# 4. Koordination nebenläufiger Prozesse

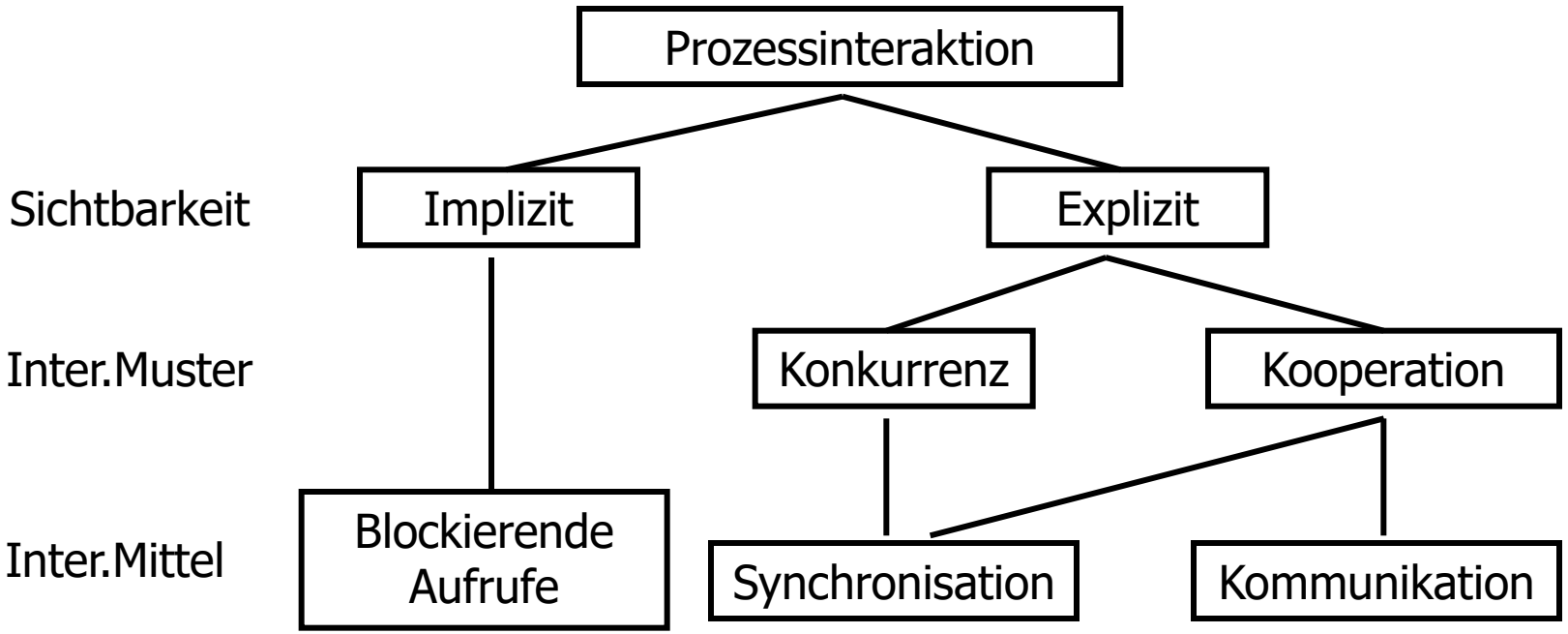
- Überblick
  - 4.1 Elementare Koordinationsoperationen
  - 4.2 Signalisierung
  - 4.3 Kritische Abschnitte
  - 4.4 Betriebssystemgestützte Prozessinteraktion
  - 4.5 Monitore

# 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
  - die Prozesse Daten aus unterschiedlichen Quellen austauschen
- Unterstützung der Prozess- und Threadinteraktion 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



# Implizite Prozessinteraktion (Beispiel)

- Klassische Kombination von Unix-Prozessen: die Pipe

```
user@machine:~$ ls -l
total 8
rw-r--r-- 1 user user 3 Jun  4 12:24 foo.txt
rw-r--r-- 1 user user 3 Jun  4 12:26 bar.txt
user@machine:~$ ls -l | grep foo
rw-r--r-- 1 user user 3 Jun  4 12:24 foo.txt
```

- Unidirektionaler Datenstrom vom Erzeuger- zum Verbraucher-Prozess

```
// ls-Prozess
while (Verzeichniseinträge) {
    write(stdout, Zeile);
}
```

```
// grep-Prozess
while (Datenstrom nicht zu Ende) {
    read(stdin, Zeile);
    if (Zeile passt zu Filter) {
        ausgeben(Zeile);
    }
}
```

