Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

6 Prozesssynchronisierung

Zusammenarbeitende Prozesse haben sich gemeinsame Daten:
- Threads (LWPs) teilen sich gesamten Prozessraum
- Prozesse haben entweder gemeinsamen (shared) Speicher oder teilen sich Dateien

Problem: Inkonsistenzen in (Verwaltungs-)Daten durch zeitlich überlagerte Zugriffe nebenläufiger Prozesse auf diesselben Daten.
Verallgemeinert: nebenläufiger Zugriff auf geteilte Resourcen

Nebenläufigkeit:
- Mehrprozessor-System: Prozessoren bearbeiten unabhängig voneinander verschiedene Prozesse
- Einprozessor-System und Mehrprozessor-System: Interrupt kann asynchron zum Prozessablauf die Bearbeitung einer Unterbrechungsroutine bewirken. Prozessumschaltung kann unvorhergesehen im Prozessablauf auftreten.

Beispiel: Banksystem, Einzahlung und Abhebung laufen nebenläufig
- Kontostand: 1000 EUR
- Einzahlung von 3000 EUR
- Abhebung von 500 EUR

Zeit	*Prozess 1: Einzahlung*	*Prozess 2: Abhebung*
T0	Liest: Kontostand 1000 EUR	-
T1	-	Liest: Kontostand 1000 EUR
T2	-	Bucht: -500 EUR
T3	-	Schreibt: Kontostand 500 EUR
T4	Bucht: +3000 EUR	-
T5	Schreibt: 4000 EUR	-

Kontostand ist 4000 EUR, richtig wäre 3500 EUR!

Die Transaktion „Lesen-Buchen-Schreiben" wird als **kritischer Abschnitt oder kritischer Bereich** bezeichnet.

Definition Wettlauf (Race-Condition):
Eine Situation, in der mehrere Prozesse auf gemeinsame Daten zugreifen und in der das Ergebnis von der – zufälligen – Reihenfolge der einzelnen Befehle abhängt, heisst Race-Condition (Wettlaufsituation).

Um einen Wettlauf zu verhindern, müssen wir gewährleisten, dass nur ein Prozess zu einer Zeit Zugriff auf die Variable hat -> Prozess-Synchronisierung

Beispiel: Produzenten-Consumer-System
- Der Produzent erzeugt Daten, die vom Consumer verarbeitet werden

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Für den Datenaustausch wird ein Datenpuffer verwendet:
  - Datenpuffer bietet die Funktionen insert_item() und remove_item()
  - Maximale Größe des Datenpuffers: N
  - Implementiert z.B. als Array:

```
const int N=100;

int buffer[N];
int in = 0;       // Zeider auf nächster freier Pufferplatz zum Schreiben
int out = 0;      // Zeiger auf nächster Element zum Lesen

void insert_item( int item ) {
      buffer[in] = item;
      in = (in+1) % N;
}

int remove_item( ) {
      int retItem;
      retItem = buffer[out]
      out = (out+1) % N;
      return retItem;
}
```

- Der Produzent- und der Consumer-Prozess greifen auf den Puffer zu; sie müssen die Anzahl der Elemente im Puffer selbst verwalten[2]

Zählvariable (gemeinsam für Producer und Consumer):

```
int counter;
```

Produzent-Prozess:

```
while(1) {
      // warte, falls Puffer voll ist
      while( counter==N )
            ; // tue nix
      insert_item( item );
      counter++;
}
```

Consumer-Prozess:

```
while(1) {
      // warte, bis ein Element im Puffer bereit steht
      while( counter==0 )
            ; // tue nix
      item = remove_item();
      counter--;
}
```

Beide Prozesse sind für sich genommen richtig, zusammen ergibt sich jedoch ein Fehler!

[2] Dies ist ein schlechtes Design. Ein guter Datenpuffer würde seine Größe selbst verwalten, und das nicht den benutzenden Programmen überlassen. Was aber soll der Puffer machen, wenn er überfüllt wird? Zur Verdeutlichung müssen in den Beispielen der P- und der C-Prozess selbst die Anzahl der Elemente im Puffer überwachen.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Der P-Prozess führt counter++ aus, das im Assembler etwa so implementiert wird:
```
P1: Lies Register1 ein aus Speicher counter
P2: Erhöhe Register1 um 1
P3: Speichere Register1 nach Speicher counter
```
- Der C-Prozess führt counter-- aus, das im Assembler etwa so implementiert wird:
```
C1: Lies Register2 ein aus Speicher counter
C2: Erniedrige Register2 um 1
C3: Speichere Register2 nach Speicher counter
```

Annahme: counter hat den Wert 5
Mögliche Abläufe:

Tabelle 1: Ablauf 1 (korrekt)

Schritt	*Register1*	*Register2*	*Counter*
	??	??	5
P1	5	??	5
P2	6	??	5
P3	6	??	6
C1	??	6	6
C2	??	5	6
C3	??	5	5

Tabelle 2: Ablauf 2 (falsch)

Schritt	*Register1*	*Register2*	*Counter*
	??	??	5
P1	5	??	5
P2	6	??	5
C1	Unverändert	5	5
C2	Unverändert	4	5
C3	Unverändert	4	4
P3	6	??	6

Tabelle 3: Ablauf 3 (falsch)

Schritt	*Register1*	*Register2*	*Counter*
	??	??	5
P1	5	??	5
P2	6	??	5
C1	Unverändert	5	5
C2	Unverändert	4	5
P3	6	Unverändert	6
C3	Unverändert	4	4

Problem: beide Prozesse verändern die Variable *counter* zur selben Zeit.

6.1 Kritische Bereiche (Critical Section)

Gegeben seien n Prozesse { P0, P1, ... PN-1 }. Jeder Prozess hat einen kritischen Code-Bereich (Critical Section), in dem er gemeinsame Daten liest oder verändert, Dateien beschreibt, etc.

Nach Dijkstra ist zu beachten:

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

1. **Gegenseitiger Ausschluss (Mutual exclusion):** Wenn ein Prozess Pi in seinem kritischen Bereich ist, dann darf kein anderer Prozess in seinem kritischen Bereich laufen.
2. Es dürfen keine Annahmen über die Abarbeitungsgeschwindigkeit Anzahl der Prozesse bzw. Prozessoren gemacht werden.
3. Kein Prozess der ausserhalb seines kritischen Bereiches läuft, darf einen anderen Prozess blockieren.
4. **Gerechtigkeit (Fairness Condition):** Jeder Prozess, der am Eingang eines kritischen Abschnittes wartet, muss garantiert werden, dass er den KA irgendwann betreten darf (kein ewiges Warten)

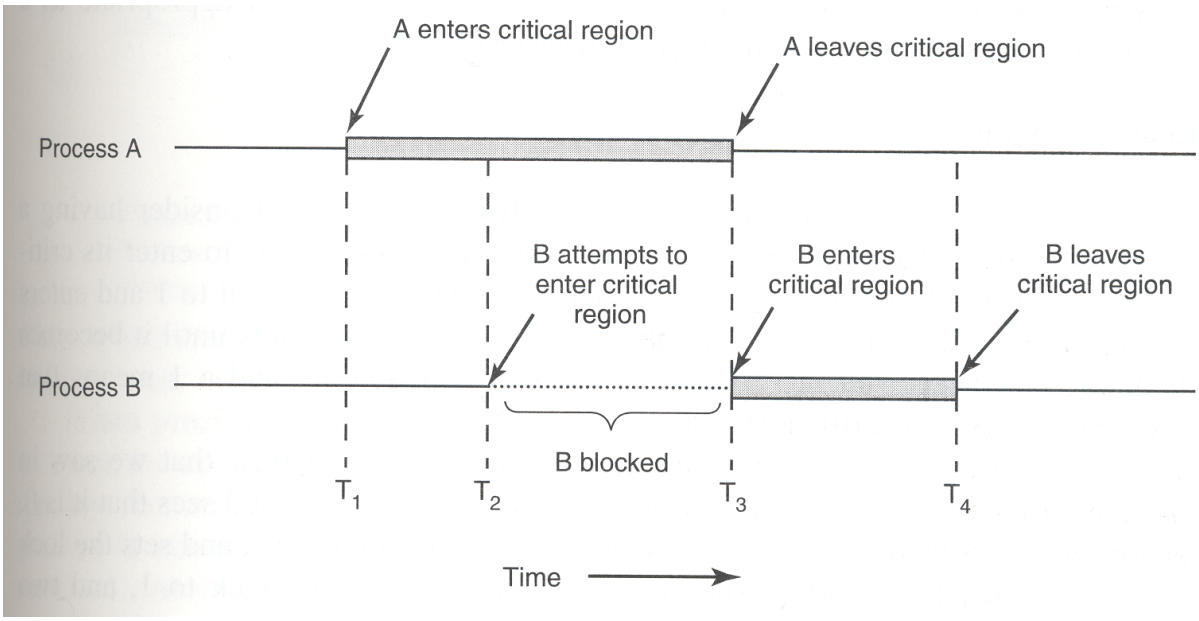


Abbildung 52: Schema kritischer Bereich

Allgemeine Struktur:
- Eintrittsbereich
- Kritischer Bereich
- Austrittsbereich
- Verbleibender Bereich

Critical-Section-Problem: Wie muss ein Verfahren aussehen, um Datenkonsistenz zu garantieren?

