IPC

INTERPROCESS COMMUNICATION

Beispiele

producer–consumer problem, PCP

Philosophenproblem

Raucherproblem

Friseur

…

Überblick Curriculum

1. Programme, Prozeduren, Prozesse und Instanzen
2. Prozeßzustände
3. Prozeßhierarchie und Prozeßrechte
4. Prozeß-Operationen des BS
5. **Prozeß-Synchronisation**
6. Prozeß-Kommunikation
7. Prozeß-Scheduling
8. Beispiele

Prozeß-Synchronisation

- Solche Situationen (Bsp. letzte Folie) heißen **zeitkritsche Abläufe**.

- Programmabschnitte, die auf gemeinsam benutzte Daten zugreifen, heißen **kritische Abschnitte.**

Prozeß-Synchronisation

Fehler in zeitkritischen Abläufen sind sehr schwer erkennbar, da sie nur sehr selten auftreten.

Sie können nur durch wechselseitigen Ausschluss vermieden werden.

Wo treten Fehler in zeitkritische Abläufen auf?

- Kritische Abschnitte treten überall auf, wo zwei oder mehr Prozesse auf einem Computer oder in einem Netz **um mindestens eine Ressource konkurrieren**, **die sie schreibend benutzen wollen.**

-> Das trifft besonders auf Datenbankserver zu (z. B. Reservierungssysteme, zentrale Karteien, usw.).

Vier Bedingungen für eine gute Lösung (nach Tanenbaum):

1. Höchstens ein Prozeß darf sich in einem kritischen Abschnitt aufhalten. (Korrektheit)

2. Es dürfen keine Annahmen über Ausführungsgeschwindigkeit und Anzahl der Prozessoren gemacht werden.

3. Kein Prozeß, der sich in einem kritischen Abschnitt befindet, darf andere blockieren.

4. Kein Prozeß soll unendlich lange warten müssen, bis er in einen kritischen Bereich eintreten darf.

- **Die letzten beiden Punkte dienen der Stabilität, sie sollen Prozeßverklemmungen verhindern.**

Bsp. für zeitkritischen Ablauf

Das Problem des schlafenden Friseurs:

- Ein Friseur bedient, sobald er Zeit dafür hat, ankommende oder wartende Kunden.
- Der Warteraum ist beschränkt auf N Stühle.
- Der Friseur (= Prozessor) startet zu Arbeitsbeginn die Funktion barbier(). Wenn kein Kunde da ist, legt er sich schlafen.
- Kommt ein Kunde, prüft er die Bedingung count >= N. Sind alle Stühle belegt, geht er wieder. Sonst bleibt er und erhöht count um eins.
- War count vorher 0, weckt er den schlafenden Friseur.
- Wenn der Friseur einen Kunden bedient hat, dekrementiert er count. Ist count dann 0 geworden, legt er sich wieder schlafen.

Die kritische Situation besteht darin, daß der Kunde zwischen der Dekrementierung von count und der Frage "Ist count gleich 0" ankommt und den Friseur wecken kann, obwohl der sich noch gar nicht schlafen gelegt hat.

Teamarbeit:

Finden Sie weitere Beispiele.

Lösungsversuche für das Problem der kritischen Abschnitte

Lösungsversuche für das Problem der kritischen Abschnitte

1. Einfachste Lösung: Sperren
2. Verfahren mit gegenseitigem Ausschluß
3. Semaphore
4. Ereigniszähler

Lösungsversuche für das Problem der kritischen Abschnitte

1. ***Einfachste Lösung: Sperren***

- Vor Eintritt in den kritischen Bereich alle Interrupts sperren und sie nach Verlassen des kritischen Bereichs wieder freigeben. Damit kann der Scheduler nicht während des kritischen Abschnitts den Prozeß unterbrechen.
- **Nachteil:** Kein verdrängendes Multitasking mehr. Der Anwenderprozeß kann den Scheduler blockieren (gewollt oder ungewollt durch einen Programmfehler).

Ist der Begriff Interrupt klar?

-> …

Exkurs: Interrupt

Wieso gibt es Interrupts?

Sinn eines Interrupts ist es,

auf Ein-/Ausgabe-Ereignisse (Signale) (z. B. von Tastatur, Maus, Festplatte, Netzwerk, Zeitgeber/Timer usw.) **sofort reagieren zu können**,

während ein anderer Programmcode (z. B. von Anwendungsprogrammen) abgearbeitet wird.

Exkurs: Interrupt x86-Architektur

Alle Intel-Prozessoren haben einen Interrupt-Signaleingang für Interrupts.

Um mehrere Interruptquellen anschließen zu können, gibt es einen eigenen Interrupt-Controller-Baustein (z. B. den Programmable Interrupt Controller (PIC)), der mehrere Interrupt-Eingänge besitzt und zu einem Signal zusammenführt.

Außerdem ist er über interne Register konfigurierbar, sodass er je nach ausgelöstem Interrupt im CPU-Interruptzyklus verschiedene, vorgegebene Interruptvektoren auf den Bus legt, die die CPU dann einliest. Bei neueren Prozessoren sind all diese Funktionalitäten mit in den Kern des Hauptprozessors integriert.

Bei x86-Prozessoren sind 256 verschiedene Interruptvektoren möglich. Der Interruptvektor wird im Interruptzyklus des Prozessors als 8-Bit-Wert vom Datenbus gelesen.

Lösungsversuche für das Problem der kritischen Abschnitte

2. Verfahren mit gegenseitigem Ausschluß

Der Programmabschnitt, in dem ein Zugriff zu dem nur exklusiv nutzbaren Betriebsmittel (z.B. die gemeinsamen Daten) erfolgt, wird kritischer Abschnitt genannt. *Es muß verhindert werden, daß sich zwei Prozesse gleichzeitig in ihren kritischen Abschnitten befinden*.

