

Betriebssysteme und Netzwerke

Vorlesung 8

Artur Andrzejak

Umfragen: <https://pingo.coactum.de/301541>

Wiederholung Vorlesung 7

Test and Set Lock, TSL?

Race Condition?

clone?

WA – Lösung von Peterson?

aktives Warten?

Wechselseitiger Ausschluss?

Interrupts und Threadwechsel?

Sperrvariablen - Locks

Bitcoins zählen?

Swap-Befehl XCHG?

Wechselseitiger Ausschluss: Lösungen mit **aktivem Warten**

Sperrvariablen - Locks

- ▶ Idee: Wir nutzen einen „Token“, dessen Besitz anzeigt, dass ein Thread in die kritische Region eintreten darf
- ▶ Man nennt solche Tokens **Sperren** bzw. **Locks**
 - ▶ Def.: Variablen, die anzeigen, dass ein Prozess in der kritischen Region ist, und kein anderer eintreten darf

```
while (TRUE) {  
    erlange die Sperre – enter_region  
    führe Code in der kritischen Region aus  
    setze die Sperre frei – leave_region  
    restlicher Code  
}
```

Achtung: die While-Schleife ist eine abstrakte Darstellung, und soll nur ausdrücken, dass die Abarbeitung von kritischen und nicht-kritischen Regionen sich abwechselt (so sieht aber ein Programm ggf. nicht aus)

Probleme der Implementierung

- ▶ Der Aufruf **enter_region** ist blockierend - keine Rückkehr, bis die Sperre erlangt ist
- ▶ Der Aufruf **leave_region** ist nicht blockierend
- ▶ Die Bedingung, dass die Sequenz „LOAD und danach STORE“ atomar ausgeführt wird, ist auf moderner HW i.A. nicht garantiert
- ▶ Macht die Implementierung kompliziert

Hardware-Lösungen - „Test and Set Lock“

▶ **TSL** RX, LOCK

- ▶ Inhalt des Speicherwortes **lock** wird ins Register RX eingelesen und ein Wert ungleich 0 wird an die Adresse von **lock** abgelegt
- ▶ Das Lesen und Schreiben bei TSL ist garantiert atomar: Zugriff auf Speicher ist während der Ausführung gesperrt!

▶ **enter_region**:

- ▶ **TSL** RX, LOCK | kopiere Sperrvariable, sperre mit != 0
- ▶ **CMP** RX, #0 | war die Sperrvariable 0?
- ▶ **JNE** **enter_region** | wenn nicht 0, war gesperrt => Schleife
- ▶ **RET** | Rücksprung, d.h. k.R. wird nun betreten

▶ **leave_region**:

- ▶ **MOVE** LOCK, #0 | speichere 0 in die Sperrvariable
- ▶ **RET** | Rücksprung

Hardware-Lösungen – Befehl Swap

▶ **XCHG** RX, LOCK

- ▶ Inhalt des Speicherwortes lock und des Registers RX werden ausgetauscht
- ▶ Auch diese Operation ist atomar

▶ **enter_region**:

- ▶ `MOVE RX, #1` | speichere 1 im Register RX
- ▶ **XCHG** RX, LOCK | vertausche Inhalte von lock und RX
- ▶ `CMP RX, #0` | war die Sperrvariable 0?
- ▶ `JNE enter_region` | wenn nicht 0, war gesperrt => Schleife
- ▶ `RET` | Rücksprung, d.h. k.R. wurde betreten

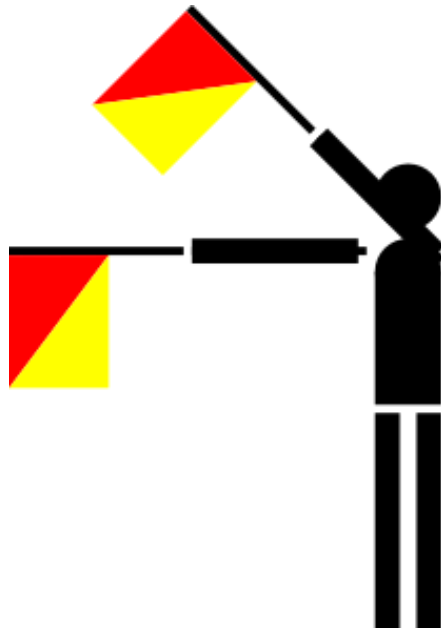
▶ **leave_region**:

- ▶ `MOVE LOCK, #0` | speichere 0 in die Sperrvariable
- ▶ `RET` | Rücksprung

Probleme des **aktiven Wartens**

- ▶ Verschwendung von Prozessorzeit
- ▶ Kann zu sog. **Prioritätsumkehr** führen
 - ▶ Prozess **H** mit hoher Priorität, Prozess **L** mit niedriger: **H** soll immer laufen, wenn er rechenbereit ist
 - ▶ Prozess **L** wird unterbrochen, wenn das der Fall ist
 - ▶ Angenommen, **L** befindet sich in der kritischen Region und **H** wird rechenbereit
 - ▶ **H** beginnt mit dem aktiven Warten
 - ▶ Aber **L** kommt nie zum Zuge, während **H** läuft!
- ▶ Bekanntes Beispiel: The Mars Pathfinder Problem
 - ▶ http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html
 - ▶ Video: <https://www.youtube.com/watch?v=lyx7kARrGeM>

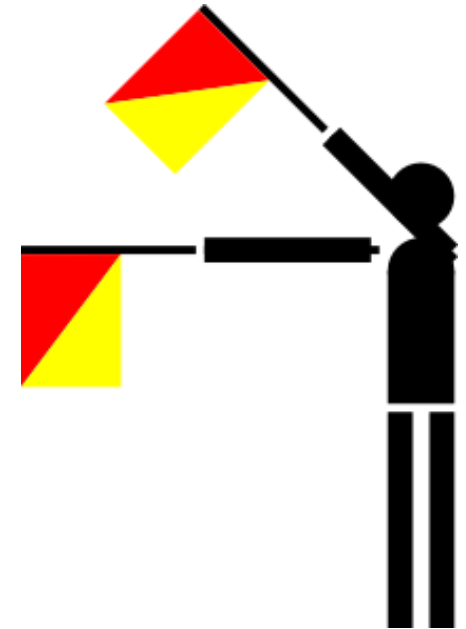
Semaphore: Definition und Anwendungen



Italienisch
„semaforo“ = Ampel

Semaphore

- ▶ **Semaphore**: ein Ansatz von E. W. Dijkstra in 1965
- ▶ Erlauben Kontrolle des Zugriffs auf eine Ressource mit mehreren Instanzen
 - ▶ z.B. Mehrere Drucker im Pool
- ▶ Allgemein: Ein Semaphor ist eine ganzzahlige Variable s , zusammen mit zwei speziellen Operationen auf s



