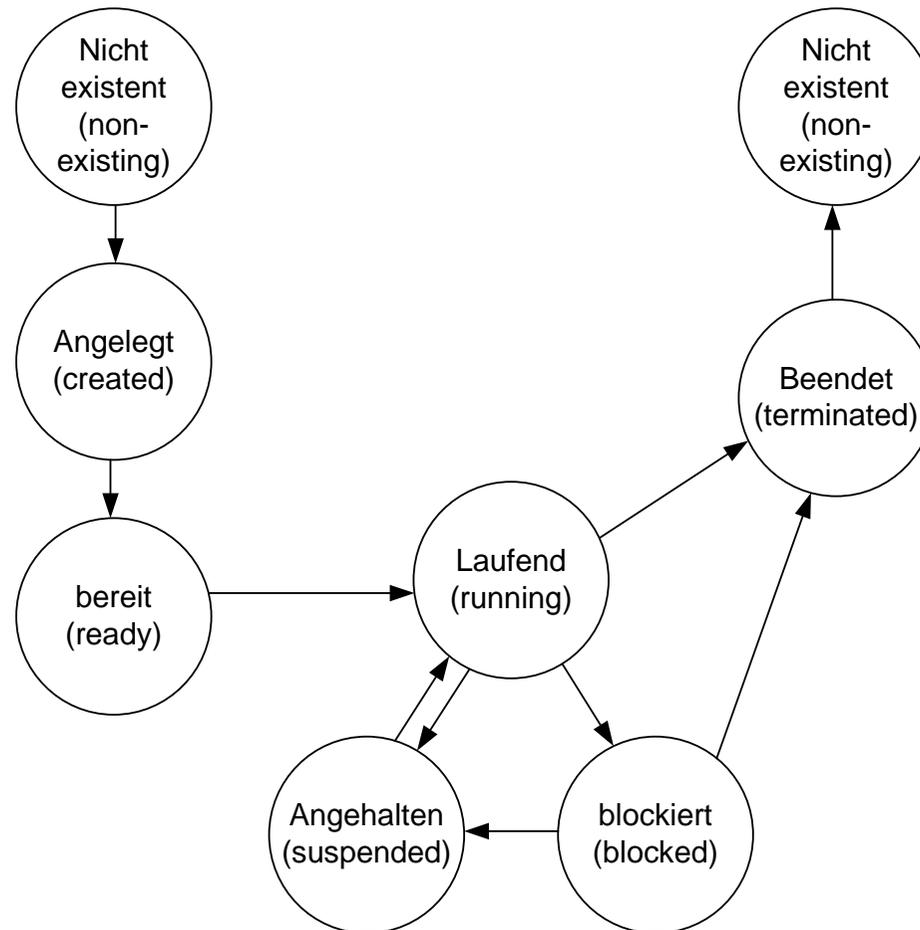
