Einführung in die Informatik: Systeme und Anwendungen

Teil: Konzepte von Betriebssystemen

Prof. Dr. Stefan Conrad

Ludwig-Maximilians–Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme
Oettingenstraße 67, D-80538 München

conrad@dbs.informatik.uni-muenchen.de

Universität München
Sommersemester 2001

— Folienskriptum —

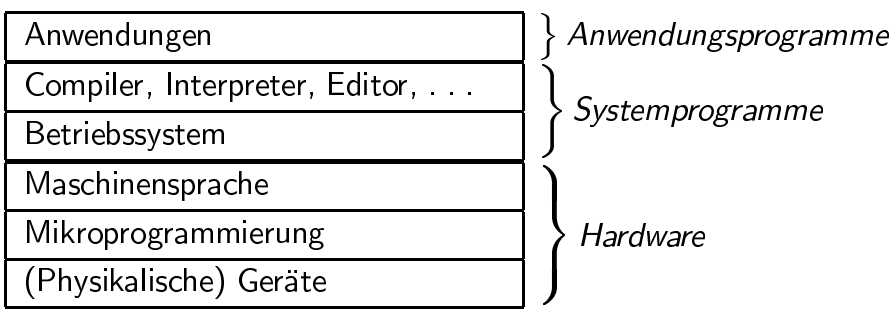
2. Grundlagen von Betriebssystemen

Betriebssystem (BS) bildet Schnittstelle für Anwendungsprogramme und spezielle Systemprogramme zur Hardware

- Hardware ist komplex und besteht aus vielfältigen Komponenten:

  Prozessor, Hauptspeicher, Taktgeneratoren, Plattenlaufwerke, Schnittstellen zu Druckern und anderen Geräten, Netzwerk-Anschluß, . . .
- **Ziel:** Nutzer vor dieser Komplexität bewahren

  $\rightsquigarrow$ Konzept der Softwareschichtung
- Betriebssystem bildet eine Software-Schicht:
  - verwaltet alle Teile des Systems
  - ist „einfach“ zu verstehen und zu programmieren



2.1 Prozesse

zentrales Konzept in jedem Betriebssystem: Prozeß

Prozeßkonzept

Moderne Systeme können mehrere „Dinge“ gleichzeitig ausführen

→ Multiprogramming

ermöglicht Multitasking
und Mehr-Benutzer-Betrieb

Motivation:

E/A-Geräte sind im Vergleich zum Prozessor of sehr langsam

⇝ Prozessor muß warten

Prozesse

Beispiel:

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

⇝ CPU-Auslastung: 0,0001/0,0031 ≈ 3,2 %

⇝ nur 3,2 % der Zeit arbeitet die CPU,
96,8 % der Zeit wartet sie

⇒ Wartezeiten verhindern bzw. für andere Aufgaben nutzen!

Idee:

Prozessor bearbeitet verschiedene Aufgaben (Programme/Prozesse) durch schnelles Umschalten „gleichzeitig“

⇝ es entsteht der Eindruck von Parallelität
(Pseudo-Parallelität!)

Prozesse

Beschreibung von Parallelität in Betriebssystemen

$\rightsquigarrow$ Prozeßmodelle

Definition: Prozeß

Ein Prozeß ist ein in Ausführung befindliches Programm. Dies umfaßt:

- den aktuellen Wert des Programmzählers
- aktuelle Werte der (CPU-) Register + Variablen

Annahme:
Jedem Prozeß steht ein eigener (virtueller) Rechner (insb. Prozessor, aber auch Hauptspeicher etc.) exklusiv zur Verfügung.

$\rightarrow$ das Betriebssystem muß eine geeignete Abbildung auf den realen Rechner leisten

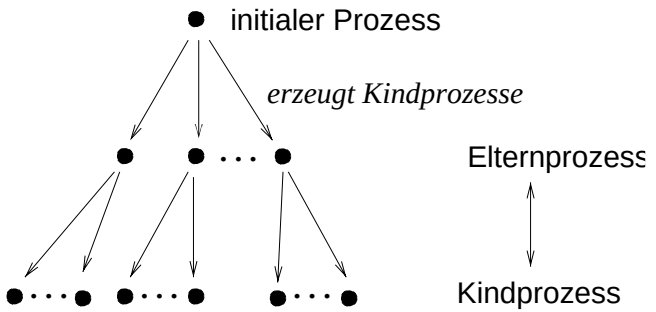
Anmerkung:
Umschalten des Prozessors zwischen mehreren Prozessen bedingt, daß Ausführungsgeschwindigkeit eines Prozesses nicht immer gleichmäßig ist
(und damit nicht reproduzierbar)

$\rightsquigarrow$ keine *a priori* Aussagen über zeitlichen Ablauf eines Prozesses möglich!

