Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung P und Freigabe V des kritischen Bereiches durch den Programmierer
- Vergißt der Entwickler z.B. die Freigabe V des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
  - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozess zugreifen kann.
  - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
  - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
  - Die Signalisierung ist innerhalb des Monitors festgelegt, der Programmierer muss sie nicht selbstständig implementieren.

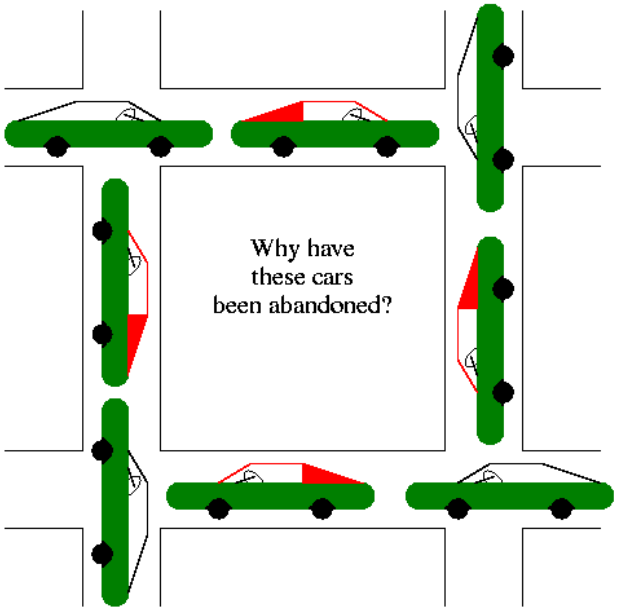
Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung**: normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphor implementiert.
  Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

```
public class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }

    synchronized public void up() {
        value++;
        if(value==1) notify();
    }

    synchronized public void down() {
        while(value==0) wait();
        value- -;
    }
}
```

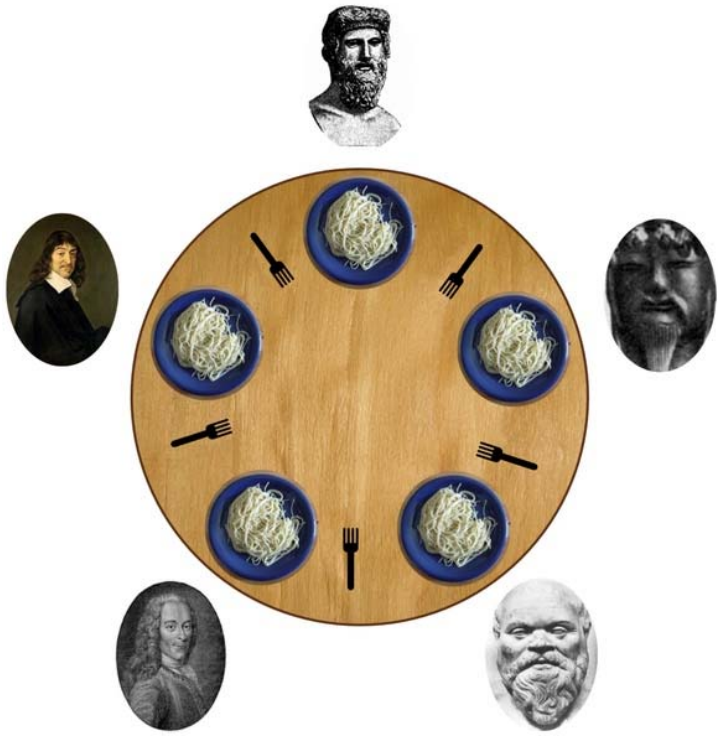
Bemerkung zu Verklemmungen (Deadlocks)

- Auch bei der korrekten Verwendung von Semaphoren und Monitoren kann es zu Deadlocks kommen, siehe Beispiel auf der folgenden Seite.
- Coffman, Elphick und Shoshani haben 1971 die vier konjunktiv notwendigen Voraussetzungen für einen Deadlock formuliert:
  1. Wechselseitiger Ausschluss: Es gibt eine Menge von exklusiven Ressourcen R_{exkl}, die entweder frei sind oder genau einem Prozess zugeordnet sind.
  2. Hold-and-wait-Bedingung: Prozesse, die bereits im Besitz von Ressourcen aus R_{exkl} sind, fordern weitere Ressourcen aus R_{exkl} an.
  3. Ununterbrechbarkeit: Die Ressourcen R_{exkl} können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
  4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.
- Umgekehrt (und positiv) formuliert: ist eine der Bedingungen nicht erfüllt, so sind Verklemmungen ausgeschlossen.