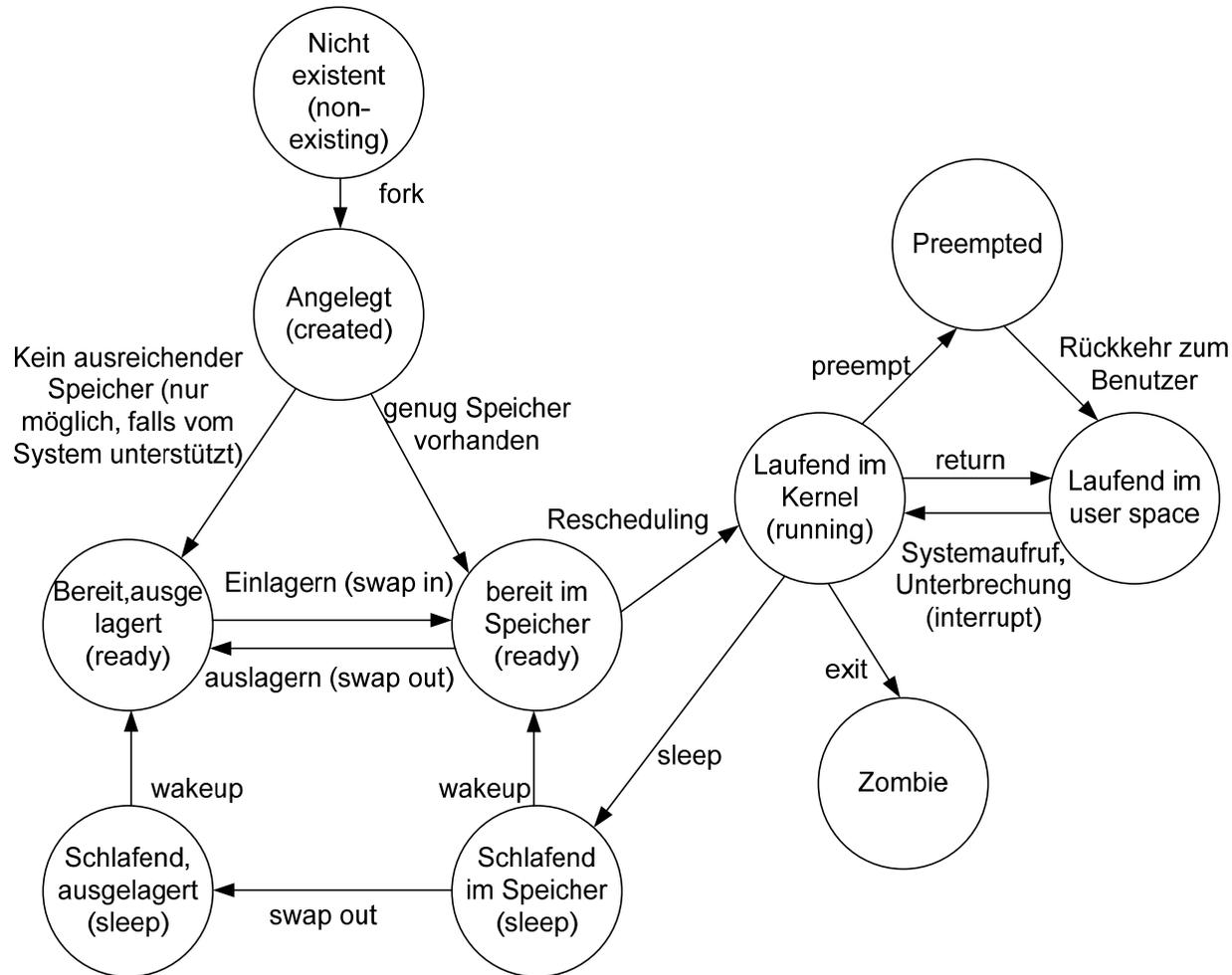


