Kapitel 4

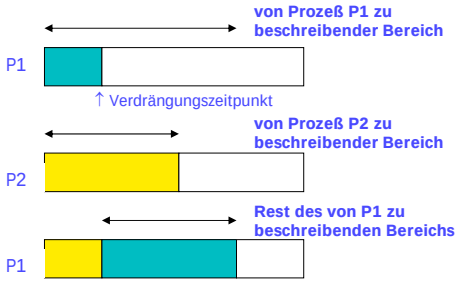
Interprozeßkommunikation und -synchronisation

Interprozeßkommunikation und -synchronisation Beispiel (1)

- **Beispiel: Betriebssystem, bei dem Prozesse direkt auf den Drucker ausgeben dürfen**
- **Zwei Prozesse P1 und P2 laufen gleichzeitig ab und drucken Daten aus**
  --> Vermischung der Daten auf dem Drucker
- **Lösungsansatz 1 "Drückampel"**
  - Der Prozeß, der zuerst "gedrückt" hat, bekommt grün und darf drucken
  - Der andere Prozeß drückt auch den Ampelknopf, bekommt aber rot und muß warten
  - --> Semaphore (~Zeichengeber, Ampel oder Signalmast in der Schiffahrt)
  - Nachteil: Ein Prozeß kann das Semaphor durch Nichtbenutzung mißachten
  - Ein Prozeß muß untätig warten, bis er an der Reihe ist
  - Wenn die Ampel nicht wieder freigegeben wird, muß der zweite Prozeß ewig warten
- **Lösungsansatz 2 "Exklusive Belegung"**
  - Betriebssystem bietet die Möglichkeit zur exklusiven Belegung des Druckers entsprechender Parameter beim Eröffnungsaufruf
  - Nachteil: Untätiges Warten des zweiten Prozesses
  - Abhilfe: Helferprozeß, welcher die zu druckenden Daten entgegennimmt

Interprozeßkommunikation und -synchronisation Beispiel (2)

- **Lösungsansatz 3**
  - Spezieller Druckprozeß belegt exklusiv den Drucker und richtet "Briefkasten" ein
  - Der Druckprozeß wartet am Briefkasten, bis ein Druckauftrag gekommen ist und wickelt ihn ab. Wenn keiner mehr vorliegt, wartet er am Briefkasten
  - Problemfälle (z.B. bei Realisierung durch gemeinsamen Speicher)
    - Nichtbeachten/Überschreiben von bereits vorhandenen Daten
    - Wiederholte Auftragsdurchführung durch den Druckprozeß
    - Inkonsistente Daten durch Unterbrechung während Kopiervorgangs
    - Zwei druckwillige Prozesse verdrängen einander während des Schreibvorgangs in den Briefkasten



Betriebssystem als Koordinierungs- und Buchhaltungsinstanz

- **Betriebssystem führt die Prozeßverwaltung durch und kann damit Prozeßwechsel "zur Unzeit" unterbinden.**
- **Interprozeßkommunikationsdienst**
  - Buchführung über eingegangene/abgeholte Aufträge und belegte Speicherbereiche
  - Aktivierung/Passivierung des Druckprozesses
  - Wartend-setzen eines druckwilligen Prozesses, wenn im Briefkasten kein Platz mehr für weitere Aufträge ist
  - dafür sorgen, daß ein Prozeß nicht verdrängt wird, wenn er gerade in den Briefkasten schreibt oder daraus liest
  - Trennung zwischen beschriebenen, aber noch nicht gelesenen Bereichen und wieder überschreibbaren Bereichen
- **Betriebssystem muß das Lesen aus dem Briefkasten und das Schreiben dort hinein zu unteilbaren oder atomaren Operationen machen**
  Dies geschieht durch Systemaufrufe, die unter Verdrängungssperre ablaufen
- **Hardware-mäßiger Speicherschutz**
  - Nur das Betriebssystem ist in der Lage, den Speicherschutz gezielt für den Speicherbereich des Briefkastens aufzuheben oder als "Briefträger" Daten aus einem prozeßeigenen Adreßraum in einen anderen zu kopieren.

