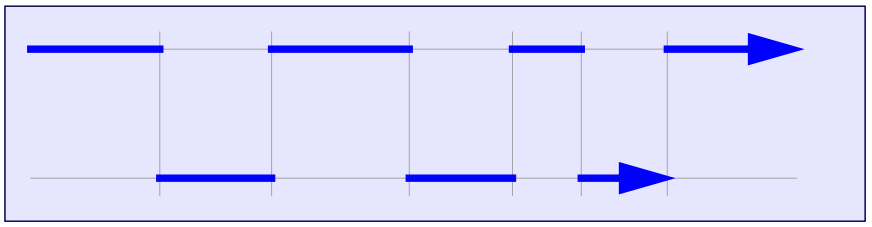
2 Koroutinen

■ Einsatz von Koroutinen

◆ einige Anwendungen lassen sich mit Hilfe von Koroutinen (auf Benutzerebene) innerhalb eines Prozesses gut realisieren



ein Prozess
zwei Koroutinen

▲ Nachteile:

◆ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)

◆ in Multiprozessorsystemen keine parallelen Abläufe möglich

◆ Wird eine Koroutine in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert.

3 Aktivitätsträger

★ **Lösungsansatz:**
Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(Lightweighted Processes, LWPs)*

◆ Eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln.

- Instruktionen
- Datenbereiche
- Dateien, Semaphoren etc.

◆ Jeder Thread repräsentiert eine eigene Aktivität:

- eigener Programmzähler
- eigener Registersatz
- eigener Stack

3 Aktivitätsträger (2)

- ◆ Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.
  - Es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
  - Speicherarbbildung muss nicht gewechselt werden.
  - Alle Systemressourcen bleiben verfügbar.

■ Ein UNIX-Prozess ist ein Adressraum mit einem Thread

- ◆ Solaris: Prozess kann mehrere Threads besitzen

■ Implementierungen von Threads

- ◆ User-level Threads
- ◆ Kernel-level Threads

4 User-Level-Threads

■ Implementierung

- ◆ Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
- ◆ Betriebssystem sieht nur einen Thread

★ Vorteile

- ◆ keine Systemaufrufe zum Umschalten erforderlich
- ◆ effiziente Umschaltung
- ◆ Schedulingstrategie in der Hand des Anwenders

▲ Nachteile

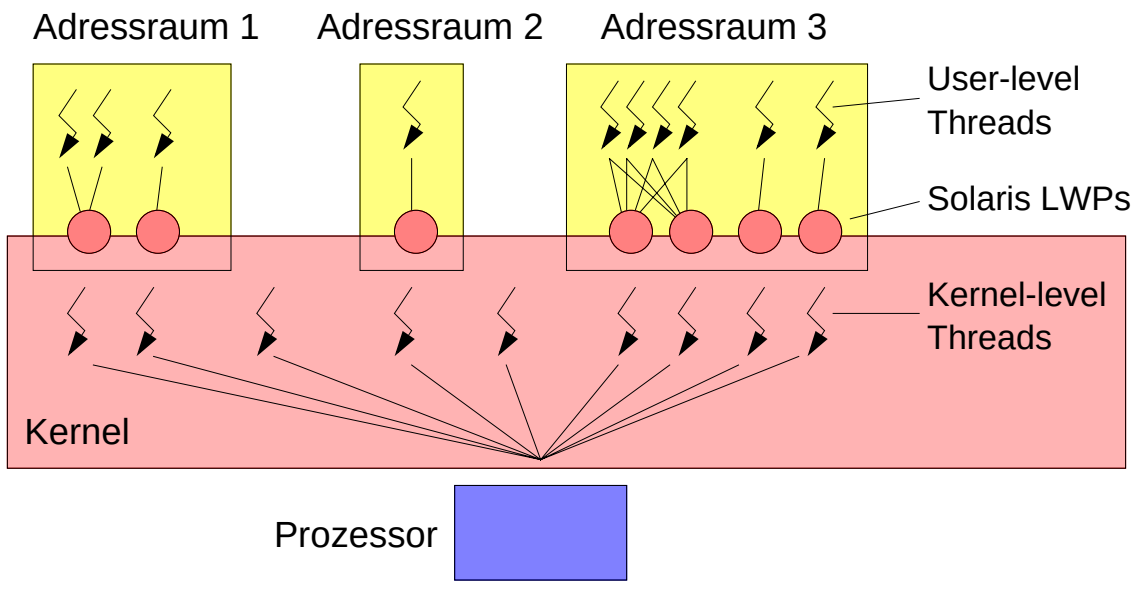
- ◆ Bei blockierenden Systemaufrufen bleiben alle User-Level-Threads stehen.

5 Kernel-Level-Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level-Threads
  - Betriebssystem schaltet Threads um
- ★ Vorteile
  - kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
- ▲ Nachteile
  - weniger effizientes Umschalten
  - Fairnessverhalten nötig
    (zwischen Prozessen mit vielen und solchen mit wenigen Threads)
  - Schedulingstrategie meist vorgegeben

6 Beispiel: LWPs und Threads (Solaris)

- Solaris kennt Kernel- ,User-Level-Threads und LWPs



Nach Silberschatz, 1994

D.6 Koordinierung

- Beispiel: Beobachter und Protokollierer
  - Mittels Induktionsschleife werden Fahrzeuge gezählt. Alle 10min druckt der Protokollierer die im letzten Zeitraum vorbeigekommene Anzahl aus.

```
int cnt= 0;

Observer
    on indication
        cnt= cnt+1;

Logger
    every 10 minutes
        printf( "Count= %d\n", cnt );
        cnt= 0;
```

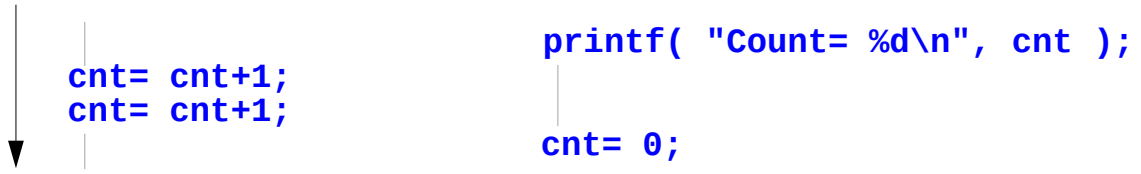

D.6 Koordinierung (2)

- Effekte:
  - Fahrzeuge gehen „verloren“
  - Fahrzeuge werden doppelt gezählt
- Ursachen:
  - Befehle in C werden nicht unteilbar (atomar) abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden.
  - In C werden keinesfalls mehrere Anweisungen zusammen atomar abgearbeitet.
  - Prozesswechsel innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen können zu Inkonsistenzen führen.

D.6 Koordinierung (3)

▲ Fahrzeuge gehen „verloren“

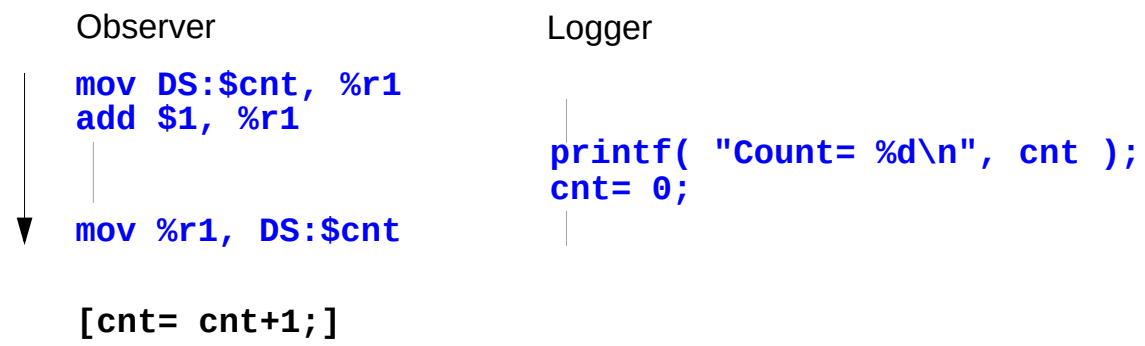
◆ Nach dem Drucken wird der Protokollierer unterbrochen. Beobachter zählt weitere Fahrzeuge. Anzahl wird danach ohne Beachtung vom Protokollierer auf Null gesetzt.



