Prozessverwaltung

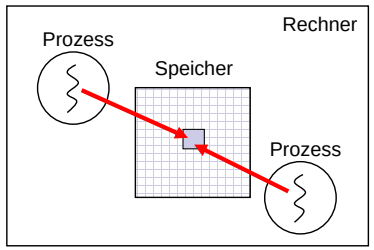
- Prozesse und Threads
- Prozess-Scheduling
- **Interprozesskommunikation**
- Verklemmungen

Interprozesskommunikation (IPC)

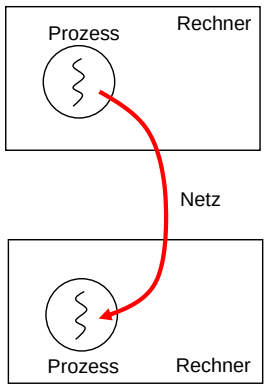
- Prozesse arbeiten oft nicht allein, sondern müssen Informationen austauschen, um eine gemeinsame Aufgabe zu erfüllen.
- Beim diesem Austausch müssen drei wichtige Fragen beantwortet werden:
  - Wie werden die Daten ausgetauscht?
    - Über gemeinsame Variablen?
    - Über Nachrichtenaustausch?
  - Wie wird sicher gestellt, dass die Prozesse nicht gleichzeitig auf gemeinsame Information zugreifen?
  - Wie wird die richtige Reihenfolge des Zugriffs sicher gestellt (Producer-Consumer-Problem)?
- Die beiden letzten Fragen beschreiben das **Synchronisationsproblem**.

Kommunikationsformen

- Gemeinsame Variablen: vor allem in Ein-Prozessor und Multiprozessor-Systemen mit gemeinsamem physikalischen Speicher



- Nachrichtenaustausch: vor allem bei verteilten Systemen, also Kommunikation über Rechnergrenzen hinweg



Erzeuger/Verbraucher-Problem

- Problemstellung**:**
  - Zwei Prozesse besitzen einen gemeinsamen Puffer mit einer festen Länge (bounded buffer). Eine Prozess schreibt Informationen in den Puffer (producer), der andere liest Informationen aus dem Puffer (consumer).
  - Der Erzeuger darf nicht in den vollen Puffer einfügen.
  - Der Verbraucher darf nicht aus dem leeren Puffer lesen.
- Eine fehlerhafte Lösung:

```
while (true) {
   produce(&item);
   while (count == N) sleep(1);
   buffer[in] := item;
   in := (in + 1) % N;
   count := count + 1;
}
```

```
while (true) {
   while (count == 0) sleep(1);
   item = buffer[out];
   out := (out + 1) % N;
   count := count - 1;
   consume(item);
}
```

Sychronisationsproblem

- Die Anweisungen
  `count := count + 1` und `count := count - 1`
  werden typischerweise zu den folgenden Maschinenbefehlen:

```
P1: register1 := count          C1: register2 := count
P2: register1 := register1+1    C2: register2 :=
                                        register2-1
P3: count := register1          C3: count := register2
```

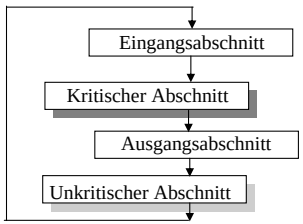
- Nehmen wir an, der Wert von `count` sei 5. Was liefert die Ausführung der Befehle in der Reihenfolge
  - (a) P_1, P_2, C_1, C_2, P_3, C_3 und die Ausführung in der Reihenfolge
  - (b) P_1, P_2, C_1, C_2, C_3, P_3?

Race Conditions

- (a) liefert den falschen Wert 4,
  (b) liefert den falschen Wert 6.
  Diese Werte sind falsch, da ja ein Element eingefügt und eines entfernt wird, der Wert müsste also bei 5 bleiben.
- Die angegebene Lösung erzeugt falsche Ergebnisse, die von der Bearbeitungsreihenfolge der Prozesse abhängen.
- Jede Situation, in der mehrere Prozesse gemeinsame Daten manipulieren, kann zu derartigen Synchronisationsproblemen (race conditions) führen. *Synchronisationsverfahren* garantieren, dass immer nur ein Prozess zu einem bestimmten Zeitpunkt gemeinsam benutzte Daten manipulieren kann.

Kritischer Abschnitt (1/2)

- Ein kritischer Abschnitt (critical section) eines Programms ist eine Menge von Instruktionen, in der das Ergebnis der Ausführung auf unvorhergesehene Weise variieren kann, wenn Variablen, die auch für andere parallel ablaufende Prozesse oder Threads zugreifbar sind, während der Ausführung verändert werden.
- Prinzipieller „Lebenszyklus“ eines Prozesses oder Threads:



Prozess wartet auf Erlaubnis

Prozess gibt bekannt, dass er den kritischen Abschnitt verlassen hat

Kritischer Abschnitt (2/2)

- Das Problem besteht darin, ein „Protokoll“ zu entwerfen, an das sich alle Prozesse oder Threads halten und das die Semantik des kritischen Abschnitts realisiert.
- Anforderungen an eine Lösung:
  - Zwei Prozesse dürfen nicht gleichzeitig in ihrem kritischen Abschnitt sein (safety).
  - Es dürfen keine Annahmen über die Bearbeitungsgeschwindigkeit von Prozessen gemacht werden.
  - Kein Prozess, der außerhalb eines kritischen Bereichs ist, darf andere Prozesse beim Eintritt in den kritischen Abschnitt behindern.
  - Kein Prozess darf ewig auf den Eintritt in den kritischen Abschnitt warten müssen (fairness).
  - Möglichst passives statt aktives Warten, da aktives Warten einerseits Rechenzeit verschwendet und andererseits Blockierungen auftreten können, wenn auf Prozesse/Threads mit niedriger Priorität gewartet werden muss.

Lösungen für die Synchronisation

- Es wurden eine Reihe von Lösungen für die Synchronisation von Prozessen entwickelt, von denen wir die wichtigsten besprechen:
  - Semaphore
  - Mutexe
  - Monitore
- Dies sind Lösungen für die Synchronisation bei Nutzung gemeinsamer Variablen. Bei Nachrichtenkommunikation wird diese Form der Synchronisation nicht benötigt.

Semaphore

- Ein Semaphor ist eine geschützte Variable, auf die nur die **unteilbaren (atomaren)** Operationen up (signal, V) und down (wait, P) ausgeführt werden können:

```
down(s)
{   s := s - 1;
    if (s < 0) queue_this_process_and_block();
}

up(s)
{   s := s + 1;
    if (s >= 0) wakeup_process_from_queue();
}
```

Eigenschaften von Semaphoren & Mutexe

- Semaphore
  - Semaphore können zählen und damit z.B. die Nutzung gemeinsamer Betriebsmittel überwachen.
  - Semaphore werden durch spezielle Systemaufrufe implementiert, die die geforderten **atomaren** Operationen up und down realisieren.
  - Semaphore können in beliebigen Programmiersprachen benutzt werden, da sie letztlich einem Systemaufruf entsprechen.
  - Semaphore realisieren ein passives Warten bis zum Eintritt in den kritischen Abschnitt.
- Mutexe
  - Oft wird die **Fähigkeit zu zählen** bei Semaphoren nicht benötigt, d.h., es genügt eine einfache binäre Aussage, ob ein kritischer Abschnitt frei ist oder nicht.
  - Dazu kann eine einfacher zu implementierende Variante, der sogenannte **Mutex** (von „mutual exclusion“), verwendet werden.