***Implizite Interaktion (read() blockiert, wenn ls „zu langsam“ Daten liefert)***

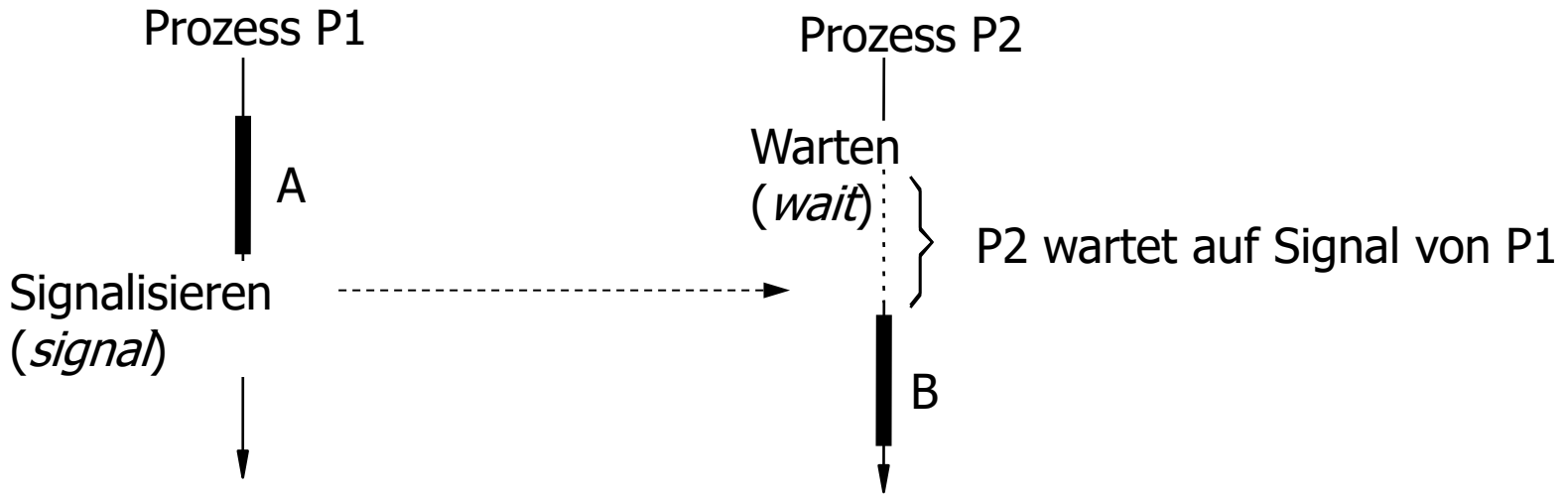
# Prozessinteraktion: Konkurrenz

- 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 zeitlich verzögert

- Die zeitliche Abstimmung konkurrierender Prozesse wird als Prozesssynchronisation bezeichnet

# 4.2 Signalisierung

- Elementare Aufgabe: Exklusives Sperren/Freigeben einer Variable (Sperrflag) durch konkurrierende Prozesse/Threads
  - Voraussetzung: 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) mit Sperrvariable s

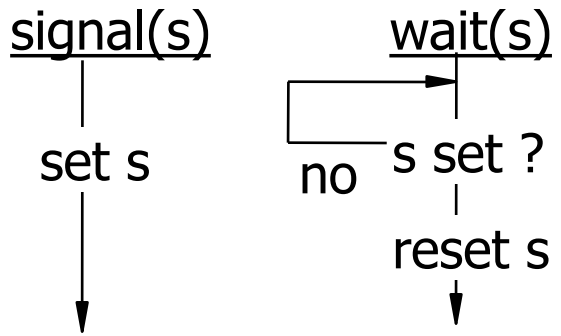



Staffelstabübergabe

# Grundform der Signalisierung: aktives Warten

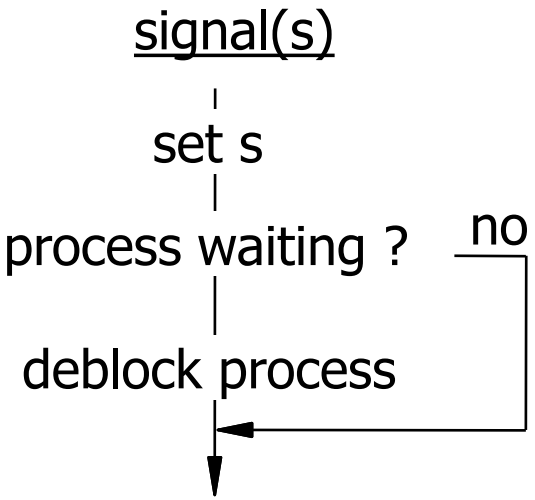
- Busy waiting (aktives Warten): Wiederholende Statusabfrage als Schleife