Motivation: Lecture 2, unit 1: Introduction to Semaphores
<https://www.youtube.com/watch?v=KZU4ANBoLTY>
ab 0:25 bis 2:30 (min:sec), [07a]

Semaphore – Operationen

- ▶ Operationen auf einem Semaphor **S** sind:
wait() und **signal()**
- ▶ **wait(S)** oder **down(S)** - „Reservieren“ / „Probieren“
 1. **Warten**, solange $S \leq 0$ ist
 2. Sobald $S > 0$, **dekrementiere** S und **verlasse** wait()
- ▶ **signal(S)** oder **up(S)** – „Freigeben“
 - ▶ **Inkrementiere** S und **verlasse** signal() (sofort)
- ▶ **S** sollte beim Erzeugen auf Wert ≥ 0 gesetzt werden

Semaphore – Anwendungen /1

- Implementation von wechselseitigen Ausschluss?

Global: Erzeuge ein Semaphor **S** und **setze es auf 1**

```
while (TRUE) {  
    wait (S);  
    // kritische Region  
    signal (S);  
    // nicht-kritische R.  
}
```

Prozess A

```
while (TRUE) {  
    wait (S);  
    // kritische Region  
    signal (S);  
    // nicht-kritische R.  
}
```

Prozess B

Semaphore – Anwendungen /2

- ▶ Wie implementiert man die **Abhängigkeit**:
Codeblock **B** (Prozess P2) darf erst nach dem
Codeblock **A** (Prozess P1) ausgeführt werden?
- ▶ **Global**: Erzeuge und setze ein Semaphor **S** auf 0

Prozess P1	Prozess P2
A ; signal (S);	wait (S); B

- ▶ N.B.: Ein Semaphor, dessen Variable nur 0 oder 1 sein kann, wird als ein **binärer S**. oder **Mutex** bezeichnet
 - ▶ Engl. mutex locks = locks for mutual exclusion

Semaphore vs. Locks

- ▶ **Locks** (oder **Sperrvariablen**) erlauben es nur, den Eintritt in die kritische Region zu blockieren und wieder freizugeben
- ▶ Semaphore können mehr machen ...
 - ▶ 1. Man kann mit Semaphoren **mehrere Instanzen von den Ressourcen** gleichen Typs verwalten
 - ▶ Z.B. $n > 2$ Prozesse verwenden zwei (2) Drucker
 - ▶ 2. Man kann **Abhängigkeiten der Codeausführung** umsetzen
 - ▶ Code B von Thread 2 wird garantiert nach Code A von Thread 1 ausgeführt

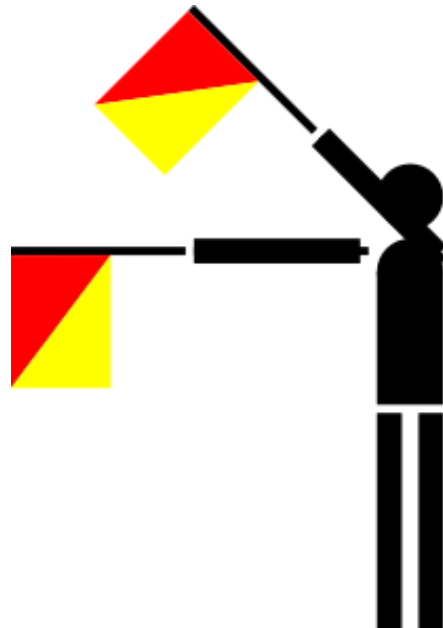
Semaphore: Welche Aussagen sind korrekt?

- ▶ A. Ein Thread muss immer zuerst ein `wait()` aufrufen, bevor er das erste Mal ein `signal()` aufrufen kann
- ▶ B. Ein Semaphor wird im BS einfacher implementiert als ein Lock
- ▶ C. Beim Aufruf von `wait()` kann ein Prozess blockiert werden, beim Aufruf von `signal()` nicht
- ▶ D. Der **TSL**-Befehl läuft garantiert un-unterbrechbar, und schreibt einen Wert in den Speicher S vor dem Lesen aus S

Empfohlene Videos

- ▶ Lecture 2, unit 1: Introduction to Semaphores
 - ▶ <https://www.youtube.com/watch?v=KZU4ANBoLTY>
- ▶ The Santa Claus Problem - Thread Synchronization
 - ▶ <https://www.youtube.com/watch?v=pqO6tKN2lc4>
- ▶ Section 1: Module 2: Part 7: Java Semaphore
 - ▶ <https://www.youtube.com/watch?v=UoaZTkot6-g>

Semaphore: Implementierung



Italienisch
„[semaforo](#)“ = Ampel

Semaphore mit aktivem Warten /1

- ▶ Wie können wir Semaphore mit **aktiven Warten** (d.h. ineffizient) implementieren? („Python-Pseudocode“)

wait (S):

repeat:

if $S > 0$:

S--

break

return

signal (S):

S++

return

- ▶ Problem: So würde das nicht funktionieren - wir müssen die Race Conditions vermeiden!
- ▶ Welche Race Conditions sind möglich?

