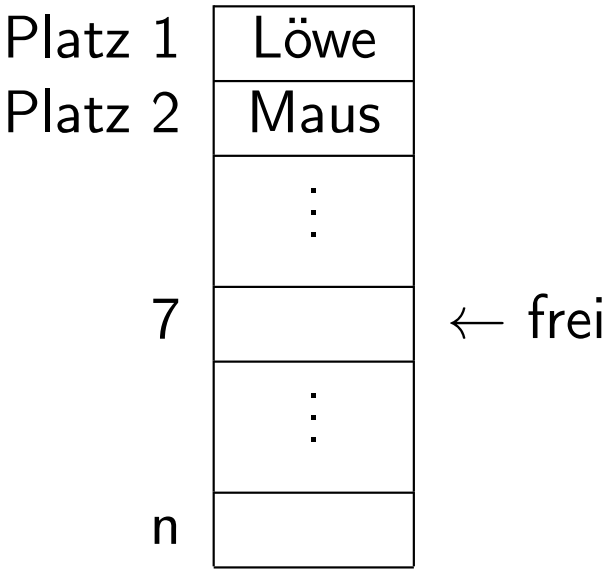
Beispiel: Platzbuchungsservice „Arche Noah"

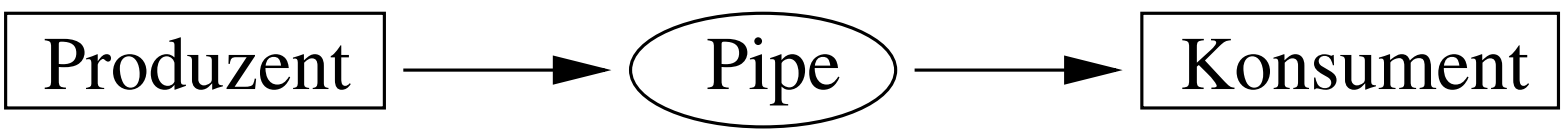
n Plätze




frei := Variable, die die Nummer des nächsten zu belegenden Platzes enthält.

Beispiel: Prozeßkommunikation mittels Pipe

Eine Pipe ist ein Datenstrom zwischen zwei Prozessen A und B. Die Pipe lenkt die Ausgabe von A auf die Eingabe von B um.

Definition 20:

a) Ist das Ergebnis einer Ausführung nebenläufiger Prozesse von der zeitlichen Reihenfolge der Read- und Write-Operationen abhängig (Race Condition), so heißt sie zeitkritische Ausführung.

b) Der Teil eines Programms, in dem lesend oder schreibend auf Variable, die mehreren Prozessen gemeinsam ist, zugegriffen wird, heißt kritischer Bereich.

Gesucht: Verfahren für wechselseitigen Ausschluß (Mutual Exclusion), d.h. Verfahren, die garantieren, daß sich zu jedem Zeitpunkt jeweils nur ein Prozeß in einem kritischen Bereich befindet.
Anforderung: Kein Prozeß soll unendlich lange warten, bis er einen kritischen Bereich betreten darf (Starvation Free).

# Signalfehler, Nordindien, Mai 2014

*Beim Zusammenstoß eines Schnellzugs mit einem Güterzug wurden 40 Menschen getötet. Der Express-Passagierzug war mit hoher Geschwindigkeit unterwegs. Dieser habe zwar noch gebremst, dennoch ist er in den auf dem Gleis stehenden Güterzug hineingerast und entgleist. Untersuchungen deuteten darauf hin, dass die beiden Züge wegen eines Signalfehlers auf das gleiche Gleis gerieten.*

*Und das obwohl sich grundsätzlich Signale in Außenanlagen gegeneinander ausschließen, die zu einer gefährlichen Bewegung führen würden.* **...**
*Im Zuge der rasant voranschreitenden Digitalisierung werden immer mehr sensible Abläufe komplett computergesteuert. Hierzu zählen auch öffentliche Verkehrsmittel. In einigen Städten, wie zum Beispiel in London, Paris, San Francisco und Nürnberg, fahren Bahnen mittlerweile sogar ohne Zugführer.*

**Quelle:** *Das „Honeytrain Project“*, Zeitschrift für Automation und Security & KES, Sonderausgabe November 2014, S. 10-12.

**Bildquelle**:

`http://www.spiegel.de/panorama/indien-viele-tote-und-verletzte-bei-zugunglueck-a-967486.html`