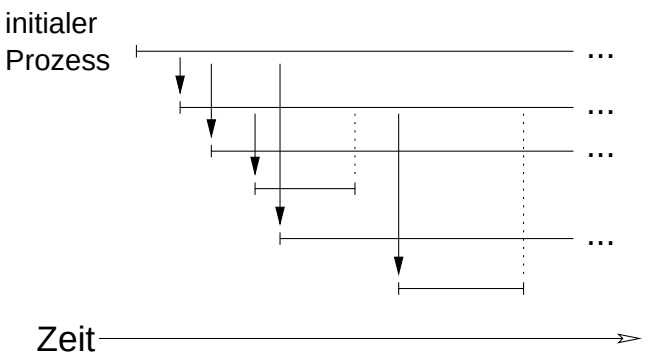
Prozesse

Prozeßentstehung: Konzept der *Prozeßhierarchie*

- es gibt einen initialen Prozeß ($\rightarrow$ Systemstart)
- jeder Prozeß kann i.d.R. neue Prozesse, sogenannte Kindprozesse (des Eltern-/Vaterprozesses), erzeugen



im zeitlichen Ablauf (Bsp.):



$\rightsquigarrow$ Eltern- und Kindprozesse können „parallel“ laufen!

Prozesse

2 prinzipielle Abhängigkeitsmöglichkeiten zwischen Eltern- und Kindprozessen:

(a) E . . . _____ . . .

K ___

Kindprozeß muß vor (oder mit) seinem Elternprozeß enden

→ Abbruch des Elternprozesses erzwingt Abbruch des Kindprozesses

(b) E . . . _____

K _____ . . .

Kindprozeß ist unabhängig

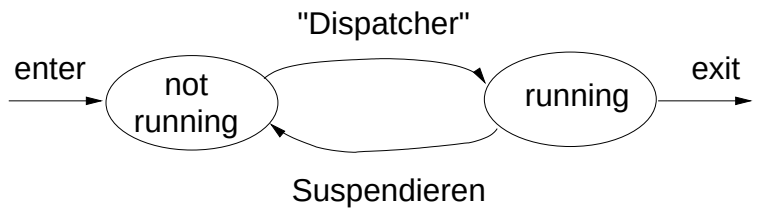
Prozesse

Prozeßzustände:

prinzipiell 2 Zustände:

- Prozeß befindet sich in Ausführung,
  d.h. der Prozessor „bearbeitet“ gerade diesen Prozeß
- Prozeß wartet (verschiedene Ursachen möglich)

2-Zustands-Prozeßmodell:



Folgerung für das Betriebssystem:

- aktueller Zustand des Prozesses muß beschreibbar sein
- Informationen für (spätere) Fortführung des Prozesses müssen abrufbar sein

Prozesse

Dispatcher:

(„erledigt“ den Prozeßwechsel)

- für den zu suspendierenden Prozeß (der zuletzt „running“ war) werden alle Informationen für die spätere Weiterverarbeitung gespeichert

- der nächste zur Ausführung anstehende Prozeß (aus der Menge der „not running“ Prozesse) wird zum „running“ Prozeß; die zur Ausführung notwendigen Informationen werden in den Prozessor geladen

Scheduler:

(plant die Reihenfolge der Ausführung der Prozesse)

- teilt dem Dispatcher mit, welcher Prozeß als nächster dran ist

- verschiedene Scheduling-Strategien möglich, z.B.:
  - einfache Warteschlange,
  - Berücksichtigung vorgegebener Prioritäten der Prozesse,
  - dynamisch veränderliche Prioritäten,
  - . . .