-> Sperrvariable

-> Aktives Warten

-> Striktes Alternieren

Sperrvariable:

Es wird eine logische Variable geführt, die mit 0 den Eintritt in den kritischen Bereich erlaubt und mit 1 sperrt.

Der in den kritischen Bereich eintretende Prozeß muß dann vor seinem Eintritt prüfen, ob der Bereich frei ist, die Sperrvariable auf 1 setzen und nach Ende wieder freigeben.

Der Abschnitt von der Prüfung der Sperrvariablen bis zu ihrem Setzen ist selbst ein kritischer Abschnitt!

Er ist zwar kürzer und damit die Konfliktwahrscheinlichkeit geringer, aber die Gefahr des Konfliktes ist nicht beseitigt!

Es gibt bei vielen CPUs jedoch Befehle von der Form "teste und setze", bei denen der kritische Abschnitt innerhalb eines Maschinenbefehls "abgehandelt" wird.

Damit sind folgende Lösungen möglich:

Aktives Warten

```
bool v;
/* Warteschleife, bis Riegelvariable RV[i]= 0 ist */
do
    v = test_and_set(&RV[i]);
while (v == 0);
...
/* kritischer Abschnitt zur Datenmenge i */
...
RV[i] : = 1;
```

Die Ver- bzw. Entriegelung wird häufig mit den Funktionen LOCK und UNLOCK formuliert:

```
L0CK(RV[i]);
...
/* kritischer Abschnitt zur Datenmenge i */
...
UNLOCK(RV[i]);
```

Viele Computer, die für Mehrprozessorbetrieb konzipiert wurden, kennen den Maschinenbefehl "Test and Set Lock". Dabei wird ein Speicherwort aus einem Register gelesen und ein Wert ungleich Null hineingeschrieben. Für die TSL-Operation wird eine gemeinsame Variable namens "Flag" verwendet. Diese koordiniert den Zugriff.

Beispiel:

enter_region: tsl register,flag kopiere flag ins Register und setzte flag auf 1
cmp register,#0 war flag Null?
jnz enter_region wenn nicht, Verriegelung gesetzt, Schleife

... kritischer Bereich ...

leave_region: mov flag,#0 flag auf 0 setzten
ret Rückkehr zum Aufrufer

Aktives Warten verbraucht CPU-Zeit durch Warteschleifen.

Striktes Alternieren

Nur zwei Prozesse erlauben sich wechselweise den Eintritt in den kritischen Abschnitt mit einer logischen Sperrvariablen.

Zum Beispiel dient die gemeinsame Sperrvariable turn zur Synchronisiation zweier Prozesse.

Es gilt: turn = i (i = 0,1) --> Prozeß Pi darf in den kritischen Bereich eintreten:

PO:

```
while (1)
{
while (turn != 0) no_operation; /* warten */
kritischer Bereich;
turn = 1;
unkritischer Bereich;
}
```

P1:

```
while (1)
{
while (turn != 1) no_operation; /* warten */
kritischer Bereich;
turn = 0;
unkritischer Bereich;
}
```

Auch hier wird CPU-Zeit durch Warteschleifen verbraucht.

Außerdem ist striktes Alternieren auch keine gute Lösung, wenn ein Prozeß wesentlich langsamer ist als der andere.

Lösungsversuche für das Problem der kritischen Abschnitte

3. Semaphore

- Bisher haben die Prozesse ihren Eintritt in den kritischen Abschnitt selbst gesteuert. Die folgenden Techniken erlauben es dem **Betriebssystem, Prozessen den Zutritt zum kritischen Abschnitt zu verwehren.**
- Im Zusammenhang mit Algol 68 entwickelte Dijkstra das Prinzip der Arbeit mit Semaphoren. **Für jede zu schützende Datenmenge wird eine Variable (Semaphor) eingeführt** (binäre Variable oder nicht-negativer Zähler).

- **Ein Semaphor (Sperrvariable, Steuervariable)** signalisiert einen Zustand (Belegungszustand, Eintritt eines Ereignisses) und gibt in Abhängigkeit von diesem Zustand den weiteren Prozeßablauf frei oder versetzt den betreffenden Prozeß in den Wartezustand.

Exkurs: Woher kommt Semaphor?

mt S

Formsignale (auch *Semaphore*) sind mechanische Eisenbahnsignale.

Zwei Zustände möglich:

- Halt
- Fahrt

Lösungsversuche für das Problem der kritischen Abschnitte

4. Ereigniszähler

Für einen Ereigniszähler E sind drei Operationen definiert:

read(E): Stelle Wert von E fest!
advance(E): Inkrementiere E (atomare Operation)
await(E,v): Warte bis E = v ist

Damit kann E nur wachsen, nicht kleiner werden!
E sollte mit 0 initialisiert werden.

```
#define N 100 /* Puffergröße */
eventcounter inputcounter = 0, outputcounter = 0; /* Ereigniszaehler */
int inputsequence = 0, outputsequence = 0;

producer()
    {
    tinhalt item;
    while (1)
        {
        produce(&item);
        inputsequence = inputsequence + 1;
        await(outputcounter,inputsequence-slots);
        buffer[(inputsequence - 1) % N] = item;
        advance(inputcounter);
        }
    }

consumer()
    {
    tinhalt item;
    while (1)
        {
        outputsequence = outputsequence + 1;
        await(inputcounter,outputsequence);
        item = buffer[(outputsequence - 1] % N);
        advance(outputcounter);
        consume(&item);
        }
    }
```

Anmerkung: Mit % wird der Modulo-Operator gekennzeichnet (= Rest der ganzzahligen Division).