Erzeuger/Verbraucher mit Semaphoren

- Das Erzeuger/Verbraucher Problem lässt sich elegant mit drei Semaphoren lösen:
  1. Ein Semaphor zum Betreten der kritischen Abschnitte (mutex).
  2. Ein Semaphor, das die freien Plätze im Puffer herunter zählt und den Prozess blockiert, der in einen vollen Puffer schreiben will (empty).
  3. Ein Semaphor, das die belegten Plätze im Puffer herauf zählt und den Prozeß blockiert, der von einem leeren Puffer lesen will (full).

```
while (true) {
   produce(&item);
   down(&empty);
   down(&mutex);
   add(&item);
   up(&mutex);
   up(&full);
}
```

```
semaphore mutex = 1,
          empty = N,
           full = 0;
```

```
while (true) {
   down(&full);
   down(&mutex);
   remove(&item);
   up(&mutex);
   up(&empty);
   consume(item);
}
```

Implementierung von Semaphoren

- Semaphore lassen sich als Systemaufrufe implementieren, wobei kurzzeitig sämtliche Unterbrechungen unterbunden werden. (Da zur Implementation nur wenige Maschinenbefehle benötigt werden, ist diese Möglichkeit akzeptabel.)
- Auf Mehrprozessor-Systemen muss ein Semaphor mit einer unteilbaren Prozessor-Operation implementiert werden, die das Semaphor vor gleichzeitigen Änderungen in anderen Prozessoren schützt.
- Beispiel: **Test-And-Set-Lock** (TSL): beim Ausführen der Operation wird der Memory-Bus für alle anderen Operationen gesperrt

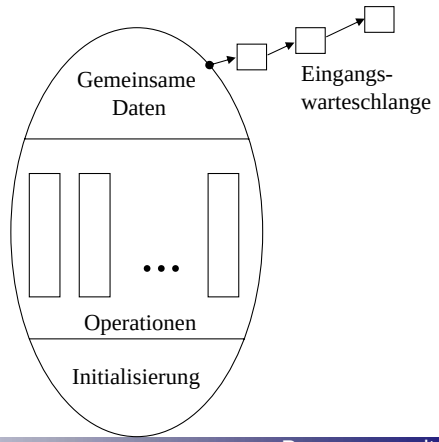
Probleme bei Semaphoren

- Programmierfehler bei der Benutzung eines Semaphors können zu Verklemmungen oder inkorrekten Ergebnissen führen. Typische Fehler:
  - Sprünge aus kritischen Bereichen, ohne das mutex-Semaphor freizugeben.
  - Sprünge in kritische Bereiche, ohne das mutex Semaphor zu setzen.
  - Vertauschungen von Semaphoren zum Schutz von kritischen Abschnitten und Semaphoren, die vorhandene Betriebsmittel zählen.
- Alles in allem sind Semaphore eine „low-level"-Lösung, die erhebliche Disziplin vom Programmierer verlangt. Eine komfortablere Lösung bieten *Monitore*.

Monitore

- Ein Monitor ist eine Sammlung von Prozeduren, Variablen und Datenstrukturen, die in einem Modul gekapselt sind.
- Prozesse können die Prozeduren des Monitors aufrufen, aber keine internen Daten ändern.
- Monitore besitzen die Eigenschaft, dass immer nur genau ein Prozess im Monitor aktiv sein kann.
- Monitore sind Konstrukte einer Programmiersprache und erfordern daher spezielle Compiler.
- Es ist die Aufgabe des Compilers, Maschinenbefehle zu generieren, die den wechselseitigen Ausschluss im Monitor garantieren.

Schematischer Aufbau eines Monitors:



Bedingungsvariable eines Monitors

- Bedingungsvariablen (condition variables) eines Monitors mit den zugehörigen Operationen `wait()` und `signal()` erlauben es, im Monitor auf andere Prozesse zu warten:
  - `wait(c)`: Der aufrufende Prozess blockiert, bis ein `signal()` auf der Bedingungsvariablen c ausgeführt wird. Ein anderer Prozess darf den Monitor betreten.
  - `signal(c)`: Ein auf die Bedingungsvariable c wartender Prozeß wird aufgeweckt. Der aktuelle Prozess muss den Monitor sofort verlassen.
- Der durch `signal()` aufgeweckte Prozess wird zum aktiven Prozess im Monitor, während der Prozess, der `signal()` ausgeführt hat, blockiert.
- Bedingungsvariablen sind keine Zähler. Ein `signal(c)` auf einer Variablen c ohne ein `wait(c)` geht einfach verloren.

Erzeuger/Verbraucher-Problem mit einem Monitor

```
monitor ProducerConsumer
  condition full, empty
  integer count;

  procedure enter
    if count = N  then wait(full);
    enter_item()
    count := count + 1;
    if count = 1  then
   signal(empty);
  end;

  procedure remove
    if count = 0 then wait(empty);
    remove_item();
    count := count - 1;
    if count = N-1 then
   signal(full);
  end;

  count := 0;
end monitor;
```

```
procedure producer
   while (true) do begin
        produce_item;
        ProducerConsumer.enter;
   end
end;
```

```
procedure consumer
   while (true) do begin
        ProducerConsumer.remove;
        consume_item;
   end
end;
```


Kritische Abschnitte mit Compilerunterstützung in Java

- In Java können kritische Abschnitte als Anweisungsfolge geschrieben werden, denen das Schlüsselwort `synchronized` voran gestellt wird.
- Die kritischen Abschnitte schließen sich bzgl. des Sperrobjekts gegenseitig aus.

```
class Buffer {
  private const int size = 8;
  private int count = 0, out = 0, in =
   0;
  private int[] pool = new int[size];

  public synchronized void insert(int i)
   {
    pool[in] = i;
    in = (in + 1) % size;    count++;
  }

  public sychronized int remove() {
    int res = pool[out];
    out = (out + 1) % size;  count--;
    return res;
  }

  public synchronized int cardinal() {
    return count;
  }
}
```


Beispiel: IPC in Windows

- Es stehen verschiedene Mechanismen zur Verfügung:
  - Pipes, Named Pipes (Kommunikation auf einem Rechner)
  - Mailslots (ähnlich Pipes, mit leicht anderen Eigenschaften)
  - Sockets (Kommunikation zwischen Rechnern)
  - Remote Procedure Calls
  - Shared Files