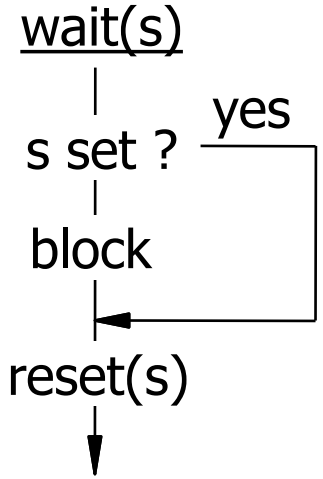
```
// Signalisierung mit Busy-Waiting

struct Signal { // Version 1
      boolean s;
} so = { false };                    //Initialisierung

void signal(struct Signal *so) {
      so->s = true;
}

void wait(struct Signal *so) {
      while (so->s == false) { ; } //Warten, bis gesetzt
      so->s = false;
}
```



# Grundform der Signalisierung: blockierendes Warten

- Aktives Warten verschwendet Prozessorkapazität und blockiert die CPU
- Lösung
  - Wenn gewartet werden muss, dann CPU freigegeben und wartenden Prozess in Zustand *blockiert* versetzen
  - Aus dem Wartezustand (*blockiert*) muss der Prozess explizit durch das Signal herausgeholt werden



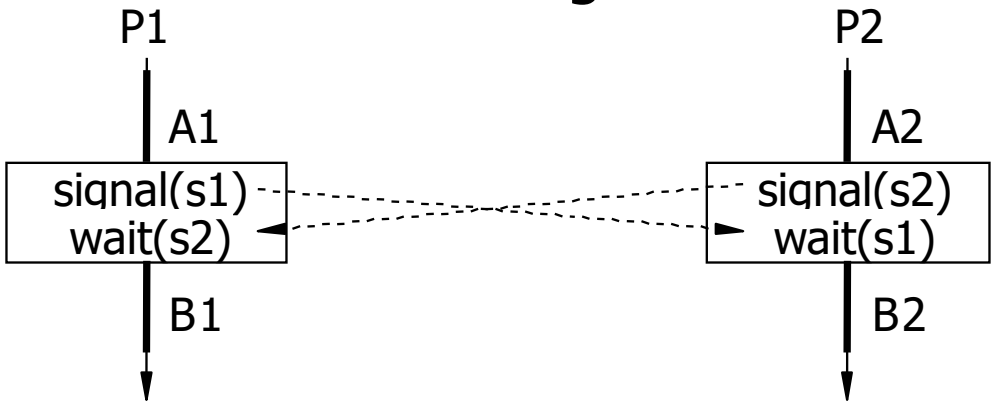



# Grundform der Signalisierung: blockierendes Warten (Code)

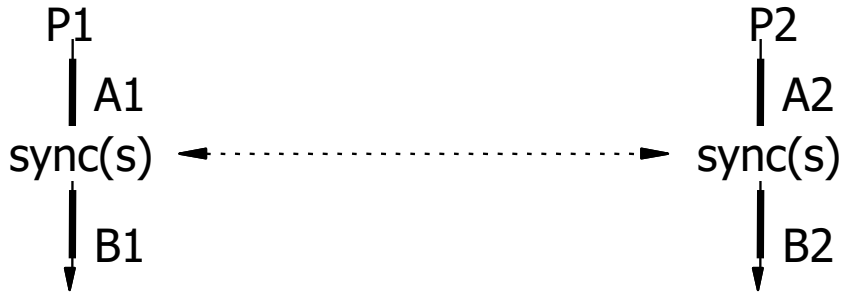
```
// Signalisierung mit blockierendem Warten

struct Signal { // Version 2
      boolean s;
      process *wp;
} so = { false, NULL };                  // Initialisierung

void signal(struct Signal *so) {
      so->s = true;
      if (so->wp != NULL) {              // Wartet ein Prozess?
            deblock(so->wp);             // Blockierung aufheben
            so->wp = NULL;               // Struktur zurücksetzen
      }
}
void wait(struct Signal *so) {
