7. Synchronisierung

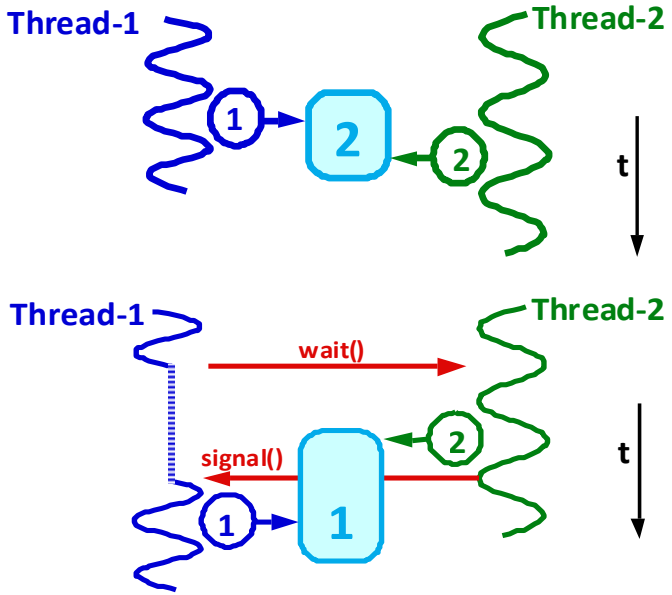
Michael Schöttner
Betriebssysteme und Systemprogrammierung

7.1 Vorschau

- Motivation
- Kritischer Abschnitt
- Wechselseitiger Ausschluss
- Pthread Mutex und Bedingungsvariablen
- Semaphore
- Beispiele für Synchronisierungsprobleme
- Verklemmungen und deren Behandlung
- Sperrfreie Synchronisierung

7.2 Ausgangssituation

- Mehrprogrammbetrieb:
  - Programme greifen auf die gemeinsame Hardware & Software zu
  - viele Programme konkurrieren "gleichzeitig" um Ressourcen, z.B. Hauptspeicher, CPU(s Disk, Devices, Interrupts ...
- Wettlaufsituation (race condition)
  - wenn nebenläufige Programme (Threads) au gemeinsame Variablen schreiben, so ist das Ergebnis nicht deterministisch.
- Synchronisierung der nebenläufigen Ausführung lässt das Resultat deterministisch werden.



7.3 Kritischer Abschnitt

- = engl. critical region: Programmabschnitte, die auf gemeinsame Variablen zugreifen und deshalb einer Synchronisierung bedürfen.
  - Wechselseitiger Ausschluss gewünscht → max. 1 Thread im kritischen Abschnitt
  - Keine Annahmen bezüglich CPU-Geschwindigkeit, #Cores, ...
  - Fairness: Wartezeit für Eintritt in kritischen Abschnitt muss begrenzt sein.
  - Keine Verklemmungen → Fortschritt garantiert

- Unterschiedliche Programmabschnitte können dieselben Variablen nutzen → Nicht Programmabschnitt, sondern Variablen werden geschützt.

Fehlerhafte Lösung für einen kritischen Abschnitt

- Wird während Prüfen des Flags umgeschaltet, so können beide Threads den kritischen Abschnitt betreten.
- Problem. Testen und Setzen des Flags geschieht nicht atomar.

```
int busyFlag = 0;
```

Thread 1

```
...
while (busyFlag);


busyFlag = 1;

/* Anweisungen */

busyFlag = 0;
...
```

Umschalten

Thread 2

```
...

while (busyFlag);
busyFlag = 1;

/* Anweisungen */

busyFlag = 0;
...
```

Synchronisierung

- Beispiel Ein Thread inkrementiert jeweils zwei globale Variablen
  - Einmal mit Synchronisierung und einmal ohne.
- Erzeugt der Haupt-Thread mehrere Threads die **`my_thread`** nebenläufig ausführen, so können wir Lost-Update-Probleme sehen

statischer Mutex

```
#define COUNT  100000

long sync=0;
long asyn=0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *my_thread(void *param) {
  long zwisch;
  int count = COUNT;

   while ( count-- > 0 ) {
      zwisch=asyn+1; asyn=zwisch;

      pthread_mutex_lock( &mutex );
      zwisch = sync+1;
      sync   = zwisch;
      pthread_mutex_unlock( &mutex );
   }

   printf("asyn: %ld\n", asyn);
   printf("sync: %ld\n", sync);
   return NULL;
}
```

sync.c

Lost-update Problem

- Wenn eine Inkrementierung unterbrochen wird und der andere Thread ebenfalls inkrementiert, kann eine Inkrementierung verloren gehen
  → „i++;“ erfolgt evt. nicht atomar
- Beispiel-Ausgabe des Programms (siehe vorherige Seite):

```
asyn: 18213815
sync: 18761335
asyn: 19452479
sync: 20000000
```

- Ausnahmsweise und je nach Last und „Laune“ des Schedulers im Betriebssystem läuft das Programm auch ohne Verlust:

```
asyn: 19843320
sync: 19843320
asyn: 20000000
sync: 20000000
```

7.4 Wechselseitiger Ausschluss

- = engl. mutual exclusion; löst das Problem des kritischen Abschnitts
- Basis hierfür ist eine atomare Test-&-Set-Instruktion:
  - Prinzip: Test = return Speicherwort; Set = setze Speicherwort auf true
  - Bieten alle Prozessoren für Desktop und Server-Betriebssysteme
    - Test&Set verhindert Interrupts am eigenen Core,
      sowie potentielle Zugriffe durch andere Cores oder Busmaster-Geräte
- Abstrakte Implementierung einer Sperre mithilfe von Test&Set

```
int  lock = 0;       /* 0=not locked, 1=locked */

void acquire () {
    while  Test_And_Set (lock_var ) ;  /*  busy polling */
}

void release () {
      lock = 0;
}
```