Synchronisation in Windows

- Verfügbare Mechanismen
  - Semaphoren
  - Mutexe
  - Kritische Sektionen
  - Ereignisse
- Synchronisation findet auf Thread-Ebene statt (andere Threads in einem Prozess sind nicht betroffen)

Prozessverwaltung

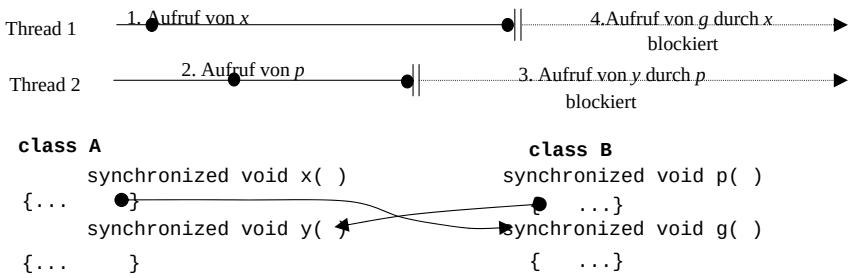
- Prozesse und Threads
- Prozess-Scheduling
- Interprozesskommunikation
- **Verklemmungen**

Verklemmungen/Deadlocks

- Prozesse benötigen Betriebsmittel, meist sogar mehrere
  - Drucker
  - Festplatte
  - Speicher (z.B. in internen BS-Tabellen)
- Was passiert wenn sich zwei Prozesse jeweils ein Betriebsmittel reservieren, das der andere auch benötigt?
  - Diese Situation wird als Deadlock bezeichnet
- Definition:
  Eine Menge von Prozessen befindet sich in einer Verklemmung (deadlock), wenn jeder Prozess der Menge auf ein Ereignis wartet, dass nur ein anderer Prozess aus der Menge auslösen kann.

Deadlock – Illustration/Beispiel

(a) Ein potenzieller deadlock. (b) Ein tatsächlicher deadlock.



Ressourcen und deren Nutzung

- Zwei Typen von Ressourcen
  - preemptable: vorzeitige Rückgabe möglich (z.B. Speicher)
  - non-preemptable: vorzeitige Rückgabe ist NICHT möglich (z.B. DVD, Drucker)
- Zuweisung, Nutzen
  - von BS oder selbst verwaltet
  - Beispiel für Selbstverwaltung: gemeinsame Variablen
  - Bei der Selbstverwaltung sind Deadlocks am häufigsten

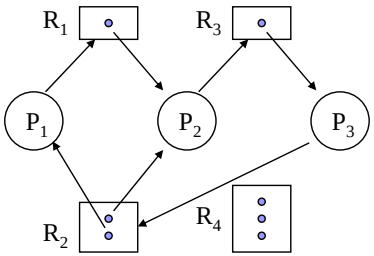
Notwendige Bedingungen für eine Verklemmung

1. Wechselseitiger Ausschluss (mutual exclusion):
   - Ein Betriebsmittel ist entweder genau einem Prozess zugeordnet oder es ist verfügbar.
2. Wartebedingung (hold and wait):
   - Es gibt einen Prozess, der ein Betriebsmittel belegt und auf ein anderes Betriebsmittel wartet, das von einem anderen Prozess belegt wird.
3. Keine Verdrängung (no preemption):
   - Einem Prozess kann ein Betriebsmittel nicht entzogen werden.
4. Zirkuläres Warten (circular wait):
   - Es gibt eine Menge {P1, P2,..., Pn} von Prozessen, so dass P1 auf ein Betriebsmittel wartet das P2 belegt, P2 wartet auf ein Betriebsmittel das P3 belegt, ..., und Pn wartet auf ein Betriebsmittel das P1 belegt.

Betriebsmittel-Zuweisungsgraph

- Ein Betriebsmittel-Zuweisungsgraph ist ein gerichteter Graph, der zur Beschreibung von Verklemmungen verwendet wird:
  - Der Graph besteht aus einer Menge von Prozessen $P=\{P_1, P_2,..., P_n\}$, einer Menge von Betriebsmitteln $R=\{R_1, R_2,..., R_m\}$ und gerichteten Kanten.
  - Eine gerichtete Kante von einem Prozess P_i zu einem Betriebsmittel R_j beschreibt, dass der Prozess P_i ein Exemplar der Betriebsmittels R_j angefordert hat (request edge).
  - Eine gerichtete Kante von einem Betriebsmittel R_j zu einem Prozess P_i beschreibt, dass ein Exemplar des Betriebsmittels R_j dem Prozess P_i zugewiesen ist (assignement edge).
- Ein Zyklus im Graph bedeutet das Vorhandensein einer Verklemmung.

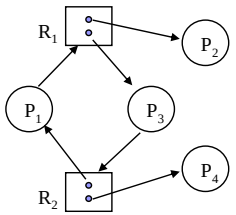
Beispiel



- P1 hat R1angefordert; R1 ist jedoch schon P2 zugeordnet.
- P2 wartet auf R3, die aber schon P3 zugewiesen ist.
- P3 wartet auf R2. R2 ist zweimal vorhanden, aber beide Instanzen sind bereits P1 bzw. P2 zugewiesen.
- ➔ Verklemmung

Eigenschaften von Zuweisungsgraphen

- Enthält ein Betriebsmittel-Zuweisungsgraph keine Zyklen, dann existiert auch keine Verklemmung.
- Besitzt ein Betriebsmittel-Zuweisungsgraph einen Zyklus und existiert von jedem beteiligten Betriebsmittel nur genau ein Exemplar, dann existiert eine Verklemmung.
- Besitzt ein Betriebsmittel-Zuweisungsgraph einen Zyklus und von den beteiligten Betriebsmitteln existieren mehrere Exemplare, so ist eine Verklemmung möglich, aber nicht unbedingt auch eingetreten.



Der Zyklus P_1, R_1, P_3, R_2, P_1 beschreibt keine Verklemmung, da Prozess P_4 beendet werden kann, womit ein Exemplar von R_2 wieder verfügbar wird.

Behandlung von Deadlocks

- Dem Deadlock-Problem kann mit einer von vier Strategien begegnet werden:

1. Ignorieren: meist keine gute Idee ...
2. Deadlock-Entdeckung und –Auflösung: lasse Deadlocks passieren und behandle sie dann
3. Deadlock-Vermeidung durch vorsichtige dynamische Ressourcen-Allokation
4. Strukturelle Deadlock-Verhinderung durch das Negieren einer der vier notwendigen Bedingungen

Verhinderung von Verklemmungen (1/4)

- Wird eine der vier notwendigen Bedingungen für das Entstehen von Verklemmungen negiert, so können keine Verklemmungen mehr entstehen.