Interprozeßkommunikation und -synchronisation

- **Enger Zusammenhang zwischen Interprozeßkommunikation und -synchronisation (IPKS)**
  - Synchronisation ist bei Kommunikation notwendig!
  - IPK erzeugt "Kosten" im Sinne von Speichermehraufwand und Rechenzeit
  - Zeitaufwand durch (eventuell mehrfaches) Umkopieren von Daten und durch Überschreiten der Adreßraumgrenzen der beteiligten Prozesse
- **Realisierungsaspekte:**
  - gemeinsame Speichersegmente für IPK bieten scheinbar Zeitvorteil. Andererseits muß hierbei der Zugriff explizit z. B. durch Semaphoroperationen koordiniert werden, was bei anderen IPK-Varianten automatisch erledigt wird.
  - Ähnliches gilt, wenn sich mehrere Subprozesse Daten innerhalb eines gemeinsamen Adreßraums teilen: die Kosten für den Adreßraumwechsel und eventuell für das Umkopieren entfallen, dafür muß auch hier der Zugriff explizit koordiniert werden.

Zeitkritische Abläufe

Prozeß 1	Prozeß 2
`x1 := a1 + b1;`	`x2 := a2 + b2;`
`x1 := a1 + b1;`	`x2 := a1 + b1;`
`a1 := a1 + b1;` `z1 := a1;`	`x2 := a1 + b1;` `z2 := x2;`
`a1 := a1 + b1;` `z1 := a1;`	`b1 := a1 + b1;` `z2 := b1;`

- **Wenn die Abläufe und Ergebnisse paralleler Prozesse von ihrer relativen Geschwindigkeit (d. h. von der Prozessorzuteilung) abhängen, heißen diese Abläufe zeitkritisch!**
- **Ein Teil im Ablauf eines Prozesses, in dem zeitkritische Vorgänge geschehen, heißt kritischer Bereich oder kritischer Abschnitt**

Beispiel "Lichtschranke"

Ein Objekt (Werkstück) durchläuft eine Lichtschranke und wird daraufhin gewogen. Die Variablen zaehler und gewichtssumme sind in einem beiden Prozessen zugänglichen gemeinsamen Speicher abgelegt. Der Berichterstatter-Prozeß gibt die Anzahl der innerhalb einer Minute aufgetretenen Ereignisse und das Durchschnittsgewicht aus.

Beobachter-Prozeß

```
wiederhole ewig
   warte auf Ereignis;

   miß gewicht;
   gewichtssumme := gewichtssumme + gewicht;
   zaehler := zaehler + 1;
```

Berichterstatter-Prozeß

```
wiederhole ewig
   warte 1 Minute
   falls zaehler <> 0
      berechne durchschnitt;
   write (zaehler, durchschnitt);

   zaehler:=0; gewichtssumme:=0;
```


Problemfälle beim Lichtschrankenbeispiel

- **Verdrängung des Beobachterprozesses durch den Berichterstatter-P.**
  - wenn das Ereignis bereits eingetreten, aber noch nicht gezählt ist:
    --> Ausgabe eines um 1 zu niedrigen Minutenwerts
  - Wenn gewichtssumme schon neu berechnet wurde:
    --> Ausgabe eines falschen Durchschnittsgewichts
    --> Zählerwert für nächste Minute um 1 zu hoch und auch hier Durchschnittsgewicht falsch bestimmt.
- **Ann.: "zaehler := zaehler + 1" besteht aus mehreren Maschinenbefehlen**
  - Wenn der Berichterstatter-Prozeß die Ausführung dieser Anweisung unterbricht:
    --> der Beobachter-Prozeß rechnet mit dem alten Wert für zaehler weiter
    --> das Nullsetzen von zaehler geht verloren. Dto. für gewichtssumme.
- **Viel häufigere Ausführung des Berichterstatter-Prozesses:**
  - der Zähler wird immer wieder gelöscht, bevor er inkrementiert werden kann
    --> viele überflüssige Nullen werden als Zähler ausgegeben
- **Verdrängung des Berichterstatterprozesses nach dem "write"**
  - --> Dann geht das letzte beobachtete Ereignis ganz verloren

