4. Koordination nebenläufiger Prozesse

- Überblick
  - 4.1 Elementare Koordinationsoperationen
  - 4.2 Signalisierung
  - 4.3 Kritische Abschnitte
  - 4.4 Sperren mit Wartezustand: Semaphore
  - 4.5 Monitore

4.1 Elementare Koordinationsoperationen

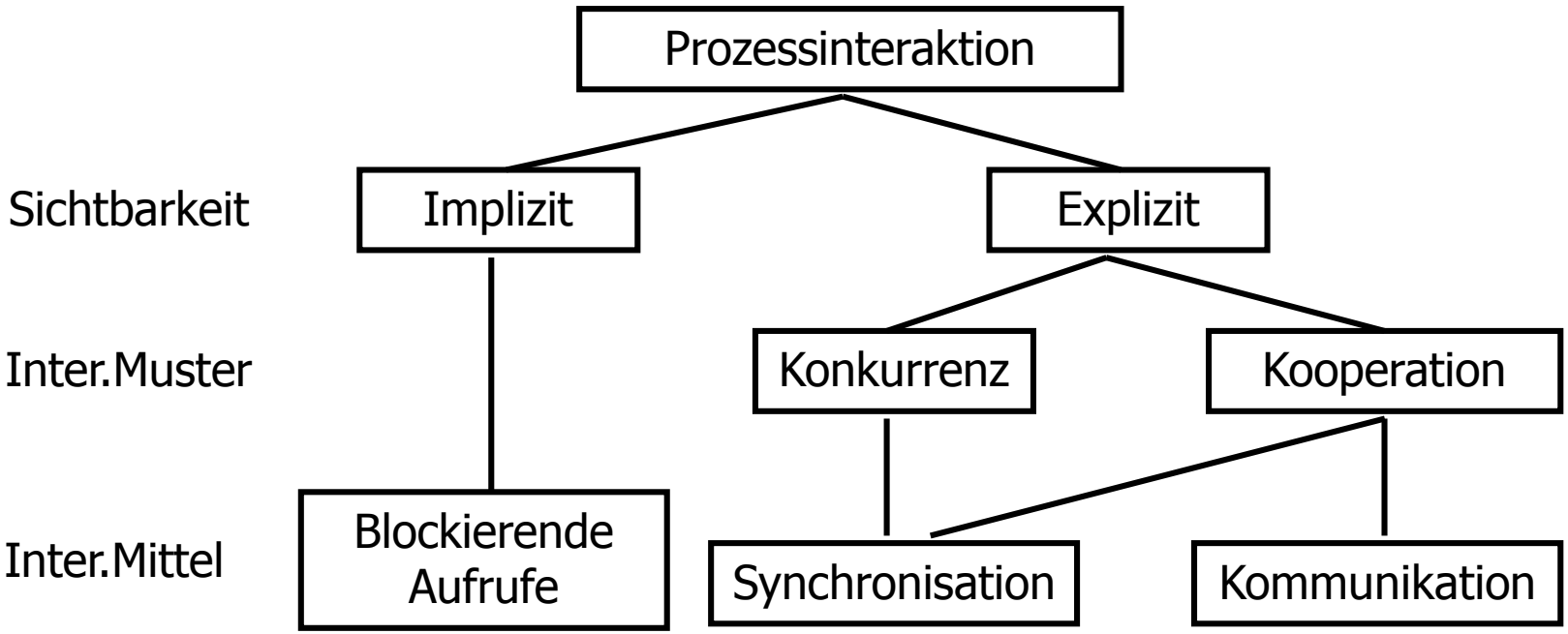
- Prozesse, die isoliert und unabhängig voneinander ablaufen, müssen nicht koordiniert werden
- Dies ist allerdings eher selten, da
  - Ein Teil der Betriebsmittel nur exklusiv belegt werden kann
  - Mehrere Prozesse für die Lösung einer gemeinsamen Aufgabe eingesetzt werden
  - Prozesse tauschen Daten aus unterschiedlichen Quellen aus

⇒ Unterstützung der Prozessinteraktion ist eine grundlegende Aufgabe der Systemsoftware

- Grundsätzlich gibt es zwei Formen der Interaktion
  - Konkurrenz
  - Kooperation

Explizite / Implizite Prozessinteraktion

- Implizite Interaktion, wenn ein Prozess eine Systemfunktion aufruft und diese mit anderen Prozessen interagiert
  - ⇒ Der aufrufende Prozess bekommt davon nichts mit
  - ⇒ Wird ggf. für die Dauer der Interaktion geblockt und wieder gestartet, wenn die Ergebnisse vorliegen
- Klassifikation

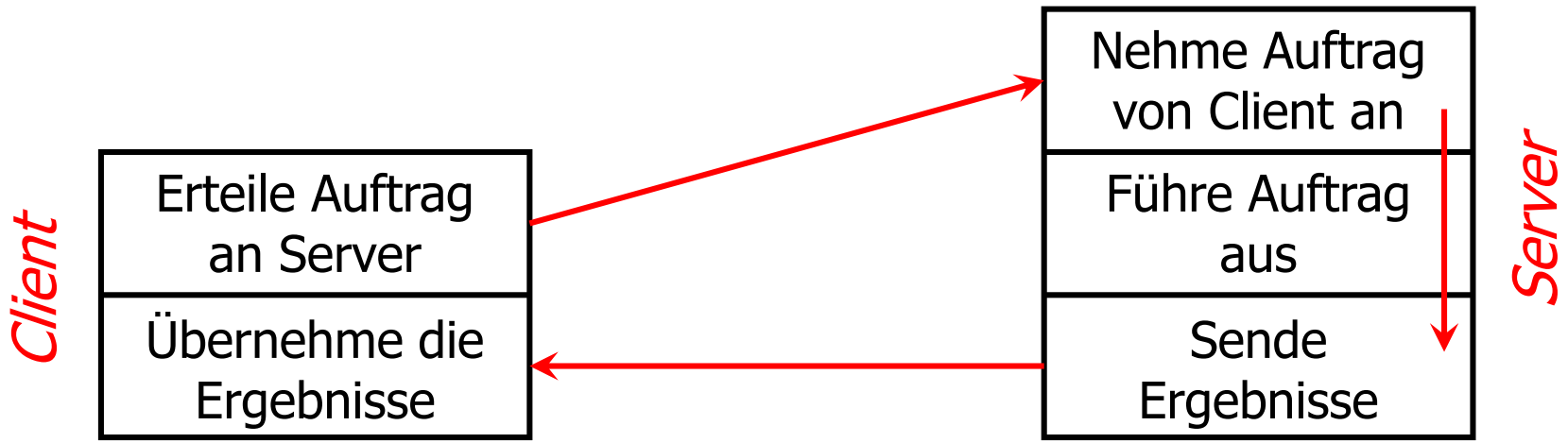


Prozessinteraktion: Konkurrenz

- Zwei oder mehrere Prozesse bewerben sich gleichzeitig um ein exklusiv benutzbares Betriebsmittel, z.B. Drucker
  - ⇒Synchronisationsmechanismen notwendig
  - ⇒Durch geeignete Koordinierung muss eine Serialisierung der Zugriffsversuche erreicht werden
  - ⇒Bei *n* konkurrierenden Prozessen werden *n* - 1 Prozesse z.B. zeitlich verzögert
- Die zeitliche Abstimmung konkurrierender Prozesse wird als Prozesssynchronisation bezeichnet

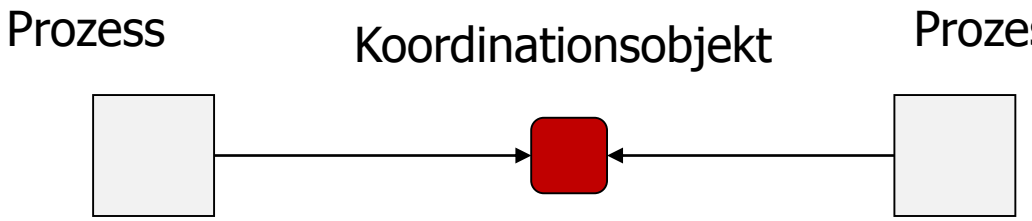
Prozessinteraktion: Kooperation

- Prozesse tauschen gezielt Informationen untereinander aus, z.B. Erzeuger / Verbraucher-Situation
  - Erzeuger füllt einen Pufferplatz mit Daten
  - Verbraucher entnimmt die Daten aus dem Puffer
- Kooperierende Prozesse müssen
  - Von der Existenz aller anderen beteiligten Prozesse wissen
  - Ausreichende Informationen über diese besitzen, z.B. Art und Funktionalität der Schnittstellen
- Klassisches Beispiel: Client/Server

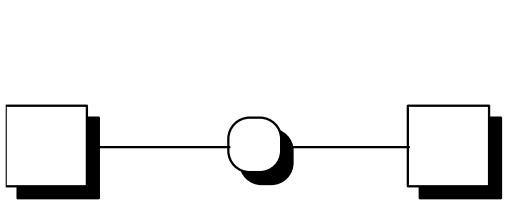


Koordinationssobjekte

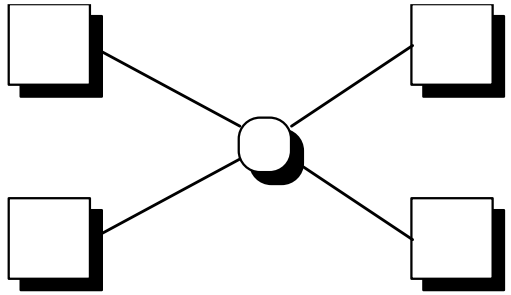
- Koordination manifestiert sich in eigenständigen Objekten



