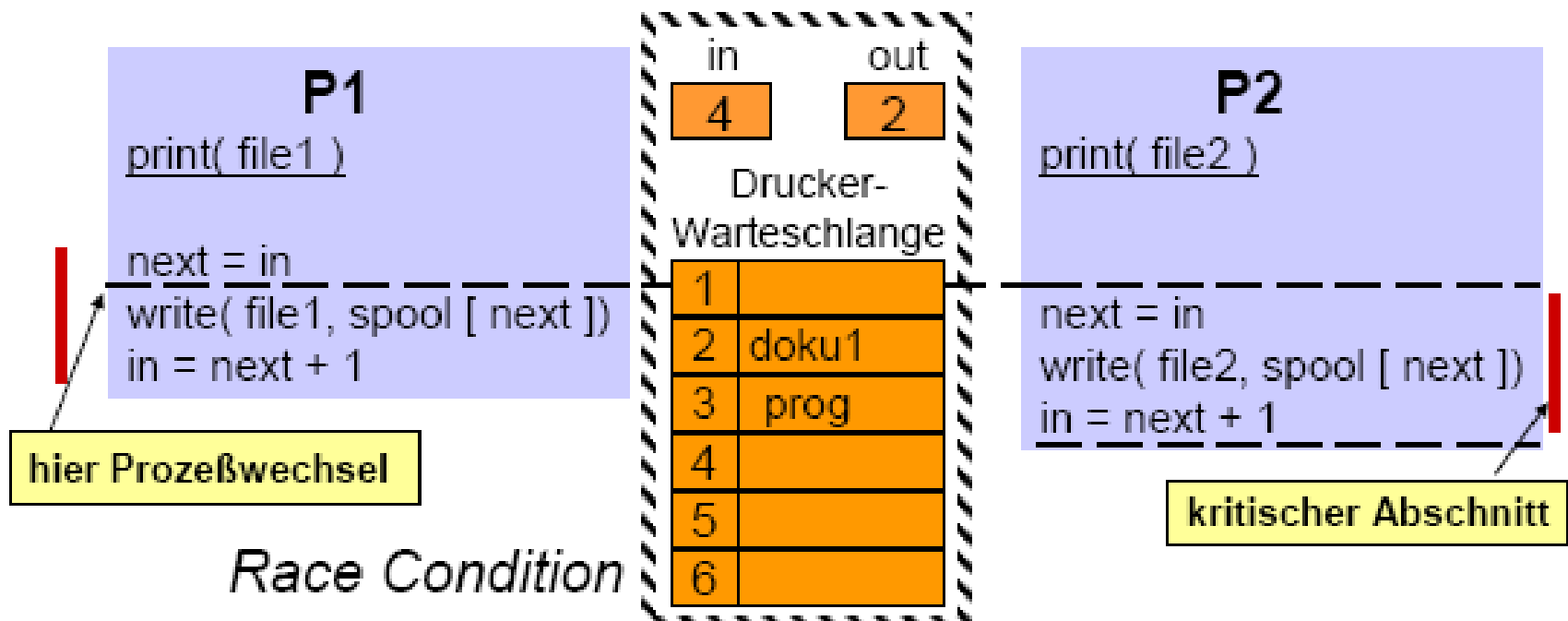


Betriebssysteme

Inter-Prozess-Kommunikation

Prof. Dr.-Ing. Torben Weis
Universität Duisburg-Essen

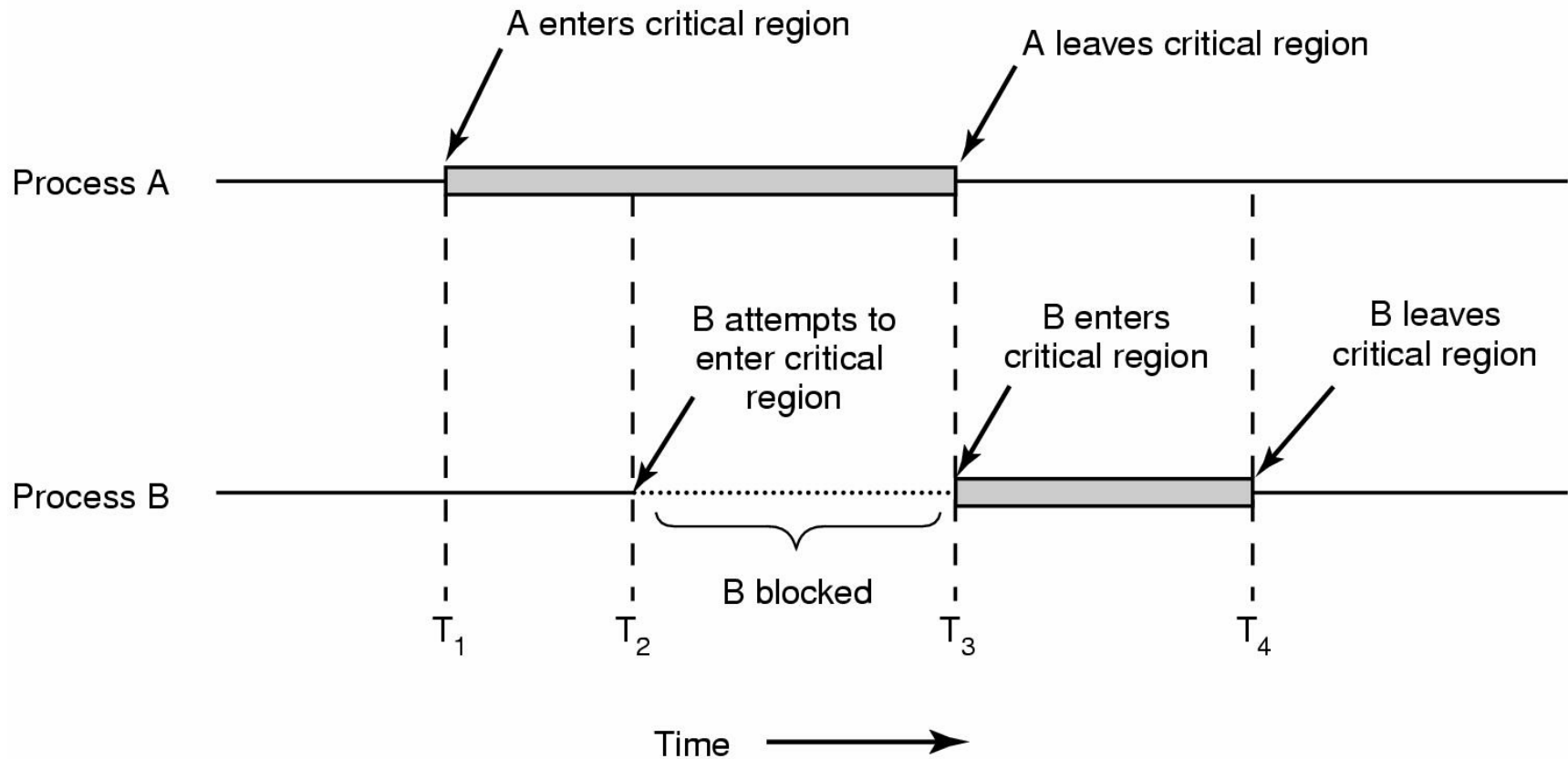
Kritischer Abschnitt



Kritischer Abschnitt

- Wechselseitiger Ausschluß (mutual exclusion)
 - Lokal: Prozesse brauchen Zugriff auf gemeinsamen Speicher für kritischen Abschnitt
 - Verteilt: Verschicken geeigneter Nachrichten
 - Wir betrachten nur den lokalen Fall
- Anforderungen an eine Lösung
 1. Höchstens 1 Prozess im KA
 2. Für beliebige Anzahl von Prozessen
 3. Kein Blockieren durch Prozesse außerhalb eines KA
 4. Jeder Prozess muss (irgendwann einmal) in KA eintreten