Möglichkeiten des gegenseitigen Ausschlusses mutual exclusion, MUTEX

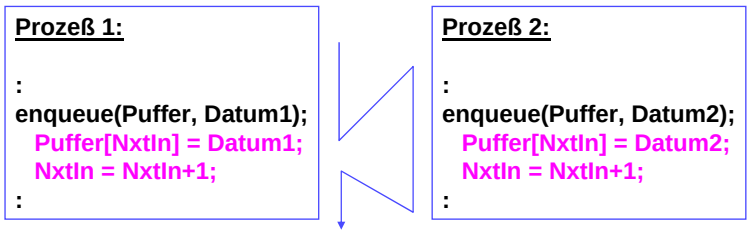
- **Unterscheidung zweier Formen von kritischen Abschnitten, nämlich "kurze" und "lange".**

Lösungsansätze für gegenseitigen Ausschluß:

- **Prozeßwechselsperre**
  - auch unbeteiligte dritte Prozesse sind davon betroffen
  - Möglichkeit zu "vergessen", die Prozeßwechselsperre wieder aufzuheben
  - Schließlich ist ihre Anwendung auf den Fall "kurzer" kritischer Abschnitte zu beschränken, da sonst das gesamte System blockiert wird.
  - Vorteil: Schnell und sicherer Zugriffsschutz
- **Suspendieren**
  - suspendieren aller anderen "zugriffsverdächtigen" Prozesse
  - der suspendierende Prozeß muß alle anderen Prozesse kennen
  - fehlerträchtig, da ggf. bei Programmerweiterungen neue Prozesse hinzukommen
- **Spin-locks**
- **Semaphore**

Beispiel für kritischen Bereich

- **z.B.: Puffer endlicher Größe**
  Pufferzeiger: NxtIn



Konsequenzen: **Datum1 wurde überschrieben; d.h. ist verloren!**
Puffer[NxtIn-1] ist undefiniert!

Peterson, 1985

```
enter_region(process = 0)
{
    interested[process] = true;
    turn = process = 0;

    while   (turn == process &&
             interested[1-process])
            ; /* warte */
    return;
}
```

```
enter_region(process = 1)
{
    interested[process] = true;
    turn = process = 1;

    while   (turn == process &&
             interested[1-process])
            ; /* warte */
    return;
}
```

Löst das Problem des gegenseitigen Ausschlusses
ohne Sperren von Unterbrechungen!

Gegenseitiger Ausschluß durch Spin-locks: TSL-Instruktion

- **Unteilbarer Maschinenbefehl Test and Set Lock**

```
wait    tsl     reg, Flag
        cmp     reg, #0
        jnz     wait
        ret
```

kopiert den alten Wert des Flags in ein Register und setzt das Flag auf 1, wenn das Flag vorher Null war

Lesen und Schreiben sind bei TSL auch bei Mehrprozessorbetrieb noch unteilbar!

Flag = 0: Keine Sperre --
niemand im kritischen Bereich

Flag = 1: Sperre / Lock gesetzt --
Warten auf das Ende der Sperre
erforderlich

Annähernd äquivalente Maschinenbefehlsfolge:

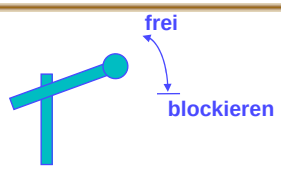
```
        DI
        LD      reg, Flag
        CMP     reg, #0
        JNZ     Ende
        ST      Flag, #1
Ende    EI
```

