4 Prozesskommunikation (3)

◆ Übertragungs- und Aufrufeigenschaften
- zuverlässig — unzuverlässig
- gepuffert — ungepuffert
- blockierend — nichtblockierend
- stromorientiert — nachrichtenorientiert — RPC

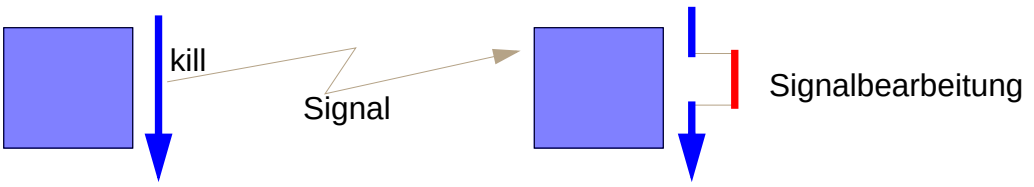
◆ Adressierung
- implizit: UNIX Pipes
- explizit: Sockets
- globale Adressierung: Sockets, Ports
- Gruppenadressierung: Multicast, Broadcast
- funktionale Adressierung: Dienste

4.1 UNIX Signale

■ Signale sind Unterbrechungen ähnlich denen eines Prozessors

◆ Sender:
- Betriebssystem bei bestimmten Ereignissen
- Prozesse mit Hilfe des Systemaufrufs **`kill`**

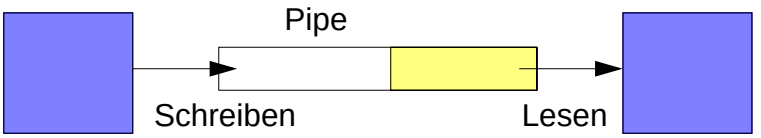
◆ Empfänger-Prozess führt eine definierte Signalbehandlung durch
- Ignorieren
- Terminierung des Prozesses
- Aufruf einer Funktion (Signalbearbeitung)
  Nach der Behandlung läuft Prozess an unterbrochener Stelle weiter



◆ kill + Signalbearbeitung = minimale Prozesskomunikation (1 Bit)

4.2 Pipes

- Kanal zwischen zwei Kommunikationspartnern
  - unidirektional
    (heute gleichzeitige Erzeugung zweier Pipes je eine pro Richtung)
  - gepuffert (feste Puffergröße), zuverlässig, stromorientiert



- Operationen: Schreiben und Lesen
  - Ordnung der Zeichen bleibt erhalten (Zeichenstrom)
  - Blockierung bei voller Pipe (Schreiben) und leerer Pipe (Lesen)

4.2 Pipes (2)

- Systemaufruf unter Solaris
  - Öffnen einer Pipe

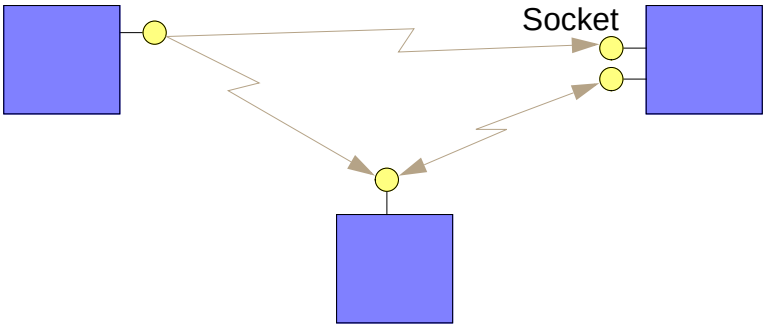
    ```
    int pipe( int fdes[2] );
    ```

  - Es werden eigentlich zwei Pipes geöffnet
    `fdes[0]` liest aus Pipe 1 und schreibt in Pipe 2
    `fdes[1]` liest aus Pipe 2 und schreibt in Pipe 1

  - Zugriff auf Pipes wie auf eine Datei: **`read`** und **`write`**, **`readv`** und **`writev`**
- Named-Pipes
  - Pipes können auch als Spezialdateien ins Dateisystem gelegt werden.
  - Standardfunktionen zum Lesen und Schreiben können dann verwendet werden.

4.3 Sockets

- Allgemeine Kommunikationsendpunkte
  - bidirektional, gepuffert



  - Auswahl einer Protokollfamilie
    - z.B. Internet (TCP/IP), UNIX (innerhalb von Prozessen der gleichen Maschine), ISO, Appletalk, DECnet, SNA, ...
    - durch die Protokollfamilie wird gleichzeitig die Adressfamilie festgelegt (Struktur zur Bezeichnung von Protokolladressen)

4.3 Sockets (2)

- Auswahl eines Sockettyps für Protokolle mit folgenden Eigenschaften:
  - stromorientiert, verbindungsorientiert und gesichert
  - nachrichtenorientiert und ungesichert (Datagramm)
  - nachrichtenorientiert und gesichert
- Auswahl eines Protokolls der Familie
  - z.B. UDP
- explizite Adressierung
  - Unicast: genau ein Kommunikationspartner
  - Multicast: eine Gruppe
  - Broadcast: alle möglichen Adressaten
- Sockets können blockierend und nichtblockierend betrieben werden.

4.3 Sockets (3)

- UNIX-Domain
  - ◆ UNIX-Domain-Sockets verhalten sich wie bidirektionale Pipes.
  - ◆ Anlage als Spezialdatei im Dateisystem möglich
- Internet-Domain
  - ◆ Protokolle:
    - TCP/IP (strom- und verbindungsorientiert, gesichert)
    - UDP/IP (nachrichtenorientiert, verbindungslos, ungesichert)
      – Nachrichten können verloren oder dupliziert werden
      – Reihenfolge kann durcheinander geraten
      – Paketgrenzen bleiben erhalten
      (Datagramm-Protokoll)
  - ◆ Adressen: IP-Adressen und Port-Nummern

4.3 Sockets (4)

- Anlegen von Sockets
  - ◆ Generieren eines Sockets mit (Rückgabewert ist ein Filedeskriptor)

```
int socket( int domain, int type, int proto );
```

  - ◆ Adresszuteilung
    - Sockets werden ohne Adressen generiert
    - Adressenzuteilung erfolgt automatisch oder durch:

```
int bind( int socket, const struct sockaddr *address,
          size_t address_len);
```

4.3 Sockets (5)

- Datagramm-Sockets
  - kein Verbindungsaufbau notwendig
  - Datagramm senden

```
ssize_t sendto( int socket, const void *message,
       size_t length, int flags,
       const struct sockaddr *dest_addr, size_t dest_len);
```

  - Datagramm empfangen

```
ssize_t recvfrom( int socket, void *buffer,
       size_t length, int flags, struct sockaddr *address,
       size_t *address_len);
```


4.3 Sockets (6)

- Stromorientierte Sockets
  - Verbindungsaufbau notwendig
  - *Client* (Benutzer, Benutzerprogramm) will zu einem *Server* (Dienstanbieter) eine Kommunikationsverbindung aufbauen
- Client: Verbindungsaufbau bei stromorientierten Sockets
  - Verbinden des Sockets mit

```
int connect( int socket, const struct sockaddr *address,
       size_t address_len);
```

  - Senden und Empfangen mit **`write`** und **`read`** (**`send`** und **`recv`**)
  - Beenden der Verbindung mit **`close`** (schließt den Socket)

