



LUDWIG-
MAXIMILIANS-
UNIVERSITY
MUNICH



DEPARTMENT
INSTITUTE FOR
INFORMATICS



DATABASE
SYSTEMS
GROUP

Skript zur Vorlesung:

Einführung in die Informatik: Systeme und Anwendungen

Sommersemester 2011

Kapitel 2: Betriebssysteme

Vorlesung: Prof. Dr. Christian Böhm

Übungen: Annahita Oswald

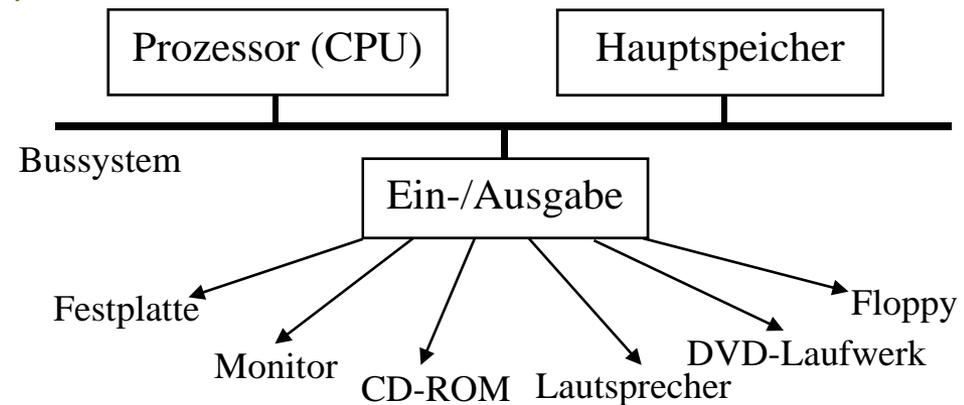
Skript © 2004 Christian Böhm, Peer Kröger

<http://www.dbs.ifi.lmu.de/cms/>

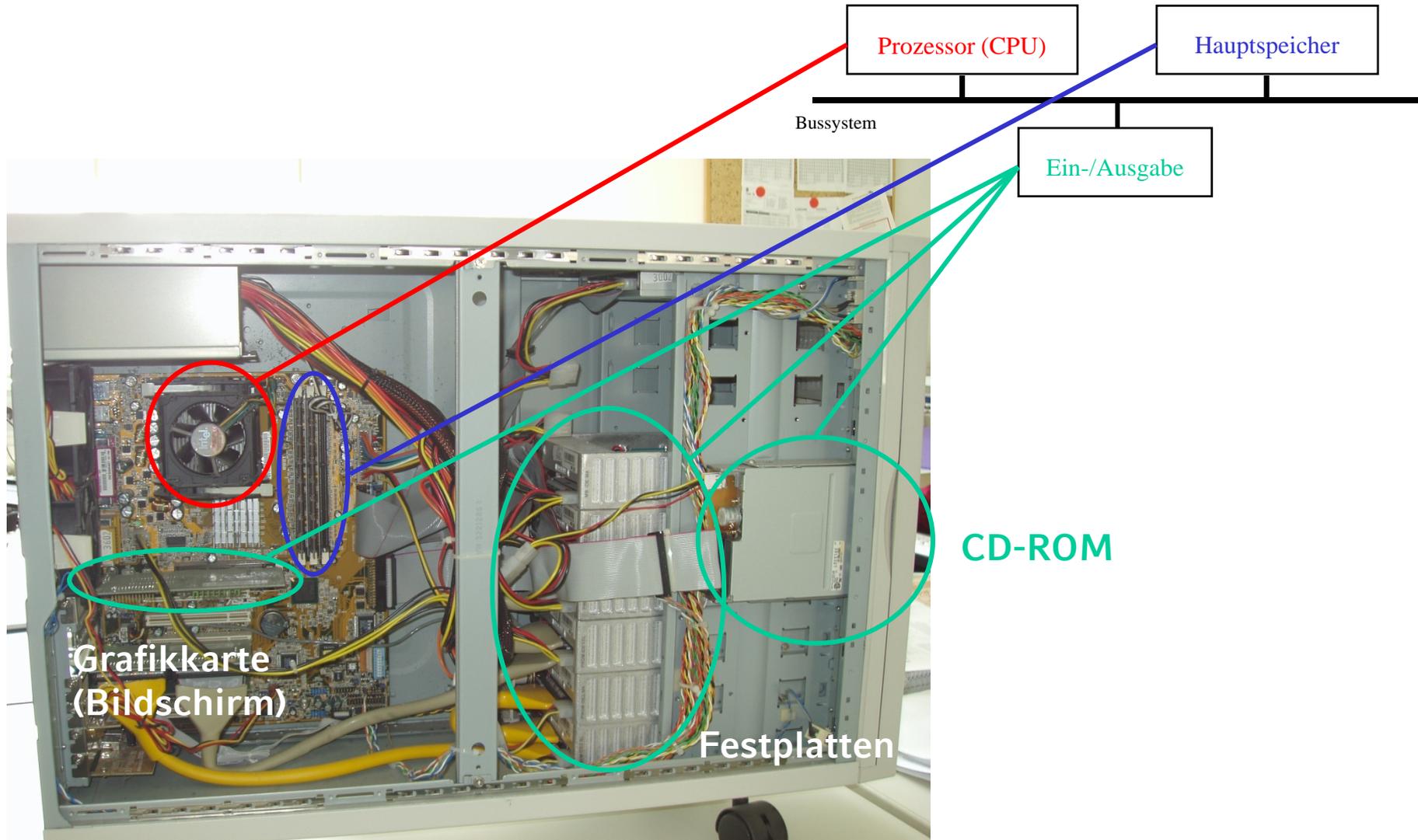
Einführung_in_die_Informatik_Systeme_und_Anwendungen

2.1 Rechner und Programme

- Komponenten eines Rechners (von-Neumann-Architektur)/Hardware
 - CPU (Prozessor)
 - Ausführung von Befehle und Ablaufsteuerung
 - Speicher (Hauptspeicher)
 - Ablegen von Daten und Programmen, binär codiert
 - Ein-/Ausgabe-Einheiten
 - Ein- und Ausgabe von Daten und Programmen
 - Bildschirm, Tastatur, Drucker, Festplatte, ...
 - Busse
 - Informationsübertragung zwischen diesen Einheiten



