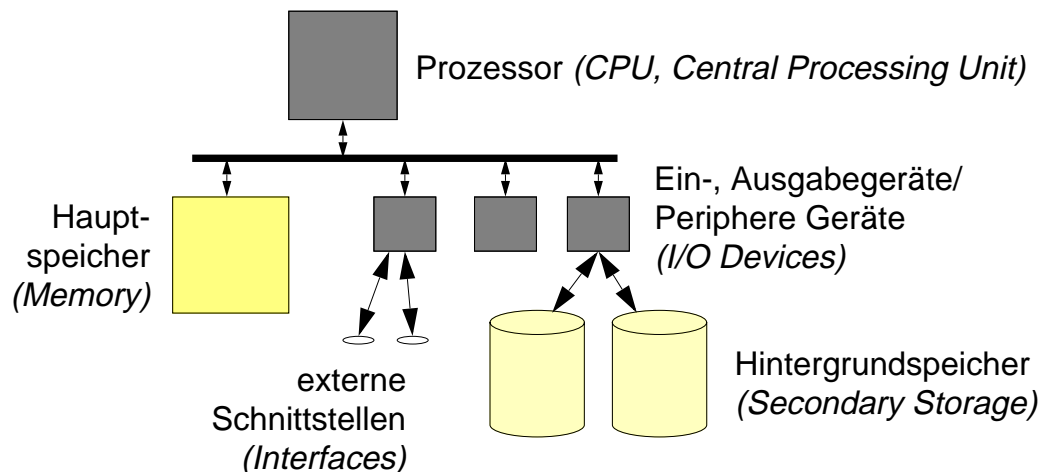
9 Zusammenfassung (3)

- UNIX Systemaufrufe
 - ◆ fork, exec, wait, nice
 - ◆ pipe, socket, bind, recvfrom, sendto, listen, accept
 - ◆ msgget, msgsnd, msgrcv
 - ◆ signal, kill, sigaction
 - ◆ semget, semop, semctl

E Speicherverwaltung

E Speicherverwaltung

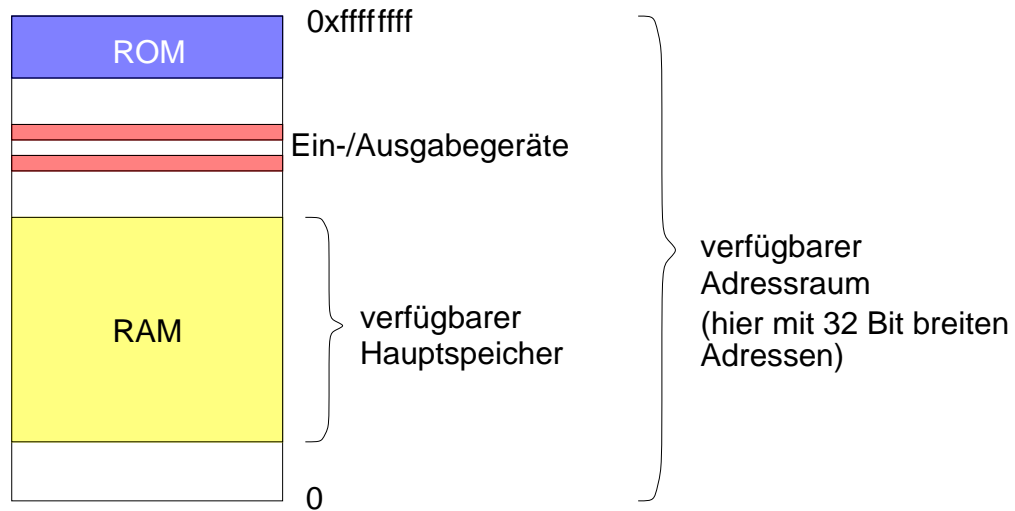
■ Betriebsmittel



1 Speichervergabe

1.1 Problemstellung

■ Verfügbarer Speicher



1.1 Problemstellung (2)

■ Belegung des verfügbaren Hauptspeichers durch

- ◆ Benutzerprogramme
 - Programmbefehle (Code, Binary)
 - Programmdateien
- ◆ Betriebssystem
 - Betriebssystemcode
 - Puffer
 - Systemvariablen

★ Zuteilung des Speichers nötig

1.2 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

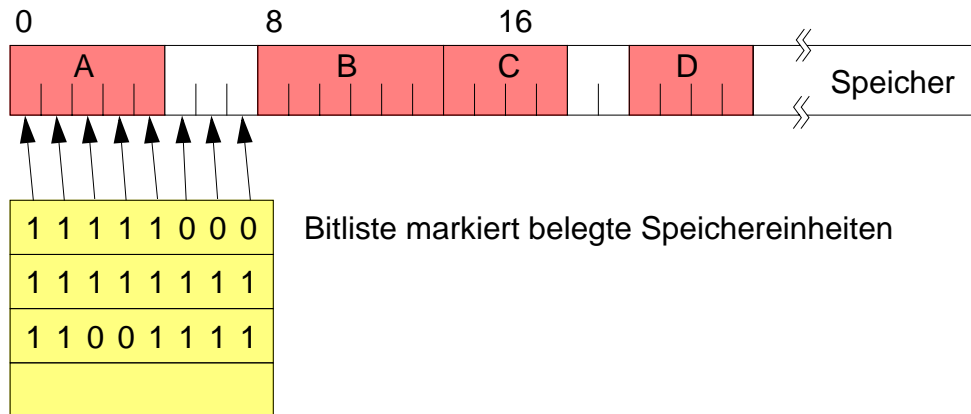
1.3 Dynamische Speicherzuteilung

- Segmente
 - ◆ zusammenhängender Speicherbereich
(Bereich mit aufeinanderfolgenden Adressen)
- Allokation (Anforderung) und Freigabe von Segmenten
- Ein Anwendungsprogramm besitzt üblicherweise folgende Segmente (siehe auch D.2.4):
 - ◆ Codesegment
 - ◆ Datensegment
 - ◆ Stacksegment (für Verwaltungsinformationen, z.B. bei Funktionsaufrufen)
- ▲ Suche nach geeigneten Speicherbereichen zur Zuteilung
- ★ Speicherzuteilungsstrategien nötig

1.4 Freispeicherverwaltung

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

- Bitlisten

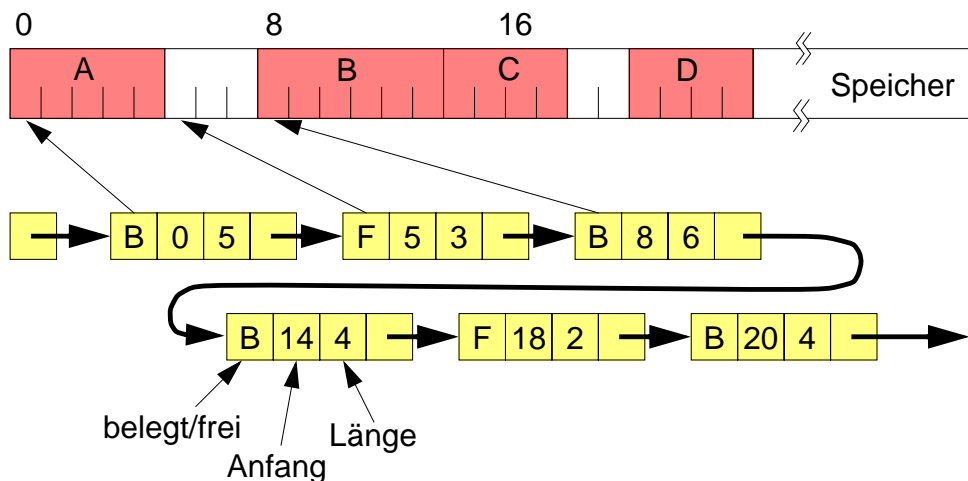


Bitliste markiert belegte Speichereinheiten

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

1.4 Freispeicherverwaltung (2)

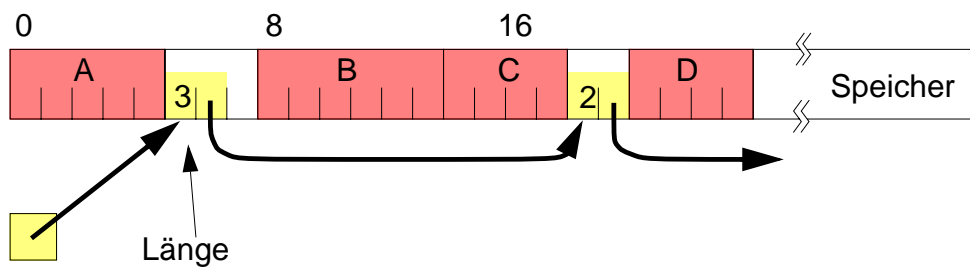
- Verkettete Liste



Repräsentation auch von freien Segmenten

1.4 Freispeicherverwaltung (3)

■ Verkettete Liste in dem freien Speicher

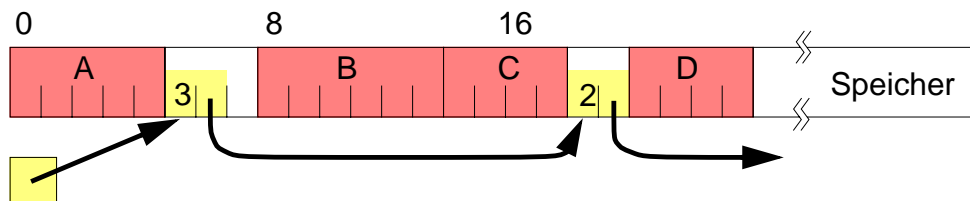


Mindestlückengröße muss garantiert werden

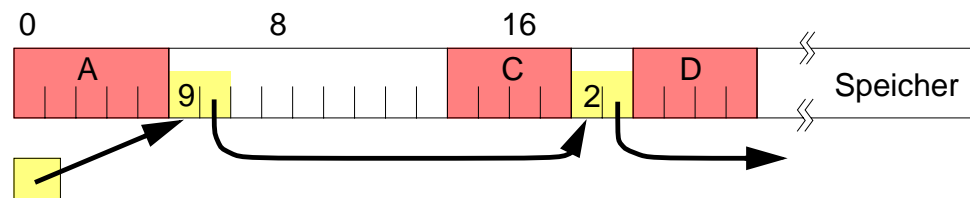
- Zur Effizienzsteigerung eventuell Rückwärtsverkettung nötig
- Repräsentation letztlich auch von der Vergabestrategie abhängig

1.5 Speicherfreigabe

■ Verschmelzung von Lücken



nach Freigabe von B:



1.6 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

1.7 Buddy Systeme

- Einteilung in dynamische Bereiche der Größe 2^n

	0	128	256	384	512	640	768	896	1024
	1024								
Anfrage 70	A	128	256	512					
Anfrage 35	A	B 64	256	512					
Anfrage 80	A	B 64	C 128	512					
Freigabe A	128	B 64	C 128	512					
Anfrage 60	128	B D	C 128	512					
Freigabe B	128	64 D	C 128	512					
Freigabe D	256	C 128	512						
Freigabe C	1024								