- An einer Koordination können mehr als zwei Prozesse beteiligt sein



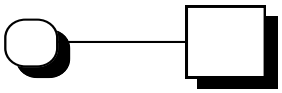
1:1- Koordination



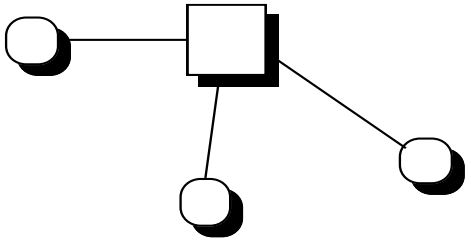
m:n - Koordination

Beziehungen

- Ein Prozess kann an mehreren Koordinationen beteiligt sein



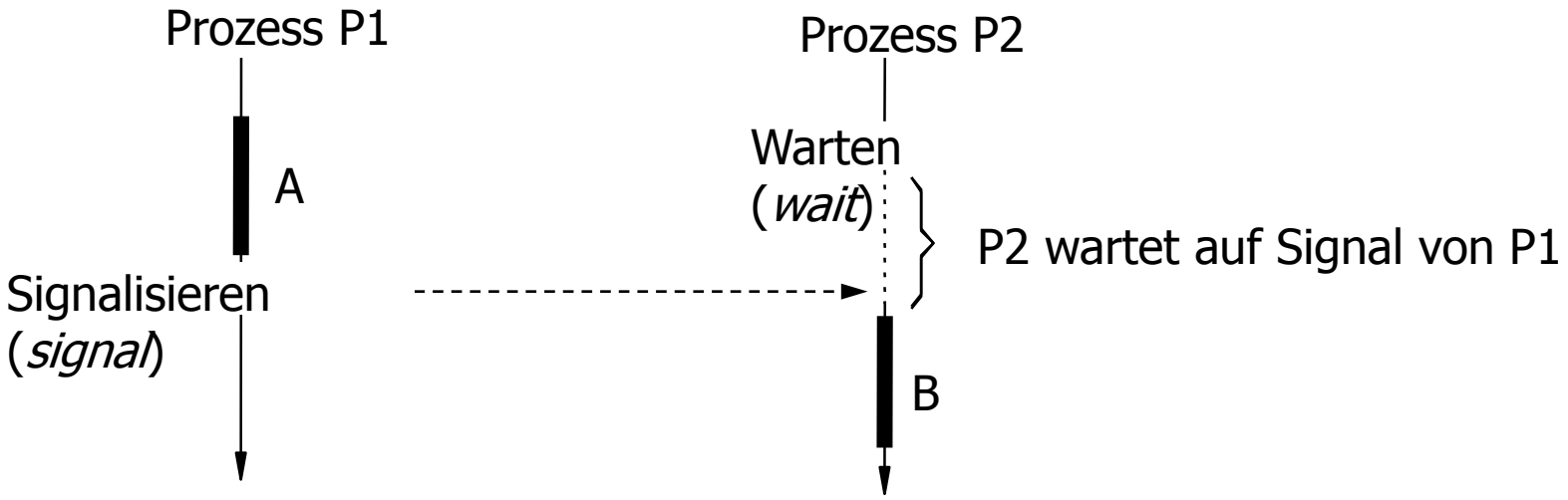
1 Koordinationsobjekt



mehrere Koordinationsobjekte

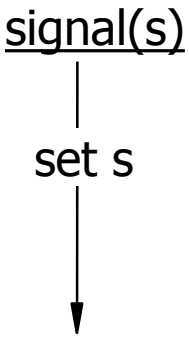
4.2 Signalisierung

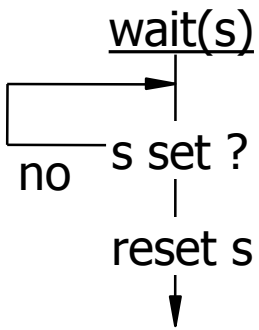
- Elementare Aufgabe: Exklusives Sperren/Freigeben einer Variable (Sperrflag) durch konkurrierende Prozesse
  - Voraussetzung: Prozesse haben Zugriff auf gemeinsamen Speicher
- Einfachste Form: Reihenfolgebeziehung (Signalisierung)
  - Prozess P2 wird fortgesetzt, erst nachdem Prozess P1 einen bestimmten Abschnitt bearbeitet hat
- Operationen signal(s) und wait(s), s Sperrflag



Grundform der Signalisierung (akt. Warten)

- Einfachste Realisierung: Busy waiting (aktives Warten)
  - Wiederholende Statusabfrage realisiert als Schleife

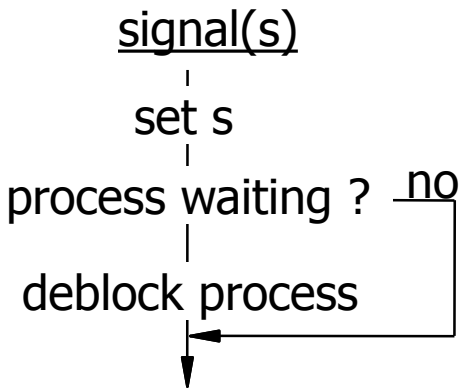


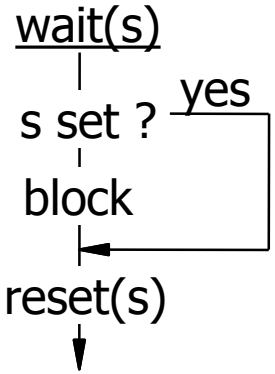



```
// signalization with busy waiting
   boolean  s = false;                  // initialization
   void signal(boolean s){
      s = true;                         // set signal
   }
   void wait(boolean s){
      while (s == false) { ;  }         // wait for signal
      s = false;
   }
```

Grundform der Signalisierung: Wartezustand

- Aktives Warten verschwendet Prozessorkapazität und blockiert die CPU währenddessen für sinnvolle Aufgaben
- Lösung
  - Wartezeit zu lange, so wird die CPU freigegeben: Signalisieren mit Wartezustand, nur ein Prozess kann warten
  - Aus diesem Wartezustand (blockiert) muss der Prozess explizit durch das Signal deblockiert werden



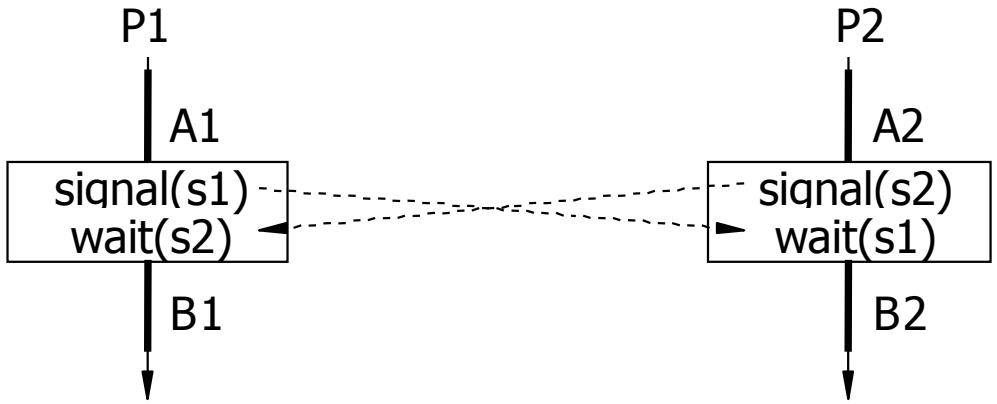


Beispiel für Signalisierungsobjekt

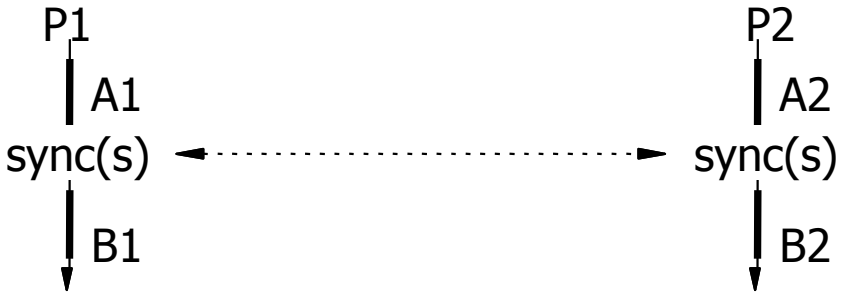
```
// signalization with wait state;

struct Signal_object {
      boolean s = false;                  // initialization
      process *wp = NULL;
}

void signal(Signal_object *so) {
      so->s = true;
      if (so->wp != NULL)                  // a process is waiting
            deblock(so->wp);               // deblock it
}

void wait(Signal_object *so) {
      if (so->s == false)
         block(so->wp);                    // wait for signal
      so->s = false;
}
```

Wechselseitige Synchronisierung

- Symmetrischer Einsatz der Operationen bewirkt, dass sowohl A1 als auch A2 ausgeführt sind, bevor B1 oder B2 ausgeführt werden



- Prozesse P1 und P2 **synchronisieren** sich an dieser Stelle
  ⇒ Zusammenfassung zu Operation **sync (Rendezvous)**



Beispielimplementierung

```
struct Sync_object {          // rendezvous synchronization
      boolean s = false;      // initialization
      process *wp = NULL;
}

void sync(Sync_object *so) {
   if (so->s == false) {      // I am first and
      so->s = true;           // indicate my arrival and
      block(so->wp);          // wait for my partner
   }
   else {                     // I am second and
      deblock(so->wp);        // deblock my waiting partner
      so->s = false           // and reset the signal for reuse
   }
}
// end of rendezvous synchronization.
```

4.3 Kritische Abschnitte

- Definition Kritischer Abschnitt (Kritischer Bereich)
  - Operationsfolgen, bei denen eine nebenläufige oder verzahnte Ausführung zu Fehlern führen kann
- Beispiele
  - Zugriff auf exklusiv benutzbare Betriebsmittel
  - Veränderungen gemeinsam benutzter Variablen
- Definition mit Sperrvariablen

```
...
Sperren (Sperrvariable)
                          } Kritischer Bereich / Abschnitt
Freigeben (Sperrvariable)
...
```

