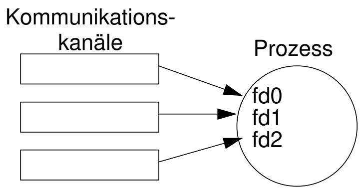
10. Tafelübung

- select (BSD, XPG4)
- Terminaltreiber konfigurieren (tcgetattr, tcsetattr)
- Pseudoterminals

select

- Prozesse die mit Pipes und Sockets arbeiten, müssen oft von verschiedenen Filedeskriptoren lesen bzw. schreiben



- diese Lese- bzw. Schreiboperationen können blockieren
- Lösungsmöglichkeiten
  - ◆ Angabe von **`O_NOBLOCK`** bei open; Problem: aktives Warten
  - ◆ fork eines Prozesses pro Filedeskriptor; Problem: teuer
  - ◆ Verwenden von **`select()`**

select

- Prototyp

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
          fd_set *errorfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

- blockiert so lange, bis einer der Filedeskriptoren bereit ist oder bis die timeout-Zeit abgelaufen ist
- timeout kann auch **`NULL`** sein (endlos warten)

select

- select markiert FD als lesbar aber read gibt 0 zurück?
  - ◆ andere Seite hat die Verbindung geschlossen

Prozeßgruppen

- jeder Prozeß gehört zu einer Prozeßgruppe; diese wird an Kinder vererbt
- Signale können an alle Prozesse einer Gruppe geschickt werden
- Ermitteln der Prozeßgruppe des eigenen Prozesses (POSIX):

```
#include <unistd.h>
pid_t getpgrp(void);
```

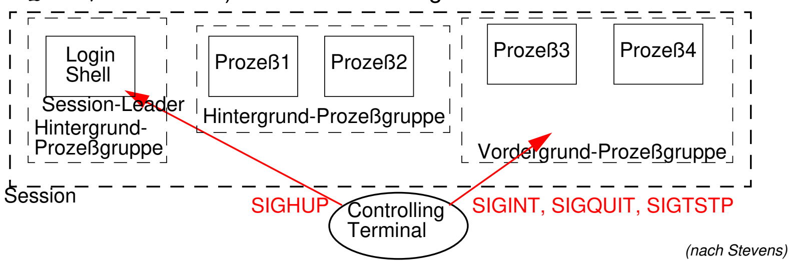
- Beitreten oder Erzeugen einer Prozeßgruppe (POSIX):

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

- ein Prozeß der Gruppe ist der Prozeßgruppenführer (process group leader) (pid == pgid)

Sessions

- eine Session enthält eine Vordergrund-Prozeßgruppe und mehrere Hintergrundprozeßgruppen
- einer Session kann ein Steuerterminal zugeordnet sein
- eine Session besitzt einen Session-Leader; dieser bekommt das **`SIGHUP`** vom Terminal zugestellt
- die Vordergrundgruppe bekommt die Jobcontrol-Signale (**`SIGINT, SIGQUIT, SIGTSTP`**) vom Terminal zugestellt



(nach Stevens)

Sessions

- mehrere Prozeßgruppen können zu einer Session gehören
- Erzeugen einer neuen Session

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
```

  - ◆ wenn Prozeß ein Prozeßgruppenleader ist: Fehler
  - ◆ wenn Prozeß kein Prozeßgruppenleader ist:
    - neue Session wird erzeugt mit Prozeß als Session Leader
    - neue Prozeßgruppe wird erzeugt mit Prozeß als Prozeßgruppenleader
    - Prozeß hat kein Controlling Terminal mehr

Sessions

- Zuordnen eines Controlling Terminals zu einer Session
  - ◆ POSIX spezifiziert nicht, wie einer Session ein Controlling Terminal zugeordnet wird
  - ◆ SVID: nächstes geöffnetes Terminal wird automatisch Controlling Terminal
  - ◆ BSD4.3: **`ioctl()`**-Aufruf mit **`TIOCSCTTY`**
- Zugriff auf Controlling Terminal
  - ◆ open("/dev/tty", ...)
- Setzen/Abfragen der Vordergrundprozeßgruppe:
  - ◆ tcsetpgrp(), tcgetpgrp()

Ermitteln von pgid/sid/tty mit ps

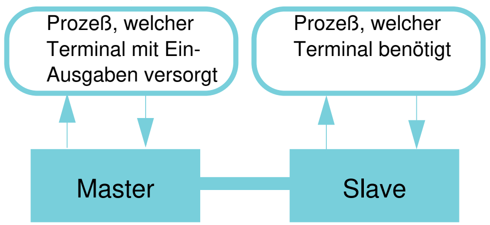
```
> xemacs &
> ps -ej | less
```