D.6 Koordinierung (4)

▲ Fahrzeuge werden doppelt gezählt:

◆ Beobachter will Zähler erhöhen und holt sich diesen dazu in ein Register. Er wird unterbrochen und der Protokollierer setzt Anzahl auf Null. Beobachter erhöht Registerwert und schreibt diesen zurück. Dieser Wert wird erneut vom Protokollierer registriert.



D.6 Koordinierung (5)

- Gemeinsame Nutzung von Daten oder Betriebsmitteln
  - kritische Abschnitte:
    - nur einer soll Zugang zu Daten oder Betriebsmitteln haben (gegenseitiger Ausschluss, *Mutual Exclusion*, *Mutex*)
    - kritische Abschnitte erscheinen allen anderen als zeitlich unteilbar
  - Wie kann der gegenseitige Ausschluss in kritischen Abschnitten erzielt werden?
- Koordinierung allgemein:
  - Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern
- ★ Hinweis:
  - Im Folgenden wird immer von Prozessen die Rede sein. Koordinierung kann/muss selbstverständlich auch zwischen Threads stattfinden.

1 Gegenseitiger Ausschluss

- Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten
  - Annahme: Maschinenbefehle sind unteilbar (atomar)
- 1. Versuch

```
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    while( turn == 1 );

    ... /* critical sec. */

    turn= 1;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    while( turn == 0 );

    ... /* critical sec. */

    turn= 0;

    ... /* uncritical */
}
```

1 Gegenseitiger Ausschluss (2)

- Probleme der Lösung
  - nur alternierendes Betreten des kritischen Abschnitts durch P_0 und P_1 möglich
  - Implementierung ist unvollständig
  - aktives Warten

- Ersetzen von **`turn`** durch zwei Variablen **`ready0`** und **`ready1`**
  - **`ready0`** zeigt an, dass Prozess 0 bereit für den kritischen Abschnitt ist
  - **`ready1`** zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist

1 Gegenseitiger Ausschluss (3)

- 2. Versuch

```
bool ready0= FALSE;
bool ready1= FALSE;
```

Prozess 0

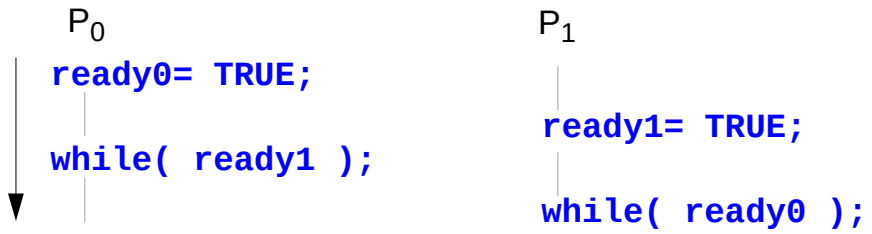
```
while( 1 ) {
    ready0= TRUE;
    while( ready1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    while( ready0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```

1 Gegenseitiger Ausschluss (4)

- Gegenseitiger Ausschluss wird erreicht
- Probleme der Lösung
  - aktives Warten
  - Verklemmung möglich *(Lifelock)*



- Kombination beider Algorithmen führt zu Versuch 3

1 Gegenseitiger Ausschluss (5)

- 3. Versuch (Algorithmus von Peterson, 1981)

```
bool ready0= FALSE;
bool ready1= FALSE;
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    turn= 1;
    while( ready1 &&
           turn == 1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    turn= 0;
    while( ready0 &&
           turn == 0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```

1 Gegenseitiger Ausschluss (6)

- Algorithmus implementiert gegenseitigen Ausschluss
  - vollständige und sichere Implementierung
  - **`turn`** entscheidet für den kritischen Fall von Versuch 2, welcher Prozess nun wirklich den kritischen Abschnitt betreten darf
  - in allen anderen Fällen ist **`turn`** unbedeutend
- Problem der Lösung
  - aktives Warten
- Algorithmus auch für mehrere Prozesse erweiterbar
  - Lösung ist relativ aufwendig

2 Spezielle Maschinenbefehle

- Spezielle Maschinenbefehle können die Programmierung kritischer Abschnitte unterstützen und vereinfachen
  - *Test-and-Set* Instruktion
  - *Swap* Instruktion
- Test-and-set
  - Maschinenbefehl mit folgender Wirkung

```
bool test_and_set( bool *plock )
{
    bool tmp= *plock;
    *plock= TRUE;
    return tmp;
}
```

  - Ausführung ist atomar

2 Spezielle Maschinenbefehle (2)

◆ Kritische Abschnitte mit Test-and-Set Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

2 Spezielle Maschinenbefehle (3)

■ Swap

◆ Maschinenbefehl mit folgender Wirkung

```
void swap( bool *ptr1, bool *ptr2)
{
    bool tmp= *ptr1;
    *ptr1= *ptr2;
    *ptr2= tmp;
}
```

◆ Ausführung ist atomar

2 Spezielle Maschinenbefehle (4)

◆ Kritische Abschnitte mit Swap Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

Prozess 1

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

3 Kritik an den bisherigen Verfahren

★ Spinlock

◆ bisherige Verfahren werden auch Spinlocks genannt

◆ aktives Warten

▲ Problem des aktiven Wartens

◆ Verbrauch von Rechenzeit ohne Nutzen

◆ Behinderung „nützlicher“ Prozesse

◆ Abhängigkeit von der Schedulingstrategie

- nicht anwendbar bei nicht-verdrängenden Strategien
- schlechte Effizienz bei langen Zeitscheiben

■ Spinlocks kommen heute fast ausschließlich in Multiprozessorsystemen zum Einsatz

◆ bei kurzen kritischen Abschnitten effizient

◆ Koordinierung zwischen Prozessen von mehreren Prozessoren

4 Sperrung von Unterbrechungen

- Sperrung der Systemunterbrechungen im Betriebssystems

Prozess 0

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

Prozess 1

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

- ◆ nur für kurze Abschnitte geeignet
  - sonst Datenverluste möglich
- ◆ nur innerhalb des Betriebssystems möglich
  - privilegierter Modus nötig
- ◆ nur für Monoprozessoren anwendbar
  - bei Multiprozessoren arbeiten andere Prozesse echt parallel

5 Semaphore

- Datenstruktur des Systems mit zwei Operationen (nach *Dijkstra*)
- ◆ P-Operation *(proberen; passeren; wait; down)*
  - wartet bis Zugang frei

```
void P( int *s )
{
    while( *s <= 0 );
    *s= *s-1;
}
```

atomare Funktion

- ◆ V-Operation *(verhogen; vrijgeven; signal; up)*
  - macht Zugang für anderen Prozess frei

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

atomare Funktion

5 Semaphore (2)

- Implementierung kritischer Abschnitte mit Semaphore

```
int lock= 1;
```

Prozess 0

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

Prozess 1

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

- Problem:
  - Implementierung von P und V

5 Semaphore (3)

- Implementierung im Betriebssystem (Monoprozessor)

P-Operation

Unterbrechungen sperren	
***s <= 0** ja	nein
ok = FALSE;	
laufenden Prozess in Zustand *blockiert* versetzen und in Warteschlange aufnehmen	***s = *s-1** **ok = TRUE;**
Prozessumschalter ansteuern und Unterbrechungen freigeben	Unterbrechungen freigeben
until: ok == TRUE	

ok ist eine prozesslokale Variable

  - jede Semaphore besitzt Warteschlange, die blockierte Prozesse aufnimmt

5 Semaphore (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

5 Semaphore (5)

★ Vorteile einer Semaphore-Implementierung im Betriebssystem

- ◆ Einbeziehen des Schedulers in die Semaphore-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

5 Semaphore (6)

- Abstrakte Beschreibung von zählenden Semaphore (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: für zählende Semaphore

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

D.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

1 Gegenseitiger Ausschluss

- Semaphore
  - ◆ eigentlich reicht eine Semaphore mit zwei Zuständen: binäre Semaphore

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

atomare Funktion

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

atomare Funktion

  - ◆ zum Teil effizienter implementierbar

1 Gegenseitiger Ausschluss (2)

- Abstrakte Beschreibungl: binäre Semaphore

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

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

1 Gegenseitiger Ausschluss (3)

- Automatische Klammerung wünschenswert
  - Beispiel: Java

```
synchronized( lock ) {

... /* critical sec. */

}
```