Effiziente Repräsentation der Lücken und effiziente Algorithmen

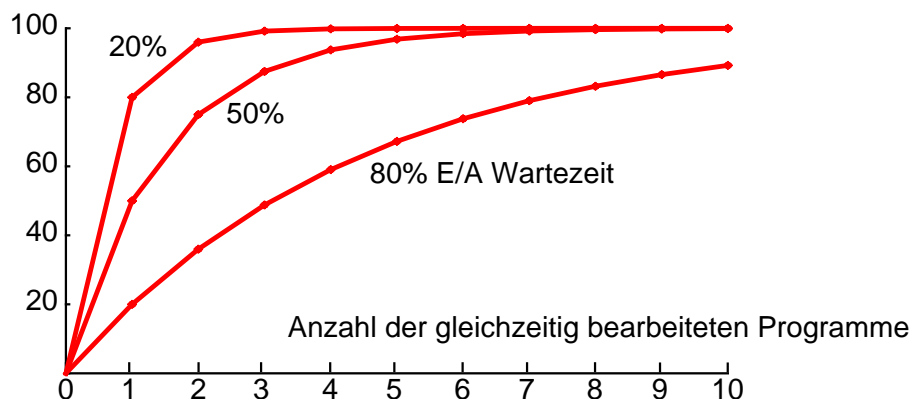
1.8 Einsatz der Verfahren

- 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*)

2 Mehrprogrammbetrieb

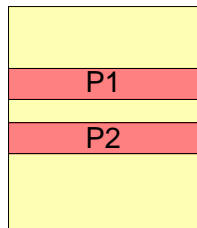
2.1 Problemstellung

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



2.1 Problemstellung (2)

- ▲ 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

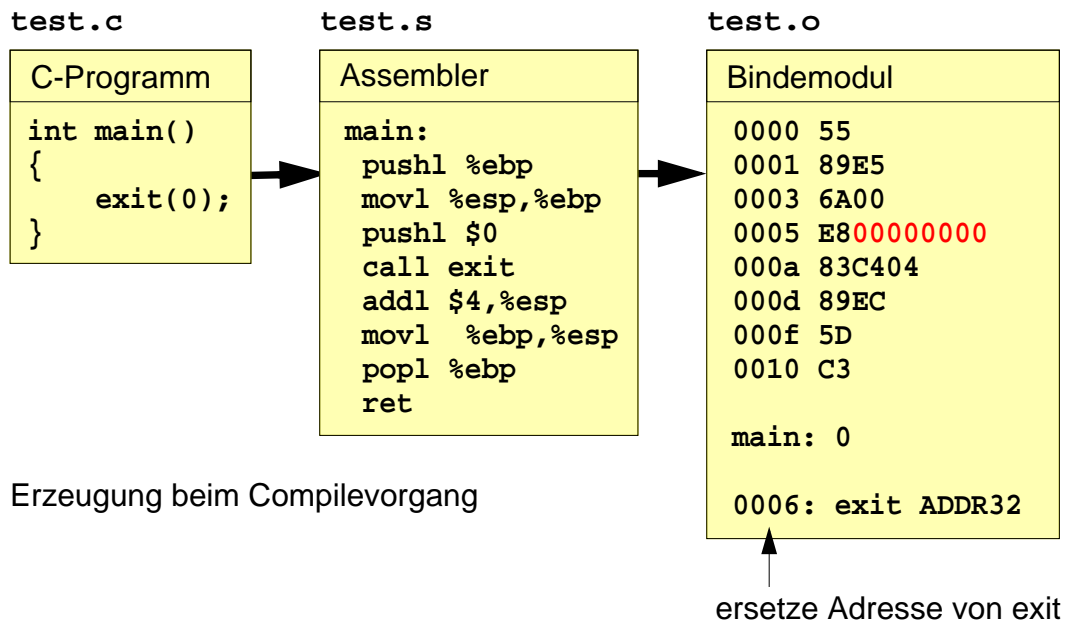
- ★ Relokation von Programmbefehlen (Binaries)
- ★ Ein- und Auslagern von Prozessen
- ★ Hardwareunterstützung

2.2 Relokation

- Festlegung absoluter Speicheradressen 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

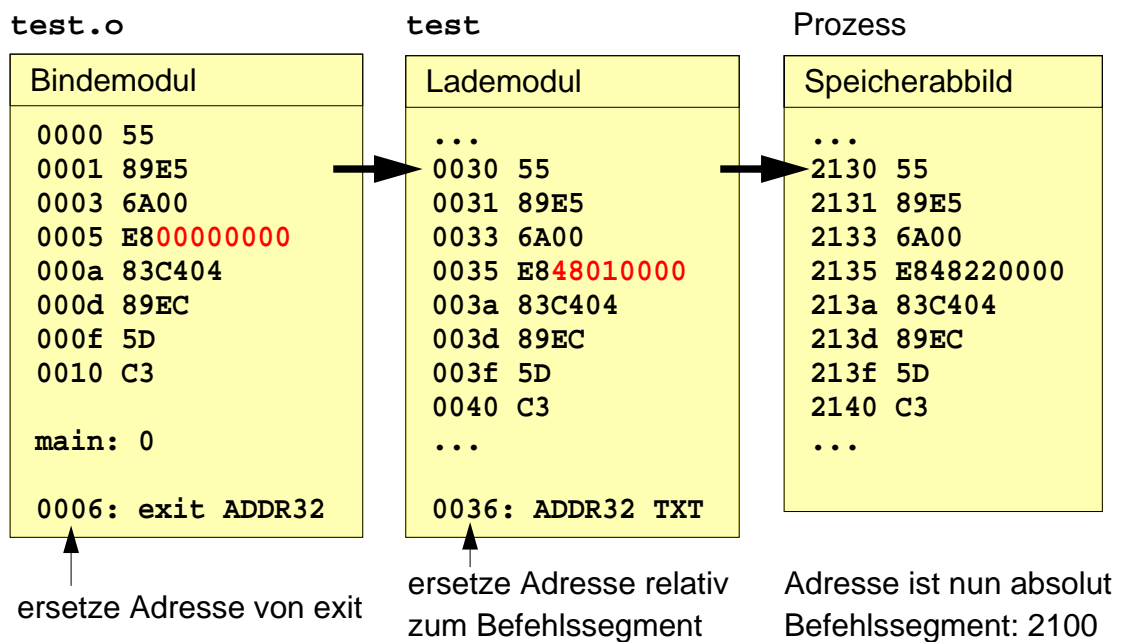
2.2 Relokation (2)

■ Compilevorgang (Erzeugung der Relokationsinformation)



2.2 Relokation (3)

■ Binde- und Ladevorgang

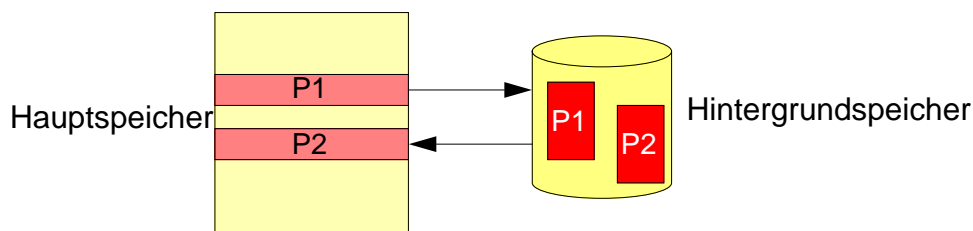


2.2 Relokation (4)

- Relokationsinformation im Bindemodul
 - ◆ erlaubt das Binden von Modulen in beliebige Programme
- Relokationsinformation 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

2.3 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

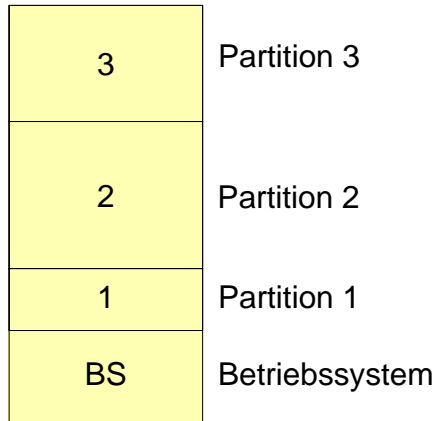


- ▲ Aus-, Einlagerzeit ist hoch
 - ◆ Latenzzeit der Festplatte
 - ◆ Übertragungszeit

2.3 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

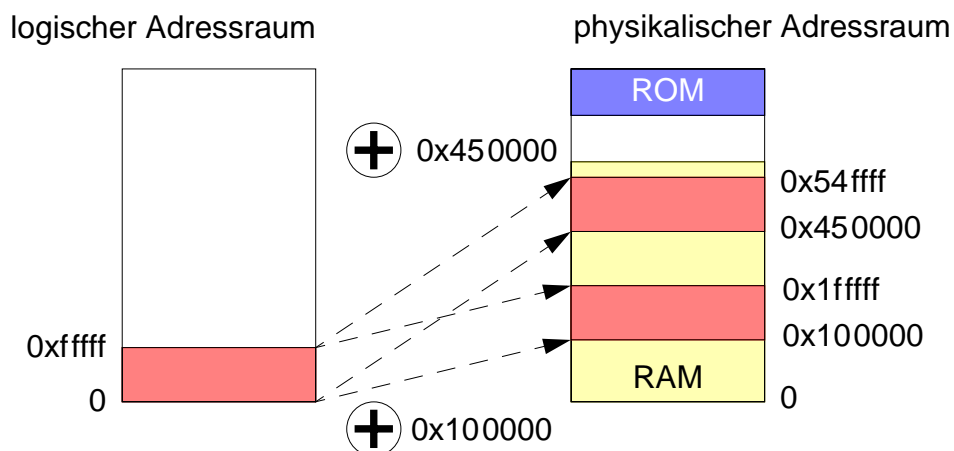


- ◆ In jeder Partition läuft nur ein Prozess
- ◆ Einlagerung erfolgt wieder in die gleiche Partition
- ◆ Speicher kann nicht optimal genutzt werden

2.4 Segmentierung

- Hardwareunterstützung: Umsetzung logischer in physikalische Adressen

- ◆ Prozesse erhalten einen logischen Adressraum

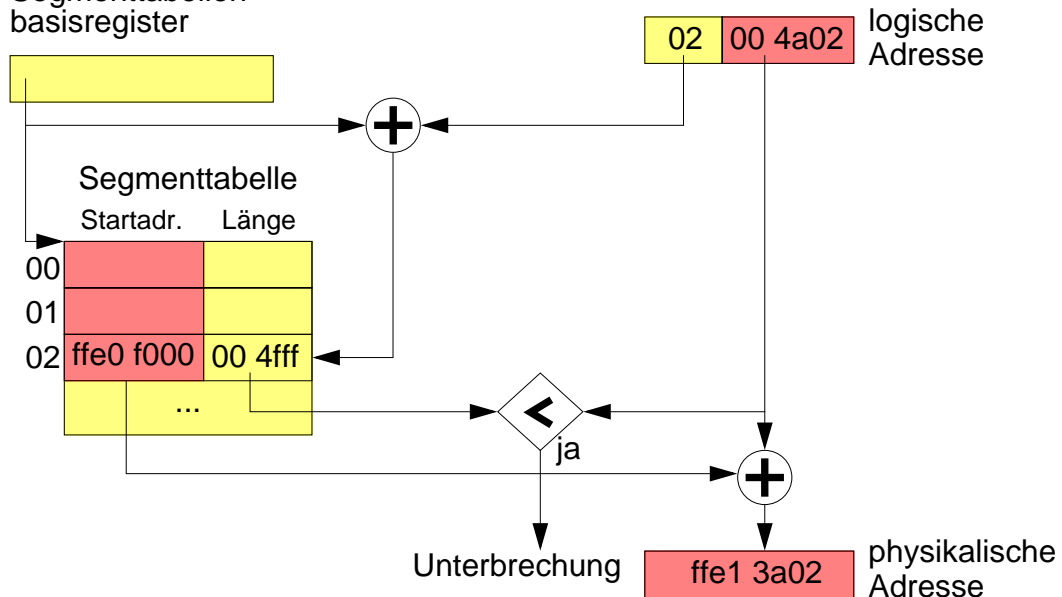


Das Segment des logischen Adressraums kann an jeder beliebige Stelle im physikalischen Adressraum liegen.