Mutex in Pthread

- Im Beispiel zum Lost-Update wurde der Mutex statisch angelegt. Dies ist einfacher, sofern die Default-Einstellungen verwendet werden soll.
- Man kann diesen auch dynamisch Allozieren und Freigeben:

```
int  pthread_mutex_init (
   pthread_mutex_t  *mutex,
   const  pthread_mutex_attr_t *mutexattr );

int pthread_mutex_destroy ( pthread_mutex_t *mutex );
```

- Weitere Funktionen:

```
int pthread_mutex_lock    ( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock ( pthread_mutex_t *mutex );
```

Mutex in Pthread, Attribute

- Mit dem Attribut kann festgelegt werden, ob ein Thread eine Sperre wiederholt erhält, wenn er sie bereits hat. Bei einer Rekursion ist dies gewünscht.
- Entscheidend ist **kind**

Non-Portable Extension

  - `PTHREAD_MUTEX_FAST_NP` (Default): blockiert ggf.
  - `PTHREAD_MUTEX_RECURSIVE_NP`: erlaubt rekursives locking
    - Intern wird ein Zähler verwendet → Es müssen genauso viele Freigaben erfolgen
  - `PTHREAD_MUTEX_ERRORCHECK_NP`: blockiert nicht, aber ggf. Fehler `EDEADLK`

Deadlock

```
int pthread_mutexattr_init    ( pthread_mutexattr_t *attr );

int pthread_mutexattr_destroy ( pthread_mutexattr_t *attr );

int pthread_mutexattr_settype ( pthread_mutexattr_t *attr, int kind );

int pthread_mutexattr_gettype ( const  pthread_mutexattr_t  *attr, int *kind );
```

Bedingungsvariablen in Pthread

- Warten auf eine Bedingung in einem kritischen Abschnitt
- Eine Bedingungsvariable (engl. condition variable) ist mit einem Mutex verknüpft
  - Dieser wird beim Warten auf die Bedingung freigegeben
- Bedingungsvariablen können statisch angelegt werden, sofern die Default-Einstellungen ausreichend sind (wie auch ein Mutex).

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init    ( pthread_cond_t *cond,
                           pthread_condattr_t *cond_attr );
int pthread_cond_destroy ( pthread_cond_t *cond );
```

NULL = default values

Bedingungsvariablen in Pthread (3)

- Warten auf das Bedingungs-Signal:

```
int pthread_cond_wait      ( pthread_cond_t *cond,
                             pthread_mutex_t *mutex );

int pthread_cond_timedwait ( pthread_cond_t  *cond,
                             pthread_mutex_t *mutex,
                             const struct timespec *abstime);
/* struct timespec -> seconds + nanoseconds */
```

- Signalisieren der Bedingung

```
/* wakeup thread with highest priorty */
int pthread_cond_signal    ( pthread_cond_t *cond );

/* wakeup all waiting threads */
int pthread_cond_broadcast ( pthread_cond_t *cond );
```

Beispiel: Bedingungsvariablen

- Siehe Quelltext: **`cond.c`**

- Ablauf:
  - Haupt-Thread erzeugt drei Threads und wartet kurz
  - Die anderen Threads warten jeweils auf die gleiche Condition-Variable
  - Haupt-Thread signalisiert die Condition-Variable
  - Ein Thread wird dadurch deblockiert. Dieser signalisiert wieder die Condition-Variable usw.

Beispiel: Bedingungsvariablen (2)

- Auszug: `cond.c`

```
void *threads (void *arg) {
   pthread_mutex_lock( &mutex );
   pthread_cond_wait( &cond, &mutex );

   /* do something */

   pthread_cond_signal( &cond );
   pthread_mutex_unlock( &mutex );
   return NULL;
}
```

- `pthread_cond_wait`:
  - Blockiert und gibt die Sperre frei.
  - Wenn das Signal eintritt, wird versucht die Sperre neu zu erwerben
  - Daher muss nach `pthread_cond_signal` ein `pthread_mutex_unlock` folgen

Bemerkungen zu Bedingungsvariablen

- Durch `pthread_cond_wait` wird der Aufrufer blockiert und die Sperre freigegeben
- Das Signal wird von einem anderen Thread `pthread_cond_signal` ausgelöst, der meist ebenfalls im kritischen Abschnitt ist.
  - Hierdurch wird ein wartender Thread aufgeweckt und dieser versucht implizit die Sperre wieder zu bekommen
  - `pthread_cond_signal` gibt die Sperre nicht automatisch frei
  - Deswegen ist wichtig, dass der signalisierende Thread die Sperre nach dem `pthread_cond_signal` wieder freigibt.
- Hierdurch ist immer nur ein Thread im kritischen Abschnitt

Abstraktes Beispiel zu Condition-Variablen

- Eine Warteschlange mit beschränkter Größe, die von vielen Threads zugegriffen wird
  - `put`: Element einfügen (blockierende Funktion)
  - `get`: Element auslesen (blockierende Funktion)
  - Diese Funktionen müssen mithilfe von einem Mutex zwischen Threads synchronisiert werden
  - Wenn ein Thread `get` aufruft und die Warteschlange leer ist, so muss er warten bis ein anderer Thread `put` aufruft.
    - Wenn er nun einfach blockiert und die Sperre nicht freigibt, so könnte kein anderer Thread `put` aufrufen und er würde verhungern
    - Mit Condition-Variablen kann er auf neue Daten warten und den Mutex implizit freigeben
  - Gleiches gilt für einen Thread der `put` aufruft. Falls die Warteschlange voll ist, so muss er warten bis ein anderer Thread `get` aufruft ...

Spurious wakeups

- Ein mit **`pthread_cond_wait`** wartender Thread kann eventuell aufgeweckt werden, ohne das eine Signalisierung stattgefunden hat.
  - Dies nennt sich „spurious wakeups“ (siehe auch man pages)
  - Die genauen Gründe liegen in der Implementierung des jeweiligen Betriebssystems, sind aber nicht ohne weiteres nachvollziehbar.
- Wichtig ist, dass eine Bedingung auch nach dem Deblockieren von **`pthread_cond_wait`** nochmals geprüft wird.
- Lösung: setze **`pthread_cond_wait`** in eine Schleife →