**Koordinationsprobleme:**

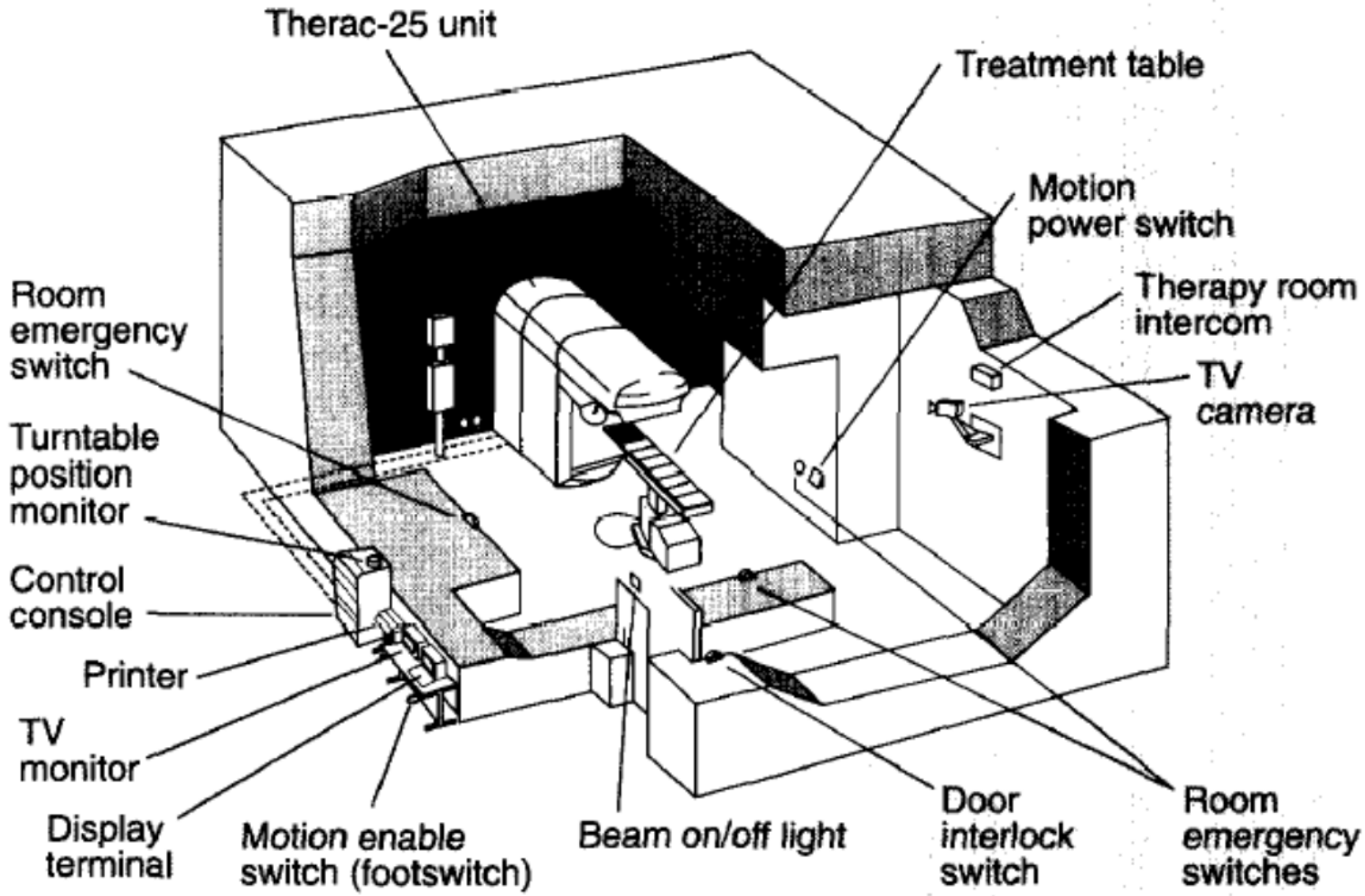
1. Kritischer Bereich
2. Producer-Consumer:
   wechselseitiger Ausschluß + Ablaufsteuerung für vollen bzw. leeren Puffer.
3. Readers & Writers:
   nebenläufige Leseprozesse sind erlaubt

# Therac-25

Nancy Leveson, Clark Turner: *"An Investigation of the Therac-25 Accidents"*

- Gerät zur Strahlentherapie von Krebspatienten der kanadischen Firma AECL
- 6 Bestrahlungsopfer durch Softwarefehler 6'85-1'87
- Therapie-Dosis 200 rad, Verletzungen duch Bestrahlung von Körperteilen mit 15000-20000 rad
- Was passierte bei den fehlerhaften Behandlungen?

# Therac-25


Therac-25 unit
Treatment table
Motion power switch
Therapy room intercom
TV camera
Room emergency switch
Turntable position monitor
Control console
Printer
TV monitor
Display terminal
Motion enable switch (footswitch)
Beam on/off light
Door interlock switch
Room emergency switches

## Ursache:

- PDP-11, eigenes Betriebssystem, Software komplett in Assembler von einer Person entwickelt
  **Vorgängermodell besaß Hardwareschutz, um eine Überdosis zu verhindern.**
  **Bei der Therac-25 wurde ganz auf die Softwaresteuerung vertraut.**
- Softwareaufgaben:
  - Meßwerterfassung und Steuerung des Geräts
  - Benutzerinteraktion (Eingaben verarbeiten)

  Eingestellte Strahlendosis ist eine der *gemeinsamen Variablen*.
  Beim Zugriff konnte es zu Race Conditions kommen.
  Ironie: Es enstanden keine Probleme, wenn die/der MTA langsam arbeitete.

- Unglücksursache: fehlende Synchronisation beim Zugriff auf gemeinsame Variable:
  *Das Kernproblem dabei war die korrekte Synchronisation der beiden Prozesse. Unter gewissen Umständen konnte es passieren, dass nach einer Korrektur der Eingabedaten durch den Bediener vom Computer bei der Ansteuerung des Gerätes nur bei einem Teil der Daten die korrigierten Daten, bei dem anderen aber die alten Daten vor der Korrektur verwendet wurden.*
- Unglücksursache: 8-Bit Kontrollvariable wurde (fälschlicherweise) hochgezählt. Bei Überlauf fand keine Positionsprüfung, ob das Wolframtarget wirklich im Strahlengang war, statt.
- Unglücksursache: Entwicklung und Testen in einer Hand,
- Unglücksursache: sinnlose Fehlermeldungen,
- Unglücksursache: fehlende Geschäftsprozesse zur Verfolgung von Störfällen verschleppten das Entdecken der Fehler

Und die Moral von der Geschicht:

Zusätzlicher Hardwareschutz ist für Safety-kritische Anwendungen zwingend erforderlich!

siehe auch Prof. Lutz Prechelt, Vorlesung „Anwendungssysteme“, Sicherheit: Therac-25, FU Berlin

und `https://de.wikipedia.org/wiki/Therac-25`

# Lösungsansätze für das erste Koordinationsproblem: Kritischer Bereich

Betritt ein Prozeß einen kritischen Bereich, darf er erst unterbrochen werden, wenn er diesen wieder verlassen hat.
$\hat{=}$ Sperren aller Unterbrechungen

Nachteil:

- nicht Starvation Free
- Verfahren wird nur für kurze Betriebssystemaufgaben benutzt, nicht für Benutzerprozesse

oft auch: Spin Locking
Softwarelösung!
Synchronisationsvariable:

$$\texttt{flag} = \begin{cases} 0 & \text{kein Prozeß ist im kritischen Bereich} \\ 1 & \text{kritischer Bereich ist belegt} \end{cases}$$

```
flag = 0                    /* Initialwert */
⋮
while (flag != 0);          /* aktives Warten */
flag = 1;
critical_region();
flag = 0;
```