Semaphore mit aktivem Warten /2

- ▶ Welche Race Conditions sind möglich?

wait (S):

repeat:

if $S > 0$:

$S--$

break

return

1. „Verlorenes“
Dekrement

signal (S):

$S++$

return

2. „Verlorenes“
Inkrement

3. Unterbrechung hier: S
könnte negativ werden

- ▶ Wir betrachten zur Vereinfachung eine Single-Core Maschine – die **Ununterbrechbarkeit reicht**
- ▶ Bei Multi-Core CPUs / Multiprozessor-Maschinen muss man zusätzlich den **Zugriff auf Speicherbus sperren**

Semaphore mit aktivem Warten /3

- ▶ Welche Codeteile müssen ununterbrechbar sein?
- ▶ Wir bezeichnen mit **[..]** Codeteile, die atomar (ohne Unterbrechung) ausgeführt werden müssen

wait (S):

repeat:

[if S > 0:

S--]

break

return

signal (S):

[S++]

return

Semaphore mit aktivem Warten /4

- ▶ Wie können wir **[..]** implementieren?
 - ▶ Hinweis: Wir betrachten zur Vereinfachung eine Single-Core Maschine – die **Ununterbrechbarkeit** reicht

=> Interrupts ein-/ausschalten

wait (S):
repeat:
 disable_interrupt
 if S > 0:
 S--
 enable_interrupt
 break
enable_interrupt
return

signal (S):
 disable_interrupt
 S++
 enable_interrupt
return

Semaphore mit aktivem Warten /5

- ▶ Geht es auch ohne Interrupts?
 - ▶ Bei Multi-Core CPUs / Multiprozessor-Maschinen müsste man zusätzlich den Zugriff auf den Speicherbus sperren
- ▶ Beobachtung: nun sind Teile von wait() und signal() die kritischen Regionen!
- ▶ Wir können **Sperrvariablen** für diese Regionen nutzen
 - ▶ z.B. mit TSL - „Test and Set Lock“ oder XCHG Swap
- ▶ D.h. wir führen pro Semaphor eine interne Sperrvariable s_lock ein und „übersetzen“:
 - ▶ **disable_interrupt => enter_region (s_lock)**
 - ▶ **enable_interrupt => leave_region (s_lock)**

Semaphore mit aktivem Warten /6

- ▶ wait() und signal() mit Sperrvariable `s_lock` und zugehörigen Methoden `enter_region()` / `leave_region()`:

wait (S):

repeat:

`enter_region (s_lock)`

 if `S > 0`:

`S--`

`leave_region (s_lock)`

 break

`leave_region (s_lock)`

return

signal (S):

`enter_region (s_lock)`

`S++`

`leave_region (s_lock)`

return

Semaphore mit und ohne Aktives Warten

- ▶ Sperren (Locks), die **aktives Warten** (repeat-Schleifen) benutzen, nennt man **Spinlocks** ([Link](#))
- ▶ Aktives Warten verschenkt Rechenzeit
- ▶ Was könnte man statt dessen machen?
- ▶ Bei **wait()**: sobald ein Prozess / Thread warten muss:
 - ▶ 1. Wir merken uns den Prozess in einer Liste zu S
 - ▶ 2. Wir lassen ihn schlafen => Zustand „**Waiting**“
- ▶ Bei **signal()**:
 - ▶ 1. Wir holen den nächsten Prozess aus der Liste zu S
 - ▶ 2. Wir versetzen den in den Zustand „**Ready**“ (erlauben Ausführung)

Semaphore ohne Aktives Warten - Implementierung

- ▶ Jeder Semaphor hat eine Datenstruktur (struct) mit
 - ▶ **count** (Integer) – der Wert der Semaphor-Variable
 - ▶ Eine Liste **list** mit Zeigern auf wartende Prozesse
- ▶ Es gibt zwei interne Operationen
 - ▶ **block**(): versetze den gerade ausführenden Prozess P (Aufrufer von block()) in den Zustand „waiting“
 - ▶ **wakeup**(P): versetze einen Prozess P (nächsten in der Liste **list**) in den Zustand „ready“ (d.h. P kann ausgeführt werden)

Semaphore ohne aktives Warten - in C

► Semaphor-Datenstruktur

```
typedef struct {  
    int count; struct process *list;  
} semaphore;
```

S->value kann jetzt negativ werden; Interpretation?

► Mögliche Implementierung (Pseudocode)?

```
wait (semaphore *S) {  
    S->count--;  
    if (S->count < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

```
signal (semaphore *S) {  
    S->count++;  
    if (S->count <= 0) {  
        get and remove process  
        P from S->list;  
        wakeup(P);  
    }  
}
```

Problem: Diese Version könnte wieder zu Race Conditions führen!

Effiziente Semaphore ohne Race Conditions

► Semaphor-Datenstruktur

Nur für Single-Core!

```
typedef struct {  
    int count; struct process *list;  
} semaphore;
```

```
wait (semaphore *S) {  
    disable_interrupts();  
    if (S->count > 0) {  
        S->count--;  
        enable_interrupts();  
        return;  
    }  
    add( this_process, S->list);  
    enable_interrupts();  
    block();  
}
```

