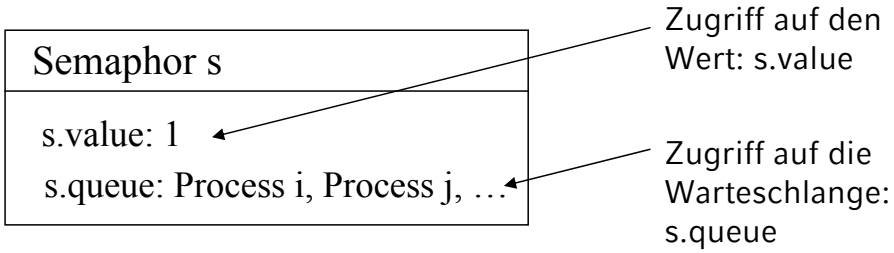
2.3 Prozessverwaltung

- Realisierung eines Semaphors:
  - Einem Semaphor liegt genau genommen die Datenstruktur „Tupel“ zugrunde
  - Speziell speichert ein Semaphor zwei Informationen:
    - Der Wert des Semaphors (0 oder 1 bei einem binären Semaphor bzw. ein **Int**-Wert bei einem Zählsemaphor)
    - Eine Liste von blockierten Prozessen (als FIFO-Warteschlange)
  - Eine FIFO (First-In-First-Out) Warteschlange ermöglicht
    - Ein Objekt (hier: Prozess) an das Ende der Liste anzufügen
    - Das erste Objekt aus der Liste zu entnehmen
    - Damit kann immer nur das Objekt entnommen werden, das am längsten in der Liste „wartet“
  - Schematisch



2.3 Prozessverwaltung

Binäres Semaphor

```
Variable  s: binarysemaphore;
```

Initialisierung

```
algorithmus init
  input s: binarysemaphore, initialvalue: {0,1}
  begin
        s.value = initialvalue;
  end
```

Wait

```
algorithmus wait
  input s: binarysemaphore
  begin
        if s.value = 1
        then s.value := 0;
        else
           blockiere Prozess und plaziere ihn in s.queue;
  end
```

Signal

```
algorithmus signal
  input s: binarysemaphore
  begin
        if s.queue ist leer
        then s.value := 1;
        else
           entnehme einen Prozess aus s.queue;
  end
```

2.3 Prozessverwaltung

Zählsemaphor

```
Variable  s: semaphore;
```

Initialisierung

```
algorithmus init
  input s: semaphore, initialvalue: Int
  begin
        s.value = initialvalue;
  end
```

Wait

```
algorithmus wait
  input s: semaphore
  begin
        s.value := s.value - 1;
        if s.value < 0
        then
           blockiere Prozess und plaziere ihn in s.queue;
  end
```

Signal

```
algorithmus signal
  input s: semaphore
  begin
        s.value = s.value + 1;
        if s.value <= 0   // gibt es noch blockierte Prozesse?
        then
           entnehme einen Prozess aus s.queue;
  end
```


2.3 Prozessverwaltung

- Programmiertechnische Nutzung von Semaphoren

```
…
variables s: binarysemaphore;
begin
        …
        init(s,1)
        …
        wait(s);
        < kritischer Bereich >;
        signal(s);
        …
end
```

- Einsatz:
  - Realisierung eines wechselseitigen Ausschlusses
    ⇒ binäres Semaphor
  - Verwaltung einer begrenzter Anzahl von Ressourcen
    ⇒ Zählsemaphor

2.3 Prozessverwaltung

– Beispiel: Flugbuchung

```
algorithmus Reservierung
  variables n, anzpl : Nat; s : binarysemaphore
  Global: init(s,1);
  begin
        …
        while Buchung nicht abgeschlossen do {
              n = Anzahl der zu buchenden Plätze;
              anzpl = aktuelle Anzahl der freien Plätze;
              wait(s);
              if anzpl ≥ n
              then anzpl um n verringern;
              else STOP mit Auskunft „Ausgebucht“;
              Reservierung bestätigen;
              signal(s);
              Frage nach weiterer Buchung;        // Nein: Verlassen der Schleife
        }
        …
  end
```


DATABASE SYSTEMS GROUP
Kapitel 2: Betriebssysteme
LMU

2.3 Prozessverwaltung

- Deadlocks (siehe Seite 18):

  Ein Deadlock ist eine dauerhafte Blockierung einer Menge M von Prozessen, die eine Menge S gemeinsamer Systemressourcen nutzen oder miteinander kommunizieren ($|M| > 1$, $|S| > 1$)

  – Lösungsstrategien:
    - Vermeidung
    - Erkennung

2.3 Prozessverwaltung

- Vermeidung von Deadlocks
  - keine gleichzeitige Beanspruchung mehrerer Betriebsmittel durch einen Prozess