Klassisches Beispiel: Speisende Philosophen

- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber nur 5 Gabeln verfügbar.
- Die Philosophen denken und diskutieren. Ist einer hungrig, so greift er zunächst zur linken und dann zur rechten Gabel. Ist eine Gabel nicht an ihrem Platz, so wartet er bis die Gabel wieder verfügbar ist (ohne eine evtl. in der Hand befindliche Gabel zurückzulegen). Nach dem Essen legt er die Gabeln zurück.
- Problem: sind alle Philosophen gleichzeitig hungrig, so nehmen sie alle ihre linke Gabel und gleichzeitig ihrem Nachbarn die rechte Gabel weg. Alle Philosophen warten auf die rechte Gabel und es entsteht eine Verklemmung (deadlock).
- Gibt ein Philosoph seine Gabel nicht mehr zurück, so stirbt der entsprechende Nachbar den **Hungertod (starvation)**.



Fragestellung: Invers zählender Semaphor

- Aufgabenstellung: Implementierung des Leser-Schreiber-Problems mit Schreiber-Priorität
- Erläuterung:
  - Auf einen Datensatz können mehrere Leser gleichzeitig oder aber ein Schreiber zugreifen.
  - Sobald ein Schreiber den Schreibwunsch äußert, soll kein weiterer Leser (oder Schreiber) mehr auf den Datensatz zugreifen können. Zum Zeitpunkt der Signalisierung bestehende Lesevorgänge können regulär beendet werden, erst danach darf der Schreiber auf die Daten zugreifen.
- Problem: Häufig wird versucht das Problem mit einem „*invers zählenden Semaphor*“ zu lösen, also einem Semaphor, der bei 0 freigibt und sonst blockiert.
- Wie geht es richtig?

Nebenläufigkeit

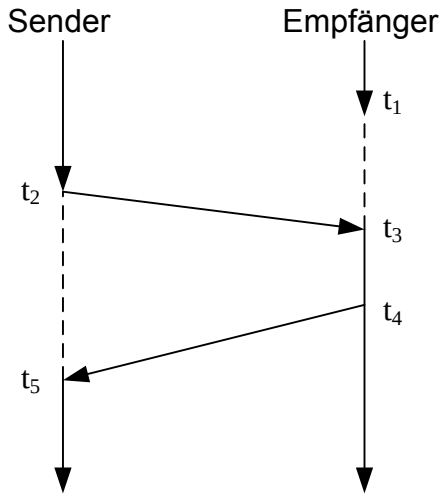
Interprozesskommunikation (IPC)

Interprozesskommunikation

- Notwendigkeit der Interprozesskommunikation
  - Prozesse arbeiten in unterschiedlichen Prozessräumen oder sogar auf unterschiedlichen Prozessoren.
  - Prozesse benötigen evtl. Ergebnisse von anderen Prozessen.
  - Zur Realisierung von wechselseitigen Ausschlüssen werden Mechanismen zur Signalisierung benötigt.
- Klassifikation der Kommunikation
  - synchrone vs. asynchrone Kommunikation
  - pure Ereignisse vs. wertbehaftete Nachrichten