```
int my_condidtion = 0;

void *threads (void *arg) {
   pthread_mutex_lock( &mutex );

   while (!my_condition)
      pthread_cond_wait( &cond, &mutex );
```

Hörsaal-Aufgabe

- Schreiben Sie ein Programm welches drei zusätzliche Threads startet.
- Jeder zusätzliche Thread soll endlos jeweils eine Ziffer ausgeben (1, 2, od. 3)
- Synchronisieren Sie die Threads, sodass die Ziffern immer abwechselnd ausgegeben werden. Bei einer korrekten Synchronisierung darf kein Zeichen mehrfach in Folge ausgegeben werden.

7.5 Semaphore

- Semaphor: (Wortbedeutung aus dem Griechischen)
  - optischer Telegraph; Mast mit Armen, durch deren Verstellung Zeichen zur Nachrichtenübermittlung weitergeleitet werden; schon im Altertum bekannt
  - Im Eisenbahnwesen auch Bezeichnung für ein Hauptsignal; in der Schifffahrt
  - Zur Anzeige von Windrichtung und Windstärke von der Küste aus.

- Hier besondere Variablen zur Synchronisierung zwischen Threads
  - Im Gegensatz zu Test&Set hier **kein** Busy-Polling

7.5 Semaphore

- Variable zur Synchronisierung mit folgenden **atomaren** Operationen:
  - vorgeschlagen durch E. Djikstra, 1968
  - binäre Semaphore mit Werten 0 oder 1
  - zählende Semaphore mit Werten 0 .. n
  - Initialisieren: InitSem(semVar)
  - "Passieren"?: P(semVar)
  - "Vreigeben": V(semVar)

```
if (semVar > 0)
     semVar = semVar - 1
else {
     warten auf V( semVar )
}
```

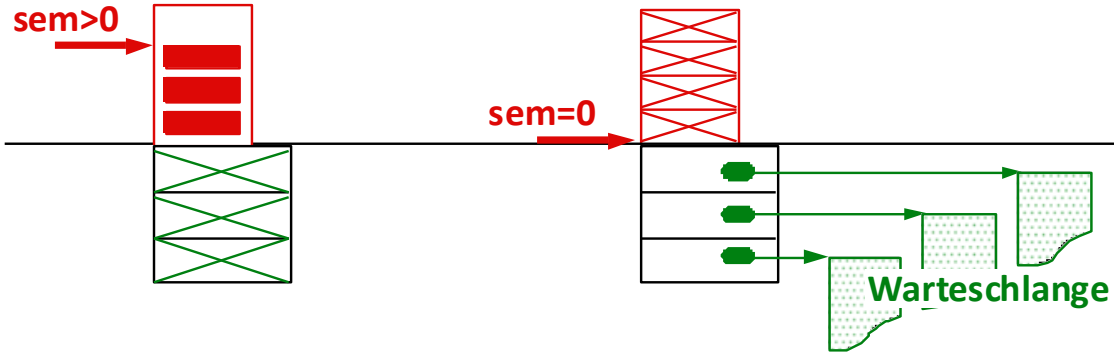
```
if ( Prozess wartet auf semVar )
     anstossen( Prozess )
else
     semVar = semVar + 1;
```

- Originalsprache Holländisch ...

7.5 Semaphore

- Funktionen P&V werden intern mithilfe der Test & Set-Instruktionen realisiert
  - Entweder direkt durch den Compiler oder das Betriebssystem

- Semaphore haben intern eine Queue zur Verwaltung wartender Threads.
  - Einträge verweisen auf Thread-Control-Block (TCB)
  - alle wartenden Threads sind blockiert → kein Busy-Waiting



Ende Teil 1

7.5.1 Leser/Schreiber mit Semaphore

- N Leser oder 1 Schreiber parallel erlaubt.

```
int             readcount=0;        /* Anzahl aktiver Leser */
semaphore       mutex=1, wrt=1;

Leser:
P(mutex); readcount++;              /* mit Leser sync. */
if (readcount==1) P(wrt);           /* kein Leser aktiv? Schreiber blockieren */
V(mutex);

reading();

P(mutex); readcount --;             /* mit Leser sync. */
if (readcount==0) V(wrt);           /* kein Leser mehr aktiv? Schreiber zulassen */
V(mutex);

Schreiber:
P(wrt);                             /* jeweils nur 1 Schreiber */
writing();
V(wrt);
```

7.5.2 Erzeuger/Verbraucher-Problem

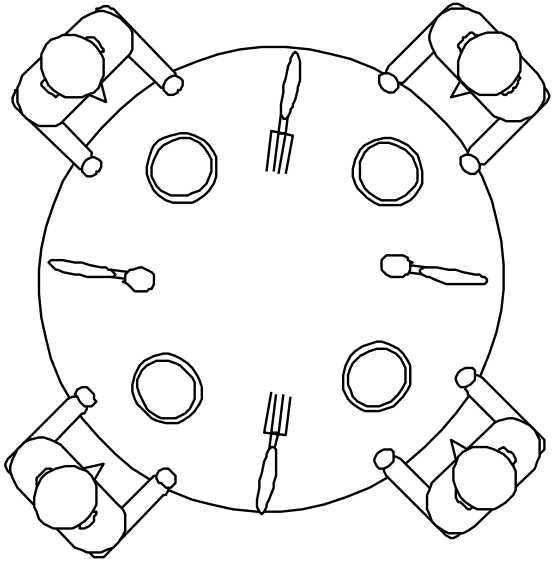
- N Erzeuger, N Verbraucher