Gegenseitiger Ausschluss



Busy Waiting (Fehlversuch Nr. 1)

```
int turn = 0;
```

```
void enter_region( int process )
```

```
{
```

```
    while (turn != 0) { }    // KA ist nicht frei  
                            // warten ("busy waiting")  
    turn = process;        // KA betreten und blocken
```

```
}
```

```
void leave_region(int process)
```

```
{
```

```
    turn = 0;
```

```
}
```

Busy Waiting (Fehlversuch Nr. 2)

```
bool interested[] = {false, false};
```

```
void enter_region( int process )  
{  
    int other = 1 - process;  
    interested[process] = true;  
    while ( interested[other] == true) { }  
}
```

```
void leave_region(int process)  
{  
    interested[process] = false;  
}
```

Busy Waiting a la Peterson

```
#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                /* whose turn is it? */
int interested[N];      /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;           /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;        /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Streng alternierendes Warten

```
while (TRUE) {  
    while (turn != 0)    /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)    /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

- Verletzt Bedingung Nummer 3
„Kein Blockieren von Prozessen außerhalb eines KA“

Busy Waiting mit Test_And_Set_Lock

enter_region:

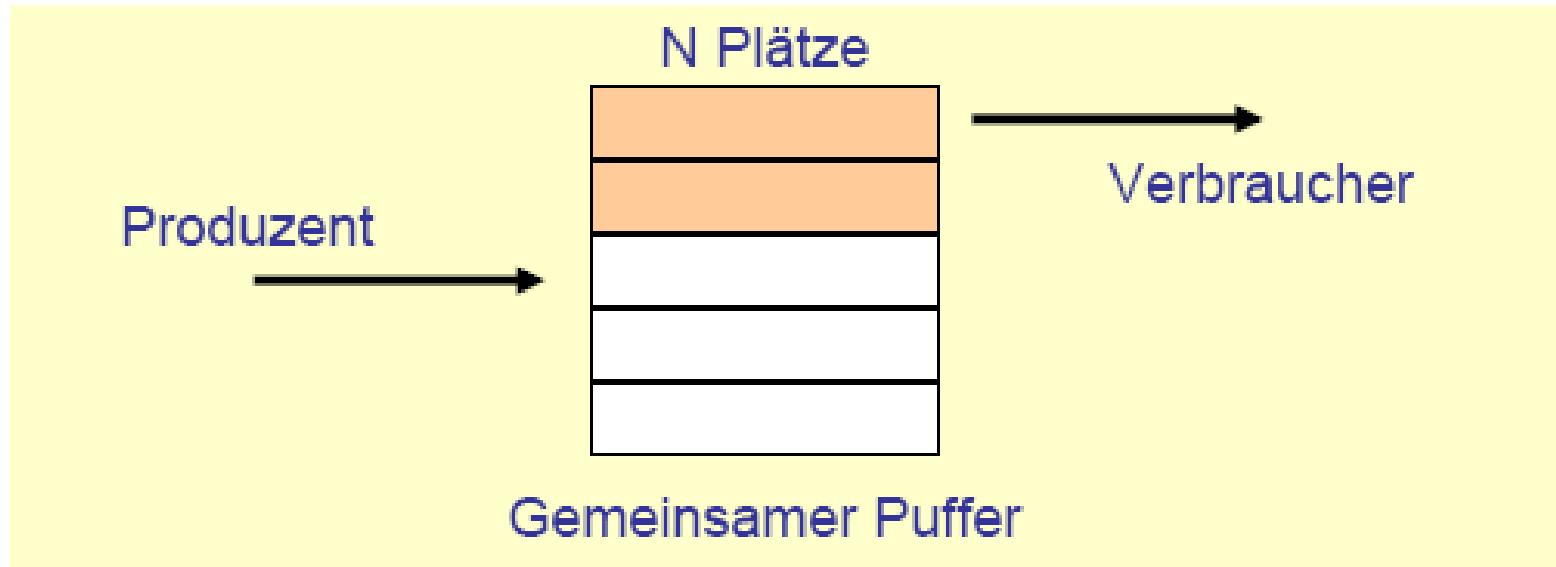
```
TSL REGISTER,LOCK      | copy lock to register and set lock to 1
CMP REGISTER,#0        | was lock zero?
JNE enter_region       | if it was non zero, lock was set, so loop
RET | return to caller; critical region entered
```

leave_region:

```
MOVE LOCK,#0          | store a 0 in lock
RET | return to caller
```

- Vorteile gegenüber Peterson
 - Weniger CPU Zyklen
 - Weniger Speicherverbrauch
- Nachteil
 - Benötigt Hardwareunterstützung

Erzeuger/Verbraucher Problem



- Voller Puffer blockiert den Produzenten
- Leerer Puffer blockiert den Verbraucher
- Alternative Bezeichnungen
 - Producer/Consumer
 - Bounded Buffer

Erzeuger/Verbraucher Problem

Erzeuger E

anzahl = 0 // max. N

Verbraucher V

```
WHILE true DO
  produziere( stück )
  IF anzahl = N THEN sleep( )
  einfügen( stück )
  anzahl = anzahl + 1
  IF anzahl = 1 THEN wakeup(V)
END
```

```
WHILE true DO
  IF anzahl = 0
    THEN sleep( )
  entnehmen( stück )
  anzahl = anzahl - 1
  IF anzahl = N-1 THEN wakeup(E)
  verbrauche( stück )
END
```

hier Prozeßwechsel

- Ohne Synchronisierung geht es nicht
- Markieren des Schleifenrumpfes als kritischer Abschnitt führt auch nicht zum Ziel
- Semaphoren bieten eine geeignete Lösung

Semaphoren

- Eine Semaphore ist
 - Eine Integer Variable
 - Mit einer Prozesswarteschlange
 - Mit 2 unteilbaren Funktionen: up and down
 - Manchmal auch P und V genannt