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

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, der als Ringpuffer wirkt
    - zwei Integer-Variablen enthalten Feldindizes auf den Anfang und das Ende des Ringpuffers
  - eine Semaphore für den gegenseitigen Ausschluss
  - je eine Semaphore für das Blockieren an den Bedingungen „Puffer voll“ und „Puffer leer“
    - Semaphore **`full`** zählt wieviele Zeichen noch in den Puffer passen
    - Semaphore **`empty`** zählt wieviele Zeichen im Puffer sind

2 Bounded Buffers (3)

```
char buffer[N];
int inslot= 0, outslot= 0;
semaphore 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;
}
```


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
  - Zählen der nebenläufig tätigen Leser: Variable **`readcount`**
  - Semaphore für gegenseitigen Ausschluss beim Zugriff auf **`readcount`**: **`mutex`**
  - Semaphore für gegenseitigen Ausschluss von Schreibern untereinander und von Schreibern gegen Leser: **`write`**

3 Erstes Leser-Schreiber-Problem (2)

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

Leser

```
...
P( &mutex );
if( ++readcount == 1 )
    P( &writer );
V( &mutex );

... /* reading */

P( &mutex );
if( --readcount == 0 )
    V( &writer );
V( &mutex );
...
```

Schreiber

```
...
P( &writer );

... /* writing */

V( &writer );
...
```


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	$S := S + k$

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 );
...
```


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 **`writecount`**
  - Semaphore für gegenseitigen Ausschluss beim Zugriff auf **`readcount`**: **`mutexR`**
  - Semaphore für gegenseitigen Ausschluss beim Zugriff auf **`writecount`**: **`mutexW`**
  - Semaphore für gegenseitigen Ausschluss von Schreibern untereinander und von Schreibern gegen Leser: **`write`**
  - Semaphore für den Ausschluss von Lesern, falls Schreiber vorhanden: **`read`**
  - Semaphore zum Klammern des Leservorspanns: **`mutex`**

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 );
...
```


4 Zweites Leser-Schreiber-Problem (3)

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

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

4 Zweites Leser-Schreiber-Problem (4)

- Implementierung mit Up-Down–Semaphore:

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

Leser

```
...
down( &reader, 1, &writer );

... /* reading */

up( &reader, 0 );
...
```

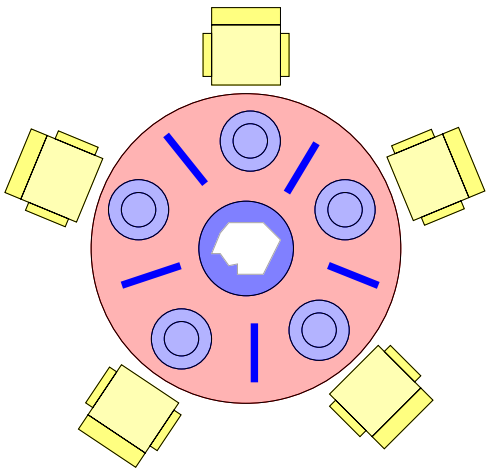
Schreiber

```
...
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`**

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

5 Philosophenproblem (2)

- Naive Implementierung
  - eine Semaphore pro Gabel

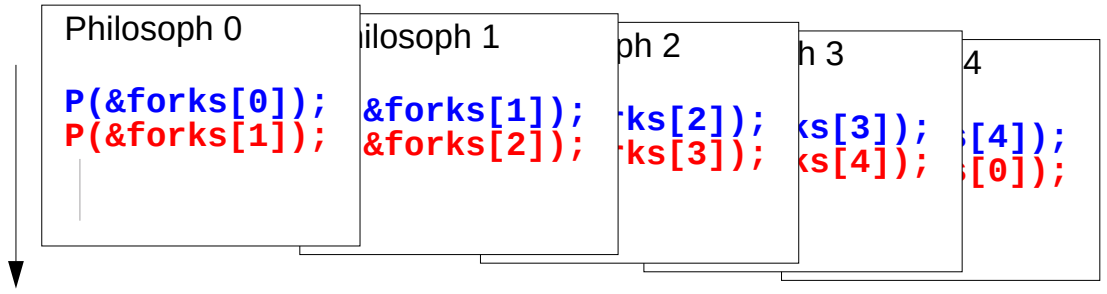
```
semaphore forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, i ∈ [0,4]

```
while( 1 ) {
    ... /* think */

    P( &forks[i] );
    P( &forks[(i+1)%5] );

    ... /* eat */

    V( &forks[i] );
    V( &forks[(i+1)%5] );
}
```


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

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 Semaphore
- ◆ Abstrakte Beschreibung:

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

5 Philosophenproblem (5)

- ◆ Implementierung mit PV-multiple–Semaphore

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

Philosoph i, i ∈ [0,4]

```
while( 1 ) {
    ... /* think */

    Pm( 2, &forks[i], &forks[(i+1)%5] );

    ... /* eat */

    Vm( 2, &forks[i], &forks[(i+1)%5] );
}
```

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 ∈ [0,3]

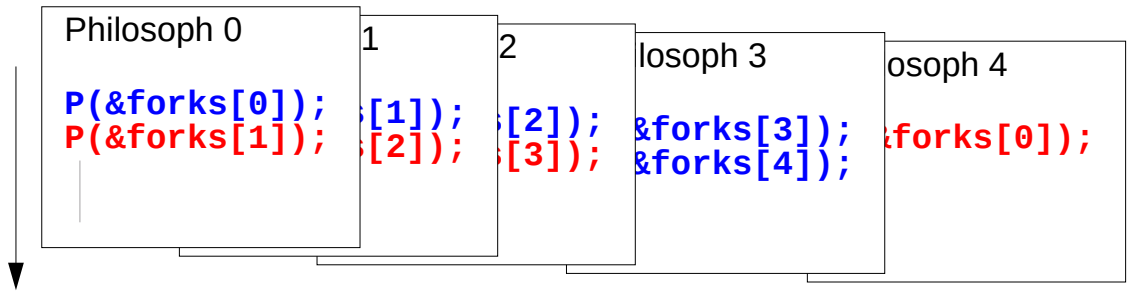
```
while( 1 ) {
    ... /* think */

    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] );
}
```


5 Philosophenproblem (7)

- Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- System verklemmt sich nicht

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`**

6 Schlafende Friseure (2)

- Implementierung mit zählenden Semaphoren (PV System)

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

Barber

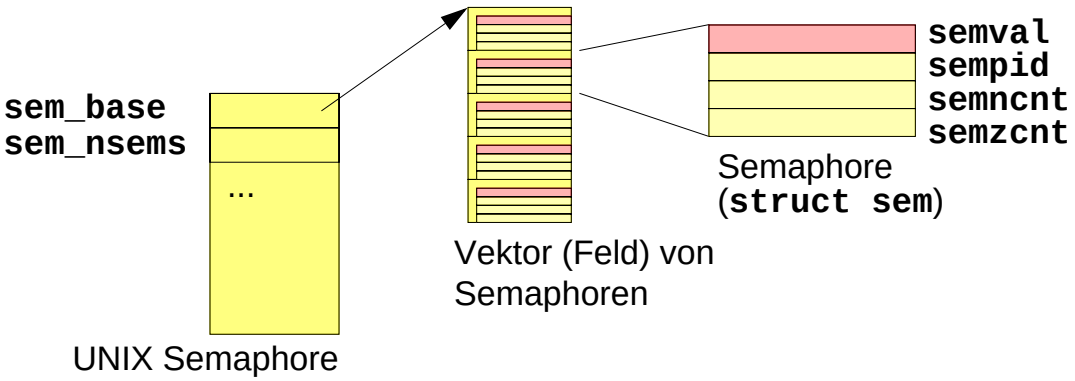
```
while( 1 ) {
    P( &customers );
    P( &mutex );
    waiting--;
    V( &barbers );
    V( &mutex );

    ... /* cut hair */
}
```

Customer

```
P( &mutex );
if( waiting < N ) {
    waiting++
    V( &customers );
    V( &mutex );
    P( &barbers );

    ... /* get hair cut */
}
else {
    V( &mutex );
}
```