```
int       buffer[N];     /* Puffer */
int       in=0, out=0;   /* Pufferzeiger */
semaphore used = 0, free = N,
          e_mutex = 1, v_mutex = 1;

void Erzeuger() {
  int item_w;

  produce(&item_w);
  P(free);       /* noch Platz im Puffer? */

  P(e_mutex);    /* Start krit. Abschnitt */
  buffer[in] = item_w;
  in = (in+1) % N;
  V(e_mutex);    /* Ende krit. Abschnitt  */

  V(used);       /* neue Daten anzeigen   */
}
```

```
void Verbraucher(){
  int item_r;

  P(used);       /* Daten im Puffer?      */

  P(v_mutex);    /* Start krit. Abschnitt */
  item_r = buffer[out];
  out = (out+1) % N;
  V(v_mutex);    /* Ende krit. Abschnitt  */

  V(free);       /* ein Eintrag wird frei */
  consume(item_r);
}
```

7.5.3 Speisende Philosophen

- N Philosophen sitzen um einen Tisch.
- Sie denken u. wollen Spaghetti essen.
- Zum Essen sind je eine Gabel und ein
- Löffel notwendig (oder 2 Gabeln).



7.5.3 Speisende Philosophen

- Lösungsansatz 1:
  - Funktioniert
  - Bietet wenig Parallelität, da immer nur 1 Philosoph essen kann

```
Philosoph i:

think();                    /* unbestimmte aber endliche Zeit   */

P(mutex);
     take_fork(i);          /* linkes Besteck nehmen            */
     take_fork((i+1)%N);    /* rechtes Besteck nehmen           */

     eat();                 /* unbestimmte aber endliche Zeit   */

     put_fork(i);           /* linkes Besteck ablegen           */
     put_fork((i+1)%N);     /* rechtes Besteck ablegen          */
V(mutex);
```

7.5.3 Speisende Philosophen

- Lösungsansatz 2:
  - Idee: drei Zustände THINK, HUNGRY, EAT pro Philosoph in Array `state[N]`
  - Essen ist nur möglich, wenn gerade kein Nachbar isst.
  - Array von Semaphore `S[N]` zum blockieren hungriger Philosophen, falls Gabeln von einem oder beiden Nachbarn in Gebrauch.
    - Alle Semaphore mit 0 initialisiert.
  - Makros `LEFT=(i-1) mod N;  RIGHT=(i+1) mod N;`

Lösungsansatz 2

```
Philosoph i:

think(); take_forks(i); eat(); put_forks(i); /* Gabeln nehmen, essen, Gabeln freigeben */

take_forks(i) {
   P(mutex);                          /* testen atomar durchführen */
   state[i] = HUNGRY;  test(i);       /* Versuch beide Gabeln zu nehmen */
   V(mutex);
   P(S[i]);                           /* blockiert, falls Gabeln nicht frei waren */
}

put_forks(i) {
   P(mutex);
   state[i] = THINK; test(LEFT); test(RIGHT); /* will linker oder rechter Nachbar? */
   V(mutex);
}

test(i) {                             /* beide Gabeln frei? */
   if (state[i]==HUNGRY && state[LEFT]!=EAT  && state[RIGHT]!=EAT) {
      state[i]= EAT;  V(S[i]);        /* Gabeln sind beide frei, Philosophen i kann essen */
   }
}
```

7.5.4 Semaphore in UNIX

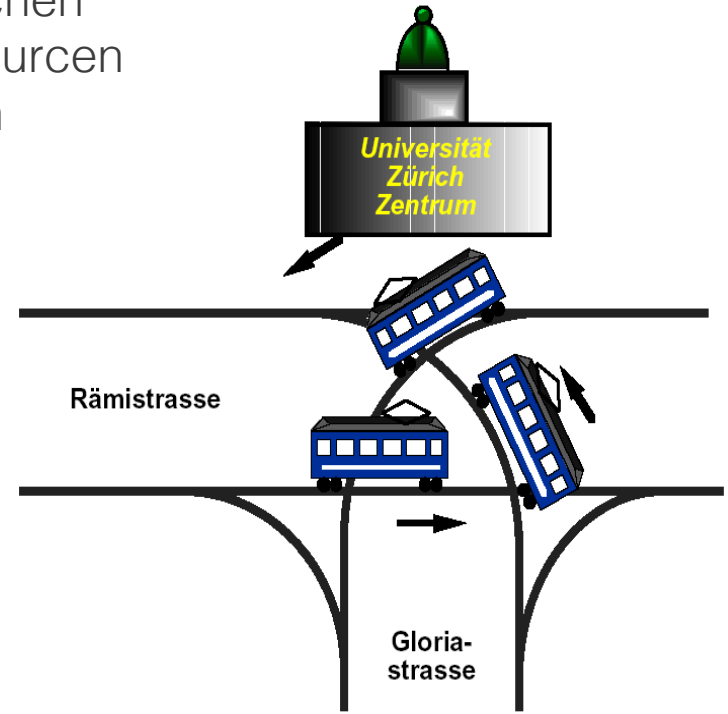
- Funktionen in UNIX