1. **Verhinderung von wechselseitigem Ausschluss:**
   - Auf alle Betriebsmittel kann von mehreren Prozessen aus gleichzeitig zugegriffen werden.
   - Beispiel: Drucker
     Anstatt einen Drucker Prozessen exklusiv zuzuordnen, werden Druckausgaben in Dateien abgelegt und in eine Warteschlange eingefügt. Ein spezieller Prozess (printer daemon) erhält exklusiven Zugriff auf den Drucker und arbeitet die Warteschlange ab. Dieser Prozess fordert selbst keine weiteren Betriebsmittel an.
   - Problem:
     Nicht alle Betriebsmittel (z.B. Speicher, Einträge in der Prozesstabelle) können zwischen Prozessen geteilt werden. Im Beispiel kann der Plattenplatz zur Ablage von Druckausgaben selbst als Betriebsmittel betrachtet werden, das Auslöser für eine Verklemmung ist.

Verhinderung von Verklemmungen (2/4)

2. **Verhinderung der Wartebedingung:**
   - Jeder Prozess darf nur dann Betriebsmittel anfordern, wenn er selbst keine anderen Betriebsmittel belegt.
   - Alternative 1:
     - Jeder Prozess fordert sämtliche Betriebsmittel an, bevor seine Bearbeitung beginnt.
     - Problem: Geringe Effizienz, da alle Betriebsmittel eines Prozesses während seiner Ausführung belegt sind. Die meisten Prozesse kennen ihre Betriebsmittelanforderungen nicht, bevor sie starten.
   - Alternative 2:
     - Ein Prozess muss alle Betriebsmittel abgeben, bevor er weitere anfordern kann.
     - Problem: Es ist einem Prozess nicht immer möglich, alle Betriebsmittel freizugeben, bevor er weitere Betriebsmittel belegt (z.B. der Eintrag in der Prozesstabelle). Außerdem kann diese Methode dazu führen, dass die Bearbeitung eines Prozesses unendlich verzögert wird, da ein benötigtes Betriebsmittel immer von einem anderen Prozess belegt ist (starvatíon).

Verhinderung von Verklemmungen (3/4)

3. **Entzug von zugewiesenen Betriebsmitteln:**
   - Bereits belegte Betriebsmittel können einem Prozess entzogen werden.
   - Beispiel:
     - Beispielsweise können einem Prozess, der bereits auf ein Betriebsmittel wartet, seine bereits belegten Betriebsmittel entzogen werden, um Betriebsmittelanforderungen anderer Prozesse befriedigen zu können. Die entzogenen Betriebsmittel werden zu der Liste der Betriebsmittel hinzugefügt, auf die der Prozess wartet.
   - Probleme:
     - Normalerweise besitzen die zugewiesenen Betriebsmittel einen Zustand, der beim Entzug gesichert und später wieder restauriert werden muss. Nicht jedes Betriebsmittel (z.B. Drucker) erlaubt es, den Zustand zu sichern und später wieder zu restaurieren.
   - Verwendung:
     - Das Verfahren findet häufig Anwendung bei Betriebsmitteln, deren Zustand leicht gesichert werden kann, wie z.B. CPU-Registerinhalte oder Hauptspeicherbereiche.

Verhinderung von Verklemmungen (4/4)

4. **Verhinderung von zirkulärem Warten:**
   - Es wird eine totale Ordnung auf den vorhandenen Betriebsmitteln definiert.
   - Ein neues Betriebsmittel darf nur dann angefordert werden, wenn das Betriebsmittel bezüglich der Ordnung größer ist als alle bereits belegten Betriebsmittel eines Prozesses.
   - Mehrere Exemplare eines Betriebsmittels müssen in einer Anfrage angefordert werden.
   - Probleme:
     - Es ist schwierig, eine totale Ordnung für Betriebsmittel zu definieren, die allen Einsatzmöglichkeiten eines Betriebssystems gerecht wird.
     - Benötigt ein Prozess ein Betriebsmittel mit geringerer Ordnung, so müssen zunächst sämtliche Betriebsmittel freigegeben werden.

Vermeidung von Verklemmungen

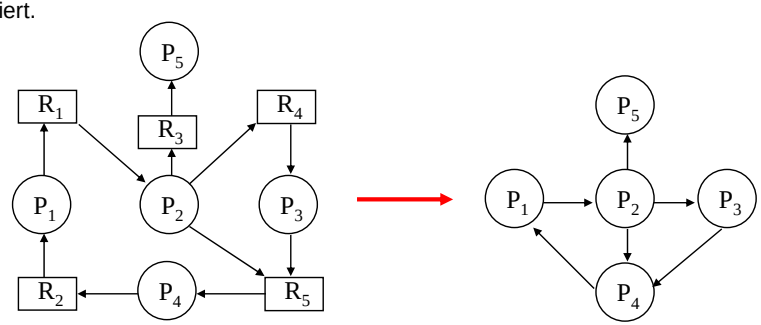
- Ansatz:
  - Anhand von zusätzlichen Informationen werden nur die Betriebsmittelanforderungen gewährt, die nicht zu einer Verklemmung führen können.
  - Im Vergleich zur Verhinderung werden Betriebsmittel effizienter genutzt und der Systemdurchsatz erhöht.
  - Annahme: Die maximale Anforderung von Betriebsmitteln ist a-priori bekannt.
- Sichere Zustände:
  - Ein System ist in einem sicheren Zustand, wenn es jedem Prozess seine maximale Betriebsmittelanforderung gewähren kann, ohne dass das System in eine Verklemmung geraten kann.
- Unsichere Zustände:
  - Es existiert die Möglichkeit einer Verklemmung.

Deadlock-Entdeckung und Beseitigung

- Erkennungsverfahren (detection algorithm):
  - Das Betriebssystem überprüft periodisch, ob eine Verklemmung vorliegt.
- Behebungsverfahren (recovery algorithm):
  - Im Fall einer Verklemmung ergreift das Betriebssystem Maßnahmen, um die Verklemmung aufzulösen.
- Notwendige Randbedingungen:
  - Die notwendigen Informationen über belegte und angeforderte Betriebsmittel muss einfach zugänglich sein.
  - Der Aufwand zur Entdeckung von Verklemmungen muss vertretbar sein. (Man beachte, dass während der Ausführung des Erkennungsverfahrens keine Betriebsmittel angefordert oder freigegeben werden dürfen.)
  - Die Kosten zur Behebung von Verklemmungen müssen vertretbar sein.

Erkennung von Verklemmungen (1/2)

- Es existiert nicht mehr als ein Exemplar von jedem Betriebsmittel:
  - Aus dem Betriebsmittel-Zuweisungsgraphen wird ein Wartegraph (wait-for graph) konstruiert, wobei alle Knoten, die Betriebsmittel repräsentieren, entfernt werden.
  - Eine Kante vom Prozess P_i zum Prozess P_j existiert im Wartegraph genau dann, wenn der Betriebsmittel-Zuweisungsgraph zwei Kanten $P_i R_q$ und $R_q P_j$ besitzt.
  - Eine Verklemmung existiert genau dann, wenn ein Zyklus im Wartegraph existiert.



Erkennung von Verklemmungen (2/2)

- Es existieren mehrere Exemplare von jedem Betriebsmittel:
  - Verklemmungen lassen sich erkennen, indem man versucht, eine Abarbeitungsfolge der Prozesse P_1,..., P_n zu finden, so dass alle bekannten Betriebsmittelanforderungen erfüllt werden können.
  - Dafür kann z.B. der Bankers-Algorithmus verwendet werden.