2.1 Rechner und Programme



2.1 Rechner und Programme

- Warum Betriebssysteme (Teil 1)?
 - CPU arbeitet schneller als Hauptspeicher
 - Beim Laden von Daten aus dem Hauptspeicher muss die CPU warten
 - Hauptspeicher arbeitet schneller als Hintergrundspeicher
 - Bei I/O-Operation (Laden von Daten, Drucken, CD-Brennen, etc.) muss CPU warten

⇒ CPU als wichtigste Ressource ist nicht ausgelastet!!!
(„von-Neumann-Flaschenhals“)

- Lösung: Prozesskonzept
 - Der Rechner führt mehrere „Arbeiten“ (Prozesse) quasi gleichzeitig aus
 - Wenn die CPU für die Bearbeitung des einen Prozesses wartet (z.B. auf eine I/O-Operation) kann die CPU mit der Bearbeitung eines anderen Prozesses fortfahren
 - Das Betriebssystem muss die Verwaltung der Prozesse für den Benutzer transparent organisieren

2.1 Rechner und Programme

- Warum Betriebssysteme (Teil 2)?
 - Ausführung von Programmen auf Rechnern
 - Zentrale Verarbeitungsschritte als „Maschinenanweisungen“ („Mikroprogramm“, ca. 50 – 300 Anweisungen je nach Hersteller)
 - Lesen/Schreiben einer Speicherzelle
 - Einfache Arithmetik
 - etc.
 - Daten sind maschinennah (meist binär) repräsentiert
 - Darstellung von Programmen:
 - Direkte Programmierung der Hardware mit „maschinennaher“ Programmiersprache (Beispiel Fahrtkosten-Algorithmus)

```
pushl    %ebp
movl     %esp, %ebp
movl     16(%ebp), %eax
subl     8(%ebp), %eax
leal    (%eax,%eax,2), %edx
```

...

2.1 Rechner und Programme

- Programmierung muss sich auch um die einzelnen Hardware-Komponenten kümmern, die benutzt werden sollen
- Beispiel: Laden einer Datei von Festplatte
 1. [optional] starten des Laufwerkmotors
 2. Positionierung des Lesekopfs
 3. Sektorenweise Einlesen des Directory
 4. Suchen der Dateiinformationen (Dateianfang) im Directory
 5. Positionierung des Lesekopfs
 6. Teil einlesen, Verknüpfung zum nächsten Teil erkennen und weiter mit Schritt 5 solange Ende der Datei noch nicht erreicht

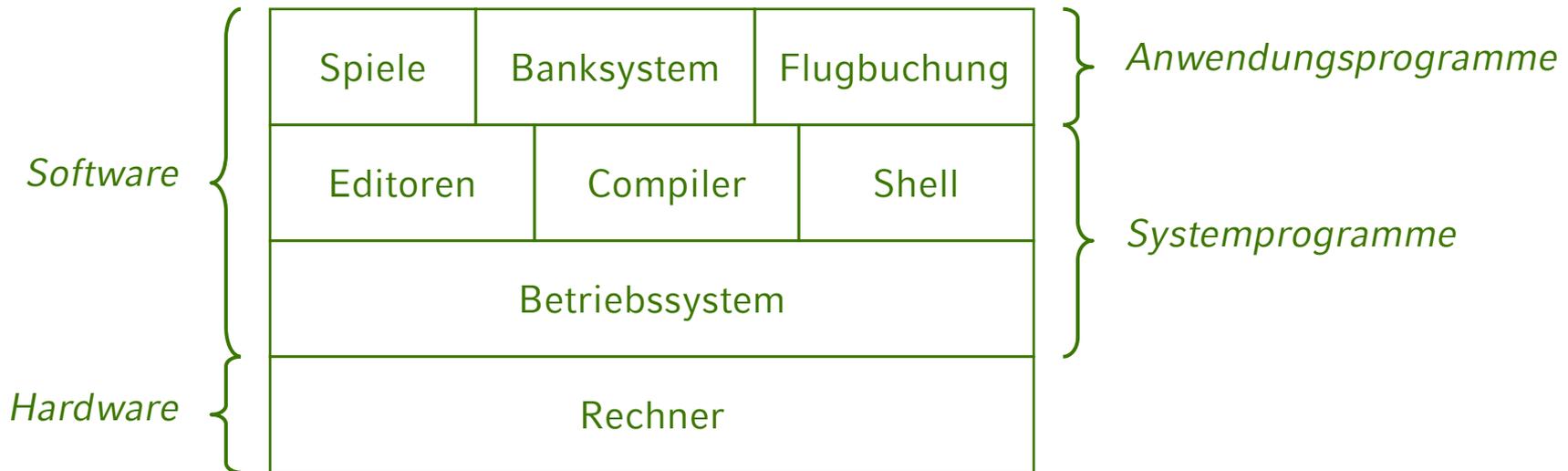
⇒ *Für jeden Befehl an den Plattenkontroller werden die Adresse der Spur, die Adresse der Hauptspeicherzelle (Ziel), die Menge der zu übertragenden Daten, der Befehlscode (Lesen/Schreiben), etc. benötigt*

– Problem:

- Programm ist sehr unübersichtlich und für Menschen schwer zu verstehen
- Hardware ist ebenfalls sehr komplex und besteht aus vielfältigen Komponenten, deren Realisierungsdetails ebenfalls schwer für den Menschen zu verstehen sind

2.1 Rechner und Programme

- Lösung: Prinzip der „Software-Schichtung“
 - BS bildet Schnittstelle für Anwendungsprogramme und spezielle Systemprogramme zur Hardware
 - ⇒ BS bewahrt den Nutzer vor der Komplexität der HW
 - ⇒ ermöglicht indirekt das Nutzen von Programmiersprachen, die für den Menschen leichter zu verstehen/benutzen sind
 - BS bildet die SW-Schicht, die alle Teile des Systems verwaltet und auf dem Anwendungen einfacher zu programmieren sind



2.1 Rechner und Programme

- Aufgaben des Betriebssystems
 - Schnittstelle für Anwendungsprogramme zur HW
 - Steuerung und Verwaltung von Computersystemen
 - **Prozessverwaltung**: Steuerung der Ausführung eines oder mehrerer Prozesse, insbesondere im Mehrprogrammbetrieb
 - **Speicherverwaltung** für den Hauptspeicher
 - **Dateiverwaltung** (im Hintergrundspeicher)
 - **Verwaltung** der E/A-Geräte
- Einige bekannte Betriebssysteme
 - Windows NT, 2000, XP, Vista, Mobile, ...
 - Apple OS X
 - LINUX
 - Unix-Varianten wie Solaris, HP/UX
 - MVS, VM/SP, CMS, BS 2000 (alle für Großrechner)