D.8 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

Zugriffssrechte; 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)

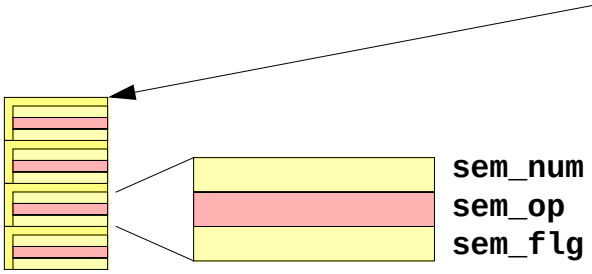
1 Erzeugen einer UNIX-Semaphore (2)

- 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.9 Zusammenfassung

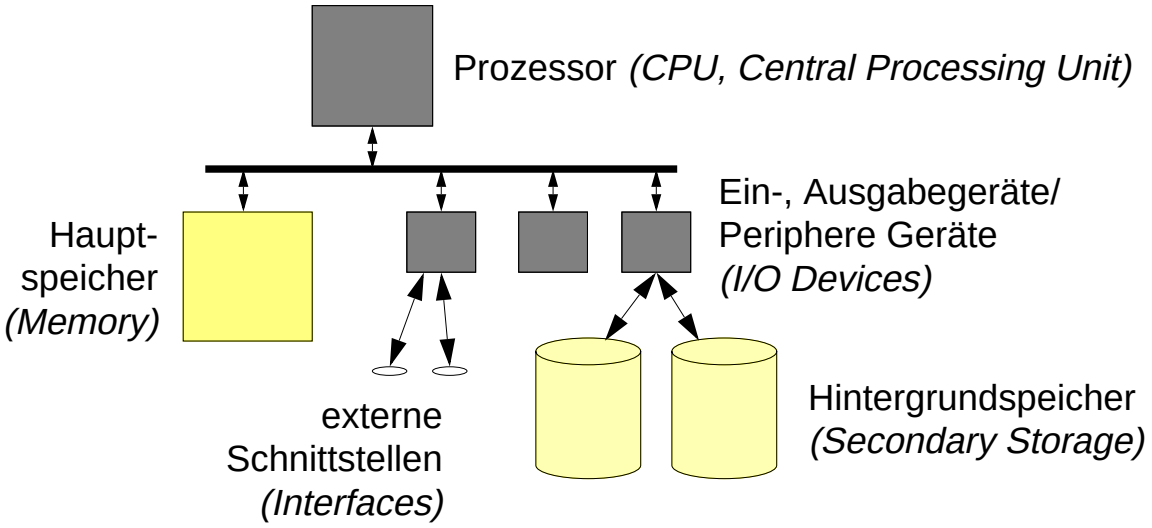
- Programmiermodell: Prozess
  - Zerlegung von Anwendungen in Prozesse oder Threads
  - Ausnutzen von Wartezeiten; Time sharing–Betrieb
  - Prozess hat verschiedene Zustände: laufend, bereit, blockiert etc.
- Auswahlstrategien für Prozesse
  - FCFS, SJF, PSJF, RR, MLFB
- Prozesskommunikation
  - Pipes,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

D.9 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, socket, bind, recvfrom, sendto, listen, accept
  - msgget, msgsnd, msgrcv
  - signal, kill, sigaction
  - semget, semop, semctl

E Speicherverwaltung

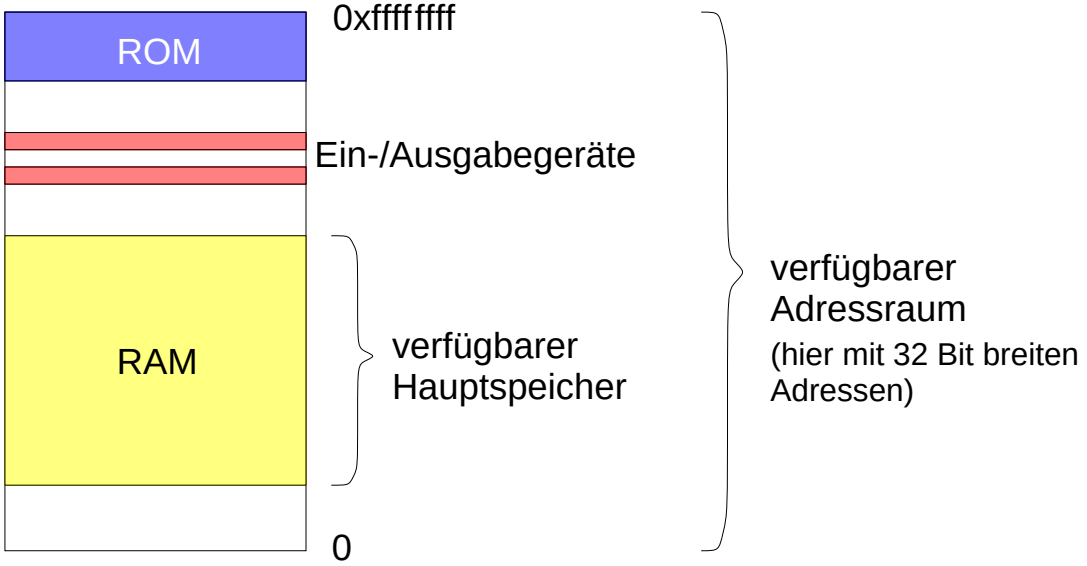
- Betriebsmittel



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

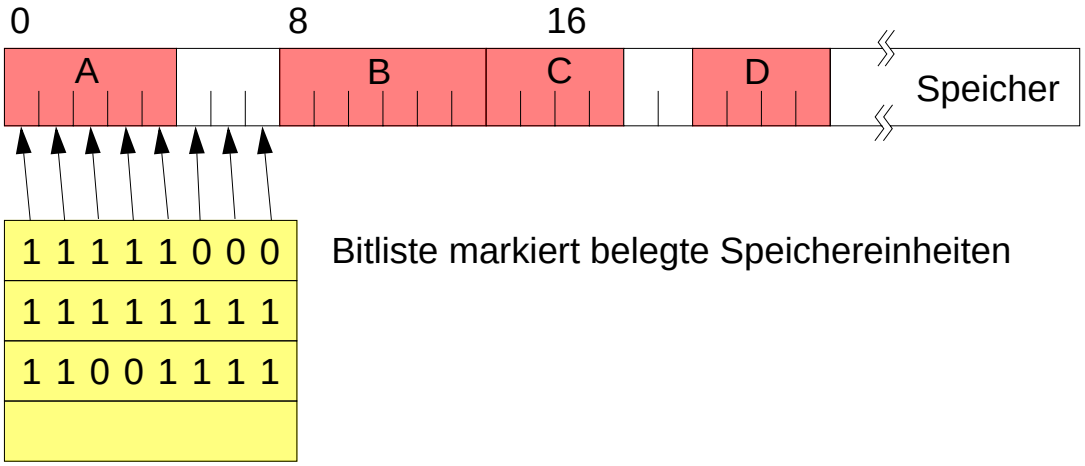
- Feste Bereiche für Betriebssystem und Benutzerprogramm
- ▲ Probleme:
  - Begrenzung anderer Ressourcen
    (z.B. Bandbreite bei Ein-/Ausgabe wg. zu kleiner Systempuffer)
  - Ungenutzter Speicher des Betriebssystems kann von Anwendungsprogramm nicht genutzt werden und umgekehrt
- ★ Dynamische Speicherzuteilung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