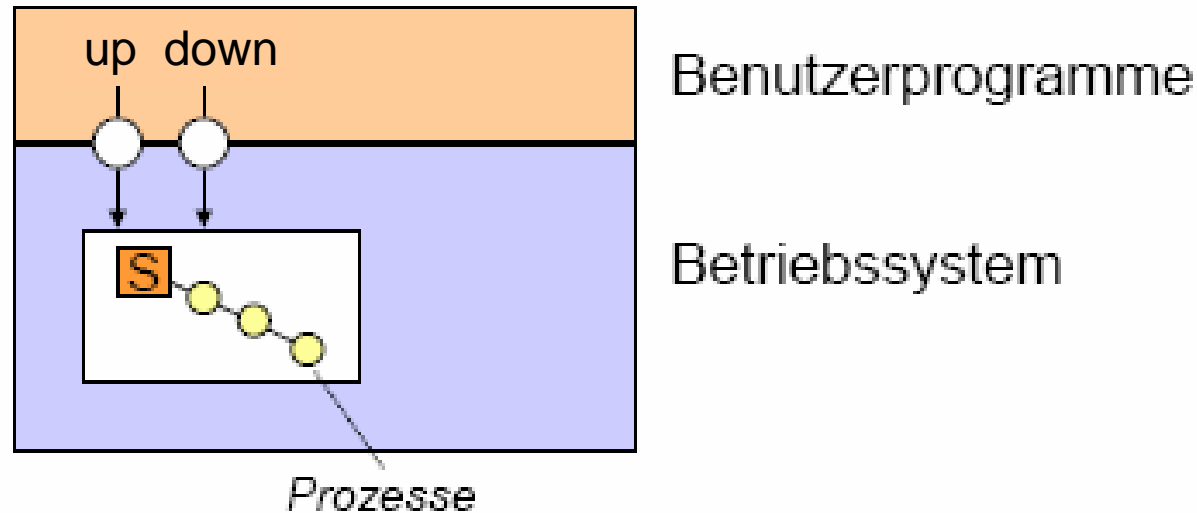
down(s)

```
if ( S > 0 ) then S = S - 1  
else warten_auf_wecksignal
```

up(s)

```
if ( wartender_prozess) then aufwecken  
else S = S + 1
```

Implementierung von Semaphoren



- Semaphoren werden vom BS bereitgestellt
- Ausführung von `up()` und `down()` in kritischem Abschnitt
 - Sonst wären die Funktionen nicht unteilbar
- Warten und Aufwecken über den Zustand im PCB
 - `down` -> Zustand „wartend“
 - `up` -> Zustand „bereit“

Binäre Semaphoren

- Eine binäre Semaphore ist ein Mutex
- Beispiel

```
int semaphore = 1;
void foobar()
{
    Down( semaphore );
    KA
    Up( semaphore );
}
```

Erzeuger/Verbraucher Problem (mit Semaphoren)

```
Semaphor mutex = 1 // binäre Semaphore  
Semaphor leer = N // Anzahl der freien Plätze  
Semaphor belegt = 0 // Anzahl der belegten Plätze
```

Erzeuger

```
WHILE true DO  
  produziere( stück )  
  P( leer )  
  P( mutex )  
  einfügen( stück )  
  V( mutex )  
  V( belegt )  
END
```

Was passiert
beim Ver-
tauschen?

```
WHILE true DO  
  P( belegt )  
  P( mutex )  
  entnehmen( stück )  
  V( mutex )  
  V( leer )  
  verbrauche( stück )  
END
```

Verbraucher

Monitore

- Programmierung mit Semaphoren ist schwierig, ein Fehler kann fatale Auswirkungen haben
- Hoare (1974) und Brinch-Hansen (1975) schlugen "Monitor-Konzept" vor
- Ein Monitor ist:
 - Konstrukt einer Programmiersprache
 - Menge von Prozeduren und Daten, die in einem speziellen Modul (= Monitor) gekapselt sind
 - Prozesse rufen Prozeduren im Monitor auf
 - Nur höchstens ein Prozess kann in einem Monitor aktiv sein (wechselseitiger Ausschluß)

Monitore

- Was geschieht, wenn ein Prozess im Monitor sich mit einem anderen Prozess synchronisieren muss?
(Beispiel: Puffer ist voll)
 - Signalisierung über WAIT() und SIGNAL() an Bedingungsvariable (keine Zähler !)
- Beispiel
 - WAIT(leer) // Prozess blockiert an Bedingung "leer"
 - SIGNAL(leer) // Prozess weckt einen beliebigen an
- SIGNAL(): Verletzung der Monitor-Bedingung ?
 - Hoare:
„aufgeweckter Prozess läuft, aufweckender blockiert sofort“
 - Brinch-Hansen:
"SIGNAL() muss letzte Anweisung im Monitor sein"

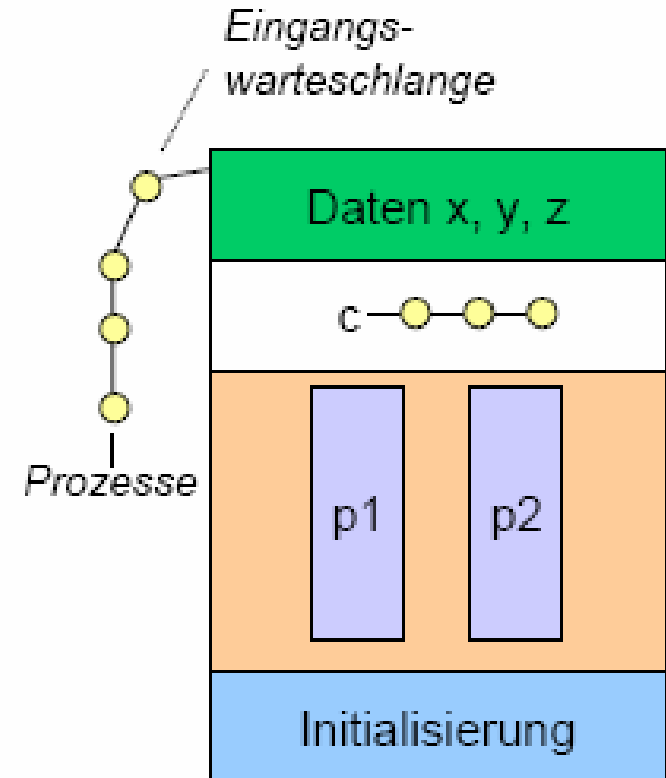
Monitore

```
MONITOR bb
BEGIN
  x, y, z          // lokale Daten
  CONDITION c     // Bedingungsvariablen

  PROCEDURE p1 ( ... )
  ....           // Anweisungen
  END

  PROCEDURE p2 ( ... )
  ....           // Anweisungen
  END

  DO .... END    // Initialisierung
END
```



Erzeuger/Verbraucher mit Monitoren

```
MONITOR prodCon
BEGIN
  INTEGER anzahl
  CONDITION belegt, leer

  PROCEDURE einfügen( s )
  IF anzahl = N THEN WAIT( belegt )
  puffer = puffer + s // in Puffer
  anzahl = anzahl + 1
  IF anzahl = 1 THEN SIGNAL( leer )
  END
```