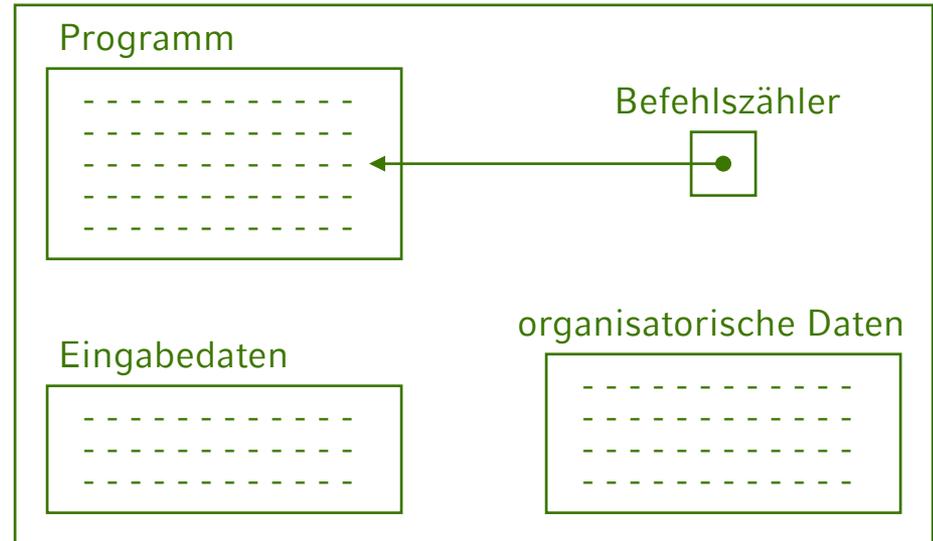
2.2 Prozesse

- Auf einem Rechner laufen „gleichzeitig“ verschiedene (Anwender- und System-) Programme

- **Prozess:**

Ein Prozess ist ein in Ausführung befindliches Programm. Dies umfasst:

- Befehlszähler (Programmzähler)
 - bestimmt den als nächstes auszuführenden Befehl
- Programmtext
- Eingabedaten
- Organisatorische Daten



2.2 Prozesse

- Multiprogramming:

- BS kann mehrere Prozesse gleichzeitig ausführen, d.h. auch ein Programm kann mehrmals gleichzeitig ausgeführt werden
- Warum ist das sinnvoll?
 - Programme benötigen außer CPU meist auch E/A-Geräte
 - E/A-Geräte sind deutlich langsamer als CPU (Flaschenhals)
⇒ Prozessor muss warten und ist nicht ausgelastet

Beispiel:

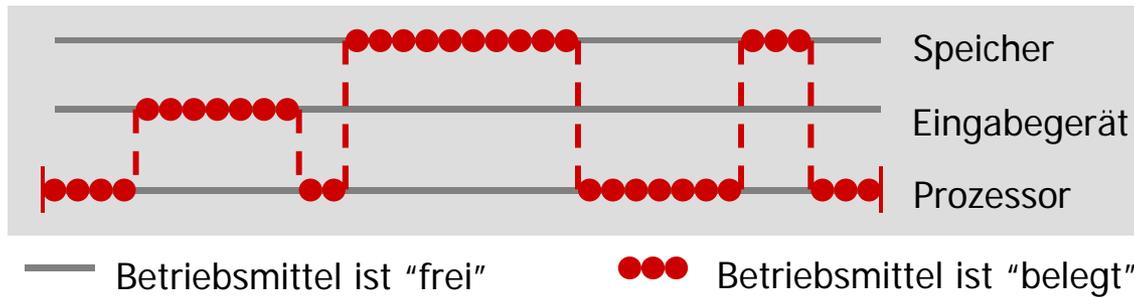
Lesen eines Datensatzes	0,0015 sec.
Ausführen von 100 Befehlen	0,0001 sec.
Schreiben des Datensatzes	0,0015 sec.
<hr/>	
	0,0031 sec.

CPU-Auslastung: $0,0001/0,0031 \approx 3,2 \%$

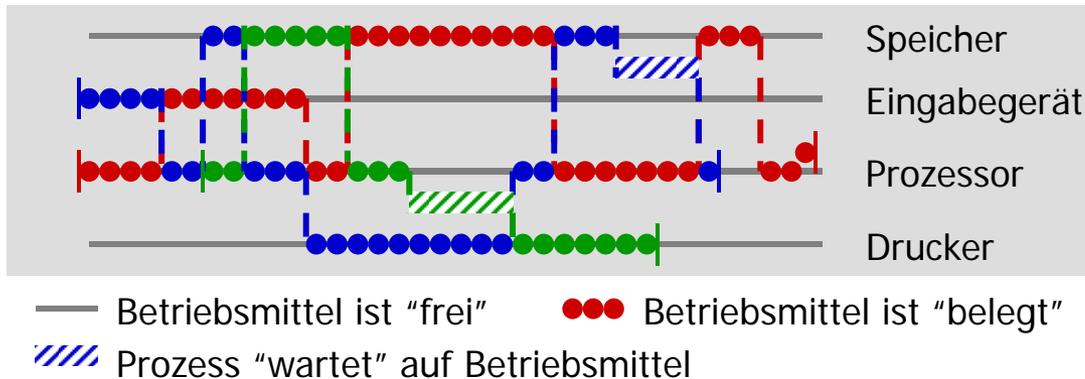
3,2 % der Zeit arbeitet CPU; 96,8 % der Zeit wartet CPU

2.2 Prozesse

- Lösung: Anwendungsprogramme sollen also dem Prozessor abwechselnd zugeteilt werden
 - Einbenutzerbetrieb (*Uniprogramming*) [1 Prozess]



- Mehrbenutzerbetrieb (*Multiprogramming*) [hier: 3 Prozesse rot/grün/blau]