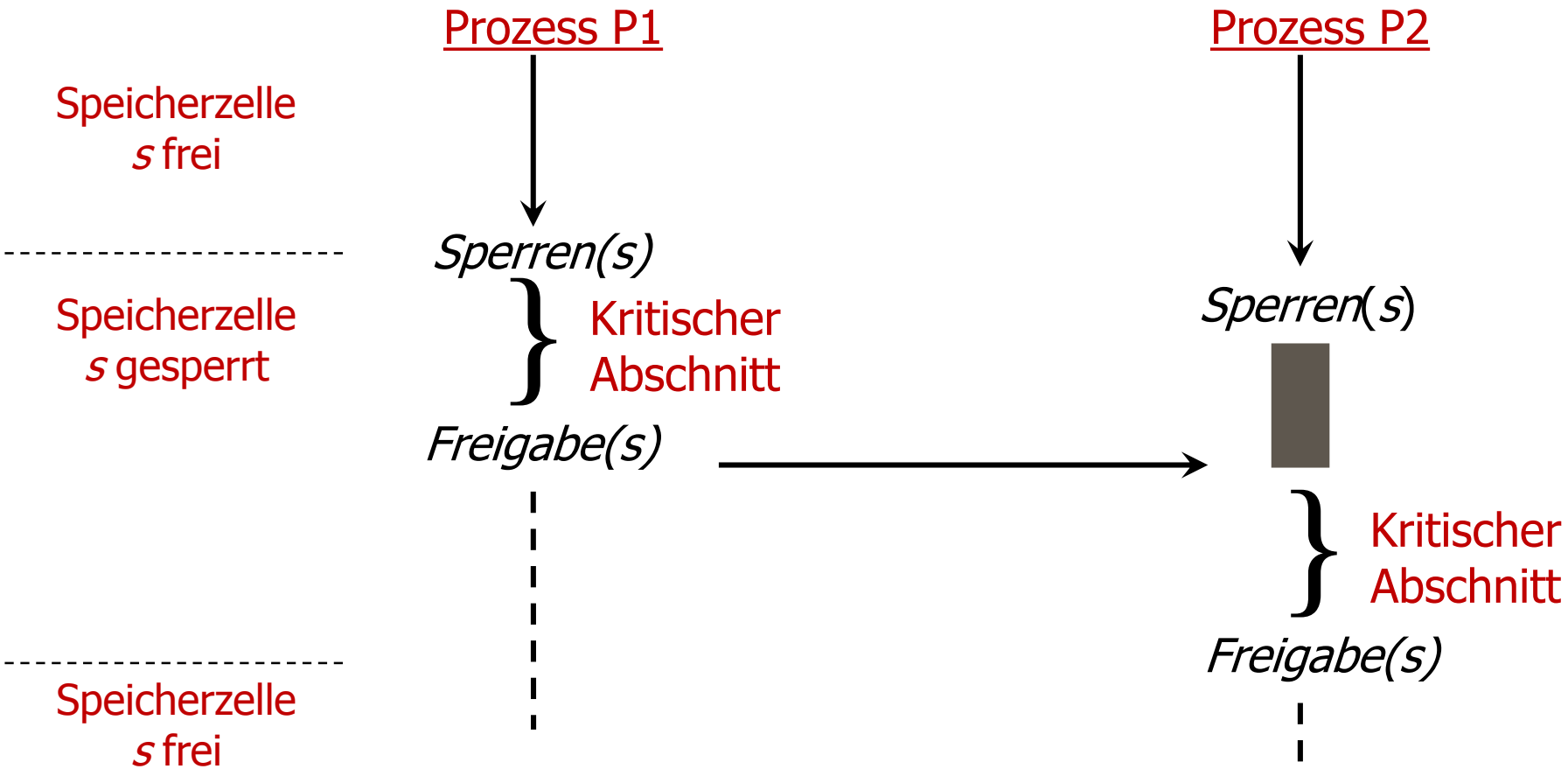
- Durch ein Sperrflag soll sichergestellt werden, dass sich ein einziger Prozess im kritischen Bereich befindet

Prozesssynchronisation mittels kritischer Abschnitte

- Ein kritischer Abschnitt darf von nur einem Prozess betreten werden
  - ⇒Prozesse müssen sich gegenseitig ausschließen (*mutual exclusion*)
- Weitere Anforderungen an Behandlung von kritischen Abschnitten
  - Keine Annahmen über Prozessorgeschwindigkeit
  - Keine Annahmen über Anzahl und Reihenfolge von Prozessen
  - Keine Verzögerung von Prozessen in unkritischen Bereichen
  - Keine Verklemmung, d.h. Prozesse dürfen sich nicht gegenseitig blockieren
  - Ein Prozess muss nach endlicher Zeit den kritischen Bereich betreten können
  - Endliche Aufenthaltszeit im kritischen Bereich

Kritische Abschnitte mit Sperrflags

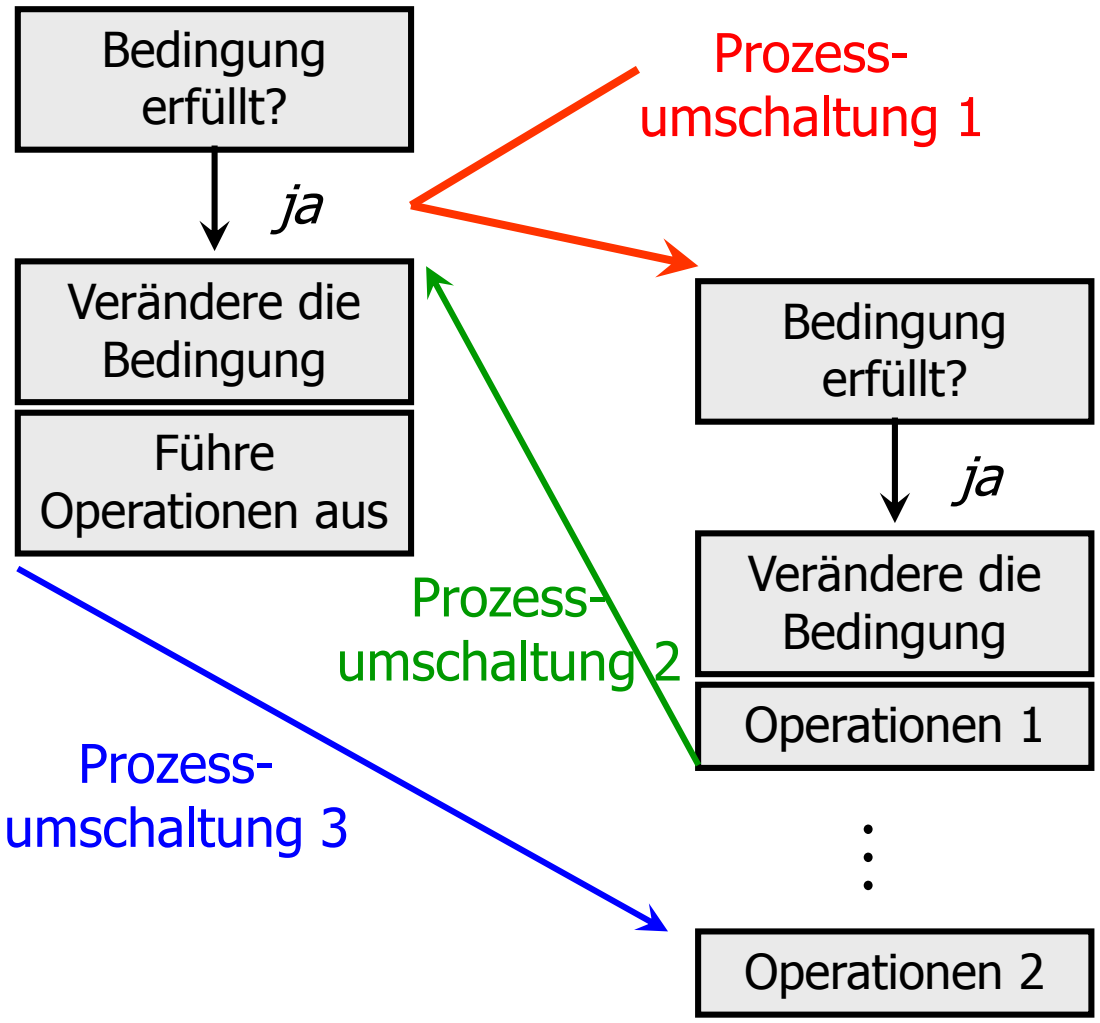
- Kritische Abschnitte können nur unter der Bedingung „keine Nebenläufigkeit“ mit Sperrflags gesichert werden



Probleme durch verzahnte Ausführung

- Bei verzahnter Ausführung kann nicht ausgeschlossen werden, dass zwischen
  - Abfrage der Bedingung UND
  - Darauf folgender Operation

  ein Umschalten stattfindet und ein anderer Prozess die Bedingung ändert



Beispiel: Zimmerbelegung im Hotel

- Es sei ein Hotel mit *anzahl* Zimmern gegeben und jedem Zimmer seien die Attribute *status* (*frei, belegt*) und Gastname zugeordnet
- Routine zur Abfrage/Belegung der Zimmer von einem Terminal

```
[1] Warte auf Signal vom Terminal
[2] if (Freizimmer>0) {  //Initial Freizimmer = anzahl;
[3]   i=SucheZimmer();
[4]   Zimmer[i].status=belegt;
[5]   Zimmer[i].gast=enterName();
[6]   Freizimmer --;
[7]   PrintMessage(Zimmer i reserviert); }
[8] else PrintMessage(Hotel belegt);
```

- Für zwei Buchungsprozesse A und B sind beispielsweise folgende Konstellationen – abhängig von der Verzahnung – möglich
  - A1…A8B1…B8
  - B1…B8A1…A8
  - A1A2A3 B1B2B3 A4A5A6A7A8 B4B5B6B7B8
- Verzahnte Ausführung hinterlässt inkonsistenten Datenbestand. Warum?