Erzeuger

```
WHILE true DO
  produziere( stück )
  prodCon.einfügen( stück )
  END
```

```
PROCEDURE entnehmen( s )
  IF anzahl = 0 THEN WAIT( leer )
  puffer = puffer - s // aus Puffer
  anzahl = anzahl - 1
  IF anzahl = N-1 THEN SIGNAL( belegt )
  END

  anzahl = 0 // Initialisierung

  END // Ende des Monitors
```

Verbraucher

```
WHILE true DO
  prodCon.entnehmen( stück )
  verbrauche( stück )
  END
```

Implementierung von Monitoren

- Implementierung eines Monitors mit Semaphoren
- Jeder Monitor hat:
 - SEMAPHORE mutex = 1 // regelt exklusiven Zugriff
 - SEMAPHORE next = 0 // für Prozess, der SIGNAL() sagt
// und deshalb blockieren muß
 - INTEGER next-anz = 0 // zählt Prozesse an "next"
- Eine Prozedur Proc() des Monitors wird erweitert zu:

```
down( mutex ) // exklusiver Zugriff
Proc( ) // eigentliche Prozedur
IF next-anz > 0 THEN up( next ) ELSE up( mutex )
```

Implementierung von Monitoren

- Jede CONDITION Variable c wird realisiert durch
semaphore $c\text{-sem} = 0$; // Noch wartet keiner
int $c\text{-anz} = 0$; // Anzahl wartender Prozesse

SIGNAL(c)

```
{  
  IF  $c\text{-anz} > 0$  THEN  
  {  
     $\text{next-anz} = \text{next-anz} + 1$   
    // einen an  $c$  aufwecken  
    up(  $c\text{-sem}$  )  
    down( next )  
     $\text{next-anz} = \text{next-anz} - 1$   
  }  
}
```

WAIT(c)

```
{  
   $c\text{-anz} = c\text{-anz} + 1$   
  // anderen aufwecken  
  IF  $\text{next-anz} > 0$   
  THEN up( next )  
  ELSE up( mutex )  
  down(  $c\text{-sem}$  )  
   $c\text{-anz} = c\text{-anz} - 1$   
}
```

Pfadausdrücke

- Konstrukt einer Programmiersprache
 - In dieser Hinsicht ähnlich wie Monitor, aber kein automatischer und vollständiger wechselseitiger Ausschluss
- Erlaubt genauere Steuerung der Reihenfolge und Parallelität der Prozesse
- Pfadausdrücke werden in *Objekten* verwendet, die Operationen und Daten kapseln
- Zu Beginn des Objektes können Beschränkungen für die Ausführung der Operationen mittels Pfadausdrücken spezifiziert werden, d.h.

```
path Pfadausdruck end
```

Pfadausdrücke

- Notation
 - $P1$ und $P2$ sind Pfadausdrücke (oder Prozeduren)
- $P1$, $P2$ paralleles Arbeiten
 - *$P1$ und $P2$ können parallel ausgeführt werden*
- $P1$; $P2$ sequentielles Arbeiten
 - *$P2$ erst, wenn ein $P1$ beendet*
- $n : (P1)$ Einschränkung
 - *höchstens n aktive Inkarnationen von $P1$*
- $[P1]$ Mehrfache Inkarnationen
 - *beliebig viele Inkarnationen von $P1$*

Pfadausdrücke (Beispiele)

- `path P1 , P2 end`
- `path 1:(P1 , P2) end`
- `path P1 ; P2 end`
- `path 1:(P1 ; P2) end`
- `path 1:(P1) , P2 end`
- `path 1:(P1) , 1:(P2) end`
- `path 6:(5:(P1) , 4:(P2)) end`
- `path 3:(P1 ; P2) end`
- `path 1:([P1] , [P2]) end`
- `path 1:([P1] , P2) end`

Pfadausdrücke (Beispiele)

path P1 , P2 end	P1 und P2 können ohne Einschränkungen ausgeführt werden
path 1:(P1 , P2) end	P1 und P2 können in beliebiger Reihenfolge arbeiten, es darf aber nur je eine Inkarnation aktiv sein
path P1 ; P2 end	P1 und P2 müssen sequentiell ablaufen, d.h. #P2 angefangen < #P1 beendet
path 1:(P1 ; P2) end	Streng alternierend P2 nach P1, nur eine Prozedur-Inkarnation darf aktiv sein
path 1:(P1) , P2 end	es darf nur eine Inkarnation von P1 aktiv sein (d.h. Inkarnationen von P1 müssen streng sequentiell ausgef. werden), aber beliebig viele P2 parallel dazu
path 1:(P1) , 1:(P2) end	maximal ein P1 und ein P2 können parallel arbeiten
path 6:(5:(P1),4:(P2)) end	maximal 5 P1 und 4 P2 gleichzeitig aktiv, aber insgesamt nicht mehr als 6
path 3:(P1 ; P2) end	jedem Aufruf von P2 muss ein Aufruf von P1 vorangehen; insgesamt dürfen nur 3 Inkarnationen aktiv sein, d.h. P1 kann P2 höchstens 3 Aufrufe voraus sein
path 1:([P1] , [P2]) end	P1 und P2 arbeiten unter wechselseitigem Ausschluss; eine der beiden darf beliebig oft aufgerufen werden (dann aber nicht die andere!); Wechsel erst, wenn alle Aufrufe einer P bearbeitet sind
path 1:([P1] , P2) end	P1 und P2 arbeiten unter wechselseitigem Ausschluss; P1 darf beliebig oft aufgerufen werden; P2 darf nur eine Inkarnation haben; Wechsel erst, wenn alle Aufrufe von P1 bearbeitet sind oder P2 bearbeitet ist

Erzeuger/Verbraucher mit Pfadausdrücken

OBJEKT puffer

```
PATH N:( 1:(einfügen) ; 1:(entfernen) ) END
```

```
BufferArray buf[ N ]
```

```
INTEGER next-in = 1, next-aus = 1
```

```
PROZEDUR einfügen( stück )
```

```
  BEGIN
```

```
    buf[ next-in ] = stück
```

```
    next-in = next-in +N 1 // modulo Addition
```

```
  END
```

```
PROZEDUR entfernen( stück )
```

```
  BEGIN
```

```
    stück = buf[ next-aus ]
```

```
    next-aus = next-aus +N 1
```

```
  END
```

```
END
```