2.2 Prozesse

- Motivation (cont.)
 - Einzelne Prozesse durchlaufen damit ihre Anweisungsfolge (zumeist) nicht in einem Schritt sondern werden häufig unterbrochen
 - Es entsteht der Eindruck von **Parallelität** aller momentan existierenden Prozesse (dies ist allerdings nur eine **Quasi-Parallelität** !)
 - Folge:
 - zeitliche Dauer eines Prozesses kann bei verschiedenen Programmausführungen unterschiedlich sein
 - Insbesondere keine *a priori* Aussagen über den zeitlichen Ablauf eines Prozesses möglich
- Rolle des Betriebssystems
 - Aus Sicht des Anwendungsprogramms steht jedem Prozess ein eigener (virtueller) Rechner (CPU aber auch Hauptspeicher, etc.) exklusiv zur Verfügung
 - BS muss Abbildung auf den realen Rechner leisten

2.2 Prozesse

- Prozesszustände

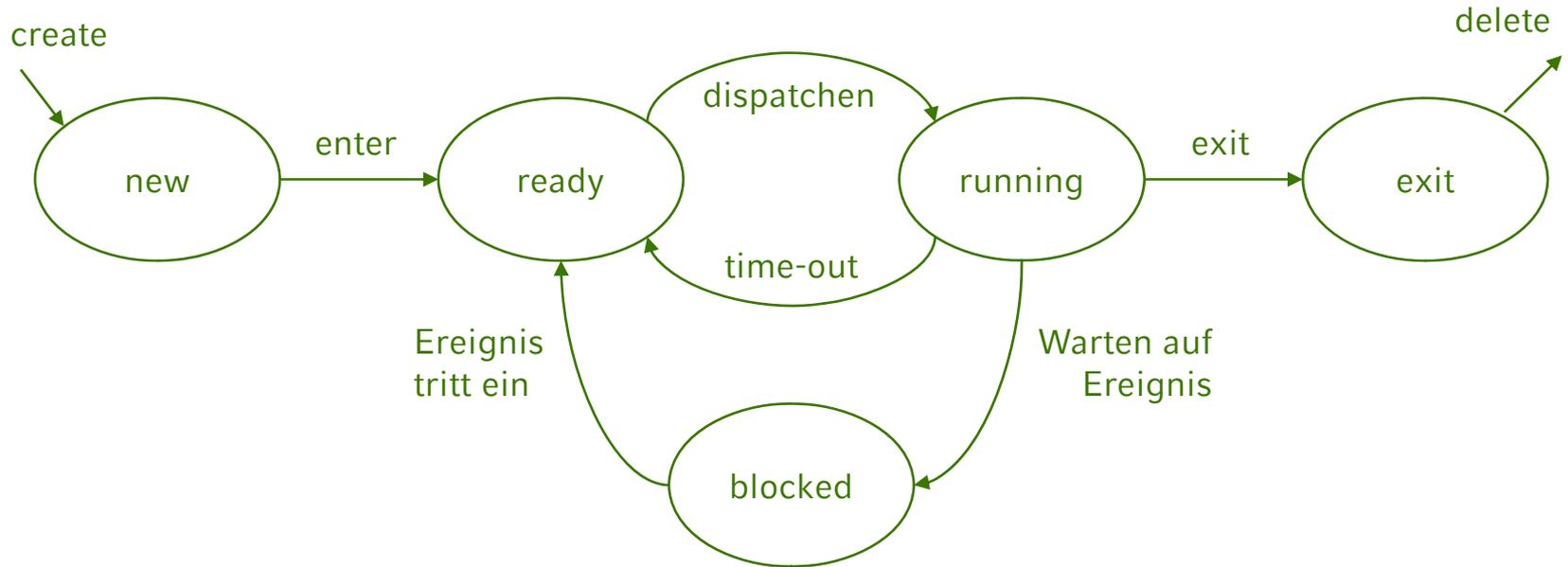
- 5-Zustands-Modell

Prozess ist entweder

- **new**: Prozess ist erzeugt, aber noch nicht zu der Menge der ausführbaren Prozesse hinzugefügt
- **ready**: Prozess ist zur Ausführung bereit, aber ein anderer Prozess ist dem Prozessor zugeteilt
- **running**: Prozess ist dem Prozessor zugeteilt und wird gerade ausgeführt
- **blocked**: Prozess ist blockiert, d.h. er wartet auf das Eintreten eines externen Ereignisses (Beendigung einer E/A-Operation, benötigtes Betriebsmittel ist belegt, ...)
- **exit**: Prozess wurde beendet, d.h. Ausführung ist abgeschlossen