Lösungsversuche für das Problem der kritischen Abschnitte

Ereigniszähler

- Die Ereignis-Zähler werden nur erhöht, nie erniedrigt.
- Sie beginnen immer bei 0.
- In unserem Beispiel werden zwei Ereignis-Zähler verwendet:
  - inputcounter zählt die Anzahl der produzierten Elemente seit dem Start.
  - outputcounter zählt die Anzahl der konsumierten Elemente seit dem Start.

Lösungsversuche für das Problem der kritischen Abschnitte

Ereigniszähler

Aus diesem Grunde muß immer gelten: outputcounter <= inputcounter.

Sobald der Produzent ein neues Element erzeugt hat, prüft er mit Hilfe des await-Systemaufrufs, ob noch Platz im Puffer vorhanden ist.

Zu Beginn ist outputcounter = 0 und (inputsequence - N) negativ - somit wird der Produzent nicht blockiert. Falls es dem Produzenten gelingt N+1 Elemente zu erzeugen, bevor der Konsument startet, muß er warten, bis outputcounter = 1 ist. Dies tritt ein, wenn der Verbraucher ein Element konsumiert.

Die Logik des Konsumenten ist noch einfacher. Bevor das m-te Element konsumiert werden kann, muß mit await(inputcounter,n) auf das Erzeugen des m-ten Elementes gewartet werden.

Monitore

- Sind higl-level Kontrukte, die auf Semaphoren basieren, aber einfacher zu verwenden sind

- Werden von einer höheren Programmiersprache zur Verfüging gestellt.
  - Concurrent Pascal, Modula-3, C++, Java...

Monitor - Wirkungsweise

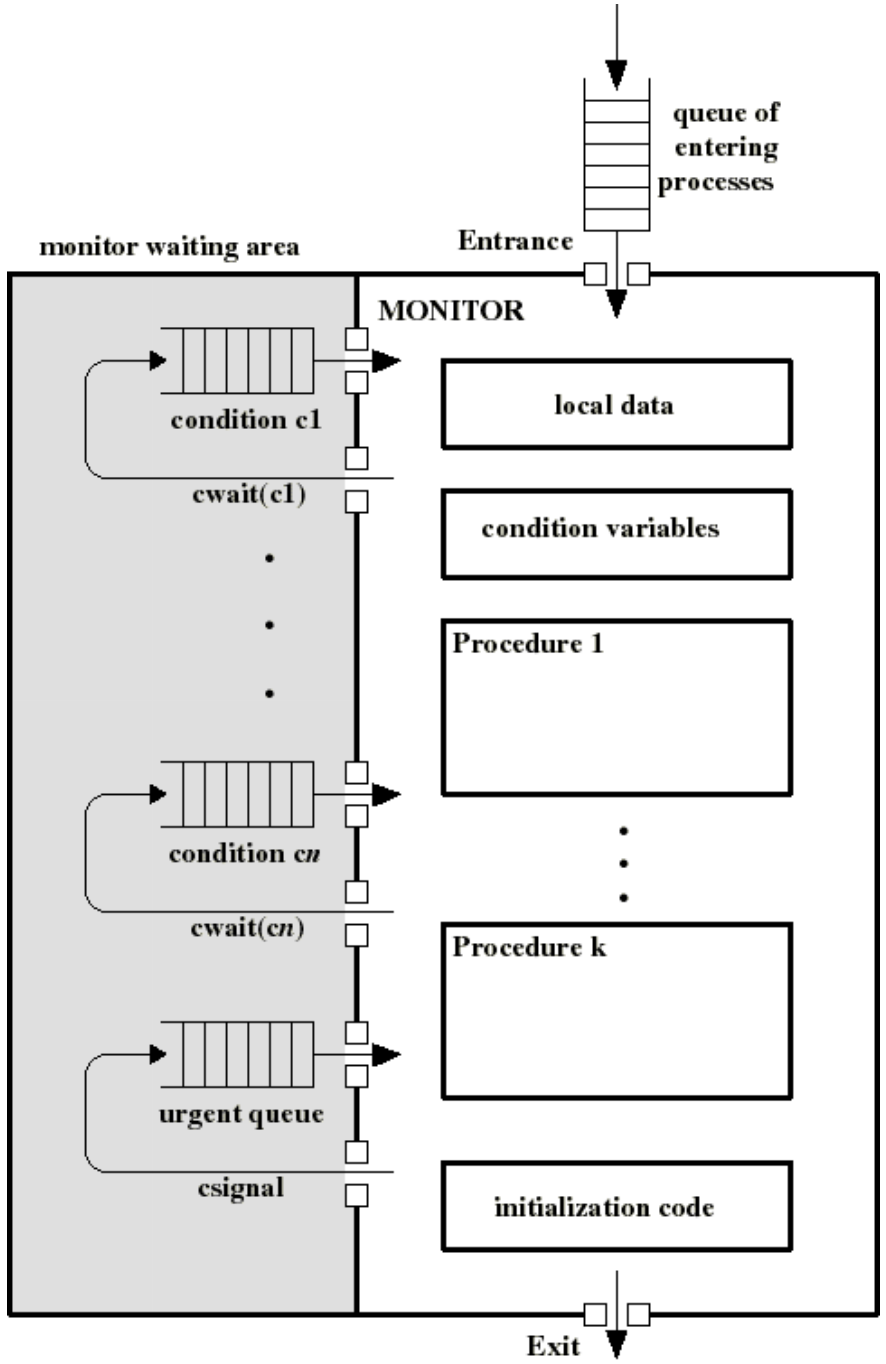
- SW-Module, bestehend aus
  - Prozeduren
  - Initialisierungssequenz
  - Lokale Variablen
- Eigenschaft:
  - Lokale Variable nur durch Prozeduren verwendbar
  - Ein Programm markiert eine CS durch Angabe eines Schlüsselwortes (Bsp: Java – synchronize)