Pthreads „hello world“ Programm

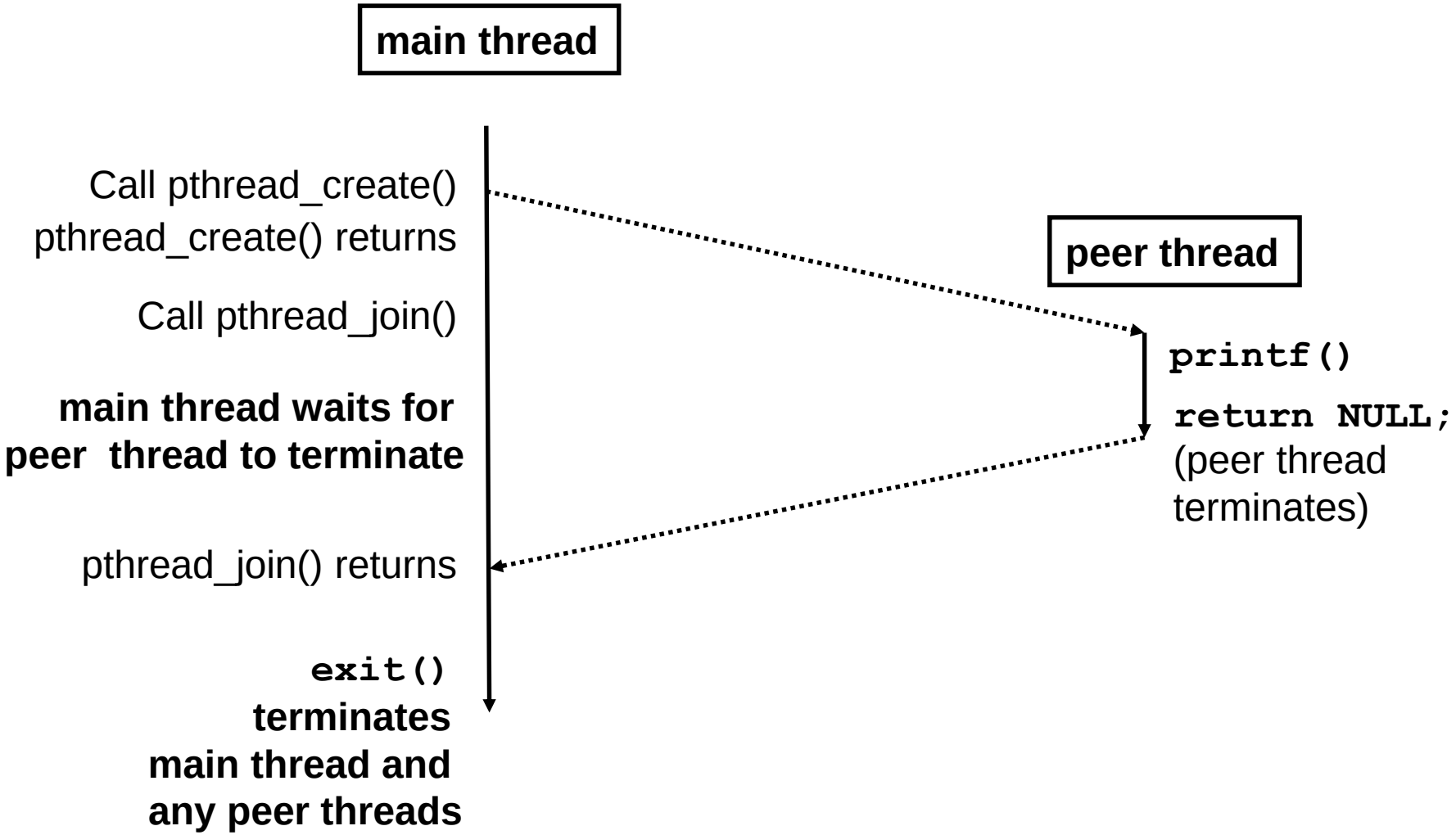
```
// hello.c - Pthreads "hello, world" program

#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void *thread(void *vargp);

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Ausführung des „threaded hello world“



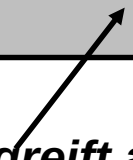
Threaded C: Gemeinsame Variablen

- Frage: Welche Variablen in einem Threaded C Programm sind gemeinsame Variablen?
  - Globale Variablen sind gemeinsam
  - Variablen auf dem Stack sind private

Beispiel: Zugriff auf Threadstack

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int svar = 0;

    printf("[%d]: %s (svar=%d)\n",
        myid, ptr[myid], ++svar);
}
```



Peer Threads greift auf Stack des Hauptthreads indirekt über die globale ptr Variable zu

Abbilden von Variablen zu Speicher

Globale Var: 1 Instanz (`ptr` [data])

Lokale automatische Var: 1 Instanz (`i.m, msgs.m`)

Lokal automatische Var: 2 Instanzen (
`myid.p0`[peer thread 0's stack],
`myid.p1`[peer thread 1's stack]
)

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int svar = 0;

    printf("[%d]: %s (svar=%d)\n",
        myid, ptr[myid], ++svar);
}
```

Lokale statische Var: 1 Instanz (`svar` [data])

Gemeinsame Variabeln Analyse

- Welche Variablen sind gemeinsam?

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
`ptr`	`yes`	`yes`	`yes`
`svar`	`no`	`yes`	`yes`
`i.m`	`yes`	`no`	`no`
`msgs.m`	`yes`	`yes`	`yes`
`myid.p0`	`no`	`yes`	`no`
`myid.p1`	`no`	`no`	`yes`

- Antwort: Eine Variable x ist gemeinsam, wenn mehrere Threads auf wenigstens eine Instanz von x referenzieren.
  - Entsprechend sind
    - **`ptr`**, **`svar`**, und **`msgs`** sind gemeinsam
    - **`i`** und **`myid`** sind NOT nicht gemeinsam

Badcnt.c: falsch synch. Programm

```
unsigned int cnt = 0; /* shared */
#define NITERS 100000000
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL,
                   count, NULL);
    pthread_create(&tid2, NULL,
                   count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n",
               cnt);
    else
        printf("OK cnt=%d\n",
               cnt);
}
```

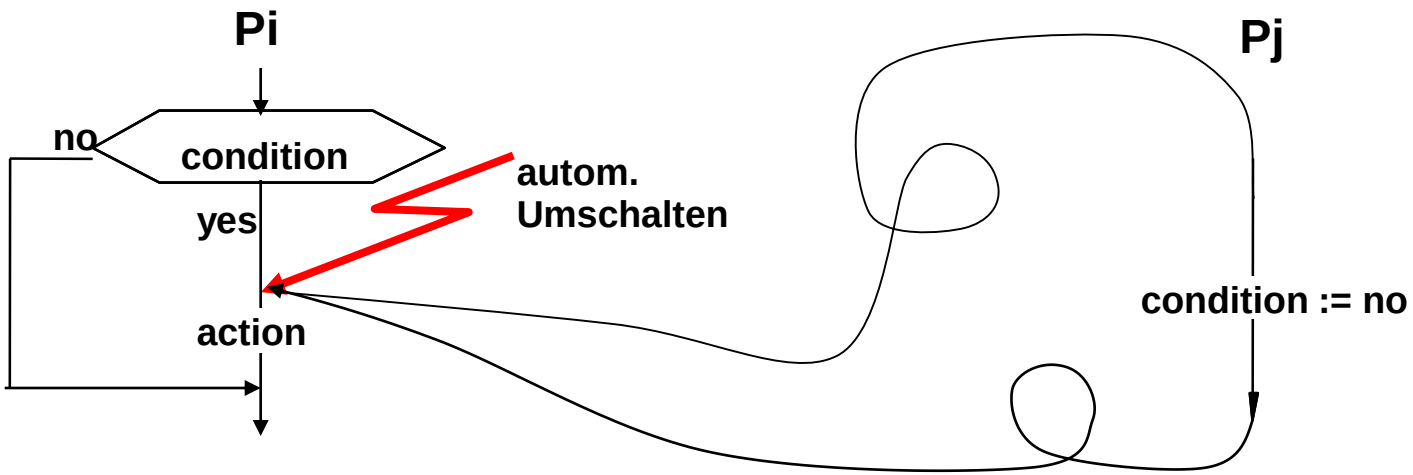
```
/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```
linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672
```

cnt sollte 200,000,000 sein. Was geht daneben?!

Diskussion

- Es ist nicht auszuschließen, dass zwischen der Abfrage der Bedingung und der Aktion ein Umschalten stattfindet und vor der Rückkehr zum unterbrochenen Prozess ein anderer Prozess die Bedingung ändert



⇒ Auswerten der Bedingung und Aktion müssen unteilbar sein, d.h. sie bilden einen kritischen Abschnitt!

⇒ Zur Sicherung eines kritischen Abschnitts haben wir Operationen eingeführt, die selbst einen (kurzen) kritischen Abschnitt darstellen!

- Wie lösen wir diese Rekursion auf ?

Hardwaregestützte Maßnahmen

- Einfachste Lösung: Maskierung der Unterbrechungen
  - Für die Dauer der Operationen wird der Unterbrechungs-mechanismus außer Kraft gesetzt, so dass keine Umschaltung stattfinden darf
  - Auf einem Einprozessorrechner kann die Sperre nur durch Hardwarefehler umgangen werden

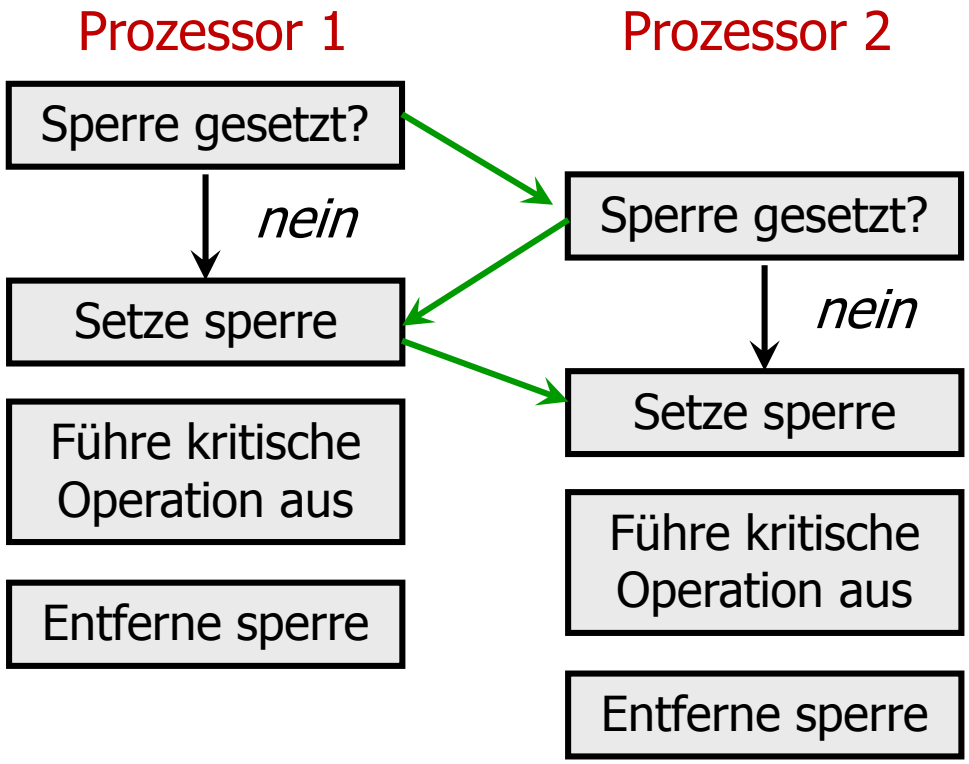
```
...
disable interrupt      /* Asynchrone Unterbrechungen sperren */