2.2 Prozesse

– Graphische Darstellung



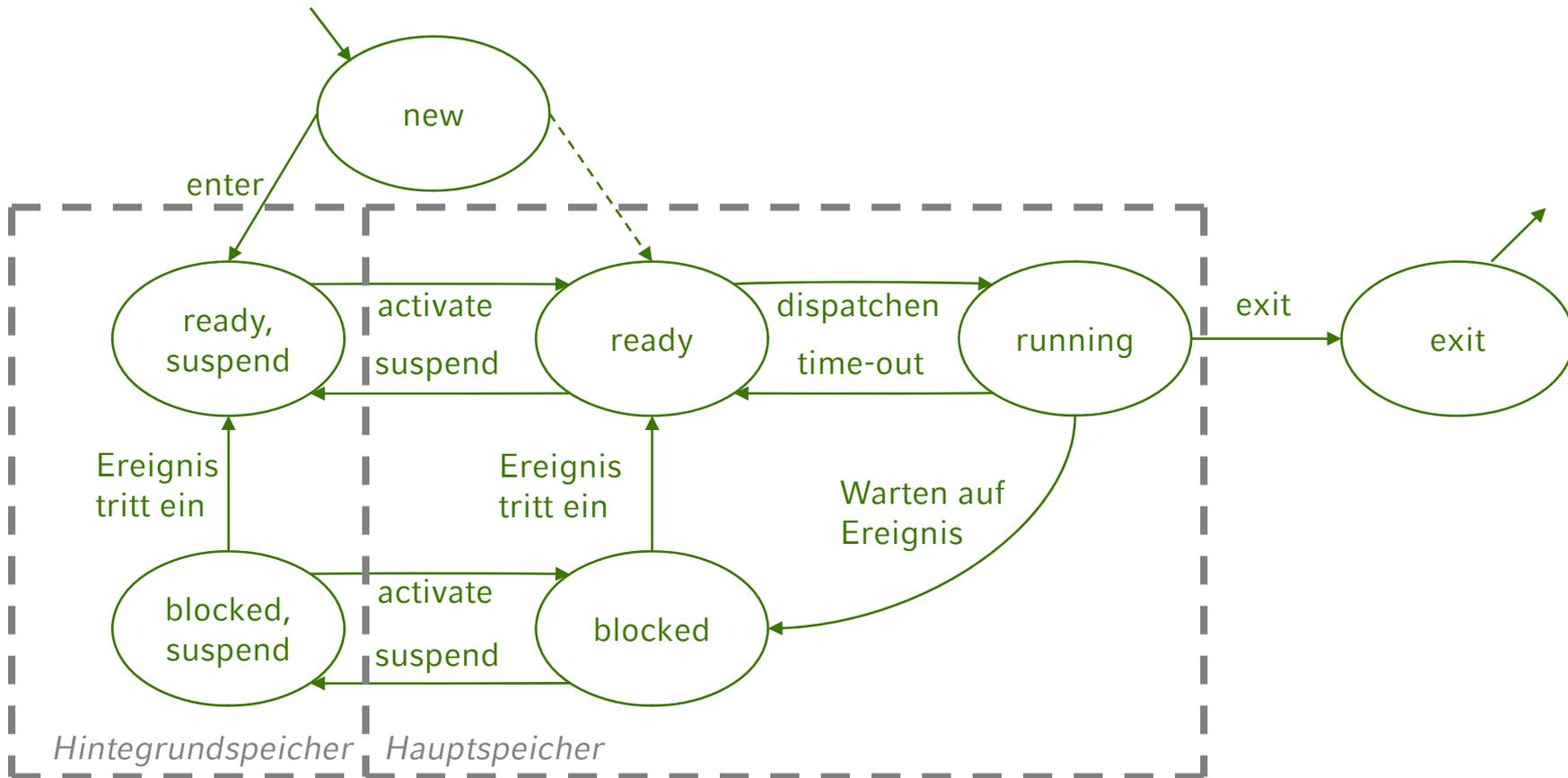
2.2 Prozesse

– 7-Zustands-Modell

- bisher: alle Prozesse werden im Hauptspeicher gehalten
- wenn alle Prozesse blockiert sind (E/A-intensive Prozesse) und der Hauptspeicher voll ist, können keine weiteren Prozesse gestartet werden
⇒ Prozessor wäre wieder unbenutzt!!!
- Lösung 1: Hauptspeicher erweitern
⇒ schlecht: kostet Geld, größere Programme?
- Lösung 2: Prozesse (oder Teile davon) auf den Hintergrundspeicher (Festplatte, „HGS“) auslagern (**Swapping**)
⇒ gut: neue Prozesse haben im Hauptspeicher Platz
⇒ schlecht: eine zusätzliche E/A-Operation
- Um übermäßig große Warteschlangen auf Festplatte zu vermeiden, sollte immer ein ausgelagerter Prozess wieder eingelagert werden; dieser sollte aber nicht mehr blockiert sein
⇒ unterscheide, ob ausgelagerte Prozesse blockiert oder bereit sind
⇒ 2 neue Zustände

2.2 Prozesse

- **ready, suspend**: auf dem HGS ausgelagert, bereit
- **blocked, suspend**: auf dem HGS ausgelagert, wartet auf Ereignis (blockiert)

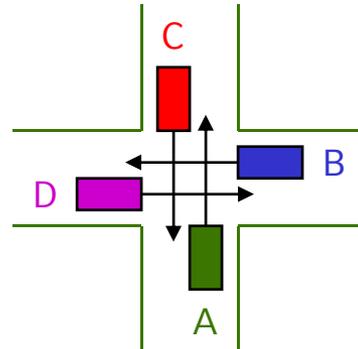


2.2 Prozesse

- Prozess-Scheduling
 - Scheduler:
 - („Faire“) Zuteilung eines Prozesses an den Prozessor
 - (Entscheidung über Swapping)
 - Scheduling-Verfahren
 - **Round Robin** (einfach und häufig verwendet)
 - Wartende Prozesse in einer (FIFO-) Warteschlange organisieren
 - Jeder Prozess hat für eine bestimmte Zeitspanne die CPU
 - Danach wird Prozess gegebenenfalls wieder in Warteschlange eingefügt
 - Zyklisches Abarbeiten der Warteschlange
 - **Prioritäts-Scheduling**
 - Jedem Prozess ist eine Priorität zugeordnet
 - Prozess mit höchster Priorität bekommt den Prozessor zugeordnet
 - **Shortest-Job-First**
 - Prozess mit der kürzesten Laufzeit

2.3 Prozessverwaltung

- Probleme der Parallelität beim Multiprogramming
 - Parallel ablaufende Prozesse können voneinander abhängig sein
 - Um fehlerhaftes Verhalten zu verhindern müssen die Prozesse geeignet koordiniert werden
 - Beispiel für fehlerhaftes Verhalten: Verklemmung (**Deadlock**)
 - Ein Deadlock ist die dauerhafte Blockierung einer Menge M von Prozessen, die eine Menge S gemeinsamer Systemressourcen nutzen oder miteinander kommunizieren ($|M| > 1, |S| > 1$)
 - Anschauliches Beispiel: 4 Fahrzeuge gleichzeitig an einer Rechts-vor-Links Kreuzung



A wartet auf B
 B wartet auf C
 C wartet auf D
 D wartet auf A

⇒ daher notwendig: **Prozessverwaltung (Prozess-Synchronisation)**

2.3 Prozessverwaltung

- Grundmuster der Prozess-Synchronisation
 - Prozesskooperation (hier nur kurz besprochen)
 - Wechselseitiger Ausschluss (hier etwas genauer besprochen)
- Prozesskooperation
 - Prozesse können zusammenarbeiten
 - Operationen eines Prozesses können voraussetzen, dass gewisse Operationen in anderen Prozessen erledigt sind
 - Grundmuster
 - Erzeuger-Verbraucher-Schema



- Auftraggeber-Auftragnehmer-Schema



2.3 Prozessverwaltung

- Wechselseitiger Ausschluss
 - Parallel abgearbeitete Prozesse können sich gegenseitig beeinflussen
 - Beispiel: Flugbuchung
 - Algorithmus zur Reservierung von n Plätzen

```

algorithmus Reservierung
variables   n, anzpl : Nat
begin
    ...
    while Buchung noch nicht abgeschlossen do {
        n = Anzahl der zu buchenden Plätze;
        anzpl = aktuelle Anzahl der freien Plätze;
        if anzpl  $\geq$  n
        then anzpl um n verringern;
        else STOP mit Auskunft „Ausgebucht“;
        Reservierung bestätigen;
        Frage nach weiterer Buchung;           // Nein: Verlassen der Schleife
    }
    ...
end
  