2.4 Segmentierung (2)

■ Realisierung mit Übersetzungstabelle

Segmenttabellen-
basisregister



2.4 Segmentierung (3)

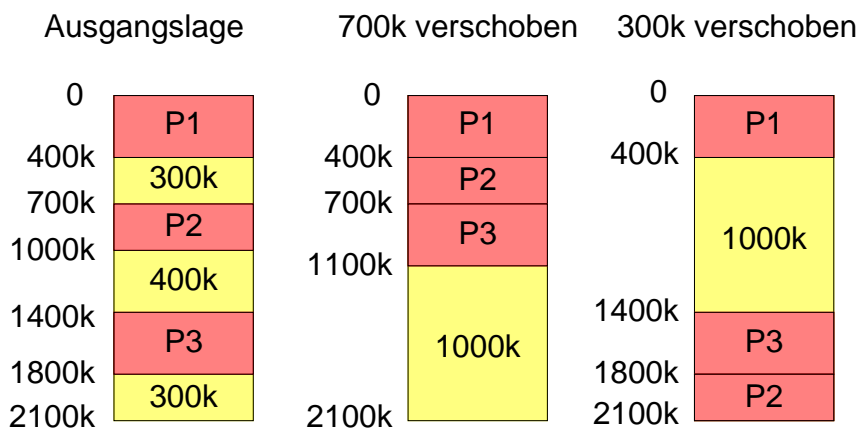
- Hardware wird MMU (*Memory Management Unit*) genannt
- Schutz vor Segmentübertretung
 - ◆ Unterbrechung zeigt Speicherverletzung an
 - ◆ Programme und Betriebssystem voreinander geschützt
- Prozessumschaltung durch Austausch der Segmentbasis
 - ◆ jeder Prozess hat eigene Übersetzungstabelle
- Ein-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

2.4 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

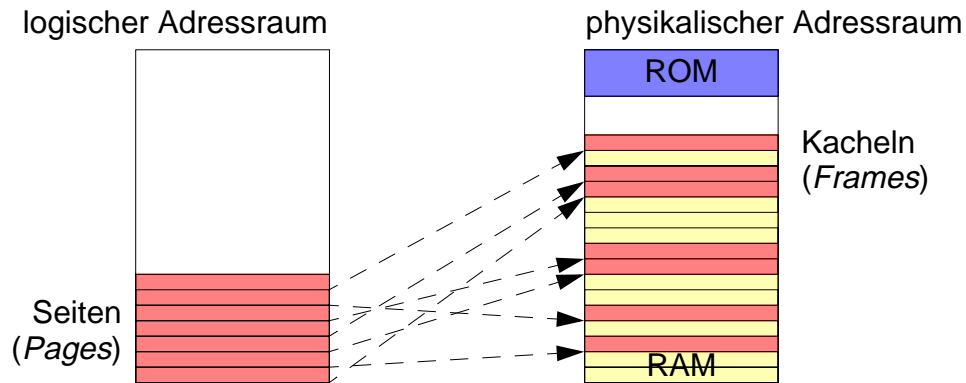
2.5 Kompaktifizieren

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



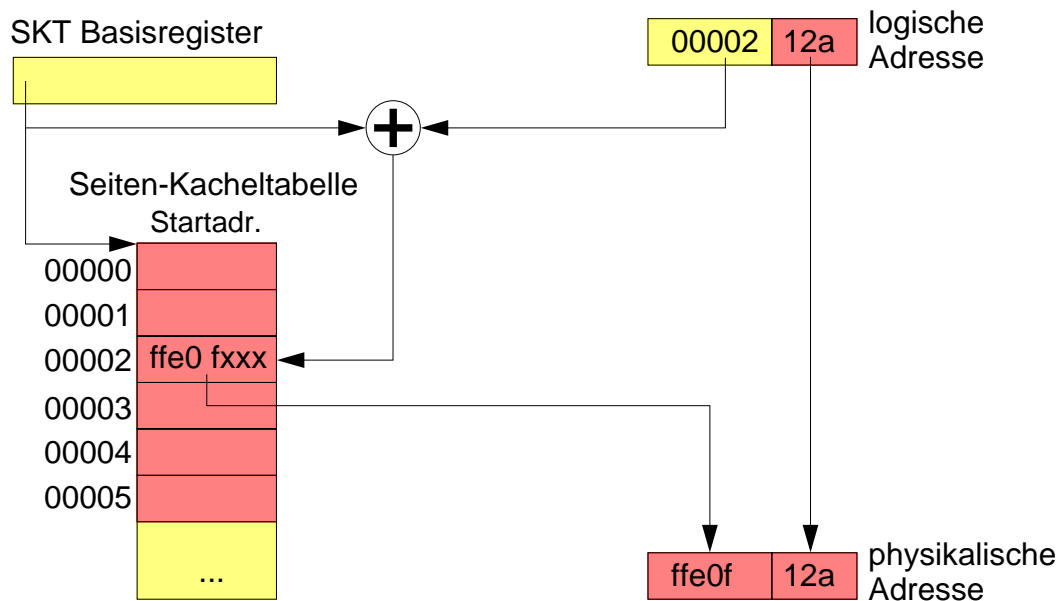
3 Seitenadressierung (*Paging*)

- 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 und Ein-, Auslagerungen



3.1 MMU mit Seiten-Kacheltabelle

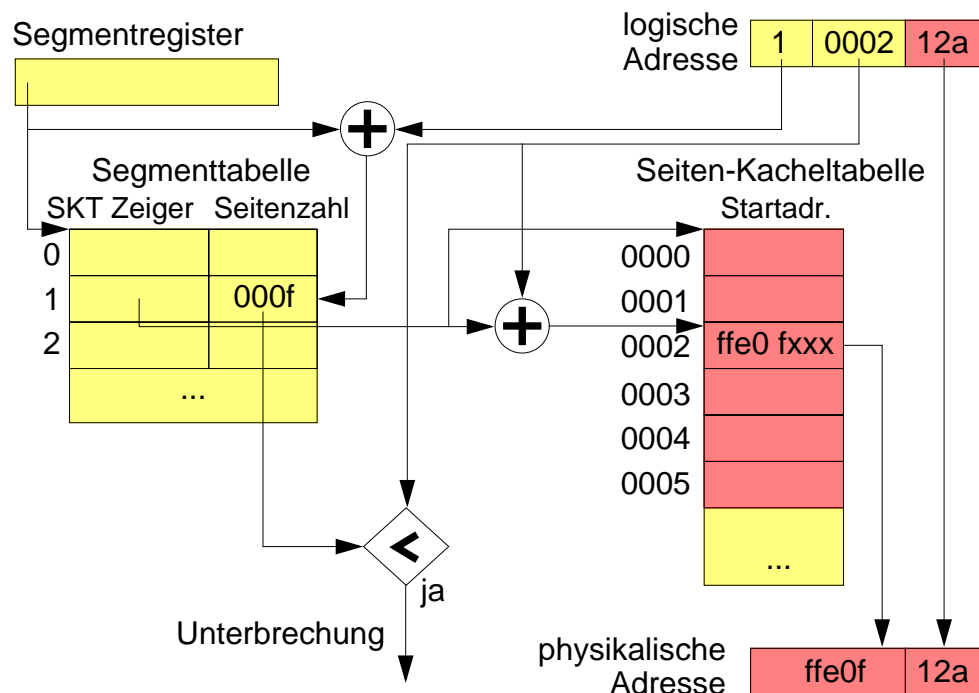
- Tabelle setzt Seiten in Kacheln um



3.1 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

3.2 Segmentierung und Seitenadressierung



3.2 Segmentierung und Seitenadressierung (2)

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

3.3 Ein- und Auslagerung von Seiten

- Es ist nicht nötig ein gesamtes Segment aus- bzw. einzulagern
 - ◆ Seiten können einzeln ein- und ausgelagert werden
- Hardware-Unterstützung

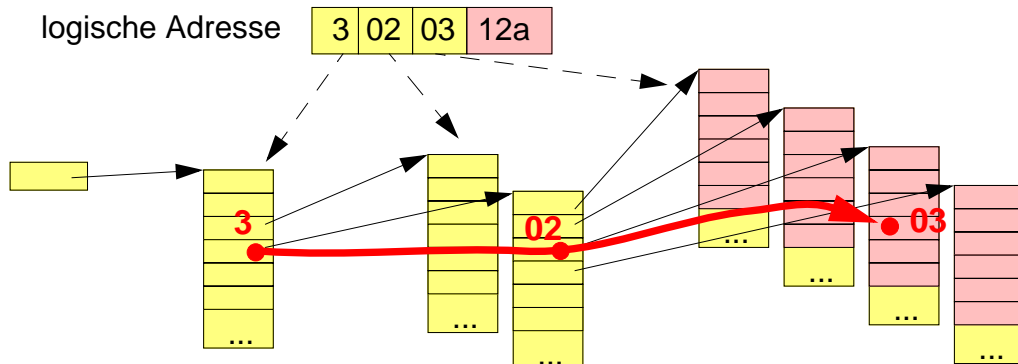
Seiten-Kachelntabelle
Startadr. Präsenzbit

0000		
0001		
0002	ffe0 fxxx	X
	...	

- ◆ 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).