        Kritische
        Operationen

enable interrupt
...
```

```
disable interrupt

critical section

enable interrupt
```

Unterbrechungssperre bei Mehrprozessorrechnern

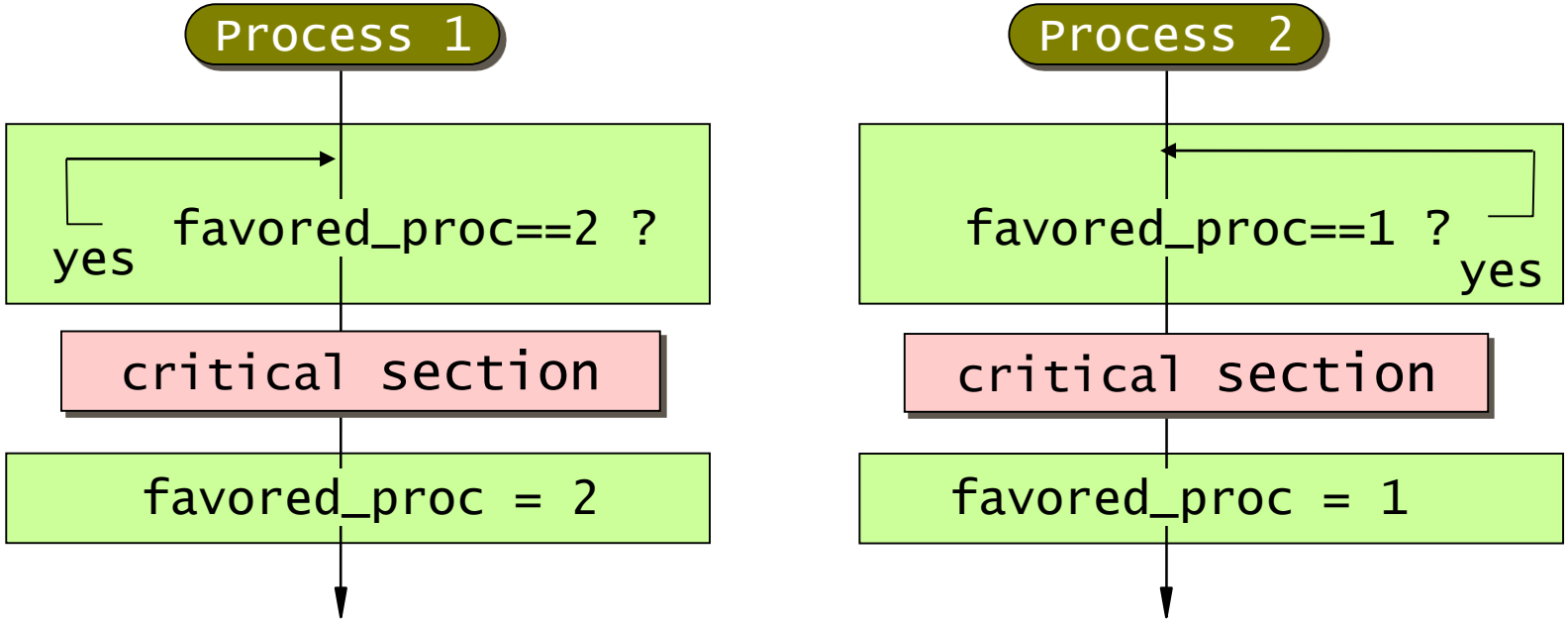
- In Mehrprozessorsystemen kann trotz Unterbrechungs-sperre eine verzahnte Ausführung von kritischen Operationen vorkommen
- Szenario
  - Kritische Operationen werden simultan auf zwei Prozessoren ausgeführt
  - Die zugehörigen Speicherzugriffe laufen verzahnt ab

⇒ Alternative Lösung für Mehrkernsysteme erforderlich



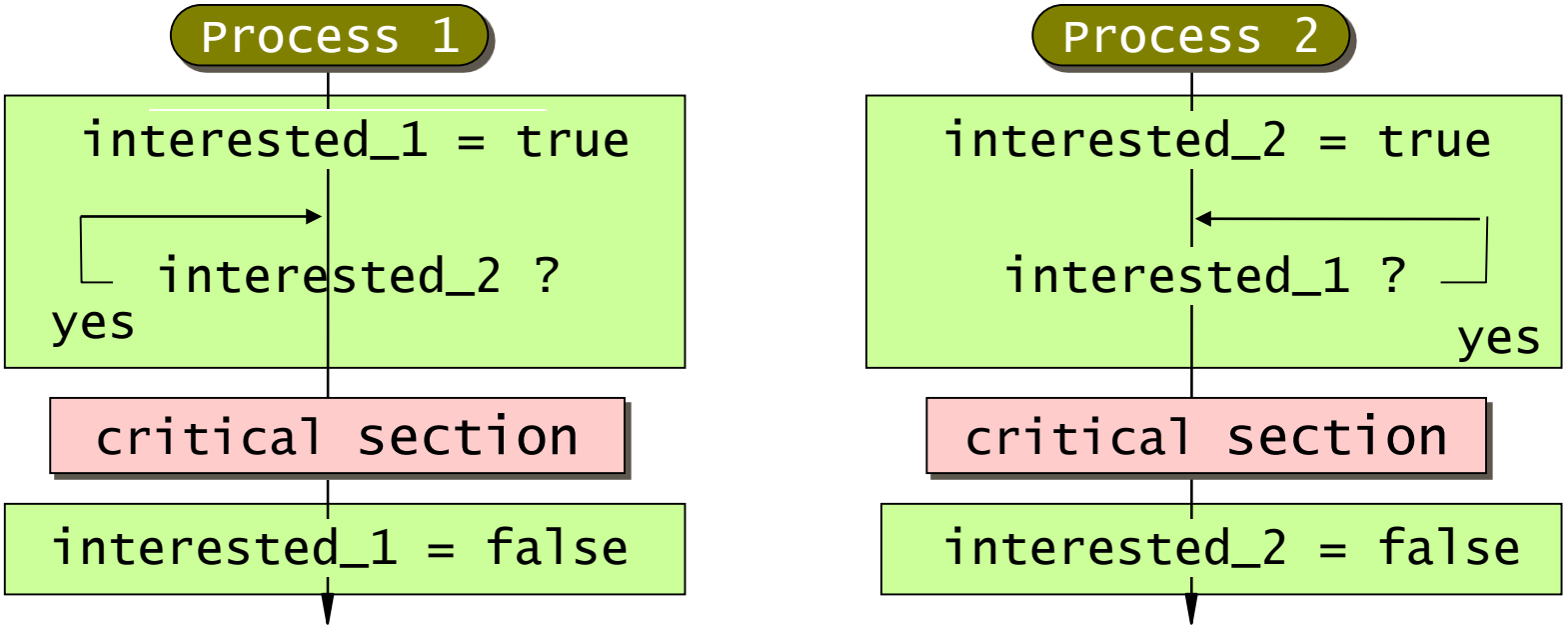
Die Prozesse werden parallel und geringfügig zeitlich versetzt ausgeführt, d.h. eine Unterbrechung findet hier nicht statt

Lösungsversuch 1



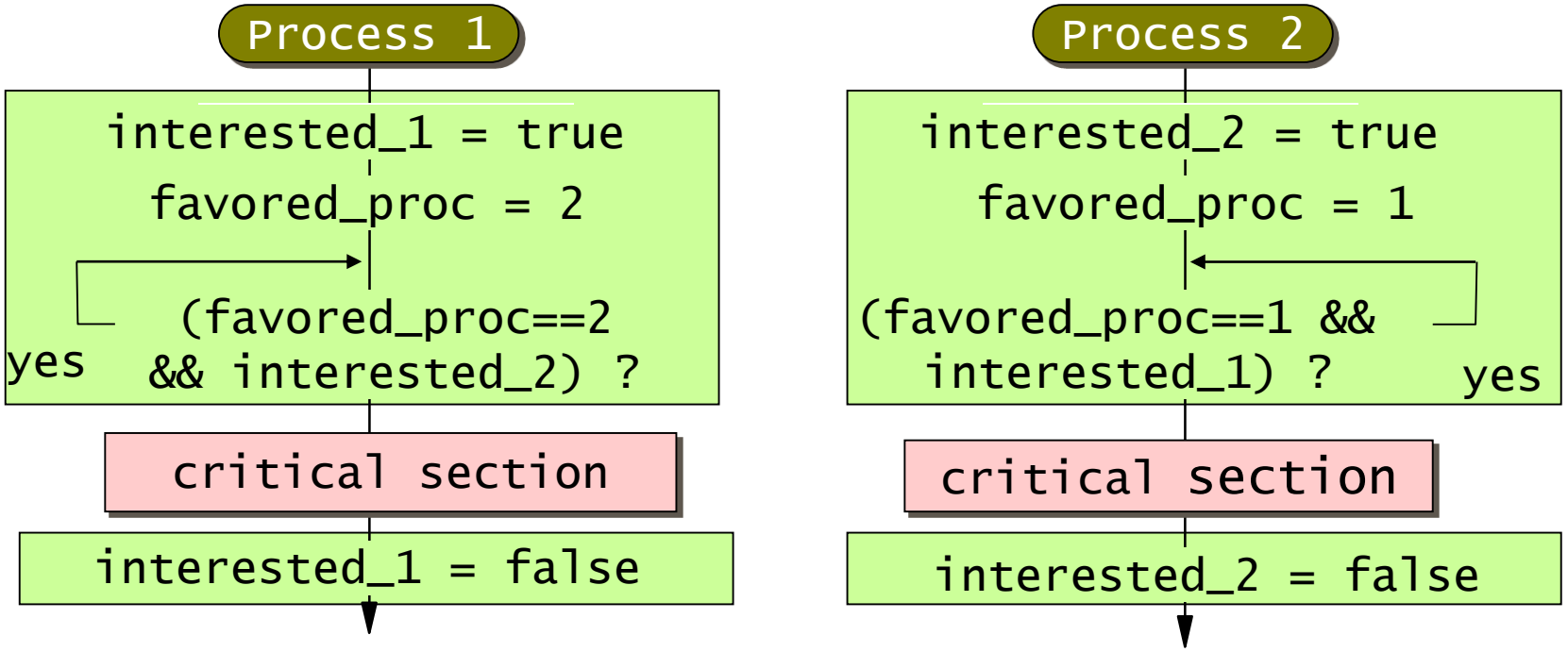
- Festlegung der auszuführenden CPU (kein Wechsel zwischen Kernen)
  - ⇒ Gegenseitiger Ausschluss funktioniert, es wird jedoch eine bestimmte Ausführung erzwungen
  - ⇒ Wenn der kritische Abschnitt in einer Schleife liegt, muss er alternierend von P1 und P2 betreten werden.
  - ⇒ Keine allgemeine Lösung!

Lösungsversuch 2



- Idee: Zugang zu CPU durch kritischen Abschnitt regeln
- Was passiert, wenn beide gleichzeitig die erste Anweisung (interested_i = true) ausführen?
  - ⇒ Deadlock (Verklemmung), da beide Prozesse in Warteschleife hängen

Lösungsversuch 3

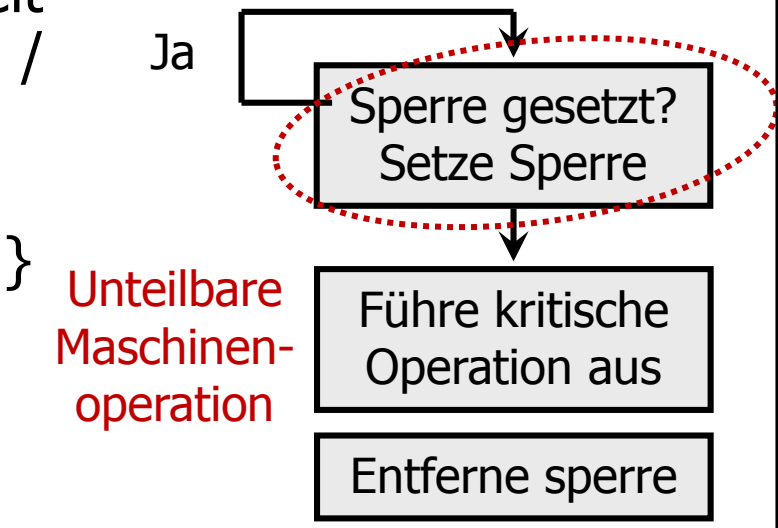


- Kombination der vorangegangenen Lösungsideen (Peterson, 1981)
  - Mit interested_x bekundet Prozess x seine Eintrittsabsicht
  - Variable favored_process nur dann wirksam, wenn beide Prozesse interessiert sind

  ⇒ Entscheidung FCFS

Konstruktion spin lock

- Hardwaregestützte Lösung: Unteilbarkeit der Kombination (Überprüfe Bedingung / Setze Bedingung) durch besonderen Maschinenbefehl realisiert
- test_and_set(reg, x)={load reg, x; x=1}
  - Sperrflag wird abgefragt UND ggf. gleichzeitig auf 1 gesetzt
- Abfrage der Minisperre in Schleife ⇒ aktives Warten (spin lock)