Bedingungsvariable

- Monitor - lokale Variable
- Zugriff nur durch 2 Funktionen
  - cwait(a): blokiert rufenden Prozess bei Bedingung (variable) a
  - csignal(a): weckt einen andern Prozess bei Bedingung (variable) a.
    - Wenn mehrere Prozesse blockiert->Zufall
    - Wenn kein Prozess blockiert->keine Wirkung

Monitor

- Nur ein Prozess kann im Monitor aktiv sein
- Wartendende Prozesse befinden sich in Eingangs- oder Conditionqueue
- Ein Prozess stellt sich selbst in Cond.-queue durch cwait(cn)
- csignal(cn) stellt 1. Prozess aus Cond.-queue in den Monitor und sich selbst in urgent queue. Diese hat höhere Prio als andere Cond.-queues



Producer/Consumer Problem

- 2 Prozesse:
  - producer
  - consumer
- Synchronization nun durch Monitore
- append(.) und take(.) befinden sich innerhalb des Monitors
- Nur durch append(.) und take(.) kann auf den Puffer zugegriffen werden.

```
Producer:
repeat
  produce v;
  append(v);
forever

Consumer:
repeat
  take(v);
  consume v;
forever
```

Monitor für das P/C Problem

- Innerhalb des Monitors sind:
  - buffer: array[0..k-1] of items;
- 2 Bedingungsvariable:
  - notfull: true wenn Puffer voll
  - notemty: true wenn Puffer nicht leer
- Pointer und Zähler:
  - nextin: Nächstes zu schreibendes Elem
  - nextout: Nächstes zu lesendens Elem
  - count: Anzahl der Elemente im Puffer

Monitor für das P/C Problem

```
Monitor boundedbuffer:
  buffer: array[0..k-1] of items;
  nextin, nextout, count: integer;
  notfull, notempty: condition;

  append(v):
    while (count=k) cwait(notfull);
    buffer[nextin]:= v;
    nextin:= nextin+1 mod k;
    count++;
    csignal(notempty);

  take(v):
    while (count=0) cwait(notempty);
    v:= buffer[nextout];
    nextout:= nextout+1 mod k;
    count--;
    csignal(notfull);
```

Nachrichtenaustausch

- Dient zur Interprozesskommunikation (IPC)
  - Innerhalb eines Systems
  - Für verteilte Systeme
- Sind ein weiteres Mittel zur Prozesssynchronisation
- 2 Funktionen:
  - send(destination, message)
  - received(source, message)
- In beiden Fällen kann der Prozess blockieren oder nicht

Nachrichtenaustausch

- Sender: wird nicht blockiert nach send(.,.)
  - Kann mehrere Nachrichten an mehrere Empfänger senden.
  - Wartet aber meist auf ACK des Empfängers
- Empfänger: kann nach blockert werden nach Aufruf von receive(.,.)
  - Der Empfänger benötigt Daten erst nach receive

Nachrichtenaustausch

- Weitere Möglichkeiten zur Synchronisation
- Bsp: blocking send, blocking receive:
  - Beide werden blockiert, wenn Message empfangen
  - Nur bei ungepuffertem Kanal)

Welche Dienste stellt das OS zur Verfügung?

- **Prozessmanagement**
  einen Prozess kann man sich als „Programm in Ausführung“ vorstellen
- **Hauptspeicherverwaltung**
  ist ein grosses Lineares Feld (Array) von Wörtern bzw. Bytes
- **Dateiverwaltung**
- **Benutzerschnittstelle**
- Netzwerkanbindung

Überblick Curriculum

1. Programme, Prozeduren, Prozesse und Instanzen
2. Prozeßzustände
3. Prozeßhierarchie und Prozeßrechte
4. Prozeß-Operationen des BS
5. Prozeß-Synchronisation
6. **Prozeß-Kommunikation**
7. Prozeß-Scheduling
8. Beispiele

Prozeß-Kommunikation

- Bei Multitasking-Betriebssystemen spielt die Kommunikation bzw. Synchronisation zwischen den quasiparallel ablaufenden Prozessen eine herausragende Rolle.
- Die Gründe dafür sind vielfältig.
  - Zum einen werden größere Softwaresysteme häufig als Systeme mit mehreren kooperierenden Prozessen gestaltet. Diese müssen normalerweise in ihren Abläufen synchronisiert werden.
  - Ferner müssen häufig Daten von einem Prozeß zum anderen transferiert werden. Ein anderer Grund liegt im Problem der kritischen Abschnitte von Prozessen beim Zugriff auf nicht gemeinsam benutzbare Betriebsmittel.
- Auch hier sind Synchronisationsmethoden erforderlich, die den gegenseitigen Ausschluß gewährleisten (siehe oben: Semaphore).

Prozess - Kommunikation

Einige Möglichkeiten der Prozess - Kommunikation (Interprocess Communication (IPC)) sind:

- Kommunikation über **gemeinsame Speicherbereiche** Prozesse können gemeinsame Datenbereiche, Variablen etc. anlegen und gemeinsam nutzen.

- Kommunikation über **gemeinsame Dateien** Prozesse schreiben in Dateien, die von anderen Prozessen gelesen werden.