3.4 Mehrstufige Seitenadressierung

■ Beispiel: zweifach indirekte Seitenadressierung



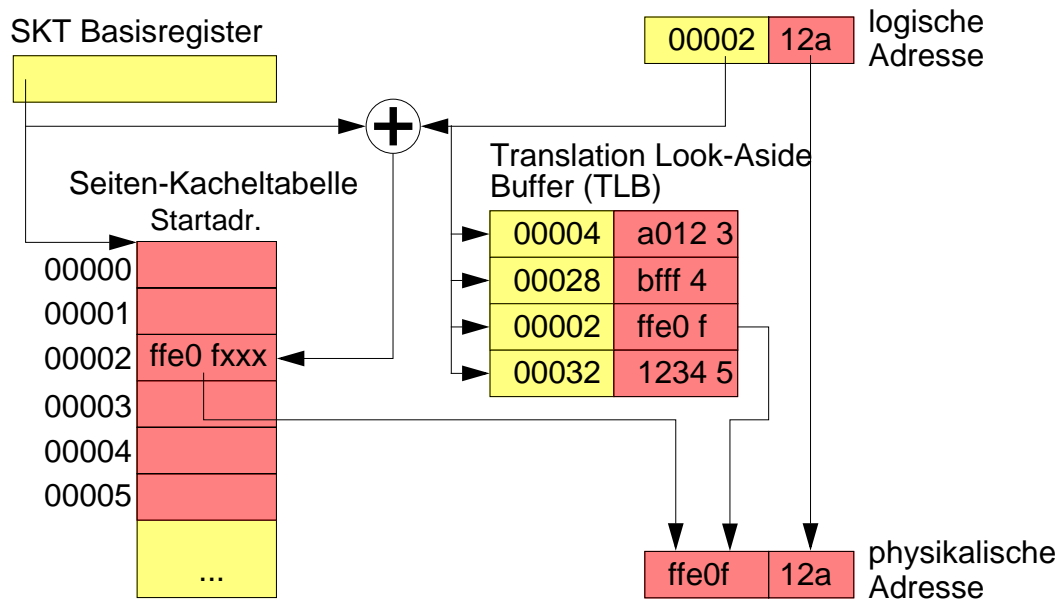
■ Präsenzbit auch für jeden Eintrag in den höheren Stufen

◆ Tabellen werden aus- und einlagerbar

▲ Noch mehr implizite Speicherzugriffe

3.5 Translation Look-Aside Buffer

■ Schneller Registersatz wird konsultiert bevor auf die SKT zugegriffen wird:

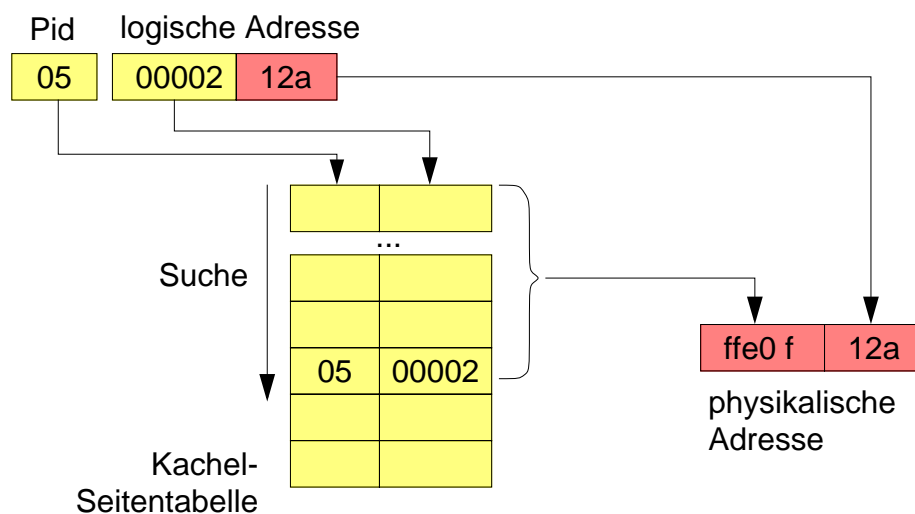


3.5 Translation Look-Aside Buffer (2)

- 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

3.6 Invertierte Seiten-Kacheltabelle

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



3.6 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

3.7 Systemaufruf

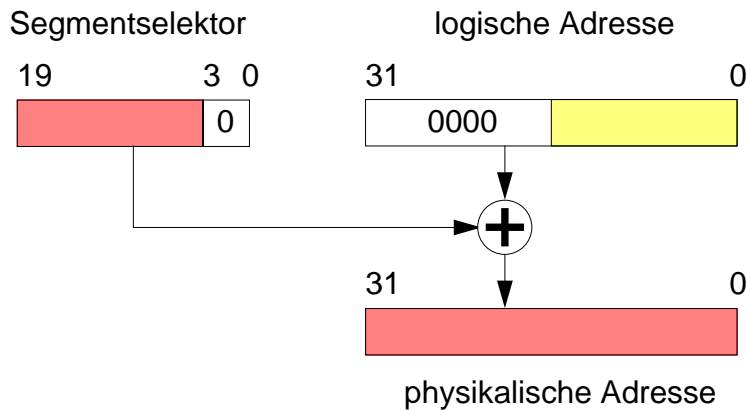
- Ermitteln der Seitengröße des Betriebssystems
`int getpagesize(void);`

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

4.1 Real Mode Adressierung

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



4.2 Protected Mode Adressierung

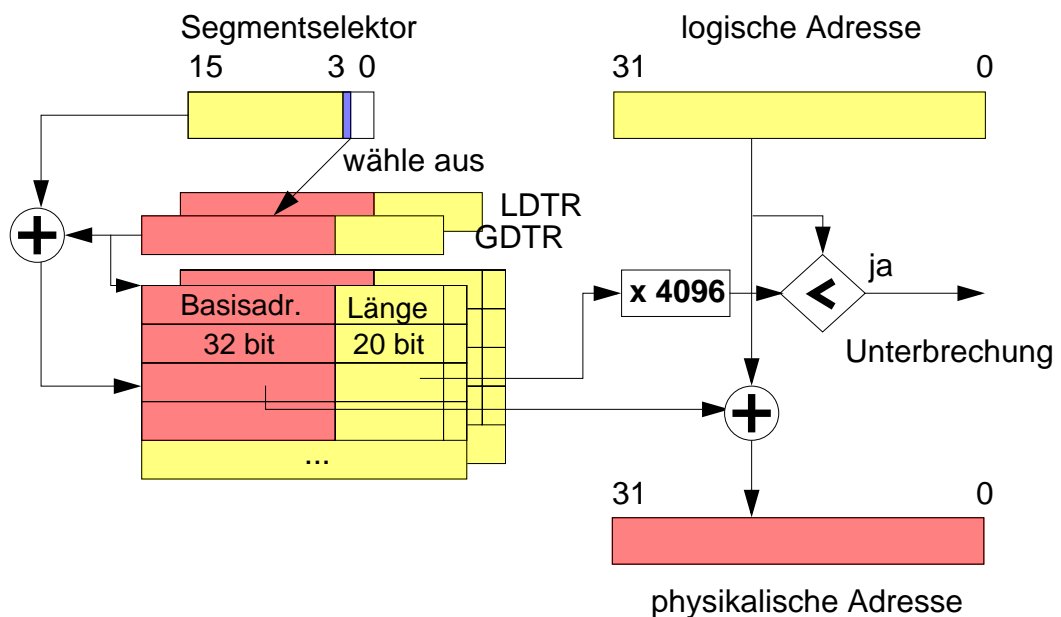
- Vier Betriebsmodi (Stufen von Privilegien)
 - ◆ Stufe 0: höchste Privilegien (privilegierte Befehle, etc.): BS Kern
 - ◆ Stufe 1: BS Treiber
 - ◆ Stufe 2: BS Erweiterungen
 - ◆ Stufe 3: Benutzerprogramme
- ◆ Speicherverwaltung kann nur in Stufe 0 konfiguriert werden
- Segmentselektoren enthalten Privilegiierungsstufe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

4.2 Protected Mode Adressierung (2)

- Deskriptortabelle
 - ◆ enthält bis zu 8192 Segmentdeskriptoren
 - ◆ Inhalt des Segmentdeskriptors:
 - 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)

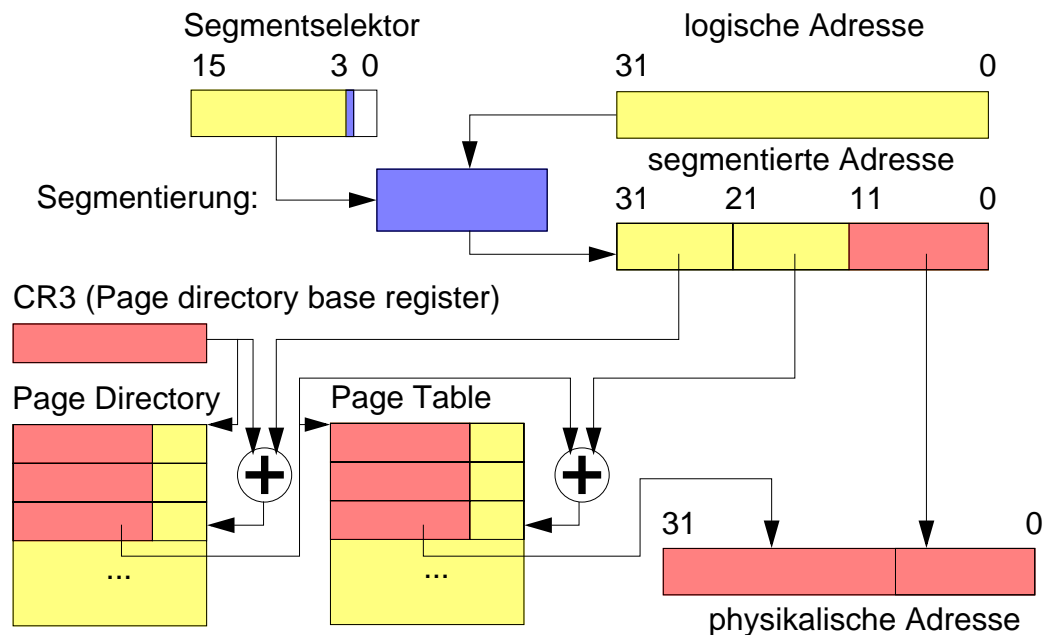
4.3 Adressberechnung bei Segmentierung

- Verwendung der Protected mode Adressierung



4.4 Adressberechnung bei Paging

- Seitenadressierung wird der Segmentierung nachgeschaltet

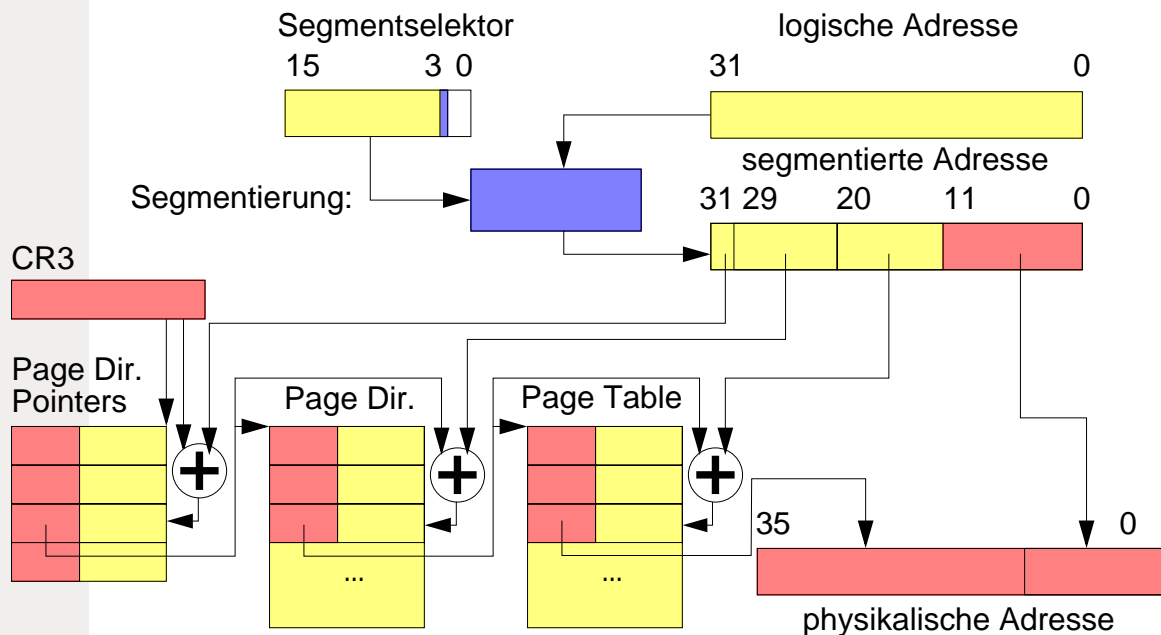


4.4 Adressberechnung bei Paging (2)

- 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

4.5 Physical-Address-Extension (PAE)

■ Nachgeschaltete dreistufige Seitenadressierung



4.5 Physical-Address-Extension (PAE) (2)

■ Ab Pentium Pro

- ◆ vierelementige Tabelle von Page-Directory-Pointers
- ◆ 24 statt 20 Bit breite physikalische Adressumsetzung für den Seitenanfang
- ◆ 64 Bit statt 32 Bit breite Tabelleneinträge
- ◆ spezieller Chipsatz erforderlich