```
signal (semaphore *S) {  
    disable_interrupts();  
    if (list is empty) {  
        S->count++;  
    } else {  
        P = RemoveFirst(S->list);  
        wakeup(P);  
    }  
    enable_interrupts();  
}
```

Synchronisation in der Praxis

Synchronisation in Posix Pthreads

- ▶ Posix hat **Sperren** oder **Mutexe** (**mutex locks**)
 - ▶ Datenstruktur vom Typ `pthread_mutex_t`
- ▶ Mutexe sind genau die Semaphore mit binären Werten (d.h. 0 = gesperrt / 1 = nicht gesperrt)

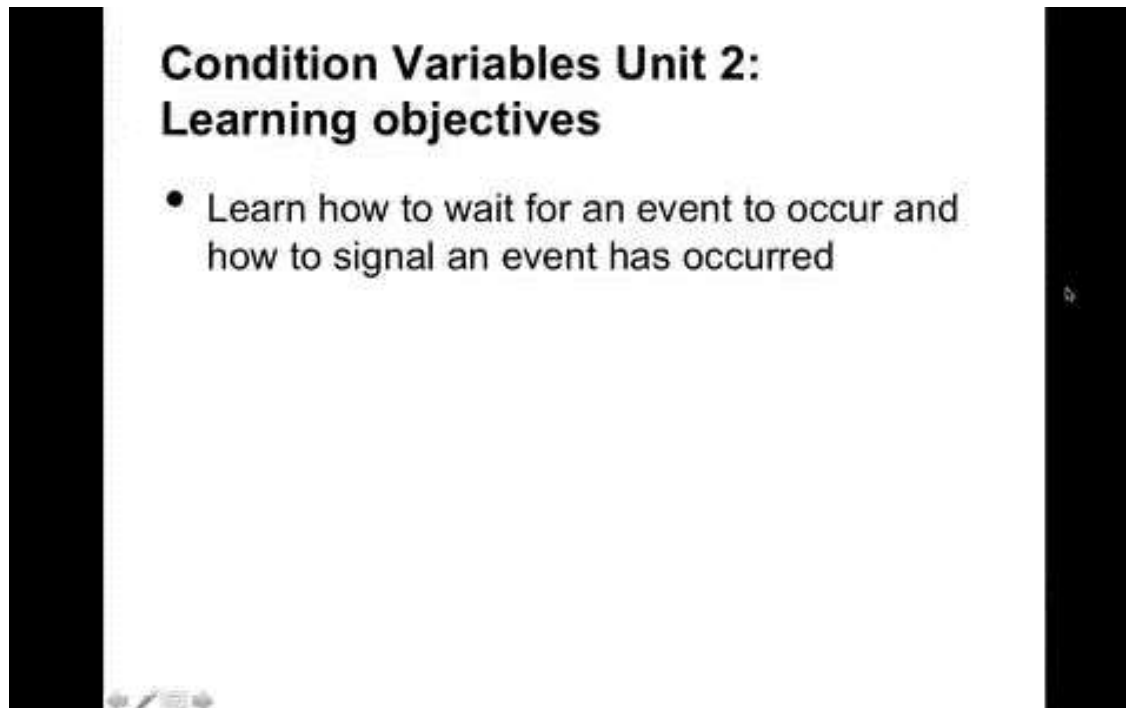
Aufruf (Pthread_mutex_*)	Beschreibung
<code>init (mutex,attr)</code>	Erzeuge ein Mutex
<code>destroy (mutex)</code>	Zerstöre ein Mutex
<code>lock (mutex)</code>	Erlange eine Sperre, oder blockiere
<code>trylock (mutex)</code>	Erlange eine Sperre, oder Fehler „busy“
<code>unlock (mutex)</code>	Gebe eine Sperre frei

Synchronisation in Posix Pthreads /2

- ▶ Die **Zustandsvariablen** (**conditions variables**) erlauben effizientes "Warten auf eine Bedingung"
 - ▶ Typ `pthread_cond_t`
- ▶ Zutreffen der Bedingung wird hier durch den Wert einer binären Variable dargestellt
 - ▶ Wenn sich der Wert der Variable ändert, werden wartende Threads automatisch aufgeweckt
- ▶ Alternative zu mehr komplizierten Verfahren:
 - ▶ Periodisch den Wert eines Mutex testen (z.B. mit `trylock()`), der eine Bedingung repräsentiert
 - ▶ Semaphore nutzen

Video zu condition variables

- ▶ Video von Mike Swift "Lecture 3, Unit 2: using condition variables,, [08a]
- ▶ Link: <http://goo.gl/stNNx5>
- ▶ Von 0:00 bis ca. 4:30 (min:sec)



Synchronisation in Posix Pthreads /3

- ▶ APIs der condition variables:
 - ▶ pthread_cond_init (condition, attr)
 - ▶ pthread_cond_destroy (condition)
 - ▶ pthread_cond_wait (condition, mut)
 - ▶ pthread_cond_signal (condition)
 - ▶ pthread_cond_broadcast (condition)
- ▶ **condition** ist die Variable, **mut** ein Mutex
- ▶ cond_signal (condition) setzt condition auf true
- ▶ cond_wait (condition, mut) gibt den Mutex **mut** frei, und legt den Aufrufer schlafen
 - ▶ **Wichtig:** zugleich „bindet“ man condition und **mut** :
Aufrufer wird automatisch aufgeweckt, wenn condition wieder wahr wird

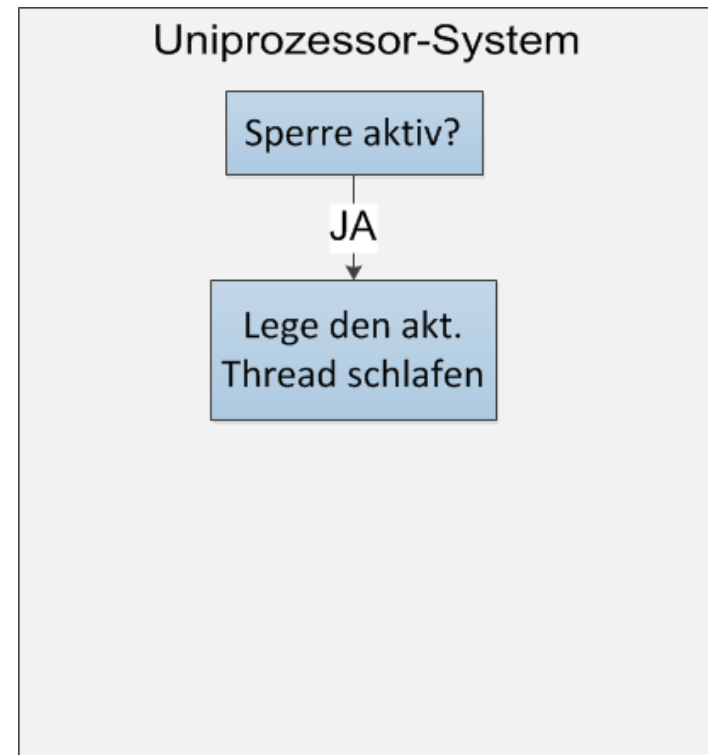
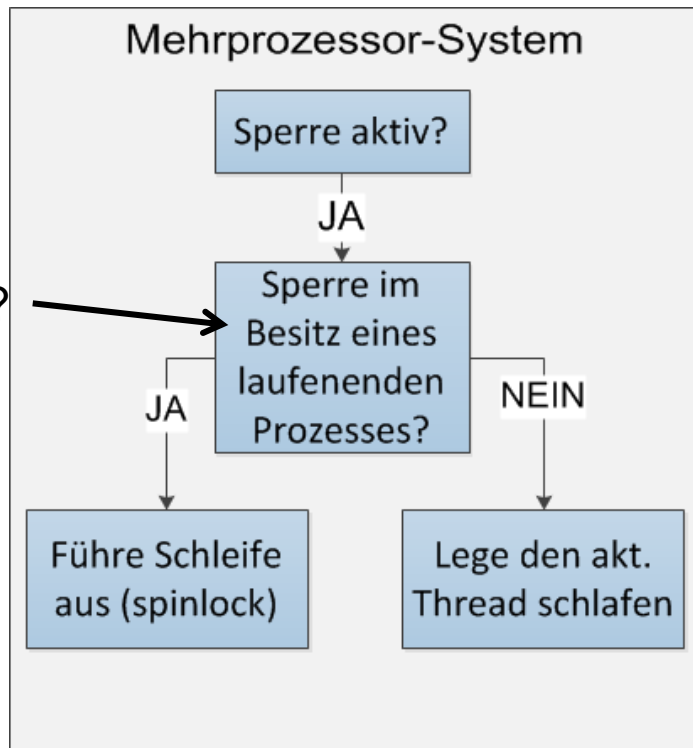
Verwendung von cond_wait (**condition**, **mut**)