```
#include <semaphore.h>

int sem_init ( sem_t *sem, int pshared, unsigned int value );
int sem_wait ( sem_t * sem );
int sem_post ( sem_t * sem );
int sem_destroy ( sem_t * sem );
```

  - `pshared`
    - 0: Nutzung zwischen Threads eines Prozesses
    - !=0: Nutzung zwischen Prozessen. In diesem Fall muss Semaphor-Variable aber in einem shared-memory Bereich liegen (siehe später)

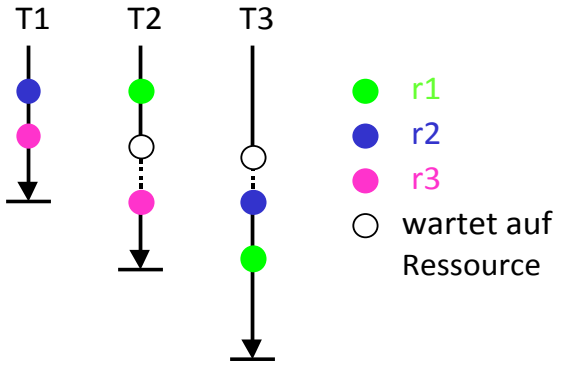
7.6 Verklemmungen

- Zwei oder mehrere Threads machen keinen Fortschritt, weil sie Ressourcen besitzen, die von einem anderen Thread benötigt würden.
- Beispiel: Verklemmung bei der Trambahn in Zürich →

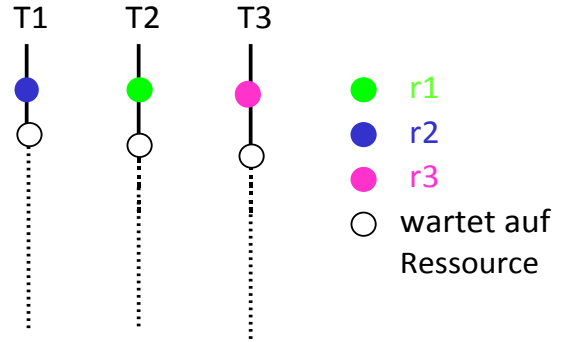


Beispiel: System mit drei Ressourcen

- Drei Threads verlangen erst eine Ressource und später noch eine zweite.
- Günstiger Verlauf →
  - alle 3 Threads terminieren:



- Ungünstig, kein Thread terminiert →
  - Verklemmung ist möglich, aber nicht zwingend.



7.6.1 Begriffe

- Deadlock
  - Passives Warten
  - Prozesszustand BLOCKED
- Livelock
  - Aktives Warten (busy polling)
  - Prozesszustand beliebig (auch RUNNING), aber kein Fortschritt

- Deadlocks sind hierbei das geringere Übel
  - Zustand eindeutig erkennbar → Basis zur „Auflösung“ gegeben
  - Nicht so bei Livelocks

7.6.2 Bedingungen für eine Verklemmung

- 1. Wechselseitiger Ausschluss (engl. mutual exclusion)
  Betroffene Ressource ist nicht gemeinsam nutzbar.
- 2. Halten & Warten (engl. hold and wait)
  Wartender Thread besitzt Ressource und wartet auf weitere.
- 3. Keine Verdrängung (engl. no preemption)
  Ressourcen können einem Thread nicht entzogen werden.

- Sobald zusätzlich auch noch die nachfolgende Bedingung erfüllt ist, so liegt eine Verklemmung vor
- 4. Zirkuläre Wartesituation (engl. circular wait)

7.6.3 Verklemmungsvorbeugung

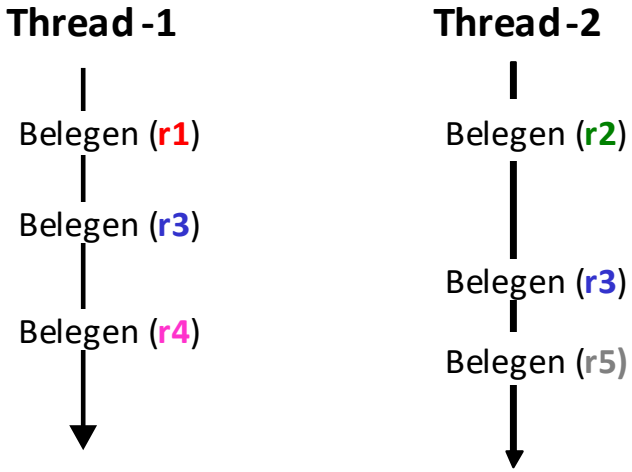
- Mindestens eine der vier Verklemmungsbedingungen darf nicht zutreffen.

- Lösung-1: Summenbelegung (preclaiming):
  - sämtliche jemals benötigten Ressourcen werden einmalig zu Beginn angefordert.
  - Schwierig, da vorab unbekannt ist, was benötigt wird

- Lösung-2: Totalfreigabe bei jeder Belegung:
  - auch hier wird eine Anforderung aus einem „besitzlosen“ Zustand vorgenommen und somit eine zirkuläre Wartesituation vermieden.

7.6.3 Verklemmungsvorbeugung

- Lösung-3: Belegung gemäß vorgegebener Ordnung:
  - Zyklen werden durch das Einhalten einer vorgegebenen Ordnung vermieden.
  - Ressourcen seien zum Beispiel aufsteigend geordnet (r1, r2, ...)



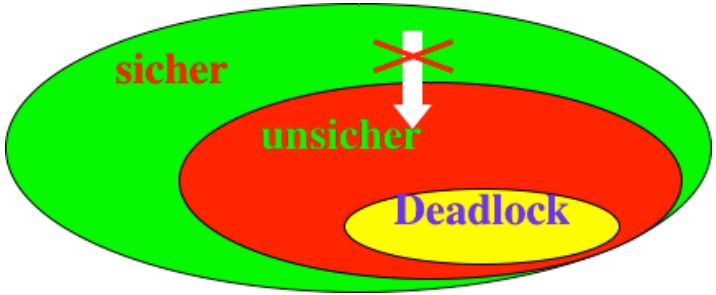
7.6.4 Vermeiden von Verklemmungszuständen

- Voraussetzung: Gesamtressourcenbedarf aller Threads muss bekannt sein.
- Für jede einzelne Ressourcenanforderung wird entschieden, ob dadurch ein Deadlock auftreten kann.
- Im ungünstigsten Fall werden alle Restanforderungen gleichzeitig gestellt.
- Könnten diese zu einem Zeitpunkt alle erfüllt werden, so heißt die aktuelle Situation sicher, andernfalls unsicher.

- Eine Anfrage nach neuen Ressourcen wird nur dann gestattet, wenn der Zustand danach sicher bleibt.

Bankier Algorithmus (Dijkstra)

- Jeder Thread nennt die maximale Anzahl Ressourcen, die er benötigt (max[i]=Kreditrahmen).
- Jeder Thread leiht sich eine Anzahl Ressourcen aus (loan[i] ≤ max[i]). (für Ressourcen mit mehrfachen Instanzen).
- Wie ein Bankier beurteilt das BS die Kreditwürdigkeit der einzelnen Threads (max) und verteilt entsprechend die Ressourcen (loans).



Bankier Algorithmus (Dijkstra)

- Sicherer Zustand (12 Ressourcen):
  - Vergabe einer Ressource an B ist zulässig.
  - Vergabe an A oder C ist unzulässig und führt zu einem unsicheren Zustand.

	loan		max
Thread A	1		4
Thread B	4		6
Thread C	5		8
Verfügbar		2	

- Unsicherer Zustand (12 Ressourcen):
  - Für keinen der beteiligten Threads kann die Terminierung garantiert werden.
  - Hoffentlich gibt bald einer der Threads wieder eine Einheit zurück.

	loan		max
Thread A	1		4
Thread B	4		6
Thread C	6		8
Verfügbar		1	

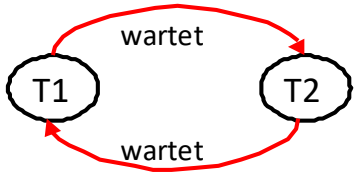
Bankier Algorithmus (Dijkstra)

Beurteilung:

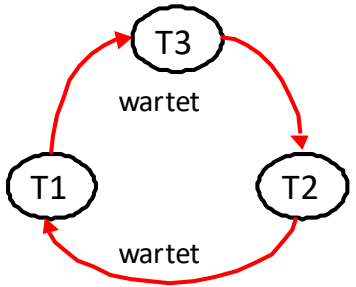
- Verhinderung von Verklemmungen trotzdem Bedingungen 1 bis 4 erfüllt sind.
- Bedarf an Ressourcen muss aber vorab bekannt sein.
- Und nur für eine feste Anzahl von Benutzern und Ressourcen.

7.6.5 Verklemmungsentdeckung

- Entdeckungsstrategien erlauben das Auftreten von Deadlocks.
- Mithilfe eines **Wartegraph (engl. wait-for graph)**:
  - Dies ist ein gerichteter Graph mit den Threads als Knoten und Wartebeziehungen als Pfeile
  - Eine Verklemmung ist genau dann vorhanden, wenn ein Zyklus im Wartegraph auftritt. →



direkter Deadlock



indirekter Deadlock

- Wird in der Praxis in Betriebssystemen nicht verwendet, da zu aufwändig.

7.6.5 Verklemmungsentdeckung

- Pragmatisches Vorgehen: Warten auf Ressource mit Timeout
- Timeout als Hinweis auf eine mögliche Verklemmung
- Verwendet in UNIX und Microsoft Windows

7.6.6 Verklemmungsauflösung

- Durchbrechen einer zirkulären Wartesituation.
- Falls ein Entzug von Ressourcen nicht möglich ist,
  so muss mindestens ein Prozess / Thread abgebrochen werden.
- Geschieht manuell durch den Benutzer.

- Standard in gängigen Betriebssystemen.

7.7 Sperrfreie Synchronisierung

- Synchronisierung über Sperren ist zeitaufwändig
  - Thread der die Sperre nicht bekommt wird blockiert
  - Nach Freigabe der Sperre kommt ein Thread in die Warteschlage des Schedulers

- Sperrfreie Synchronisierung vermeidet diese Nachteile und stützt sich „nur“ auf sequentielle Speicherzugriffe
  - Dies ermöglicht sehr kurze Latenzen zwischen Sperr-Freigabe und -Belegung
  - Ist aber meist sehr kompliziert und verwendet Busy-Polling → hohe CPU-Last
  - Nur dort sinnvoll wo Latenz sehr wichtig ist

7.7.1 Peterson Algorithmus für 2 Threads

- Wechselseitiger Ausschluss nur gestützt auf sequentiellen Speicherzugriff

- Variablen: `boolean t1Tries, t2Tries`
  - Ein Flag pro Thread
  - Zeigt an, ob der Thread die Sperre möchte
  - Wird nur von einem Thread geschrieben -> keine Race-Condition

- Variable: `int victim`
  - Wird konkurrierend von den beteiligten Threads geschrieben
  - Zeigt an welcher Thread sein Flag zuletzt geschrieben hat
  - Derjenige der später schreibt, lässt dem Schnelleren den Vortritt

7.7.1 Peterson Algorithmus für 2 Threads