Aktives Warten

- **Beispiel TSL:**
  - (P2) hat kritischen Bereich betreten
  - (P1) wartet ... bis zur Unterbrechung durch den Scheduler
  - (P2) bearbeitet kritischen Bereich
  - (P1) wartet immer noch ...
  - :
  - (P2) verläßt kritischen Bereich
  - (P1) verläßt nach Neuaktivierung die Warteschleife
- **Rechenzeitvergeudung; Nur (P2) kann die Sperre aufheben!**
- **Besser:**
  - (P1) erkennt Sperre und wird stillgelegt
  - (P2) bearbeitet kritischen Bereich
weckt schlafenden (P1) nach Verlassen des
kritischen Bereichs wieder auf

Semaphore, E.W. Dijkstra (gr. Zeichenträger)

```
typedef struct semaphore {
        unsigned   Zähler;
        Tqueue     PQueue;
}
```




```
P(S) / DOWN(S) :
   disable interrupts;
   if (S.Zähler > 0)
      S.Zähler--;
   else {
      Prozeßstatus = "Blockiert";
      Füge ihn S.PQueue hinzu;
      reschedule;
   }
   enable interrupts;
   return;
```

```
V(S) / UP(S) :
   disable interrupts;
   if (S.Zähler == 0 &&
        S.PQueue ist nicht leer)  {
      Wähle Prozeß aus S.PQueue aus;
      Prozeß.Status = "Bereit";
   }
   else
      S.Zähler ++;
   enable interrupts;
   return;
```

P(mutex) | **kritischer Bereich** | **V(mutex)**

```
P(mutex)
   if (mutex > 0) mutex - -;
   else {
      Block(P);
      AddToQueue(P);
      Reschedule; }
```

```
Puffer[NxtIn] = Datum;
```

```
NxtIn = NEXT(NxtIn);
```

```
V(mutex)
    if (mutex == 0 &&
         NOT EMPTY(mutex.PQueue) {
         Dequeue(mutex.PQueue, P);
         Activate(P);  }
    else
         mutex.zähler ++;
```

Prozeß 1	Prozeß 2
mutex = 0	
Puffer ...	Block(P2); AddToQueue... Reschedule ...
NxtIn ...	
Dequeue(P2); Activate(P2);	
mutex = 0	Puffer ... NxtIn ...
	mutex = 1

Beispiel: Leser / Schreiber - Problem

- viele Leser - ein Schreiber
- Zur Zeit des Schreibens darf keiner lesen

```
semaphore   mutex, db;
int         nLeser;
```

```
Schreiber( )
{
  P(db);
  Schreibe_in_die_Datenbank( )
  V(db);
}
```

```
Leser( )
{
  while (TRUE) {
      P(mutex);
      nLeser ++;
      if (nLeser == 1)  P(db);
      V(mutex);
      Datenbank_lesen( );
      P(mutex);
      nLeser - -;
      if (nLeser == 0)  V(db);
      V(mutex);
      Verarbeitung_der_Daten( );
  }
}
```

Weitere Verwendungsmöglichkeiten von Semaphoren

- **Init = 1** — **Wechselseitiger Ausschluß**
  P(S) kritischer Bereich V(S)
- **Init = N** — **Zählende Semaphore**
  Bis zu N verschiedene Prozesse dürfen in den kritischen Bereich eintreten
  (z.B. N=4: Nutzung der vier verfügbaren Bandgeräte)
- **Init = 0** — **Blockierende Semaphore**
  dient zur Prozeßsynchronisation:

Prozeß 1:	Prozeß 2:
:	:
{A}	P(S)
V(S)	{B}
:	:

Gefahren beim Einsatz von Semaphoren

- **Ausschlußreihenfolge ist entscheidend!**

Prozeß 1:

```
P(Bandstation);
P(Drucker);
Drucke_die_Daten( );
V(Drucker);
V(Bandstation);
```

Prozeß 2:

```
P(Drucker);
P(Bandstation);
Drucke_die_Daten( );
V(Bandstation);
V(Drucker);
```