Prozess - Kommunikation

Einige Möglichkeiten der Prozess - Kommunikation (Interprocess Communication (IPC)) sind:

- **Kommunikation über Pipes**

- Dies sind unidirektionale Datenkanäle zwischen zwei Prozessen. Ein Prozeß schreibt Daten in den Kanal (Anfügen am Ende) und ein anderer Prozeß liest die Daten in der gleichen Reihenfolge wieder aus (Entnahme am Anfang).
- Realisierung im Speicher oder als Dateien.
- Lebensdauer in der Regel solange beide Prozesse existieren.

Prozess - Kommunikation

Einige Möglichkeiten der Prozess - Kommunikation (Interprocess Communication (IPC)) sind:

- Kommunikation über **Signale**
- Signale sind asynchron auftretende Ereignisse, die eine Unterbrechung bewirken (--> Software Interrupt).
- In der Regel zur Kommunikation zwischen BS und Benutzerprozeß.

- ¨ Auslösung vom Benutzer (z.B. Tastendruck) ¨
- Auslösung durch Programmfehler (z.B. Division durch 0)
- Auslösung durch andere Prozesse (z.B. Plattenzugriff durch BS-Dienstroutine, "Daten sind bereit")

Prozess - Kommunikation

Einige Möglichkeiten der Prozess - Kommunikation (Interprocess Communication (IPC)) sind:

- Kommunikation über **Nachrichten** (Botschaften, Messages)
- Nachrichten werden vom BS verwaltet.
- Dieses stellt eine für die beteiligten Prozesse gemeinsam nutzbare Transportinstanz (z. B. "Mailbox") zur Verfügung.
- Auf diese greifen die Prozesse über bestimmte Transport-Funktionen des BS (Systemaufrufe) zu.

Prozeß A sendet z. B. eine Botschaft an Prozeß B, indem er sie in der Mailbox ablegt (send(message);).

Der Prozeß B holt die Nachricht dann von der Mailbox ab (receive(message);).

Prozess - Kommunikation

Einige Möglichkeiten der Prozess - Kommunikation (Interprocess Communication (IPC)) sind:

- Kommunikation über **Streams**
  Streams ermöglichen die Kommunikation über Rechnernetze. Logisch gesehen haben Streams dieselbe Aufgabe wie die lokalen Pipes.

- Kommunikation über **Prozedurfernaufrufe** (remote procedure call)
  Ein Prozeß ruft eine in einem anderen Prozeß angesiedelte Prozedur auf (also über seine Adreßgrenzen hinweg). Besonders für Client-Server-Beziehungen geeignet

Prozess - Kommunikation

Einige Möglichkeiten der Prozess - Kommunikation (Interprocess Communication (IPC)) sind:

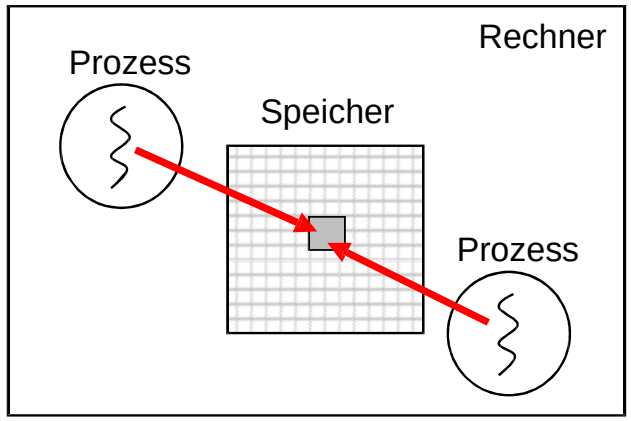
->>> Selbst bei sehr einfachen Betriebssystemen ist eine IPC notwendig, da zumindest eine Kommunikation zwischen einem Prozess und dem Scheduler möglich sein muss.

Interprozesskommunikation (IPC)

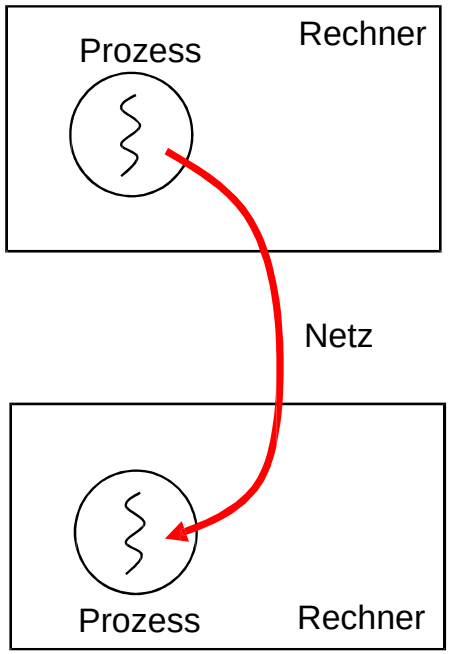
- Prozesse arbeiten oft nicht allein, sondern müssen Informationen austauschen, um eine gemeinsame Aufgabe zu erfüllen.
- Bei diesem Austausch müssen drei wichtige Fragen beantwortet werden:
  - Wie werden die Daten ausgetauscht?
    - Über gemeinsame Variablen?
    - Über Nachrichtenaustausch?
  - Wie wird sicher gestellt, dass die Prozesse nicht gleichzeitig auf gemeinsame Information zugreifen?
  - Wie wird die richtige Reihenfolge des Zugriffs sicher gestellt (Producer-Consumer-Problem)?
- Die beiden letzten Fragen beschreiben das **Synchronisationsproblem**.