4.3 Sockets (7)

- Server
  - bindet Socket an eine Adresse (sonst nicht zugreifbar)
  - bereitet Socket auf Verbindungsanforderungen vor durch

    ```
    int listen(int s, int backlog);
    ```

  - akzeptiert einzelne Verbindungsanforderungen durch

    ```
    int accept(int s, struct sockaddr *addr, int *addrlen);
    ```

    - gibt einen neuen Socket zurück, der mit dem Client verbunden ist
    - blockiert, falls kein Verbindungswunsch vorhanden

  - liest Daten mit **`read`** und führt den angebotenen Dienst aus
  - schickt das Ergebnis mit **`write`** zurück zum Sender
  - schließt den neuen Socket

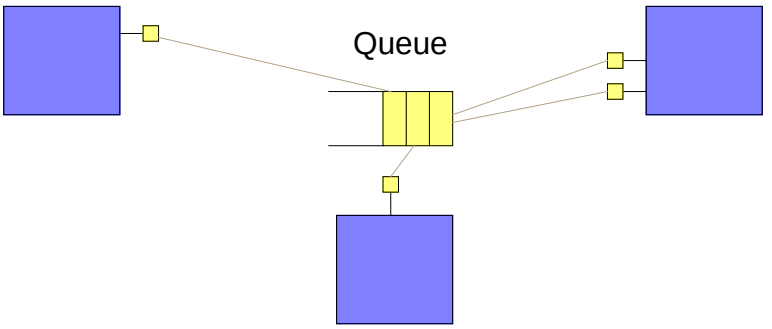
4.4 UNIX Queues

- Nachrichtenpuffer *(Queue, FIFO)*
  - rechnerlokale Adresse *(Key)* dient zur Identifikation eines Puffers
  - prozesslokale Nummer *(MSQID)* ähnlich dem Filedeskriptor
    (wird bei allen Operationen benötigt)
  - Zugriffsrechte wie auf Dateien
  - ungerichtete Kommunikation, gepuffert (einstellbare Größe pro Queue)
  - Nachrichten haben einen Typ (**`long`**-Wert)
  - Operationen zum Senden und Empfangen einer Nachricht
  - blockierend — nichtblockierend
  - alle Nachrichten — nur ein bestimmter Typ

4.4 UNIX Queues (2)

- Systemaufrufe unter Solaris 2.5
  - Erzeugen einer Queue bzw. Holen einer MSQID

```
int msgget( key_t key, int msgflg );
```



  - Alle kommunizierenden Prozesse müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann keine Queue mit gleichem Key erzeugt werden

4.4 UNIX Queues (3)

- Es können Queues ohne Key erzeugt werden (private Queues)
  - Nicht-private Queues sind persistent
  - Sie müssen explizit gelöscht werden

```
int msgctl( int msqid, int cmd, struct msqid_ds *buf );
```

- Systemkommandos zum Behandeln von Queues
  - Listen aktiver Message-Queues

```
ipcs -q
```

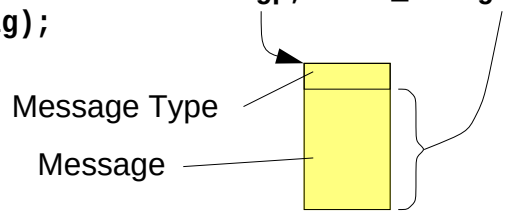
  - Löschen von Queues

```
ipcrm -Q <key>
```

4.4 UNIX Queues (4)

- Operationen auf Queues
  - Senden einer Nachricht

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



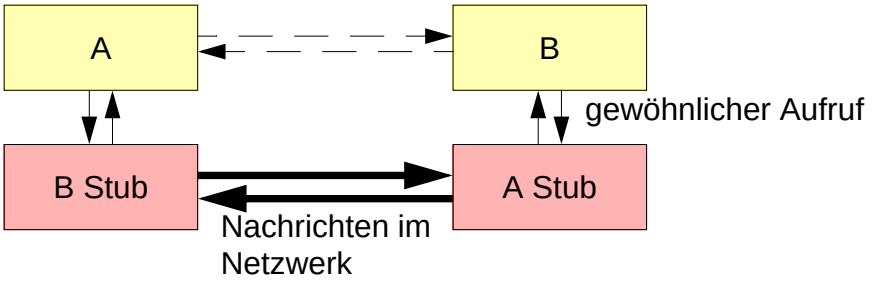
  - Empfangen einer Nachricht

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

  - Zugriffsrechte werden beachtet

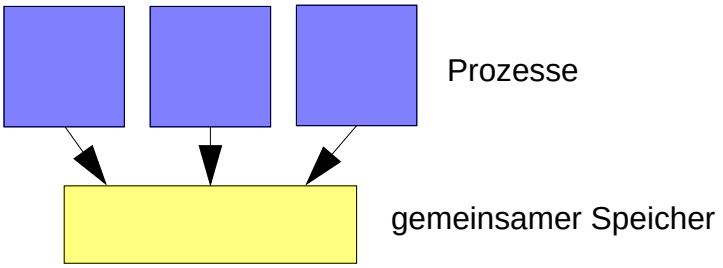
4.5 Fernaufruf *(RPC)*

- Funktionsaufruf über Prozessgrenzen hinweg *(Remote procedure call)*
  - hoher Abstraktionsgrad
  - selten wird Fernaufruf direkt vom System angeboten; benötigt Abbildung auf andere Kommunikationsformen z.B. auf Nachrichten
  - Abbildung auf mehrere Nachrichten
    - Auftragsnachricht transportiert Aufrufabsicht und Parameter.
    - Ergebnisnachricht transportiert Ergebnisse des Aufrufs.



4.6 Gemeinsamer Speicher

- Zwei Prozesse können auf einen gemeinsamen Speicherbereich zugreifen
  - gemeinsame Variablen oder Datenstrukturen



- Einrichten von gemeinsamem Speicher erst im Abschnitt E.5.

5 Aktivitätsträger *(Threads)*

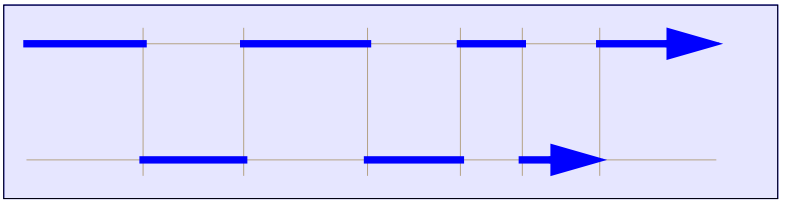
- Mehrere Prozesse zur Strukturierung von Problemlösungen
  - Aufgaben eines Prozesses leichter modellierbar, wenn in mehrere kooperierende Prozesse unterteilt
    z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
    z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
  - Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
    z.B. wissenschaftliches Hochleistungsrechnen (Aerodynamik etc.)
  - Client-Server-Anwendungen unter UNIX: pro Anfrage wird ein neuer Prozess gestartet
    z.B. Webserver

5.1 Prozesse mit gemeinsamem Speicher

- Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse
- Nachteile
  - viele Betriebsmittel zur Verwaltung eines Prozesses notwendig
    - Dateideskriptoren
    - Speicherabbildung
    - Prozesskontrollblock
  - Prozessumschaltungen sind aufwendig.
- Vorteil
  - In Multiprozessorsystemen sind echt parallele Abläufe möglich.