■ Adressierbarer Hauptspeicher von 64 GByte

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

5 Gemeinsamer Speicher (2)

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

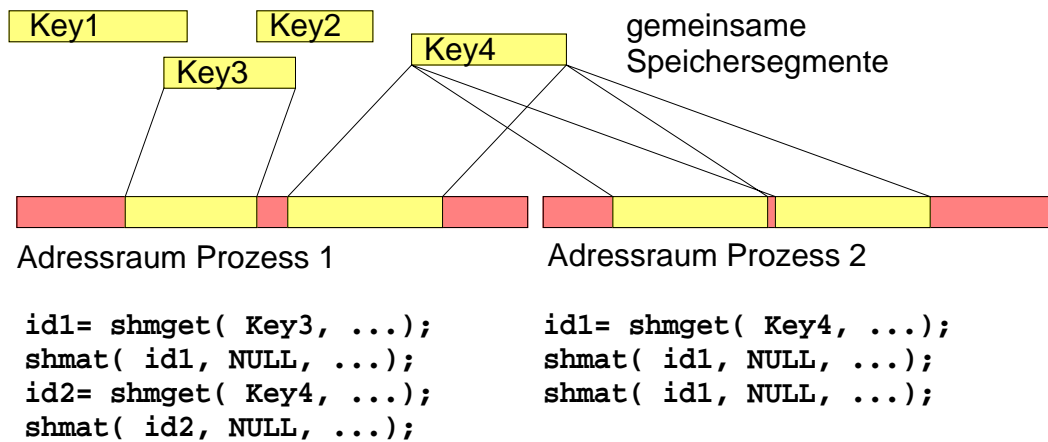
```
int shmget( key_t key, int size, int shmflg );
```
 - ◆ Einblenden und Ausblenden des Segments in den Speicher

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

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

5 Gemeinsamer Speicher (3)

■ Prinzip der `shm*` Operationen



5 Gemeinsamer Speicher (4)

■ Verwendung des Keys

- ◆ Alle Prozesse, die auf ein Speichersegment zugreifen wollen, müssen den Key kennen
- ◆ Keys sind eindeutig innerhalb eines (Betriebs-)Systems
- ◆ Ist ein Key bereits vergeben, kann kein Segment mit gleichem Key erzeugt werden
- ◆ Ist ein Key bekannt, kann auf das Segment zugegriffen werden
 - gesetzte Zugriffsberechtigungen werden allerdings beachtet
- ◆ Es können Segmente ohne Key erzeugt werden (private Segmente)

■ Keys werden benutzt für:

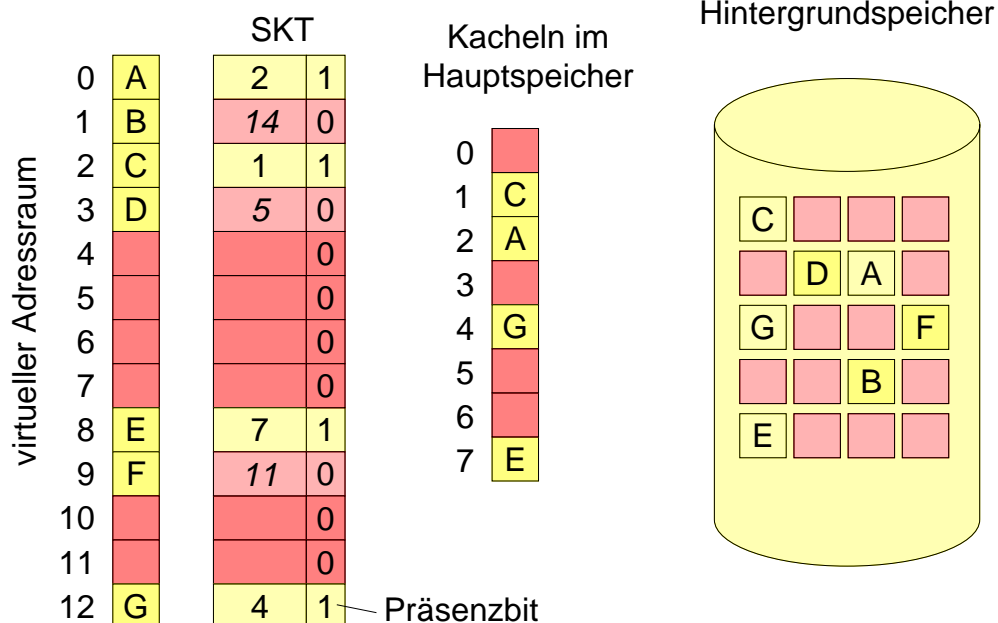
- ◆ Queues
- ◆ Semaphore
- ◆ Shared memory segments

6 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
 - ◆ Einblenden benötigter Speicherbereiche
 - ◆ Abfangen von Zugriffen auf nicht-eingeblendete Bereiche
 - ◆ Bereitstellen der benötigten Bereiche auf Anforderung
 - ◆ Auslagern nicht-benötigter Bereiche

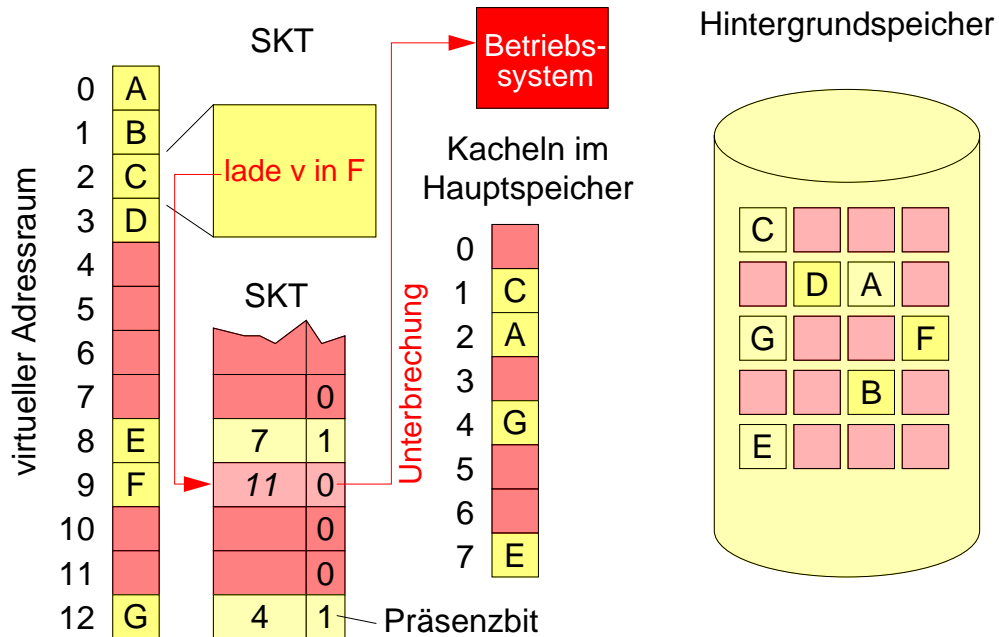
6.1 Demand Paging

- Bereitstellen von Seiten auf Anforderung



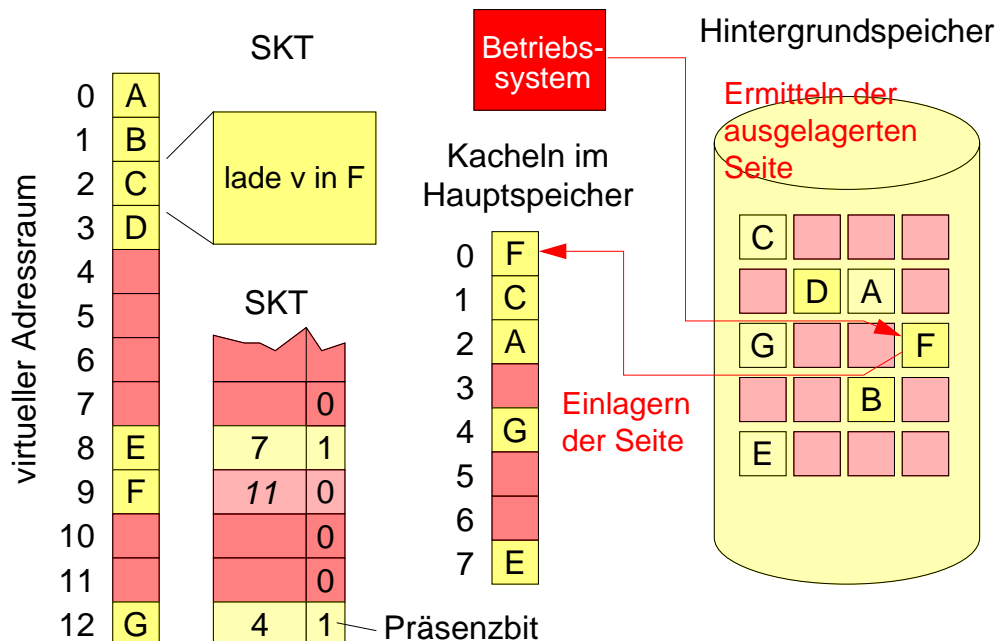
6.1 Demand Paging (2)