      if (so->s == false) {              // noch kein Signal?
            so->wp = MYSELF;
            block();
      }
      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)



# 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
  - Modifikation gemeinsamer Strukturen (z.B. verkettete Liste)
- Absicherung durch Sperrvariable („Sperrflag“)

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

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

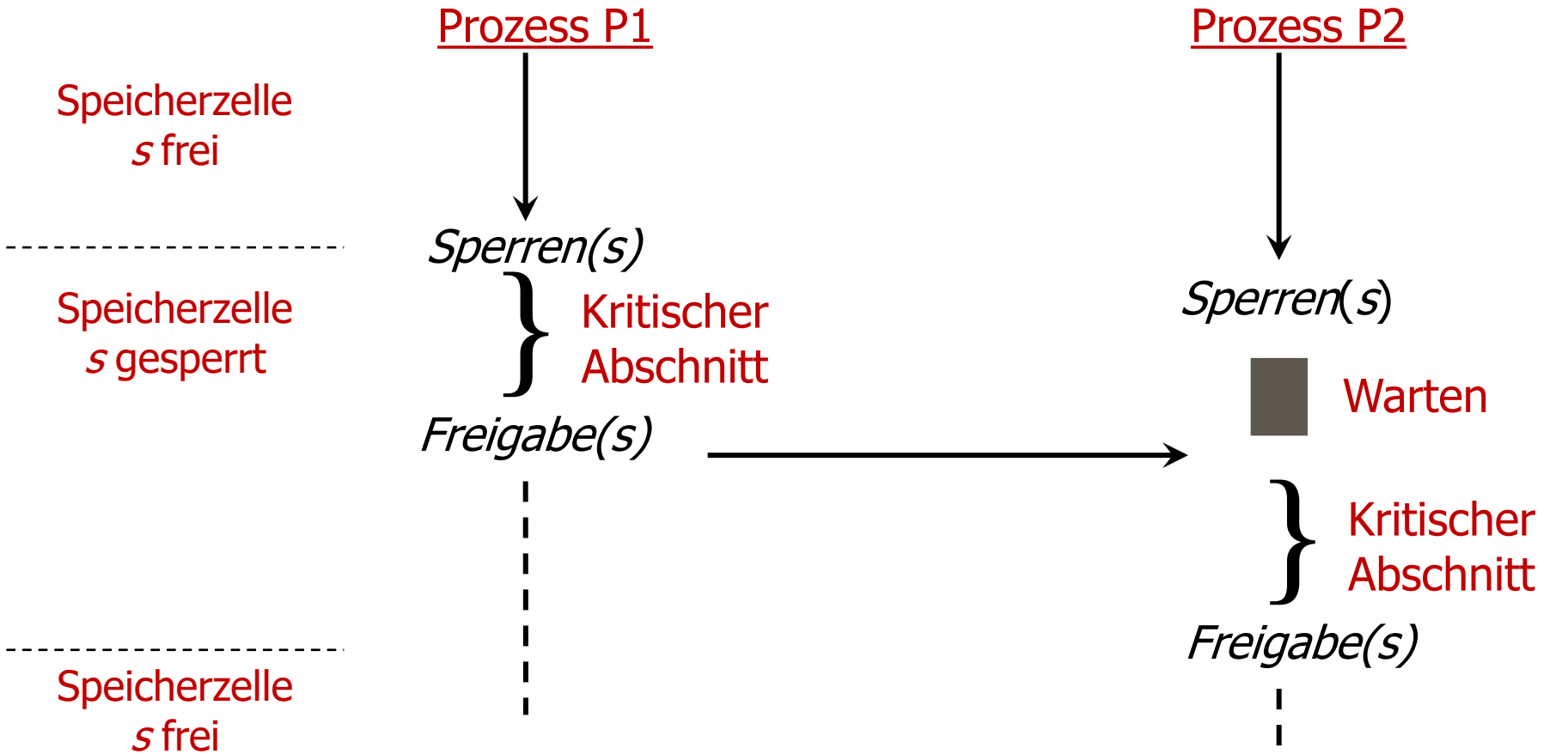
# Prozesssynchronisation mittels kritischer Abschnitte

- 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

- Idee: Sperrflag initial frei – beim Betreten des Abschnitts Sperrflag setzen und so andere Prozesse ausschließen – nach Verlassen Freigeben des Sperrflags („Der Nächste, bitte!“)



# 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);
```

- Verzahnte Ausführung hinterlässt inkonsistenten Datenbestand. Warum?

# Nebenläufigkeit und fehlende Synchronisierung

- Der Teufel steckt im Detail
- Annahme: verzahnte Ausführung zweier Threads A und B, Umschaltung innerhalb des roten Blocks
  1. A: liest den Wert 10 → v0, inkrementiert v0
  2. Umschaltung zu B: Mehrere Durchläufe, cnt wird auf 20 erhöht
  3. Umschaltung zu A: schreibt v0 (mit Wert 11!) in Variable cnt
- Eine Menge Iterationen sind verloren gegangen!