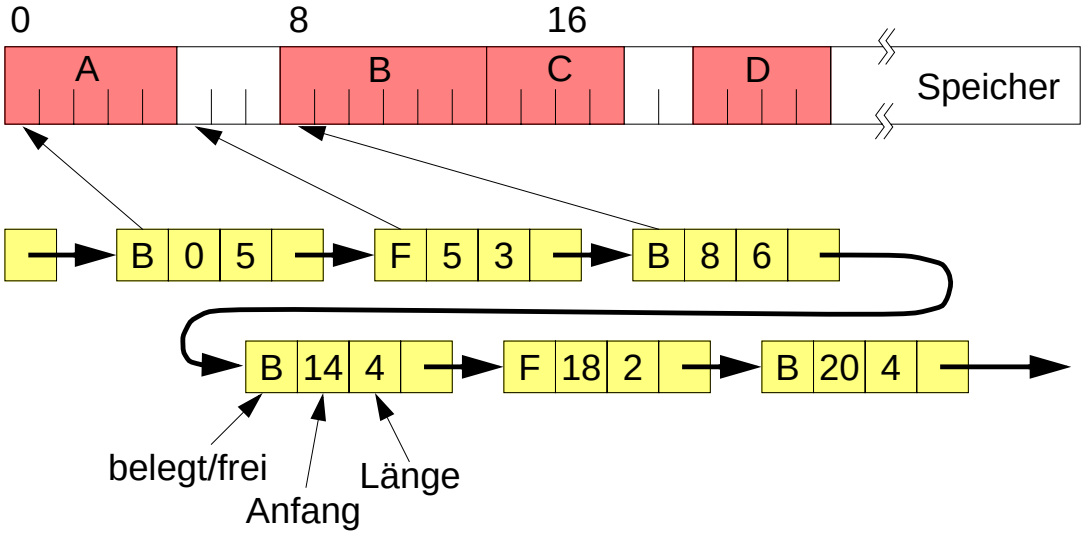


Bitliste markiert belegte Speichereinheiten

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

4 Freispeicherverwaltung (2)

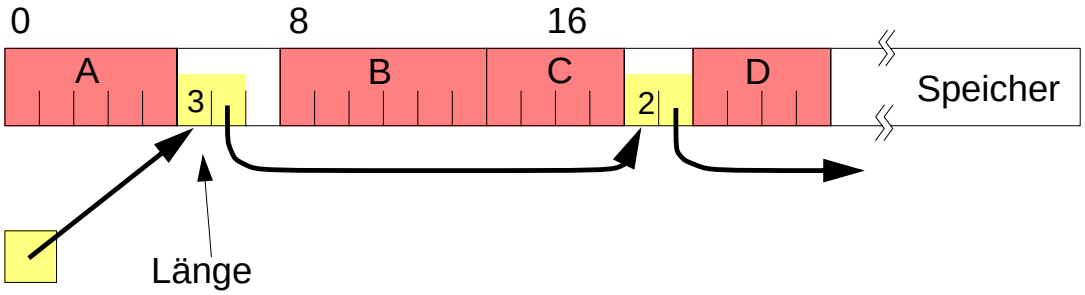
- Verkettete Liste



Repräsentation auch von freien Segmenten

4 Freispeicherverwaltung (3)

- Verkettete Liste in dem freien Speicher

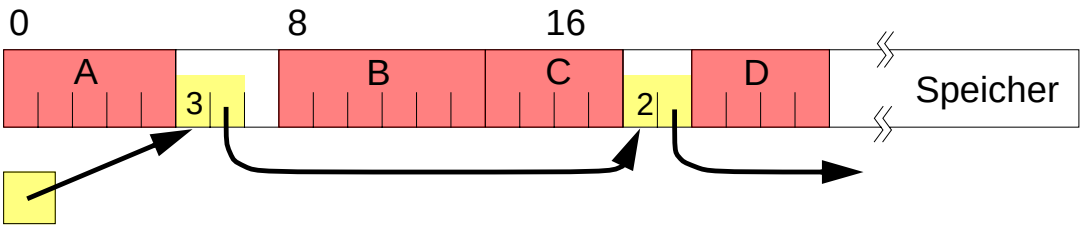


Mindestlückengröße muss garantiert werden

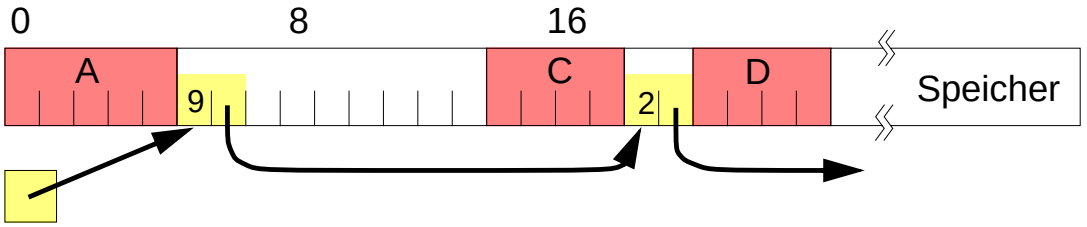
- Zur Effizienzsteigerung eventuell Rückwärtsverkettung nötig
- Repräsentation letztlich auch von der Vergabestrategie abhängig

5 Speicherfreigabe

- Verschmelzung von Lücken



nach Freigabe von B:

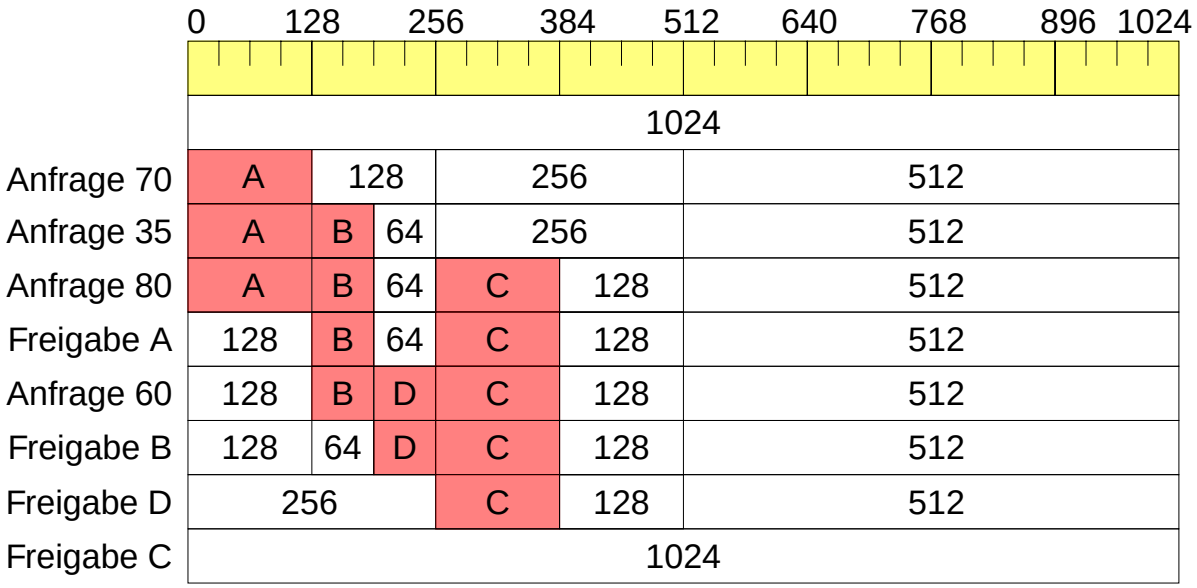


6 Vergabestrategien

- 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

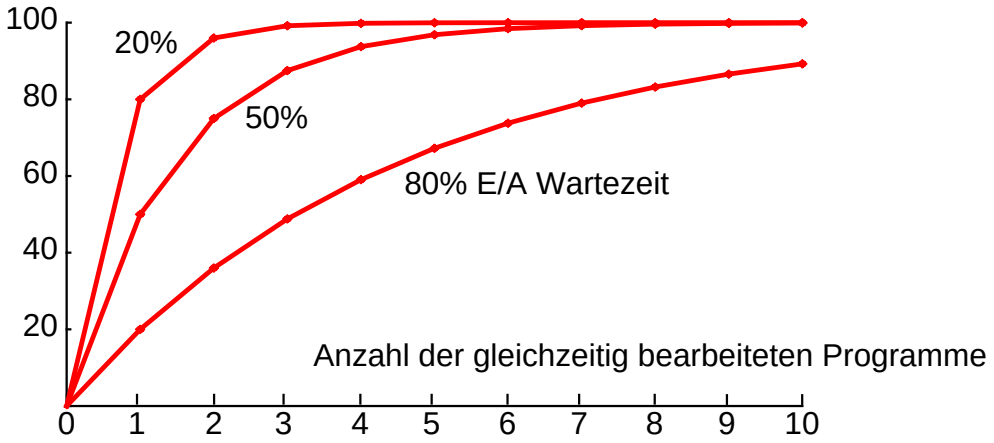
8 Einsatz der Verfahren

- 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

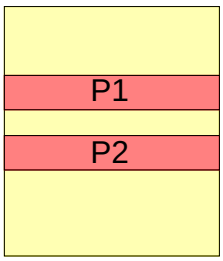
1 Problemstellung

- 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 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 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