6.2 Lösungen

6.2.1 Abschalten der Interrupts

Einfachste Lösung: Interrupts im kritischen Bereich abschalten.
Keine Interrupts->kein Prozesswechsel
- Wird häufig in einfachen Echtzeit-Systemen verwendet,
- oder innerhalb des Kernels für kritische Bereiche

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Aber: Abschalten der Interrupts im User-Prozess: was passiert, wenn das User-Programm in eine Endlosschleife geht? -> Das gesamte OS bleibt stehen!

6.2.2 Zwei-Prozess Lösungen

6.2.2.1 Software-Lösung Versuch 1: Sperr-Variable (lock variable)

Wir verwenden eine Schutz-Variable:

```
int lock=0;


L1:   while (lock!=0 )        // warte bis KA frei ist
L2:         ;                 // tue nix
L3:   lock = 1;               // wir betreten KA: zunächst sperren
L4:   // in KA: tue irgendwas
L5:   lock = 0;               // KA verlassen
```

Definition Spin Lock:
Eine Sperre, die in einer Warteschleife auf die Freigabe wartet, heisst Spin-Lock.

L1-L3: EnterCriticalSection
L4: LeaveCriticalSection

Problem: funktioniert so nicht!
Sei lock = 0:

Zeit	*P1*	*P2*
T0	L1	-
T1	-	L1
T2	-	L3
T3	-	L4
T4	L3	-
T5	L4 Fehler!	

6.2.2.2 Lösungsversuch 2: Abwechselnder Zugang (strict alternation)

Wir verwenden eine Anzeiger-Variable. Jeder Prozess kommt abwechselt dran.

Gegeben: Zwei Prozesse, jeder bekommt eine Nummer zugeteilt (0 oder 1):

Prozess i (i = 0 oder i = 1):

```
L1:   while (turn!=i)             // warte bis ich an der Reihe bin
L2:         ;                     // tue nix
L3:   // in KA: tue irgendwas
L4:   turn = 1 - i;               // KA verlassen
```

Mutual Exclusion: ok, da nicht beide Prozesse gleichzeitig in der KA sein können!

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Fairness: Nein, da die Prozesse können nur abwechselt drankommen:
- Wenn als Prozess 0 in der KA war, ist turn = 1
- Kommt nun P0 wieder in die KA, muss er warten!
- Dies wiederspricht Forderung 3: Kein Prozess darf auf einen anderen warten, der nicht in der KA ist!

6.2.2.3 Peterson's Lösung

Kombination aus den beiden vorigen Lösungen:
- Feld von zwei Variablen flag[0] und flag[1] zeigt an, dass ein Prozess den KA betreten will
- Eine turn-Variable entscheidet, welcher von beiden drankommt

Peterson's Lösung (nach [2]):[3]

```
// Zwei Prozesse P0 und P1
int turn;               // Wechselnder Zugang, Initialisierung egal
int flag[2];            // Zutrittswunsch

// Prozess i (i=0 oder 1)
int other = 1 - i;

      // melde Wunsch an
L1:   flag[i] = TRUE;
      // anderer Prozess hat Vortritt
L2:   turn = other;
      // bin ich der einzige?
L3:   while (flag[other]
L4:         && (turn==other))
L5:               ; // tue nix
L6:   // in KA: tue irgendwas
      // KA verlassen
L7:   flag[i]=FALSE;
```

[3] In der Vorlesung vom 3.5.2004 habe ich die Lösung nach Tanenbaum präsentiert, die sich in der Benutzung der turn-Variable unterscheiden. Beide Lösungen sind richtig, und der Beweis ist derselbe.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Beweis:
Zu Verdeutlichung hier der Code für die beiden Prozesse P0 und P1:

```
int turn=0;          // Wechselnder Zugang
int flag[2];         // Zutrittswunsch
```

```
      // Prozess 0
      // melde Wunsch an
L1:   flag[0] = TRUE;
      // P1 hat Vortritt
L2:   turn = 1;
      // bin ich der einzige?
L3:   while (flag[1]
L4:         && (turn==1))
L5:               ; // tue nix
L6:   // in KA: tue irgendwas
      // KA verlassen
L7:   flag[0]=FALSE;
```

```
      // Prozess 1
      // melde Wunsch an
L1:   flag[1] = TRUE;
      // P0 hat Vortritt
L2:   turn = 0;
      // bin ich der einzige?
L3:   while (flag[0]
L4:         && (turn==0))
L5:               ; // tue nix
L6:   // in KA: tue irgendwas
      // KA verlassen
L7:   flag[1]=FALSE;
```

Wir müssen zeigen:

1. **Mutual Exclusion** – nur ein Prozess kann zu einem Zeitpunkt in der KA sein.
   - Es reicht zu beweisen, dass wenn ein Prozess in der KA ist, der andere diese nicht betreten kann. Annahme: P0 ist in KA, P1 nicht.
   - Dann gilt: flag[0] = TRUE.
   - Sei t_x der Zeitpunkt, an dem P0 die KA betreten hat, d.h. zu t_x wurde die while-Schleife verlassen wurde.
   - Fall 1: Zum Zeitpunkt t_x war flag[1] = FALSE. Dann kann P1 die KA nicht betreten, da es selbst turn = 0 setzt.
   - Fall 2: Zum Zeitpunkt t_x war flag[1] = TRUE.
   - Fall 2a: Falls turn = 0, dann kann P1 den KA nicht betreten.
   - Fall 2b: Falls turn = 1, dann kann P0 den KA nicht betreten haben => Widerspruch zur Annahme, dass P0 in KA ist!
   - Nach t_x kann P1 die KA nicht betreten, da flag[0] = TRUE und turn = 0 ist solange P0 in der KA ist.
2. **Keine Annahmen über Geschwindigkeit, etc.**
3. **Blockierung:**
   - Wenn ein Prozess i die KA verlässt, so setzt er flag[i] = FALSE, und somit blockiert er den anderen Prozess nicht mehr
4. **Fairness**: jeder Prozess kommt in endlicher Zeit dran.
   - Dazu reicht es zu zeigen: Wenn P1 auf Zutritt in die KA wartet, dann kommt es auch irgendwann (nach endlicher Zeit) dran).
   - Annahme: P1 wartet an der Schleife
   - Dann gilt: flag[0] = TRUE und turn = 0.
   - Wenn P0 in der KA ist, dann wird es diese irgendwann verlassen -> flag[0] = FALSE -> P1 kann KA betreten
   - Will in der Zwischenzeit P0 wieder die KA betreten, so führt es L1 aus: flag[0] ist wieder TRUE. Im nächsten Schritt wird jedoch turn = 1, so dass auf jeden Fall zunächst P1 drankommt!

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

6.2.3 Test And Set Lock (TSL)

Einfache Lösung für ein Lock durch Hardware Unterstützung!
Spezielle Hardware-Anweisung TSL RX, LOCK (TSL = "Test and Set Lock"):

- Liest den Speicherinhalt an der Stelle LOCK in das Register RX ein und speicht einen Wert ungleich Null an der Stelle LOCK ab
- Die gesamte Operation ist unteilbar – kein anderer Prozess (oder Prozessor) kann auf die Speicheradresse zugreifen solange die Anweisung nicht beendet wurde
- Bei mehreren CPUs sperrt die zugreifende CPU kurzzeitig den Speicherbus und verhindert so einen Zugriff der anderen CPUs

Wie kann man damit einen kritischen Bereich schützen?

- Wir benutzen eine gemeinsame Variable *lock* als Sperrvariable.
- Wenn *lock* Null ist, kann jeder Prozess die Variable auf 1 setzen und den kritischen Bereich betreten.
- Ist *lock* jedoch nicht Null, so wartet der Prozess bis *lock* Null ist

```
enter_region:
      TSL RX, LOCK            | copy lock to register
      CMP RX,#0               | was lock zero?
      JNE enter_region        | no -> loop
      RET                     | return to caller: enter critical section

leave_region:
      MOVE LOCK,#0            | set lock to zero
      RET                     | done
```

6.2.4 Sleep And Wakeup

Nachteil der bisherigen Lösungen: "busy wait", d.h. der Prozess läuft in einer Endlosschleife! Auf einem Uniprozessor-System ist das Zeitverschwendung!

Definition Spin-Lock:
Ein Spin-Lock ist eine Sperre, an der ein Prozess in einer Schleife auf den Zugang zu einem kritischen Abschnitt wartet.