```
int victim = 0;
boolean t1Tries = false;
boolean t2Tries = false;
```

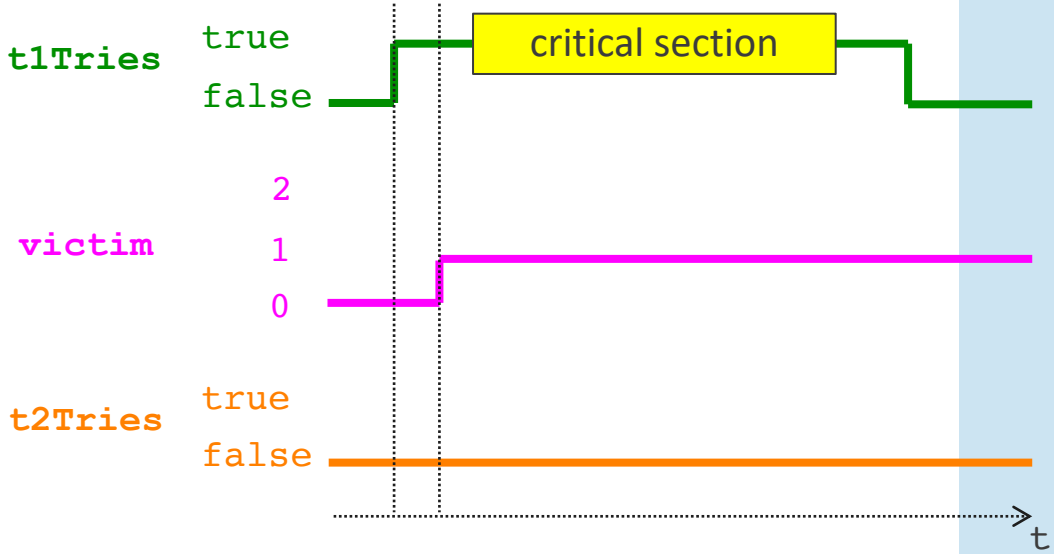
```
public void thread1() {
    t1Tries = true;
    victim  = 1;
    while (t2Tries==true &&
           victim==1) {};

