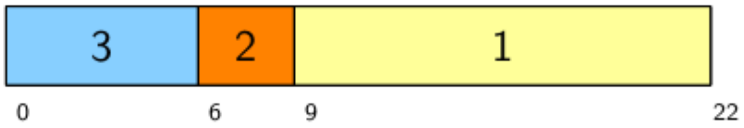
-Prioritäts-Scheduling: Jeder Auftrag hat statische Priorität
höchste Priorität hat Vorrang
Bei gleicher Priorität FCFS

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit	Priorität
1	13	2
2	3	3
3	6	4

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	6+3=9	22
2	6	9
3	0	6

Durchschnittliche Wartezeit[4]:
$(9+6)/3 = 5$

[4] In diesem Beispiel – abhängig von Prioritätsvergabe sind auch alle anderen Ergebnisse möglich

5) Prozesssynchronisation

Definitionen

-Prozesskonflikt: zwei nebenläufige (Concurrent) Prozesse heißen im Konflikt zueinander stehend oder überlappend, wenn es eine BM gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie unabhängig oder disjunkt

- Zeitkritische Abläufe (race conditions): Folgen von Lese/Schreib-Operationen der verschiedenen Prozesse heißen zeitkritische Abläufe (engl. race conditions), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese/Schreib-Operationen sind.

- Wechselseitiger Ausschluss (mutual exclusion): Ein Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess auf ein gemeinsames Datum zugreift, heisst Verfahren zum wechselseitigen Ausschluss.
Bemerkung: ein solches Verfahren vermeidet zeitkritische Abläufe und löst somit ein Basisproblem des Concurrent Programming

- Kritischer Abschnitt (critical section): Der Teil eines Programms, in dem auf gemeinsam benutzte Datenbereiche zugegriffen wird
Bemerkung: Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe. Kritische Abschnitte realisieren sog. komplexe unteilbare oder atomare Operationen.

Anforderungen an einen guten Algorithmus

- Immer nur ein Prozess in seinem kritischen Abschnitt (Korrektheit, Basisforderung)
-Kein Prozess,der nicht in seinem kritischen Bereich ist,darf andere Prozesse blockieren (Fortschritt)
- Alle Prozesse werden gleich behandelt (Fairness)
- Kein Prozess darf unendlich lange warte müssen, bis er in seinen kritischen Bereich eintreten kann (starvation)

Synchronisationsprimitive

Wechselseitiger Ausschluss mit aktivem Warten

- Funktionen: enter_critical_section und leave_critical_section
- Lösungen:
  1 Sperren aller Unterbrechungen: Interruptkonfiguration i.d.R. nur im Kernmodus möglich, unbrauchbar bei Multiprozessor-Systeme
  2 Sperrvariablen: Zwischen Abfrage der Sperrvariablen und folgendem Setzen kann der Prozess unterbrochen werden
  3 Striktes Alternieren: erfüllt im Vergleich zu 2 die Korrektheits-bedingung. Wenn ein Prozess viel langsamer als der andere ist kann die Fortschrittsbedingung verletzt werden
  4 Peterson: enter_critical Funktion zeigt eigenes Interesse und setzt Marke. leave_critical verlässt kritischen Bereich (kein Interesse mehr)
  5 Atomare read-modify-write Instruktionen: Algorithmen sind komplex, fehleranfällig und starvation-anfällig. Lösung durch HW-Unterstützung. Atomare Maschinenbefehle (TAS = Test And Set)
- -Lock Holder Preemption Problem (kann lange dauern [Quantum])
  -ein (virtueller) Prozessor hat einen durch Spinlock geschützten Bereich betreten und wird dort unterbrochen
  - anderer (virtueller) Prozessor wartet auf Freigabe des Spinlocks

Wechselseitiger Ausschluss mit passivem Warten

- Einfachste Primitive heißen meistens SLEEP() und WAKEUP(process)
- Mutex-Locks (lock() als Prolog und unlock als Epilog)
- Problem der Prioritätsinversion: beim prioritätsbasierten Scheduling muss ein Prozess mit hoher Priorität auf einen Prozess mit niedriger Priorität warten weil dieser den kritschen Abschnitt noch nicht freigegeben hat
- Semaphore: Supermarkt-Einkaufswagen Analogie besteht aus Zählvariable, die begrenzt, wieviele Prozesse momentan ohne Blockierung passieren dürfen. Und einer Warteschlange für (passiv) wartende Prozesse.

