

Nebenläufigkeit

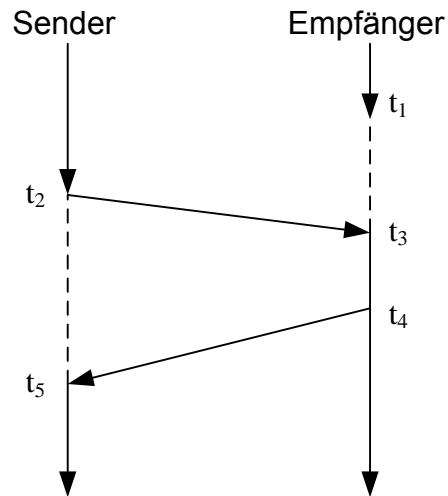
Interprozesskommunikation (IPC)

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

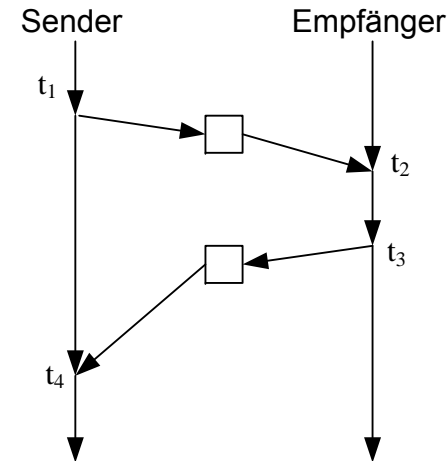
Synchron vs. Asynchron

Synchrone Kommunikation



- t₁ : Empfänger wartet auf Nachricht
- t₂ : Sender schickt Nachricht und blockiert
- t₃ : Empfänger bekommt Nachricht, die Verarbeitung startet
- t₄ : Verarbeitung beendet, Antwort wird gesendet
- t₅ : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



- t₁ : Sender schickt Nachricht an Zwischenspeicher und arbeitet weiter
 - t₂ : Empfänger liest Nachricht
 - t₃ : Empfänger schreibt Ergebnis in Zwischenspeicher
 - t₄ : 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
- Synchroner 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)`.

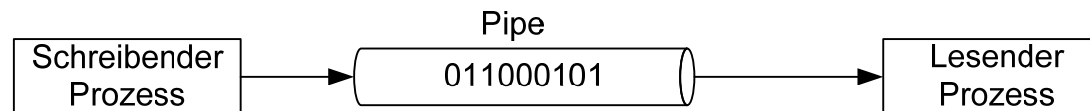
Fragestellungen beim Datenaustausch

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

In diesem Kapitel 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

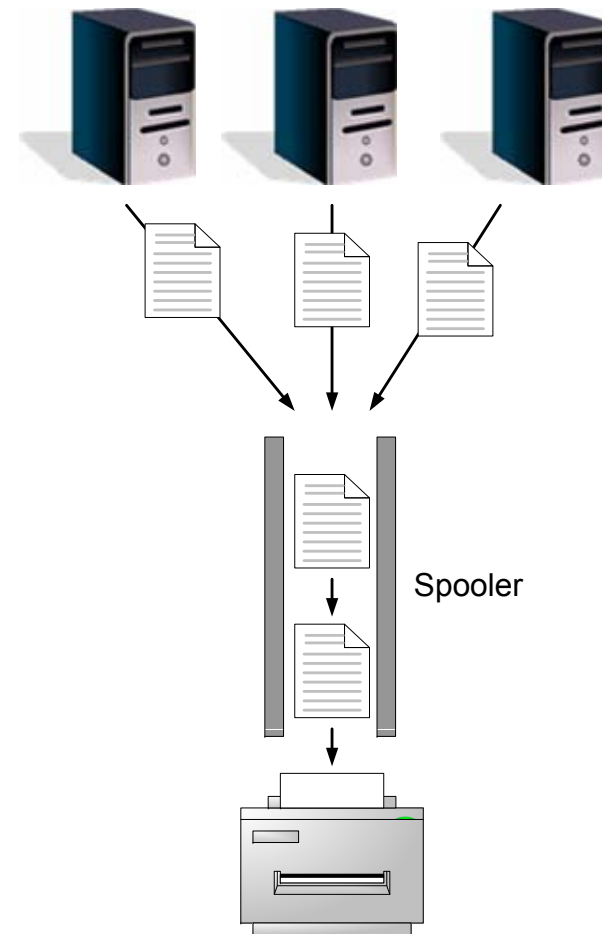
- Pipes bringen einige Nachteile mit sich:
 - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
 - Daten sind nicht priorisierbar.
 - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung: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

Nachrichtenschlangen (message queues)

- Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenschlangen, 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.

Nachrichtwarteschlangen

- 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 Nachrichtenwarteschlangen:

```
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.

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 Semaphore 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*