⋮ **K.n.i.F.!!!**

# Lösungsverfahren für 2 Prozesse: Alternierender Zutritt

$$\texttt{turn} = \begin{cases} 0 & \text{Prozeß A darf den kritischen Bereich betreten} \\ 1 & \text{Prozeß B darf den kritischen Bereich betreten} \end{cases}$$

```
turn = 0   /* Initialwert*/
```

**Prozeß A**

```
⋮
while (turn != 0);
critical_region();
turn = 1;
⋮
```

**Prozeß B**

```
⋮
while (turn != 1);
critical_region();
turn = 0;
⋮
```

Bem.:

+ Verfahren garantiert wechselseitigen Ausschluß
! In Praxis: `turn` muß als *volatile* deklariert werden!
– Busy Waiting in der While-Schleife $\Longrightarrow$ CPU-Zeit-Verschwendung!
– striktes Alternieren

1965 Dekkers Algorithmus: Erstmals eine Softwarelösung ohne striktes Alternieren!

1981 Petersen-Algorithmus: einfacher!

# Peterson's Algorithmus (1981)

Für $i = 1, 2$:

$$\texttt{interested\_i} = \begin{cases} \text{true} & \text{Prozeß } P_i \text{ will kritischen Bereich betreten} \\ \text{false} & \text{Prozeß } P_i \text{ will kritischen Bereich nicht betreten} \end{cases}$$

$$\texttt{turn} = \begin{cases} 1 & \text{Prozeß } P_1 \text{ hat Vortritt.} \\ 2 & \text{Prozeß } P_2 \text{ hat Vortritt.} \end{cases}$$

Initialisierung:

```
turn          =  1
interested_1  =  false
interested_2  =  false
```

```
   Prozess   P_1:                    Prozess P_2:

          ...                              ...
    interested_1 = true;             interested_2 = true;
Z1: turn = 2;                    Z2: turn = 1;
    WHILE (interested_2              WHILE (interested_1
       and (turn = 2))                  and (turn = 1))
    DO skip;                         DO skip;
    critical_section();              critical_section();
    interested_1 = false;            interested_2 = false;
```

Theorem 2: Der Peterson Algorithmus garantiert wechselseitigen Ausschluß ohne strenges Alternieren und ist fair.

Wiederholung: Was war das Problem bei der ersten Softwarelösung für Busy Waiting/Spin Locking?

Synchronisationsvariable:

$$\texttt{flag} = \begin{cases} 0 & \text{kein Prozeß ist im kritischen Bereich} \\ 1 & \text{kritischer Bereich ist belegt} \end{cases}$$

```
flag = 0                    /* Initialwert */
⋮
while (flag != 0);          /* aktives Warten */
flag = 1;
critical_region();
flag = 0;
⋮
```

K.n.i.F.!!!

Viele (moderne) CPU-Architekturen unterstützen atomare, d.h. nicht-unterbrechbare Instruktionen.
Beispiel: compare-and-exchange (CMPXCHG)

Die **atomare** Instruktion `cmpxchg` testet auf den Wert `_old`:

```
cmpxchg (lock, _new, _old) {
   if *lock == _old then
      *lock := _new
      return TRUE
   else
      return FALSE
}
```

**Beispiel:**

```
cmpxchg (lock, 1, 0)
```

Falls lock==0: schreibe 1 an die Stelle von lock und liefere TRUE zurück!

# Wozu kann man `cmpxchg` benutzen?

- `cmpxchg` ändert Werte im Speicher atomar
- `cmpxchg` ist ein nicht-priviligiertes Kommando, das auch im User-Mode ausgeführt werden kann.

$\Longrightarrow$ Kann benutzt werden, um Locks zu implementieren!
(vorausgesetzt man weiß, wie man CPU-Instruktionen aus C/C++-Anwendungen heraus benutzen kann)

# Synchronisation mittels Spin Locking mit CMPXCHG

Synchronisationsvariable:

$$\texttt{lockflag} = \begin{cases} 0 & \text{kein Prozeß ist im kritischen Bereich} \\ 1 & \text{kritischer Bereich ist belegt} \end{cases}$$

```
lock(lockflag);    /* aktives Warten */
critical_region();
unlock(lockflag);
⋮
```

$\Longrightarrow$ `lock()` muß auf den Wert 0 testen!

```
typedef unsigned long lock_t;

void lock(lock_t * lock) {
  lock_t _old   = 0;
  lock_t _new = 1;
  do {
     /* retry until lock is set to 0
      * and changing to 1 succeeds
      */
  } while ( ! cmpxchg (lock, _new, _old) );
}

void unlock(lock_t * lock) {
  *lock = 0;
}
```

Wo steckt das Busy Waiting?

siehe: Stallings, 5.2 Mutual Exclusion: Hardware Support

## Ab jetzt betrachten wir Lösungen

1. für alle drei Koordinationsprobleme!
   1. Kritischer Bereich
   2. Producer-Consumer:
      wechselseitiger Ausschluß + Ablaufsteuerung für vollen bzw. leeren Puffer.
   3. Readers & Writers:
      nebenläufige Leseprozesse sind erlaubt
2. die Busy Waiting vermeiden!

$\Longrightarrow$ Semaphore, Monitore, Nachrichtensystem

1965 Dijkstra: Benutzen eines speziellen Variablentyps `semaphor`:

Definition 21:

> Ein Semaphor ist eine Variable, auf die nur mit den nicht-unterbrechbaren (atomaren) Operationen `DOWN` und `UP` zugegriffen werden kann.