5.2 Koroutinen

- Einsatz von Koroutinen
  - einige Anwendungen lassen sich mit Hilfe von Koroutinen (auf Benutzerebene) innerhalb eines Prozesses gut realisieren



ein Prozess
zwei Koroutinen

- Nachteile:
  - Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
  - in Multiprozessorsystemen keine parallelen Abläufe möglich
  - Wird eine Koroutine in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert.

5.3 Aktivitätsträger

★ **Lösungsansatz:**
Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(Lightweighted Processes, LWPs)*

- Eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln.
  - Instruktionen
  - Datenbereiche
  - Dateien, Semaphoren etc.
- Jeder Thread repräsentiert eine eigene Aktivität:
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack

5.3 Aktivitätsträger (2)

- Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.
  - Es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
  - Speicherabbildung muss nicht gewechselt werden.
  - Alle Systemressourcen bleiben verfügbar.

■ Ein UNIX-Prozess ist ein Adressraum mit einem Thread
- Solaris: Prozess kann mehrere Threads besitzen

■ Implementierungen von Threads
- User-level Threads
- Kernel-level Threads

5.4 User-Level-Threads

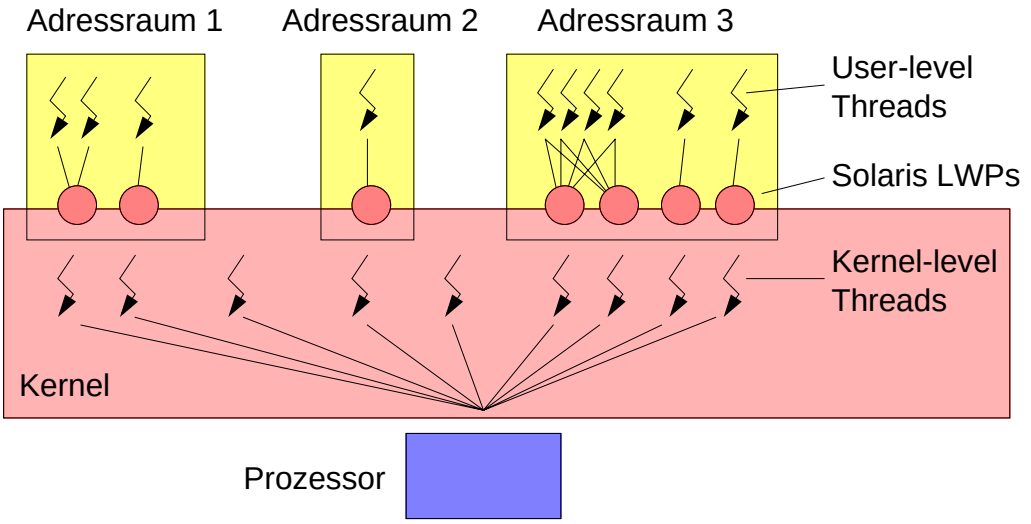
- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - Betriebssystem sieht nur einen Thread
- ★ Vorteile
  - keine Systemaufrufe zum Umschalten erforderlich
  - effiziente Umschaltung
  - Schedulingstrategie in der Hand des Anwenders
- ▲ Nachteile
  - Bei blockierenden Systemaufrufen bleiben alle User-Level-Threads stehen.
  - Kein Ausnutzen eines Multiprozessors möglich

5.5 Kernel-Level-Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level-Threads
  - Betriebssystem schaltet Threads um
- ★ Vorteile
  - kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
- ▲ Nachteile
  - weniger effizientes Umschalten
  - Fairnessverhalten nötig
    (zwischen Prozessen mit vielen und solchen mit wenigen Threads)
  - Schedulingstrategie meist vorgegeben

5.6 Beispiel: LWPs und Threads (Solaris)

- Solaris kennt Kernel-, User-Level-Threads und LWPs



Nach Silberschatz, 1994

6 Koordinierung

- Beispiel: Beobachter und Protokollierer
  - Mittels Induktionsschleife werden Fahrzeuge gezählt. Alle 10min druckt der Protokollierer die im letzten Zeitraum vorbeigekommene Anzahl aus.

```
int cnt= 0;

Observer
    on indication
        cnt= cnt+1;

Logger
    every 10 minutes
        printf( "Count= %d\n", cnt );
        cnt= 0;
```

6 Koordinierung (2)

- Effekte:
  - Fahrzeuge gehen „verloren“
  - Fahrzeuge werden doppelt gezählt
- Ursachen:
  - Befehle in C werden nicht unteilbar (atomar) abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden.
  - In C werden keinesfalls mehrere Anweisungen zusammen atomar abgearbeitet.
  - Prozesswechsel innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen können zu Inkonsistenzen führen.

6 Koordinierung (3)

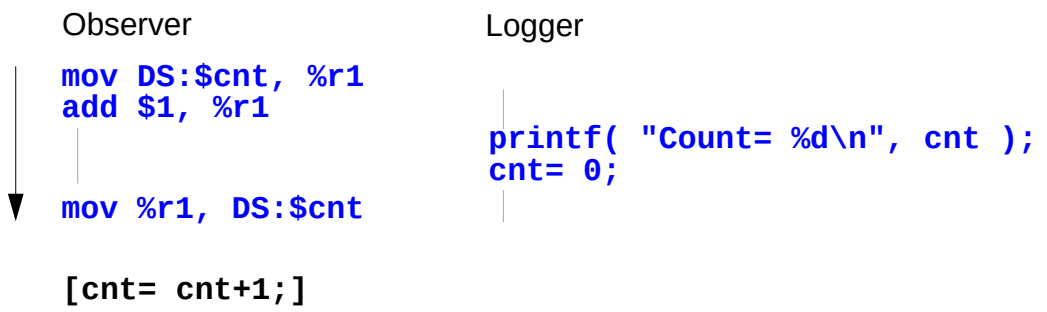
- Fahrzeuge gehen „verloren“
  - Nach dem Drucken wird der Protokollierer unterbrochen. Beobachter zählt weitere Fahrzeuge. Anzahl wird danach ohne Beachtung vom Protokollierer auf Null gesetzt.

Observer	Logger
	`printf( "Count= %d\n", cnt );`
`cnt= cnt+1;`	
`cnt= cnt+1;`	
	`cnt= 0;`

6 Koordinierung (4)

▲ Fahrzeuge werden doppelt gezählt:

◆ Beobachter will Zähler erhöhen und holt sich diesen dazu in ein Register. Er wird unterbrochen und der Protokollierer setzt Anzahl auf Null. Beobachter erhöht Registerwert und schreibt diesen zurück. Dieser Wert wird erneut vom Protokollierer registriert.



6 Koordinierung (5)

■ Gemeinsame Nutzung von Daten oder Betriebsmitteln

◆ kritische Abschnitte:

- nur einer soll Zugang zu Daten oder Betriebsmitteln haben (gegenseitiger Ausschluss, *Mutual Exclusion*, *Mutex*)
- kritische Abschnitte erscheinen allen anderen als zeitlich unteilbar

◆ Wie kann der gegenseitige Ausschluss in kritischen Abschnitten erzielt werden?

■ Koordinierung allgemein:

◆ Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern

★ Hinweis:

◆ Im Folgenden wird immer von Prozessen die Rede sein. Koordinierung kann/muss selbstverständlich auch zwischen Threads stattfinden.