    /* critical section */

    t1Tries = false;
}
```

```
public void thread2() {
    t2Tries = true;
    victim  = 2;
    while (t1Tries==true &&
           victim==2) {};

    /* critical section */

    t2Tries = false;
}
```

Fall 1: Nur Thread 1 will in den kritischen Abschnitt

```
int victim = 0;
boolean t1Tries = false;
boolean t2Tries = false;
```

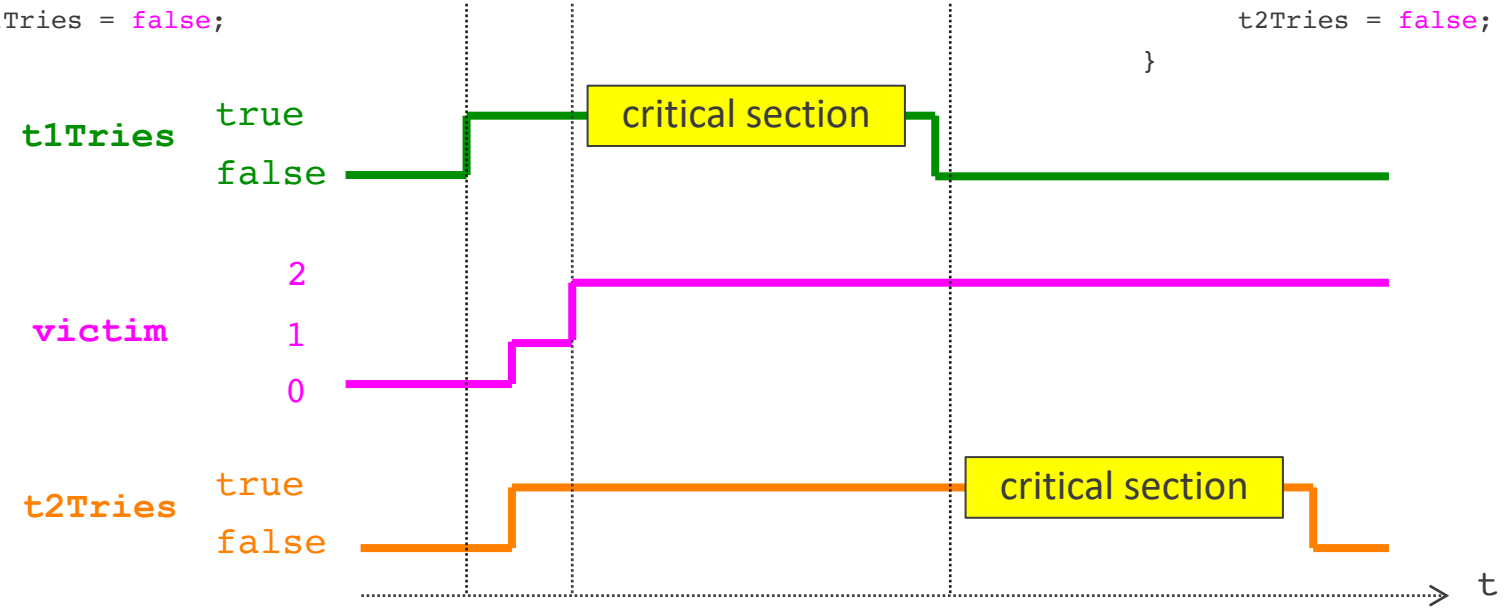
```
public void thread1() {
    t1Tries = true;
    victim  = 1;
    while (t2Tries==true &&
           victim==1) {};

    /* critical section */

    t1Tries = false;
}
```



Fall 2: Beide Threads möchten in den krit. Abschnitt

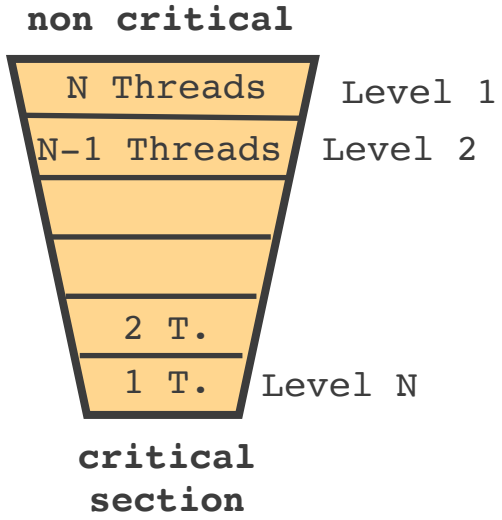
```
public void thread1() {
    t1Tries = true;
    victim  = 1;
    while (t2Tries==true &&
           victim==1) {};
      /* critical section */
      t1Tries = false;
}
```

```
public void thread2() {
    t2Tries = true;
    victim  = 2;
    while (t1Tries==true &&
           victim==2) {};
      /* critical section */
      t2Tries = false;
}
```



7.7.2 Peterson Algorithmus für N Threads

- Jeder Thread muss N Level (oder Warteräume) durchlaufen
- Auf jeder Stufe wird ein Thread blockiert, sofern N Threads konkurrierend die Levels durchlaufen



7.7.2 Peterson Algorithmus für N Threads

- Jeder Thread muss N Level (oder Warteräume) durchlaufen
- Auf jeder Stufe wird ein Thread blockiert, sofern N Threads konkurrierend die Levels durchlaufen