Operationen:

- Zähler auf initialen Wert (# Freie Einkaufswagen) setzen
- P(): Passierwunsch (auch DOWN() genannt)
- V(): Freigeben (auch UP() genannt)
- P() und V() sind atomar
- kein Prozess wird bei der Ausführung von V() blockiert
- i.d.R. als Systemaufrufe implementiert
- Einprozessorsysteme sperren Interrupts bei P() und V()
- Multiprozessorsysteme beschützen Semaphore (unkritisch) durch Spinlocks. Es kann immer nur ein Prozessor den Semaphor manipulieren

-Binär- und Zählsemaphore [Programmierung ist fehleranfällig)

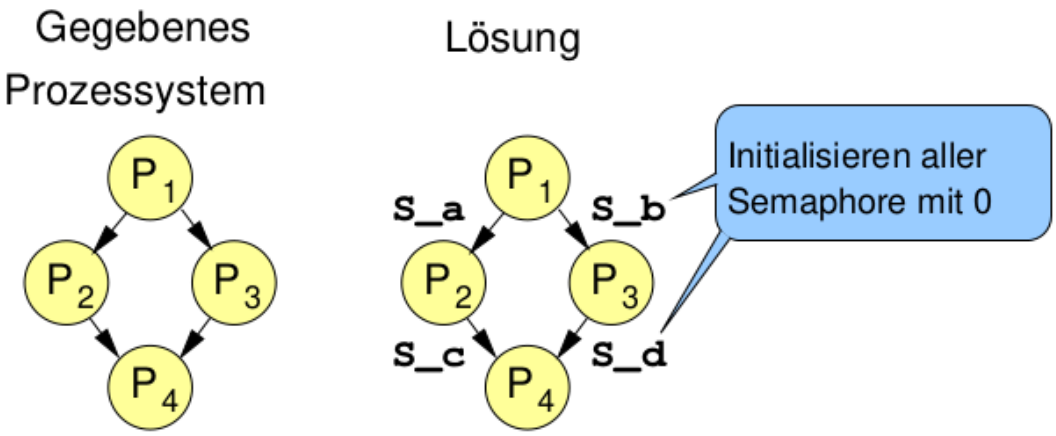
```
/* Semaphore initialisieren:
 * empty = Puffergroesse,
 * full=0, mutex=1
 */

void insert(int item) {
    P(&empty);
    P(&mutex);
/* hier: item in Puffer
   stellen                */
    V(&mutex);
    V(&full);
}

int remove(void) {
    P(&full);
    P(&mutex);
/* hier: vorderstes Item
   aus Puffer holen    */
    V(&mutex);
    V(&empty);
}
```

full: zählt belegte Einträge im Puffer, verhindert Entnahme aus leerem Puffer

empty: verwaltet freie Plätze im Puffer, verhindert Einfügen in vollen Puffer

mutex: schützt den kritischen Bereich vor gleichzeitigem Betreten (binär-Semaphor: nimmt nur Werte 1/0 an)

Beispiel: Vorrangrelation




```
P1() {
.. work ..
 V(S_a);
 V(S_b);
 exit();
}
```

```
P2() {
 P(S_a);
.. work ..
 V(S_c);
 exit();
}
```

```
P3() {
 P(S_b);
.. work ..
 V(S_d);
 exit();
}
```

```
P4() {
 P(S_c);
 P(S_d);
.. work ..
 exit();
}
```

Weitere Ansätze

-Condition Variable

-Monitore

Klassische Synchronisationprobleme

-Erzeuger-Verbraucher Problem

Erzeuger: Will Einfügen, aber Puffer ist voll.
Lösung: Lege dich schlafen, lass dich vom Verbraucher wecken, wenn er ein Datum entnommen hat.
Verbraucher: Will Entnehmen, aber Puffer ist leer.
Lösung: Lege dich schlafen, lass dich vom Erzeuger wecken, wenn er ein Datum eingefügt hat.