Implementierung von Pfadausdrücken

- Implementierung von Pfadausdrücken mit Semaphoren
 - Für jede Prozedur im Objekt wird *Prolog* und *Epilog* generiert
- Verkürzte Schreibweise
 - PATH "Ausdruck A" END \rightarrow $\langle A \rangle$
- Rekursive Umwandlung von $\langle A \rangle$ mit vier Regeln:
 1. $\langle L P1 , P2 R \rangle \rightarrow \langle L P1 R \rangle \langle L P2 R \rangle$
 2. $\langle L P1 ; P2 R \rangle \rightarrow \langle L P1 \text{ Up}(s1) \rangle \langle \text{Down}(s1) P2 R \rangle$
 - s1 ist neue, mit 0 initialisierte Semaphore
 3. $\langle L n:(P1) R \rangle \rightarrow \langle \text{Down}(s2) L P1 R \text{ Up}(s2) \rangle$
 - s2 ist neue, mit n initialisierte Semaphore
 4. $\langle L [P1] R \rangle \rightarrow \langle \text{DD}(c,s,L) P1 \text{ UU}(c,s,R) \rangle$
 - s ist neue, mit 1 initialisierte Semaphore
 - c ist neues, mit 0 initialisiertes INTEGER

Implementierung von Pfadausdrücken

```
DD( c, s, L )  
{  
  down( s )  
  c = c + 1  
  IF c = 1 THEN L  
  up( s )  
}
```

Bin ich der erste Prozess P1 im Ausdruck [P1] ?
Dann muss zuerst auf L gewartet werden.
Ansonsten ist L schon erfüllt.

```
UU( c, s, R )  
{  
  down( s )  
  c = c - 1  
  IF c = 0 THEN R  
  up( s )  
}
```

Bin ich der letzte Prozess P1 aus dem Ausdruck [P1] ?
Dann muss am Ende auf R gewartet werden.
Ansonsten muss noch nicht auf R gewartet werden.

Pfadausdrücke

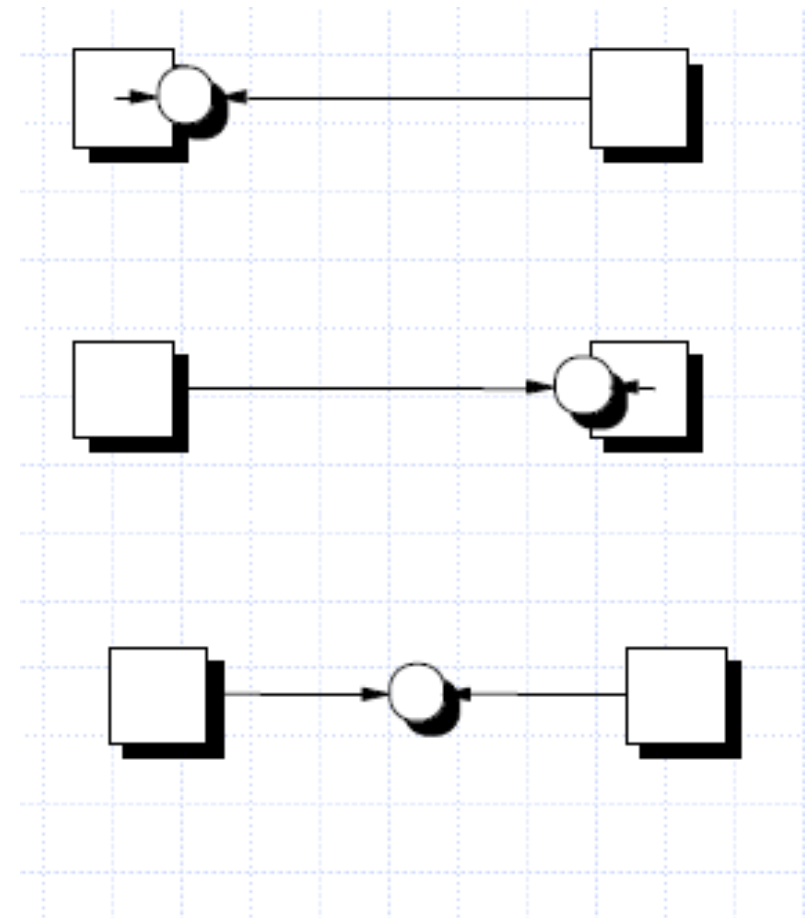
- Beispiel für Umwandlung: "Produzent-Verbraucher"
p = PATH N:(1:(einfügen) ; 1:(entfernen)) END
- 1. Einschränkung auflösen
 - < down(s0) 1:(einfügen) ; 1:(entfernen) up(s0) >
 - // s0 = N ist neue Semaphor
- 2. Sequenz auflösen
 - < down(s0) 1:(einfügen) up(s1) >
 - < down(s1) 1:(entfernen) up(s0) >
 - // s1 = 0 ist neue Semaphor
- 3. Einschränkung auflösen
 - < down(s2) down(s0) einfügen up(s1) up(s2) >
 - < down(s3) down(s1) entfernen up(s0) up(s3) >
 - // s2, s3 = 1 sind neue Semaphore

Mailbox

- Synchronisation über Nachrichten–Austausch
 - Ganz allgemein: Message Passing Systems
- Zwei Funktionsprimitive
 - `send(mailbox, message)`
 - `receive(mailbox, message)`
- Beide Funktionen arbeiten blockierend
- Die Mailbox hat eine beschränkte Größe
- Erzeuger/Verbraucher Problem lässt sich leicht abbilden, da die Mailbox ein beschränkter Puffer ist

Mailbox

- Die Mailbox kann ...
- beim Sender
- beim Empfänger,
- oder dazwischen realisiert sein



```
#define N 100
```

```
/* number of slots in the buffer */
```

```
void producer(void)
```

```
{
```

```
    int item;
```

```
    message m;
```

```
/* message buffer */
```

```
    while (TRUE) {
```

```
        item = produce_item( );
```

```
/* generate something to put in buffer */
```

```
        receive(consumer, &m);
```

```
/* wait for an empty to arrive */
```

```
        build_message(&m, item);
```

```
/* construct a message to send */
```

```
        send(consumer, &m);
```

```
/* send item to consumer */
```

```
    }
```

```
}
```

```
void consumer(void)
```

```
{
```

```
    int item, i;
```

```
    message m;
```

```
    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
```

```
    while (TRUE) {
```

```
        receive(producer, &m);
```

```
/* get message containing item */
```

```
        item = extract_item(&m);
```

```
/* extract item from message */
```

```
        send(producer, &m);
```

```
/* send back empty reply */
```