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

```
count:
        [...]
        lw v0, 40(gp)
        addiu v0, v0, 1
        sw v0, 40(gp)
        [...]
```

```
lw = load word from address
40(gp) into v0
addiu= add immediate
unsigned no overflow:v0=v0+1
sw = store word at the
specified address
gp = global pointer
```

# Nebenläufigkeit und Sperrflag-Operationen

- Was bedeutet das für Idee mit Sperrflags?
- Annahme: zwei Threads A und B betreten wait()
  1. A:
     - Sperrflag auslesen → v0
     - Bedingung testen: FREI!
  2. Umschaltung zu B:
     - Sperrflag auslesen → v0
     - Bedingung testen: FREI!
  3. Danach in A und B:
     - Sperrflag setzen, Bereich betreten
- Beide haben die Bedingung passiert! Unter Umständen passiert eine Katastrophe!

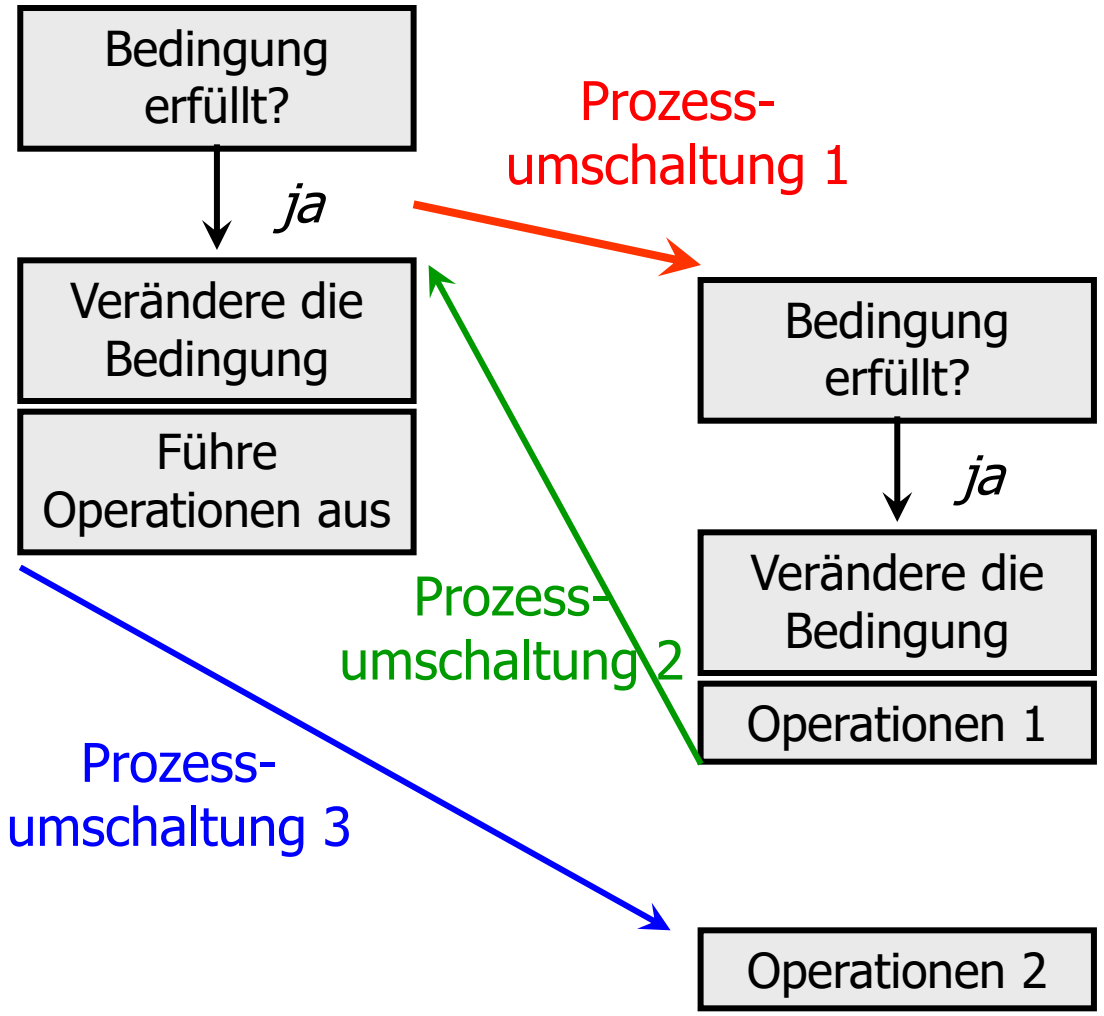
```
/* teste&setze Sperrflag */
void wait(boolean *flag) {
   while (*flag == false)
      { ; }
   *flag = false;
}
```

```
wait:

        [...]
T:      lw v0, 0(a0)
        beqz v0, T
        [...]
        sw zero, 0(a0)
        [...]
```

beqz v0,T -> # branch to T if register v0 == 0

# Nebenläufigkeit und Sperrflags: Diskussion

- Bei verzahnter Ausführung kann nicht ausgeschlossen werden, dass zwischen Abfrage und Änderung der Bedingung ein Umschalten stattfindet und dadurch zwei (oder noch mehr) Prozesse den kritischen Bereich betreten