Lösung mit Semaphoren (1)

Hochschule RheinMain

Wurde bereits besprochen (vgl. 5.2.3)

```
#define N 100                   /* Kapazitaet des Puffers        */

/* gemeinsame Variablen
*/
semaphore mutex = 1;            /* kontrolliert krit. Bereich    */
semaphore empty = N;            /* zaehlt leere Eintraege        */
semaphore full  = 0;            /* zaehlt belegte Eintraege      */

void erzeuger(void)             /* Erzeuger                      */
{
    int item;
    while (TRUE) {
        produce_item(&item);    /* erzeuge Eintrag               */
        P(&empty);              /* besorge freien Platz          */
        P(&mutex);              /* tritt in krit. Abschnitt ein  */
        enter_item(item);       /* fuege Eintrag in Puffer ein   */
        V(&mutex);              /* verlasse krit. Bereich        */
        V(&full);               /* erhoehe Anz. belegter Eintr.  */
    }
}
```


Lösung mit Semaphoren (2)

Hochschule RheinMain

```
void verbraucher(void)/* Verbraucher                    */
{
    int item;
    while(TRUE) {
        P(&full);           /* belegter Eintrag vorhanden?   */
        P(&mutex);          /* tritt in krit. Abschnitt ein  */
        remove_item(&item); /* entnimm Eintrag aus Puffer    */
        V(&mutex);          /* verlasse krit. Bereich        */
        V(&empty);          /* erhoehe Anz. freier Eintraege */
        consume_item(item); /* verarbeite Eintrag            */
    }
}
```

Lösung mit Nachrichtenaustausch (1)

Hochschule RheinMain

Grundlage: Funktionen zum Nachrichtenaustausch

`send(process, int* message)` - Nachricht an Prozess senden

`receive(process, int* message)` - Nachricht von Prozess empfangen

```
#define N      100              /* Kapazitaet des Puffers        */
#define MSIZE    4              /* Nachrichtengroesse            */

typedef int message[MSIZE];     /* Nachrichtentyp                */

void producer(void)             /* Erzeuger                      */
{
   int item;
   message m;

   while (TRUE) {
      produce_item(&item);      /* erzeuge Eintrag               */
      receive(consumer, &m);    /* warte auf leere Nachricht     */
      build_message(&m, item); /* erzeuge zu sendende Nachricht*/
      send(consumer, &m);       /* sende Nachricht z Verbraucher*/
   }
}
```


Lösung mit Nachrichtenaustausch (2)

Hochschule RheinMain

```
void consumer(void)             /* Verbraucher                   */
{
   int item, i;
   message m;

   for (i = 0; i < N; i++)      /* sende N leere Nachrichten     */
      send (producer, &m);

   while (TRUE) {
      receive(producer, &m);    /* empfange Nachricht v Erzeuger*/
      extract_item(&m, &item); /* entnimm Eintrag               */
      send(producer, &m);       /* sende leere Nachricht zurueck*/
      consume_item(item);       /* verarbeite Eintrag            */
   }
}
```

- Dining Philosophers Problem

Lösung mit Semaphoren (1)

Hochschule RheinMain

```
#define N           5       /* Anzahl der Philosophen              */
#define LEFT  (i-1+N)%N     /* Nummer des linken Nachbarn von i    */
#define RIGHT (i+1)%N       /* Nummer des rechten Nachbarn von i   */
#define THINKING    0       /* Zustand: Denkend                    */
#define HUNGRY      1       /* Zust: Versucht, Gabeln zu bekommen  */
#define EATING      2       /* Zustand: Essend                     */

/* gemeinsame Variablen                                           */
int state[N];               /* Zustaende aller PhilosophInnen      */
semaphore mutex = 1;        /* fuer wechselseitigen Aussschluss    */
semaphore s[n];             /* Semaphor fuer jeden Philosoph       */

void philosopher(int i) {  /* i:0..N-1, welcher Philosoph      */
    while (TRUE) {
        think();            /* Denken                              */
        take_forks(i);      /* Greife beide Gabeln oder blockiere  */
        eat();              /* Essen                               */
        put_forks(i);       /* Ablegen beider Gabeln               */
    }
}
```