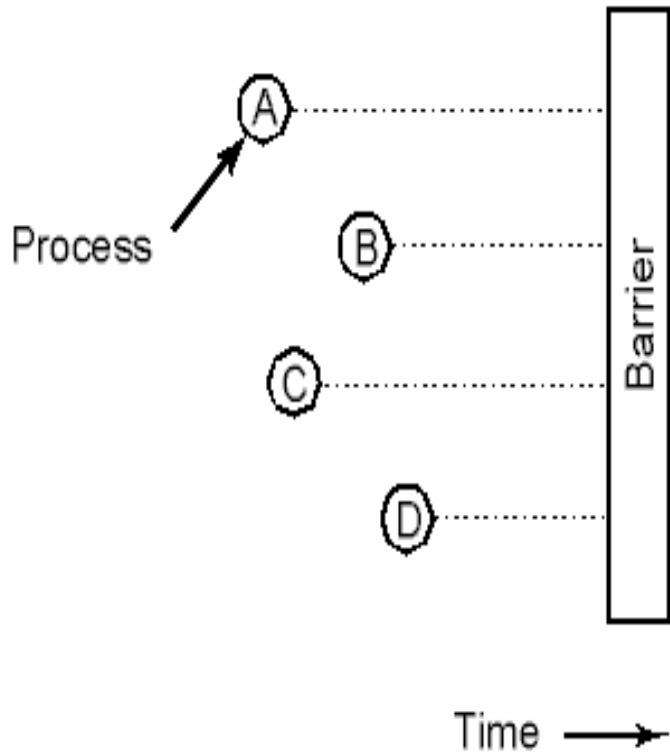
```
        consume_item(item);
```

```
/* do something with the item */
```

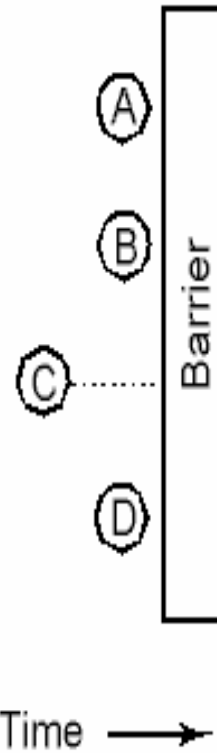
```
    }
```

```
}
```

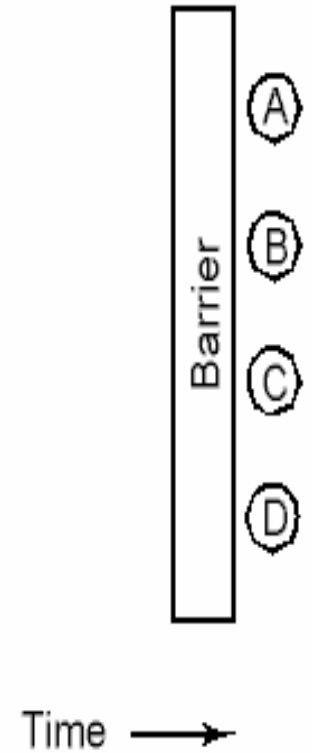
Barrieren



(a)



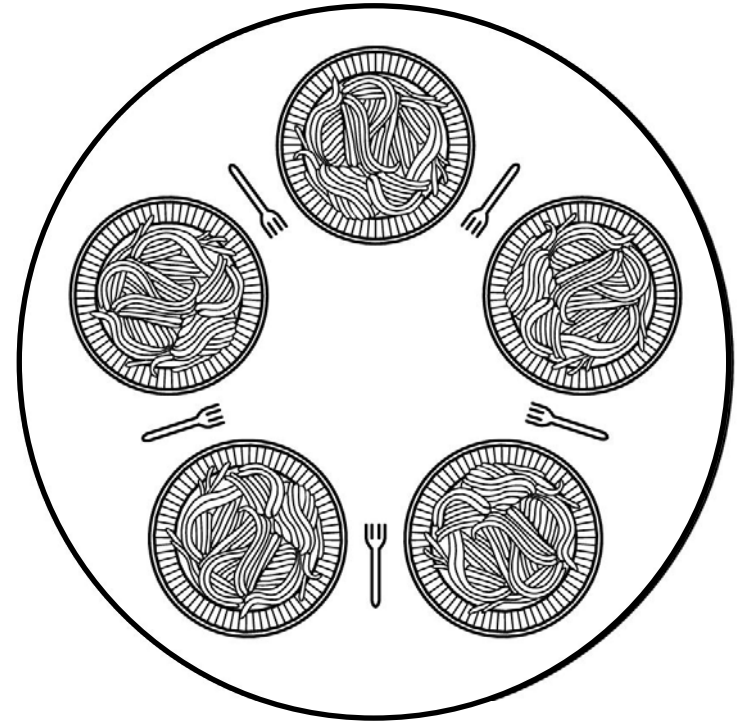
(b)



(c)

Klassische Synchronisationsprobleme

- **Dinierende Philosophen**
- **Problemstellung**
 - Jeder Philosoph denkt oder er will essen
 - n Philosophen und n Gabeln
 - Zum Essen braucht ein Philosoph 2 Gabeln
- **Falscher Ansatz**
 - 1.) Greife linke Gabel
 - 2.) Greife rechte Gabel
 - 3.) Essen



```

#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N  /* number of i's left neighbor */
#define RIGHT     (i+1)%N   /* number of i's right neighbor */
#define THINKING  0          /* philosopher is thinking */
#define HUNGRY    1          /* philosopher is trying to get forks */
#define EATING    2          /* philosopher is eating */
typedef int semaphore;      /* semaphores are a special kind of int */
int state[N];              /* array to keep track of everyone's state */
semaphore mutex = 1;      /* mutual exclusion for critical regions */
semaphore s[N];           /* one semaphore per philosopher */

void philosopher(int i)    /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {        /* repeat forever */
        think( );        /* philosopher is thinking */
        take_forks(i);   /* acquire two forks or block */
        eat( );          /* yum-yum, spaghetti */
        put_forks(i);    /* put both forks back on table */
    }
}

```

```

void take_forks(int i)                               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = HUNGRY;                               /* record fact that philosopher i is hungry */
    test(i);                                         /* try to acquire 2 forks */
    up(&mutex);                                       /* exit critical region */
    down(&s[i]);                                      /* block if forks were not acquired */
}

void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = THINKING;                             /* philosopher has finished eating */
    test(LEFT);                                      /* see if left neighbor can now eat */
    test(RIGHT);                                     /* see if right neighbor can now eat */
    up(&mutex);                                       /* exit critical region */
}

void test(i)                                         /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

Dinierende Philosophen mit Monitor

MONITOR Philosophen

```
{
  Zustand zustand[ N ] // Hunger, Essen, Denken
  CONDITION s[ N ] // zur Verzögerung

  PROCEDURE aufnehmen( i )
  {
    zustand[ i ] = Hunger
    test( i )
    IF ( zustand[ i ] != Essen ) THEN WAIT( s[ i ] )
  }

  PROCEDURE ablegen( i )
  {
    zustand[ i ] = Denken
    test( (i-1) mod N )
    test( (i+1) mod N )
  }
}
```

... next slide

Dinierende Philosophen mit Monitor

...