Prozesszustände (allgemein)



Prozeßzustände in Unix





Fragen bei der Implementierung

- Welche Betriebsmittel sind notwendig?
- Welche Ausführungszeiten besitzen einzelne Prozesse?
- Wie können Prozesse kommunizieren?
- Wann soll welcher Prozess ausgeführt werden?
- Wie können Prozesse synchronisiert werden?



Klassifikation von Prozessen

- periodisch vs. aperiodisch
- statisch vs. dynamisch
- Wichtigkeit der Prozesse (kritisch, notwendig, nicht notwendig)
- speicherresident vs. verdrängbar
- Prozesse können auf
 - einem Rechner (Pseudoparallelismus)
 - einem Multiprozessorsystem mit Zugriff auf gemeinsamen Speicher
 - oder auf einem Multiprozessorsystem ohne gemeinsamen Speicher ausgeführt werden.



Nebenläufigkeit

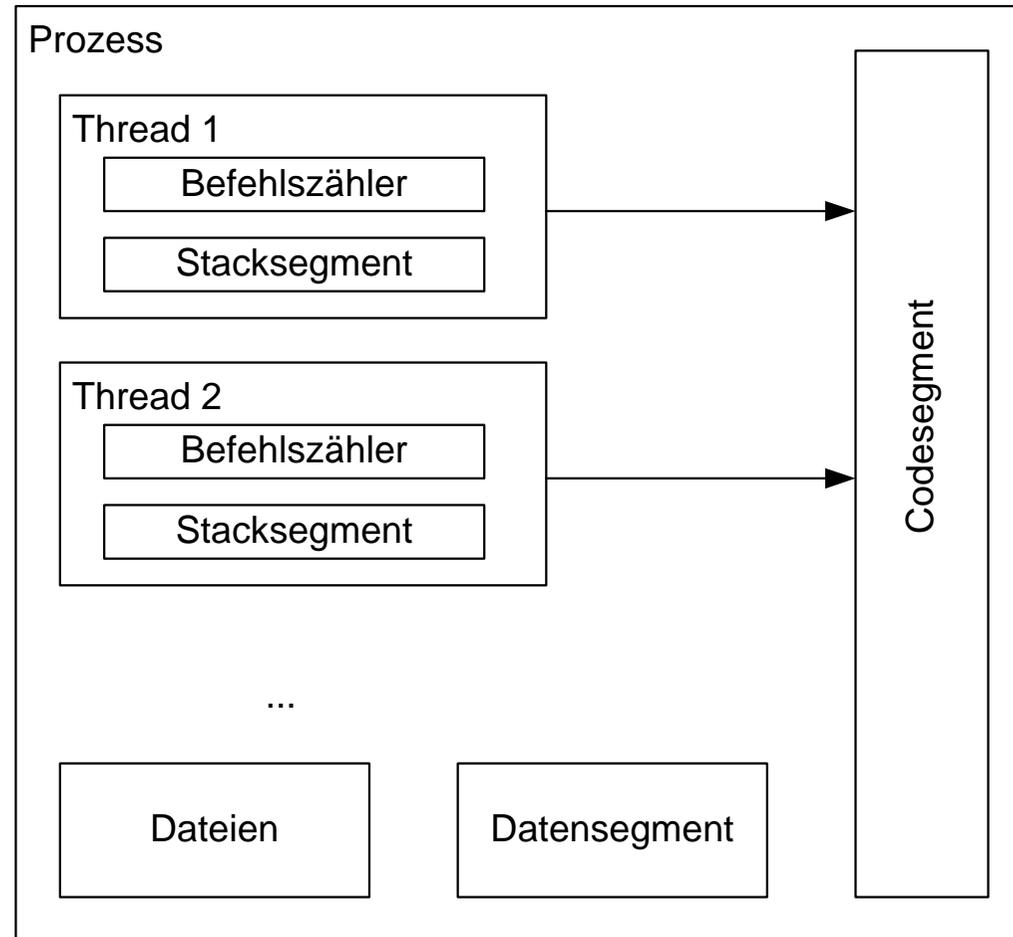
Threads



Leichtgewichtige Prozesse (Threads)

- Der Speicherbedarf von Prozessen ist in der Regel groß (CPU-Daten, Statusinformationen, Angaben zu Dateien und EA-Geräten...).
 - Bei Prozesswechsel müssen die Prozessdaten ausgetauscht werden \Rightarrow hohe Systemlast, zeitaufwendig.
 - Viele Systeme erfordern keine komplett neuen Prozesse.
 - Vielmehr sind Programmabläufe nötig, die auf den gleichen Prozessdaten arbeiten.
- \Rightarrow Einführung von Threads

Threads





Prozesse vs. Threads

- Verwaltungsaufwand von Threads ist deutlich geringer
- Effizienzvorteil: bei einem Wechsel von Threads im gleichen Prozessraum ist kein vollständiger Austausch des Prozesskontextes notwendig.
- Kommunikation zwischen Threads des gleichen Prozesses kann über gemeinsamen Speicher erfolgen.
- Zugriffe auf den Speicherbereich anderer Prozesse führen zu Fehlern.
- Probleme bei Threads: durch die gemeinsame Nutzung von Daten kann es zu Konflikten kommen.