Im Original: P(S) „passeren", V(S) „vrygeven"
($\hat{=}$ Hard–/Softwarelösung)

binäre Semaphore für wechselseitigen Ausschluß:

$$\text{Semaphor } S \begin{cases} \leq 0 & \text{Prozeß darf den kritischen Bereich nicht betreten} \\ = 1 & \text{Prozeß darf den kritischen Bereich betreten} \end{cases}$$

## Wechselseitiger Ausschluß:

```
    ⋮
DOWN(S);
critical_section();
UP(S);
    ⋮
```

## Ablauf:

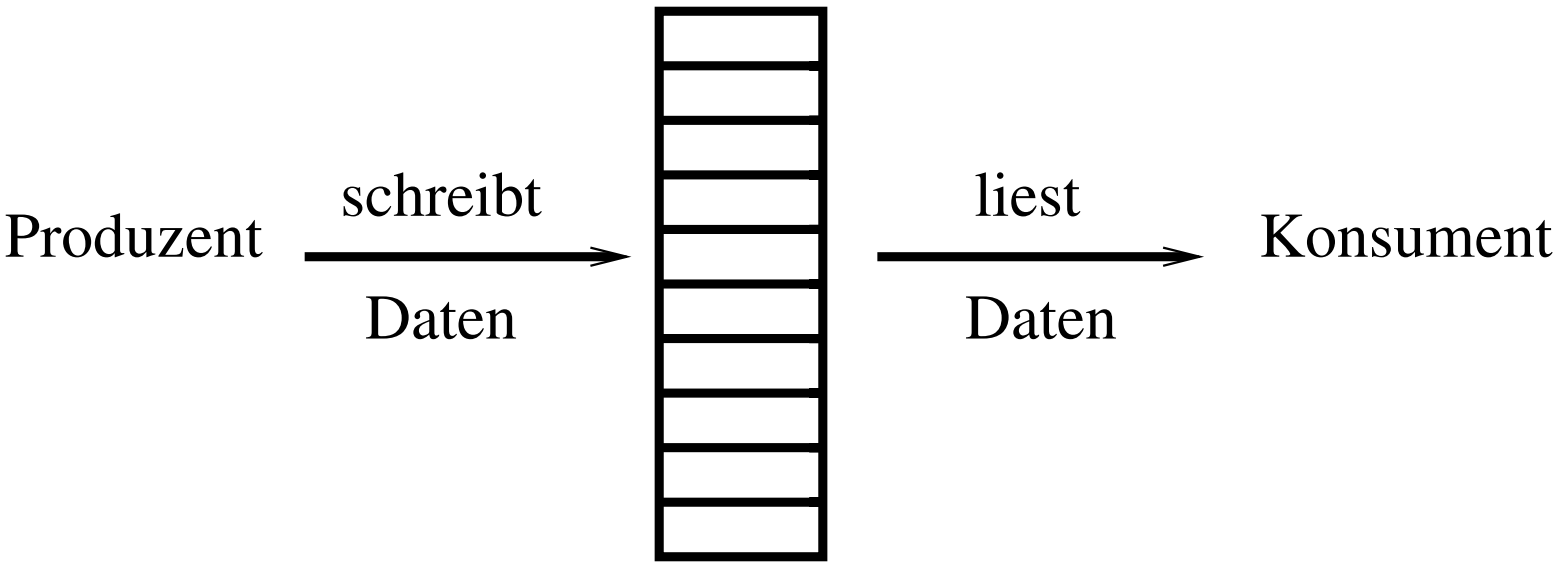
1. Prozeß A will den kritischen Bereich betreten.
2. Prozeß A fragt den Semaphorwert ab mit `DOWN(S)`.
3. Falls $S = 0$ erfolgt eine synchrone Unterbrechung, d.h., A wartet im Zustand „blockiert", bis der kritische Bereich frei ist. Andernfalls betritt A den kritischen Bereich, d.h. A rechnet weiter.
4. Nach Verlassen des kritischen Bereichs wird der kritische Bereich mittels `UP(S)` wieder freigegeben.

```
DOWN(S)
    { S = S - 1;
     if ( S < 0 )
       then { der aufrufende Prozeß wird blockiert
       und in eine Warteschlange für S eingetragen }
    }


UP(S)
{ S = S + 1;
    if (Warteschlange von S nicht leer)
    erster Prozeß in der WS wird geweckt; }
```

Vorteile:

+ wechselseitiger Ausschluß,
+ `UP`, `DOWN` sind kurze Operationen, für die Nichtunterbrechbarkeit gerechtfertigt ist,
+ kein aktives Warten,
+ ermöglicht nicht nur wechselseitigen Ausschluß, sondern ist auch zur Synchronisation von Prozeßabläufen geeignet (Prozeßkoordination).

# 1. Beispiel: Producer – Consumer – Problem




$n$ Pufferplätze

**1. Problem:** $n$ Pufferplätze, die zyklisch belegt werden
Producer und Consumer dürfen sich nicht überholen!

**2. Problem:** $k \geq 1$ Producer, $l \geq 1$ Consumer

$\Longrightarrow$ Komplexeres Problem als nur der Schutz eines kritischen Bereichs!

# 2. Beispiel: Readers & Writers

Konkurrierender Zugriff auf Datei

Synchronisationsregeln:

1. Beliebig viele Prozesse dürfen nebenläufig lesend auf die Datei zugreifen.
2. Schreiben findet exklusiv statt.
3. Falls ein Schreibwunsch eintrifft, wird kein neuer Lesezugriff zugelassen.

Semaphorlösung von Courtois, Heymans, Parnas (1971)
Korrektur von Habermann

Reader

```
⋮
start_read();
read(Daten);
end_read();
⋮
```

Writer

```
⋮
start_write();
write(Daten);
end_write();
⋮
```

**gemeinsame Variablen:**

```
readcount = 0; /* Zähler für Lesewünsche */
writecount = 0; /* Zähler für Schreibwünsche */
```

Was ist zu überprüfen?

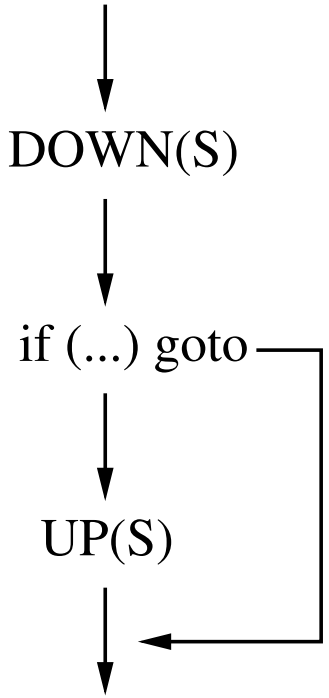
1. Gelten die Synchronisationsregeln?
2. Verklemmungsfrei?