```

2.3 Prozessverwaltung

- Variable `anzpl` gibt zu jedem Zeitpunkt die Anzahl noch verfügbarer Plätze an
- Zwei parallel ablaufende Prozesse A und B (Buchungen für jeweils 2 Plätze) führen den Algorithmus aus zum Zeitpunkt `anzpl = 3`
- Möglicher Ablauf:

Prozess A	Prozess B	Wert der Variablen <code>anzpl</code>
<code>anzpl ≥ n (3 ≥ 2) ?</code>		3
	<i>Prozesswechsel</i> →	
	<code>anzpl ≥ n (3 ≥ 2) ?</code>	3
	<code>anzpl um n=2 verringern</code>	1
	<code>Reservierung bestätigen</code>	1
	<i>Prozesswechsel</i> ←	
<code>anzpl um n=2 verringern</code>		-1
<code>Reservierung bestätigen</code>		-1

2.3 Prozessverwaltung

- Effekt: es wurden 4 Plätze vergeben, obwohl nur noch 3 Plätze frei waren
- Wie ist das zu verhindern?
 - Im Algorithmus Reservierung gibt es einen Bereich, der „kritisch“ ist
 - if** $\text{anzpl} \geq n$
 - then** anzpl um n verringern
 - else** STOP mit Auskunft „Ausgebucht“
 - Reservierung bestätigen
 - Gleichzeitige Ausführung dieses **kritischen Bereiches** eines Algorithmus durch mehrere verschiedene Prozesse muss verhindert werden
 - **Wechselseitiger Ausschluss:**
 - „Keine zwei Prozesse befinden sich gleichzeitig in ein und dem selben kritischen Bereich“

2.3 Prozessverwaltung

- Kritischer Bereich
 - Ein **kritischer Bereich** ist ein Programm(stück), das auf gemeinsam benutzte Ressourcen (globale Daten, Dateien, bestimmte E/A-Geräte, ...) zugreift oder den Zugriff darauf erfordert.
 - Die Ressource wird entsprechend **kritische Ressource** genannt
 - ⇒ Aufteilung von Programmen in kritische und unkritische Bereiche
- Wechselseitiger Ausschluss
 - Solange ein Prozess sich in einem kritischen Bereich befindet, darf sich kein anderer Prozess in diesem kritischen Bereich befinden!

2.3 Prozessverwaltung

- Anforderungen an den wechselseitigen Ausschluss
 1. *mutual exclusion*

zu jedem Zeitpunkt darf sich höchstens ein Prozess im kritischen Bereich befinden
 2. *progress – no deadlock*

wechselseitiges Aufeinanderwarten muss verhindert werden
Beispiel: Prozesse A und B; kritische Ressourcen a und b;
A belegt a,
B belegt b,
B möchte a belegen \Rightarrow muss auf A warten
A möchte b belegen \Rightarrow muss auf B warten
 \Rightarrow Deadlock !!!
 3. *bounded waiting – no starvation*

bei 3 Prozessen A, B, C könnten sich z.B. A und B in der Nutzung einer kritischen Ressource immer abwechseln
 \Rightarrow C müsste beliebig lange warten („**Verhungern**“, **starvation**) !!!

2.3 Prozessverwaltung

- Lösung nach Dekker (ca. 1965)

- Prinzip:

- globale, geschützte Variable `turn` zeigt an, welcher Prozess in kritischen Bereich eintreten darf
- nur wenn Variable den Wert des entsprechenden Prozesses enthält, darf dieser Prozess in den kritischen Bereich
- Nach Verlassen des kritischen Bereichs wird die Variable auf den Wert des anderen Prozesses gesetzt

- Schema:

prozess P0

```

...
while turn <> 0
  do { nothing };
< kritischer Bereich >;
turn = 1;
...

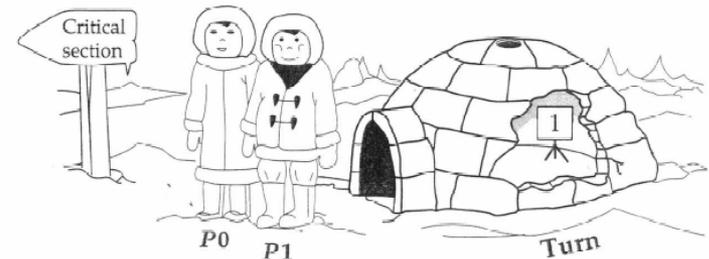
```

prozess P1

```

...
while turn <> 1
  do { nothing };
< kritischer Bereich >;
turn = 0;
...

```



2.3 Prozessverwaltung

– Probleme:

- Prozesse können nur abwechselnd in den kritischen Bereich eintreten. Auch bei Erweiterung auf $n > 1$ Prozesse (dann würde `turn` Werte von 0 bis $n-1$ annehmen) wäre die Reihenfolge des Eintritts in den kritischen Bereich festgeschrieben.
- Terminiert ein Prozess (im unkritischen Bereich), so kann der andere nur noch einmal in den kritischen Bereich eintreten. Bei allen weiteren Versuchen, in den kritischen Bereich einzutreten würde er dann blockiert werden

=> *progress*-Eigenschaft wird verletzt!