■ Reaktion auf Seitenfehler



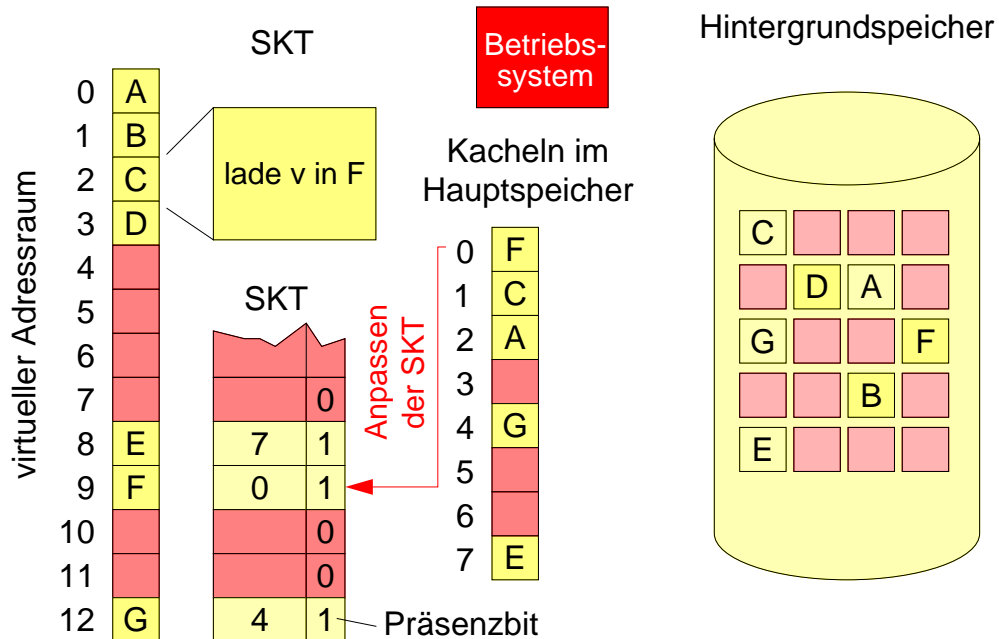
6.1 Demand Paging (3)

■ Reaktion auf Seitenfehler



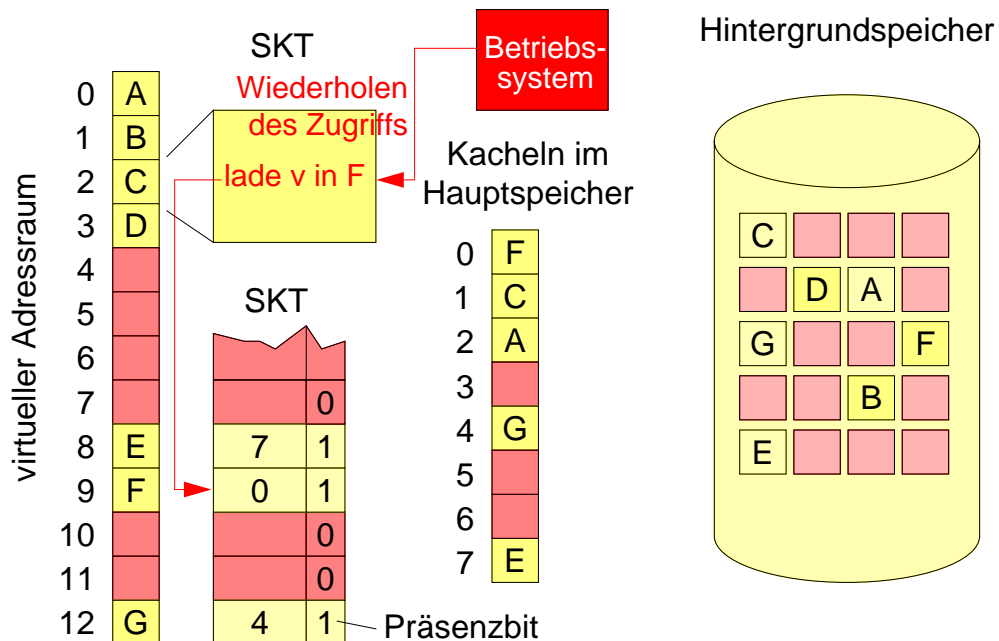
6.1 Demand Paging (4)

■ Reaktion auf Seitenfehler



6.1 Demand Paging (5)

■ Reaktion auf Seitenfehler



6.1 Demand Paging (6)

- ▲ Performanz von Demand paging
 - ◆ Keine Seitenfehler
 - effektive Zugriffszeit zw. 10 und 200 Nanosekunden
 - ◆ Mit Seitenfehler
 - p sei Wahrscheinlichkeit für Seitenfehler; p nahe Null
 - Annahme: Zeit zum Einlagern einer Seite vom Hintergrundspeicher gleich 25 Millisekunden (8 ms Latenz, 15 ms Positionierzeit, 1 ms Übertragungszeit)
 - Annahme: normale Zugriffszeit 100 ns
 - Effektive Zugriffszeit:
$$(1 - p) \times 100 + p \times 25000000 = 100 + 24999900 \times p$$
- ▲ Seitenfehler müssen so niedrig wie möglich gehalten werden
- Abwandlung: *Demand zero* für nicht initialisierte Daten

6.2 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?

7 Ersetzungsstrategien

- Betrachtung von Ersetzungsstrategien und deren Wirkung auf Referenzfolgen
- Referenzfolge
 - ◆ Folge von Seitennummern, die das Speicherzugriffsverhalten eines Prozesses abbildet
 - ◆ Ermittlung von Referenzfolgen z.B. durch Aufzeichnung der zugegriffenen Adressen
 - Reduktion der aufgezeichneten Sequenz auf Seitennummern
 - Zusammenfassung von unmittelbar hintereinanderstehenden Zugriffen auf die gleiche Seite
 - ◆ Beispiel für eine Referenzfolge: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

7.1 First-In, First-Out

- Älteste Seite wird ersetzt
- Notwendige Zustände:
 - ◆ Alter bzw. Einlagerungszeitpunkt für jede Kachel
- Ablauf der Ersetzungen (9 Einlagerungen)

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	4	4	4	5	5	5	5	5	5
	Kachel 2		2	2	2	1	1	1	1	1	3	3	3
	Kachel 3			3	3	3	2	2	2	2	2	4	4
Kontrollzustände (Alter pro Kachel)	Kachel 1	0	1	2	0	1	2	0	1	2	3	4	5
	Kachel 2	>	0	1	2	0	1	2	3	4	0	1	2
	Kachel 3	>	>	0	1	2	0	1	2	3	4	0	1

7.1 First-In, First-Out (2)

- Größerer Hauptspeicher mit 4 Kacheln (10 Einlagerungen)

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	5	5	5	5	4	4
	Kachel 2		2	2	2	2	2	2	1	1	1	1	5
	Kachel 3			3	3	3	3	3	3	2	2	2	2
	Kachel 4				4	4	4	4	4	4	3	3	3
Kontrollzustände (Alter pro Kachel)	Kachel 1	0	1	2	3	4	5	0	1	2	3	0	1
	Kachel 2	>	0	1	2	3	4	5	0	1	2	3	0
	Kachel 3	>	>	0	1	2	3	4	5	0	1	2	3
	Kachel 4	>	>	>	0	1	2	3	4	5	0	1	2

- FIFO Anomalie (Belady's Anomalie, 1969)

7.2 Optimale Ersetzungsstrategie

- Vorwärtsabstand
 - ◆ Zeitdauer bis zum nächsten Zugriff auf die entsprechende Seite
- Strategie B_0 (OPT oder MIN) ist optimal (bei fester Kachelmenge):
minimale Anzahl von Einlagerungen/Ersetzungen (hier 7)
 - ◆ „Ersetze immer die Seite mit dem größten Vorwärtsabstand!“

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	1	1	1	3	4	4
	Kachel 2		2	2	2	2	2	2	2	2	2	2	2
	Kachel 3			3	4	4	4	5	5	5	5	5	5
Kontrollzustände (Vorwärts- abstand)	Kachel 1	4	3	2	1	3	2	1	>	>	>	>	>
	Kachel 2	>	4	3	2	1	3	2	1	>	>	>	>
	Kachel 3	>	>	7	7	6	5	5	4	3	2	1	>

7.2 Optimale Ersetzungsstrategie (2)

- Vergrößerung des Hauptspeichers (4 Kacheln): 6 Einlagerungen

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	1	1	1	1	4	4
	Kachel 2		2	2	2	2	2	2	2	2	2	2	2
	Kachel 3			3	3	3	3	3	3	3	3	3	3
	Kachel 4				4	4	4	5	5	5	5	5	5
Kontrollzustände (Vorwärts- abstand)	Kachel 1	4	3	2	1	3	2	1	>	>	>	>	>
	Kachel 2	>	4	3	2	1	3	2	1	>	>	>	>
	Kachel 3	>	>	7	6	5	4	3	2	1	>	>	>
	Kachel 4	>	>	>	7	6	5	5	4	3	2	1	>

- ◆ keine Anomalie

7.2 Optimale Ersetzungsstrategie (3)

- Implementierung von B_0 nahezu unmöglich
 - ◆ Referenzfolge müsste vorher bekannt sein
 - ◆ B_0 meist nur zum Vergleich von Strategien brauchbar
- Suche nach Strategien, die möglichst nahe an B_0 kommen
 - ◆ z.B. *Least recently used* (LRU)

7.3 Least Recently Used (LRU)

- Rückwärtsabstand
 - ◆ Zeitdauer, seit dem letzten Zugriff auf die Seite
- LRU Strategie (10 Einlagerungen)
 - ◆ „Ersetze die Seite mit dem größten Rückwärtsabstand !“

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	4	4	4	5	5	5	3	3	3
	Kachel 2		2	2	2	1	1	1	1	1	1	4	4
	Kachel 3			3	3	3	2	2	2	2	2	2	5
Kontrollzustände (Rückwärts- abstand)	Kachel 1	0	1	2	0	1	2	0	1	2	0	1	2
	Kachel 2	>	0	1	2	0	1	2	0	1	2	0	1
	Kachel 3	>	>	0	1	2	0	1	2	0	1	2	0

7.3 Least Recently Used (2)

- Vergrößerung des Hauptspeichers (4 Kacheln): 8 Einlagerungen

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	1	1	1	1	1	5
	Kachel 2		2	2	2	2	2	2	2	2	2	2	2
	Kachel 3			3	3	3	3	5	5	5	5	4	4
	Kachel 4				4	4	4	4	4	4	3	3	3
Kontrollzustände (Rückwärts- abstand)	Kachel 1	0	1	2	3	0	1	2	0	1	2	3	0
	Kachel 2	>	0	1	2	3	0	1	2	0	1	2	3
	Kachel 3	>	>	0	1	2	3	0	1	2	3	0	1
	Kachel 4	>	>	>	0	1	2	3	4	5	0	1	2

7.3 Least Recently Used (3)

■ Keine Anomalie

- ◆ Allgemein gilt: Es gibt eine Klasse von Algorithmen (Stack-Algorithmen), bei denen keine Anomalie auftritt:
 - Bei Stack-Algorithmen ist bei n Kacheln zu jedem Zeitpunkt eine Untermenge der Seiten eingelagert, die bei $n+1$ Kacheln zum gleichen Zeitpunkt eingelagert wären!
 - LRU: Es sind immer die letzten n benutzten Seiten eingelagert
 - B_0 : Es sind die n bereits benutzten Seiten eingelagert, die als nächstes zugegriffen werden

▲ Problem

- ◆ Implementierung von LRU nicht ohne Hardwareunterstützung möglich
- ◆ Es muss jeder Speicherzugriff berücksichtigt werden

7.3 Least Recently Used (4)

■ Hardwareunterstützung durch Zähler

- ◆ CPU besitzt einen Zähler, der bei jedem Speicherzugriff erhöht wird (inkrementiert wird)
- ◆ bei jedem Zugriff wird der aktuelle Zählerwert in den jeweiligen Seitendeskriptor geschrieben
- ◆ Auswahl der Seite mit dem kleinsten Zählerstand