- Bei den bisherigen Beispielen handelt es sich um ein Spin-Locks

Was wäre besser? Anstelle in einer "Endlosschleife" zu laufen, sollte der Prozess schlafen gelegt werden und ein anderer Prozess aufgeweckt werden!
-> sleep-Befehl und wakeup-Befehl

- sleep() ohne Parameter, wakeup() mit Angabe der PID (oder Thread-ID)
- oder Zuordnung über einen eindeutigen Parameter: sleep(void*) und wakeup(void*)

Produzenten-Prozess: (hat ein neues Element in *nextProduced*)

```
while(1) {
      // L1: warte, falls Puffer voll ist
      if( counter==N ) sleep();
      insert_item( item );
      counter++;
      // Falls Queue leer war: wecke Consumer auf
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
        if(counter==1)wakeup(consumer);
}
```

Consumer-Prozess: (liest ein Element nach *nextConsumed* aus)

```
while(1) {
        // warte, bis ein Element im Puffer bereit steht
        if( counter==0 ) sleep();
        item = remove_item();
        counter--;
        // Falls Queue voll war: wecke Produzent auf
        if( counter==BUFFER_SIZE-1 ) wakeup(producer);
}
```

Achtung! Obiger Code hat einen fatalen Fehler:

- Der Produzent liest in L1 die Counter-Variable ein und testet, ob der Puffer voll ist.
- Angenommen, der Puffer ist voll. Dann legt sich der P-Prozess schlafen.
- Weiter angenommen, dass *nach* dem Vergleich, jedoch *vor* dem sleep-Befehl ein Prozesswechsel stattfindet.
- Nun kommt der C-Prozess dran: er nimmt ein Element aus der Queue, verarbeitet dieses und weckt nun den P-Prozess auf. Da dieser jedoch nicht schläft, geht der wakeup-Befehl ins Leere.
- Nach einem weiteren Prozesswechsel kommt wieder der P-Prozess dran: er legt sich schlafen, und schläft nun für immer!

6.2.5 Semaphoren

Deshalb schlug Dijkstra (1965) eine weitere Lösung vor:

- Integer-Variable, welche die Anzahl der Wakeup-Signale zählt
- Eine Semaphore hat den Wert Null, wenn keine Wakeup-Signale vorliegen, oder einen positiven Wert der der Anzahl der noch nicht verarbeiteten Wakeups entspricht
- Dijkstra schlug zwei Operationen vor, *up()* und *down()* (bzw. P() für holl. *proberen*, testen und V() *verhoegen*, erhöhen)
- Die Operation down() prüft, ob der Wert der Integer-Variable größer Null ist:
    - Falls ja, wird die Variable um eins erniedrigt, und das Programm rechnet weiter.
    - Ist die Variable Null, so wird der Prozess schlafen gelegt
- Die Operation *up()* erhöht den Wert der Semaphore um eins. Falls ein oder mehrere Prozesse auf die Semaphore warten, so wird einer vom System ausgewählt und er darf seine *down()*-Operation beenden. Die Semaphore wird dann wieder den Wert Null haben, dafür wartet jedoch ein Prozess weniger auf die Semaphore.
- Die Operationen *up()* und *down()* müssen atomar, d.h. unteilbar sein. Wenn ein Prozess ein *up()* oder *down()* auf eine Semaphore durchführt, so darf kein anderer Prozess auf die Semaphore zugreifen.

Beispiel: Mutual Exclusion mit Semaphoren

```
semaphore s(1);   // initialisiere Semaphore mit 1

// Prozess rechnet
// ...
down(s);                // Eintritt in den kritischen Abschnitt
// ....                 Kritischer Abschnitt
up(s);                  // kritischen Abschnitt verlassen
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Binäre Semaphore: Mutex (für Mutual Exclusion), hat nur zwei Werte: 0 und 1, wird für gegenseitigen Ausschluss verwendet.

6.2.5.1 Implementierung

- Typischerweise als Systemaufruf
- Mutual Exclusion:
  - Ein-Prozessor-System: Bei Zugriff auf die Semaphore werden kurzzeitig die Interrutps gesperrt;
  - Mehr-Prozessor-Lösung: mittels TSL-Anweisung
- Warten auf das Aufwachen
  - Spin-Lock (Endlosschleife): ineffektiv
  - besser: jede Semaphore verwaltet eine Liste der schlafenden Prozesse; ein *up()* wählt einen Prozess aus der Liste auf und weckt ihn auf.

```
// Beispiel-Implementierung für Semaphore (Pseudo-Code)
// hier fehlt noch der Zugangsschutz für die Semaphore!

typedef struct {
      int value;
      struct process *p_list;
} semaphore;

void down(semaphore s)
{
      s.value--;
      if(s.value<0) {
            // füge den Prozess zu der Warteschlange hinzu
            // blockiere den Prozess
      }
}

void up(semaphore s)
{
      s.value++;
      if( s.value<=0 ) {
            // entferne einen Prozess von der Warteschlange
            // wecke den Prozess auf!
      }
}
```

6.2.5.2 Beispiel: Consumer-Producer mit Semaphore

Zwei Semaphoren für die Anzahl, eine Mutex um den Puffer zu schützen:

```
semaphore empty=N;      // Ist noch Platz in der Queue?
semaphore full=0;       // Sind Elemente in der Queue verhanden?
semaphore mutex=1;      // Puffer: gegenseitiger Ausschluss
```

Produzenten-Prozess: (hat ein neues Element in *nextProduced*)

```
while(1) {
      // teste, ob Platz in der Queue ist
      down(&empty);
      // schütze Zugriff auf Puffer
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
        down(&mutex);
        insert_item( item );
        up(&mutex);
        // Ein weiteres Element ist in der Queue verfügbar
        up(&full);
}
```

Consumer-Prozess: (liest ein Element nach *nextConsumed* aus)

```
while(1) {
        // teste, ob ein Element verfügbar ist
        down(&full);
        // schütze Zugriff auf Puffer
        down(&mutex);
        item = remove_item();
        up(&mutex);
        // Ein weiteres Element hat Platz in der Queue
        up(&empty);
}
```

Semaphoren werden im Beispiel für zwei verschiedene Zwecke verwendet:
1. Mutual Exclusion: *mutex*
2. Prozesssynchronisierung (Benachrichtigung durch Aufwecken): *empty, full*

6.2.6 Monitors

Semaphoren sind besser als bisherige Lösung, jedoch immer noch sehr kompliziert!

- Obiges Beispiel, P-Prozess: wenn down(&empty) und down(&mutex) vertauscht werden, gibt es ein Deadlock!
- Deshalb haben Hoare (1974) und Brinch Hansen (1975) ein hochwertiges Synchronisierungsmittel vorgeschlagen, genannt **monitor**
- Ein Monitor ist eine Gruppe von Funktionen, Variablen und Datenstrukturen, die in einem Modul (oder Klasse) zusammengruppiert sind
- Datenkapselung: Prozesse können die Funktionen aufrufen, jedoch sind die inneren Daten vor direktem Zugriff geschützt
- Beispiel (Pidgin-Pascal):

```
monitor example
        integer i;
        condition c;

        procedure producer();
        .
        .
        .
        end;

        procedure consumer();
        .
        .
        .
        end;
end monitor;
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Mutual exclusion: nur ein Prozess kann zu einer Zeit innerhalb des Monitors laufen. Dies wird von der Programmiersprache sichergestellt.
- Aber wie soll ein Prozess auf einen anderen warten? Zum Beispiel wenn der Produzent im Producer-Consumer-Beispiel kein weiteres Element in der Queue ablegen kann?
- Deshalb Einführung von condition-Variablen, mit zwei Operationen: *wait()* und *signal()*.
- Wenn ein Prozess 1 auf eine condition-Variable *c* warten will, so ruft er *wait(c)* auf. Der Prozess 1 geht in den WAITING-Zustand über. Gleichzeitig kann der Monitor von einem anderen Prozess P2 betreten werden.
- Mittels s*ignal(c)* kann Prozess P2 den Prozess 1 wieder aufgewecken.
- Es muss vermeiden werden, dass zwei aktive Prozesse im Monitor laufen:
    - Lösung nach Hoare: der laufende Prozess P2 wird suspendiert, während der eben erwachte Prozess P1sofort weiterläuft; verlässt P1 den Monitor, kann auch P2 wieder weiterrechnen.
    - Lösung nach Brinch-Hansen: Der Prozess P2, der *signal(c)* aufruft, verlässt den Monitor sofort; dann kann P1 im Monitor verbleiben.
    - Weitere Lösung: P1 wird zwar aufgeweckt, kann jedoch er im Monitor weiterrechnen, wenn P2 den Monitor verlassen hat (Java-Lösung)

Beispiel: Producer-Consumer mit Monitore (nach [1])

```
Monitor ProducerConsumer
{
       final static int N = 5;    // Maximale Anzahl an Items im Puffer
       static int count = 0;      // Zähler für Items im Puffer
       condition full = false;    // Bedingungsvariable: Puffer ist voll
                                  //(zunächst nicht voll)
       condition empty = true;    // Bedingungsvariable: Puffer ist leer
                                  // (Anfangszustand)

       // Operation zum Einfügen eines Items im Puffer
       void insert(item: integer) {
          if (count == N)
             wait(full);          // Warten, bis Puffer nicht mehr voll ist
          // Füge  Item ein
          count+=1;
          if (count == 1)
             signal(empty);       // Signalisieren, dass Puffer nicht mehr leer ist
       }

       // Operation zum Entfernen eines Items aus dem Puffer
       int remove() {
          if (count == 0)
             wait(empty);         // Warten, bis Puffer nicht mehr leer ist
          // Entferne Item
          count--;
          if (count == (N-1)
             signal(full);        // Signalisieren, dass Puffer nicht mehr voll ist
            return (item);
       }
}
```