```
// spinlock (busy waiting)
   boolean  s = false;              // initialization
   void lock( boolean s){
      while (s == true) ;           // wait for signal
      s = true;
   }
   void unlock( boolean s){
      s = false;                     // set signal
   } // spinlock.
```

4.4 Betriebssystemgestützte Mechanismen für Prozessinteraktion

- Aktives Warten verschwendet Ressourcen
  - ⇒Blockierung der wartenden Prozesse, bis der aktuelle Prozess den kritischen Bereich verlassen hat
  - ⇒Wahl des nächsten Prozesses aus der Warteschlange (FCFS, Prioritäten oder andere Strategien)
- Betriebssystemunterstützung für Prozesssynchronisation
  - Semaphore (Dijkstra, 1965)
  - Monitore
- Operationen Sperren/Freigabe wirken direkt auf die Prozesszustände (Bereit, Laufend, Blockiert, Beendet)
- Voraussetzung
  - Operationen der Form test_and_set sind als atomare Operationen realisiert

Semaphore

- Semaphore stellen eine Zählsperre dar und bestehen aus
  - Nicht-negativ initialisiertem Zähler
  - Liste mit Verweisen auf involvierte Prozesse
- Bei negativen Zählerwerten (*nach Dekrementierung des Zählers! siehe Source Code*) werden die anfragenden Prozesse blockiert, bis mind. ein Prozess den kritischen Bereich verlässt
- Grundoperation P(*s*) (*Passieren des Semaphors*)
  - Der aktuelle Wert des Semaphorzählers wird dekrementiert
  - Passieren der Sperre und Eintritt in den kritischen Bereich
- Grundoperation V(*s*) (*Verlassen des kritischen Bereichs*)
  - Austritt aus dem kritischen Bereich
  - Der aktuelle Wert des Semaphorzählers wird inkrementiert
- Alternative Namen sind DOWN/UP

Beispielrealisierung der Operationen P und V

```
void P (Semaphor s) {
    zaehler(s)--;
    if (zaehler(s)<0) { /* Prozess(e)
    im kritischen Bereich */
        Zustand des aktuellen
          Prozesses Ta sichern
        Blockiere den Prozess Ta
          und füge Ta in die
          zugeordnete
          Warteschlange
        Wähle bereiten Prozess Tb
        Kontext von Tb laden;
    }
}
```