▲ Aufwendige Implementierung

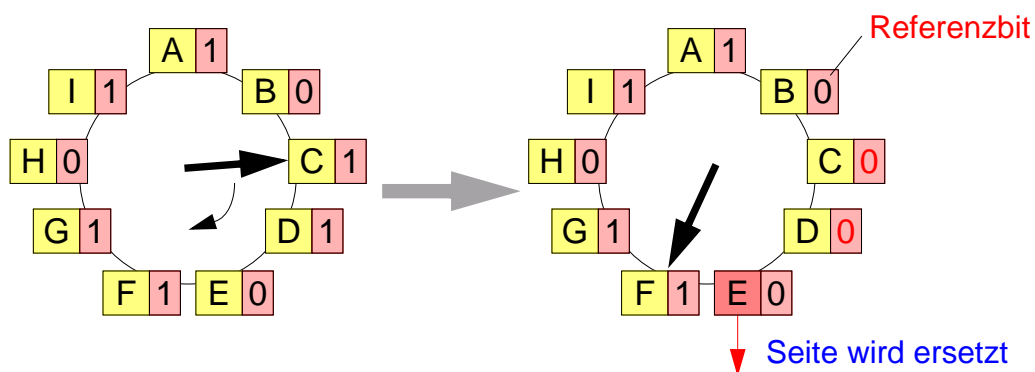
- ◆ viele zusätzliche Speicherzugriffe

7.4 Second Chance (Clock)

- Einsatz von Referenzbits
 - ◆ Referenzbit im Seitendeskriptor wird automatisch durch Hardware gesetzt, wenn die Seite zugegriffen wird
 - einfacher zu implementieren
 - weniger zusätzliche Speicherzugriffe
 - moderne Prozessoren bzw. MMUs unterstützen Referenzbits (z.B. Pentium: *Access bit*)
- Ziel: Annäherung von LRU
 - ◆ das Referenzbit wird zunächst auf 0 gesetzt
 - ◆ wird eine Opferseite gesucht, so werden die Kacheln reihum inspiziert
 - ◆ ist das Referenzbit 1, so wird es auf 0 gesetzt (zweite Chance)
 - ◆ ist das Referenzbit 0, so wird die Seite ersetzt

7.4 Second Chance (2)

- Implementierung mit umlaufendem Zeiger (*Clock*)



- ◆ an der Zeigerposition wird Referenzbit getestet
 - falls Referenzbit eins, wird Bit gelöscht
 - falls Referenzbit gleich Null, wurde ersetzbare Seite gefunden
 - Zeiger wird weitergestellt; falls keine Seite gefunden: Wiederholung
- ◆ falls alle Referenzbits auf 1 stehen, wird Second chance zu FIFO

7.4 Second Chance (3)

■ Ablauf bei drei Kacheln (9 Einlagerungen)

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	4	4	4	5	5	5	5	5	5
	Kachel 2		2	2	2	1	1	1	1	1	3	3	3
	Kachel 3			3	3	3	2	2	2	2	2	4	4
Kontroll- zustände (Referenzbits)	Kachel 1	1	1	1	1	1	1	1	1	1	0	0	1
	Kachel 2	0	1	1	0	1	1	0	1	1	1	1	1
	Kachel 3	0	0	1	0	0	1	0	0	1	0	1	1
	Umlaufzeiger	2	3	1	2	3	1	2	2	2	3	1	1

7.4 Second Chance (4)

■ Vergrößerung des Hauptspeichers (4 Kacheln): 10 Einlagerungen

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	5	5	5	5	4	4
	Kachel 2		2	2	2	2	2	2	1	1	1	1	5
	Kachel 3			3	3	3	3	3	3	2	2	2	2
	Kachel 4				4	4	4	4	4	4	3	3	3
Kontroll- zustände (Referenzbits)	Kachel 1	1	1	1	1	1	1	1	1	1	1	1	1
	Kachel 2	0	1	1	1	1	1	0	1	1	1	0	1
	Kachel 3	0	0	1	1	1	1	0	0	1	1	0	0
	Kachel 4	0	0	0	1	1	1	0	0	0	1	0	0
	Umlaufzeiger	2	3	4	1	1	1	2	3	4	1	2	3

7.4 Second Chance (5)

- Second chance zeigt FIFO Anomalie
 - ◆ Wenn alle Referenzbits gleich 1, wird nach FIFO entschieden
- Erweiterung
 - ◆ Modifikationsbit kann zusätzlich berücksichtigt werden (*Dirty bit*)
 - ◆ drei Klassen: (0,0), (1,0) und (1,1) mit (Referenzbit, Modifikationsbit)
 - ◆ Suche nach der niedrigsten Klasse (Einsatz im MacOS)

7.5 Freiseitenpuffer

- Statt eine Seite zu ersetzen wird permanent eine Menge freier Seiten gehalten
 - ◆ Auslagerung geschieht im „voraus“
 - ◆ Effizienter: Ersetzungszeit besteht im Wesentlichen nur aus Einlagerungszeit
- Behalten der Seitenzuordnung auch nach der Auslagerung
 - ◆ Wird die Seite doch noch benutzt bevor sie durch eine andere ersetzt wird, kann sie mit hoher Effizienz wiederverwendet werden.
 - ◆ Seite wird aus Freiseitenpuffer ausgetragen und wieder dem entsprechenden Prozess zugeordnet.

7.6 Seitenanforderung

- ▲ Problem: Zuordnung der Kacheln zu mehreren Prozessen
- Begrenzungen
 - ◆ Maximale Seitenmenge: begrenzt durch Anzahl der Kacheln
 - ◆ Minimale Seitenmenge: abhängig von der Prozessorarchitektur
 - Mindestens die Anzahl von Seiten nötig, die theoretisch bei einem Maschinenbefehl benötigt werden
(z.B. zwei Seiten für den Befehl, vier Seiten für die adressierten Daten)
- Gleiche Zuordnung
 - ◆ Anzahl der Prozesse bestimmt die Kachelmenge, die ein Prozess bekommt
- Größenabhängige Zuordnung
 - ◆ Größe des Programms fließt in die zugeteilte Kachelmenge ein

7.6 Seitenanforderung

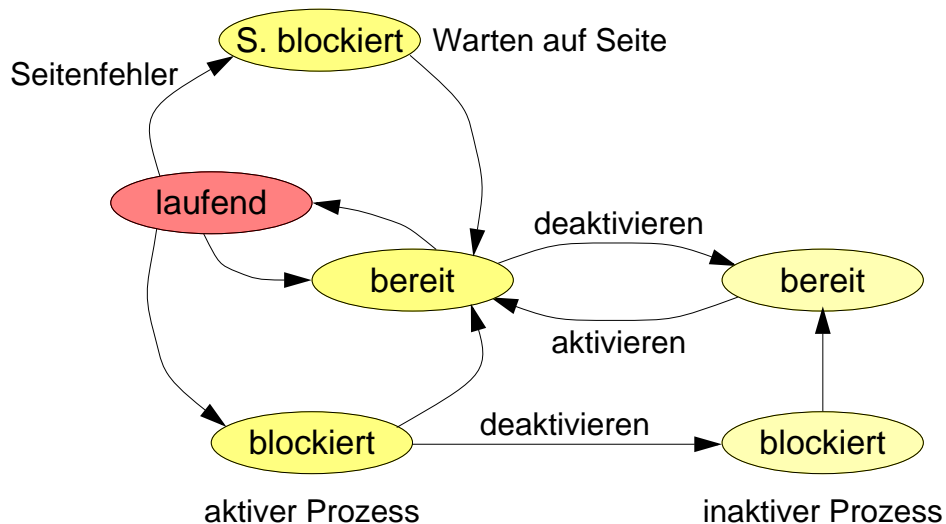
- Globale und lokale Anforderung von Seiten
 - ◆ lokal: Prozess ersetzt nur immer seine eigenen Seiten
 - Seitenfehler-Verhalten liegt nur in der Verantwortung des Prozesses
 - ◆ global: Prozess ersetzt auch Seiten anderer Prozesse
 - bessere Effizienz, da ungenutzte Seiten von anderen Prozessen verwendet werden können

8 Seitenflattern (*Thrashing*)

- Ausgelagerte Seite wird gleich wieder angesprochen
 - ◆ Prozess verbringt mehr Zeit mit dem Warten auf das Beheben von Seitenfehler als mit der eigentlichen Ausführung
- Ursachen
 - ◆ Prozess ist nahe am Seitenminimum
 - ◆ zu viele Prozesse gleichzeitig im System
 - ◆ schlechte Ersetzungsstrategie
- ★ Lokale Seitenanforderung behebt Thrashing zwischen Prozessen
- ★ Zuteilung einer genügend großen Zahl von Kacheln behebt Thrashing innerhalb der Prozessseiten
 - ◆ Begrenzung der Prozessanzahl

8.1 Deaktivieren von Prozessen

- Einführung von „Superzuständen“



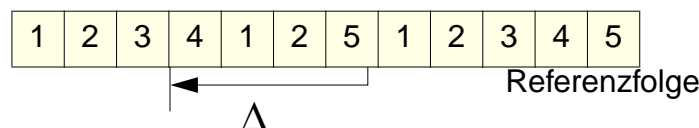
- ◆ inaktiver Prozess benötigt keine Kacheln; Prozess ist vollständig ausgelagert (swapped out)

8.1 Deaktivieren von Prozessen (2)

- 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
 - ◆ guter Kandidat: Prozess mit wenigen Seiten im Hauptspeicher
 - geringe Latenz bei Wiedereinlagerung bzw. wenige Seitenfehler bei Aktivierung und Demand paging

8.2 Arbeitsmengenmodell

- Menge der Seiten, die ein Prozess wirklich braucht (*Working set*)
 - ◆ kann nur angenähert werden, da üblicherweise nicht vorhersehbar
- Annäherung durch Betrachten der letzten Δ Seiten, die angesprochen wurden
 - ◆ geeignete Wahl von Δ
 - zu groß: Überlappung von lokalen Zugriffsmustern zu klein: Arbeitsmenge enthält nicht alle nötigen Seiten



- **Hinweis:** $\Delta >$ Arbeitsmenge, da Seiten in der Regel mehrfach hintereinander angesprochen werden

8.2 Arbeitsmengenmodell (2)

- Beispiel: Arbeitsmengen bei verschiedenen Δ

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
$\Delta = 3$	Seite 1	x	x	x		x	x	x	x	x	x		
	Seite 2		x	x	x		x	x	x	x	x	x	
	Seite 3			x	x	x					x	x	x
	Seite 4				x	x	x					x	x
	Seite 5							x	x	x			x
$\Delta = 4$	Seite 1	x	x	x	x	x	x	x	x	x	x	x	
	Seite 2		x	x	x	x	x	x	x	x	x	x	x
	Seite 3			x	x	x	x				x	x	x
	Seite 4				x	x	x	x				x	x
	Seite 5							x	x	x	x		x