Benutzung des Monitors:

```
class UseMonitor
{
     ProducerConsumer mon = new ProducerConsumer();
     ...
     // Produzent, der Items erzeugt
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
        void producer() {
           while (true) {
              // Erzeuge Item
              mon.insert(item);     // Einfügen des Items in den Puffer,
                                    // evtl. Muss gewartet werden
           }
        }

        // Konsument, der Items verwendet
        void consumer() {
           while (true) {
              item = mon.remove();  // Hole ein Item aus dem Puffer
                                    // warte, wenn Puffer leer ist
              // Konsumiere ein Item
           }
        }
     }
```

6.2.7 Weitere Verfahren zur Prozesssynchronisierung

Haben wir schon bei Interprozesskommunikation (Kapitel 3) besprochen:
- Message Passing
- Pipes
- Sockets
- RPC
- sowie Atomic Transactions:
    - werden in Datenbanken benutzt, um die Integrität der Daten sicher zu stellen
    - Ein kritischer Bereich wird entweder ganz ausgeführt oder gar nicht

Beispiel für Transaktionen
- Vorbereitung der TA: BEGIN_TRANSACTION
- Durchführen der TA: COMMIT_TRANSACTION
- Aufgeben (abbrechen) der TA: CANCEL_TRANSACTION
    - Alle durchgeführten Operation (seit BEGIN_TRANSACTION) werden rückgängig gemacht
    - Wird auch *Rollback* genannt

6.3 Beispiele

6.3.1 Windows 2000

6.3.1.1 Interlocked-Funktionen

Einfache Thread-sichere Funktionen:
- *InterlockedIncrement(DWORD*par):* erhöht die Variable *par um 1
- *InterlockedDecrement(DWORD*par):* erniedrigt die Variable *par um 1
- weitere Interlocked-Funktionen, z.B. *InterlockedExchanged()*, etc.

6.3.1.2 Critical Section

Bietet gegenseitigen Ausschluss für Threads innerhalb eines Prozesses
- Sehr effektiv
- Implementiert als Spin-Lock
- Verwendung:

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
// Definition
CRITICAL_SECTION myCriticalSection;

// Initialisierung
InitializeCriticalSection( &myCriticalSection );

// Betreten und verlassen des kritischen Abschnitts mittels
EnterCriticalSection( &myCriticalSection );
// im kritischen Abschnitt...
LeaveCriticalSection( &myCriticalSection );
```

- Weitere Befehle

```
// teste ob die KA belegt ist
BOOL isLocked = TryCriticalSection( &myCriticalSection );
// Löschen der Critical Section
DeleteCriticalSection( &myCriticalSection );
```

6.3.1.3 Dispatcher Objects: Events, Mutex, Semaphore

- Dispatcher Objekte sind Win32 Kernel Objekte. Ein Kernel-Objekt wird über eine HANDLE identifiziert.
- Erzeugen der Objekte: *CreateEvent(), CreateMutex(), CreateSemaphore()*
- Benannte Objekte: Beim Erzeugen wird ein Name (string) angegeben. Damit können Prozess-übergreifende Objekte angelegt werden. Typischerweise werden GUID (global unique identifier) zur eindeutigen Identifizierung der Objekte verwendet (z.B. {11EE3539-E892-4409-BF6B-C5778F3472EB})
- Warten auf ein Objekt (down()-Operation) mittels *WaitForSingleObject()* (ein Objekt) oder *WaitForMultipleObject()* (mehrere Objekte)
- Hochzählen (up()-Operation): Semaphore: *ReleaseSemaphore(),* Mutex: *ReleaseMutex(),* Event: *PulseEvent()*

6.3.2 Unix

6.3.2.1 Semaphore

- *Semget() zum Erzeugen einer Semaphore und zum Aufnehmen einer Verbindung zu einer Semaphore*
- *semop() für die Semaphore-Operation hochzählen (up(), V()) und Herunterzählen (down(), P()) und zum Abfragen der Semaphore*
- *semctl() zum Setzen und Abfragen einer Semaphore. Beispielsweise kann mit der Operation abgefragt werden, wieviele Prozesse gerade durch das Warten auf die Semaphore blockiert sind.*

6.3.3 Java

- Unterstützt Monitore
- *synchronized*-Keywort schützt Objekte vor gleichzeitigem Zugriff
- Innerhalb einer synchronized-Umgebung können Threads mittels *wait()* schlafen gelegt werden
- Mittels *notify()* oder *notifyAll()* werden sie wieder aufgeweckt
- Extra Übung!

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

6.4 Klassische Probleme der Synchronisierung

6.4.1 Bounded-Buffer (Producer-Consumer-Problem)

- Ein Produzent schreibt in einen Puffer begrenzter Größe, ein Consumer liest Elemente aus dem Puffer
- Produzent und Consumer müssen synchronisiert werden:
    - Wenn der Puffer voll ist, muss der Produzent auf den Consumer warten
    - Wenn der Puffer leer ist, muss der Consumer auf den Produzenten warten
    - Typischerweise dürfen nicht beide gleichzeitig auf den Puffer zugreifen

6.4.2 Readers-Writers-Problem

Problem:

- Lesender und schreibender Zugriff auf gemeinsame Daten
- Beliebig viele Leser können gleichzeitig lesen
- Bei einem Schreibzugriff müssen jedoch die Daten für alle anderen (Leser und Schreiber!) gesperrt sein

Beispiel:

- ein Leser R1 liest Daten
- nun möchte gleichzeitig ein Leser R2 die Daten lesen, während ein Schreiber W1 die Daten ändern will
- Möglichkeit A: der Leser R2 darf die Daten lesen, dies ist ja gleichzeitig zu R1 möglich. W1 muss warten.
  Problem: es können immer neue Leser kommen, und W1 kommt nie dran.
- Möglichkeit B: R2 muss warten, bis W1 die Daten geschrieben hat. In der Zeit können weitere Leseanforderungen kommen, die auch alle warten müssen, d.h. ein Schreiber blockiert viele Leser.
- Wichtig in Datenbanken, wo viele Clients gleichzeitigen Zugriff auf die Daten haben

6.4.3 Fünf speisende Philosophen

- Klassisches Problem der Prozesssynchronisierung
- 5 Philosophen sitzen an einem rundem Tisch mit 5 Tellern und je einer Gabeln dazwischen (insgesamt 5 Gabeln)
- Philosophen denken eine Weile nach, und dann bekommen sie Hunger und möchten essen
- Sie brauchen jedoch zum essen beide Gabeln (links und rechts von ihrem Teller). Wenn ein Philosoph beide Gabeln bekommen hat, isst er eine Weile, dann legt er sie wieder hin.
- Wie sollen sich die Philosophen verhalten, damit
    a) jeder drankommt (keiner verhungert)
    b) keiner umsonst warten muss?

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Hier der Tisch:

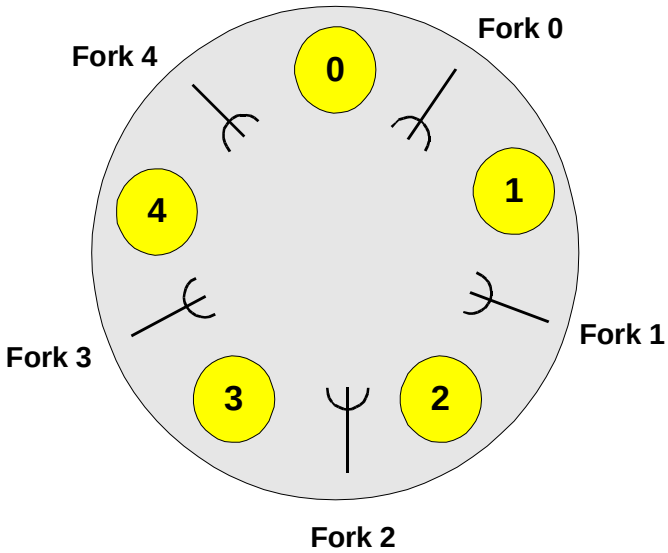


Abbildung 53: Ein Philosophen-Esstisch

Vorschlag 1:

```
#define N 5       // fünf Philosophen

void philosopher(int i) {
      while(1) {
            think();                 // denken
            take_fork(i);            // linke Gabel
            take_fork( (i+N-1)%N ); // rechte Gabel
            eat();
            put_fork(i);             // linke Gabel niederlegen
            put_fork( (i+N-1)%N );  // rechte Gabel niederlegen
      }
}
```

Problem: was passiert, wenn z.B. alle Philosophen gleichzeitig die linke Gabel nehmen? Dann warten sie ewig auf die rechte... sie "verhungern"!

Vorschlag 2: Ein Philosoph nimmt die linke Gabel, dann testet er, ob die rechte verfügbar ist. Falls nicht, so legt er die linke Gabel wieder hin und wartet eine Weile.
Problem: Alle Philosophen nehmen die linke Gabel, und legen sie gleich wieder hin; das wiederholt sich in alle Ewigkeit...

Vorschlag 3: Dann warten sie eben eine zufällig verteilte Zeit; irgendwann wird es schon mal passen.
Problem: Würden sie z.B. eine Kernkraftwerkssteuerung einem Zufalls-Algorithmus anvertrauen?

Vorschlag 4: Gesamten Tisch durch eine Mutex schützen. Der erste Philosoph, der eine Gabel aufnimmt, bekommt den ganzen Tisch, dann kann er auch eine zweite Gabel aufnehmen.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Problem: Keine Starvation, fairer Algorithmus, aber Resourcen-Verschwendung: es kann immer nur ein Philosoph zu einer Zeit essen.

Lösung aus 1 und 4: Wie Vorschlag 4, nur lassen wir bis zu 4 Philosophen an den Tisch. Wenn 4 Philosophen essen wollen, so kann einer auch immer essen.
Problem mit Vorschlag 1 tritt auf, wenn alle 5 Philosophen gleichzeitig essen wollen. Tisch durch eine Mutex schützen. Der erste Philosoph, der eine Gabel aufnimmt, bekommt den ganzen Tisch, dann kann er auch eine zweite Gabel aufnehmen.

Einfache Lösung (nicht optimal in der Performance):