Prozesse

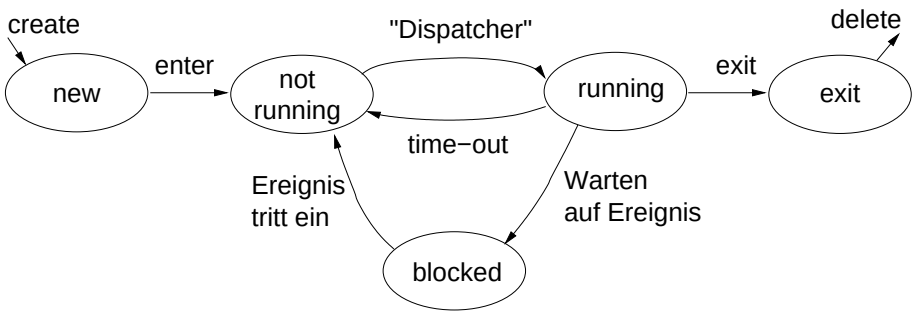
2-Zustands-Prozeßmodell ist eine (zu) starke Vereinfachung, da verschiedene Ursachen für Suspendierung eines Prozesses vorhanden sein können

→ Verfeinerung des Prozeßmodells nötig!

5-Zustands-Prozeßmodell:

Motivation

- Suspendieren eines „running“ Prozesses nach Ablauf einer bestimmten Zeit (*time-out*)
- Suspendieren bei Warten auf E/A-Operation (Prozeß ist „blockiert“)



Prozesse

Zustände des 5-Zustands-Prozeßmodells:

new: Prozeß ist erzeugt, aber noch nicht der Menge der ausführbaren Prozesse hinzugefügt

(i.d.R. wird nur eine bestimmte Anzahl von Prozessen zur Ausführung „zugelassen“)

ready: Prozeß ist zur Ausführung bereit

running: Prozeß wird gerade ausgeführt

blocked: Prozeß ist blockiert,
d.h. er wartet auf Eintreten eines externen Ereignisses (Beendigung einer E/A-Operation, . . .)

exit: Prozeß wurde beendigt, d.h. die Ausführung ist abgeschlossen

Anmerkung:
In realen Betriebssystemen gibt es zumeist weitere Zustände; dabei wird i.d.R. der Zustand „blockiert“ weiter verfeinert.

2.2 Kontrollstrukturen in Betriebssystemen

Betriebssystem muß Informationen über aktuellen Status jedes Prozesses und jeder Ressource haben!

Hierzu werden 4 Typen von Tabellen verwaltet; und zwar für

- den Speicher,
- die E/A-Geräte,
- Dateien und
- Prozesse

Speichertabellen:

- Hauptspeicher des Systems wird von mehreren Prozessen gleichzeitig genutzt

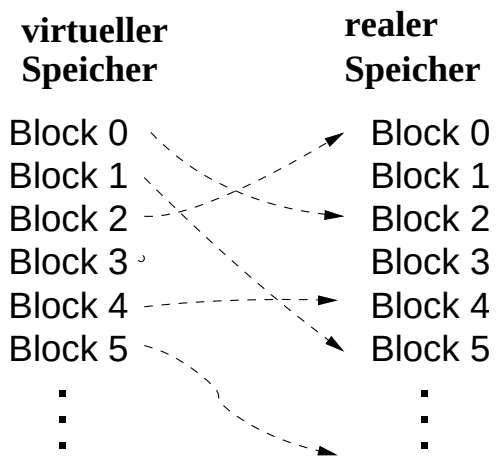
$\rightsquigarrow$ Aufteilung des Speichers, so daß Prozesse sich nicht gegenseitig stören können

$\Rightarrow$ Konzept des *virtuellen Speichers*:
für jeden Prozeß wird ein eigener (virtueller) Hauptspeicher angenommen

Kontrollstrukturen in Betriebssystemen

virtueller Speicher:

- Aufteilung des Speichers (des realen Hauptspeichers sowie des virtuellen Speichers) in gleichgroße Blöcke (Seiten, Kacheln, Frames, . . .)
- Blockgröße für den Hauptspeicher entspricht oft der Zugriffs-Blockgröße für Sekundärspeicher (Festplatten, Diskettenlaufwerke, . . .) oder einem Vielfachen davon; z.B. 1024 Bytes
- benutzte Blöcke des virtuellen Speichers werden durch Speichertabellen (Seiten-Kachel-Tabellen, . . .) den realen Hauptspeicherblöcken zugeordnet, z.B.:



Kontrollstrukturen in Betriebssystemen

virtueller Speicher (Forts.):

- Da der reale Hauptspeicher begrenzt ist, kann es vorkommen, daß Blöcke auf eine schnelle Platte ausgelagert werden, um für andere benötigte Blöcke Platz zu haben.