Beseitigung von Verklemmungen

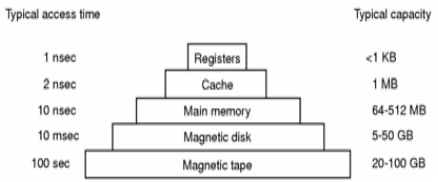
- Zwangsweise Beendigung von Prozessen (process termination):
  - Einfach zu implementieren und sehr effektiv.
  - Auswahl der zu beendenden Prozesse schwierig.
  - Zwangsweise Beendigung von Prozessen kann Inkonsistenzen erzeugen.
  - Bereits verbrauchte Rechenzeit ist normalerweise verloren.
- Zwangsweiser Entzug von Betriebsmitteln (resource preemption):
  - Betriebsmittel werden zwangsweise einem Prozess entzogen, um damit die Verklemmung aufzuheben.
  - Auswahl des Prozesses, dem Betriebsmittel entzogen werden, ist schwierig.
  - Nicht jedes Betriebsmittel (z.B. Prozesstabelleneintrag) kann entzogen werden.
- Rücksetzen von Prozessen (rollback of processes):
  - In speziellen Fällen können Prozesse auf einen vorher gesicherten Zustand (checkpoint) zurückgesetzt werden.
  - Insbesondere Datenbank-Prozesse verwalten häufig ein Logbuch über durchgeführte Transaktionen, so dass ein Prozess ohne Datenverlust gerade soweit zurückgesetzt werden kann, wie zur Behebung der Verklemmung notwendig.

Speicherverwaltung

- Einführung
- Swapping
- Virtual Memory
- Seitenersetzungsstrategien
- Segmentierung

Einführung

- Verschiedene Arten von Speichern, hierarchisch organisiert:
  - Cache-Speicher
  - Hauptspeicher
  - Sekundärspeicher
  - Archiv-Speicher



- Eigenschaften des Hauptspeichers:
  - Der Hauptspeicher besteht aus einer Menge von Wörtern oder Bytes, die jeweils über eine eigene Adresse verfügen.
  - Sowohl die CPU als auch E/A-Geräte greifen auf den Hauptspeicher zu.
  - Ausführbare Programme befinden sich zumindest teilweise im Hauptspeicher.
  - Die CPU kann normalerweise nur auf Daten im Hauptspeicher direkt zugreifen

Speicher und dessen Verwaltung

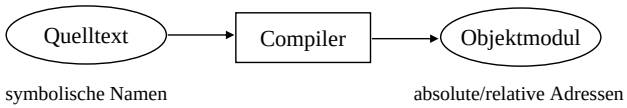
- Betriebssystemteil verantwortlich für Speichermanagement: *Memory Manager*
- Aufgaben
  - Buchhaltung: welche Speicherbereiche werden benutzt, welche sind frei
  - Speichervergabe und -rücknahme an/von Prozessen
  - Datenverschiebung zwischen den Speicherhierarchien

Grundlegende Speicherverwaltung

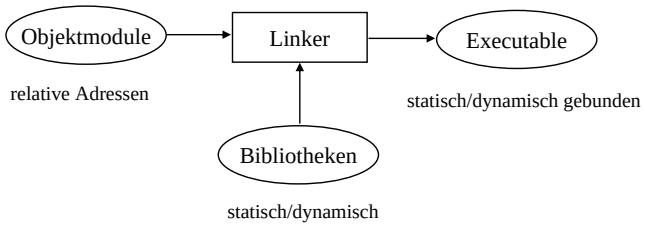
- Speichermanagement: grundsätzlich zwei Klassen:
  - Verschieben von Prozessen vom Hauptspeicher auf Festplatte und zurück (Swapping, Paging)
  - Oder nicht (einfache Variante)
- Bei Verzicht auf Swapping und Paging:
  - Monoprogramming:
    - immer nur ein Prozess sowie das OS im Speicher
    - Einsatzgebiet: frühe Batchsysteme, eingebettete Systeme
  - Multiprogramming mit festen Partitionen
    - Speicher wird in feste Blöcke eingeteilt, Programme bekommen einen Speicherbereich zugewiesen
    - Einsatz im OS/360 von IBM
    - Oft wurde die Größe der Partitionen einmal am Tag festgesetzt

Abbildungen von Speicheradressen

- Ein Compiler/Assembler übersetzt symbolische Adressen (Namen von Variablen und Funktionen) in absolute oder relative Speicheradressen.

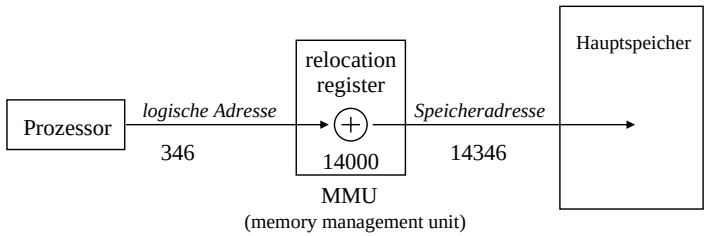


- Ein Linker bindet mehrere Objektmodule mit relativen Adressen und die benötigten Bibliotheken zu einem ausführbaren Programm (executable).



Präzisierung der Aufgaben

- Bereitstellung und Zuweisung des Speicherplatzes an die Prozesse (allocation).
- Einrichtung und Koordination von Speicherbereichen, die von mehreren Prozessen gemeinsam benutzt werden können (shared memory).
- Schutz der Informationen im Hauptspeicher vor fehlerhaften oder unbefugten Zugriffen.
- Abbildung von logischen Adressen auf physikalische Adressen (address translation).



Moderne Speicherverwaltung

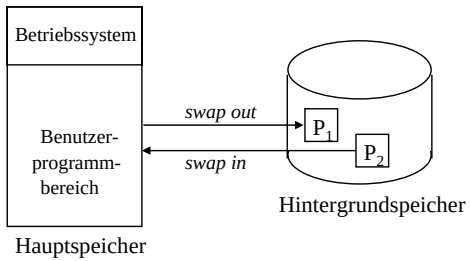
- Heutige Computer haben andere Anforderungen an die Verwaltung des Speichers
- Insbesondere
  - Laufen meist viele Prozesse
  - Haben die Prozesse oft mehr Speicherbedarf als physikalischer Speicher vorhanden ist
- Zwei wichtige Lösungen:
  - **Swapping:** Verschieben von Prozessen zwischen Hauptspeicher und Platte
  - **Virtual Memory**: Prozesse sind nur zum Teil im Hauptspeicher

Speicherverwaltung

- Einführung
- **Swapping**
- Virtual Memory
- Seitenersetzungsstrategien
- Segmentierung