```
#define N         5                  // fünf Philosophen

semaphore table(4);                  // Nur 4 Phil. am Tisch erlaubt!
semaphore fork[N];                   // Gabeln

void init()                          // initialisiere alle Gabeln mit 1
{
      int i;
      for(i=0; i<N; i++ )
            fork[i] = 1;
}

void philosopher(int i)              // i: Nummer des Philosophen, 0..N-1
{
      while(1) {                     // Endlosschleife
            think();                 // Denken
            take_forks(i);           // Nimm zwei Gabeln
            eat();                   // Essen
            put_forks(i);            // Gabeln wieder hinlegen
      }
}

void take_forks(int i)               // i: Nummer des Philosophen, 0..N-1
{
      down( &table );                // betrete Tisch
      down( &fork(i) );              // linke Gabel nehmen
      down( &fork( (i+N-1)%N );          // rechte Gabel nehmen
}

void put_forks(int i)                // i: Nummer des Philosophen, 0..N-1
{
      up( &fork(i) );                // linke Gabel freigeben
      up( &fork( (i+N-1)%N );        // rechte Gabel freigeben
      up( &table );                  // verlasse Tisch
}
```

Anmerkung:

- Für Windows kann die ::WaitForMultipleObjects-Funktion zum Warten verwendet werden. Dann können die 3 down auf einmal ausgeführt werden.
- Anstelle vom Semaphore-Namen "Margull_Table" kann auch ein GUID verwendet werden, z.B. "{11EE3539-E892-4409-BF6B-C5778F3472EB}_Table".

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
#define N         5                  // fünf Philosophen

HANDLE hSemaTable;                   // Tisch-Semaphore
HANDLE hSemaFork[N];                 // Gabeln-Semaphore

void init()                          // initialisiere alle Gabeln mit 1
{
      hSemaTable = ::CreateSemaphore( NULL, 5, 99, "Table");
      hSemaFork[0] = ::CreateSemaphore( NULL, 1, 99, "Fork0");
      hSemaFork[1] = ::CreateSemaphore( NULL, 1, 99, "Fork1");
      hSemaFork[2] = ::CreateSemaphore( NULL, 1, 99, "Fork2");
      hSemaFork[3] = ::CreateSemaphore( NULL, 1, 99, "Fork3");
      hSemaFork[4] = ::CreateSemaphore( NULL, 1, 99, "Fork4");
}

void philosopher(int i)              // i: Nummer des Philosophen, 0..N-1
{
      while(1) {                     // Endlosschleife
            think();                 // Denken
            take_forks(i);           // Nimm zwei Gabeln
            eat();                   // Essen
            put_forks(i);            // Gabeln wieder hinlegen
      }
}

void take_forks(int i)               // i: Nummer des Philosophen, 0..N-1
{
      HANDLE handles[3];
      handles[0] = hSemaTable;
      handles[1] = hSemaFork[i];
      handles[2] = hSemaFork[ (i+N-1)%N ];
      ::WaitForMultipleObjects( 3, handles, TRUE, INFINITE );
}

void put_forks(int i)                // i: Nummer des Philosophen, 0..N-1
{
      LONG prev;
      ::ReleaseSemaphore( hSemaFork[ (i+N-1)%N ], 1, &prev);
      ::ReleaseSemaphore( hSemaFork[ i ], 1, &prev);
      ::ReleaseSemaphore( hSemaTable, 1, &prev);
}
```

- Das komplette Beispiel kann auf der Skript-Seite heruntergeladen werden.

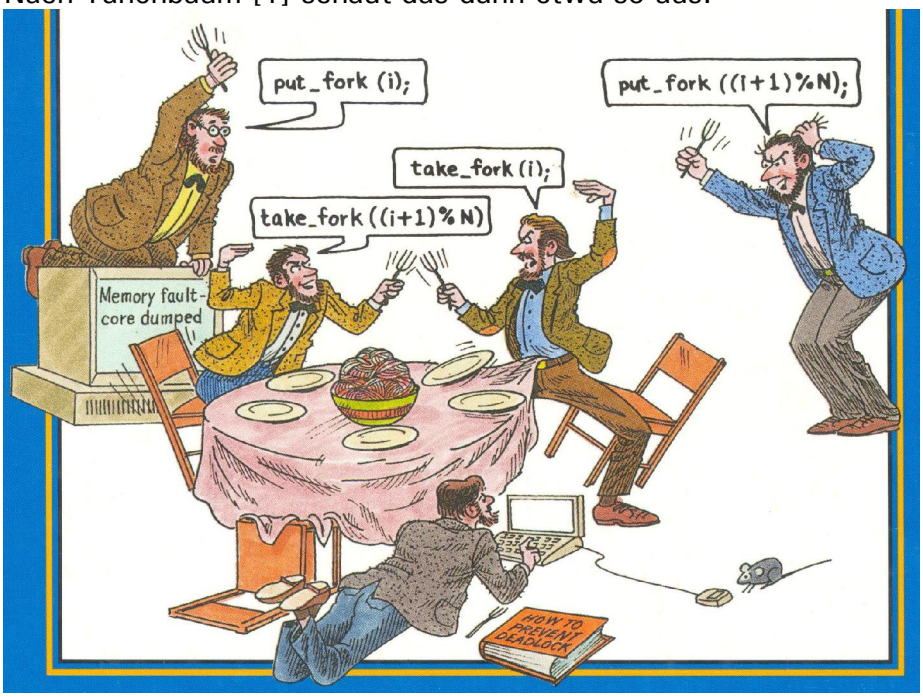
Lösung nach Tanenbaum:

```
#define N           5                  // fünf Philosophen
#define LEFT  (i+N-1)%N          // Nummer des linken Nachbarns
#define RIGHT       (i+1)%N            // Nummer des rechten Nachbarns
#define THINKING    0                  // Zustand 0: Philosoph denkt
#define HUNGRY      1                  // Zustand 1: Philosoph ist hungrig
#define EATING      2                  // Zustand 2: Philosoph isst

int state[N];                          // Feld mit Philosophen-Zuständen
semaphore mutex(1);              // Mutex: Schutz des Zustandsfeldes
semaphore s[N];                        // Warte-Zustände des Philosophen

void philosopher(int i)                // i: Nummer des Philosophen, 0..N-1
{
      while(1) {                       // Endlosschleife
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
                think();                    // Denken
                take_forks(i);              // Nimm zwei Gabeln
                eat();              // Essen
                put_forks(i);               // Gabeln wieder hinlegen
        }
}

void take_forks(int i)                  // i: Nummer des Philosophen, 0..N-1
{
        down(&mutex);                   // betrete kritischen Abschnitt
        state[i] = HUNGRY;          // Phil. i ist hungrig
        test(i);                        // teste, ob beide Gabeln frei sind
        up(&mutex);                     // verlasse kritischen Abschnitt
        down( &s[i] );                  // blockiere falls keine zwei Gabeln
                                        // verfügbar sind
}

void put_forks(int i)                   // i: Nummer des Philosophen, 0..N-1
{

        down(&mutex);                   // betrete kritischen Abschnitt
        state[i] = THINKING;            // Phil. i beendet sein Essen
        test(LEFT);                     // Kann linker Nachbar jetzt essen?    test
(RIGHT);                    // Kann rechter Nachbar jetzt essen?
        up(&mutex);                     // verlasse kritischen Abschnitt
}

void test(i)                        // i: Nummer des Philosophen, 0..N-1
{
        if( state[i]==HUNGRY                    // Ist Phil. i hungrig?
                && state[LEFT]!=EATING          // und linke Gabel frei?
                && state[RIGHT]!=EATING )  // und rechte Gabel frei?
        {
                state[i] = EATING;  // yumm yumm essen!
                up( &s[i]);                 // Blockierung von Phil.i aufheben
        }
}
```

Nach Tanenbaum [1] schaut das dann etwa so aus:



Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

6.5 Prioritäteninversion

Prioritäteninversion tritt in Prioritäts-gesteuerten Systemen auf:
1. Ein niedrig-priorer Prozess P1 ist aktiv und allokiert eine Resource A
2. Ein mittel-priorer Prozess P2 wird aktiv und unterbricht P1
3. Ein hoch-priorer Prozess P3 wird aktiv, P2 wird unterbrochen
4. P3 benötigt die Resource A und geht in den Wartezustand über
5. Als nächstes rechnet P2 zum Ende
6. Dann erst kommt P1 dran: P1 rechnet weiter und gibt A wieder frei
7. Nun erst kommt P3 (höchst-prior) wieder dran und unterbricht P1

Folge: P2 hat vor P3 fertiggerechnet, obwohl die Priorität von P2 niedriger ist als die von P3!