Initialisierung gemeinsamer Variablen:

```
readcount = 0; /* Zaehler fuer Lesewuensche */
writecount = 0; /* Zaehler fuer Schreibwuensche */

mutex_rc = 1; /* binaeres Semaphor fuer exklusiven
                 Zugriff auf readcount */
mutex_wc = 1; /* binaeres Semaphor fuer exklusiven
                 Zugriff auf writecount */

sem_w = 1; /* binaeres Semaphor fuer exklusives
              Schreiben */
sem_r = 1; /* falls r=1 dann Lesen erlaubt,
              falls r=0 dann Lesen nicht erlaubt
              (d. h. Schreibwunsch liegt vor) */

mutex_str = 1; /* binaeres Semaphor fuer exklusiven
Zugriff auf start_read */
```

## Nachteile von Semaphoren:

- Ablaufsteuerung ist mit Semaphoren nicht intuitiv, sondern kompliziert
- unübersichtliche Programmierung $\Longrightarrow$ Gefahr von fehlerhaftem Code mit Deadlock-Potential
  ebenfalls fehleranfällig: `goto`–Anweisungen:




- Vor.: Prozesse haben gemeinsamen Arbeitsspeicher für Semaphorrealisierung (in verteilten Systemen nicht gegeben!).

Hoare 1974
hier: Version von Brinch Hansen, 1975

Definition 22:

Ein Monitor ist ein Programmiersprachenkonstrukt, in dem alle Prozeduren und Datenstrukturen, die den Zutritt zum kritischen Bereich kontrollieren, gesammelt werden mit der Eigenschaft, daß nur ein Prozeß zu einem Zeitpunkt in einem Monitor aktiv sein kann.

($\hat{=}$ Raum mit nur einem Schlüssel!)

„Bedingungsvariable" (condition) mit Operationen `WAIT` und `SIGNAL` zur Koordination/Ablaufsteuerung:

1. Kann eine Monitorfunktion in der Abarbeitung nicht fortfahren, ruft sie `WAIT(<condition name>)` auf, wodurch der aufrufende Prozeß blockiert wird, d.h., der Prozeß ist nicht mehr im Monitor aktiv.
   Innerhalb eines Monitors können Prozesse auf unterschiedliche Ereignisse/Bedingungen warten.
2. Aufwecken von Prozessen:
   `SIGNAL(<condition name>):` Falls ein Prozeß auf `<condition name>` wartet, wird dieser geweckt. Die Warteschlange wird gemäß FIFO verwaltet.
   `SIGNAL`-Semantik: Es muß garantiert sein, daß der geweckte Prozeß als nächster die Kontrolle über den Monitor erhält.
3. `WAIT` und `SIGNAL` dürfen nur innerhalb des Monitors benutzt werden.

Hoare: Der `SIGNAL` aufrufende Prozeß wird blockiert und erst geweckt, wenn der fortgesetzte Prozeß den Monitor verläßt.

Brinch Hansen: `SIGNAL` darf nur als letzte Anweisung in einer Monitor–Funktion stehen!!!

**Bem.:** Die Bedingungsvariablen sind keine Zähler! Sie summieren **nicht** die Signal-Aufrufe. Im Gegensatz dazu summieren Semaphore in dem zugeordneten Semaphorzähler die Anzahl der UP-Aufrufe und subtrahieren die Anzahl der Down-Aufrufe.

# Monitorlösung für Producer-Consumer

```
monitor   producer_consumer
{  condition full, empty;
   integer count;   /* Anzahl belegter Pufferplätze */


   enter(item)
   { if (count == N)  WAIT(full);
      write_item(item);
      count = count + 1;
      if (count == 1)  SIGNAL(empty);
   }
```

```
    remove(item)
    { if (count == 0)  WAIT(empty);
       remove_item(item);
       count = count - 1;
       if (count == N-1)  SIGNAL(full);
    }
}
```

Wer hat's erfunden?
C.A.R. Hoare: *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, No. 10, 1974, pp. 549–557.
enthält eine Danksagung:

*The development of the monitor concept is due to frequent discussions and communications with E. W. Dijkstra and P. Brinch-Hansen. A monitor corresponds to the "secretary" described in [9], and is also described in [1,3].*

[9] von Dijkstra, 1972
[1, 3] von Brinch-Hansen, 1972 und 1973

# Implementation von Monitoren

Spezieller Compiler, der Bibliotheksfunktionen einsetzt, falls

- Monitor betreten wird
- Monitor verlassen wird
- wait–Aufruf
- signal–Aufruf

siehe auch: C.A.R. Hoare: *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, No. 10, 1974, pp. 549–557.

Vorteil von Monitoren: Monitore erzwingen übersichtlichen Programmierstil, bei dem „Synchronisationscode" vom restlichen Programmtext getrennt wird.

Monitore werden zur Synchronisation benutzt z.B. in

- Concurrent Pascal (Brinch Hansen, 1975),
- Concurrent Euclid,
- Mesa, Modula-3 und Java implementieren eine **Monitor-Variante**, die von Lampson und Redell 1980 vorgeschlagen wurde, die einfacher zu implementieren ist.

# Thread-Synchronisation in Java: Variante von Hoare's Monitorkonzept

- Methoden werden mit Schlüsselwort `synchronized` versehen. Wird eine `synchronized`-Methode des Objekts ausgeführt, kann kein anderer Thread eine `synchronized`-Methode dieses Objekts ausführen.

## Beispiel:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

1. Wenn ein Thread eine *synchronisierte* Methode eines Objekts ausführt, dann werden alle nebenläufigen Aufrufe einer synchronisierten Methode desselben Objekts blockiert.
2. Nachdem ein Thread eine synchronisierte Methode verläßt, werden alle Zustandsänderungen des Objekts für alle anderen Threads sichtbar.