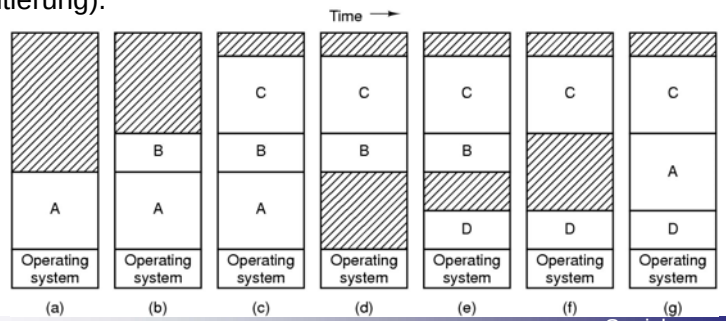
Prinzip des Swapping

- Der komplette Adressraum eines Prozesses wird beim Prozesswechsel auf den Hintergrundspeicher ausgelagert und ein anderer Adressraum eingelagert.
- Ist im Betriebssystem ohne weitere Hardware-Unterstützung zu realisieren.
- Extrem aufwändige Prozesswechsel, da die Zugriffszeiten auf den Hintergrundspeicher im allgemeinen um Größenordnungen langsamer sind als Zugriffe auf den Hauptspeicher.
- Wurde von MS Windows 3.* benutzt, um Prozesse auszulagern.
- Wird bei UNIX-Systemen benutzt, um bei einer Überbelegung des Hauptspeichers einigen Prozessen das Betriebsmittel Hauptspeicher zwangsweise zu entziehen.



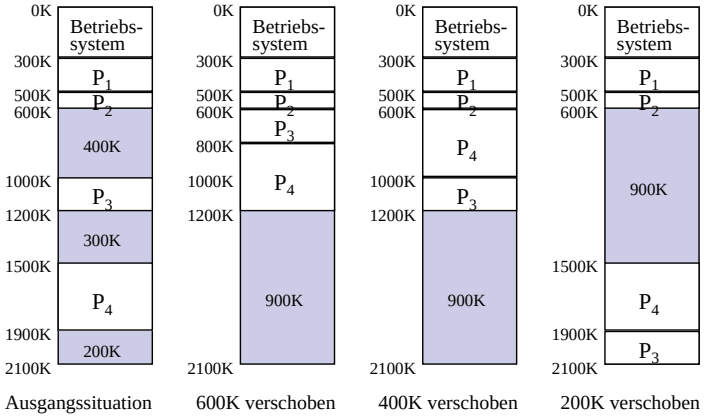
Speichersegmente und Fragmentierung

- Der Hauptspeicher wird vom Betriebssystem in Segmente variabler Länge eingeteilt, die den Prozessen zugewiesen werden. Zur Verwaltung dienen Segmenttabellen.
- Segmente können verschiedene Zugriffsrechte besitzen, zwischen Prozessen geteilt werden, oder bei Bedarf wachsen.
- Durch das Entfernen und Einfügen von Segmenten entstehen langristig kleine unbenutzte Speicherbereiche (externe Fragmentierung).
- Beispiel:



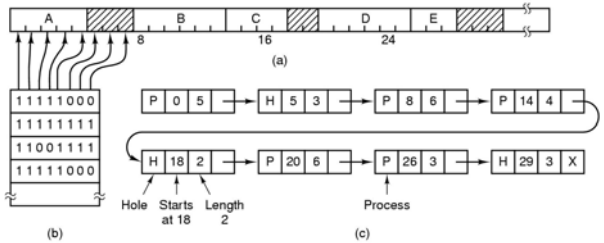
Kompaktifizierung

- Durch das Verschieben von Segmenten im Speicher können viele kleine Löcher zu einem großen Stück freien Speichers zusammengefügt werden (Kompaktifizierung, compaction).
- Kompaktifizierung setzt voraus, dass die Adressen dynamisch abgebildet werden.
- Die Suche nach einer optimalen Kompaktifizierungsstrategie ist schwierig.
- Kompaktif. benötigt viel CPU-Zeit.



Speicherbuchhaltung

- Wie merkt sich der Memory Manager, welche Speicherbereiche frei bzw. belegt sind?
- Zwei Lösungen: Bitmaps oder verkettete Listen
- Beispiel:
- Jedes Bit identifiziert einen kleinen Speicherbereich, daher feste Größe.
- Listen vereinfachen die dynamische Verwaltung.



- Wichtige Frage: Wie finde ich das passende Segment, wenn ein neuer Prozess Speicher anfordert?

Positionierungsstrategien

- **best fit:** Auswahl des kleinsten Loches, das das Segment aufnehmen kann. Diese Strategie lässt einerseits große Löcher lange bestehen, während sie andererseits eine Vielzahl kleiner und nutzloser Überreste erzeugt.
- **worst fit:** Auswahl des jeweils größten Loches. Dieses Verfahren tendiert dazu, alle Löcher auf etwa die gleiche Länge zu bringen, die dann aber eventuell zu klein zur Aufnahme eines bestimmten Segments sein kann.
- **first fit:** Auswahl des ersten hinreichend großen Loches. Dieses Verfahren liegt in seinem Verhalten zwischen den beiden anderen und ist sehr effizient.
- **next fit:** Dieses Verfahren ist eine Variation von first fit. Um zu verhindern, dass sich Löcher einer bestimmten Größe an einer Stelle des Speichers häufen, beginnt jede Suche am Ende der vorherigen Suche. Der Speicher wird also ringförmig durchsucht.
- **buddy system:** Die Löcher werden in *k* Listen so einsortiert, dass die *i*-te Liste jeweils Löcher der Länge größer gleich 2^i für $i=1,...,k$ enthält. Dabei können zwei benachbarte Löcher der *i*-ten Liste effizient zu einem Loch der *i+1*-ten Liste zusammengefügt werden. Umgekehrt kann ein Loch der *i*-ten Liste einfach in zwei Löcher der *i-1*-ten Liste aufgeteilt werden.

Speicherverwaltung

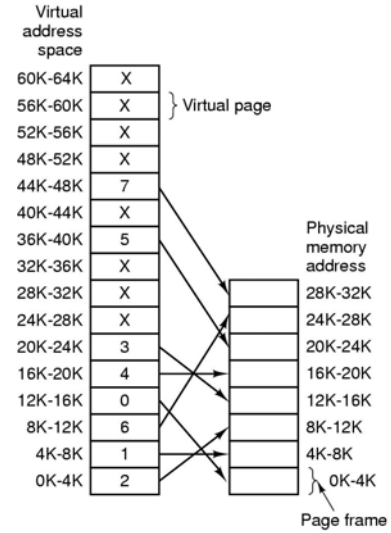
- Einführung
- Swapping
- **Virtual Memory**
- Seitenersetzungsstrategien
- Segmentierung