Nebenläufigkeit

Probleme



Probleme

- **Race Conditions:**
 - Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
 - Lösung: Einführung von **kritischen Bereichen** und **wechselseitiger Ausschluss**.
- **Starvation (Aussperrung):**
 - Situation, in der ein Prozess unendlich lange auf ein Betriebsmittel wartet. Wichtig: sinnvolle Realisierung von Warteschlangen bei der Betriebsmittelvergabe, z.B. Prioritätenbasierte Warteschlangen
- **Priority Inversion (Prioritätsinversion):**
 - Wichtige Prozesse können durch unwichtigere Prozesse, die Betriebsmittel belegt haben verzögert werden, genaue Problemstellung siehe Kapitel Scheduling



Bedingungen an Lösung für wechselseitigen Ausschluss

- An eine gute Lösung für den wechselseitigen Ausschluss (WA) können insgesamt vier Bedingungen gestellt werden:
 1. Es dürfen niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
 2. Es dürfen keine Annahmen über die Geschwindigkeit oder Anzahl der Prozessoren gemacht werden.
 3. Kein Prozess darf außerhalb von kritischen Regionen andere Prozesse blockieren.
 4. Kein Prozess soll unendlich auf das Eintreten in den kritischen Bereich warten müssen.



Kritische Bereiche

- Um einen solchen Bereich zu schützen, sind Mechanismen erforderlich, die ein gleichzeitiges Betreten verschiedener Prozesse bzw. Prozeßklassen dieser Bereiche verhindern.
 - Darf maximal nur ein Prozess gleichzeitig auf den kritischen Bereich zugreifen, so spricht man vom **wechselseitigen Ausschluss**.
 - Wird verhindert, daß mehrere (unterschiedlich viele) Instanzen unterschiedlicher Prozeßklassen auf den Bereich zugreifen, so entspricht dies dem Leser-Schreiber-Problem (so dürfen beispielsweise mehrere Instanzen der Klasse `Leser` auf den Bereich gleichzeitig zugreifen, Instanzen der Klasse `Schreiber` benötigen den exklusiven Zugriff).
- Aus dem Alltag sind diverse Mechanismen zum Schutz solcher Bereiche bekannt:
 - Signale im Bahnverkehr
 - Ampeln zum Schutz der Kreuzung
 - Schlösser für einzelne Räume
 - Vergabe von Tickets

Falsche Lösung: Verwendung einer globalen Variable

```
bool block = false; //global variable
```

```
...  
while(block){}; //busy wait  
block=true;  
... critical section ...  
block=false;  
...
```

- Die obige Implementierung ist nicht korrekt,
 - da der Prozess direkt nach dem while-Abschnitt unterbrochen werden könnte und evtl. dann fortgesetzt wird, wenn block bereits durch einen anderen Prozess belegt ist.
 - Zudem ist die Lösung ineffizient (busy wait)



1.Möglichkeit: Peterson 1981 (Lösung für zwei Prozesse)

```
int turn=0;
boolean ready[2];
ready[0]=false;
ready[1]=false;
```

Deklaration globale Variablen

```
...
ready[0]=true;
turn = 1;
while(ready[1]
      && turn==0); //busy waiting
... critical section ...
ready[0]=false;
...
```

Prozess 0

```
...
ready[1]=true;
turn = 0;
while(ready[0]
      && turn==1); //busy waiting
... critical section ...
ready[1]=false;
...
```

Prozess 1

- Das Problem der Realisierung für n Prozesse ist unter dem Bakery Algorithmus bekannt.