1. Thread **A** blockiert Mutex **mut** und betritt seine kritische Region (mit mutex_lock (**mut**))
2. Wenn eine Bedingung nicht erfüllt ist, ruft **A** cond_wait (**condition**, **mut**) auf
 - ▶ Dabei wird Mutex **mut** automatisch freigegeben, und Thread **A** geht schlafen
3. Thread **B** erlangt Mutex, erfüllt irgendwann die Bedingung; dann weckt **A** via cond_signal (**condition**) auf
4. Thread **A** wacht automatisch auf, und Mutex **mut** wird automatisch blockiert
5. Thread **A** beendet die kritische Region, gibt **mut** frei

Synchronisation in Solaris

- ▶ Implementiert eine Vielzahl von Sperren, inklusive Unterstützung für Echtzeit-Threads
- ▶ Bei kurzen Codesegmenten
 - ▶ Benutzt aus Effizienzgründen **adaptive Mutexe**

Warum?



Zusammenfassung

- ▶ Wechselseitiger Ausschluss: Lösungen mit aktivem Warten (Fortsetzung)
- ▶ Semaphore: Grundlagen und Implementation
- ▶ Synchronisation in in der Praxis / POSIX: locks, condition variables
- ▶ Zusatzfolien:
 - ▶ Warten auf eine Bedingung
 - ▶ Monitore, Monitore in Java
- ▶ Quellen:
 - ▶ Synchronisation: Silberschatz et al., Kapitel 6, Tanenbaum et al., Kapitel 2
 - ▶ Speicher: Silberschatz et al., Kap. 8+9; Tanenbaum Kap. 3.2 + 3.3

Zusätzliche Folien

Warten auf eine Bedingung

Abfragen einer Bedingung

- ▶ Typische Situation: ein Thread wartet darauf, dass eine Bedingung zutrifft oder eine Zustandsänderung erfolgt ...
- ▶ ... Die nur von anderen Threads erzeugt werden kann
 - ▶ Firefox wartet auf weitere Daten von der Linux-Netzwerkschicht, um die Webseite anzuzeigen
 - ▶ Text Editor wartet auf den nächsten Tastendruck / Mausklick
 - ▶ "Memory cleaner" / garbage collector wartet, bis 90% des Speichers belegt ist
- ▶ Wie kann dieses "Warten" implementiert werden?
- ▶ Eine passable Lösung ist **Polling**: periodisches Abfragen einer Bedingung (z.B. des Variablenwertes) in einer Schleife, bis die Bedingung eintritt

Polling-Mechanismus

Thread 1

code...

while (Bedingung X nicht
erfüllt):

 sleep k milliseconds;

code...

Thread 2

code ...

<Manipuliert Bedingung X>

code ...



- ▶ Probleme beim Polling?
- ▶ Rechenzeit wird sinnlos vergeudet
- ▶ **Trade-off** zwischen CPU-Verschwendung und Reaktionsgeschwindigkeit (Verzögerung der Verarbeitung bis zu k Milisekunden)

Effizientes Warten

Thread 1

code ...

while (Bedingung X ist
nicht erfüllt):