    … get A … release A … get B … release B …

    Das ist aber nicht immer möglich !!!
  - wenn ein Prozess mehrere Betriebsmittel gleichzeitig benötigt, muss er diese auf einmal belegen; die Freigabe kann nach und nach erfolgen

    … get A, B, C … release B … release A, C …

    Problem:

    Alle benötigten Betriebsmittel müssen vorab bekannt sein

    Ggf. werden dadurch zu viele Ressourcen belegt, nur weil die Möglichkeit besteht, dass sie benötigt werden könnten

2.3 Prozessverwaltung

- Erkennung von Deadlocks
  - notwendige Voraussetzungen für Deadlock (trifft eine nicht zu, kann kein Deadlock entstehen):
    - ***Mutual Exclusion***

      Es gibt mind. 2 Ressourcen, die nur von einem Prozess gleichzeitig benutzt werden
    - ***Hold and Wait***

      Ein Prozess muss eine Ressource behalten, während er auf eine weitere Ressource wartet
    - ***No Preemption***

      Eine Ressource kann einem Prozess, der sie behält, nicht wieder entzogen werden

2.3 Prozessverwaltung

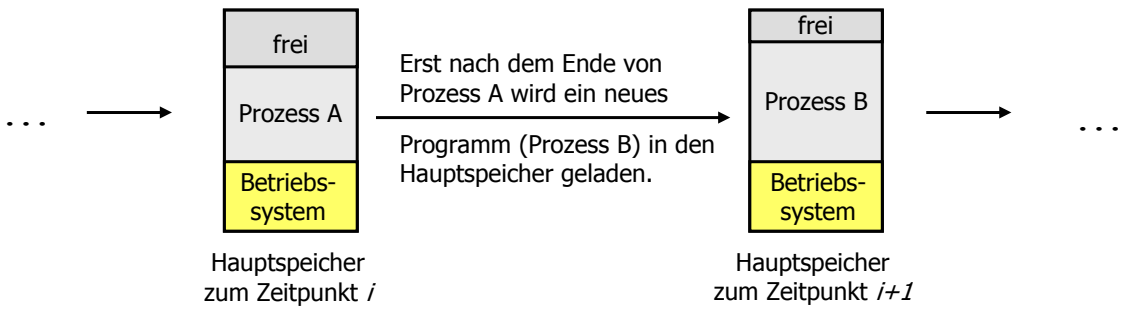
- Sind alle drei Bedingungen erfüllt, muss noch eine vierte Bedingung zutreffen, damit ein Deadlock eintritt:
  - ***Circular Wait***

    Es existiert eine geschlossene Kette von Prozessen, so dass jeder Prozess mindestens eine Ressource hält, die von einem anderen Prozess der Kette gebraucht wird

- Was kann dann getan werden?
  - Es muss eine der drei Bedingungen auf der vorigen Folie verletzt werden, z.B. *No Preemption*: einem Prozess wird eine Ressource, die dieser gerade hält, wieder entzogen

2.4 Speicherverwaltung

- Motivation:
  - beim Multiprogramming muss der Hauptspeicher mehreren Prozessen gleichzeitig zur Verfügung gestellt werden
  - Aufgaben der Hauptspeicherverwaltung
    - ***Zuteilung*** (***allocation***) von ausreichend Speicher an einen ausführenden Prozess
    - ***Schutz*** (***protection***) vor Zugriffen auf Hauptspeicherbereiche, die dem entsprechenden Prozess nicht zugewiesen sind
  - einfachster Fall: Uniprogramming



2.4 Speicherverwaltung

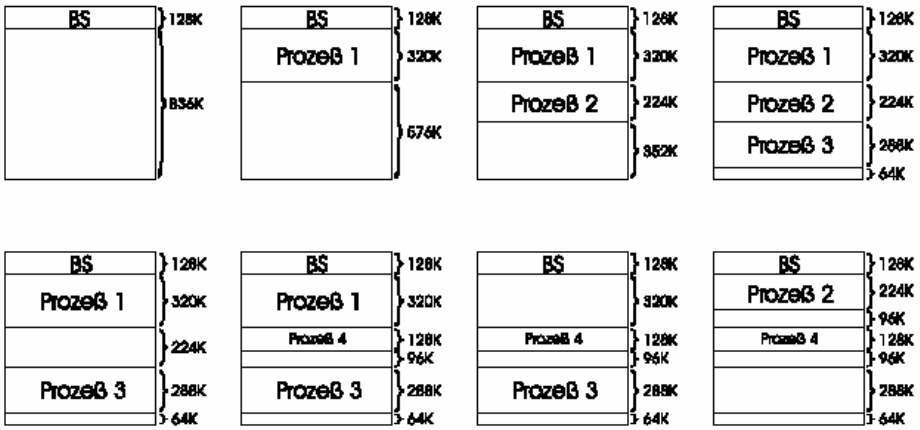
- Für das Multiprogramming ist eine Unterteilung (***Partitionierung***) des Speichers für mehrere Prozesse erforderlich
  - ***Feste Partitionierung*** in gleich große Partitionen
    - ABER: nicht alle Prozesse gleich groß => unterschiedlich große Partitionen
    - TROTZDEM: große Prozesse müssen zerlegt werden
    - ⇒ ***(interne) Fragmentierung***: Teile der Partitionen bleiben unbenutzt