```
PROCEDURE test( i )
{
  IF ( zustand[ i ] = Hunger AND
      zustand[ (i-1) mod N ] != Essen AND
      zustand[ (i+1) mod N ] != Essen )
  THEN
  {
    zustand[ i ] = Essen
    SIGNAL( s[ i ] )
  }
}
} // Ende des Monitors
```

```
WHILE ( true )
{
  Philosophen.aufnehmen( i )
  Essen()
  Philosophen.ablegen( i )
}
```

Klassische Synchronisationsprobleme

- Viele Leser / Ein Schreiber
- Beispiel: Zugriff auf gemeinsame Datenbank
 - Mehrere Prozesse dürfen gleichzeitig lesen, aber zeitgleich darf keiner schreiben
 - Nur ein Prozess darf schreiben, dann darf aber kein anderer lesen
- Lösung
 - Erster Leser sperrt Datenbank, andere Leser werden gezählt
 - Letzter Leser entsperrt die Datenbank
 - Leser haben Vorrang vor Schreibern
 - Schreiber muss warten, bis keine Leser mehr da sind

```

semaphore mutex = 1;          /* controls access to 'rc' */
semaphore db = 1;            /* controls access to the database */
int rc = 0;                  /* # of processes reading or wanting to */

```

```

void reader(void)
{
    while (TRUE) {           /* repeat forever */
        down(&mutex);        /* get exclusive access to 'rc' */
        rc = rc + 1;         /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);          /* release exclusive access to 'rc' */
        read_data_base();    /* access the data */
        down(&mutex);        /* get exclusive access to 'rc' */
        rc = rc - 1;         /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex);          /* release exclusive access to 'rc' */
        use_data_read();     /* noncritical region */
    }
}

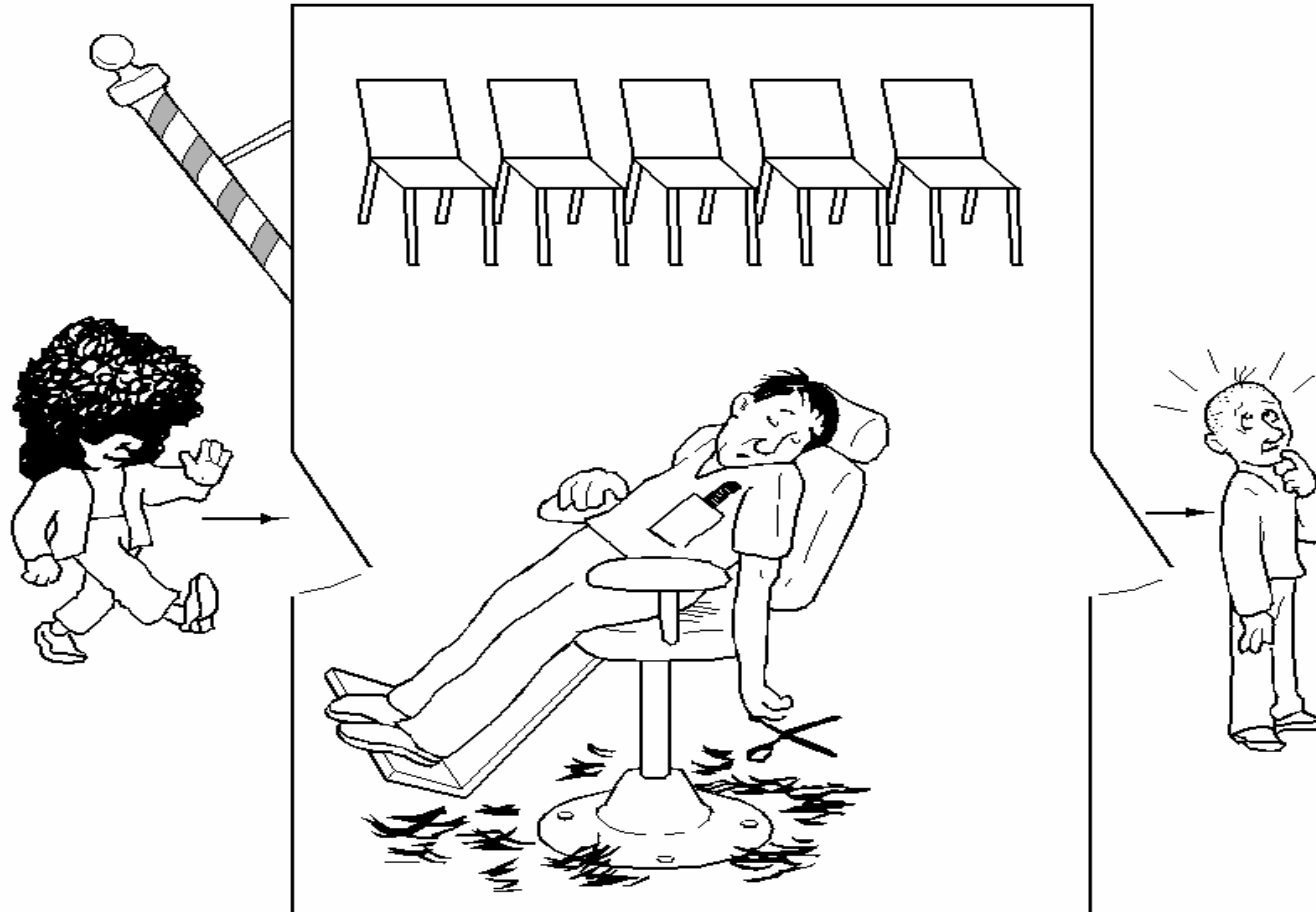
```

```

void writer(void)
{
    while (TRUE) {           /* repeat forever */
        think_up_data();     /* noncritical region */
        down(&db);           /* get exclusive access */
        write_data_base();   /* update the data */
        up(&db);             /* release exclusive access */
    }
}

```

Das Problem des schlafenden Friseurs



```

semaphore customers = 0;      /* # of customers waiting for service */
semaphore barbers = 0;      /* # of barbers waiting for customers */
semaphore mutex = 1;        /* for mutual exclusion */
int waiting = 0;            /* customers are waiting (not being cut) */

```

```

void barber(void)
{
    while (TRUE) {
        down(&customers);    /* go to sleep if # of customers is 0 */
        down(&mutex);        /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);        /* one barber is now ready to cut hair */
        up(&mutex);          /* release 'waiting' */
        cut_hair();          /* cut hair (outside critical region) */
    }
}

```