`wait_until` (X wurde
manipuliert);

code ...



blockierender Aufruf

Thread 2

code ...

<Manipuliert Bedingung X>

<Dann: **Versetzt Thread 1 in
Zustand "ready"**>

code ...

- ▶ Statt periodisch nachzufragen, legt man Thread 1 "schlafen" (in den Prozess-Zustand **waiting**)
- ▶ Thread 1 wird (von anderen Threads) geweckt, wenn die Bedingung sich verändert hat

Verständigung der Threads

- ▶ Wir brauchen also ein Werkzeug, um ...
- ▶ ... einen Thread schlafen zu legen, d.h. in Zustand "waiting" zu versetzen und
- ▶ .. diesen von einem anderen aufwecken zu können
- ▶ Schon bekannt?
- ▶ Semaphore können das leisten:
- ▶ Initialisiere Semaphor: $S := 0$
- ▶ `wait(S)` – Aufrufer T geht in Zustand "waiting"
- ▶ Ein anderer Thread kann mit `signal(S)` den Thread T wieder aufwecken

Beispiel: Producer - Consumer

- ▶ Producer und Consumer warten auf eine Bedingung
- ▶ Consumer: "Puffer nicht leer" => entnehme Zeichen
- ▶ Producer: "Puffer nicht voll" => speichere Zeichen
- ▶ Wie können wir das mit Semaphoren umsetzen?

freier Pufferplätze

belegter Pufferplätze

- ▶ $\text{emptyCount} := N$, $\text{fullCount} := 0$, $\text{useQueue} := 1$

Produce (item):
wait (emptyCount)
wait (useQueue)
putItemIntoQueue (item)
signal (fullCount)
signal (useQueue)

item = **Consume**():
wait (fullCount)
wait (useQueue)
item := **getItemFromQueue**()
signal (emptyCount)
signal (useQueue)

Monitore



Monitore

- ▶ Semaphore sind universell, aber fehlerhaft
- ▶ Man hat deshalb eine höherstufige Basisoperation eingeführt: **Monitore**
 - ▶ Brinch Hansen (1973) und Hoare (1974)
- ▶ Ein Monitor ist wie eine Klasse (OOP), und enthält
 1. Von Prozessen gemeinsam genutzten Daten
 2. Ihre Zugriffsprozeduren (oder Methoden)
- ▶ Idee:
 - ▶ Alle kritischen Regionen (zu denselben gemeinsamen Daten) werden zu Prozeduren in einem Monitor
 - ▶ Monitor garantiert: Nur ein einziger Thread auf einmal kann innerhalb einer dieser Prozeduren aktiv sein

Monitore - Funktionsweise

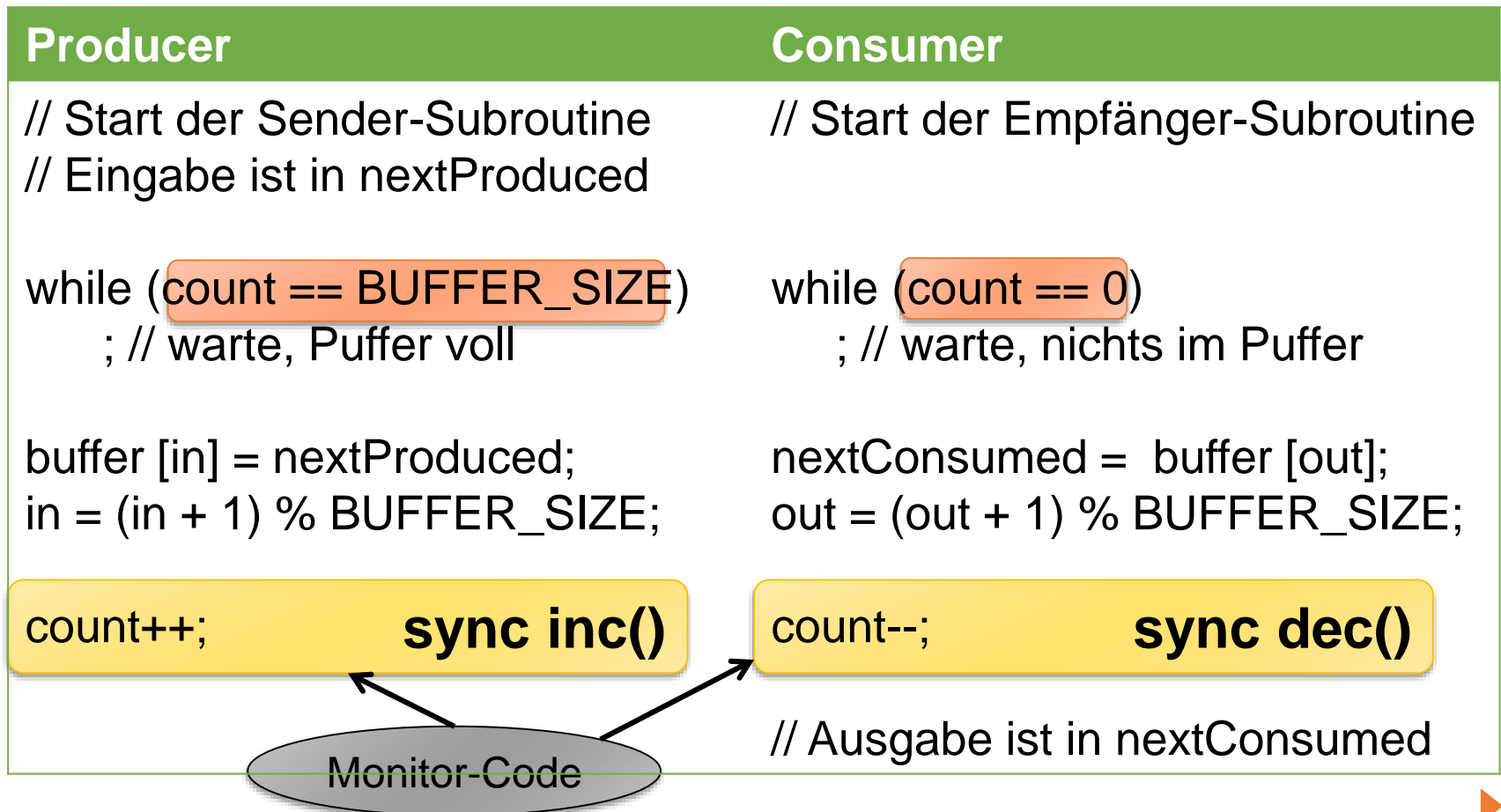
- ▶ Nur ein Thread auf einmal darf den Code in speziell gekennzeichneten Prozeduren (die mit **sync**) ausführen
 - ▶ Falls Prozess 1 Code in A ausführt, und Prozess 2 auch A aufruft, wird P2 blockiert
- ▶ Vorteile?
- ▶ Der Programmierer muss sich nicht mehr um den „Semaphor-Business“ kümmern
- ▶ Weniger Fehler