6.1 Gegenseitiger Ausschluss

- Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten
  - Annahme: Maschinenbefehle sind unteilbar (atomar)
- 1. Versuch

```
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    while( turn == 1 );
    ...
    /* critical sec. */
    ...
    turn= 1;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    while( turn == 0);
    ...
    /* critical sec. */
    ...
    turn= 0;

    ... /* uncritical */
}
```


6.1 Gegenseitiger Ausschluss (2)

- Probleme der Lösung
  - nur alternierendes Betreten des kritischen Abschnitts durch P_0 und P_1 möglich
  - Implementierung ist unvollständig
  - aktives Warten
- Ersetzen von **`turn`** durch zwei Variablen **`ready0`** und **`ready1`**
  - **`ready0`** zeigt an, dass Prozess 0 bereit für den kritischen Abschnitt ist
  - **`ready1`** zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist

6.1 Gegenseitiger Ausschluss (3)

- 2. Versuch

```
bool ready0= FALSE;
bool ready1= FALSE;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    while( ready1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    while( ready0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```


6.1 Gegenseitiger Ausschluss (4)

- Gegenseitiger Ausschluss wird erreicht
  - leicht nachweisbar durch Zustände von **`ready0`** und **`ready1`**
- Probleme der Lösung
  - aktives Warten
  - Verklemmung möglich

6.1 Gegenseitiger Ausschluss (5)

- Betrachtung der nebenläufigen Abfolgen

P_0	P_1
`ready0= TRUE;`	`ready1= TRUE;`
`while( ready1 ); +`	`while( ready0 ); +`
`<critical> +`	`<critical> +`
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical> +`	`<noncritical> +`
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

`+` = mehrfach, mind. einmal
`*` = mehrfach oder gar nicht

◆ Durchspielen aller möglichen Durchmischungen

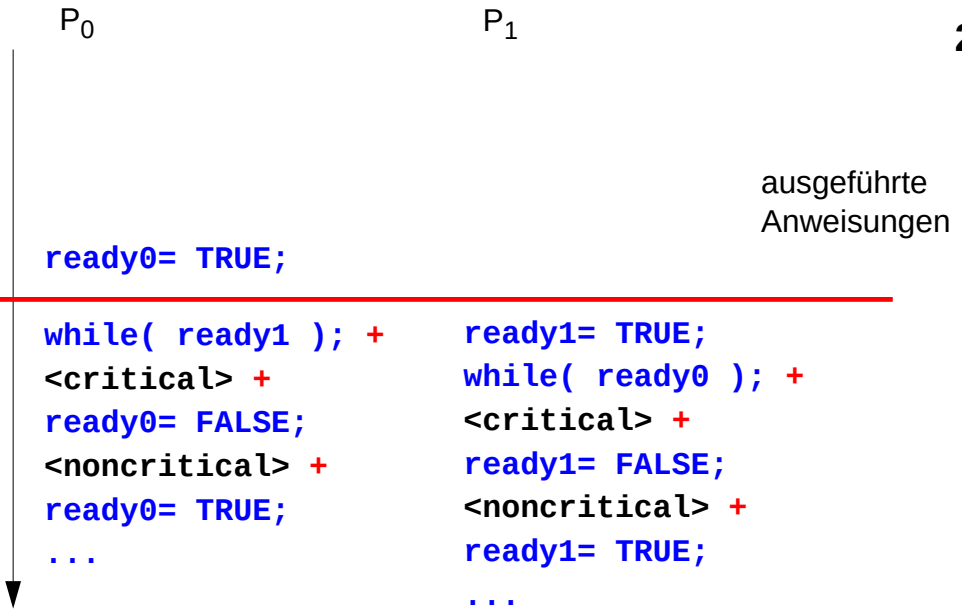