Virtual Memory

- Idee (Fotheringham, 1961): ist ein Programm größer als der zur Verfügung stehende Hauptspeicher, dann halte immer nur die aktuell notwendigen Teile im Speicher; lade andere Teile bei Bedarf nach
- Wichtige Fragen:
  - Welche Teile sind notwendig? Welche Teile behalte ich tunlichst im Speicher? → Lade- und Ersetzungsstrategien
- *Zweistufiges* Adressensystem: *virtuelle* Adressen, die die Programme benutzen, werden von der *Memory Management Unit* in *physikalische* Adressen umgewandelt und dann erst an den Speicher gegeben
- Wichtigste Technik: Paging

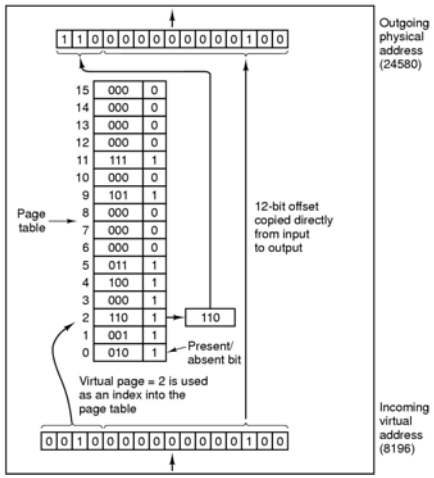
Paging

- Der physikalische Speicher wird in Kacheln (frames) fester Größe eingeteilt.
- Der logische Adressraum wird in Seiten (pages) gleicher Größe eingeteilt.
- Logische Adressen werden in eine Seitennummer (page number) und eine seitenrelative Adresse (page offset) aufgeteilt.
- Eine Umsetzungstabelle (page table) bildet die Seiten auf die verfügbaren Kacheln ab.
- Die Seiten eines Adressraums können beliebig auf die verfügbaren Kacheln verteilt sein.



Page Tables

- In der Seitentabelle gibt der Index die Nummer der virtuellen Seite an; der Tabelleneintrag gibt dann die zugehörige physikalische Seite an.
- Beispiel: 16 mögliche virtuelle Seiten werden auf 8 vorhandene physikalische Kacheln abgebildet.
- Ein spezielles Bit gibt an, ob die virtuelle Seite vorhanden ist.



Eigenschaften des Paging

- Keine externe Fragmentierung. Allerdings wird die letzte Seite eines logischen Adressraums normalerweise nicht vollständig genutzt (interne Fragmentierung).
- Gemeinsamer Speicher lässt sich mit Hilfe von gemeinsamen Seiten realisieren.
- Speicherschutz wird durch Schutzbits realisiert. Weitere Bits geben an, ob eine Kachel gültig ist (valid) oder ob eine Kachel modifiziert wurde (modified).
- Es muss nicht der gesamte Adressraum eines Prozesses im Speicher sein, um ihn auszuführen. Zugriffe auf gerade nicht geladene Seiten lösen einen Seitenfehler (page fault) aus.
- Das Betriebssystem behandelt Seitenfehler, indem es die benötigte Seite in eine freie Kachel einlagert und den Befehl, der den Seitenfehler auslöste, neu startet.
- Realisierungsprobleme:
  - Die Adressabbildung muss sehr schnell sein.
    (In manchen Fällen sind mehrere Umsetzungen für einen Maschinenbefehl notwendig.)
  - Die Umsetzungstabelle kann extrem groß werden.
    (32-Bit Adressen erfordern bei einer Seitengröße von 4096 Bytes eine Tabelle mit ca. 1 Million Einträgen.)

Behandlung von Seitenfehlern

1. Die Hardware erkennt den Seitenfehler und erzeugt eine Unterbrechung.
2. Das Betriebssystem sichert die Register des Prozesses.
3. Die Prozedur zur Behandlung von Seitenfehlern ermittelt die Adresse der fehlenden Seite.
4. Das Betriebssystem prüft, ob auf die Adresse überhaupt zugegriffen werden darf.
5. Das Betriebssystem wählt eine freie Kachel aus. Falls keine freie Kachel existiert, wird eine belegte Kachel ausgewählt.
6. Falls die ausgewählte Kachel belegt ist und modifiziert wurde, wird sie auf dem Hintergrundspeicher gesichert. Ein Prozesswechsel findet statt, sofern rechenbereite Prozesse existieren.
7. Sobald eine freie Kachel da ist, wird eine E/A-Operation gestartet, um die benötigte Kachel vom Hintergrundspeicher zu laden. Ein Prozesswechsel findet statt, sofern rechenbereite Prozesse existieren.
8. Sobald die Kachel geladen ist, wird die Umsetzungstabelle aktualisiert.
9. Der Befehlszähler wird auf den Befehl zurückgesetzt, der den Seitenfehler auslöste.
10. Der Prozess wird in den Zustand *ready* gesetzt und in die CPU-Warteschlange eingereiht.

Speicherverwaltung

- Einführung
- Swapping
- Virtual Memory
- **Seitenersetzungsstrategien**
- Segmentierung

Ladestrategien

- Die Ladestrategie bestimmt, wann Seiten in den Hauptspeicher geladen werden:
  - Swapping:
    Übertragung eines ganzen Adressraums mit einem einzigen Zugriff auf den Hintergrundspeicher.
  - Demand-Paging:
    Die benötigten Seiten werden genau dann in den Speicher geladen, wenn auf sie zugegriffen wird.
  - Pre-Paging:
    Es werden Seiten geladen, auf die in der Zukunft ein Zugriff erwartet wird. Erfordert Kenntnisse über typische Zugriffsmuster.
  - Page-Clustering:
    Gemeinsame Übertragung von mehreren zusammengehörigen Seiten. Ermöglicht die Nutzung großer Seiten auf Hardware, die nur geringe Seitengrößen unterstützt.
- In der Praxis dominiert Demand-Paging, obwohl dies bei Transportkosten, die nicht monoton mit der Anzahl der transportierten Seiten wachsen, nicht unbedingt optimal ist.

Ersetzungsstrategien

- Die Ersetzungsstrategie (replacement algorithm) bestimmt, welche der belegten Kacheln ausgelagert werden, damit wieder freie Kacheln zur Einlagerung von benötigten Seiten vorhanden sind.
- Lokale Strategien weisen jedem Prozess eine konstante Anzahl von Seiten zu. Seitenfehler wirken sich daher nur auf den verursachenden Prozess negativ aus.
- Bei globalen Strategien wird der gesamte Speicher dynamisch auf alle Prozesse verteilt, um eine effiziente Nutzung des Hauptspeichers zu erreichen.