2.3 Prozessverwaltung

- Beispiel: Flugbuchung mit nur zwei Prozessen (d.h. es dürfen nur zwei Reisebüros weltweit gleichzeitig Flüge buchen)

<i>Prozess 0</i>	<i>Prozess 1</i>
<pre> ... while Buchung nicht abgeschlossen do { n = Anzahl der zu buchenden Plätze; anzpl = aktuelle Anzahl der freien Plätze; while turn < 0 do { nothing; } if anzpl ≥ n then anzpl um n verringern; else STOP mit Auskunft „Ausgebucht“; Reservierung bestätigen; turn = 1; Frage nach weiterer Buchung; } ... </pre>	<pre> ... while Buchung nicht abgeschlossen do { n = Anzahl der zu buchenden Plätze; anzpl = aktuelle Anzahl der freien Plätze; while turn < 1 do { nothing; } if anzpl ≥ n then anzpl um n verringern; else STOP mit Auskunft „Ausgebucht“; Reservierung bestätigen; turn = 0; Frage nach weiterer Buchung; } ... </pre>

- Problem: wenn z.B. Prozess 0 endet (keine weiteren Buchungen), kann Prozess 1 nur noch eine weitere Buchung machen

2.3 Prozessverwaltung

- HW-Lösung: Unterbrechungsvermeidung
 - bei Einprozessorsystemen können Prozesse nicht echt parallel ausgeführt werden
 - Prozesswechsel während des Aufenthalts im kritischen Bereich verursacht die Probleme
 - Idee: Unterbrechungen (Prozesswechsel) im kritischen Bereich ausschließen
 - Muster:

```
...  
< unkritischer Bereich >;  
ab hier: verbiete Unterbrechungen;  
< kritischer Bereich >;  
ab hier: erlaube Unterbrechungen;  
< unkritischer Bereich >;  
...
```

Funktioniert nicht bei Multiprozessorsystemen !!!

(Warum?)

2.3 Prozessverwaltung

– Beispiel: Flugbuchung

algorithmus Reservierung

variables n, anzpl : **Nat**

begin

...

while Buchung nicht abgeschlossen **do** {

n = Anzahl der zu buchenden Plätze;

anzpl = aktuelle Anzahl der freien Plätze;

ab hier: verbiete Unterbrechungen;

if anzpl \geq n

then anzpl um n verringern;

else STOP mit Auskunft „Ausgebucht“;

Reservierung bestätigen;

ab hier: erlaube Unterbrechungen;

Frage nach weiterer Buchung;

}

...

end

2.3 Prozessverwaltung

- Semaphore
 - Semaphore sind spezielle (globale) Variablen, die Signale zwischen Prozessen übertragen
 - Ein Semaphor kann
 - Werte 0 oder 1 annehmen \Rightarrow binäres Semaphor
 - Beliebige **Int**-Werte annehmen \Rightarrow Zählsemaphor
 - Zusätzlich wird von einem Semaphor noch eine Menge von Prozessen verwaltet (in einer Warteschlange)
 - Als Datenstruktur bietet sich für Semaphore der Record-Typ an
 - Wir müssen zwei Werte von unterschiedlichem Typ verwalten
 - Einen Wert aus $\{0,1\}$ bzw. **Int**
 - Eine Menge von Prozessen (z.B. deren Ids als **Nat**)

2.3 Prozessverwaltung

– Record-Typ **BinarySemaphore**

```
RECORD BinaerSemaphor =  
(  
    value : {0,1},  
    queue : NatList,    // Liste von ProzessIDs  
)
```

– Record-Typ **Semaphore**

```
RECORD ZaehlSemaphor =  
(  
    value : Int,  
    queue : NatList,    // Liste von ProzessIDs  
)
```

2.3 Prozessverwaltung

– Exkurs: Globale Variable

- Bereits bei der Lösung nach Dekker (siehe Folie 25) haben wir eine „globale“ Variable verwendet.
- Bisher: Variablen wurden innerhalb eines Algorithmus verwendet um Ein- und Ausgabedaten zu verwalten, Zwischenergebnisse zu berechnen, etc.
- Dabei waren die Variablen streng genommen nur für den Algorithmus selber (den Prozess, der diesen Algorithmus ausführt) sichtbar.
- Selbst wenn zwei Prozesse den selben Algorithmus ausführen, gibt es für jede Variable zwei unterschiedliche Versionen, d.h. jeder Prozess kann nur seine eigenen „Zettel“ lesen bzw. schreiben
- Damit Prozesse miteinander kommunizieren zu können benötigen wir stattdessen „Zettel“ (Variablen), die von allen Prozessen gelesen/geschrieben werden können
- Diese Variablen sind globale Variablen.
- Wenn wir globale Variablen verwenden, wird das vor der Beschreibung eines Algorithmus gekennzeichnet

2.3 Prozessverwaltung

- Realisierung eines Semaphors:
 - Ein Semaphor speichert also zwei Informationen:
 - Der Wert des Semaphors
 - Eine Liste von blockierten Prozessen (bzw. deren IDs in einer 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“
 - Operationen auf Semaphor `s`:
 - `init(s, Anfangswert)` setzt `s.value` auf den Anfangswert
 - `wait(s)` versucht `s.value` zu dekrementieren; ein negativer Wert (bzw. „0“ bei einem binären `S`.) blockiert `wait`
 - `signal(s)` inkrementiert `s.value`; ggfs. wird dadurch die Blockierung von `wait` aufgehoben

2.3 Prozessverwaltung

Binäres Semaphor

Initialisierung	<pre> algorithmus init input s: BinarySemaphore, initialvalue: {0,1} begin s.value = initialvalue; end </pre>
Wait	<pre> algorithmus wait input s: BinarySemaphore begin if s.value = 1 then s.value := 0; else blockiere Prozess und plaziere ihn in s.queue; end end </pre>
Signal	<pre> algorithmus signal input s: BinarySemaphore begin if s.queue ist leer then s.value := 1; else entnehme einen Prozess aus s.queue; end end </pre>

2.3 Prozessverwaltung

Zählsemaphor

Initialisierung	<pre> algorithmus init input s: Semaphore, initialvalue: Int begin s.value = initialvalue; end </pre>
Wait	<pre> algorithmus wait input s: Semaphore begin s.value := s.value - 1; if s.value < 0 then blockiere Prozess und <u>plaziere ihn in s.queue;</u> end </pre> <p style="text-align: right; color: red;"><i>entspricht: append(s.queue,prozessID);</i></p>
Signal	<pre> algorithmus signal input s: Semaphore begin s.value = s.value + 1; if s.value <= 0 // gibt es noch blockierte Prozesse? then entnehme einen <u>Prozess aus s.queue;</u> end </pre> <p style="text-align: right; color: red;"><i>entspricht: getAndDeleteFirst(s.queue);</i></p>

2.3 Prozessverwaltung

– Programmiertechnische Nutzung von Semaphoren

```
global variable      s: BinarySemaphore;  
global operation    init(s,1);  
algorithm  
...  
begin  
    ...  
    wait(s);  
    < kritischer Bereich >;  
    signal(s);  
    ...  
end
```

– Einsatz:

- Realisierung eines wechselseitigen Ausschlusses
⇒ binäres Semaphore
- Verwaltung einer begrenzter Anzahl von Ressourcen
⇒ Zählsemaphor

2.3 Prozessverwaltung

– Beispiel: Flugbuchung

global variable **s : BinarySemaphore**
 global operation **init(s,1);**

algorithmus Reservierung

variables n, anzpl : **Nat**;

begin

...

while Buchung nicht abgeschlossen **do** {

 n = Anzahl der zu buchenden Plätze;

 anzpl = aktuelle Anzahl der freien Plätze;

wait(s);

if anzpl \geq n

then anzpl um n verringern;

else STOP mit Auskunft „Ausgebucht“;

 Reservierung bestätigen;

signal(s);

 Buchung abschließen?;

 // Nein: Verlassen der Schleife

 }

...

end

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 („Preclaiming“); 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

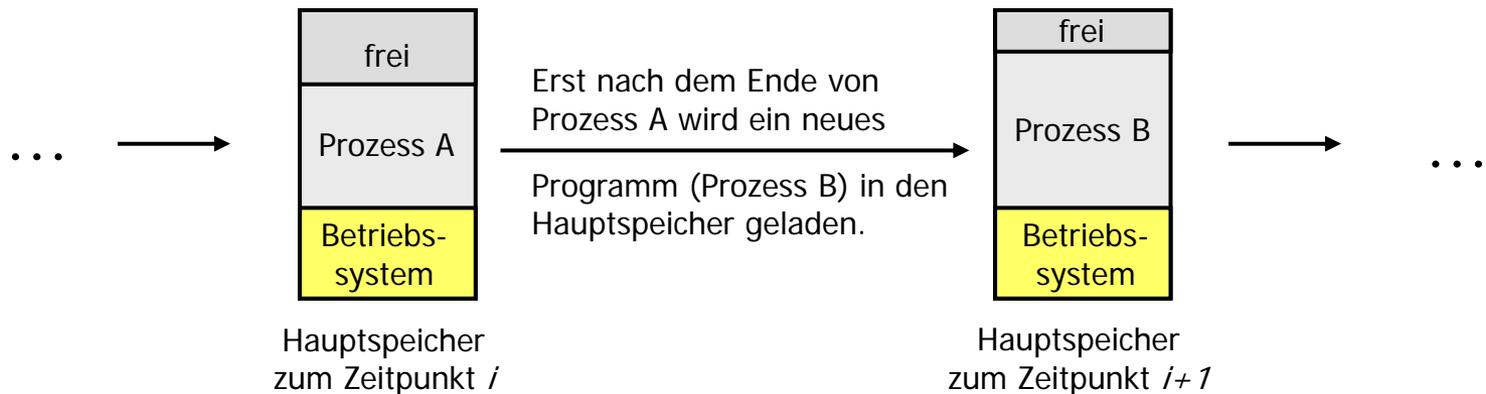
- Erkennung von Deadlocks
 - notwendige Voraussetzungen für Deadlock (trifft eine nicht zu, kann kein Deadlock entstehen):
 - **Mutual Exclusion**
Es gibt mind. 2 Ressourcen, die nur von einem (jeweils unterschiedlichen) Prozess gleichzeitig benutzt werden
 - **Hold and Wait**
Ein Prozess muss eine Ressource 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
- Praktisch kann man Deadlocks erkennen z.B. mittels
 - Time-Out-Strategien (heuristisch)
 - Wartegraphen
- Was kann getan werden um einen Deadlock zu beheben?
 - 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

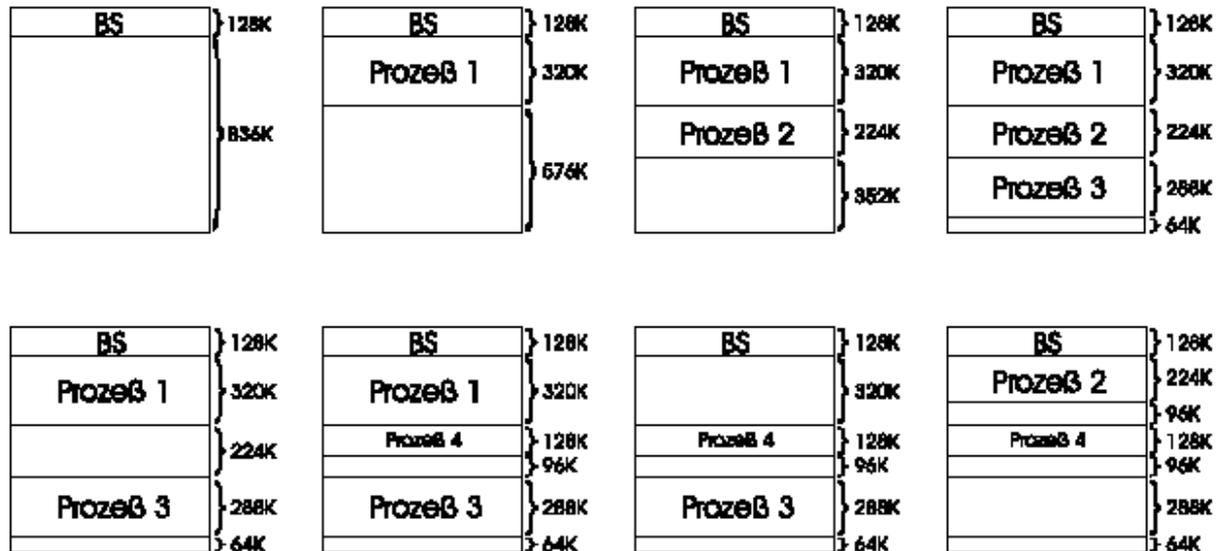
- Für das Multiprogramming ist eine Unterteilung (**Partitionierung**) des Speichers für mehrere Prozesse erforderlich
 - **Feste Partitionierung** in gleich große Partitionen
 - ABER: nicht alle Prozesse gleich groß => unterschiedlich große Partitionen
 - TROTZDEM: große Prozesse müssen zerlegt werden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

8 M (BS)
8 M
8M

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

2.4 Speicherverwaltung

- **Dynamische Partitionierung** in Partitionen variabler 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
 - Frage: was ist der Unterschied zu Swapping???
 - 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)

2.4 Speicherverwaltung

- Seiteneretzungsstrategien (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)