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

P_0	P_1
	1
	ausgeführte Anweisungen
`ready0= TRUE;`	`ready1= TRUE;`
`while( ready1 ); +`	`while( ready0 ); +`
`<critical> +`	`<critical> +`
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical> +`	`<noncritical> +`
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

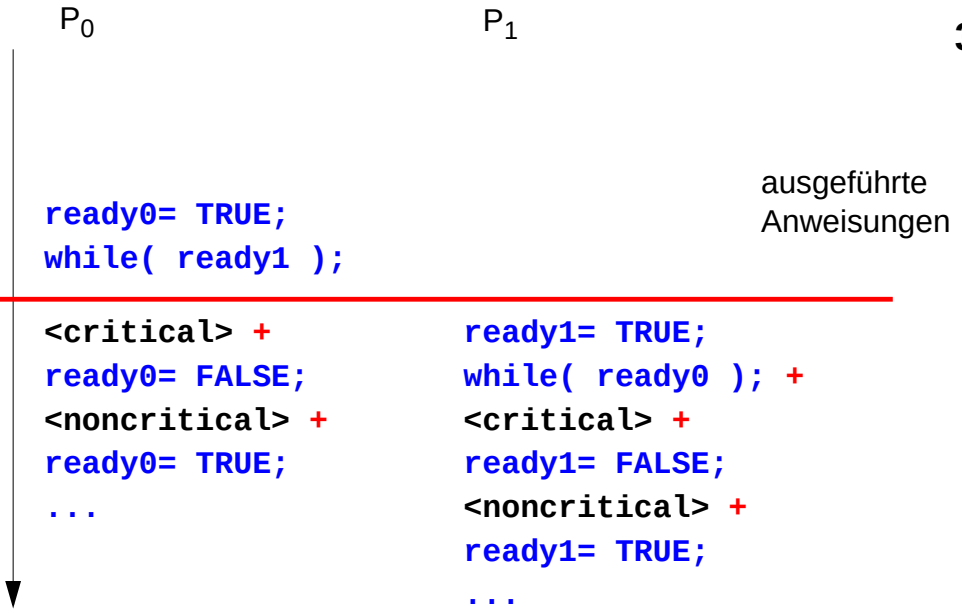
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung



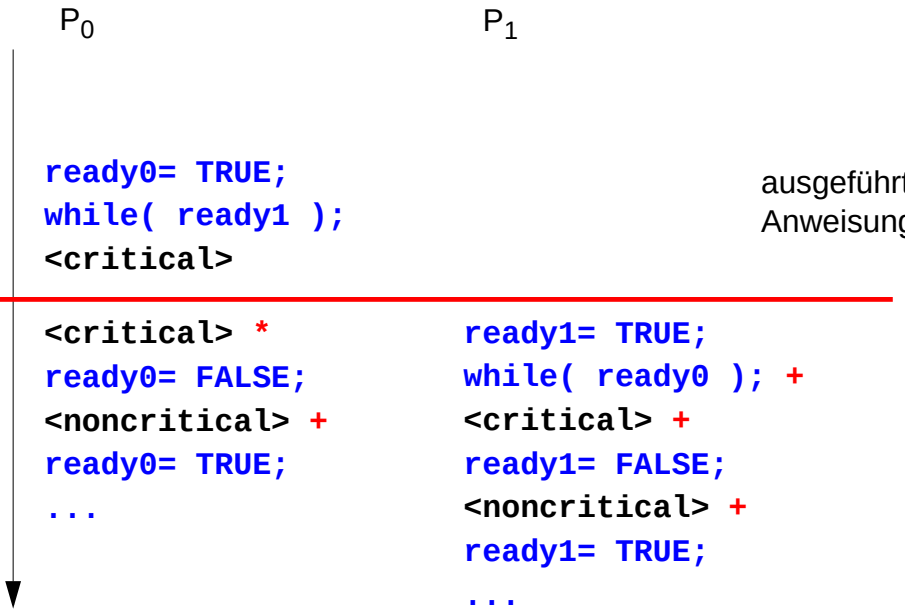
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung



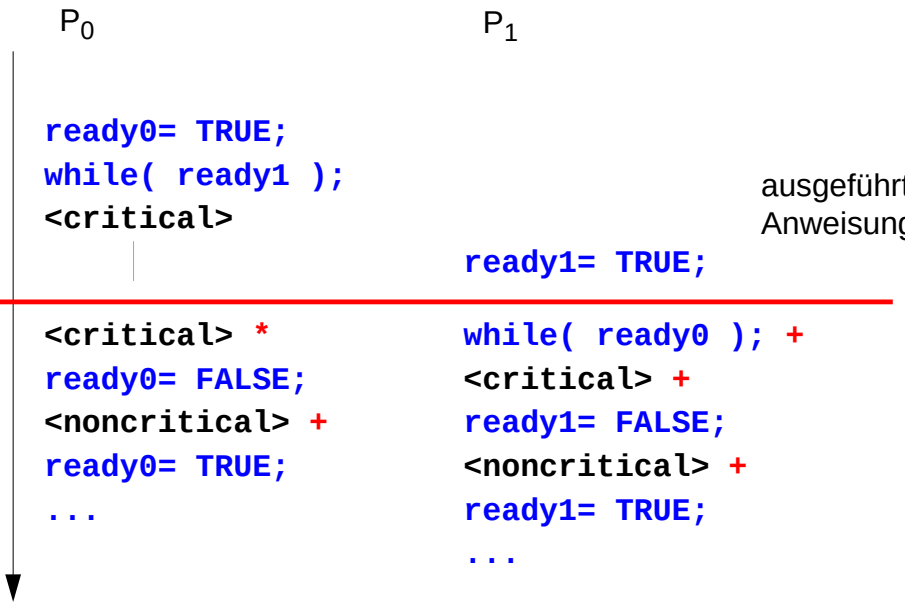
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung



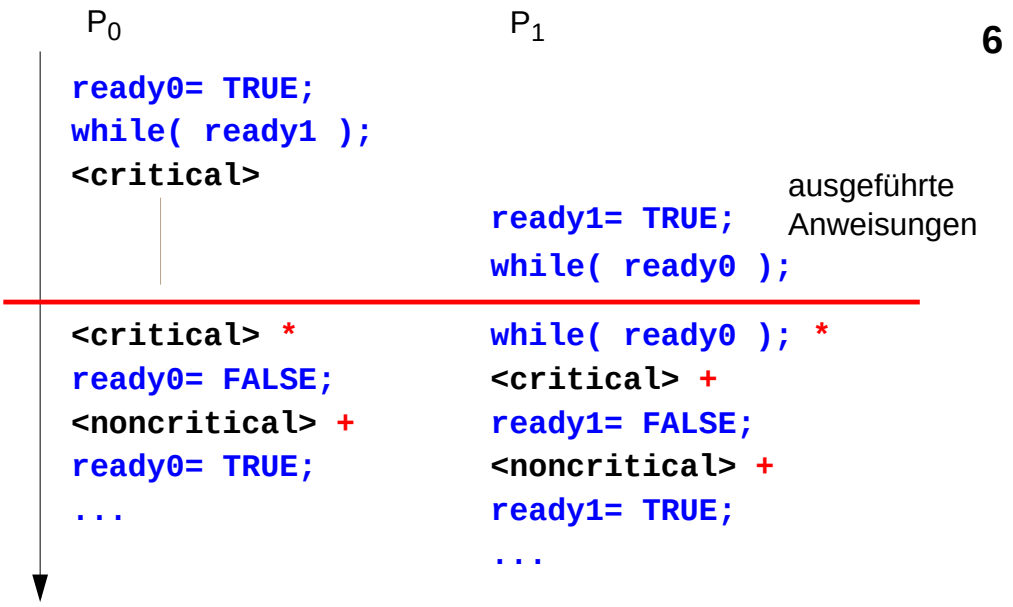
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung



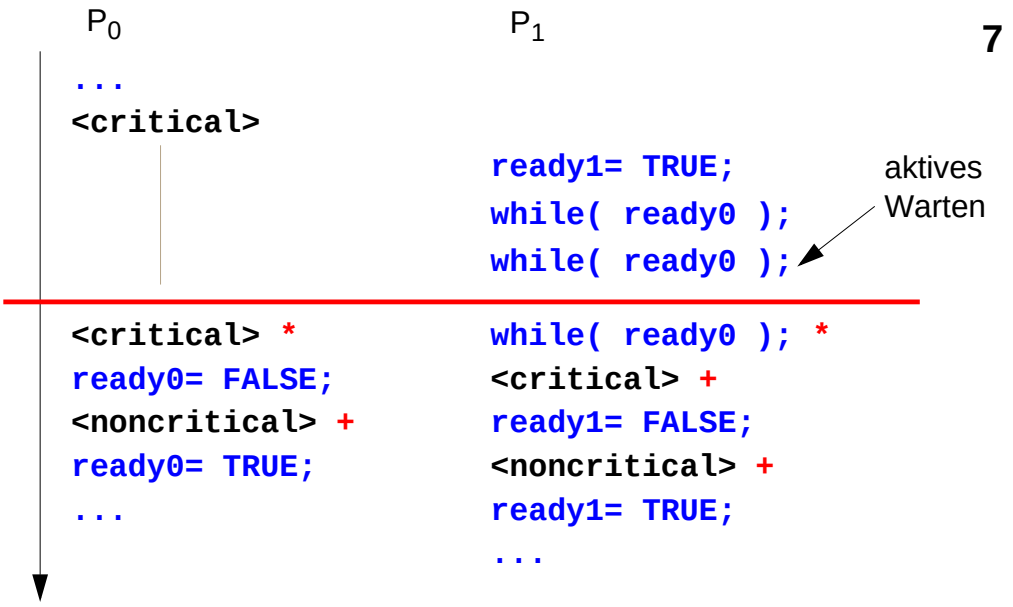
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung



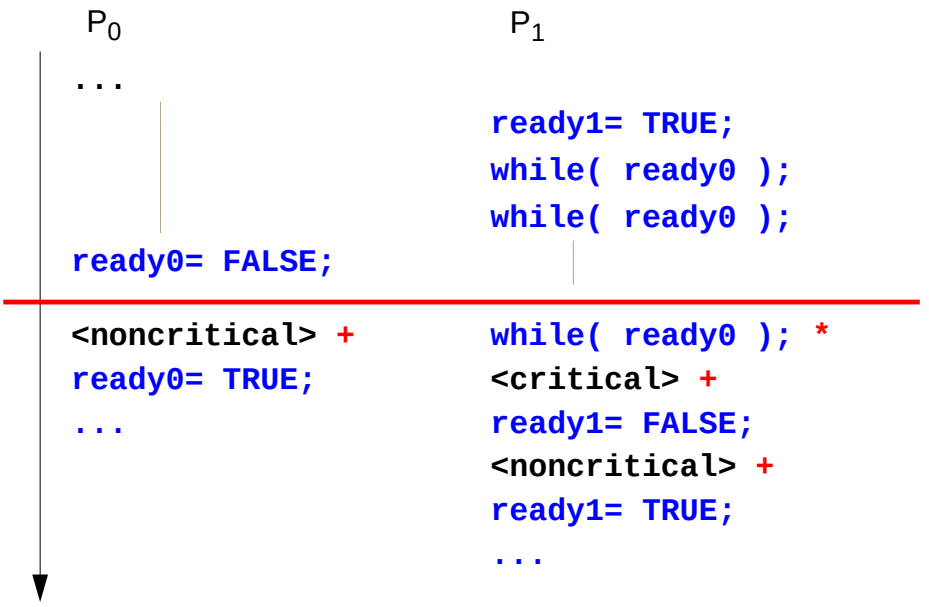
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung



6.1 Gegenseitiger Ausschluss (6)

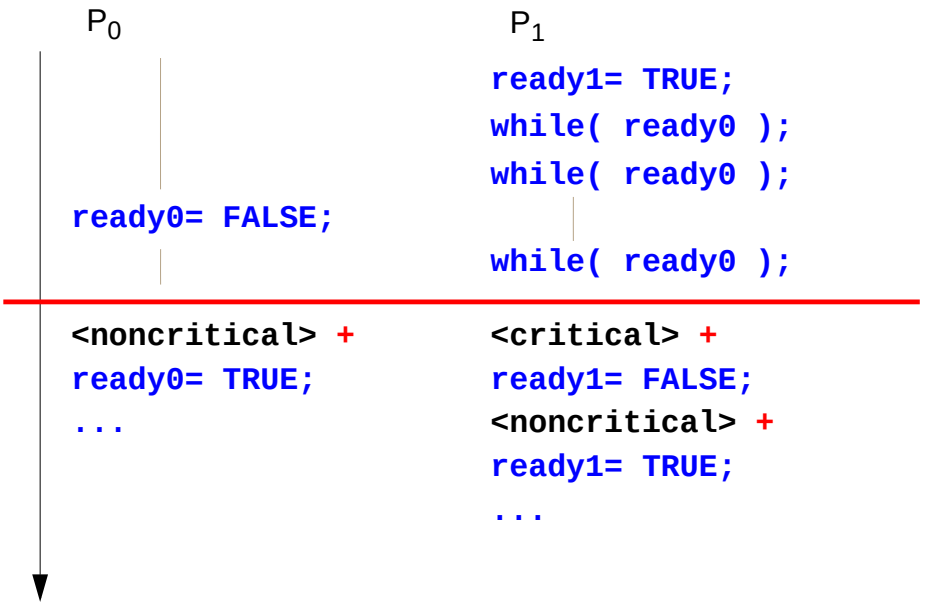
- Harmlose Durchmischung



8

6.1 Gegenseitiger Ausschluss (6)

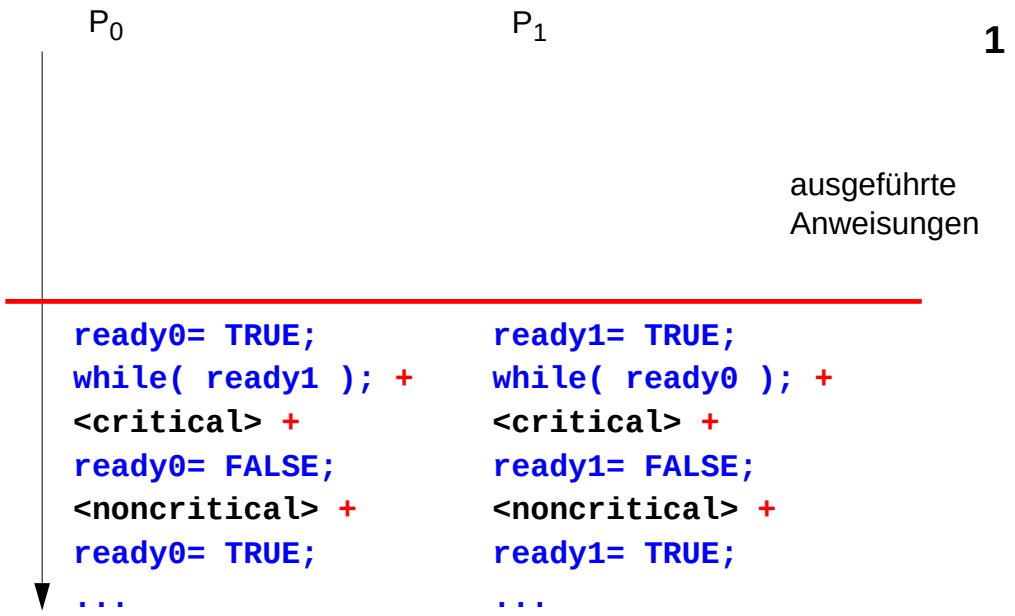
- Harmlose Durchmischung



9

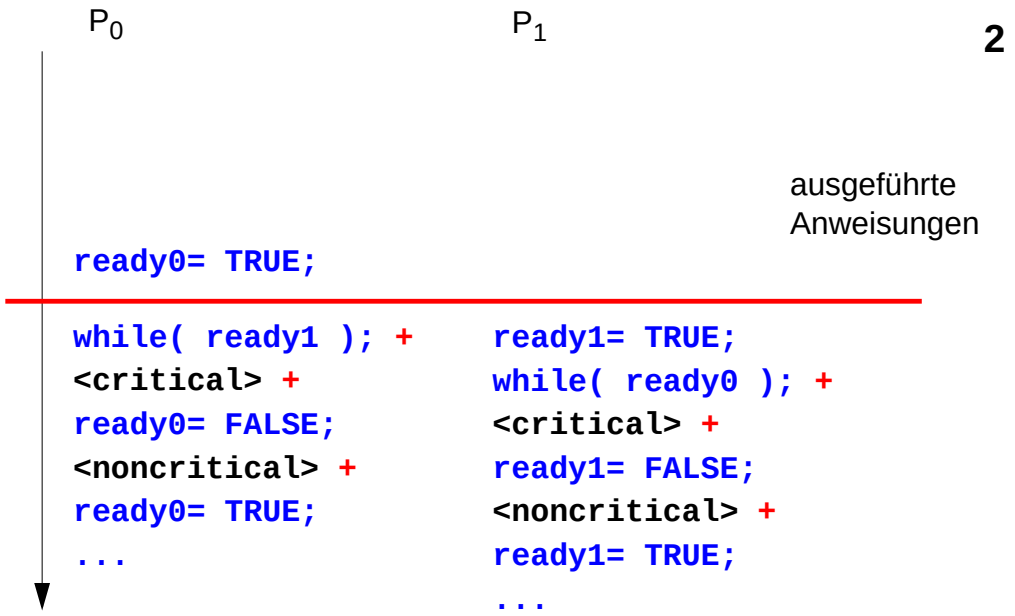
6.1 Gegenseitiger Ausschluss (7)

- Verklemmung



6.1 Gegenseitiger Ausschluss (7)

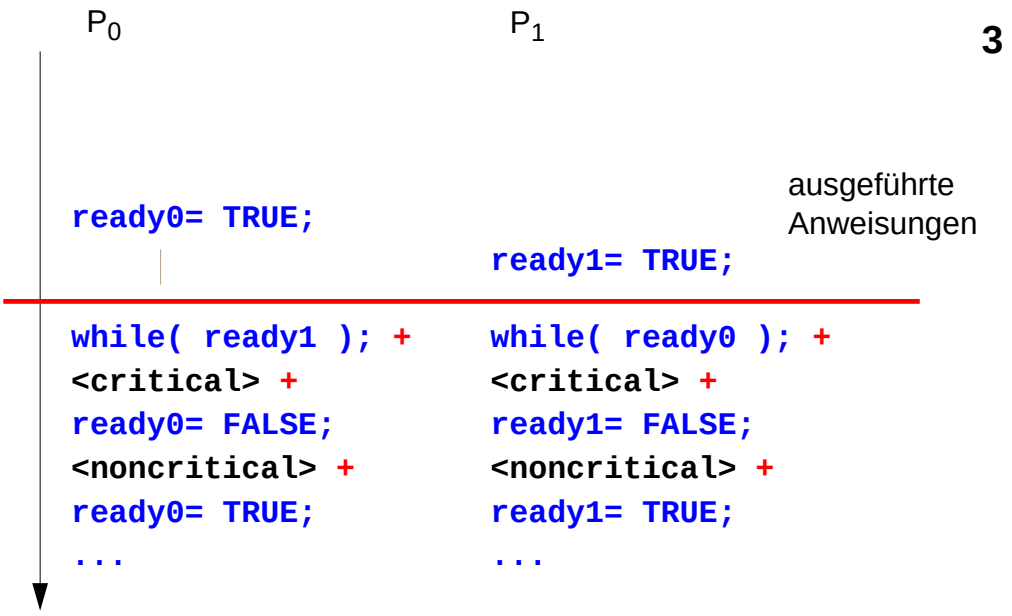
- Verklemmung



Systemprogrammierung I
D – 80
© 1997-2003, F. J. Hauck, W. Schröder-Preikschat, Inf 4, FAU Erlangen-Nürnberg[D-Proc.fm, 2003-11-26 18.32]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

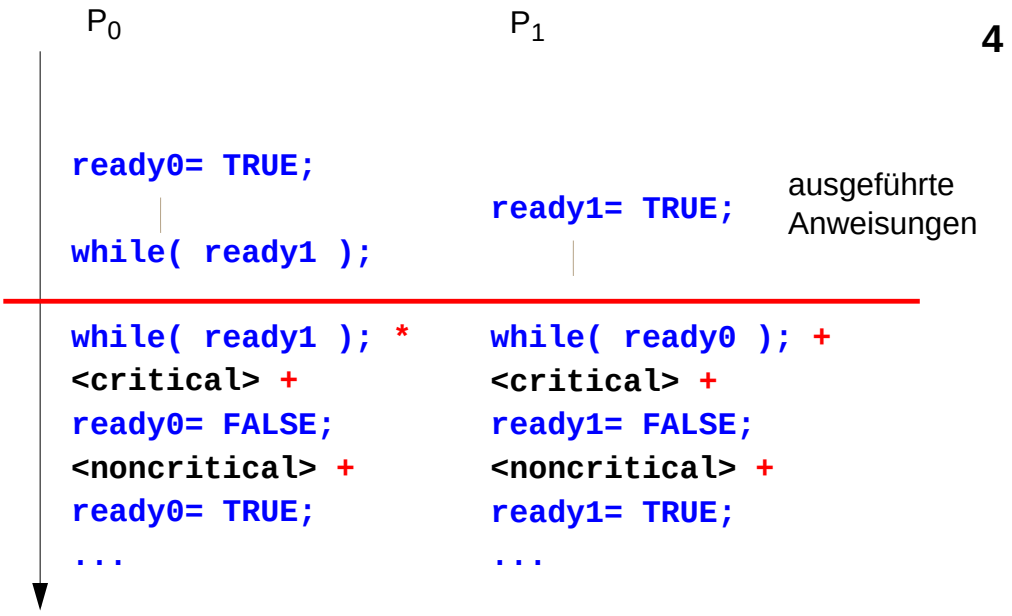
6.1 Gegenseitiger Ausschluss (7)

- Verklemmung



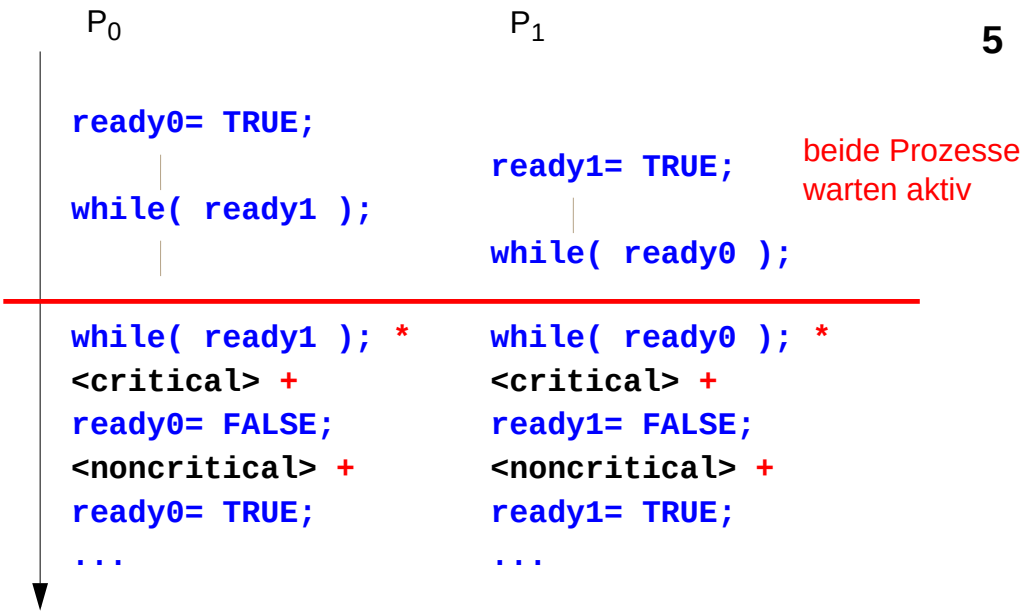
6.1 Gegenseitiger Ausschluss (7)

- Verklemmung



6.1 Gegenseitiger Ausschluss (7)

- Verklemmung *(Lifelock)*



6.1 Gegenseitiger Ausschluss (8)

- 3. Versuch (Algorithmus von Peterson, 1981)