  Dann verweist die Speichertabelle auf den Auslagerungsbereich (Swap-Bereich) auf der Platte.

- Von einer virtuellen Hauptspeicheradresse werden
  - die höherwertigsten Bits zur Adressierung des entsprechenden Blocks über eine Speichertabelle

    und
  - die niederwertigsten Bits zur direkten Adressierung innerhalb des Blocks genutzt.

Anmerkung:
Real kann es sich auch um eine zweistufige Abbildung handeln: Der virtuelle Speicher eines Prozesses wird auf den virtuellen (Gesamt-)Speicher abgebildet und dieser dann auf den tatsächlich zur Verfügung stehenden Hauptspeicher.

Kontrollstrukturen in Betriebssystemen

E/A-Tabellen:

- Verwaltung von E/A-Geräten
- ein E/A-Gerät ist entweder verfügbar oder einem bestimmten Prozeß zugeordnet
- bei Benutzung eines E/A-Geräts weiß das Betriebssystem
  - den Status der E/A-Operation

    sowie
  - den Ort im Hauptspeicher, der als Ziel bzw. Quelle des E/A-Transfers genutzt wird.

Kontrollstrukturen in Betriebssystemen

Dateitabellen:

- Informationen über
  - die Existenz von Dateien,
  - über ihren Ort im Hintergrundspeicher,
  - ihren aktuellen Status (wird eine Datei gerade von einem Prozeß bearbeitet, . . .)

    sowie
  - weitere Attribute (Dateiname, Dateityp, Größe, Datum der letzten Änderung, Besitzer, Benutzerrechte, . . .).
- Bei Nutzung eines separaten File Management System (Dateiverwaltungssystem) übernimmt dieses viele dieser Aufgaben.

  Das Betriebssystem muß dann selbst entsprechend weniger über die Dateien im einzelnen wissen.

Kontrollstrukturen in Betriebssystemen

Prozeßtabellen

Für jeden Prozeß wird ein Prozeßkontrollblock (PCB) angelegt.

Ein PCB enthält u.a.

- die Prozeßidentifikation: die Prozeßnummer
- den Identifikator des Elternprozesses
- den zu dem Prozeß gehörigen Nutzernamen (genauer: die Nutzer-ID); also zu welchem Nutzer der Prozeß gehört

  damit kann z.B. beim Zugriff auf Dateien überprüft werden, ob der Nutzer die dafür notwendigen Rechte besitzt
- den aktuellen Prozeßzustand: running, not running, blocked, . . .
- die Inhalte der Prozessorregister und des Programmzählers (wenn der Prozeß gerade nicht in Ausführung ist)
- . . .

2.3 Prozeßkoordination

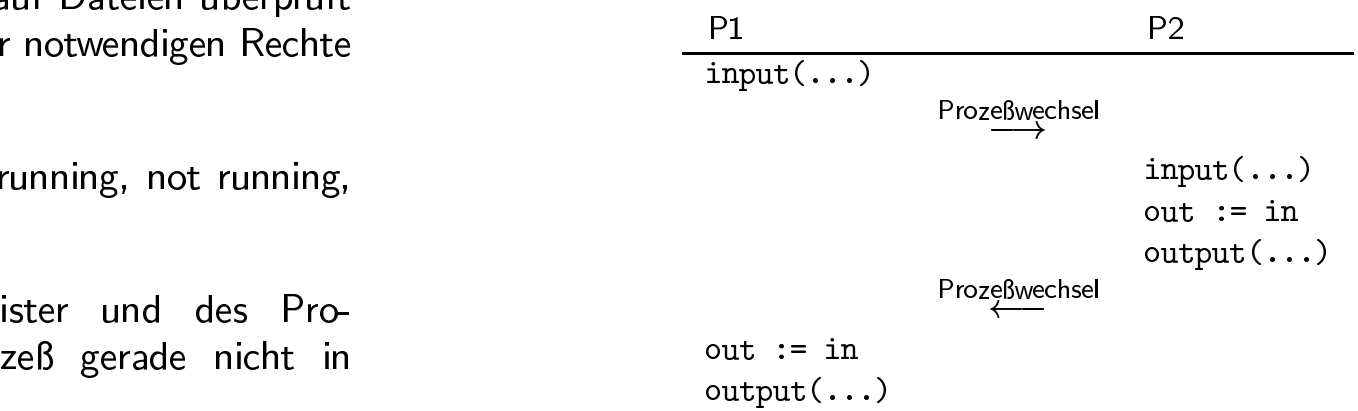
Problem der Nebenläufigkeit (d.h. parallele Abarbeitung von Prozessen, die sich gegenseitig beeinflussen können)