2. Möglichkeit: Ausschalten von Unterbrechungen zum WA

- Prozesswechsel beruhen immer auf dem Eintreffen einer Unterbrechung (Interrupt) (z.B. neues Ereignis, Ablauf einer Zeitdauer)
- Die einfachste Möglichkeit einen Kontextwechsel zu verhindern ist das Ausschalten von Unterbrechungen während sich ein Prozess im kritischen Bereich befindet.
- Vorteile:
 - einfach zu implementieren, keine weiteren Konzepte sind nötig
 - schnelle Ausführung
- Nachteile:
 - Für Multiprozessorsysteme ungeeignet
 - Keine Gerätebehandlung während der Sperre
 - Lange Sperrern kritisch bei Echtzeitanwendungen



3. Möglichkeit: Semaphor

- Semaphor (griechisch von Zeichenträger, Signalmast) wurden von Edsger W. Dijkstra im Jahr 1965 eingeführt.
- Ein Semaphor ist eine Datenstruktur, bestehend aus einer Zählvariablen, sowie den Funktionen `down()` oder `wait()` (bzw. `P()`, von probeer te verlagen) und `up()` oder `signal()` (bzw. `V()`, von verhogen).

```
Init(Semaphor s, Int v)    V(Semaphor s)    P(Semaphor s)
{                          {                          {
  s = v;                   s = s+1;          while (s <= 0) {} ; // Blockade, unterschiedliche Implementierungen
}                          }                          s = s-1 ;          // sobald s>0 belegt eine Ressource
                           }                          }
```

- Bevor ein Prozess in den kritischen Bereich eintritt, muss er den Semaphor mit der Funktion `down()` anfordern. Nach Verlassen wird der Bereich durch die Funktion `up()` wieder freigegeben.
- **Wichtige Annahme:** die Ausführung der Funktionen von `up` und `down` darf nicht unterbrochen werden (atomare Ausführung), siehe Realisierung
- Solange der Bereich belegt ist (Wert des Semaphors ≤ 0), wird der aufrufende Prozess blockiert.



Beispiel: Bankkonto

- Durch Verwendung eines gemeinsamen Semaphors `semAccount` kann das Bankkonto auch beim Zugriff von zwei Prozessen konsistent gehalten werden:

Prozess A

```
P(semAccount);  
x=readAccount(account);  
x=x+500;  
writeAccount(x,account);  
V(semAccount);
```

Prozess B

```
P(semAccount);  
y=readAccount(account);  
y=y-200;  
writeAccount(y,account);  
V(semAccount);
```

- Zur Realisierung des wechselseitigen Ausschlusses wird ein binärer Semaphor mit zwei Zuständen: 0 (belegt), 1 (frei) benötigt. Binäre Semaphore werden auch *Mutex* (von *mutal exclusion*) genannt.



Erweiterung: zählender Semaphore

- Nimmt ein Wert auch einen Wert größer eins an, so wird ein solch ein Semaphor auch als **zählender Semaphor** (counting semaphore) bezeichnet.
- Beispiel für den Einsatz von zählenden Semaphoren: In einem Leser-Schreiber-Problem kann die Anzahl der Leser aus Leistungsgründen z.B. auf 100 gleichzeitige Lesezugriffe beschränkt werden:

```
semaphore sem_reader_count;  
init(sem_reader_count, 100);
```

- Jeder Leseprozess führt dann folgenden Code aus:

```
P(sem_reader_count);  
read();  
V(sem_reader_count);
```