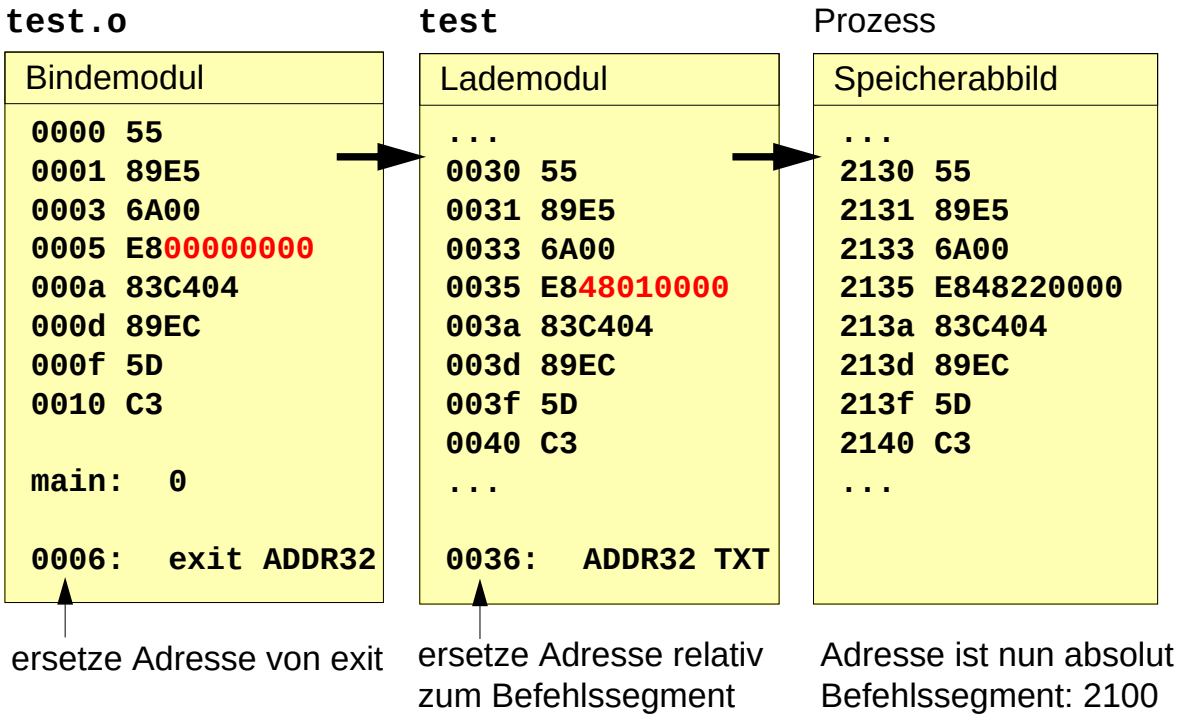
```
0000 55
0001 89E5
0003 6A00
0005 E800000000
000a 83C404
000d 89EC
000f 5D
0010 C3

main:   0

0006:   exit ADDR32
```

Erzeugung beim Compilevorgang

ersetze Adresse von exit

2 Relokation (3)

- Binde- und Ladevorgang

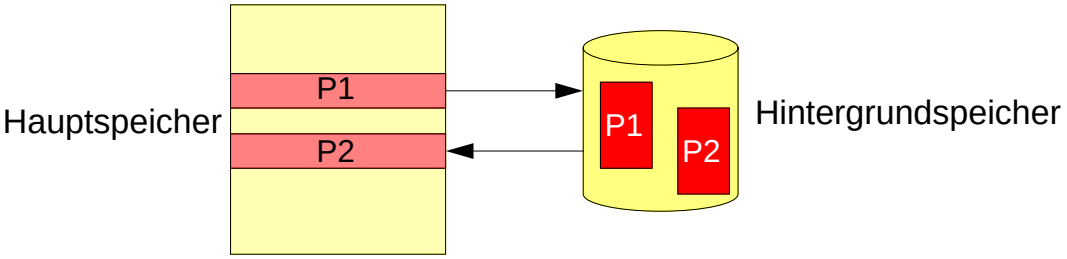


2 Relokation (4)

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

3 Ein-, Auslagerung (*Swapping*)

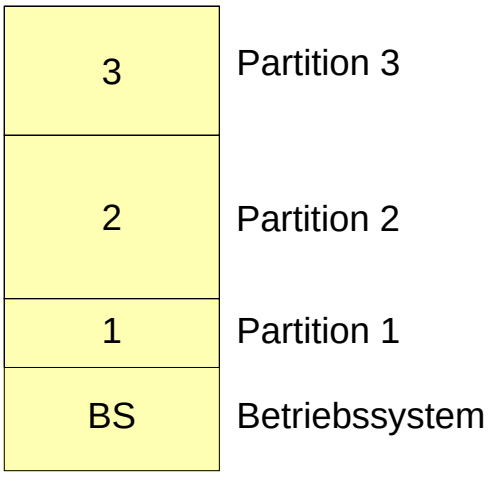
- Segmente eines Prozesses werden auf Hintergrundspeicher ausgelagert und im Hauptspeicher freigegeben
  - z.B. zur Überbrückung von Wartezeiten bei E/A oder Round-Robin Schedulingstrategie
- Einlagern der Segmente in den Hauptspeicher am Ende der Wartezeit



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

3 Ein-, Auslagerung (2)

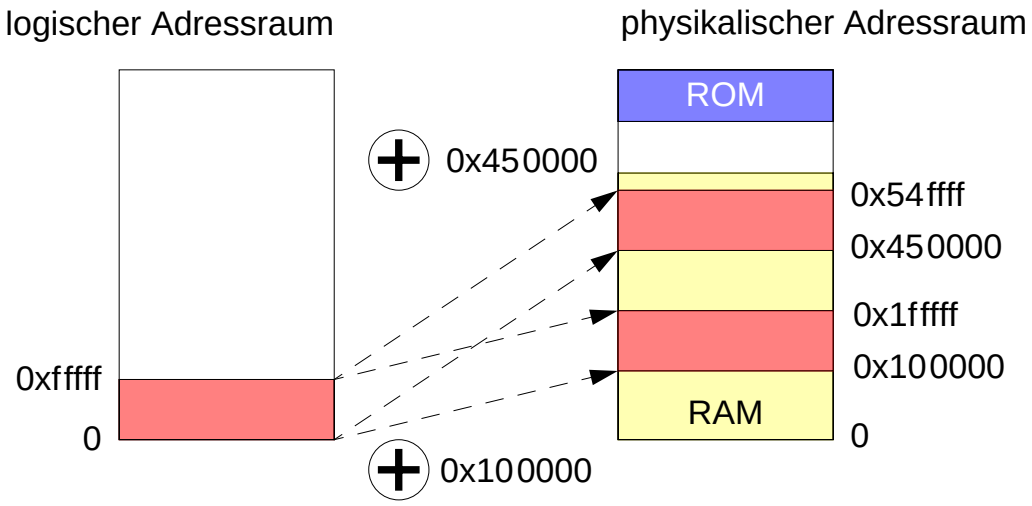
- Prozess ist statisch gebunden
  - kann nur an gleiche Stelle im Hauptspeicher wieder eingelagert werden
  - Kollisionen mit eventuell neu im Hauptspeicher befindlichen Segmenten
- Mögliche Lösung: Partitionierung des Hauptspeichers