```
monitor name {  
    // Gemeinsame Daten  
    ...  
    sync procedure A (...) {  
        ...  
    }  
    sync procedure B (...) {  
        ...  
    }  
    procedure noSync (...) {  
        ...  
    }  
}
```

Daneben kann es ggf. auch normale Prozeduren (ohne kritische Abschnitte) geben

Producer – Consumer mit Monitoren

- ▶ Producer schreibt in einen Puffer, Consumer liest heraus
- ▶ Lösung mit einem Monitor?

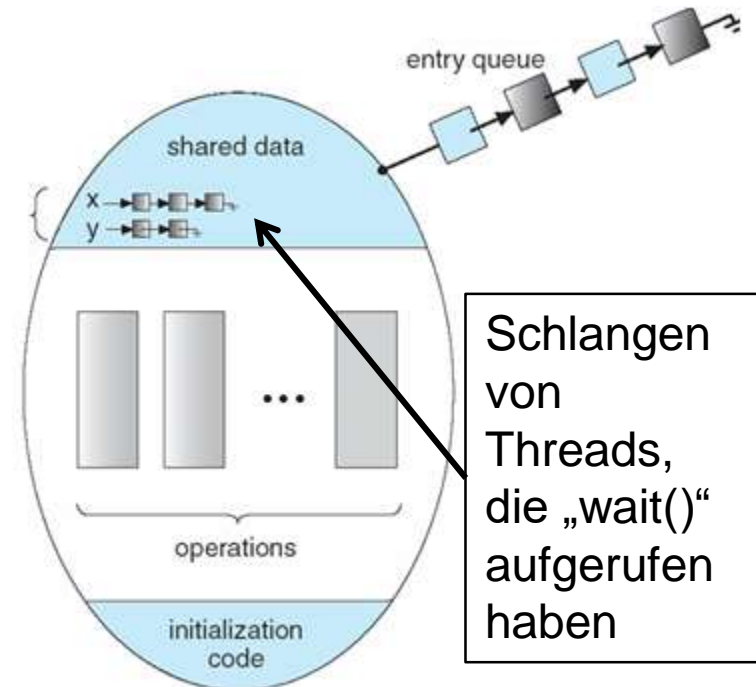


Ist die gezeigte Umsetzung effizient?

- ▶ Die while-Schleifen verschwenden Rechenzeit
- ▶ Für das Testen „count == BUFFER_SIZE“ und „count == 0“ muss der Monitor betreten und wieder verlassen werden – ggf. ist das auch ineffizient
- ▶ Wie müsste man hier Monitor-Konzept erweitern?
- ▶ Wir hätten gerne die Möglichkeit, ...
 - ▶ Einen Thread schlafen zu legen, solange eine bestimmte Bedingung nicht erfüllt ist,
 - ▶ ... und diesen automatisch aufzuwecken, sobald eine Bedingung zutrifft
- ▶ Bedingungen z.B.
 - ▶ „count == BUFFER_SIZE“
 - ▶ „count == 0“

Zustandsvariablen (**Conditions**)

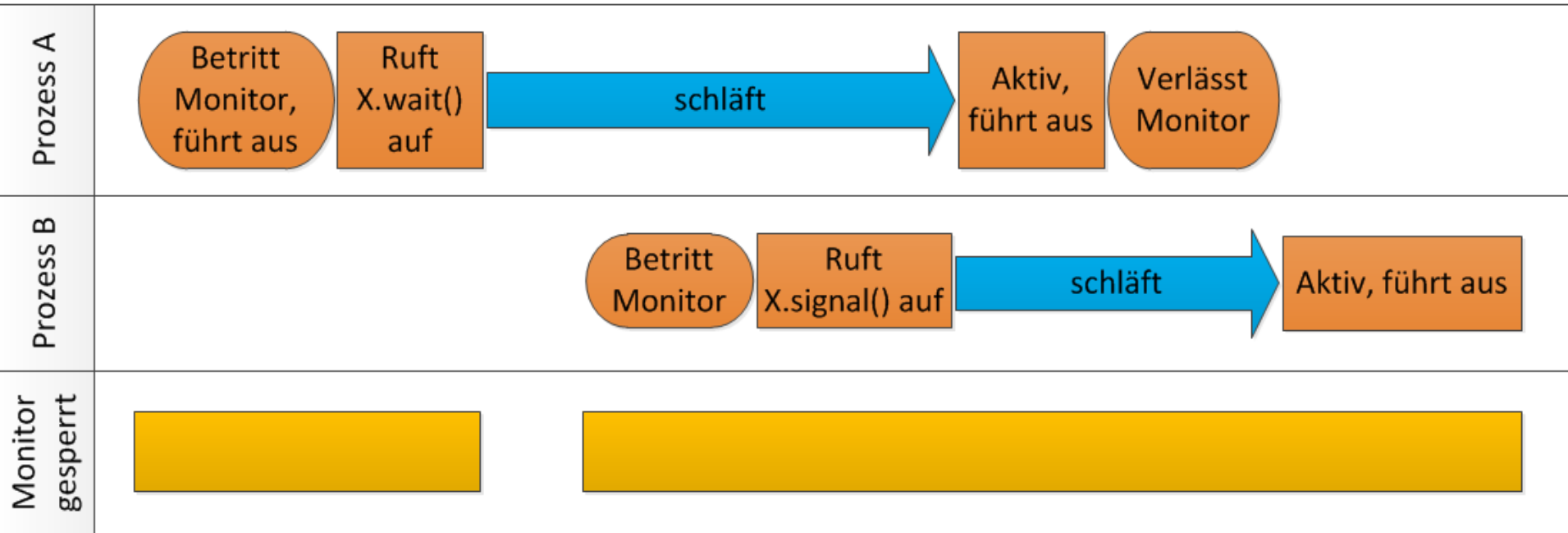
- ▶ Eine **Zustandsvariable oder Bedingungsvariable X** (**condition variable**) verhält sich ähnlich einem Semaphor
- ▶ **X.wait()**: blockiert den ausführenden Thread / Prozess; ein anderer Prozess kann den Monitor betreten
- ▶ **X.signal()**: aktiver Thread A weckt einen schlafenden Thread B wieder auf
 - ▶ Aufruf wird ignoriert, wenn niemand schläft (anders als bei Semaphoren!)
- ▶ Aber A und B können nicht zugleich ausführen – welche Möglichkeiten gibt es nun?



Monitore - Varianten

- ▶ ... **X.signal** (): aktiver Thread A weckt einen schlafenden Thread B wieder auf
- ▶ Aber A und B können nicht zugleich ausführen – welche Möglichkeiten gibt es nun?
- ▶ **signal and wait**: A schläft ein, bis B fertig ist
 - ▶ Oder B durch eine andere Zustandsvariable schlafengelegt wurde
- ▶ **signal and continue**: B wartet, bis A den Monitor verlassen hat
 - ▶ Oder A durch eine andere Zustandsvariable schlafengelegt wurde

Beispiel Zustandsvariable



Ist das „**signal and wait**“ oder „**signal and continue**“?

Monitore in Java

- ▶ In Java ist jede Klasse schon ein Monitor
- ▶ Das Schlüsselwort **synchronized** vor einer Methode f bewirkt, dass f nur von max. einem Thread auf einmal betreten werden kann
- ▶