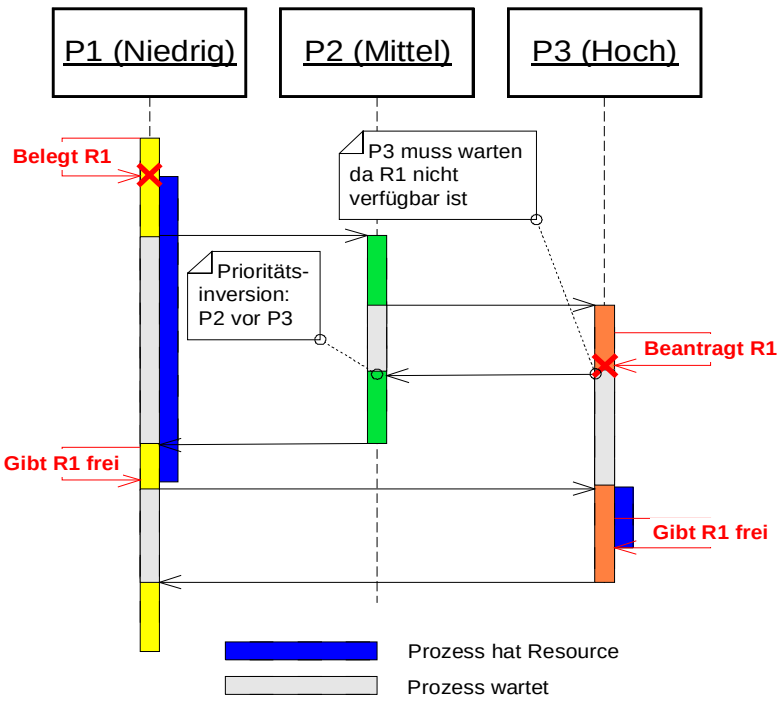


Abbildung 54: Prioritätsinversion (P2 beendet vor P3)

Mögliche Alternativen:

- ***Prioritätsvererbung (priority inheritance):*** warten ein oder mehrere Prozesse auf eine Resource, die ein anderer Prozess mit niedriger Priorität hält, so erhält dieser Prozess kurzzeitig die Priorität des höchsten wartenden Prozesses
- ***Priority Ceiling:*** Für jede Resource wird im Vorfeld eine Ceiling-Priorität festgelegt, die höher ist als die jedes Prozesses der diese Resource zu irgendeinem Zeitpunkt nutzen will. Ein Prozess, der die Resource aquiriert, erhält automatisch die Ceiling-Priorität.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax -9
www.margull.de

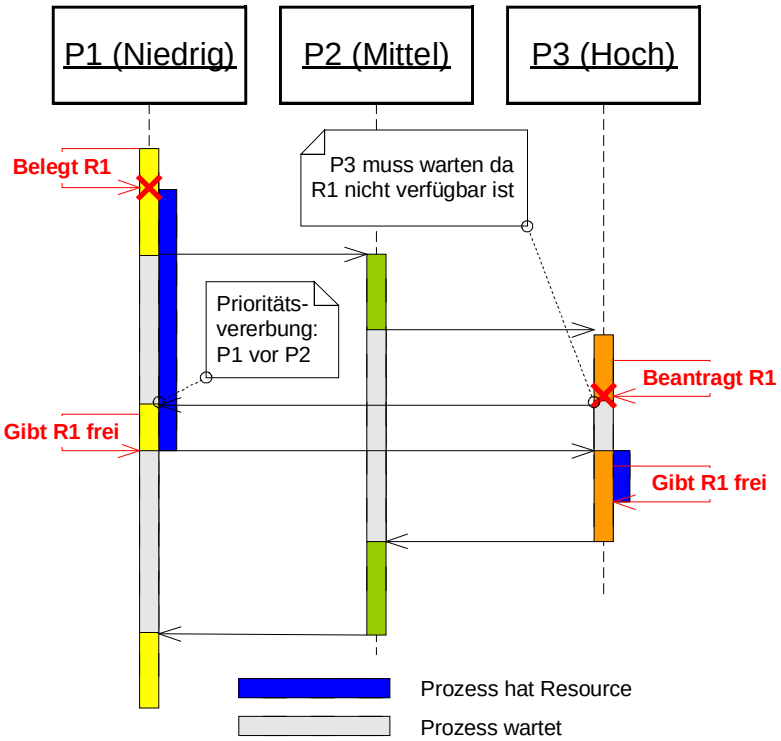


Abbildung 55: Prioritäten-Vererbung (P1 erhält dieselbe Priorität wie P3)

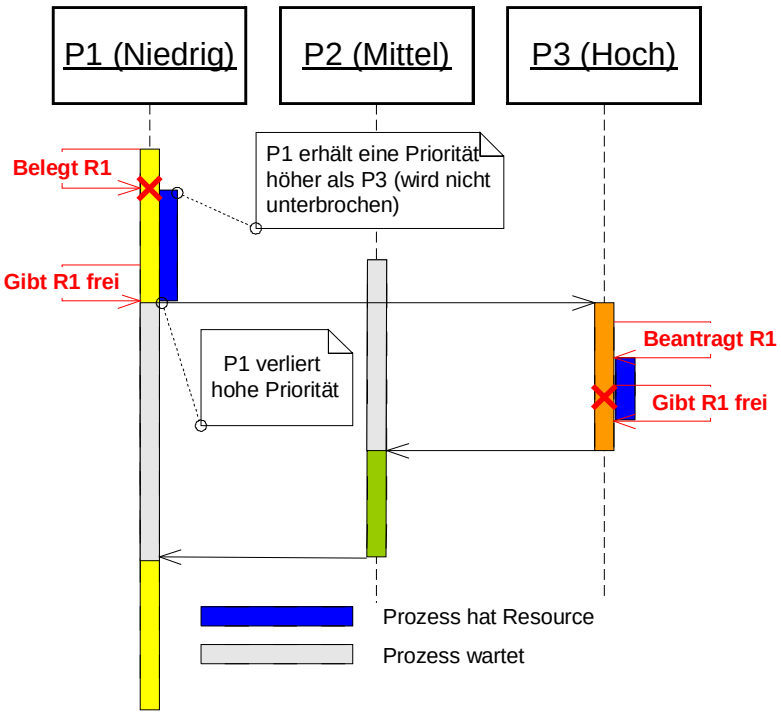


Abbildung 57: Priority-Ceiling(P1 erhält kurzzeitig eine hohe Priorität)

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

6.6 Deadlocks

6.6.1 Wiederholung: Begrifflichkeiten

Blockierung:
Ein Prozess P1 belegt ein Betriebsmittel, ein zweiter Prozess P2 benötigt dasselbe Betriebsmittel und wird daher blockiert, bis es P1 freigegeben hat.

Verhungern (Starvation):
Ein Prozess erhält trotz Rechenbereitschaft keine CPU-Zeit zugeteilt, z.B. weil ihm immer wieder Prozesse mit höherer Priorität vorgezogen werden.

Verklemmung (Deadlocks):
Zwei oder mehr Prozesse halten jeder für sich ein oder mehrere Betriebsmittel belegt und versuchen ein weiteres zu belegen, das aber von einem anderen belegt ist. Es liegt ein Zyklus von Abhängigkeiten vor. Kein Prozess gibt seine Betriebsmittel frei und alle Prozesse warten daher ewig.

6.6.2 Bedingungen für ein Deadlock

Ein Deadlock kann nur eintreten, wenn folgende vier Bedingungen eintreffen:
1. Gegenseitiger Ausschluss (**Mutual Exclusion**) für die benötigten Betriebsmittel: nur ein Prozess kann die Resource verwenden. Andere Prozesse müssen warten.
2. **Hold and Wait:** Prozesse belegen Betriebsmittel und fordern weitere an.
3. **No Preemption:** Kein Entzug eines Betriebsmittels ist möglich.
4. **Circular Wait:** Zwei oder mehrere Prozesse warten in einer verketteten Liste (in einer Schleife, circular waiting) auf weitere Betriebsmittel.

Betriebsmittelbelegungsgraphen:
Eine in der Praxis häufig eingesetzte Technik ist das Erkennen und Beseitigen von Deadlocks zur Laufzeit, wobei man hierzu sog. Betriebsmittelbelegungsgraphen einsetzt. Dies sind Graphen, die als Knoten Ressourcen und Prozesse/Threads enthalten und Kanten, welche die Belegung der Ressourcen durch Prozesse/Threads aufzeigen. Als Maßnahmen zur Beseitigung eines Deadlocks sind das Abbrechen eines Prozesses/Threads oder das Entziehen eines Betriebsmittels möglich.