Lösung mit Semaphoren (1)

Hochschule RheinMain

```
void take_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?  */
{
    P(&mutex);          /* tritt in krit. Bereich ein          */
    state[i] = HUNGRY;  /* zeige, dass du hungrig bist         */
    test(i);            /* versuche, beide Gabeln zu bekommen  */
    V(&mutex);          /* verlasse krit. Bereich              */
    P(&s[i]);           /* bockiere, falls Gabeln nicht frei   */
}

void put_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?   */
{
    P(&mutex);          /* tritt in krit. Bereich ein          */
    state[i] = THINKING;/* zeige, dass du fertig bist          */
    test(LEFT);         /* kann linker Nachbar jetzt essen ?   */
    test(RIGHT);        /* kann rechter Nachbar jetzt essen ?  */
    V(&mutex);          /* verlasse krit. Bereich              */
}

void test(int i) /* i:0..N-1, welche(r) PhilosophIn? */
{
    if (state[i]==HUNGRY &&
        state[LEFT]!=EATING && state[RIGHT]!=EATING) {
        state[i]=EATING;/* jetzt kann Phil i essen !           */
        V(&s[i]);       /* "sage es ihm"                       */
    }
}
```

- Leser Schreiber Problem
Zu jedem Zeitpunkt dürfen entweder mehrere Leser oder ein Schreiber zugreifen.
Verboten: gleichzeitiges Lesen und Schreiben
Wie sollten Leser- und Schreiber-Programme aussehen?

Lösung mit Semaphoren (1)

Hochschule RheinMain

Lesezugriff:

```
/* gemeinsame Variablen: */
semaphore mutex = 1;      /* wechsels. Aussschluss fuer rc     */
semaphore db = 1;         /* Semaphor fuer Datenbestand        */
int rc = 0;               /* readcount: Anzahl Leser           */

void reader(void)         /* Leser                             */
{
   while (TRUE) {
      P(&mutex);          /* erhalten exkl. Zugriff auf rc     */
      rc = rc + 1;        /* ein zusaetzlicher Leser           */
      if (rc==1)          /* Erster Leser?                     */
          P(&db);         /* ja -> reserviere Daten            */
      V(&mutex);          /* freigeben exkl. Zugriff auf rc    */

      read_data_base();   /* lies Datenbestand                 */

      P(&mutex);          /* erhalten exkl. Zugriff auf rc     */
      rc = rc - 1;        /* ein Leser weniger                 */
      if (rc==0)          /* letzter Leser ?                   */
          V(&db);         /* ja -> Daten freigeb.              */
      V(&mutex);          /* freigeben exkl. Zugriff auf rc    */
      use_data_read();    /* unkrit. Bereich                   */
   }
}
```


Lösung mit Semaphoren (2)

Hochschule RheinMain

Schreibzugriff:

```
void writer(void)         /* Schreiber                         */
{
   while (TRUE) {
      create_data();      /* unkrit. Bereich                   */
      P(&db);             /* erhalten exkl. Zugriff auf Daten  */
      write_data_base();  /* schreibe Datenbestand             */
      V(&db);             /* freigeben exkl. Zugriff auf Daten */
   }
}
```

`mutex` sichert krit. Abschnitt bezüglich des Read-Counters `rc`.
`db` sichert Zugriff auf den Datenbestand, so dass **entweder** mehrere Leser **oder** ein Schreiber zugreifen können.
Der erste Leser führt eine P-Operation auf `db` aus, alle weiteren inkrementieren nur `rc`.
Der letzte Leser führt eine V-Operation auf `db` aus, so dass ein wartender Schreiber Zugriff erhält.
Die **Lösung bevorzugt Leser**: Neu eintreffende Leser erhalten Zugriff vor einem schon wartenden Schreiber, wenn noch mindestens ein Leser Zugriff hat.