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

4 Segmentierung

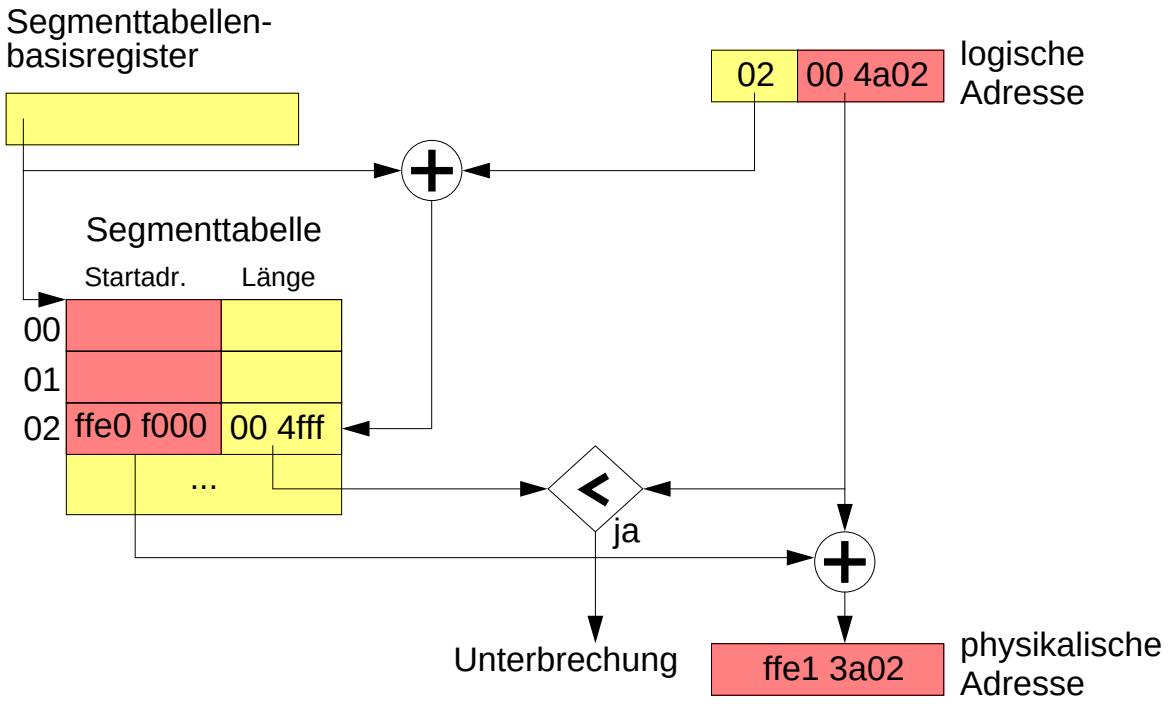
- Hardwareunterstützung: Umsetzung logischer in physikalische Adressen
  - Prozesse erhalten einen logischen Adressraum



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

4 Segmentierung (2)

- Realisierung mit Übersetzungstabelle



4 Segmentierung (3)

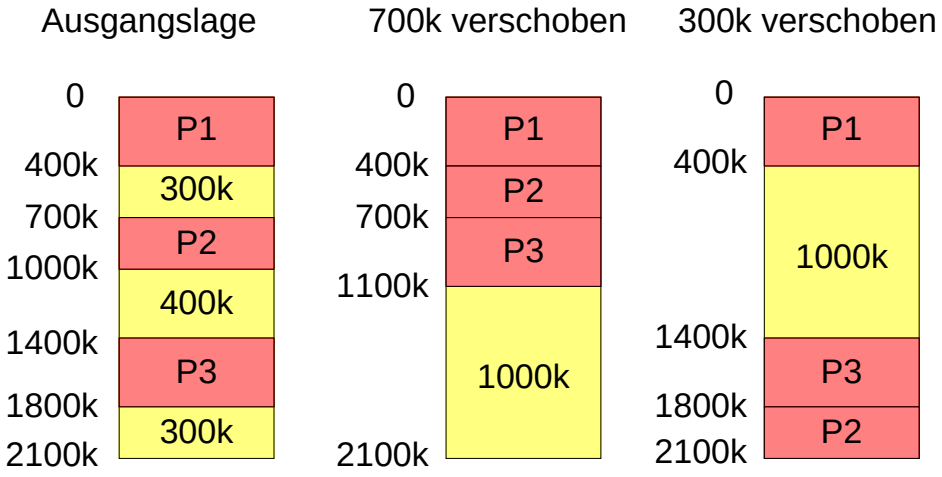
- Hardware wird MMU *(Memory management unit)* genannt
- Schutz vor Segmentübertretung
  - ◆ Unterbrechung zeigt Speicherverletzung an
  - ◆ Programme und Betriebssystem voreinander geschützt
- Prozessumschaltung durch Austausch der Segmentbasis
  - ◆ jeder Prozess hat eigene Übersetzungstabelle
- Ein- 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

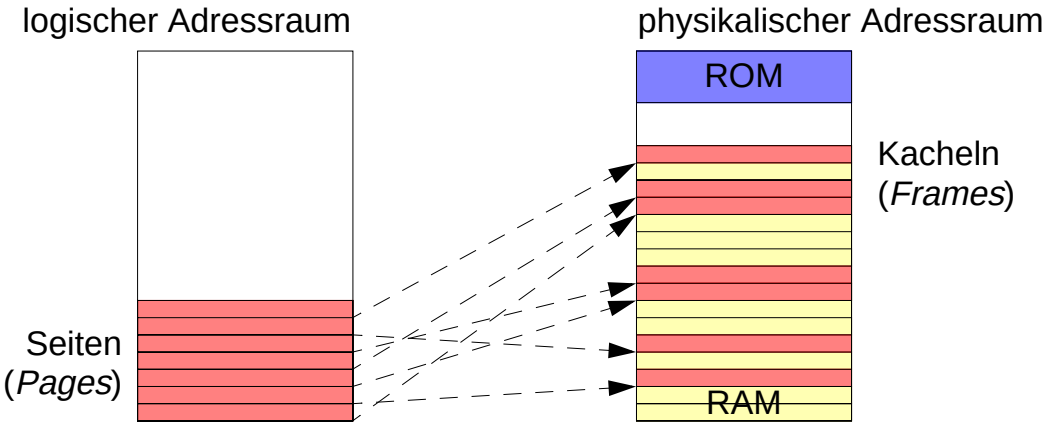
5 Kompaktifizieren

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



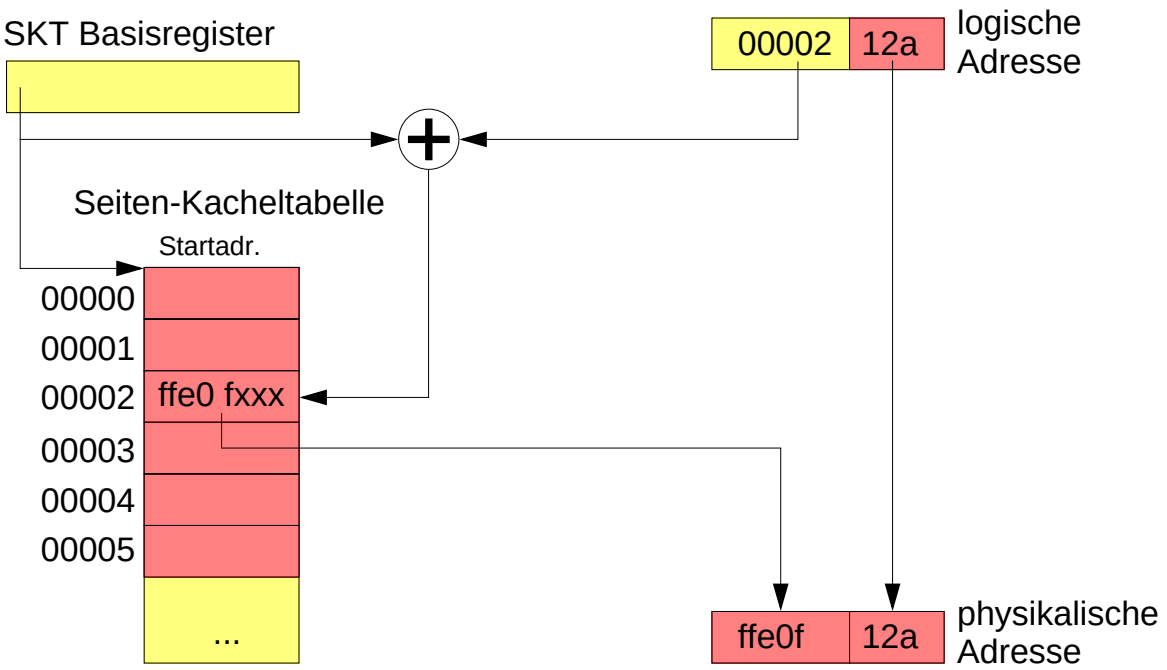
E.3 Seitenadressierung (*Paging*)