Kommunikationsformen

- **Gemeinsame Variablen**: vor allem in Ein-Prozessor und Multiprozessor-Systemen mit gemeinsamem physikalischen Speicher



- **Nachrichtenaustausch**: vor allem bei verteilten Systemen, also Kommunikation über Rechnergrenzen hinweg



Nachrichtensysteme

- Durch den Austausch von Nachrichten lassen sich Prozesse (Threads) synchronisieren. Benötigt werden zwei primitive Operationen:
  - `send(destination, message)`
    Sendet eine Nachricht an einen Empfänger (destination).
  - `receive(source, message)`
    Emfpängt eine Nachricht von einer Quelle (source).
- Blockierende Nachrichtensysteme blockieren den Prozess im `send()` bzw. `receive()` Aufruf, wenn beim Senden kein Empfänger empfangsbereit ist oder beim Empfangen kein aktiver Sender existiert („rendezvous“).
- Speichernde Nachrichtensysteme verwalten Nachrichten in speziellen Speicherbereichen (Mailbox, Warteschlange), so dass Prozesse asynchron Nachrichten austauschen können.

Zusammenfassung Nachrichtensysteme

- Eigenschaften von Nachrichtensystemen
  - Nachrichtensysteme erlauben die Synchronisation von Prozessen oder Threads, die auf verschiedenen Rechnern ohne gemeinsamen Hauptspeicher laufen.
  - Nachrichtensysteme lassen sich auf Benutzerebene implementieren und können somit ohne spezielle Übersetzer überall benutzt werden.
  - Nachrichtensysteme sind normalerweise langsamer als Synchronisationsverfahren, die auf gemeinsamem Speicher beruhen.
- Anforderungen an das Netz
  - Nachrichten dürfen nicht während der Übertragung verloren gehen.
  - Nachrichten dürfen nicht während der Übertragung dupliziert werden.
  - Die Zieladressen müssen eindeutig identifiziert werden können.
  - Es darf nicht möglich sein, dass Prozesse beliebige Nachrichten an beliebige andere Prozesse schicken dürfen.

Erzeuger/Verbraucher-Problem

- Problemstellung:
  - Zwei Prozesse besitzen einen gemeinsamen Puffer mit einer festen Länge (bounded buffer). Eine Prozess schreibt Informationen in den Puffer (producer), der andere liest Informationen aus dem Puffer (consumer).
  - Der Erzeuger darf nicht in den vollen Puffer einfügen.
  - Der Verbraucher darf nicht aus dem leeren Puffer lesen.
- Eine mögliche Lösung:

```
while (true) {
   produce(&item);
   while (count == N) sleep(1);
   buffer[in] := item;
   in := (in + 1) % N;
   count := count + 1;
}
```

```
while (true) {
   while (count == 0) sleep(1);
   item = buffer[out];
   out := (out + 1) % N;
   count := count - 1;
   consume(item);
}
```

Sychronisationsproblem

- Die Anweisungen `count := count + 1` und `count := count - 1` werden typischerweise zu den folgenden Maschinenbefehlen:

```
P1: register1 := count
P2: register1 := register1+1
P3: count := register1
```

```
C1: register2 := count
C2: register2 := register2-1
C3: count := register2
```

- Nehmen wir an, der Wert von `count` sei 5. Was liefert die Ausführung der Befehle in der Reihenfolge (a) P_1, P_2, C_1, C_2, P_3, C_3 und die Ausführung in der Reihenfolge (b) P_1, P_2, C_1, C_2, C_3, P_3?

Synchronisationsproblem

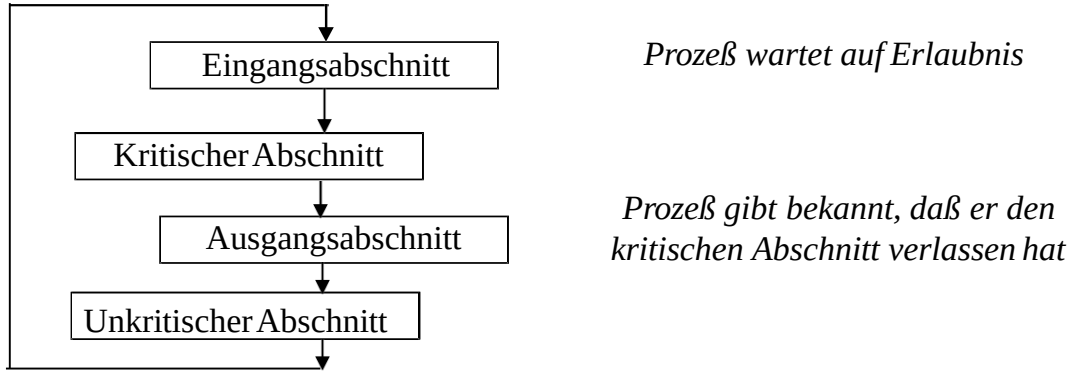
```
P1: register1 := count
P2: register1 := register1+1
                          C1: register2 := count
                          C2: register2 := register2-1
P3: count := register1
                          C3: count := register2
```

Race Conditions

- (a) liefert den falschen Wert 4, (b) liefert den falschen Wert 6. Diese Werte sind falsch, da ja ein Element eingefügt und eines entfernt wird, der Wert müsste also bei 5 bleiben.
- Die angegebene Lösung erzeugt falsche Ergebnisse, die von der Bearbeitungsreihenfolge der Prozesse abhängen.
- Jede Situation, in der mehrere Prozesse gemeinsame Daten manipulieren, kann zu derartigen Synchronisationsproblemen (race conditions) führen. *Synchronisationsverfahren* garantieren, dass immer nur ein Prozess zu einem bestimmten Zeitpunkt gemeinsam benutzte Daten manipulieren kann.