am Beispiel:

```
Prozess P1                      Prozess P2

Procedure echo;                 Procedure echo;
Var out, in: CHAR;              Var out, in: CHAR;
Begin                           Begin
  input(in, keyboard);            input(in, keyboard);
  out := in;                      out := in;
  output(out, display);           output(out, display);
End;                            End;
```

ein möglicher nebenläufiger Ablauf (Annahme: eine gemeinsame Tastatur und ein gemeinsames Display):



Bei Eingabe von „A“, „B“ auf der Tastatur wird „B“, „A“ auf dem Display ausgegeben!

Prozeßkoordination

noch schlimmer (?):

Wenn die Prozedur `echo` eine vom Betriebssystem zur Verfügung gestellte Prozedur ist, weil sie den Zugriff auf E/A-Geräte ermöglicht, wird sie nur einmal realisiert sein.

The Variablen `in` und `out` werden dann zu globalen Variablen, wenn zwei Prozesse diese Prozedure „gleichzeitig“ aufrufen:

P1		P2	Wert in `in`	Wert in `out`
`input(...)`			„A“	
	Prozeßwechsel $\longrightarrow$			
		`input(...)`	„B“	
		`out := in`	„B“	„B“
		`output(...)`	„B“	„B“
	Prozeßwechsel $\longleftarrow$			
`out := in`			„B“	„B“
`output(...)`			„B“	„B“

$\Rightarrow$ gleichzeitige Ausführung solcher „kritischer“ Prozeduren durch mehrere Prozesse muß verhindert werden!

Kritischer Bereich

Definition: 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.

$\rightsquigarrow$ Aufteilung von Programmen in kritische und unkritische Bereiche:

Solange ein Prozeß sich in einem kritischen Bereich befindet, darf sich kein anderer Prozeß in diesem kritischen Bereich befinden.

Wie vermeidet man, daß Prozesse sich gleichzeitig in ein- und demselben kritischen Bereich befinden?

$\rightarrow$ sogenannter wechselseitiger Ausschluß

Wechselseitiger Ausschluß

Anforderungen an 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.

   Problematische Situation:
   2 Prozesse A, B; 2 kritische Ressourcen a, b;
   beide Prozesse benötigen beide kritische Ressourcen

   Ablauf, z.B.:
   A belegt Ressource a,
   B belegt Ressource b,
   B möchte Ressource a $\rightarrow$ muß auf A warten
   A möchte Ressource b $\rightarrow$ muß auf B warten
   $\rightsquigarrow$ 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
   $\rightsquigarrow$ C würde beliebig lange warten müssen (starvation)!

Algorithmen für wechselseitigen Ausschluß

im folgenden betrachtet: 2 Softwarelösungen

1. Lösung: [Dekker, ca. 1965]

Es wird eine geschützte, aber gemeinsam genutzte globale Variable eingeführt (hier: `turn`), die anzeigt, welcher Prozeß in den kritischen Bereich eintreten darf.

Nur wenn die 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.

```
Prozess P0                    Prozess P1

. . .                         . . .
WHILE turn <> 0               WHILE turn <> 1
   DO  {nothing};                DO  {nothing};
< kritischer Bereich >;       < kritischer Bereich >;
turn := 1;                    turn := 0;
...                           ...
```

Algorithmen für wechselseitigen Ausschluß

1. Lösung (Forts.):

Probleme:

- Die Prozesse können nur abwechselnd den kritischen Bereich betreten.

  Auch bei Erweiterung auf mehrere 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 und wird danach bei einem nochmaligen Versuch, den kritischen Bereich zu betreten, blockiert.

  $\leadsto$ Verletzung der *progress*-Eigenschaft!

Algorithmen für wechselseitigen Ausschluß

1. Lösung (Forts.):

Verbesserungen:

1. Statt einer einzigen globalen Variable wird für jeden Prozeß ein Variable vorgesehen.

   Diese gibt den Status des Prozesses an, d.h., ob er Anspruch auf Eintritt in den kritischen Bereich hat (mit `true` für Anspruch, `false` sonst).