- 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 und Ein-, Auslagerungen



1 MMU mit Seiten-Kacheltabelle

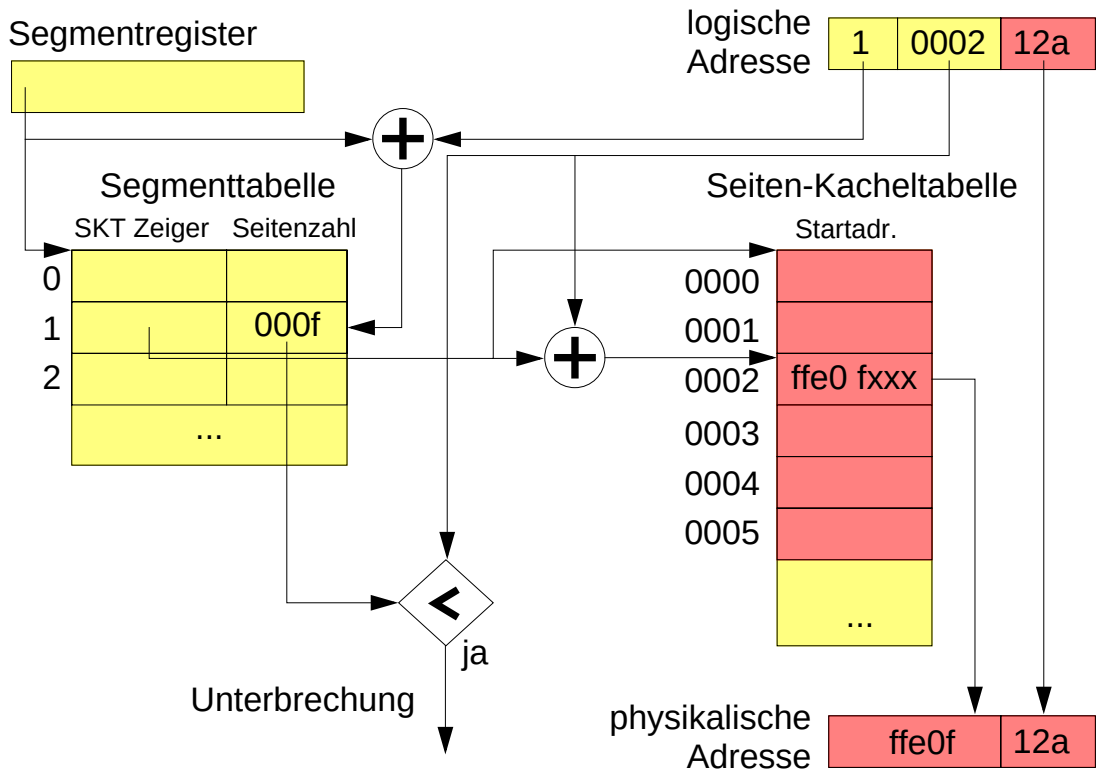
- Tabelle setzt Seiten in Kacheln um



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

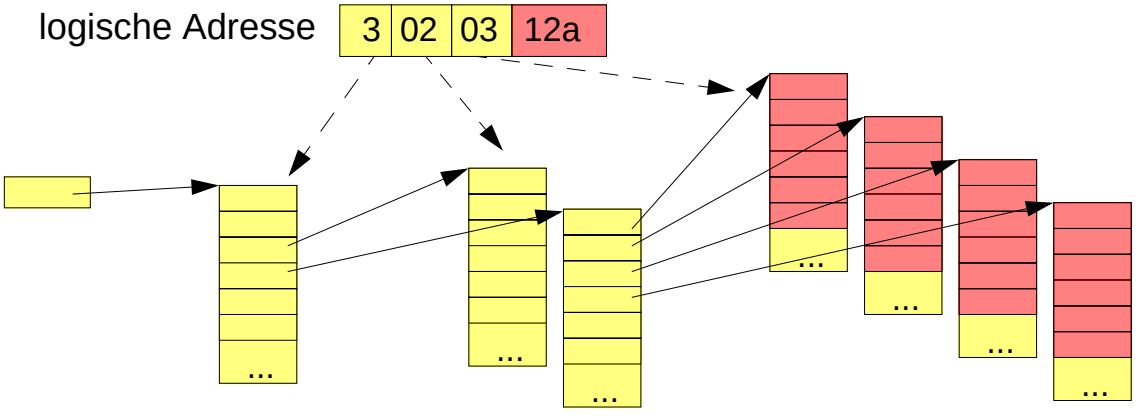
2 Segmentierung und Seitenadressierung



2 Segmentierung und Seitenadressierung (2)

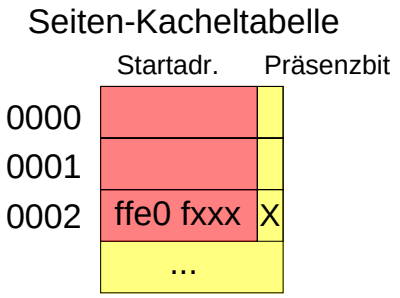
- ▲ noch mehr implizite Speicherzugriffe
- ▲ große Tabellen im Speicher
- ★ Mehrstufige Seitenadressierung mit Ein- und Auslagerung

3 Mehrstufige Seitenadressierung



3 Mehrstufige Seitenadressierung (2)

- Ein- und Auslagerung von Seiten



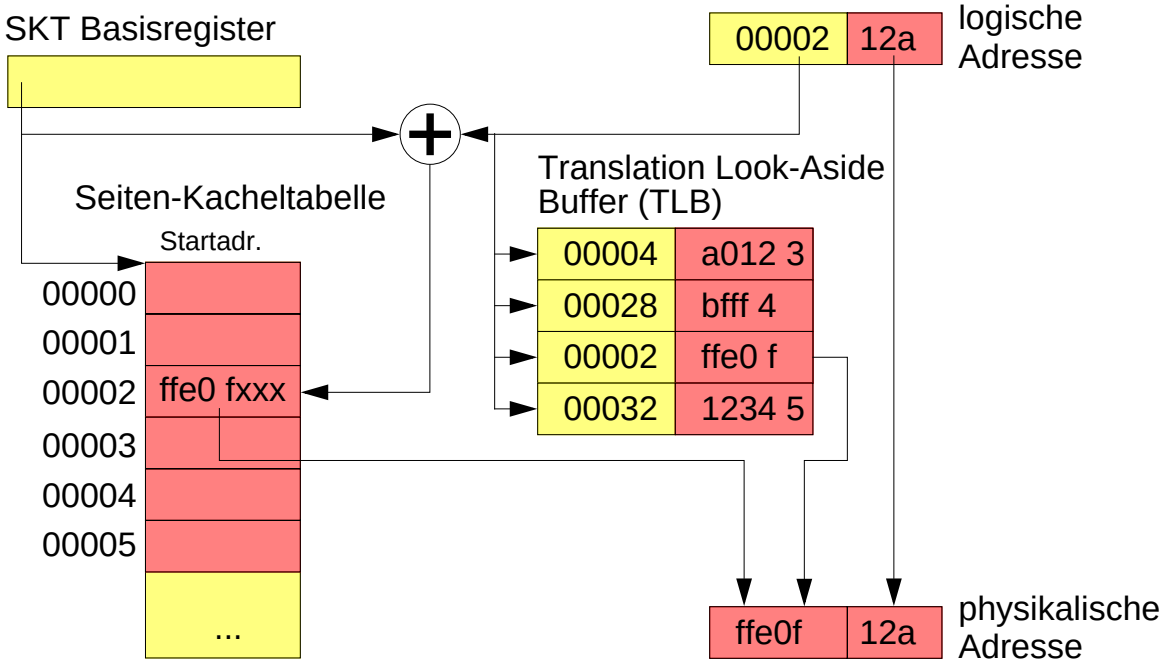
  - Ist das Präsenzbit gesetzt, bleibt alles wie bisher.
  - Ist das Präsenzbit gelöscht, wird eine Unterbrechung ausgelöst (*Page fault*).
  - Die Unterbrechungsbehandlung kann nun für das Laden der Seite vom Hintergrundspeicher sorgen und den Speicherzugriff danach wiederholen (benötigt HW Support in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