# Thread-safe = Programmierkonzept für Multithread-Code

- Thread-Safe Garantie
  - Veränderung gemeinsam genutzte Datenstrukturen so, dass sich alle konkurrierenden Threads gemäß ihrer Designspezifikationen ohne unbeabsichtigte Interaktion verhalten

- Ansatz 1:  Vermeidung gemeinsamer Zustände
  - Wiedereintritt: Speichern von Statusinformationen in lokalen Variablen
  - Thread-lokaler Speicher: jeder Thread hat seine eigene private Kopie aller Variablen
  - Unveränderliche Objekte: nur schreibgeschützte Daten werden gemeinsam genutzt

- Ansatz 2: Synchronisation, wenn gemeinsamer Zustand notwendig
  - Gegenseitiger Ausschluss: Serialisierung der Zugriffe
  - Atomare Operationen: Zugriff über nicht unterbrechbare Operationen → Verwendung spezieller Maschinenbefehle wie Compare-and-Swap oder Load-Linked/Store-Conditional

# 4.4 Betriebssystemgestützte 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: atomare Instruktionssequenz für Prüfen/Setzen verfügbar (ist heute auf allen gängigen Plattformen der Fall)

# Semaphor

- Semaphore stellen eine Zählsperre dar und bestehen aus
  - Nicht-negativ initialisiertem Zähler
  - Liste mit Verweisen auf involvierte Prozesse

- Grundoperation P(s) (Passieren des Semaphors)
  - Der aktuelle Wert des Semaphorzählers wird dekrementiert
  - Wenn Zähler nicht-negativ: Passieren der Sperre, Eintritt in den kritischen Bereich
  - Wenn Zähler negativ: Blockieren

- Grundoperation V(s) (Verlassen des kritischen Bereichs)
  - Austritt aus dem kritischen Bereich
  - Wenn Zähler negativ: Deblockieren eines wartenden Prozesses
  - Der aktuelle Wert des Semaphorzählers wird inkrementiert

# Semaphor

Signal

Warteschlange

FIFO

# Funktionsweise des Semaphors

- **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 maximalen Downloadprozessen

- **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)
  - FIFO-Abweichungen beim Echtzeitbetrieb notwendig, z.B. Berücksichtigung von Prioritäten

# Beispielrealisierung der Operationen P und V

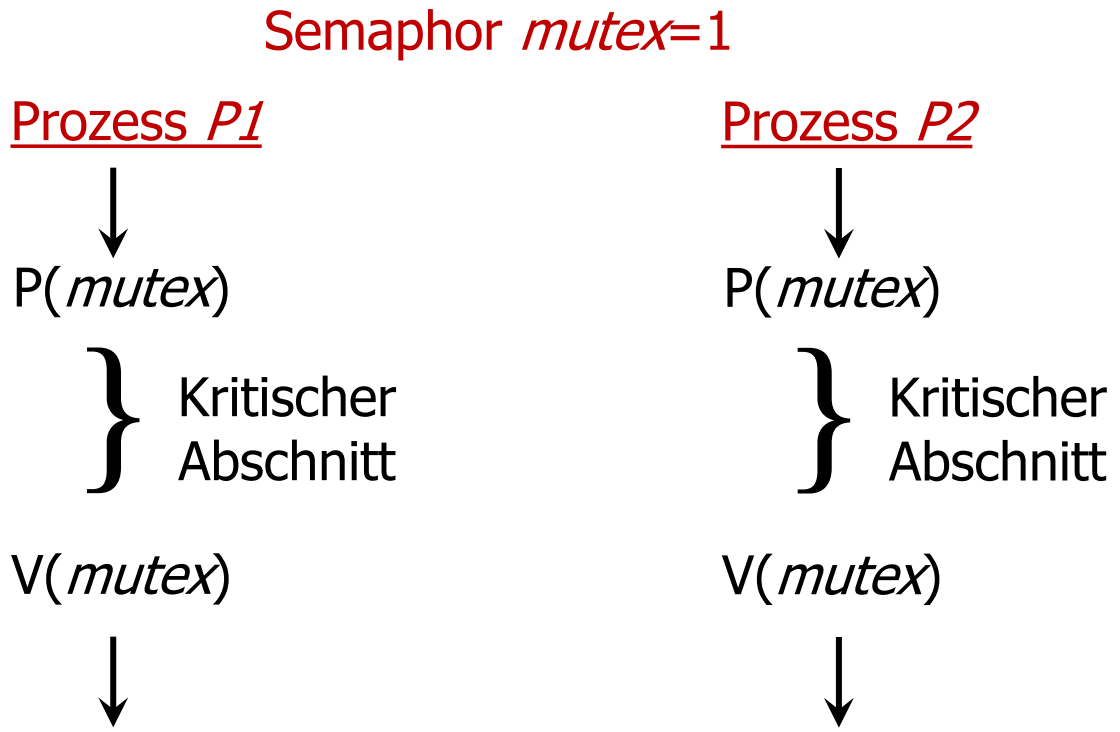
```
void P(Semaphor s) {
    s.zaehler--;                          // Zähler dekrementieren
    if (s.zaehler < 0) {                  // kleiner 0: Semaphor sperrt
        currentProcess.state = BLOCKED;
        enqueue(s.queue, currentProcess);
        scheduler();                      // anderen Prozess auswählen
    }
}

void V(Semaphor s) {
    if (s.zaehler < 0) {                  // es gibt wartende Prozesse:
        Process p = dequeue(s.queue);     // einen auswählen... (FCFS?)
        p.state = READY;                  // ... und aufwecken
        /* ggf. auch hier: scheduler() (Verdrängung prüfen!) */
    }
    s.zaehler++;                          // Zähler inkrementieren
}
```