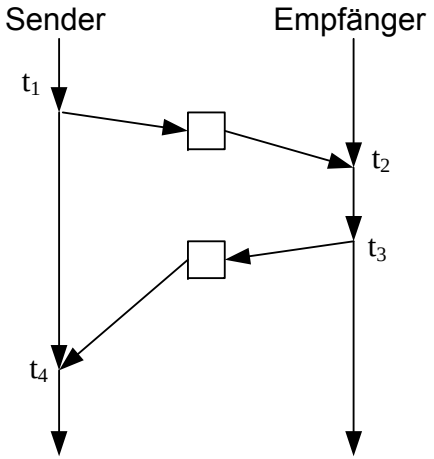
Synchron vs. Asynchron

Synchrone Kommunikation



t_1 : Empfänger wartet auf Nachricht
t_2 : Sender schickt Nachricht und blockiert
t_3 : Empfänger bekommt Nachricht, die Verarbeitung startet
t_4 : Verarbeitung beendet, Antwort wird gesendet
t_5 : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



t_1 : Sender schickt Nachricht an Zwischenspeicher und arbeitet weiter
t_2 : Empfänger liest Nachricht
t_3 : Empfänger schreibt Ergebnis in Zwischenspeicher
t_4 : Sender liest Ergebnis aus Zwischenspeicher

(Nicht eingezeichnet: zusätzliche Abfragen des Zwischenspeichers und evtl. Warten)

IPC-Mechanismen

- Übermittlung von Datenströmen:
  - direkter Datenaustausch
  - Pipes
  - Nachrichtenwarteschlangen (Message Queues)
- Signalisierung von Ereignissen:
  - Signale
  - Semaphore
- Synchrone Kommunikation
  - Barrieren/Rendezvous
  - Kanäle wie z.B. Occam
- Funktionsaufrufe:
  - RPC
  - Corba

Nebenläufigkeit

IPC: Kommunikation durch Datenströme

Direkter Datenaustausch

- Mit Semaphoren und Monitoren geschützte Datenstrukturen eignen sich sehr gut für den Austausch von Daten:
  - schnelle Kommunikation, da auf den Speicher direkt zugegriffen werden kann.
- Allerdings kann die Kommunikation nur lokal erfolgen und zudem müssen die Prozesse eng miteinander verknüpft sein.
- Programmiersprachen, Betriebssysteme, sowie Middlewareansätze bieten komfortablere Methoden zum Datenaustausch.
- Grundsätzlich erfolgt der Austausch über das Ausführen von Funktionen `send(receiver address, &message)` und `receive(sender address, &message)`.

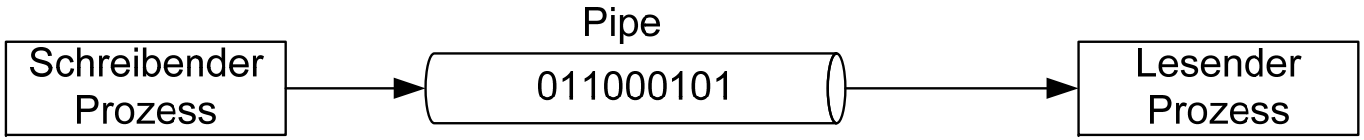
Fragestellungen beim Datenaustausch

- Nachrichtenbasiert oder Datenstrom?
- Lokale oder verteilte Kommunikation?
- Kommunikationsparameter:
  - mit/ohne Bestätigung
  - Nachrichtenverluste
  - Zeitintervalle
  - Reihenfolge der Nachrichten
- Adressierung
- Authentifizierung
- Performance
- Sicherheit (Verschlüsselung)

Heute: vor allem lokale Kommunikation, echtzeitfähige Protokolle zur Kommunikation in eigenem Kapitel

Pipes

- Die Pipe bezeichnet eine gepufferte, unidirektionale Datenverbindung zwischen zwei Prozessen nach dem **First-In-First-Out-** (**FIFO**-)Prinzip.
- Über den Namen der Pipe (ähnlich einem Dateinamen) können Prozesse unterschiedlichen Ursprungs auf eine Pipe lesend oder schreibend zugreifen. Zur Kommunikation zwischen Prozessen gleichen Ursprungs (z.B. Vater-, Kindprozess) können auch anonyme Pipes verwendet werden. Die Kommunikation erfolgt immer asynchron.