Lokalitätsprinzip

- Lokalität bezeichnet das Verhalten eines Programms, innerhalb einer bestimmten Zeit seine Speicherzugriffe auf einen kleinen Teil seines Adressraums zu beschränken.
- Das Lokalitätsprinzip gilt für Seitenzugriffsverhalten mit folgenden Eigenschaften:
  - Zu jeder Zeit verteilt ein Programm seine Speicherzugriffe in nicht gleichförmiger Weise über seine Seiten.
  - Die Korrelation zwischen den Zugriffsmustern für unmittelbare Vergangenheit und unmittelbare Zukunft ist im Mittel hoch, und die Korrelation zwischen sich nicht überlappenden Referenzstrings geht mit wachsendem Abstand zwischen ihnen gegen 0.
  - Die Referenzdichten der einzelnen Seiten, d.h. die Wahrscheinlichkeit mit der eine Seite zum Zeitpunkt *t* zum Referenzstring gehört, ändern sich nur langsam, d.h. sie sind quasi stationär.
- Praktische Gründe für die Gültigkeit des Lokalitätsprinzips sind Schleifen und Modulbereiche von Programmen. Allerdings ist Lokalität bei manchen Programmen oder Algorithmen nicht unbedingt gegeben (z.B. Datenbanken oder einige Verfahren zur Manipulation sehr großer Matrizen).

Ersetzungsstrategien

- Beladys Optimalalgorithmus (BO)
  - Es wird die Seite ersetzt, auf die in der Zukunft am längsten nicht zugegriffen wird (Realisierbarkeit??).
- Least Recently Used (LRU)
  - Es wird die Seite ersetzt, auf die am längsten nicht mehr zugegriffen wurde.
- Least Frequently Used (LFU)
  - Es wird die Seite ersetzt, auf die am wenigsten zugegriffen wurde.
- First In First Out (FIFO)
  - Es wird die Seite ersetzt, die bereits am längsten im Speicher steht.
- Second Chance (SC)
  - Es wird wie beim FIFO ersetzt. Allerdings werden Seiten übersprungen, auf die seit dem letzten Seitenfehler zugegriffen wurde, sofern es Seiten gibt, auf die nicht zugegriffen wurde.

☑ Zusätzlich wird oftmals betrachtet, ob die Seite modifiziert wurde oder nicht, da eine unmodifizierte Seite nicht auf dem Hintergrundspeicher gesichert werden muss.

Beladys Anomalie

- Intuitiv erwartet man, dass sich Seitenfehler reduzieren, wenn man den verfügbaren Hauptspeicher vergrößert.
- ABER: folgendes Beispiel:
  Referenzstring *w = 1 2 3 4 1 2 5 1 2 3 4 5* und FIFO Ersetzungsstrategie:

Jüngste Seite	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
Älteste Seite			1	2	3	4	1	1	1	2	5	5

Speichergröße m=3
(9 Seitenfehler)

1	2	3	4	4	4	5	1	2	3	4	5	
	1	2	3	3	3	4	5	1	2	3	4	
		1	2	2	2	3	4	5	1	2	3	
			1	1	1	2	3	4	5	1	2	

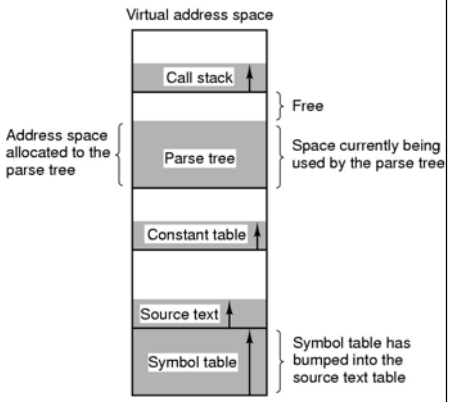
Speichergröße m=4
(10 Seitenfehler)

Speicherverwaltung

- Einführung
- Swapping
- Virtual Memory
- Seitenersetzungsstrategien
- **Segmentierung**

Segmentierung

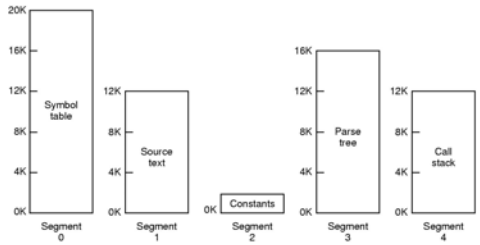
- Bisher: eindimensionaler Adressraum, in dem alle Tabellen, Programmtext etc. in ein- und demselben Adressraum abgelegt werden.
- Problem: was passiert, wenn die Tabellen nach dem Anlegen und der fortschreitenden Übersetzung eines Programms zu groß werden und in einen anderen Bereich „hineinwachsen?



- Alle „Lösungen“ mit einem Adressraum erfordern Modifikationen am Compiler, die man vermeiden möchte.

Segmentierung

- Andere Idee: stelle jedem Programm mehrere virtuelle Adressräume zur Verfügung, die es für die jeweiligen Daten verwenden kann → Segmentierung des virtuellen Adressraums



- Eine Adresse besteht dann aus Segment-Nummer und Adresse.
- Vorteile der Segmentierung
  - Segmente können wachsen, Management durch das BS
  - Linken wird wegen der jeweiligen Startadresse 0 stark vereinfacht
  - Shared Libraries können leicht realisiert werden (Schutz!)

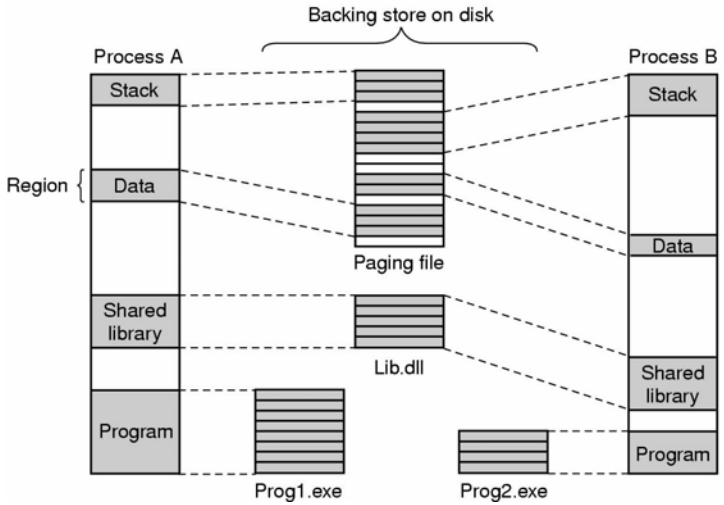
Vergleich Paging - Segmentierung

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Kombination von Segmentierung und Paging

- Segmentierung und Paging kann gleichzeitig eingesetzt werden, um die Vorteile beider Ansätze zu kombinieren.
- Zum Beispiel wird es sinnvoll sein, Segmente aufzuteilen, wenn sie nicht komplett in den Hauptspeicher passen.
- Moderne Prozessorfamilien unterstützen meist beide Modelle, um für beliebige Betriebssysteme offen zu sein.
- Ein Pentium besitzt insgesamt 16 K Segmente, wobei jedes bis zu 10^9 32-Bit-Worte speichern kann. Diese Segmentgröße ist relativ groß.

Shared Libraries



Speicherverwaltung in Windows

- Windows besitzt ein sehr anspruchsvolles virtuelles Speichersystem.
- Jeder Benutzerprozess hat 4 GB virtuellen Speicher verfügbar (32-Bit-Adressen).
- Seitengröße für Paging: 4 KB
- Shared files (shared libraries) können verwendet werden