Zähler positiv: Anzahl Prozesse, die den Semaphor passieren dürfen

Zähler negativ: Anzahl Prozesse, die bereits am Semaphor warten

# Beispiel: Einfacher kritischer Abschnitt

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



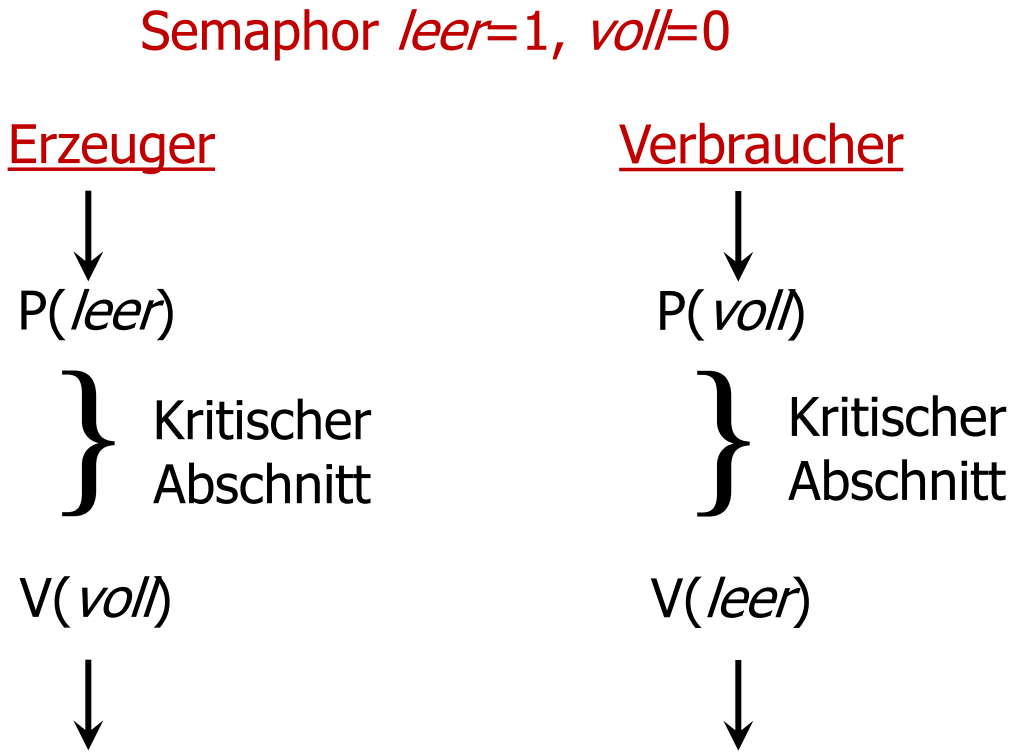

*Mutex* = Mutual Exclusion (*Gegenseitiger Ausschluss, Wechselseitiger Ausschluss*)

# 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

- Durch Anordnung der Semaphore voll und leer „über Kreuz“ wird eine abwechselnde Nutzung des Puffers gewährleistet (kein Unter-, kein Überlauf)



# Reader-Writer-Problem

- Es seien k Sorten von Prozessen gegeben mit

  c1 Prozessen der Sorte 1 und/oder
  c2 Prozessen der Sorte 2 und/oder
  …
  ck Prozessen 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 ⇒ unkritische Zugriffe
- Im kritischen Bereich sind entweder 1 Writer oder beliebig viele Reader
- Verlässt ein Writer den Abschnitt und warten sowohl Reader also auch Writer, so kann man
  - Einen einzigen Writer deblockieren (Schreibervorrang)
  - Alle wartenden Reader deblockieren (Leservorrang)