Pipes in Posix

- POSIX (Portable Operating System Interface) versucht durch Standardisierung der Systemaufrufe die Portierung von Programmen zwischen verschiedenen Betriebssystemen zu erleichtern.
- POSIX.1 definiert folgende Funktionen für Pipes:

```
int mkfifo(char* name, int mode);       /*Erzeugen einer benannten Pipe*/
int unlink ( char *name );              /*Loeschen einer benannten Pipe*/
int open ( char *name, int flags);      /*Oeffnen einer benannten Pipe*/
int close ( int fd );                   /*Schliessen des Lese- oder Schreibendes einer
                                          Pipe*/
int read ( int fd, char *outbuf, unsigned bytes ); /*Lesen von einer Pipe*/
int write ( int fd, char *outbuf,unsigned bytes ); /*Schreiben an eine Pipe*/
int pipe ( int fd[2] );                 /*Erzeugen eine unbenannte Pipe*/
```

Nachteile von Pipes

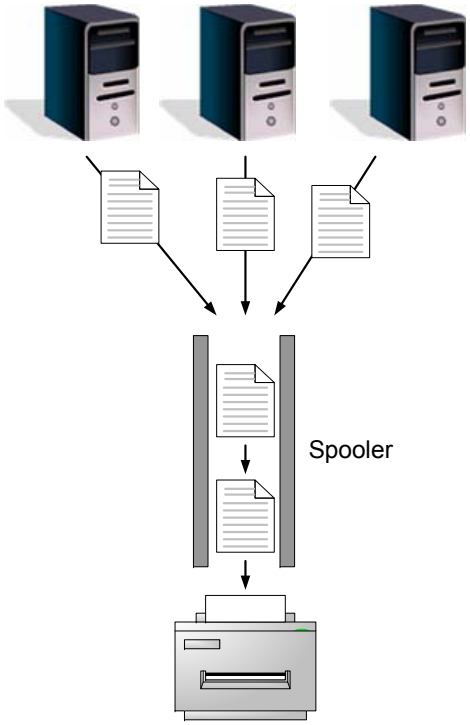
- Pipes bringen einige Nachteile mit sich:
  - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
  - Daten sind nicht priorisierbar.
  - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung:
  - Es können keine Daten aufgehoben werden.
  - Beim Öffnen blockiert der Funktionsaufruf, bis auch das zweite Zugriffsende geöffnet wird (Verhinderung durch O_NDELAY Flag).
- Lösung: Nachrichtenwartschlangen

Nachrichtenwarteschlangen (message queues)

- Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenwarteschlangen, wie in POSIX 1003.1b (Echtzeiterweiterung von POSIX) definiert, betrachtet.
- Eigenschaften der POSIX MessageQueues:
  - Beim Anlegen einer MessageQueue wird der benötigte Speicher reserviert. ⇒ Speicher muss nicht erst beim Schreibzugriff angelegt werden.
  - Die Kommunikation erfolgt nachrichtenorientiert. Die Anzahl der vorhandenen Nachrichten kann dadurch abgefragt werden.
  - Nachrichten sind priorisierbar → Es können leichter Zeitgarantien gegeben werden.

Nachrichtenwarteschlangen

- Schreibzugriff in Standardsystemen: Der schreibende/sendende Prozess wird nur dann blockiert werden, falls der Speicher der Datenstruktur bereits voll ist. **Alternative in Echtzeitsystemen: Fehlermeldung ohne Blockade**.
- Lesezugriff in Standardsystemen: Beim lesenden/empfangenden Zugriff auf einen leeren Nachrichtenspeicher wird der aufrufende Prozess blockiert bis eine neue Nachricht eintrifft. **Alternative: Fehlermeldung ohne Blockade**.
- Ein anschauliches Beispiel für den Einsatzbereich ist der Spooler eines Druckers: dieser nimmt die Druckaufträge der verschiedenen Prozesse an und leitet diese der Reihe nach an den Drucker weiter.