# Thread-Synchronisation in Java: Variante von Hoare's Monitorkonzept

- Methoden werden mit Schlüsselwort `synchronized` versehen. Wird eine `synchronized`-Methode des Objekts ausgeführt, kann kein anderer Thread eine `synchronized`-Methode dieses Objekts ausführen.
- `wait()`: Stellt den aufrufenden Thread in die „Standby"-Warteschlange des Monitors. Der Monitor wird automatisch freigegeben.
  PRO MONITOR WIRD NUR EINE STANDBY-WARTESCHLANGE FÜR BEDINGUNGEN BENUTZT!!!

- `notifyAll()`-Methode weckt alle Threads, die in der „Standby"-Warteschlange warten. Diese stellen sich dann wieder vor dem Monitor in der Monitor-Warteschlange an. Falls mehrere Threads in der Warteschlange warten, ist im Java-Laufzeitsystem nicht festgelegt, welcher Thread als nächster den Monitor erhält.
- `notify()`: Wird nicht empfohlen: Ein beliebiger der wartenden Threads wird geweckt.

# CubbyHole-Beispiel aus Sun-Tutorial

```
class ProducerConsumerTest {
  public static void main(String[] args) {
    CubbyHole c = new CubbyHole();
    Producer p1 = new Producer(c, 1);
    Consumer c1 = new Consumer(c, 1);

    p1.start();
    c1.start();
  }
}
```

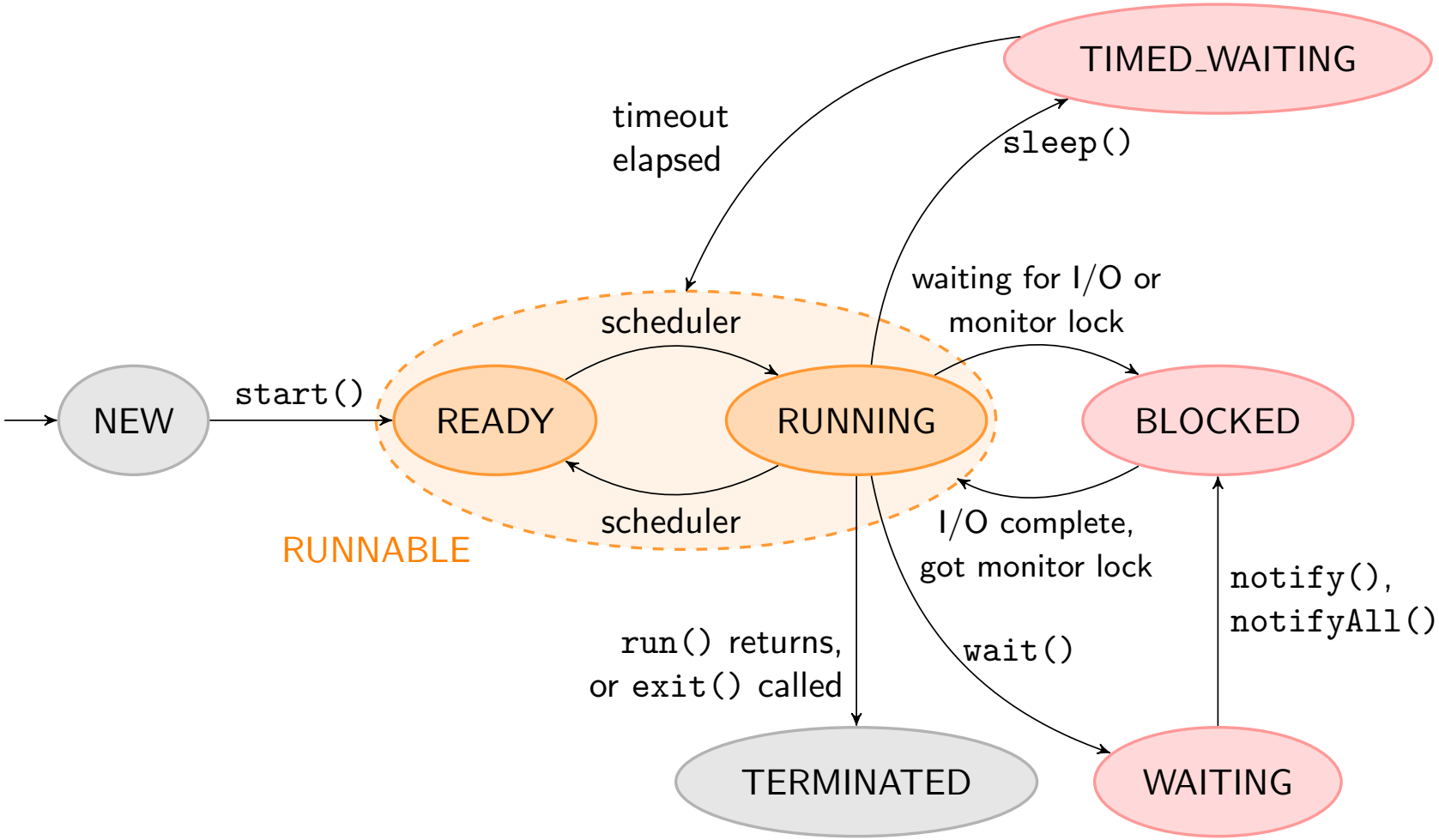
```
class CubbyHole {
private int contents; //this is the condition variable.
private boolean available = false;

public synchronized void put(int value) {
  while (available == true) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }
  contents = value;
  available = true;
  notifyAll();
}
```

```
class CubbyHole {
private int contents; //this is the BUFFER!!!
private boolean available = false;

public synchronized void put(int value) {
  while (available == true) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }
  contents = value;
  available = true;
  notifyAll();
}
```

```
public synchronized int get() {
  while (available == false) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }
  available = false;
  notifyAll();
  return contents;
}
}
```

```
class Producer extends Thread {
  private CubbyHole cubbyhole;
  private int number;
  public Producer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      cubbyhole.put(i);
      System.out.println("Producer␣#" + this.number +
      "␣put:␣" + i);
      try {
        sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) { }
    }
  }
}
```

```
class Consumer extends Thread {
  private CubbyHole cubbyhole;
  private int number;
  public Consumer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
  }

  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = cubbyhole.get();
      System.out.println("Consumer #" + this.number +
      " got: " + value);
    }
  }
}
```