Kritischer Abschnitt

- Ein kritischer Abschnitt (critical section) eines Programms ist eine Menge von Instruktionen, in der das Ergebnis der Ausführung auf unvorhergesehene Weise variieren kann, wenn Variablen, die auch für andere parallel ablaufende Prozesse oder Threads zugreifbar sind, während der Ausführung verändert werden.
- Prinzipieller „Lebenszyklus“ eines Prozesses oder Threads:



Kritischer Abschnitt

- Das Problem besteht darin, ein „Protokoll“ zu entwerfen, an das sich alle Prozesse oder Threads halten und das die Semantik des kritischen Abschnitts realisiert.
- Anforderungen an eine Lösung:
  - Zwei Prozesse dürfen nicht gleichzeitig in ihrem kritischen Abschnitt sein (safety, mutual exclusion).
  - Es dürfen keine Annahmen über die Bearbeitungsgeschwindigkeit von Prozessen gemacht werden.
  - Kein Prozess, der außerhalb eines kritischen Bereichs ist, darf andere Prozesse beim Eintritt in den kritischen Abschnitt behindern.
  - Kein Prozess darf ewig auf den Eintritt in den kritischen Abschnitt warten müssen (fairness, bounded waiting, progress).
  - Möglichst passives statt aktives Warten, da aktives Warten einerseits Rechenzeit verschwendet und andererseits Blockierungen auftreten können, wenn auf Prozesse/Threads mit niedriger Priorität gewartet werden muss.

Lösungen für die Synchronisation

- Es wurden eine Reihe von Lösungen für die Synchronisation von Prozessen entwickelt, von denen wir die wichtigsten besprechen:
  - Semaphore
  - Mutexe
  - Monitore
- Für Interessierte: Es gibt auch reine Programmierlösungen (ohne spezielle Hardware oder scheduling Unterstützung) gemäß dem „Bakery Algorithm“ (Bäcker-Algorithmus), bei dem Prozesse
  - fortlaufende Nummern ziehen
  - Dabei kann es passieren, dass 2 Prozesse gleiche Nummern ziehen (race condition)
  - Wenn 2 Prozesse, die gleiche Nummer haben, dann gewinnt z.B. der Prozess mit der kleinsten Prozess-ID.

  Details siehe z.B. Silberschatz „Operating System Concepts“
- Dies sind Lösungen für die Synchronisation bei Nutzung gemeinsamer Variablen. Zur Erinnerung Bei Nachrichtenkommunikation wird diese Form der Synchronisation nicht benötigt. Dies übernehmen die Transportprotokolle z.B. TCP

Semaphore

- Ein Semaphor ist eine geschützte Variable, auf die nur die unteilbaren (atomaren) Operationen `up` (signal, V) und `down` (wait, P) ausgeführt werden können:

```
down(s)
{   s := s - 1;
    if (s < 0) queue_this_process_and_block();
}

up(s)
{   s := s + 1;
    if (s <= 0) wakeup_process_from_queue();
}
```

Eigenschaften von Semaphoren

- Semaphore können zählen und damit z.B. die Nutzung gemeinsamer Betriebsmittel überwachen.
- Semaphore werden durch spezielle Systemaufrufe implementiert, die die geforderten atomaren Operationen `up` und `down` realisieren.
- Semaphore können in beliebigen Programmiersprachen benutzt werden, da sie letztlich einem Systemaufruf entsprechen.
- Semaphore realisieren ein passives Warten bis zum Eintritt in den kritischen Abschnitt.

Mutexe

- Oft wird die **Fähigkeit zu zählen** bei Semaphoren nicht benötigt, d.h., es genügt eine einfache binäre Aussage, ob ein kritischer Abschnitt frei ist oder nicht.
- Dazu kann eine einfacher zu implementierende Variante, der sogenannte **Mutex** (von „mutual exclusion“), verwendet werden.

Erzeuger/Verbraucher mit Semaphoren

- Das Erzeuger/Verbraucher Problem lässt sich elegant mit drei Semaphoren lösen:
  1. Ein Semaphor zum Betreten der kritischen Abschnitte (mutex).
  2. Ein Semaphor, das die freien Plätze im Puffer herunter zählt und den Prozess blockiert, der in einen vollen Puffer schreiben will (numItemsLeft).
  3. Ein Semaphor, das die belegten Plätze im Puffer herauf zählt und den Prozeß blockiert, der von einem leeren Puffer lesen will (numItems).

```
while (true) {
   produce(&item);
   down(&numItemsLeft);
   down(&mutex);
   add(&item);
   up(&mutex);
   up(&numItems);
}
```

```
semaphore
   mutex = 1,
   numItemsLeft = N,
   numItems = 0;
```

```
while (true) {
   down(&numItems);
   down(&mutex);
   remove(&item);
   up(&mutex);
   up(&numItemsLeft);
   consume(item);
}
```

Implementierung von Semaphoren

- Semaphore lassen sich als Systemaufrufe implementieren, wobei kurzzeitig sämtliche Unterbrechungen unterbunden werden. (Da zur Implementation nur wenige Maschinenbefehle benötigt werden, ist diese Möglichkeit akzeptabel.)
- Auf Mehrprozessor-Systemen muss ein Semaphor mit einer unteilbaren Prozessor-Operation implementiert werden, die das Semaphor vor gleichzeitigen Änderungen in anderen Prozessoren schützt.
- Beispiel: **Test-And-Set-Lock** (`TSL`): beim Ausführen der Operation wird der Memory-Bus für alle anderen Operationen gesperrt