Klausur WS06/07 - Nebenläufigkeit

Prozess: *tankendes Auto*

```
fahreInWartebereich();
```

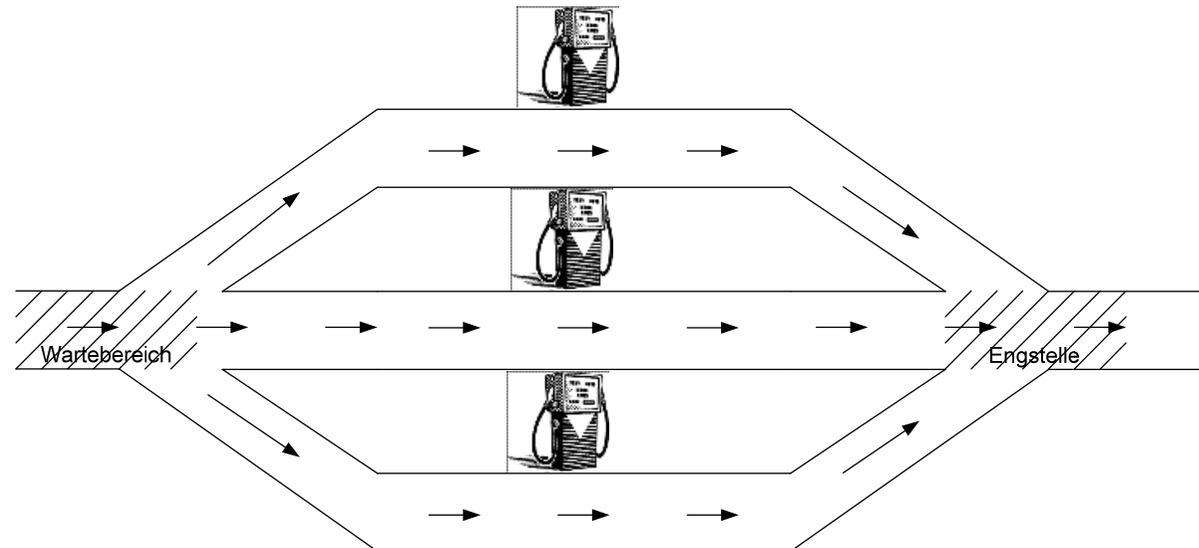
```
fahreAnZapfsaeule();
```

```
tanke();
```

```
bezahle();
```

```
fahreInEngstelle2();
```

```
verlasseEngstelle2();
```



- Geben Sie die notwendigen Semaphore (mitsamt Initialisierung) an, um das gegebene Problem zu lösen. Beispiel: `semAuto(1)` würde bedeuten, Sie verwenden einen Semaphor `semAuto`, der mit 1 initialisiert ist.
- Ergänzen Sie den folgenden Autoprozess mit passenden `up()` und `down()`-Methoden, um Kollisionen zu vermeiden. Achten Sie darauf, dass es zu keiner Verklemmung kommt. **Anmerkung:** Es muss nicht an jeder freien Stelle Code eingefügt werden. Beispiel: 1: `down(semAuto); up(semAuto);` bedeutet das Einfügen der beiden Operationen in Zeile 1.

Klausur WS06/07 - Nebenläufigkeit

Prozess: *tankendes Auto*

```
fahreInWartebereich();
```

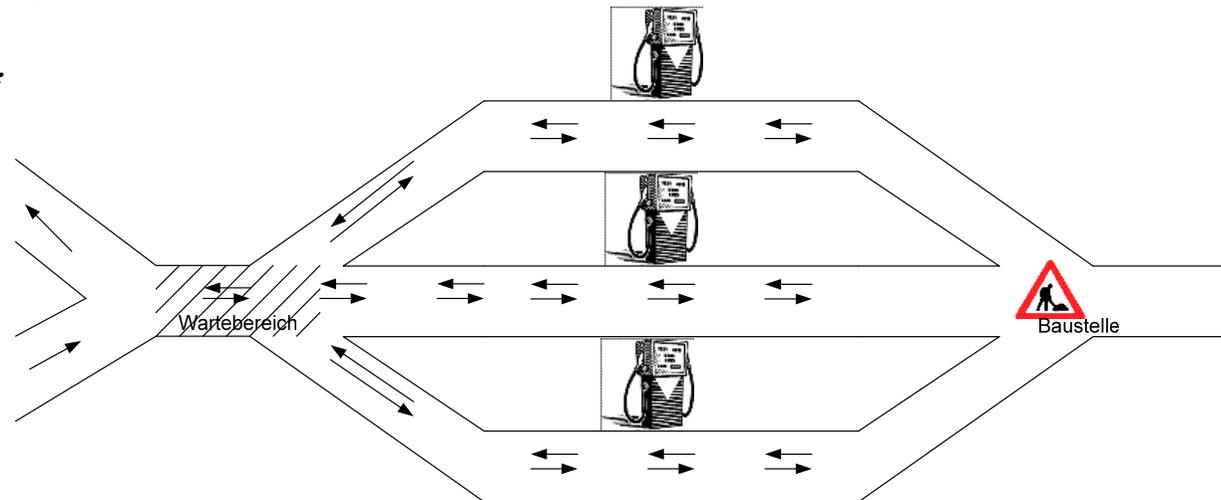
```
fahreAnZapfsaeule();
```

```
tanke();
```

```
bezahle();
```

```
fahreInEngstelle2();
```

```
verlasseEngstelle2();
```