Java-Threads-Zustandsdiagramm

Beispiel für Anwendung mit Livelock: Peter Welch: *Wot, no chickens?*

Die Pthread-Bibliothek unterstützt binäre Semaphore, die sogenannten Mutexe:

| Funktion | Beschreibung |
|---|---|
| pthread_mutex_init() | Erzeuge ein Mutex |
| pthread_mutex_lock() | Erlange die Sperre oder blockiere |
| pthread_mutex_trylock() | Erlange die Sperre oder erzeuge eine Fehlermeldung |
| pthread_mutex_unlock() | Gebe Sperre wieder frei |
| pthread_mutex_destroy() | Löschen/Aufräumen |

Vorsicht: pthread_mutex_trylock() verleitet zur Programmierung von Busy Waitung :-(

Wie erreicht man damit Ablaufsteuerung, wie sie z.B. für das Producer-Consumer-Beispiel notwendig ist?

Die Pthread-Bibliothek bietet auch Bedingungsvariable (Condition Variables) an:

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Man kann mehrere Bedingungsvariablen erzeugen!

Attribute für Bedingungsvariablen sind im POSIX-Standard vorgesehen, aber werden auf Linux nicht unterstützt.

**Bedingungsvariable:**

| Funktion | Beschreibung |
|---|---|
| pthread_cond_init() | Erzeuge eine Bedingungsvariable |
| pthread_cond_wait() | Blockiere, um auf das Eintreffen der Bedingung zu warten |
| pthread_cond_signal() | Sende Signal an einen auf die Bedingung wartenden Thread |
| pthread_cond_broadcast() | Wecke alle Threads, die auf die Bedingung warten |
| pthread_cond_destroy() | Löschen/Aufräumen |

Achtung: Wenn pthread_cond_signal() bzw. pthread_cond_broadcast() aufgerufen wird, aber **kein** Thread auf die Bedingung wartet, geht das Signal verloren!

**Bem.:** Da die Pthread-Bibliothek mehrere Bedingungsvariable ermöglicht, kann für jede Wartesituation eine eigene Bedingungsvariable definiert werden. Somit sollte pthread_cond_broadcast() in der Regel unnötig sein.

```
NAME
pthread_cond_wait - wait on a condition

SYNOPSIS
#include <pthread.h>

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

Wieso hat das pthread_cond_wait() zwei Argumente?

## Semantik von pthread_cond_wait():

```
DESCRIPTION

The pthread_cond_timedwait() and pthread_cond_wait()
functions shall block on a condition variable.

They shall be called with mutex locked (!) by the calling
thread or undefined behavior results.
...

Upon successful return, the mutex shall have been locked
and shall be owned by the calling thread. (!)
```

$\Longrightarrow$ Wie beim Monitor-Konzept dürfen `pthread_cond_wait()` und `pthread_cond_signal()` nur innerhalb eines geschützten Raums, d.h. innerhalb eines geschützten kritischen Bereichs benutzt werden.

## Condition Wait Semantics: Re-evaluate the predicate:

It is important to note that when pthread_cond_wait() and pthread_cond_timedwait() **return without error**, the associated predicate may still be false.

...

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

**It is thus recommended that a condition wait be enclosed in the equivalent of a "while loop" that checks the predicate.**

**Quelle:** POSIX Programmer's Manual

***An added benefit*** *of allowing spurious wakeups is that applications are* ***forced*** *to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, IEEE Std 1003.1-2001 explicitly documents that spurious wakeups may occur.*

Quelle: POSIX Programmer's Manual zu `pthread_cond_signal`

**Beispiel:** Adaptiert nach dem Producer-Consumer-Beispiel im Tanenbaum, 3. Auflage, Abb. 2.32, 2009, S. 180.

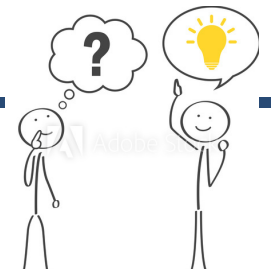
Entspricht dem CubbyHole-Beispiel: Es wird ein Producer und ein Consumer gestartet.
Der Producer schreibt nacheinander die Werte 1 bis MAX in den gemeinsamen Buffer. Der Consumer liest MAX-mal den Buffer aus, gibt den Wert aus und schreibt jeweils eine Null in den Buffer.

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000          /* Anzahl erzeugte Nummern */

pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* Bedingungsvariablen fuer */
                             /* Consumer und Producer */

int buffer = 0;              /* gemeinsamer Puffer zwischen */
                             /* Producer und Consumer*/

void *producer(void *arg)
{int i;

  for (i= 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex);   /* erlange exklusiven Zugriff
                                      /* auf Puffer */
    while (buffer !=0) pthread_cond_wait(&condp, &the_mutex);
    buffer = i;                       /* schreibe Element in Puffer
    pthread_cond_signal(&condc);      /* wecke ggf. Verbraucher auf
    pthread_mutex_unlock(&the_mutex); /* gib Zugriff auf Puffer fr
  }
  pthread_exit(0);
}
```

```
void *consumer(void *number)
{int i;

  for (i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex);   /* erlange exklusiven Zugriff
                                      /* auf Puffer */
    while (buffer ==0) pthread_cond_wait(&condc, &the_mutex);
    printf ("consumer␣%d␣gets␣%d␣\n", (int) number, buffer);
    buffer = 0;                       /* Puffer "leeren"*/
    pthread_cond_signal(&condp);      /* wecke Erzeuger ggf. auf */
    pthread_mutex_unlock(&the_mutex); /* gib Zugriff auf Puffer fr
  }
  pthread_exit(0);
}
```

```
int main(int argc, char **argv)
{
  pthread_t pro, con;
  pthread_mutex_init(&the_mutex, 0);
  pthread_cond_init(&condc, 0);
  pthread_cond_init(&condp, 0);
  pthread_create(&pro, 0, producer, (void *)0);
  pthread_create(&con, 0, consumer, 0);
  pthread_join(pro, 0);
  pthread_join(con, 0);
  pthread_cond_destroy(&condc);
  pthread_cond_destroy(&condp);
  pthread_mutex_destroy(&the_mutex);
}
```