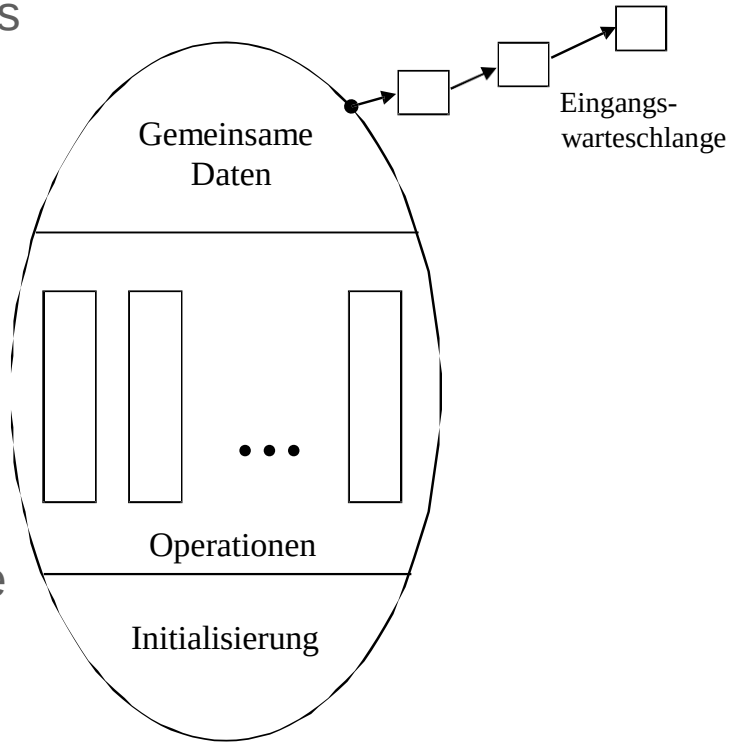
Probleme bei Semaphoren

- Programmierfehler bei der Benutzung eines Semaphors können zu Verklemmungen oder inkorrekten Ergebnissen führen. Typische Fehler:
  - Sprünge aus kritischen Bereichen, ohne das mutex-Semaphor freizugeben.
  - Sprünge in kritische Bereiche, ohne das mutex Semaphor zu setzen.
  - Vertauschungen von Semaphoren zum Schutz von kritischen Abschnitten und Semaphoren, die vorhandene Betriebsmittel zählen.
- Alles in allem sind Semaphore eine „low-level“-Lösung, die erhebliche Disziplin vom Programmierer verlangt. Eine komfortablere Lösung bieten *Monitore*.

Monitore

- Ein Monitor ist eine Sammlung von Prozeduren, Variablen und Datenstrukturen, die in einem Modul gekapselt sind.
- Prozesse können die Prozeduren des Monitors aufrufen, aber keine internen Daten ändern.
- Monitore besitzen die Eigenschaft, dass immer nur genau ein Prozess im Monitor aktiv sein kann.
- Monitore sind Konstrukte einer Programmiersprache und erfordern daher spezielle Compiler.
- Es ist die Aufgabe des Compilers, Maschinenbefehle zu generieren, die den wechselseitigen Ausschluss im Monitor garantieren.

Schematischer Aufbau eines Monitors:



Bedingungsvariable eines Monitors

- Bedingungsvariablen (condition variables) eines Monitors mit den zugehörigen Operationen `wait()` und `signal()` erlauben es, im Monitor auf andere Prozesse zu warten:
  - `wait(c)`: Der aufrufende Prozess blockiert, bis ein `signal()` auf der Bedingungsvariablen `c` ausgeführt wird. Ein anderer Prozess darf den Monitor betreten.
  - `signal(c)`: Ein auf die Bedingungsvariable `c` wartender Prozeß wird aufgeweckt. Der aktuelle Prozess muss den Monitor sofort verlassen.
- Der durch `signal()` aufgeweckte Prozess wird zum aktiven Prozess im Monitor, während der Prozess, der `signal()` ausgeführt hat, blockiert.
- Bedingungsvariablen sind keine Zähler. Ein `signal(c)` auf einer Variablen `c` ohne ein `wait(c)` geht einfach verloren.

Erzeuger/Verbraucher-Problem mit einem Monitor

```
monitor ProducerConsumer
  condition full, empty
  integer count;

  procedure enter
      if count = N  then
         wait(full);
      enter_item()
      count := count + 1;
      if count = 1  then
         signal(empty);
  end;

  procedure remove
      if count = 0 then
         wait(empty);
      remove_item();
      count := count - 1;
      if count = N-1 then
         signal(full);
  end;
```

```
procedure producer
   while (true) do begin
         produce_item;
         ProducerConsumer.enter;
   end
end;
```

```
procedure consumer
   while (true) do begin
         ProducerConsumer.remove;
         consume_item;
   end
end;
```

Kritische Abschnitte mit Compilerunterstützung in Java

- In Java können kritische Abschnitte als Anweisungsfolge geschrieben werden, denen das Schlüsselwort `synchronized` voran gestellt wird.
- Die kritischen Abschnitte schließen sich bzgl. des Sperrobjekts gegenseitig aus.

```
class Buffer {
  private const int size = 8;
  private int count = 0, out = 0, in =
     0;
  private int[] pool = new int[size];

  public synchronized void insert(int i)
     {
    pool[in] = i;
    in = (in + 1) % size;    count++;
  }

  public sychronized int remove() {
    int res = pool[out];
    out = (out + 1) % size;  count--;
    return res;
  }

  public synchronized int cardinal() {
    return count;
  }
}
```

Überblick

1. Prozesse und Threads ✓
2. Prozess-Scheduling ✓
3. Interprozesskommunikation (IPC) ✓
   - Kritische Regionen
   - Konstrukte zur Synchronisation
   - Klassische Probleme
4. Verklemmungen
   - Ressourcen
   - Deadlockerkennung und –behebung
   - Deadlockvermeidung