Message Queues in POSIX

- POSIX definiert folgende Funktionen für Nachrichtenwartschlangen:

```
mqd_t mq_open(const char *name, int oflag, ...);  /*Oeffnen einer Message Queue*/

int mq_close(mqd_t mqdes);                        /*Schliessen einer Message Queue*/

int mq_unlink(const char *name);                  /*Loeschen einer
                                                    Nachrichtenwarteschlange*/

int mq_send(mqd_t mqdes, const char *msg_ptr,
    size_t msg_len, unsigned int msg_prio);       /*Senden einer Nachricht*/

size_t mq_receive(mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned int *msg_prio);      /*Empfangen einer Nachricht*/

int mq_setattr(mqd_t mqdes, const struct
    mq_attr *mqstat, struct mq_attr *mqstat);     /*Aendern der Attribute*/

int mq_getattr(mqd_t mqdes,
    struct mq_attr *mqstat);                      /*Abrufen der aktuellen
                                                    Eigenschaften*/

int mq_notify(mqd_t mqdes,
    const struct sigevent *notification);         /*Anforderung eines Signals bei
                                                    Nachrichtenankunft*/
```

Nebenläufigkeit

IPC: Kommunikation durch Ereignisse

Signale

- **Signale** werden in Betriebssystemen typischerweise zur Signalisierung von Ereignissen an Prozessen verwendet.
- Signale können verschiedene Ursachen haben:
  - Ausnahmen, z.B. Division durch Null (SIGFPE) oder ein Speicherzugriffsfehler (SIGSEGV)
  - Reaktion auf Benutzereingaben (z.B. Ctrl / C)
  - Signal von anderem Prozess zur Kommunikation
  - Signalisierung von Ereignissen durch das Betriebssystem, z.B. Ablauf einer Uhr, Beendigung einer asynchronen I/O-Funktion, Nachrichtankunft an leerer Nachrichtenwarteschlange (siehe mq_notify())

Prozessreaktionen auf Signale

- Der Prozess hat drei Möglichkeiten auf Signale zu reagieren:
  1. Ignorierung der Signale
  2. Ausführen einer Signalbehandlungsfunktion
  3. Verzögerung des Signals, bis Prozess bereit für Reaktion ist
- Zudem besteht die Möglichkeit mit der Standardreaktion auf das bestimmte Signal zu reagieren. Da aber typischerweise die Reaktion auf Signale die Beendigung des Empfängerprozesses ist, sollte ein Programm über eine vernünftige Signalbehandlung verfügen, sobald ein Auftreten von Signalen wahrscheinlich wird.

POSIX Funktionen für Signale

- POSIX 1003.1 definiert folgende Funktionen:

Funktion	Bedeutung
kill	Senden eines Signals an einen Prozess oder eine Prozessgruppe
sigaction	Spezifikation der Funktion zur Behandlung eines Signals
sigaddset	Hinzufügen eines Signals zu einer Signalmenge
sigdelset	Entfernen eines Signals von einer Signalmenge
sigemptyset	Initialisierung einer leeren Signalmenge
sigfillset	Initialisierung einer kompletten Signalmenge
sigismember	Test, ob ein Signal in einer Menge enthalten ist
sigpending	Rückgabe der aktuell angekommenen, aber verzögerten Signale
sigprocmask	Setzen der Menge der vom Prozess blockierten Signale
sigsuspend	Änderung der Liste der blockierten Signale und Warten auf Ankunft und Behandlung eines Signals

Einschränkungen der Standardsignale

- POSIX 1003.1 Signale haben folgende Einschränkungen:
  - Es existieren zu wenige Benutzersignale (`SIGUSR1` und `SIGUSR2`)
  - Signale besitzen keine Prioritäten
  - Blockierte Signale können verloren gehen (beim Auftreten mehrerer gleicher Signale)
  - Das Signal enthält keinerlei Informationen zur Unterscheidung von anderen Signalen gleichen Typs (z.B. Absender)

Erweiterungen in POSIX 1003.1b

- Zur Benutzung von Echtzeitsystemen sind in POSIX 1003.1b folgende Erweiterungen vorgenommen worden:
  - Einen Menge von nach Priorität geordneten Signalen, die Benutzern zur Verfügung stehen (Bereich von `SIGRTMIN` bis `SIGRTMAX`)
  - Einen Warteschlangenmechanismus zum Schutz vor Signalverlust
  - Mechanismen zur Übertragung von weiteren Informationen
  - schnellere Signallieferung beim Ablauf eines Timers, bei Ankunft einer Nachricht an einer leeren Nachrichtenwarteschlange, bei Beendigung einer I/O-Operation
  - Funktionen, die eine schnellere Reaktion auf Signale erlauben