8 M (BS)
8 M
8M
8M
8M
8M
8M

8 M (BS)
4M
6M
8M
12M
16M

2.4 Speicherverwaltung

- ***Dynamische Partitionierung*** in Partitionen variabeler Größe, d.h. für jeden Prozess wird genau der benötigte Speicherplatz zugewiesen
  - ***(externe Fragmentierung)***: zwischen den Partitionen können Lücken entstehen



  - Speicherbelegungsstrategien wie z.B. „Best Fit“, „First Fit“, „Next Fit“ nötig
- ***Defragmentierung*** durch Verschieben der Speicherbereiche prinzipiell möglich, aber sehr aufwändig (normaler Betriebsablauf muss komplett unterbrochen werden) !!!

2.4 Speicherverwaltung

- Nachteile der bisherigen Konzepte
  - Prozess komplett im HS, obwohl oft nur ein kleiner Teil benötigt wird
  - Platz für Programme und Daten ist durch Hauptspeicherkapazität begrenzt
  - zusammenhängende Speicherbelegung für einen Prozess verschärft Fragmentierung
  - Speicherschutz muss vom BS explizit implementiert werden
- Lösung: ***virtueller Speicher***
  - Prozessen mehr Speicher zuordnen als eigentlich vorhanden
  - nur bestimmte Teile der Programme werden in HS geladen
  - Rest wird auf dem Hintergrundspeicher (HGS) abgelegt
  - Prozesse werden ausgeführt, obwohl nur zum Teil im HS eingelagert
  - physischer Adressraum wird auf einen virtuellen (logischen) Adressraum abgebildet

2.4 Speicherverwaltung

- Virtueller Speicher
  - Paging
    - ein Programm wird in Seiten fester Größe (***Pages***) aufgeteilt
    - der reale HS wird in Seitenrahmen (***Frames***) fester Größe aufgeteilt
    - jeder Frame kann eine Page aufnehmen
    - Pages können nach Bedarf in freie Frames (im Hauptspeicher) eingelagert werden
    - Programm arbeitet mit virtuellen Speicheradressen (Pages), die bei Adressierung (Holen von Befehlen, Holen und Abspeichern von Operandenwerten, usw.) in eine reale Hauptspeicheradresse umgesetzt werden müssen
    - Diese Zuordnung übernimmt das Betriebssystem und die Memory Management Unit (MMU) mit Hilfe einer Seitentabelle (page table)

2.4 Speicherverwaltung

- Prinzip des Paging:
  - um für die momentan arbeitenden Prozesse genügend Platz zu haben, werden nicht benötigte Seiten auf den HGS ausgelagert
  - wird eine Seite benötigt, die nicht im Hauptspeicher sondern nur auf dem HGS lagert, so tritt ein ***Seitenfehler*** auf
  - die benötigte Seite muss in den Hauptspeicher geladen werden
  - falls der Hauptspeicher schon voll ist, muss eine/mehrere geeignete Seiten ausgelagert werden
  - Abschließend wird die Seitentabelle entsprechend aktualisiert
- Behandlung von Seitenfehlern: ***Seitenersetzung***
  - Behandlung von Seitenfehlern muss effizient sein, sonst wird der Seitenwechsel leicht zum Flaschenhals des gesamten Systems
  - Bei der Auswahl der zu ersetzenden Seite(n) sollten immer solche ausgewählt werden, auf die möglichst nicht gleich wieder zugegriffen wird (sonst müssten diese gleich wieder eingelagert werden, auf Kosten von anderen Seiten)

DATABASE SYSTEMS GROUP
Kapitel 2: Betriebssysteme

2.4 Speicherverwaltung

LMU

- Seitenersetzungsstrategien (Auswahl)
  - Optimal (OPT)

    ersetzt die Seite, die am längsten nicht benötigt wird; leider nicht vorhersehbar und daher nicht realisierbar
  - Random

    zufällige Auswahl; schnell und einfach zu realisieren, aber keine Rücksicht auf das Seitenreferenzverhalten des Prozesses
  - First In, First Out (FIFO)

    ersetzt die älteste Seite im Hauptspeicher; einfach zu realisieren, aber Seitenzugriffshäufigkeit wird nicht berücksichtigt
  - Least Recently Used (LRU)

    ersetzt die Seite, auf die am längsten nicht mehr zugegriffen wurde; berücksichtigt die Beobachtung, dass Seiten, die länger nicht mehr verwendet wurden, auch in Zukunft länger nicht benötigt werden (Prinzip der Lokalität)