```
void V (Semaphor s) {
    if (zaehler(s)<0) { /*
    Prozess(e) im Zustand
    blockiert vorhanden */
        Bestimme z.B. den am
          längsten wartenden
          Prozess Tc
        Versetze Tc in den
          Zustand bereit
    }
    zaehler(s)++;
}
```

Funktionsweise der Semaphore

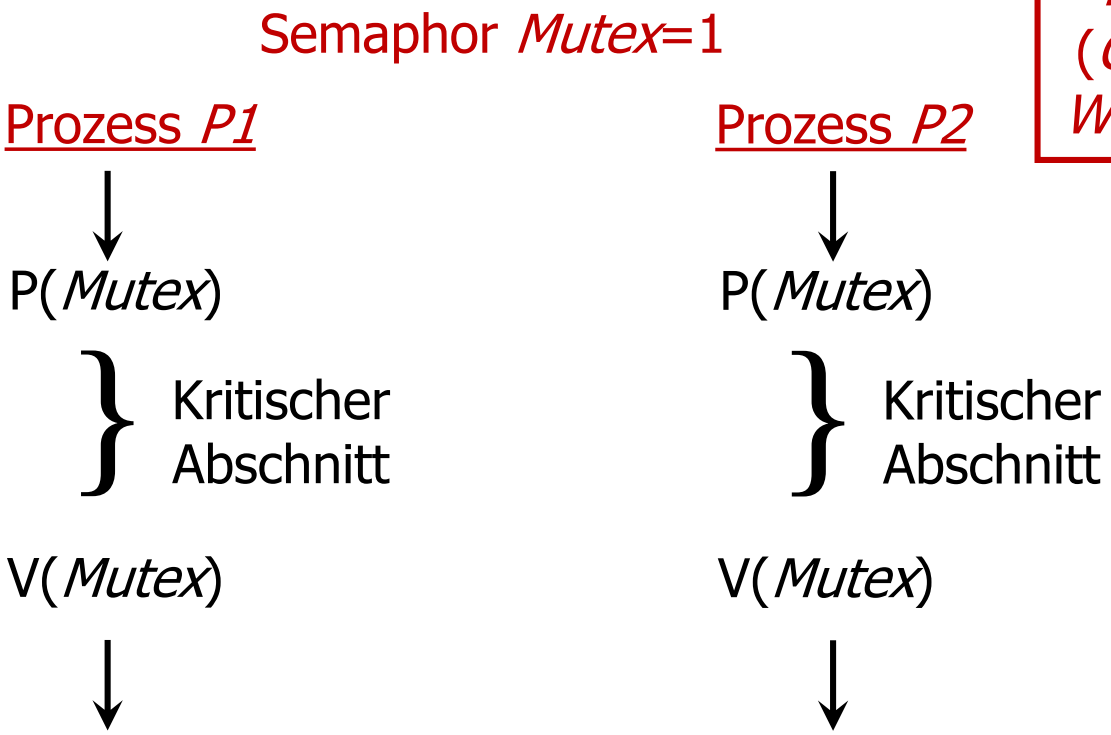
- P-Operation löst bei einem Zählerstand kleiner 0 (*nach Dekrementierung!*) eine Blockade des aktuellen Prozesses und eine zwangsläufige Umschaltung zu einem bereiten Prozess aus
  - Initialisierung des Zählers bestimmt, wie viele Prozesse sich gleichzeitig im kritischen Bereich aufhalten dürfen
    - Zähler = 1 $\Rightarrow$ binäre Variable, gegenseitiger Ausschluss
    - Zähler = k>1 $\Rightarrow$ k Prozesse dürfen im kritischen Bereich sein, z.B. k = Anzahl von FTP-Benutzern

- V-Operation ermöglicht bei jedem Aufruf den Einsatz eines blockierten Prozesses aus zugehöriger Warteschlange
  - Ankommende Prozesse werden üblicherweise in der Reihenfolge des Eintreffens in die Warteschlange eingefügt (FIFO)
  - Abweichungen von der FIFO-Reihenfolge sind zum Beispiel beim Echtzeitbetrieb notwendig, etwa Berücksichtigung von Prioritäten

Beispiel: Einfacher kritischer Abschnitt

- Zwei Prozesse *P1* und *P2* konkurrieren um den Eintritt in den kritischen Bereich

Mutex=Mutual Exclusion (*Gegenseitiger Ausschluss, Wechselseitiger Ausschluss*)



Semaphor

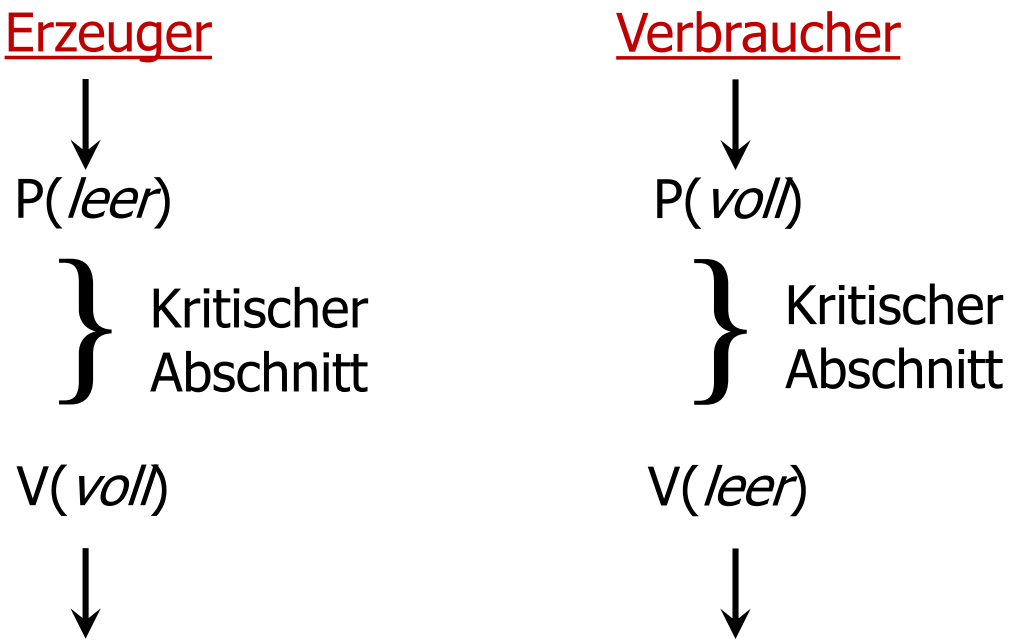
```
struct Semaphore {
   int count;          // process counter
   Queue *wp;          // count=1: free, count<=0: occupied
}                      // if count<0 : |count| is the
                       // number of waiting processes
void init (Semaphore *s, int i) {
    s->count = i;     // set i=1 for mutual exclusion
    s->wp = NULL;
}
void P(Semaphore *s) {
     s->count--;
     if (s->count < 0) block(s->wp);  // enqueue process
}
void V(Semaphore *s) {
     s->count++;
     if (s->count <= 0) deblock(s->wp) // deblock first of
}                                      // queue
```

Beispiel: Erzeuger – Verbraucher-System

- Zwei Prozesse kommunizieren über gemeinsamen Puffer
  - Der Erzeuger füllt den Puffer
  - Der Verbraucher konsumiert den Pufferinhalt
- Nebenbedingungen
  - Der Puffer hat eine beschränkte Aufnahmekapazität, d.h. der Erzeuger darf nichts hinzufügen, wenn der Puffer voll ist
  - Der Verbraucher darf nicht auf den Puffer zugreifen, wenn dieser leer ist
- Mögliche Ereignisse
  - Der Puffer ist voll $\Rightarrow$ Semaphor *voll* wird eingeführt
  - Der Puffer ist leer $\Rightarrow$ Semaphor *leer* wird eingeführt

Erzeuger – Verbraucher mit Semaphoren

Semaphor *leer*=1, *voll*=0



- Durch die Anordnung der Semaphore *voll* und *leer* „*Überkreuz*" wird eine abwechselnde Nutzung des Puffers gewährleistet

Reader-Writer-Problem

- Es seien *k* Sorten von Prozessen gegeben mit

c1	Prozesse der Sorte 1	und/oder
c2	Prozesse der Sorte 2	und/oder
…		
ck	Prozesse der Sorte *k*	

- Spezialfall Reader-Writer *k*=2, *c1*=1 und *c2*= ∞
  - Prozesse der Sorte 1 schreiben gemeinsam genutzte Daten
  - Prozesse der Sorte 2 lesen die Daten $\Rightarrow$ unkritische Zugriffe
- Im Kooperationsabschnitt dürfen sich daher entweder 1 Writer oder beliebig viele Reader aufhalten
- Verlässt ein Writer den Abschnitt und warten sowohl Reader also auch Writer, so kann man
  - Einen einzigen Writer deblockieren (*Schreibervorrang*)
  - Die Reader deblockieren (*Leservorrang*)

Lösung von Reader-Writer bei Bevorzugung der Reader

```
int Readernr=0;
Semaphor w, mutex=1
```

```
PROCESS Reader {
    ...
    P(mutex);
    Readernr++;
    if (Readernr==1) P(w);
    V(mutex);
    Lese Daten;
    P(mutex);
    Readernr--;
    if (Readernr==0) V(w);
    V(mutex);
... }
```

```
PROCESS Writer {
    ...
    P(w);
    Modifiziere Daten;
    V(w);
... }
```

Freigabe erst, wenn keine Reader mehr da sind

Lösung von Reader-Writer bei Bevorzugung der Writer

```
int Readernr, Writernr=0 ;
Semaphor mutex1, mutex2, mutex3, w, r=1;
```

```
PROCESS Reader {…
P(mutex3);
   P(r);
      P(mutex1);
           Readernr++;
           if (Readernr==1) P(w);
      V(mutex1);
   V(r);
V(mutex3);
Lese Daten;
P(mutex1);
         Readernr--;
         if (Readernr==0) V(w);
V(mutex1); … }
```

```
PROCESS Writer {…
P(mutex2);
   Writernr++;
   if (Writernr==1) P(r);
V(mutex2);
P(w);
Modifiziere Daten;
V(w)
P(mutex2);
   Writernr--;
   if (Writernr==0) V(r);
V(mutex2);
…
}
```

Posix Semaphoren

```
/* Initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process */
void Sem_init(sem_t *sem, int pshared, unsigned int value) {
  if (sem_init(sem, pshared, value) < 0)
    unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
  if (sem_wait(sem))
    unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
  if (sem_post(sem))
    unix_error("V");
}
```

Teilen mittels Posix Semaphoren

```
/* goodcnt.c - properly sync'd
counter program */
#include "csapp.h"
#define NITERS 10000000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;

    Sem_init(&sem, 0, 1); /* sem=1 */

    /* create 2 threads and wait */
    ...

   if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}
```

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

Erzeuger/Verbraucher: 1 Element Puffer

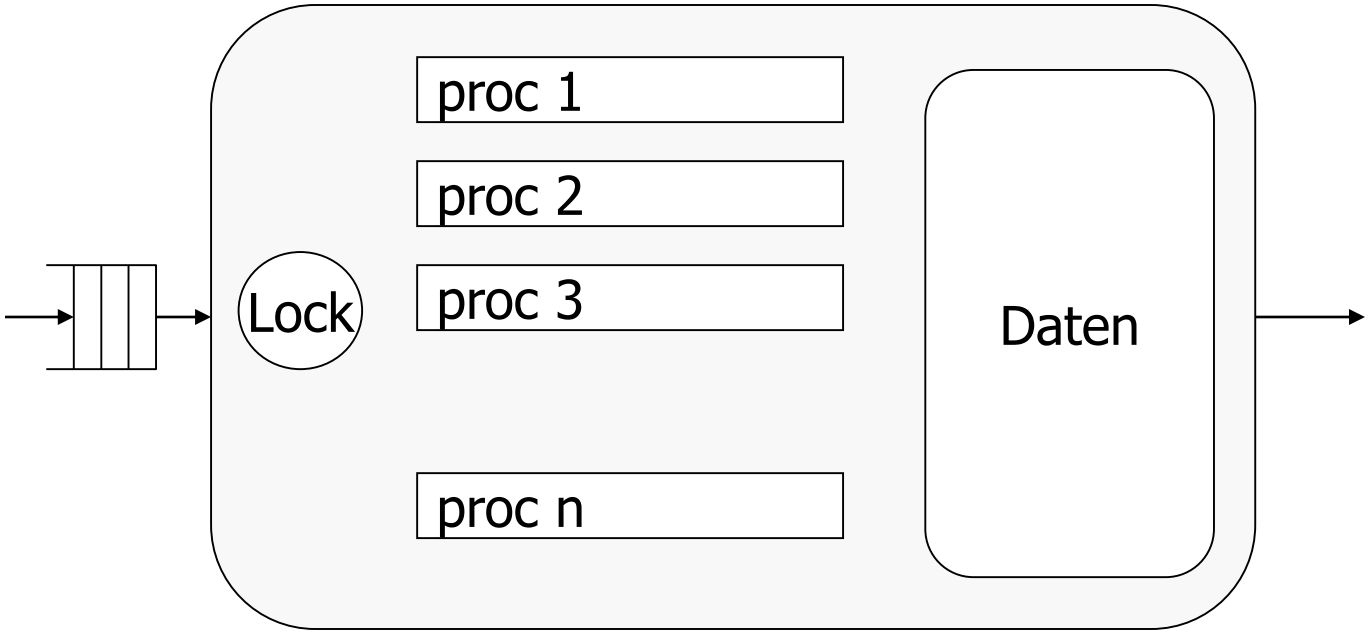
- Am Anfang: empty = 1 und full = 0