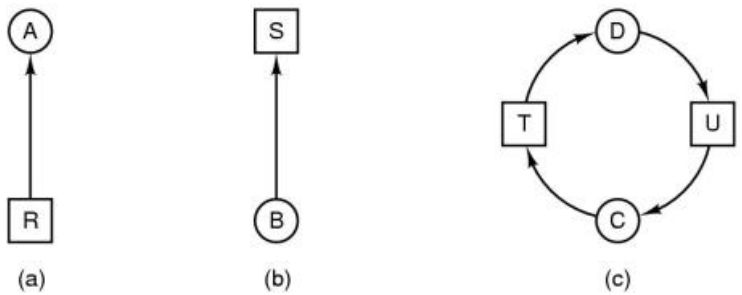


Abbildung 58: Betriebsmittelbelegungsgraphen
(A, B, C, D: Prozesse, R, S, T, U: Resourcen)

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

In der Graphik:
(a) Prozess A hält Resource R belegt
(b) Prozess B wartet auf Resource S
(c) Deadlock:
- Prozess D hält T und möchte U
- Prozess C hält U und möchte T

(Ein weiteres Beispiel ist am Ende dieses Abschnittes zu sehen)

Strategien zur Deadlock-Behandlung:
- **Ignorieren des Problems:** Deadlocks passieren. Thats life.
- **Zulassen:** Entdeckung (detection) und Auflösung (Recovery) der Verklemmung.
- **Vermeidung (prevention):** Durch Eliminierung einer (oder mehrerer) der oben genannten Ursachen.
- **Verhinderung (avoidance):** Erfüllbare Resource-Anforderungen werden nicht gewährt, wenn Verklemmungsgefahr besteht.

6.6.3 Ignorieren des Problems (Vogel-Strauss-Strategie)

Einfachste Strategie, z.B. von Unix und Windows verwendet.
Jdeoch nicht unbedingt geeignet für:
- Echtzeitsysteme, z.B. Automobilsteuerungen[4]
- Kernkraftwerkssteuerungen[5]

6.6.4 Entdeckung und Behandlung

Wenn Verklemmung aufgetreten ist, wird diese beseitigt:
- Algorithmus zum Erkennen von Deadlocks notwendig
- Was tun, wenn man ein Deadlock erkannt hat?
    - Einen Prozess aus der Schleife entfernen (killen); bei Batchsystemen kann dieser später nochmals gestartet werden (Operator muss alles Papier aus dem Drucker nehmen und wegschmeissen)
    - Weitere Möglichkeiten, z.B. Prozess anhalten und einen früheren Zustand wiederherstellen

6.6.5 Vermeidung von Deadlocks

Vermeidung = Eliminierung einer der notwendigen Bedingungen für ein Deadlock

Eliminierung von: Mutual Exclusion
Im Allgemeinen nicht vermeidbar. In der Betriebssystementwicklung wird jedoch diese Bedingung immer mehr aufweicht:
- Speicher: war früher ausschliesslich einem Prozess zugeordnet; heute durch Speichermanagement (und zusätzliche Hardware) mehrere Prozesse gleichzeitig im Hauptspeicher

[4]Wird trotzdem oft verwendet. Ein Watchdog (Hardware-Überwachung) überwacht das Steuergerät, und wenn es nicht mehr reagiert, wird ein Reset ausgelöst.
[5]Leider weiss ich nicht, wie hier Deadlocks vermieden werden.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Ebenso CPU: früher (z.B. MS-DOS) nur ein Programm zu einem Zeitpunkt möglich; um ein neues Programm zu starten, musste zunächst das Alte beendet werden. Heute quasi-parallele Verarbeitung möglich
- Drucker: früher hat nur 1 Programm zu einem Zeitpunkt drucken können. Heute: Drucker-Spooler sammeln Druckaufträge ein und drucken sie nacheinander.
- u.s.w.

Eliminierung von: Hold and Wait
- Gesamtanforderung aller Betriebsmittel bei Programmstart -> extrem schlechte Auslastung der Resourcen
- Bei neuer Anforderung werden zunächst alle vom Prozess angeforderten Resourcen freigegeben, um dann erneut in einem Schritt angefordert zu werden -> Möglichkeit der Starvation
- Anforderung aller benötigten Resourcen auf einmal (z.B. *WaitForMultipleObject()*)

Eliminierung von: No Preemption
- Wartende Prozesse werden zwangsenteignet
  Problem: Resourcen können sich in einem inkonsistenten Zustand befinden

Eliminierung von: Circular Wait
- Vorgeschriebene Reihenfolge bei der Anforderung von Resourcen:
  - alle Resourcen werden linear angeordnet, z.B. Drucker = 1, Band = 2, CDROM = 3
  - Wenn ein Prozess mehrere Resourcen benötigt, so muss er sie in der vorgegebenen Reihenfolge anfordern
  - Beispiel: Ein Prozess braucht z.B. Band und CDROM: erst Band, dann CDROM anfordern
  - Beispiel: Ein Prozess hat das CDROM angefordert und will nun das Band haben -> nicht möglich; zuerst muss CDROM freigegeben werden, dann Band und dann CDROM angefordert werden
- Bei Spezialsystemen, z.B. Echtzeitsystemen, sehr effektive Massnahme
- In General-Purpose Systemen mit vielen Resourcen, vielen Programmen, vielen Prozessen, etc. jedoch nicht praktikabel.

6.6.6 Verhinderung von Deadlocks

Deadlocks können u.U. verhindert werden, wenn augenblicklich erfüllbare Anforderungen zurückgewiesen werden.
- **Sicherer Zustand:**
  a) kein Deadlock
  b) Selbst wenn alle Prozesse sofort alle benötigten Resourcen anfordern, gibt es eine Scheduling-Reihenfolge, die alle Anforderungen ohne Deadlock erfüllen kann.
- **Unsicherer Zustand:** das System kann in einen Deadlock-Zustand geraten (oder ist schon im Deadlock), d.h. a) oder b) im obigen Punkt sind nicht erfüllt.

Beispiel:
Zwei Prozesse A und B fordern die Resource "Printer" und "Plotter" an. Seien I1 – I4 Zustände von A, die in der Reihenfolge I1, I2, I3, I4 durchlaufen werden, und I5 – I8 Zustände von B (wieder in der Reihenfolge I5, I6, ... durchlaufen).

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Bei A_1 fordert A den Printer an, bei A_2 den Plotter, bei A_3 wird der Printer wieder freigegeben, bei A_4 der Plotter
- Bei B_5 fordert B den Plotter an, bei B_6 den Printer, bei B_7 wird der Plotter wieder freigegeben, bei B_8 der Printer
- p, q, r, s, t, u sind Zustände die das gesamte System (A und B) durchlaufen
- Innerhalb der schraffierten Gebiete haben sowohl A als auch B dieselbe Resource bekommen; dies verletzt die Mutual-Exclusion Bedingung. Diese Gebiete können also nicht betreten werden.
- Bei r fordert A die Resource "Printer": die Anforderung wird gewährleistet. Bei s rechnet B weiter
- Bei t fordert B nun die Resource Plotter an: Falls diese Anforderung gewährleistet wird, ist ein Deadlock unvermeidlich. Deshalb wird die Anforderung nicht gewährt, B muss warten.
- Das unsichere Gebiet ist gelb gezeichnet. Innerhalb des Gebietes gilt: Wenn A und B ihre Maximalforderungen stellen, so tritt ein Deadlock auf.

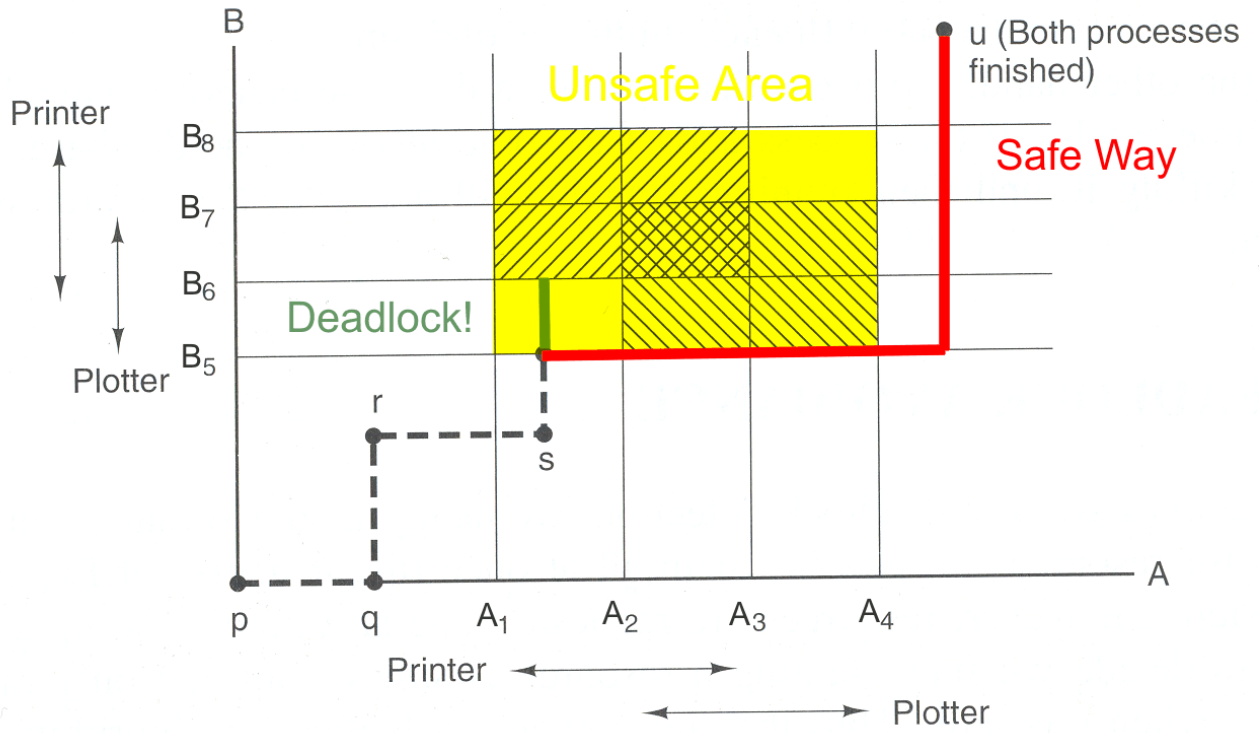


Abbildung 59: Deadlock-Trajektorie (nach [1])

- Generell gilt: ein unsicheres Gebiet muss nicht immer zwangsläufig zum Deadlock führen. Betritt ein System das unsichere Gebiet, so kann jedoch unter Umständen ein Deadlock nicht mehr verhindert werden. Verzichtet z.B. B in manchen Situationen auf den Printer, so führt der grüne Weg nicht zum Deadlock.