```
> ps -ej     gibt Job-Informationen aus
             wählt alle Prozesse aus

  PID  PGID   SID TTY          TIME CMD
    1     0     0 ?        00:00:07 init
    2     1     1 ?        00:00:01 kflushd
    3     1     1 ?        00:00:00 kpiod
    4     1     1 ?        00:00:08 kswapd
17930 17930 17930 pts/0    00:00:00 tcsh        gemeinsame
17945 17945 17930 pts/0    00:00:48 xemacs      Session
20093 20093 20093 ttyp0    00:00:00 gdb
20260 20260 20093 ttyp0    00:00:00 ryash
30339 30339 30339 tty1     00:00:00 getty
17830 17830 17830 pts/2    00:00:00 tcsh
17889 17889 17930 pts/0    00:00:00 ps         gemeinsame
17890 17889 17930 pts/0    00:00:00 less       Prozeßgruppe
```


Pseudo-Terminals

- Problem:
  - ◆ ein Rechner hat üblicherweise nur ein physikalisches Terminal (die Console)
  - ◆ wenn der Rechner einen Remote-Login-Dienst anbietet oder ein Fenstersystem (z.B. X11) besitzt, benötigt er mehrere "virtuelle" Terminals
- neben echten Terminals (**`/dev/console`**, **`/dev/ttyS0`**, ...) stellen SVR4 und BSD4.4-Systeme sog. Pseudoterminals zur Verfügung
- diese verhalten sich wie Terminals, können aber von Programmen mit Eingaben versorgt, bzw. gelesen werden



Pseudoterminals

- BSD-Bibliotheksfunktionen **`openpty`**, **`forkpty`**

```
#include <pty.h>
int openpty ( int *amaster, int *aslave, char *name,
              struct termios *termp, struct winsize *winp)
```

  - ◆ öffnet Pseudoterminal und gibt Master- und Slave-Filedescriptor in amaster bzw. aslave zurück
  - ◆ falls **`name!=NULL`**: Dateiname des Slave-pty wird in name abgespeichert Vorsicht: falls **`name`** nicht ausreichend groß ist, kann ein Pufferüberlauf auftreten (Sicherheitsproblem)
  - ◆ falls **`termp!=NULL`**: Terminaleinstellungen werden für Slave verwendet
  - ◆ falls **`winp!=NULL`**: Slave-pty wird auf entspr. Fenstergröße eingestellt

Pseudoterminals

```
#include <pty.h>
int forkpty ( int *amaster, char *name, struct termios *termp,
              struct winsize *winp)
```

- ◆ führt dieselben Aktionen aus wie **`openpty()`**, und zusätzlich
  - Erzeugt neuen Prozeß (**`fork()`**)
  - Kind erzeugt neue Session (**`setsid()`**)
  - Kind öffnet Pseudoterminal-Slave (Slave wird zum Controlling-Terminal)
  - **`dup2()`** des Slave-Filedeskriptors nach **`stdin`**, **`stdout`**, **`stderr`**
- openpty und forkpty sind in **`libutil.a`** implementiert, d.h. Compiler (Linker) muß mit **`-lutil`** aufgerufen werden

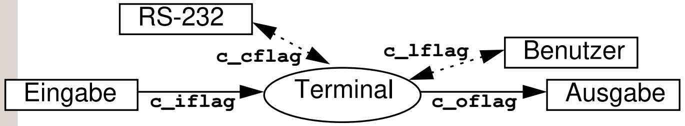
Terminaltreiber konfigurieren

```
#include <termios.h>
int tcgetattr(int fildes, struct termios *termios_p);
int tcsetattr( int fildes, int optional_actions,
               const struct termios *termios_p);
```

◆ **`fildes`**: Filedeskriptor des Terminals

◆ **`termios_p`**: Zeiger auf **`termios`**-Struktur

```
struct termios {
    tcflag_t   c_iflag;      /* input flags */
    tcflag_t   c_oflag;      /* output flags */
    tcflag_t   c_cflag;      /* control flags */
    tcflag_t   c_lflag;      /* local flags */
    cc_t       c_cc[NCCS];   /* control characters */
};
```



Terminaltreiber konfigurieren

- **`c_iflag`**: z.B. **`ICRNL`**: Umwandlung von **`'\r'`** in **`'\n'`** (POSIX)
- **`c_oflag`**: z.B. **`ONLCR`**: **`'\n'`** nach **`'\r\n'`** (SVID und BSD4.3)
- **`c_cflag`**: z.B. **`PARENB`**: parity enable (POSIX)
- **`c_lflag`**: z.B.

◆ **`ECHO`**: Zeichen auf lokalem Terminal ausgeben (POSIX)

◆ **`ISIG`**: spezielle Eingabezeichen erzeugen Signale

◆ **`ICANON`**: kanonische Eingabeverarbeitung (auch als cooked-mode bekannt)

  • Terminaltreiber puffert Eingaben zeilenweise

  • nicht-kanonische Eingabeverarbietung(raw-mode):
  Verhalten hängt von **`c_cc[VMIN]`** und **`c_cc[VTIME]`** ab (siehe Manual)