8.2 Arbeitsmengenmodell (3)

- Annäherung der Zugriffe durch die Zeit
 - ◆ bestimmtes Zeitintervall ist ungefähr proportional zu Anzahl von Speicherzugriffen
- ▲ Virtuelle Zeit des Prozesses muss gemessen werden
 - ◆ nur die Zeit relevant, in der der Prozess im Zustand laufend ist
 - ◆ Verwalten virtueller Uhren pro Prozess

8.3 Arbeitsmengenbestimmung mit Zeitgeber

- Annäherung der Arbeitsmenge mit
 - ◆ Referenzbit
 - ◆ Altersangabe pro Seite (Zeitintervall ohne Benutzung)
 - ◆ Timer-Interrupt (durch Zeitgeber)
- Algorithmus
 - ◆ durch regelmäßigen Interrupt wird mittels Referenzbit die Altersangabe fortgeschrieben:
 - ist Referenzbit gesetzt (Seite wurde benutzt) wird das Alter auf Null gesetzt;
 - ansonsten wird Altersangabe erhöht.
 - Es werden nur die Seiten des gerade laufenden Prozesses „gealtert“.
 - ◆ Seiten mit Alter $> \Delta$ sind nicht mehr in der Arbeitsmenge des jeweiligen Prozesses

8.3 Arbeitsmengenbestimmung mit Zeitgeber (2)

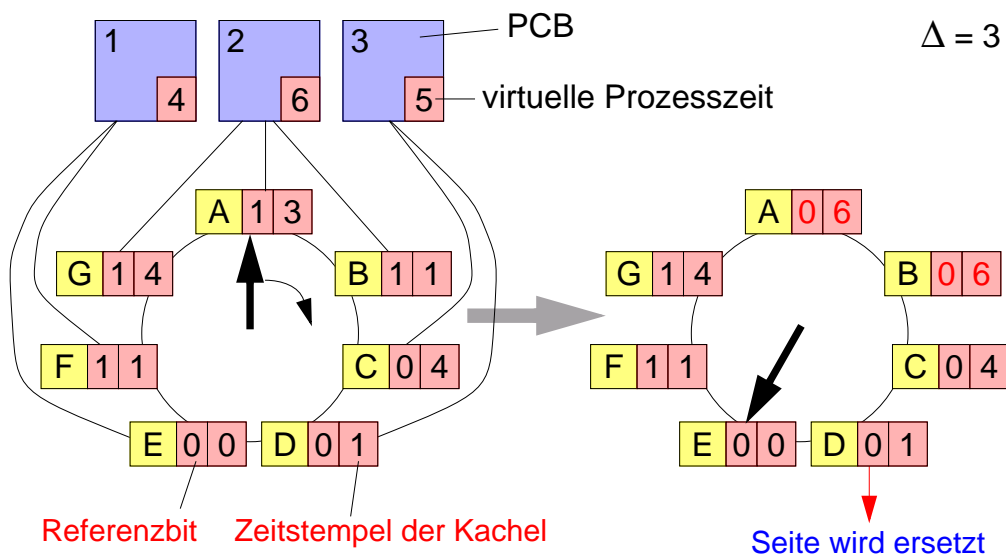
- ▲ Ungenau: System ist aber nicht empfindlich auf diese Ungenauigkeit
 - ◆ Verringerung der Zeitintervalle: höherer Aufwand, genauere Messung
- ▲ Ineffizient
 - ◆ große Menge von Seiten zu betrachten

8.4 Arbeitsmengenbestimmung mit WSClock

- Algorithmus WSClock (Working set clock)
 - ◆ arbeitet wie Clock
 - ◆ Seite wird nur dann ersetzt, wenn sie nicht zur Arbeitsmenge ihres Prozesses gehört oder der Prozess deaktiviert ist
 - ◆ Bei Zurücksetzen des Referenzbits wird die virtuelle Zeit des jeweiligen Prozesses eingetragen, die z.B. im PCB gehalten und fortgeschrieben wird
 - ◆ Bestimmung der Arbeitsmenge erfolgt durch Differenzbildung von virtueller Zeit des Prozesses und Zeitstempel in der Kachel

8.4 Arbeitsmengenbestimmung mit WSClock (2)

■ WSClock Algorithmus

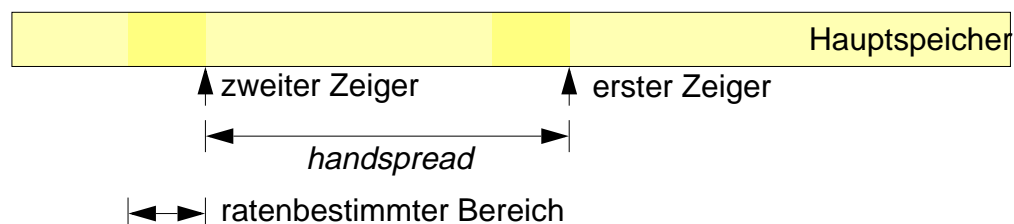


8.5 Probleme mit Arbeitsmengen

- ▲ Zuordnung zu einem Prozess nicht immer möglich
 - ◆ gemeinsam genutzte Seiten in modernen Betriebssystemen eher die Regel als die Ausnahme
 - Seiten des Codesegments
 - Shared libraries
 - Gemeinsame Seiten im Datensegment (*Shared memory*)
- ★ moderne System bestimmen meist eine globale Arbeitsmenge von Seiten

8.6 Ersetzungsstrategie bei Solaris

- Prozess *pageout* arbeitet Clock-Strategie ab
 - ◆ Prozess läuft mehrmals die Sekunde (4x)
 - ◆ adaptierbare Rate: untersuchte Seiten pro Sekunde
 - ◆ statt ein Zeiger: zwei Zeiger
 - am ersten Zeiger werden Referenzbits zurückgesetzt
 - am zweiten Zeiger werden Seiten mit gelöschttem Ref.-Bit ausgewählt
 - nötig, weil sonst Zeitspanne zwischen Löschen und Auswählen zu lang wird (großer Hauptspeicher; 64 MByte entsprechen 8.192 Seiten)
 - Zeigerabstand einstellbar (*handspread*)



8.6 Ersetzungsstrategie bei Solaris (2)

- ◆ ermittelte Seiten werden ausgelagert (falls nötig) und
- ◆ in eine Freiliste eingehängt
- ◆ aus der Freiliste werden Kacheln für Einlagerungen angefordert
- ◆ Seitenfehler können unbenutzte Seiten aus der Freiliste wieder zurückfordern (*Minor page faults*)

8.6 Ersetzungsstrategie bei Solaris (3)

- Verhalten von *pageout* orientiert sich an Größe der Freiliste (Menge des freien Speichers)

— Bereich von *freemem* —————>



Prozesse werden deaktiviert falls mehr als 5sec unter *minfree*

Deaktivierung falls mehr als 30sec unter *desfree*
pageout wird explizit aufgeweckt

pageout läuft regelmäßig

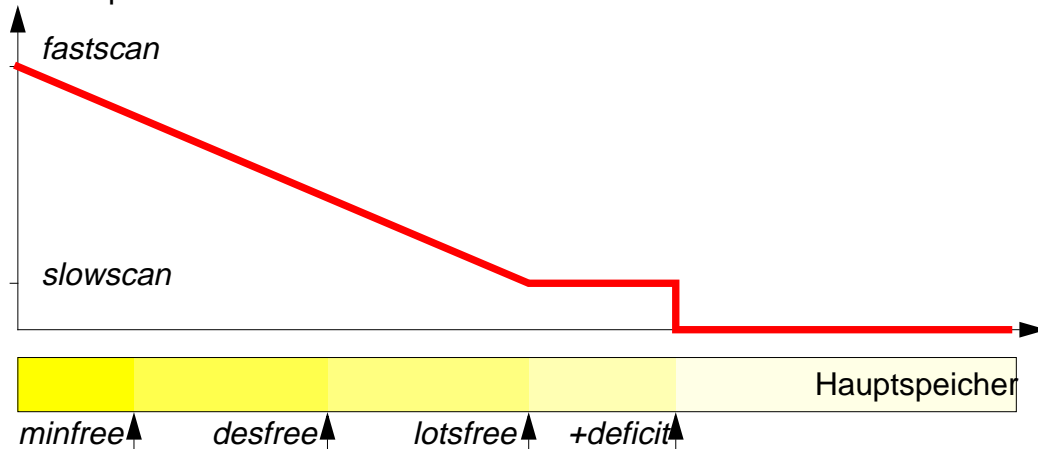
pageout läuft nicht

- ◆ *deficit* wird dynamisch ermittelt (0 bis *lotsfree*) und auf *lotsfree* addiert
 - entspricht Vorschau auf künftige große Speicheranforderungen

8.6 Ersetzungsstrategie bei Solaris (4)

■ Seitenuntersuchungsrate des *pageout* Prozesses

Seiten pro Sekunde



◆ je weniger freier Speicher verfügbar ist, desto höher wird die Untersuchungsrate

◆ *slowscan* und *fastscan* sind einstellbar

Systemprogrammierung I

© 1997-2001, Franz J. Hauck, Inf 4, Univ. Erlangen-Nürnberg [E-Memory.fm, 2001-11-08 10.09]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E - 91

8.6 Ersetzungsstrategie bei Solaris (5)

■ Weitere Parameter

◆ *maxpgio*: maximale Transferrate bei Auslagerungen (vermeidet Plattensaturierung)

◆ *autoup*: Zeitdauer des regelmäßigen Auslagerns alter Seiten durch den Prozess *flushd* (Default: alle 30 sec)

■ Aktivieren und Deaktivieren (*Swap in*, *Swap out*)

◆ Auswahl wird dem Scheduler überlassen

◆ Deaktivierung wird lediglich von Speicherverwaltung angestoßen

Systemprogrammierung I

© 1997-2001, Franz J. Hauck, Inf 4, Univ. Erlangen-Nürnberg [E-Memory.fm, 2001-11-08 10.09]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E - 92

8.6 Ersetzungsstrategie bei Solaris (6)

■ Typische Werte

- ◆ *minfree*: 1/64 des Hauptspeichers (Solaris 2.2), 25 Seiten (Solaris 2.4)
- ◆ *desfree*: 1/32 des Hauptspeichers (Solaris 2.2), 50 Seiten (Solaris 2.4)
- ◆ *lotsfree*: 1/16 des Hauptspeichers (Solaris 2.2), 128 Seiten (Solaris 2.4)
- ◆ *deficit*: 0 bis *lotsfree*
- ◆ *fastscan*: min(1/4 Hauptspeicher, 64 MByte) pro Sekunde (Solaris 2.4)
- ◆ *slowscan*: 800 kBytes pro Sekunde (Solaris 2.4)
- ◆ *handspread*: wie *fastscan* (Solaris 2.4)

8.7 Ersetzungsstrategie bei Windows 2000

■ Freiliste von freien Kacheln

■ Arbeitsmenge pro Prozess

- ◆ zunächst vorbestimmt
- ◆ Anpassung der Arbeitsmenge durch Working-Set-Manager
- ◆ Arbeitszyklus des Working-Set-Managers wird durch Speicherknappheit beschleunigt
- ◆ Auslagerungsstrategie nach WSClock (nur Monoprozessorvariante) unter Berücksichtigung von Prozessklassen
- ◆ prozessspezifische Konfiguration der Arbeitsmenge durch Anwender