POSIX Funktionen für Signale

- POSIX 1003.1b definiert folgende zusätzliche Funktionen:

Funktion	**Bedeutung**
sigqueue	Sendet ein Signal inklusive identifizierende Botschaften an Prozess
sigtimedwait	Wartet auf ein Signal für eine bestimmte Zeitdauer, wird ein Signal empfangen, so wird es mitsamt der Signalinformation zurückgeliefert
sigwaitinfo	Wartet auf ein Signal und liefert das Signal mitsamt Information zurück

Beispiel: Programmierung von Signalen

- Im Folgenden wird der Code für ein einfaches Beispiel dargestellt: die periodische Ausführung einer Funktion.
- Der Code besteht aus folgenden Codeabschnitten:
  - Initialisierung eines Timers und der Signale
  - Setzen eines periodischen Timers
  - Wiederholtes Warten auf den Ablauf des Timers
  - Löschen des Timers
  - Hauptfunktion

Beispiel: Programmierung von Signalen I

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <signal.h>

int main ()
{
    int i=0;
    int test;
    struct timespec current_time;
    struct sigevent se;
    sigset_t set;   /* our signal set */
    timer_t timerid; /* timerid of our timer */
    struct itimerspec timer_sett; /* timer settings */

    timer_sett.it_interval.tv_sec = 0;          /* periodic interval length s */
    timer_sett.it_interval.tv_nsec = 500000000; /* periodic interval length ns */
    timer_sett.it_value.tv_sec = 0;             /* timer start time s */
    timer_sett.it_value.tv_nsec = 500000000; /* timer start time ns */
```

Beispiel: Programmierung von Signalen II

```
    se.sigev_notify = SIGEV_SIGNAL;  /* timer should send signals */
    se.sigev_signo = SIGUSR1;     /* timer sends signal SIGUSR1 */

    sigemptyset(&set); /* initialize signal set */
    sigaddset(&set, SIGUSR1); /* add signal which will be catched */
    timer_create(CLOCK_REALTIME, &se, &timerid); /* create timer */
    timer_settime(timerid, 0, &timer_sett, NULL);  /* set time settings for timer */

    for (i = 0; i < 5; i++)
    {
        sigwait(&set,&test); /* wait for signal defined in signal set */
        clock_gettime(CLOCK_REALTIME, &current_time); /* retrieve startup time */
        printf("Hello\n");
    }

    timer_delete(timerid); /* delete timer */
    return 0;
}
```

Semaphore zur Vermittlung von Ereignissen

- Semaphore können neben der Anwendung des wechselseitigen Ausschlusses auch zur Signalisierung von Ereignissen verwendet werden.
- Es ist zulässig, dass Prozesse (Erzeuger) Semaphore andauernd freigeben und andere Prozesse (Verbraucher) Semaphore dauern konsumieren.
- Es können auch benannte Semaphoren erzeugt werden, die dann über Prozessgrenzen hinweg verwendet werden können.
- Notwendige Funktionen sind dann:
  - sem_open(): zum Erzeugen und / oder Öffnen eines benannten Semaphors
  - sem_unlink(): zum Löschen eines benannten Semaphors

Signalisierung durch Semaphore: Beispiel

- Beispiel: ein Prozeß **Worker** wartet auf einen Auftrag (abgespeichert z.B. in einem char-Array job) durch einen Prozess **Contractor**, bearbeitet diesen und wartet im Anschluß auf den nächsten Auftrag:

Worker*:

```
while(true)
{
        down(sem); /*wait for
                   next job*/
        execute(job);
}
```

Contractor*:

```
...
job=...  /*create new job and save
         address in global variable*/
up(sem); /*signal new job*/
...
```

** sehr stark vereinfachte Lösung, da zu einem Zeitpunkt nur ein Job verfügbar sein darf*