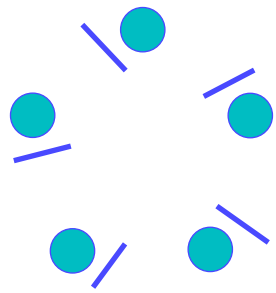
!!! Verklemmung (Dead Lock) !!!

muß (vom Betriebssystem) erkannt und behandelt werden

Semaphore bei Ein- bzw. Mehrprozessorsystemen

- **Einprozessorsystem:**
  - aktives Warten vergeudet Rechenzeit
  - nur der andere Prozeß kann Sperre aufheben
  - Prozeßwechsel ist erforderlich ⇨ Semaphore
- **Mehrprozessorsystem:**
  - Prozeßwechsel ist NICHT erforderlich!
  - Prozeßwechsel kostet Zeit!
  - In Abhängigkeit von der Komplexität des kritischen Bereichs kann aktives Warten oder Taskwechsel besser sein!
  - Zugriff auf die Semaphorvariable muß geschützt werden

Denksportaufgabe: Speisende Philosophen, Dijkstra, 1965



- **Philosophen:**
  - Denken oder Essen
  - Zwei Stäbchen sind zum Essen erforderlich
- **!Achtung!**
  - Verhungerung ausschließen
  - max. Auslastung des Betriebsmittels 'Stäbchen' ist wünschenswert

Erzeuger/Verbraucher - Problem

```
TYPE   NT = (* Typ der Nachricht *);

VAR
   Puffer: ARRAY [0..L-1] OF NT;
   Schreibzeiger, Lesezeiger: 0 .. L-1;
   (* Puffer liegt im gemeinsamen Speicher;
      Schreib- und Lesezeiger dto. *)
   frei, belegt: Semaphor;

PROCEDURE sende (Nachricht: NT);
BEGIN
   P(frei);       (* anfangs frei.Z = L *)
      Puffer [Schreibzeiger] := Nachricht;
      Schreibzeiger := (Schreibzeiger + 1) MOD L;
   V(belegt);     (* anfangs belegt.Z = 0 *)
END;   (* sende *)

PROCEDURE empfange (VAR Nachricht: NT);
BEGIN
   P(belegt); (* anfangs belegt.Z = 0 *)
      Nachricht := Puffer [Lesezeiger];
      Lesezeiger := (Lesezeiger + 1) MOD L;
   V(frei);     (* anfangs frei.Z = L *)
END;   (* empfange *)
```

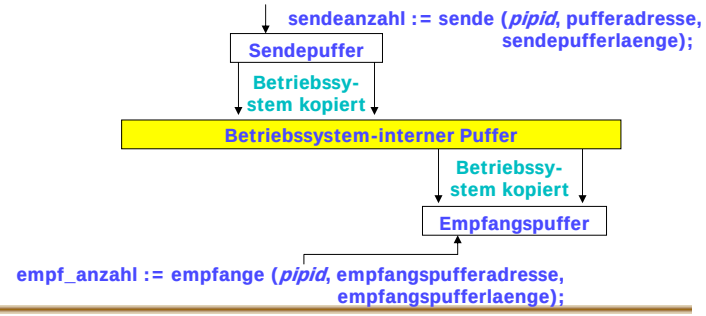
```
PROCESS Erzeuger;
VAR
   NR: NT;
BEGIN (* Erzeuger *)
   REPEAT
      erzeuge Nachricht NR;
      sende (NR);
   UNTIL ewig;
END;  (* Erzeuger *)

PROCESS Verbraucher;
VAR
   NR: NT;
BEGIN (* Verbraucher *)
   REPEAT
      empfange (NR);
      verarbeite Nachricht NR;
   UNTIL ewig;
END;  (* Verbraucher *)

BEGIN    (* Initialisierung *)
   ...
END.
```


Interprozeßkommunikation mit Pipelines

- Naheliegend: BS soll die Funktionen "sende" und "empfange" als Systemaufrufe anbieten
- --> Betriebsmitteltyp Pipe(line), verbunden mit den atomaren BS-Operationen
- BM Pipeline beinhaltet einen betriebssystemeigenen Puffer, der in der Länge definierbar ist und als unstrukturierte Bytefolge behandelt wird.
- Betriebssystem übernimmt das Kopieren der Daten vom Sende- in den Empfangspuffer