Algorithmus:

- Bedingung: Prozesse müssen im voraus angeben, welche Resourcen (und wieviel davon) sie benötigten.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Bei jeder Anforderung prüft das System, ob diese gefahrlos ausgeführt werden kann, d.h. ob das System in einem sicherem Zustand bleibt.
- Beispiel: Bankers Algorithmus (Dijkstra, 1965)

6.6.7 Beispiel (nach [1])

Im folgenden Beispiel fordern die 3 Prozesse A, B und C jeweils zwei Resourcen an:
- Prozess A fordert R und S an
- Prozess B fordert S und T an
- Prozess C fordert T und R an

-> Es kommt zu einem Deadlock!

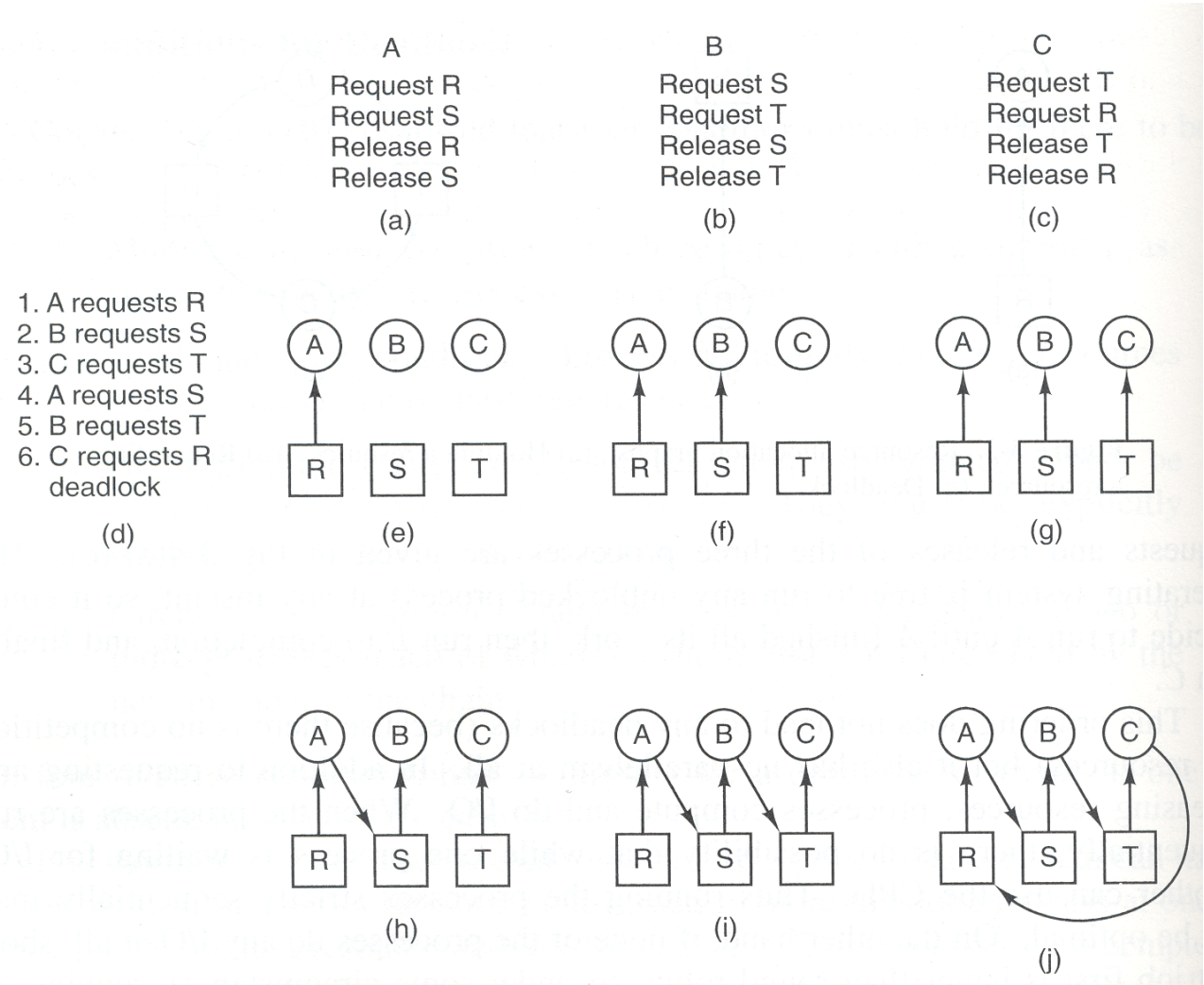


Abbildung 60: Deadlock!

Fragen:
- Ab welchem Bild befindet sich das System in einem unsicheren Zustand?
- Welche Anforderung muss es ablehnen, um sicher ein Deadlock zu vermeiden?
- Würde eine lineare Ordnung bei den Resourcenanforderungen helfen?

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax -9
www.margull.de

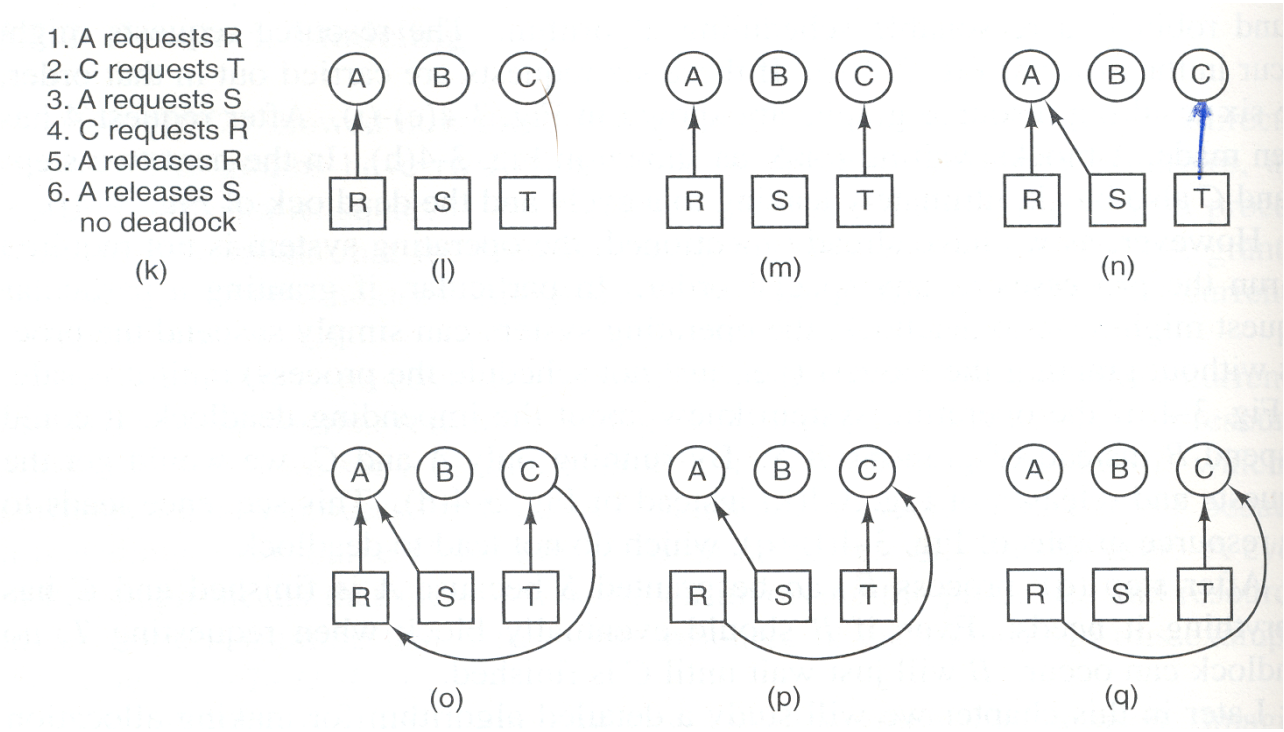


Abbildung 61: Kein Deadlock!