   Verwendungsweise:

   - Vor Eintritt in den kritischen Bereich müssen alle Variablen der anderen Prozesse kontrolliert werden.
   - Sind diese alle auf `false`, darf die eigene Variable auf `true` gesetzt und der kritische Bereich betreten werden.
   - Abschließend wird die eigene Variable wieder auf `false` gesetzt.

Algorithmen für wechselseitigen Ausschluß

Probleme (dieser 1. Verbesserung):

- Diese Prüfung vieler Variablen kostet Zeit.

  Aufgrund von Prozeßwechseln können zwei Prozesse „gleichzeitig“ feststellen, daß kein anderer Prozeß Anspruch auf Eintritt in den kritischen Bereich hat.

- Bei Abbruch des Prozesses direkt nach Setzen der eigenen Variable auf `true` ist der kritische Bereich dauerhaft blockiert.

2. Der kritische Bereich wird von einem „Schiedsrichter“ (z.B. als separater Prozeß realisiert) verwaltet.

   Zum Betreten des kritischen Bereichs zeigt jeder Prozeß seinen Wunsch in einer Variable an, die nur der Prozeß selbst und der Schiedsrichter verändern darf.

   Der Schiedsrichter kann nun einen Prozeß auswählen und ihm den Eintritt in den kritischen Bereich gewähren.

Algorithmen für wechselseitigen Ausschluß

2. Lösung: [Peterson]

Prinzip der „Höflichkeit“:

1. ich zeige an, daß *ich will*.

2. ich räume dem anderen den Vortritt ein, d.h. *du darfst*.

3. Falls *du willst und du darfst*, so lasse ich Dir den Vortritt (und warte), sonst betrete ich den kritischen Bereich.

$\rightarrow$ für jeden Prozeß 2 Variablen:

1. um *ich will* anzuzeigen
2. damit der andere Prozeß *du darfst* mitteilen kann

Funktioniert so für zwei Prozesse, läßt sich aber auch auf mehrere Prozesse erweitern.

Hardwarelösungen für wechselseitigen Ausschluß

1. Lösung: Unterbrechungsvermeidung

Bei Einprozessorsystemen können Prozesse nicht wirklich gleichzeitig ausgeführt werden

⇒ Prozeßwechsel während des Aufenthalts im kritischen Bereich verursacht die Probleme

⇝ Unterbrechungen (also Prozeßwechsel) dort unmöglich machen

Einführung besondere (Maschinen-) Befehle, die Unterbrechungen ausschließen

⇝ Programmmuster:

```
...
< unkritischer Bereich >;
< verbiete Unterbrechungen >;
< kritischer Bereich >;
< erlaube Unterbrechungen >;
< unkritischer Bereich >;
...
```

Achtung: funktionert nicht bei Multiprozessorsystemen!

Hardwarelösungen für wechselseitigen Ausschluß

2. Lösung: spezielle Maschinenbefehle

Wichtig: Maschinenbefehle sind nicht unterbrechbar!

Spezieller Maschinenbefehl, der „in einem Schritt“ den Wert einer Variablen (Register) testet, abhängig davon einen Wert (`true` oder `false`) zurückgibt und die Variable auf einen festen Wert setzt:

`testset` : testet und setzt die Variable i:

- i=0 → i:=1 und Testergebnis `true`
- i=1 → i:=1 und Testergebnis `false`

Realisierung des wechselseitigen Ausschlusses mit `testset`:

- bei Testergebnis `true` darf der kritsche Bereich betreten werden,
- bei `false` muß gewartet und weitergetestet werden.

```
...
WHILE not testset(i) DO  { nothing };
< kritscher Bereich >;
i := 0;
...
```

Semaphor

Synchronisationskonzept für Prozesse
typischerweise in Programmiersprachen eingebettet

Idee:
Prozesse kommunizieren über Signale;
Semaphore sind spezielle Variablen, die diese Signale übertragen

Ein Semaphor kann entweder

- die Werte 0 und 1 annehmen $\rightarrow$ binäres Semaphor
- die Werte $0, \ldots, n$ annehmen $\rightarrow$ Zählsemaphor
  (genauer: bei Zählsemaphoren können auch negative Werte auftreten)

Auf einem Semaphor können 3 Operationen ausgeführt werden:

- Ein Semaphor kann mit einem nichtnegativen Wert initialisiert werden.
- Die Operation `wait` dekrementiert einen Semaphorwert; ein negativer Wert blockiert die wait-Operation.
- Die Operation `signal` inkrementiert den Semaphorwert, ggfs. wird dadurch die Blockierung der `wait`-Operation aufgehoben.