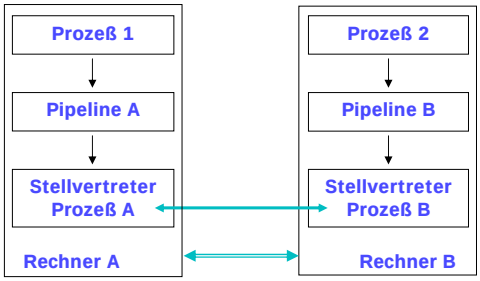


Betriebsmitteltyp "Pipeline"

- **Operationen**
  - Einrichten (create) einer Pipeline (gegebenenfalls mit Längenangabe).
    Ergebnis: Identifikator für die Pipeline (pipid) als Referenz für übrige Aufrufe.
  - Löschen (delete) einer Pipeline.
  - Senden von Daten (write) aus einem Puffer. Rückgabewert: Anzahl Bytes
  - Empfangen von Daten (read) in einen Puffer. Rückgabewert: Anzahl Bytes
- **Unterscheidung: benannte und unbenannte Pipelines**
  - Unbenannte Pipelines: nur an Sohnprozesse weitervererbbar
- **Informationsmenge in der Pipeline als Bytefolge 'strukturiert'**
  - K. Prozesse müssen sich auf gem. Nachrichtenformat einigen
  - Eine Sendeoperation darf nur ganz oder garnicht ausgeführt werden
  - Bei Empfangsoperationen ist partielles Lesen (bedingt) erlaubt
  - Problemfälle: Prozeß blockieren oder Routine mit Fehlerstatus beenden
- **Prozeßbeendigung**
  - Verschiedene Formen der Behandlung/Information der Kommunikationspartner

Pipelines vs. gemeinsamer Speicher

- **Pipelines: Abstraktion der darunterliegenden Implementierung**
  - Höherer Komfort als bei Kom. über gemeinsamen Speicher+Semaphore
  - Kommunikation auch über Rechnergrenzen hinweg möglich:
    Stellvertreterprozesse + Kommunikationsdienste



Anwendungsbeispiel: COM

Prozeßsynchronisationsformen

- **Suspendieren von Prozessen**
  - erfordert Kenntnis aller anderen synchronisationswilligen Prozesse
- **Prozeßwechselsperren**
  - behindert auch nicht synchronisationswillige Prozesse
- **Semaphore**
- **Monitore**
  - Sprachkonzept, welches von der Programmiersprache unterstützt werden muß
  - Unterprogramm, das immer nur von einem Prozeß aufgerufen werden kann
  - Andere Prozesse müssen warten, bis der Monitor wieder verlassen wurde
- **Ereignisflaggen (event flags)**
  - Anzeigen des Eintritts eines Ereignisses
  - Prozeß kann warten (blockiert) oder andere Aufgaben erledigen
- **Signale**
  - Reagieren auf asynchron eintreffende Ereignisse (Unterprog./Warten/Ignorieren)

Interprozeßkommunikationsformen

- botschaftsorientiert (+) vs. datenstromorientiert (-)
- für die Kommunikation zwischen Rechnern geeignet?
- Implizite Synchronisation des Datenzugriffs? (Blockieren?)

gemeinsamer Speicher (Vorteil: schneller Zugriff)	(- , - , -)
normale Dateien (Vorteil: Persistenz)	(- , - , -)
hauptspeicherresidente Dateien (memory mapped files)	(- , - , -)
Pipelines	(- ,+,+)
Briefkästen und Botschaftswarteschlangen	(+,+,+)
Rendezvous	(+,+,+)
Fernaufruf (RPC, remote procedure call)	(+,+,+)

- (Warten an der Datenübergabestelle im Code (Systemaufruf))
- Auch der Sender muß grundsätzlich warten