```

void customer(void)
{
    down(&mutex);            /* enter critical region */
    if (waiting < CHAIRS) { /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);        /* wake up barber if necessary */
        up(&mutex);            /* release access to 'waiting' */
        down(&barbers);        /* go to sleep if # of free barbers is 0 */
        get_haircut();         /* be seated and be serviced */
    } else {
        up(&mutex);            /* shop is full; do not wait */
    }
}

```

IPC in Windows NT

- Semaphore

- Wird mit einer positiven Zahl initialisiert und im Sinne einer anzahlbegrenzenden Kooperation (Capacity Lock) zur Betriebsmittelverwaltung verwendet
- Mit `CreateSemaphore()` wird das Objekt erzeugt und kann nach `OpenSemaphore()` benutzt werden
- Mit einer Warteoperation wie `WaitForSingleObject()` (entspricht down-Operation) wird der Zählerwert dekrementiert und mit `ReleaseSemaphore()` (entspricht up-Operation) inkrementiert
- Wenn der Zähler den Wert 0 erreicht hat, wirkt die Warteoperation blockierend
- Semaphore können adressraumübergreifend eingesetzt werden, (d.h. zwischen Threads unterschiedlicher Prozesse)

IPC in Windows NT

- Event

- Wird zur Signalisierung eingesetzt
- Synchronisationsobjekt wird mit `CreateEvent()` erzeugt und kann nach `OpenEvent()` benutzt werden
- `SetEvent()` entspricht dem signal, und für wait können die allgemeinen Wartefunktionen in NT wie `WaitForSingleObject()` eingesetzt werden
- Ein `ResetEvent()` erlaubt das explizite Rücksetzen des Signals
- Normalerweise arbeitet der Event als Gruppensignalisierung, d.h. durch ein Signal werden alle wartenden Threads deblockiert
- Wird bei der Event-Erzeugung jedoch die `AutoReset`-Option verwendet, so löst ein Signal nur einen Thread aus der Warteschlange (wie bei der Einzelsignalisierung).

IPC in Windows NT

- **Mutex**

- Ein Mutex dient dem gegenseitigen Ausschluss
- Nach `CreateMutex()` und `OpenMutex()` kann mit einer Warteoperation (z.B. `WaitForSingleObject()`) der kritische Abschnitt betreten werden
- Bei Verlassen des kritischen Abschnitts wird die Sperre mit `ReleaseMutex()` wieder freigegeben
- Ein Mutex kann von beliebigen Threads im System benutzt werden

IPC in Windows NT

- **Critical Section**

- Ein Critical-Section-Objekt ist eine vereinfachte und effizientere Variante des Mutex speziell für den gegenseitigen Ausschluss zwischen Threads im selben Prozess
- Mit `InitializeCriticalSection()` angelegt, wird der kritische Bereich über `EnterCriticalSection()` betreten und mit `LeaveCriticalSection()` wieder verlassen

IPC in Windows NT

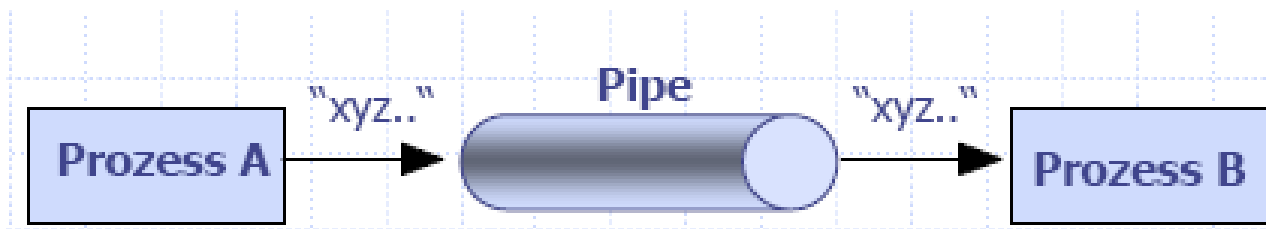
- Auszug aus dem Windows Platform SDK
- The `InitializeCriticalSectionAndSpinCount` function initializes a critical section object and sets the spin count for the critical section. Spinning means that when a thread tries to acquire a critical section that is locked, the thread enters a loop, checks to see if the lock is released, and if the lock is not released, the thread goes to sleep.
- `BOOL InitializeCriticalSectionAndSpinCount(LPCRITICAL_SECTION lpCriticalSection, DWORD dwSpinCount);`
- Parameters
- *lpCriticalSection*
 - [in, out] Pointer to the critical section object.
- *dwSpinCount*
 - [in] Spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0 (zero). On multiprocessor systems, if the critical section is unavailable, the calling thread spin *dwSpinCount* times before performing a wait operation on a semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

IPC in UNIX (IEEE POSIX Standard)

- Mutex dient dem gegenseitigen Ausschluss
 - `pthread_mutex_init()` initialisiert ein Mutex-Objekt
 - `pthread_mutex_lock()` wird beim Betreten des krit. Abschnitts aufgerufen und blockiert den Aufrufer, falls belegt
 - `pthread_mutex_trylock()` ist die nichtblockierende Variante: Falls frei, wird belegt; Falls belegt, wird mit entsprechendem Hinweis zurückgekehrt
 - `pthread_mutex_unlock()` beim Verlassen des kritischen Abschnitts
- Eine Condition-Variable dient der Signalisierung
 - `pthread_cond_wait()` blockiert, falls Signal nicht gesetzt
 - `pthread_cond_timedwait()` zusätzlich mit Fristablauf (time-out)
 - `pthread_cond_signal()` setzt das Signal und deblockiert den „vordersten“ Thread (Priorität bzw. FCFS)
 - `pthread_cond_broadcast()` setzt Signal und deblockiert alle wartenden Prozesse

Pipes unter Windows und UNIX

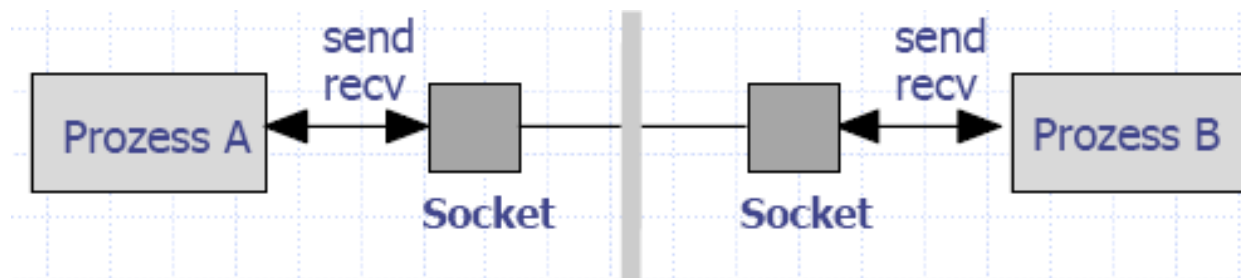
- Spezieller 1:1 Kanal für kontinuierlichen, gerichteten Zeichenstrom



- Die Pipe hat eine begrenzte Kapazität.
- Ist die Pipe voll, so wird ein sendender (schreibender) Prozess blockiert.
- Ist die Pipe leer, so wird ein empfangender (lesender) Prozess blockiert.
- Nur lokaler Mechanismus zwischen genau zwei Prozessen
- Lesen und Schreiben in unterschiedlichen Blockgrößen (im Gegensatz zur Mailbox)

Sockets unter Windows und UNIX

- Sockets sind Endpunkte einer Duplex-Verbindung
- Ein Socket kann von mehreren Prozessen benutzt werden
- Verschiedene Socket-Typen werden angeboten:
 - stream socket: verbindungsorientiert
 - datagram socket: paketorientiert
 - raw socket: Durchgriff auf zugrundeliegende Protokolle
- Einsatz vor allem zur nichtlokalen Kommunikation (Verteilte Systeme)
- Blockierend (synchron) oder nichtblockierend (asynchron)



Sockets unter Windows und UNIX