```
class Anything {  
    ▶ private ... sharedData;  
  
    ▶ synchronized public void f (...)  
      { ... }  
  
    ▶ synchronized public void g (...)  
      { ... }  
  
    ▶ }
```

Bedingungssynchronisation in Java


- ▶ Java hat keine „reinen“ Zustandsvariablen
- ▶ Bis Java 5 hat man den **wait()/notify()**-Mechanismus benutzt (ab Java 5 gibt es bessere Abstraktionen)
 - ▶ **obj.wait()**: legt den Thread schlafen, der **wait()** an **obj** aufgerufen hat (in eine Schlange zu **obj**)
 - ▶ **obj.notify()** (bzw. **obj.notifyAll()**): weckt irgendeinen (bzw. alle) Threads, die in der Schlange zu **obj** schlafen
- ▶ Beispiel: ein Pool an DB-Verbindungen – wie?
 1. Ein Thread holt sich eine Verbindung aus dem Pool, falls eine frei ist; sonst ruft **pool.wait()** auf
 2. Wenn er fertig ist, gibt er die Verbindung an das Pool zurück, und ruft **pool.notify()** auf

Monitore - Umsetzung von Mutual Exclusion

- ▶ Annahme: „signal and wait“-Verhalten
 - ▶ d.h. „signal“-Aufrufer wartet nach „signal()“
- ▶ Semaphore 1: **mutex** - initialisiert zu 1
 - ▶ Jeder Prozess / Thread führt vor dem Betreten des Monitors WAIT (**mutex**), und SIGNAL (**mutex**) danach
- ▶ Semaphore 2: **next** - initialisiert zu 0
 - ▶ Anzahl der „signal“-Aufrufer, die z.Z. schlafen („signal and wait“)
- ▶ Variable **next_count**: Hilfsvariable zu **next**
- ▶ Eine Monitor-Prozedur F wird zu:

*WAIT, SIGNAL:
Funktionen des
Semaphors!*

D.h. falls es irgendwelche wartenden „signal“-Aufrufer gibt, rufe signal (next) auf, sonst „normales“ signal (mutex)

```
WAIT (mutex);   
Code von F;  
if (next_count > 0)  
    SIGNAL (next);  
else  
    SIGNAL (mutex);
```

Monitore - Umsetzung von Zustandsvariablen

- ▶ Für jede Zustandsvariable x brauchen wir
 - ▶ Semaphore x_sem , anfangs 0
 - ▶ Integervariable x_count , anfangs auch 0
 - ▶ In etwa: Anzahl $x.wait()$'s minus Anzahl $x.signal()$'s
- ▶ **$x.wait()$** ist dann:
- $x.signal()$** ist:

```
x_count++;  
if (next_count > 0)  
    SIGNAL (next);  
else  
    SIGNAL (mutex);  
WAIT (x_sem);  
x_count--;
```

Prozess / Thread A:
„wait“-Aufrufer

```
if (x_count > 0) {  
    next_count++;  
    SIGNAL (x_sem);  
    WAIT (next);  
    next_count--;  
}
```

Prozess / Thread B:
„signal“-Aufrufer

wecke „signal“-
Aufrufer auf

oder gebe den
den Monitor frei

warte via
Semaphore von x
wieder lebendig!
(„wait“-Aufrufer)

ggf. wecke einen
„wait“-Aufrufer
lege den „signal“-
Aufrufer schlafen
wieder lebendig!
(„signal“-Aufrufer)

Zusätzliche Folien: Synchronisation in der Praxis

Synchronisation in Solaris

- ▶ Bei längeren Codesegmenten
 - ▶ Benutzt Zustandsvariablen und sog. Lese-Schreib-Sperren (**readers-writer locks**)
- ▶ **Readers-Writers Problem**
 - ▶ Mehrere Threads dürfen eine Datenstruktur lesen, aber nur ein Thread darf schreiben
 - ▶ Wird durch **readers-writer lock** (genannt auch **multi-reader lock**) gelöst
 - ▶ Konstruiert durch Mutexe + Zustandsvariablen oder durch Semaphore

Synchronisation in Windows XP

- ▶ Uniprozessor-Systeme
 - ▶ Im Kern werden **Interrupt-Masken** (d.h. Ausschalten der Interrupts) benutzt
- ▶ Mehrprozessor-Systeme nutzen Spinlocks
 - ▶ Nur für kurze Codeabschnitte
 - ▶ Ein Thread, der die Sperre besitzt (d.h. andere warten lässt), wird nie unterbrochen (engl. to be preempted)
- ▶ Es gibt auch universelle **dispatcher objects**
 - ▶ Helfen, die Abarbeitung auf „später“ zu verschieben
 - ▶ Diese können als Mutexe, Semaphore, Timer arbeiten
 - ▶ Sie können auch sog. **Ereignisse** (events) liefern, die den Zustandsvariablen ähnlich sind

Zustand „Suspended“

- ▶ **suspended** = suspendiert, temporär ausgesetzt
 - ▶ Generalisierung: Ein Prozess, der nicht sofort ausgeführt werden kann, unabhängig, ob dieser auf ein Ereignis wartet oder nicht
- ▶ Andere Gründe als Auslagerung auf die Festplatte?
 - ▶ Ausgesetzt wegen Probleme (z.B. zu speicherintensiv)
 - ▶ Benutzer möchte ihn debuggen / hat „Ctrl-Z“ gedrückt
 - ▶ Es ist ein Hintergrund- bzw. / behelfsmäßiger Prozess, der selten benutzt wird
 - ▶ Wird periodisch, aber selten ausgeführt
 - ▶ Elternprozess möchte ihn modifizieren oder zwischen mehreren Kindprozessen die Arbeit koordinieren