- c) Aufgrund einer Baustelle ist die Ausfahrt blockiert (siehe Abbildung), so dass die Wartebereich sowohl zur Einfahrt, als auch zur Ausfahrt genutzt werden muss. Ergeben sich notwendige Änderungen im Vergleich zur Lösung der Aufgabe b) und wenn ja welche?



Fragestellung: Invers zählender Semaphor

- Aufgabenstellung: Implementierung des Leser-Schreiber-Problems mit Schreiber-Priorität
- Erläuterung:
 - Auf einen Datensatz können mehrere Leser gleichzeitig oder aber ein Schreiber zugreifen.
 - Sobald ein Schreiber den Schreibwunsch äußert, soll kein weiterer Leser (oder Schreiber) mehr auf den Datensatz zugreifen können. Zum Zeitpunkt der Signalisierung bestehende Lesevorgänge können regulär beendet werden, erst danach darf der Schreiber auf die Daten zugreifen.
- Problem: Häufig wird versucht das Problem mit einem „*invers zählenden Semaphor*“ zu lösen, also einem Semaphor, der bei 0 freigibt und sonst blockiert.
- Wie geht es richtig?



Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.
⇒ Die Funktionen $up()$ und $down()$ dürfen nicht unterbrochen werden, da sonst der Semaphor selbst inkonsistent werden kann.
- Funktionen die nicht unterbrechbar sind, werden **atomar** genannt.
- Realisierungsmöglichkeiten:
 1. Kurzfristige Blockade der Prozeßwechsel während der Bearbeitung der Funktionen $up()$ und $down()$. Implementierung durch Verwendung einer Interrupt-Sperre, denn sämtliche Prozesswechsel werden durch **Unterbrechungen (Interrupts)** ausgelöst.
 2. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den In-halt (typ. 0 für frei, 1 für belegt) eines Speicherwortes in ein Register und schreibt ununterbrechbar einen Wert (typ. $\neq 0$, z.B. 1 für belegt) in das Speicherwort.
 3. **Spinlock**: Programmieretechnik auf der Basis von Busy Waiting. Vorteil: Unabhängig vom Betriebssystem und auch in Mehrprozessorsystemen zu implementieren, jedoch massive Verschwendung von Rechenzeit. Im Gegensatz dazu können die Lösungen von 1 und 2 mit Hilfe von Warteschlangen sehr effizient realisiert werden.



Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung und Freigabe des kritischen Bereiches durch den Programmierer
- Vergißt der Entwickler z.B. die Freigabe des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
 - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozess zugreifen kann.
 - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
 - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
 - Die Signalisierung ist innerhalb des Monitors festgelegt, der Programmierer muss sie nicht selbstständig implementieren.



Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung:** normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphore implementiert. Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

```
public class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }

    synchronized public void up() {
        value++;
        if(value==1) notify();
    }

    synchronized public void down() {
        while(value==0) wait();
        value- -;
    }
}
```