# Lösung von Reader-Writer bei Bevorzugung der Reader

```
Semaphor w=1;
```

```
PROCESS Reader {
        …

        P(w);
        Lese Daten;
        V(w);

… }
```

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

… }
```

# Lösung von Reader-Writer bei Bevorzugung der Reader

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

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

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

... }
```

# Lösung von Reader-Writer bei Bevorzugung der Reader

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

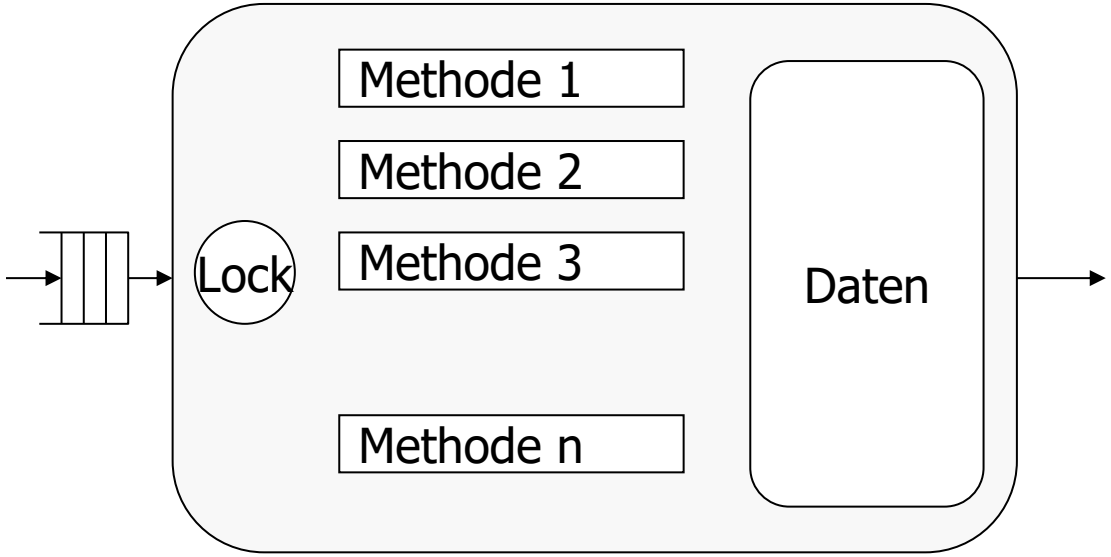
```
int Readernr, Writernr=0 ;
Semaphor mutex1, mutex2, mutex3, w, r=1;
```

```
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);
…
}
```

# 4.5 Monitore

- Umgang mit Sperren ist fehleranfällig, da Programmierer diese explizit und in korrekter Weise setzen müssen
- Bedarf: automatisches Setzen und Freigeben der Sperren
  - Monitor-Objekten: Methoden und Datenstrukturen, die zu jedem Zeitpunkt nur von einem Prozess benutzt werden dürfen
  - Sicherstellung des gegenseitigen Ausschlusses ohne explizites Einfügen von Sperren



# Monitor-Beispiel: Zähler

- Monitor führt automatisch und implizit das Setzen und Freigeben von Sperren durch
- Alle Funktionen schließen einander gegenseitig aus

```
monitor shared_counter {
  int c = 0;

  public void increment() {
    c = c + 1;
  }
  public void decrement() {
    c = c - 1;
  }
};
```

=

```
class shared_counter {
  private mutex m;
  int c = 0;

  public void increment() {
    mutex_lock(m);
    c = c + 1;
    mutex_unlock(m);
  }
  public void decrement() {
    mutex_lock(m);
    c = c - 1;
    mutex_unlock(m);
  }
};
```

# Bedingungssynchronisation

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

- Konzept der Bedingungssynchronisation
  - cwait(c) Prozess gibt Monitor frei und wartet (blockiert) auf das nachfolgende csignal(c). Nach Auflösen der Blockierung ergreift er den Monitor wieder.
  - csignal(c) Ein wartender Prozess wird freigegeben. 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

# Monitor-Beispiel Bounded Buffer

```
public class BoundedBuffer {
        Object buffer[n];
        int head;  int tail; int count;

        public BoundedBuffer() { /* initialize */ }
        public synchronized void deposit(Object data) {
                while (count == n) wait();
                buffer[tail] = data;
                tail = (tail + 1) % n; count++;
                notifyAll();
  }
        public synchronized Object fetch() {
                Object result;
                while (count == 0) wait();
                result = buffer[head];
                head = (head + 1) % n; count--;
                notifyAll();
                return result;} };
```