```
int victim[] = new int[N];     // victim auf Level L

int level[] = new int[N];      // Zeigt an, auf welchem Level
                               // ein Thread sich befindet
```

Alle Elemente beider Arrays initialisiert mit 0

7.7.2 Peterson Algorithmus für N Threads

- Pseudo-Code angelehnt an Java
- i: Thread-Nummer

```
public void lock(int i) {
   for (int L = 1; L < N; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃k!=i mit level[k] >= L) &&
         victim[L] == i ) {};
   }
}

public void unlock(int i) {
   level[i] = 0;
}
```

Stufe für Stufe

7.7.2 Peterson Algorithmus für N Threads

- Pseudo-Code angelehnt an Java
- i: Thread-Nummer

```
public void lock(int i) {
   for (int L = 1; L < N; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃k!=i mit level[k] >= L) &&
         victim[L] == i ) {};
   }
}

public void unlock(int i) {
   level[i] = 0;
}
```

Zeige an, dass Thread i in Level L eintreten möchte

7.7.2 Peterson Algorithmus für N Threads

- Pseudo-Code angelehnt an Java
- i: Thread-Nummer

```
public void lock(int i) {
    for (int L = 1; L < N; L++) {
        level[i]  = L;
        victim[L] = i;
        while ((∃k!=i mit level[k] >= L) &&
            victim[L] == i ) {};
    }
}

public void unlock(int i) {
    level[i] = 0;
}
```

Thread i wartet, solange ein anderer Thread auf gleichem oder höheren Level ist und Thread i designiertes Victim ist

7.7.2 Peterson Algorithmus für N Threads

- Pseudo-Code angelehnt an Java
- i: Thread-Nummer

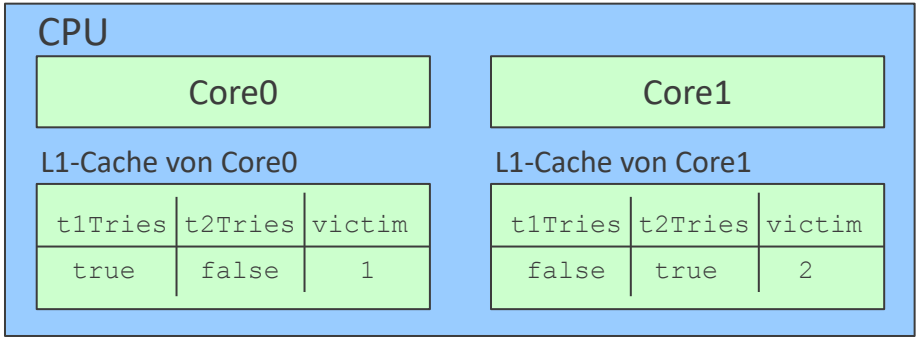
```
public void lock(int i) {
   for (int L = 1; L < N; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃k!=i mit level[k] >= L) &&
        victim[L] == i ) {};
   }
}
```

```
public void unlock(int i) {
   level[i] = 0;
}
```

Nächstes Mal müssen wieder alle Level, ab 0 durchlaufen werden

7.7.3 Peterson Algorithmus für Multicore

- Die vorhergehenden Implementierungen funktionieren auf einem Single-Core
- Bei einem Multi-Core-Prozessor sind die Schreibzugriffe auf die Variablen zwischen den Threads eventuell nicht sichtbar, da diese nur in dem jeweils pro Core separatem L1-Cache stattfinden.
- Beispiel:



7.7.3 Peterson Algorithmus für Multicore

- Deswegen bieten die x86-CPUs dafür HW-Unterstützung
  - `cmpxchg(`r_1`64/m64, `r_2`64)`: vergleiche Register r_1 oder Speicherinhalt bei `m64` mit r_2. Wenn die Inhalte gleich sind, dann wird r_2 in r_1/`m64` gespeichert
  - `xchg(`r_1`64, `r_2`64/m64)`: tausche den Inhalt von Register r_2 oder Speicherinhalt bei `m64` mit r_1.
  - `lock`: die beiden Instruktionen oben haben implizit den Präfix `lock`. Dadurch wird die nächste Instruktion atomar auf dem Bus durchgeführt und die betroffene Cache-Line wird Core-übergreifen synchronisiert
  - `mfence`: Serialisierungsoperation garantiert, dass alle Lade- und Speicheroperationen vor dem `mfence`-Befehl global sichtbar werden, vor anderen Lade- und Speicheroperationen nach dem `mfence`-Befehl.
    - Instruktion ist teuer, daher nur dort verwenden wo notwendig
  - ...

7.7.3 Peterson Algorithmus für Multicore

- Auch GCC bietet hierfür Unterstützung: atomic operations
  - void **__atomic_store** (type *ptr, type *val, int memmodel)
  - void **__atomic_exchange** (type *ptr, type *val, type *ret, int memmodel)
  - ...
  - memmodel: 6 Speichermodelle, siehe auch C++11 Standard
    - **__ATOMIC_SEQ_CST**: Full barrier in both directions and synchronizes with acquire loads and release stores in all threads.
    - ...
    - **__ATOMIC_RELAXED**: No barriers or synchronization.
  - https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/_005f_005fatomic-Builtins.html

Hörsaal-Aufgabe

- Schreiben Sie das Beispiel zum Lost-Update von S.6 aus diesem Kapitel um und synchronisieren Sie den Zugriff auf die Variable sync mithilfe des Peterson-Algorithmus (für zwei Threads).

- Testen Sie Ihr Programm mit zwei Threads auf einem Multicore-Rechner
  - Zunächst ohne `mfence`
  - Und dann mit `mfence`