```
/* producer thread */
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return NULL;
}
```

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* consume item */
    printf("consumed %d\n",
            item);
  }
  return NULL;
}
```

4.5 Monitore

- Umgang mit Sperren ist fehleranfällig, da der Programmierer diese explizit und in korrekter Weise setzen muss
- Bedarf: automatisches Setzen und Freigeben der Sperren
  - ⇒ Entwicklung von Monitor-Objekten: Prozeduren und Datenstrukturen, die zu jedem Zeitpunkt nur von einem Prozess benutzt werden dürfen
  - ⇒ Sicherstellung des gegenseitigen Ausschlusses ohne dass der Programmierer explizit Sperroperationen einfügt (Hoare, 1974)



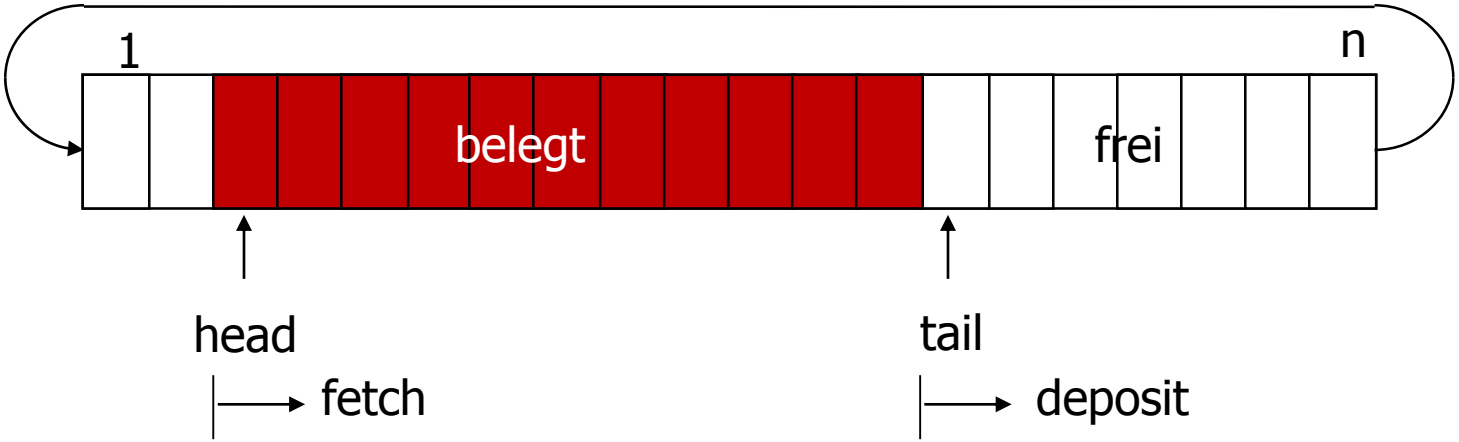
Monitor-Beispiel: Zähler

- Monitor führt automatisch und implizit das Setzen und Freigeben von Sperren durch
- Die von außen zugänglichen Monitor-Methoden sind mit `public` gekennzeichnet.

```
monitor shared_counter{
  public increment, decrement;
  int c = 0;

  void increment() {
        c = c + 1;
  }
  void decrement() {
        c = c - 1;
  }
} // end of shared_counter
```

Monitor-Beispiel Bounded Buffer

- Mehrere Prozesse benutzen einen gemeinsamen beschränkten Pufferbereich
  - Prozesse können Daten dort ablegen: **deposit(data)**
  - Prozesse können Daten dort abholen: **fetch(data)**
- Neben der Sicherstellung des gegenseitigen Ausschlusses (automatisch durch Monitor) müssen offensichtlich noch weitere Bedingungen berücksichtigt werden:
  - deposit darf nur aufgerufen werden, wenn noch Platz im Puffer ist
  - fetch darf nur aufgerufen werden, wenn der Puffer nicht leer ist



Monitor-Beispiel Bounded Buffer

```
monitor bounded_buffer {
public deposit, fetch;
   struct buffer_object {
        dataType buffer[n];
        int head = 1;
        int tail = 1;
        int count = 0;
        queue *WPD, *WPF;
   }
   void deposit(buffer_object *BB, dataType *data) {
        while (BB->count == n) block(BB->WPD);
        BB->buffer[BB->tail] = &data;
        BB->tail = (BB->tail % n) + 1; BB->count++;
        if (BB->WPF != NULL) deblock(BB->WPF);
    }

   void fetch(buffer_object *BB, dataType *result) {
        while (BB->count == 0) block(BB->WPF);
        &result = BB->buffer[BB->head];
        BB->head = (BB->head % n) + 1; BB->count--;
        if (BB->WPD != NULL) deblock(BB->WPD);
    }
} // bounded_buffer;
```

blockiert auch den Monitor

Bedingungssynchronisation

- Während ein Prozess auf eine Bedingungsvariable (im Beispiel: nicht leer / nicht voll) wartet, muss der Monitor für andere Prozesse freigegeben werden.

  $\Rightarrow$ gezeigte Lösung führt daher zu einer wechselseitigen Blockierung

- Konzept der Bedingungssynchronisation

  1. cwait(c) — Prozess gibt Monitor frei und wartet auf das nachfolgende csignal(c), d.h. das Eintreten der Bedingung c. Danach setzt er im Monitor fort. Der Prozess wird auf jeden Fall blockiert!
  2. csignal(c) — Ein wartender Prozess wird freigegeben. Der Monitor ist wieder belegt.

     Gibt es keinen wartenden Prozess, so hat die Prozedur keinen Effekt.

- Wartende Prozesse werden in einer Warteschlange verwaltet

Monitore in Java

- Zur Synchronisation nebenläufiger Threads stellt Java ein Monitor-Konzept zur Verfügung.
  - Methoden eines Objekts, die mit „synchronized" gekennzeichnet sind, stehen automatisch unter gegenseitigem Ausschluss
  - wait() und notify() stehen zur Bedingungssynchronisation zur Verfügung
  - wait() gibt temporär alle implizit durch synchronized belegten Sperren frei
- Implementierung
  - Anwendung auf eine Methode oder auf einen Block innerhalb einer Methode
  - Interne Realisierung
    - Setzen einer Sperre auf eine Objektvariable
    - Setzen einer Sperre auf den this-Pointer wenn sich synchronized auf eine komplette Methode bezieht

Monitore (2)

- Anwendung von synchronized auf Block von Anweisungen

  synchronized(getClass()){ System.out.println(zaehler++); ...}

  - Erzeugung eines neuen Klassenobjekts (getClass()), das bei jedem Zugriff auf den Zähler gesperrt wird

    ⇒Synchronisation mit einem Sperrobjekt (analog zu Sperrvariable)

- Anwendung von synchronized auf eine Methode

  - Mehrere Threads müssen ein Objekt werden

    ⇒Objektzugriff muss synchronisiert werden

  - Verwendung von synchronized in der Klassendefinition

    public synchronized int zaehler() { int ret = zaehler; ... }

    ⇒beim Aufruf der Methode wird this gesperrt

    ⇒Zugriff für andere Threads unmöglich

    ⇒Nach Verlassen der Methode wird this freigegeben

Blockieren und Freigeben von Threads

- Blockieren von Threads
  - void wait(): Thread blockiert bis ein anderer Thread notify() oder notifyAll() aufruft
  - void wait(long timeout): Wartezeit explizit begrenzt
  - Thread wird in Warteschlange des gesperrten Objekts eingereiht
- Freigeben von blockierten Threads
  - void notify(): Setzt einen Thread nach vorgegebener Strategie (meistens FIFO) frei
  - void notifyAll(): alle blockierten Threads werden freigegeben und konkurrieren erneut um die Sperrvariable
- wait() und notify()/notifyAll() können nur im Kontext von synchronized aufgerufen werden

Ergänzende Literatur

- Stallings,W.: Operating Systems 6th ed., Prentice Hall Chapter 5
- Bacon, J., Harris,T.: Operating Systems, Chap 9-14
- Hoare, C.A.R.: Monitors: An Operating System Structuring Concept. Comm. ACM 17 (1974). p. 549
- Lamport, L.: The Mutual Exclusion Part I: A Theory of Interprocess Communication, Journal of the ACM 33 (1986), pp. 313-326.
- Lamport, L.: The Mutual Exclusion Part II: Statement and Solutions, Journal of the ACM 33 (1986), pp. 327-348.
- Raynal, M.: Algorithms for Mutual Exclusion. MIT Press (1986)
- Wettstein, H.: The Problem of Nested Monitor Calls Revisited, ACM Operating System Review 12 (1978), pp.19-23