```
bool ready0= FALSE;
bool ready1= FALSE;
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    turn= 1;
    while( ready1 &&
           turn == 1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    turn= 0;
    while( ready0 &&
           turn == 0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```

6.1 Gegenseitiger Ausschluss (9)

- Algorithmus implementiert gegenseitigen Ausschluss
  - vollständige und sichere Implementierung
  - **`turn`** entscheidet für den kritischen Fall von Versuch 2, welcher Prozess nun wirklich den kritischen Abschnitt betreten darf
  - in allen anderen Fällen ist **`turn`** unbedeutend
- Problem der Lösung
  - aktives Warten
- Algorithmus auch für mehrere Prozesse erweiterbar
  - Lösung ist relativ aufwendig

6.2 Spezielle Maschinenbefehle

- Spezielle Maschinenbefehle können die Programmierung kritischer Abschnitte unterstützen und vereinfachen
  - *Test-and-Set* Instruktion
  - *Swap* Instruktion
- Test-and-set
  - Maschinenbefehl mit folgender Wirkung

```
bool test_and_set( bool *plock )
{
    bool tmp= *plock;
    *plock= TRUE;
    return tmp;
}
```

  - Ausführung ist atomar

6.2 Spezielle Maschinenbefehle (2)

◆ Kritische Abschnitte mit Test-and-Set Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

6.2 Spezielle Maschinenbefehle (3)

■ Swap

◆ Maschinenbefehl mit folgender Wirkung

```
void swap( bool *ptr1, bool *ptr2)
{
    bool tmp= *ptr1;
    *ptr1= *ptr2;
    *ptr2= tmp;
}
```

◆ Ausführung ist atomar

6.2 Spezielle Maschinenbefehle (4)

- Kritische Abschnitte mit Swap-Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;
  ... /* uncritical */
}
```

Prozess 1

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;
  ... /* uncritical */
}
```

- ★ Code ist identisch und für mehr als zwei Prozesse geeignet

6.3 Kritik an den bisherigen Verfahren

- ★ Spinlock
  - ◆ bisherige Verfahren werden auch Spinlocks genannt
  - ◆ aktives Warten
- ▲ Problem des aktiven Wartens
  - ◆ Verbrauch von Rechenzeit ohne Nutzen
  - ◆ Behinderung „nützlicher“ Prozesse
  - ◆ Abhängigkeit von der Schedulingstrategie
    - nicht anwendbar bei nicht-verdrängenden Strategien
    - schlechte Effizienz bei langen Zeitscheiben
- Spinlocks kommen heute fast ausschließlich in Multiprozessorsystemen zum Einsatz
  - ◆ bei kurzen kritischen Abschnitten effizient
  - ◆ Koordinierung zwischen Prozessen von mehreren Prozessoren

6.4 Sperrung von Unterbrechungen

- Sperrung der Systemunterbrechungen im Betriebssystems

Prozess 0

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

Prozess 1

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

- ◆ nur für kurze Abschnitte geeignet
  - sonst Datenverluste möglich
- ◆ nur innerhalb des Betriebssystems möglich
  - privilegierter Modus nötig
- ◆ nur für Monoprozessoren anwendbar
  - bei Multiprozessoren arbeiten andere Prozesse echt parallel

6.5 Semaphor

- Ein Semaphor (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach *Dijkstra*)
  - ◆ P-Operation *(proberen; passeren; wait; down)*
    - wartet bis Zugang frei

```
void P( int *s )
{
    while( *s <= 0 );
    *s= *s-1;
}
```

atomare Funktion

  - ◆ V-Operation *(verhogen; vrijgeven; signal; up)*
    - macht Zugang für anderen Prozess frei

```
void V( int *s )
{
    *s= *s+1;
}
```

atomare Funktion

6.5 Semaphor (2)

- Implementierung kritischer Abschnitte mit einem Semaphor

```
int lock= 1;
```

Prozess 0

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

Prozess 1

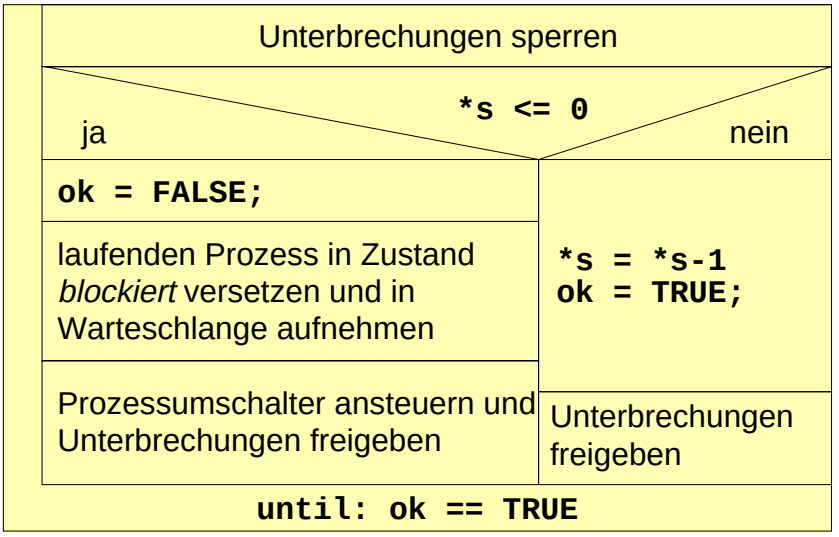
```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

- Problem:
  - Implementierung von P und V

6.5 Semaphor (3)

- Implementierung im Betriebssystem (Monoprozessor)

P-Operation



ok ist eine prozesslokale Variable

- jeder Semaphor besitzt Warteschlange, die blockierte Prozesse aufnimmt

6.5 Semaphor (4)

V-Operation

Unterbrechungen sperren
`*s = *s+1`
alle Prozess aus der Warteschlange in den Zustand *bereit* versetzen
Unterbrechungen freigeben
Prozessumschalter ansteuern

- Prozesse probieren immer wieder, die P-Operation erfolgreich abzuschließen
- Schedulingstrategie entscheidet über Reihenfolge und Fairness
  - leichte Ineffizienz durch Aufwecken aller Prozesse
  - mit Einbezug der Schedulingstrategie effizientere Implementierungen möglich

6.5 Semaphor (5)

★ Vorteile einer Semaphor-Implementierung im Betriebssystem
- Einbeziehen des Schedulers in die Semaphor-Operationen
- kein aktives Warten; Ausnutzen der Blockierzeit durch andere Prozesse

■ Implementierung einer Synchronisierung
- zwei Prozesse P_1 und P_2
- Anweisung S_1 in P_1 soll vor Anweisung S_2 in P_2 stattfinden

```
int lock= 0;
```

Prozess 1
```
...
S1;
V( &lock );
...
```

Prozess 2
```
...
P( &lock );
S2;
...
```

★ Zählende Semaphore

6.5 Semaphor (6)

- Abstrakte Beschreibung von zählenden Semaphoren (PV System)
  - ◆ für jede Operation wird eine Bedingung angegeben
    - falls Bedingung nicht erfüllt, wird die Operation blockiert
  - ◆ für den Fall, dass die Bedingung erfüllt wird, wird eine Anweisung definiert, die ausgeführt wird
- Beispiel: zählende Semaphore

Operation	Bedingung	Anweisung
P(S)	S > 0	S := S – 1
V(S)	TRUE	S := S + 1

7 Klassische Koordinierungsprobleme

- Reihe von bedeutenden Koordinierungsproblemen
  - ◆ Gegenseitiger Ausschluss *(Mutual exclusion)*
    - nur ein Prozess darf bestimmte Anweisungen ausführen
  - ◆ Puffer fester Größe *(Bounded buffers)*
    - Blockieren der lesenden und schreibenden Prozesse, falls Puffer leer oder voll
  - ◆ Leser-Schreiber-Problem *(Reader-writer problem)*
    - Leser können nebenläufig arbeiten; Schreiber darf nur alleine zugreifen
  - ◆ Philosophenproblem *(Dining-philosopher problem)*
    - im Kreis sitzende Philosophen benötigen das Besteck der Nachbarn zum Essen
  - ◆ Schlafende Friseure *(Sleeping-barber problem)*
    - Friseure schlafen solange keine Kunden da sind