# Was Sie jetzt wissen sollten ...




**Bemerkung:**

- Die im POSIX-Standard spezifizierten Synchronisationsprimitiven sind eine Mischung aus Mutex und Bare-Metal-Monitor-Konzept: Es gibt Mutexe, Bedingungsvariablen, WAIT und SIGNAL.
  Der Monitorraum muss aber durch ein explizites Mutex geschützt werden!
- Man kann auf verschiedene Bedingungen warten (anders als bei Java).
- Der POSIX-Standard empfiehlt: Re-Evaluate the predicate!
- Der POSIX-Standard hat Schwächen:

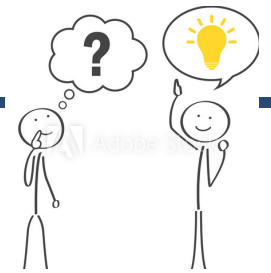
`>man pthread_cond_wait`:

- When a thread waits on a condition variable, having specified a particular mutex to either the `pthread_cond_timedwait()` or the `pthread_cond_wait()` operation, a dynamic binding is formed between that mutex and condition variable that remains in effect as long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined.

> `man pthread_cond_broadcast`:

- The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).
- The `pthread_cond_broadcast()` or `pthread_cond_signal()` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_wait()` or `pthread_cond_timedwait()` have associated with the condition variable during their waits; **however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling** `pthread_cond_broadcast()` **or** `pthread_cond_signal()`.
- ...

# Was Sie jetzt wissen sollten ...




**Bemerkung:**

- Die im POSIX-Standard spezifizierten Synchronisationsprimitiven sind eine Mischung aus Mutex und Bare-Metal-Monitor-Konzept: Es gibt Mutexe, Bedingungsvariablen, WAIT und SIGNAL.
  Der Monitorraum muss aber durch ein explizites Mutex geschützt werden!
- Man kann auf verschiedene Bedingungen warten (anders als bei Java).
- Der POSIX-Standard empfiehlt: Re-Evaluate the predicate!
- Der POSIX-Standard hat Schwächen:
  `wait` und `signal` *sollten (besser: müssen)* immer aus einem geschützten Bereich heraus aufgerufen werden.
- Die jeweilige Implementierung des POSIX-Standards muß nicht POSIX-konform sein – und das kann gut sein!

# Read-Write-Locks

- pthread_rwlock_init() - initialize a read-write-lock
- pthread_rwlock_rdlock() - get a shared read lock
- pthread_rwlock_wrlock() - get an exclusive write lock
- pthread_rwlock_destroy() - destroy a read-write-lock
- ...

**Quelle:** Peter S. Pacheco: *An Introduction to Parallel Programming*, 2011.
Beispiel aus Pacheco: Verwaltung einer Liste: *member()*, *insert()*, *delete()*

Source Code – Errata beachten!

Nachricht (message): Information, die von einem Prozeß zu einem anderen Prozeß übergeben wird.

Mailbox: Pufferplatz für Nachrichten, die abgeschickt, aber noch nicht empfangen wurden.

`send` (Mailbox-Adresse, Nachricht): Kopiert Nachricht in die zur Zieladresse zugehörige Mailbox. Falls die Mailbox voll ist, wird blockiert gewartet, bis Platz in der Mailbox ist.

`receive` (Mailbox-Adresse, Nachricht): Kopiert die Nachricht aus der Mailbox in den Adreßraum des Empfängers, löscht die Nachricht in der Mailbox. Falls die Mailbox leer ist, wird blockiert gewartet, bis eine Nachricht eintrifft.

# Mailbox Producer–Consumer

## Producer

```
while (TRUE)
{ item=produce_item();
   send(‘‘Producer_Consumer’’, item);
}
```

## Consumer

```
while (TRUE)
{ receive(‘‘Producer_Consumer’’, &item);
   consume_item(item);
}
```

# Mailbox–Implementation

| Zähler `empty` für freie Nachrichtenplätze |
|---|
| Zähler `full` für belegte Nachrichtenplätze |
| Warteschlange mit Prozessen, die auf Senden warten |
| Warteschlange mit Prozessen, die auf Empfangen warten |
| Nachricht 1 |
| ⋮ |
| Nachricht N |

Mailbox-Adressierung ermöglicht:
- 1-1-Kommunikation
- n-1-Kommunikation
- 1-n-Kommunikation (Broadcast)

**Beispielimplementierungen:**

1. UNIX lokal: `msgsnd, msgrcv`
2. blockierende UNIX-Sockets

```
NAME
       msgrcv, msgsnd - System V message queue operations

SYNOPSIS
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>

  int msgsnd(int msqid, const void *msgp, size_t msgsz,
                                                  int msgflg);

  ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
                                       long msgtyp, int msgflg);
```

If insufficient space is available in the queue, then the default behavior of msgsnd() is to block until space becomes available. If IPC_NOWAIT is specified in msgflg, then the call instead fails with the error EAGAIN.

Vorteile:

- übersichtlicher als Semaphore,
- „einfache" Realisierung als Systemaufrufe,
- getrennter Adreßraum von Sender und Empfänger $\Rightarrow$ weniger fehleranfällig,
- auch zur Kommunikation bzw. Synchronisation nicht-lokaler Prozesse geeignet.

Theorem 3: Semaphore, Monitore und Nachrichtensysteme bieten die gleiche Mächtigkeit zur Prozeßsynchronisation, d.h., alle Synchronisationsprobleme, die sich mit einem der 3 Modelle lösen lassen, sind auch mit den beiden anderen Modellen lösbar.