Semaphor

Programmiertechnische Nutzung von Semaphoren

Sei `s` ein Semaphor (Realisierung folgt noch);
Programmstück mit kritischem Bereich:

```
...
wait(s);
< kritischer Bereich >;
signal(s);
...
```

Einsatz von Semaphoren

- zur Realisierung eines wechselseitigen Ausschlusses

  $\rightsquigarrow$ binäres Semaphor

- Verwaltung einer begrenzten Anzahl von Betriebsmitteln (Ressourcen

  $\rightsquigarrow$ Zählsemaphor

Semaphor

Realisierung eines binären Semaphor

```
TYPE binarysemaphore = RECORD
                         value: (0,1);
                        queue: LIST OF process;
                       END;
VAR  s: binarysemaphore;
```

Operationen:

- `waitB(s)`:

```
IF s.value =1
   THEN
      s.value := 0;
   ELSE BEGIN
      <plaziere diesen Prozess in s.queue>;
      <blockiere diesen Prozess>;
   END;
```

- `signalB(s)`:

```
IF s.queue is empty
   THEN
     s.value := 1;
   ELSE BEGIN
     <entnehme einen Prozess aus s.queue>;
     <setze diesen Prozess auf die ready-Liste>;
   END;
```


Semaphor

Realisierung eines Zählsemaphors

```
TYPE semaphore = RECORD
                   count: INTEGER;
                   queue: LIST OF process;
                  END;
VAR  s: semaphore;
```

Semaphor besteht aus der Variablen, die den Semaphorwert enthält, und einer Warteschlange (Liste) für Prozesse.

Realisierung der Operationen:

- `wait(s)`:

```
s.count := s.count - 1;
IF s.count < 0 THEN BEGIN
   <plaziere diesen Prozess in s.queue>;
   <blockiere diesen Prozess>;
   END;
```

- `signal(s)`:

```
s.count := s.count + 1;
IF s.count =< 0 THEN BEGIN
   <entnehme einen Prozess aus s.queue>;
   <setze diesen Prozess auf die ready-Liste>;
END;
```

2.4 Deadlocks

Definition: Deadlock (Verklemmung)

Ein Deadlock ist die dauerhafte Blockierung einer Menge M von Prozessen, die gemeinsame Systemressourcen S nutzen oder miteinander kommunizieren ($|M| > 2$, $|S| > 2$).

typisches Beispiel zur Entstehung eines Deadlocks:
(2 Prozesse P und Q, 2 Betriebsmittel A und B)

Prozeß P	Prozeß Q
...	...
get A	get B
...	...
get B	get A
...	...
release A	release B
...	...
release B	release A
...	...

Problem:
Es gibt zeitliche Abläufe, bei denen die Prozesse nach einigen Schritten wechselseitig aufeinander warten
— weil beide Prozesse zwischenzeitlich beide Betriebsmittel exklusiv haben wollen.

Deadlocks

2 prinzipiell unterschiedliche Lösungsstrategien:

- *Vermeidung* von Deadlocks
- *Erkennung* und Auflösung von Deadlocks

Vermeidung von Deadlocks:

1. Keine gleichzeitige Beanspruchung mehrerer Betriebsmittel durch einen Prozeß:

   also: ... get A ... release A ... get B ... release B ...

   Das ist aber nicht immer möglich!

2. Wenn ein Prozeß mehrere Betriebsmittel gleichzeitig benötigt, muß er diese auf einmal belegen; die Freigabe kann nach und nach erfolgen.

   also: get A,B,C ... release C ... release A,B ...

   Problem:
   Alle benötigten Betriebsmittel müssen vorab bekannt sein. Gegebenenfalls werden so auch zu viele Betriebsmittel belegt, nur weil die Möglichkeit besteht, daß sie benötigt werden könnten.

Deadlocks

Erkennung von Deadlocks

notwendige Vorraussetzungen für einen Deadlock:

1. *Mutual Exclusion:*
   Es gibt mindestens 2 Ressourcen, die nur von einem Prozeß gleichzeitig genutzt werden.
2. *Hold and Wait:*
   Ein Prozeß muß eine Ressource behalten, während er auf eine weitere Ressource wartet.
3. *No Preemption* (keine Unterbrechung):
   Eine Ressource kann einem Prozeß, der sie behält, nicht wieder entzogen werden.

dann:

4. *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.