

# Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier  
(Lehrstuhl Informatik 4)

## Übung 9



Sommersemester 2017



## Prozesse

- Prozesse sind eine Ausführungsumgebung für Programme
  - haben eine Prozess-ID (PID, ganzzahlig positiv)
  - führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft
  - Speicher
  - Adressraum
  - geöffnete Dateien
  - ...
- Visualisierung von Prozessen: `ps`, `htop`



## Inhalt

Prozesse  
System-Schnittstelle  
Minimale Shell  
Einlesen von der Standard-Eingabe  
Stringmanipulation mit `strtok`

Signale  
Allgemein  
Signale zustellen  
Signale maskieren  
Signale behandeln  
Auf Signale warten  
Signale vs. Interrupts

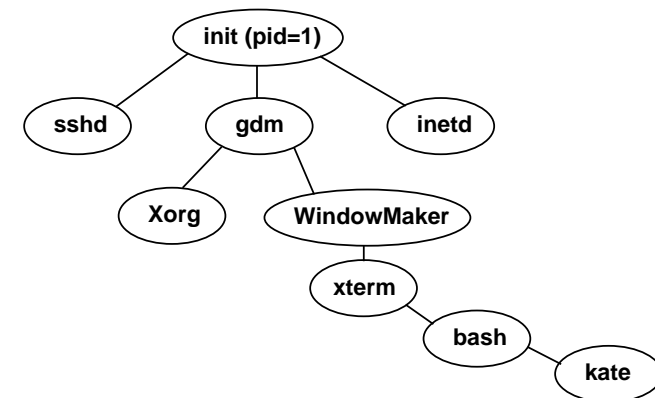
Aufgabe: `mish`

Hands-on: `run`



## Prozesshierarchie

- Zwischen Prozessen bestehen Vater-Kind-Beziehungen
  - der erste Prozess wird direkt vom Systemkern gestartet (z.B. `init`)
  - es entsteht ein Baum von Prozessen bzw. eine Prozesshierarchie



- `kate` ist ein Kind von `bash`, `bash` wiederum ein Kind von `xterm`



## Kindprozess erzeugen – fork

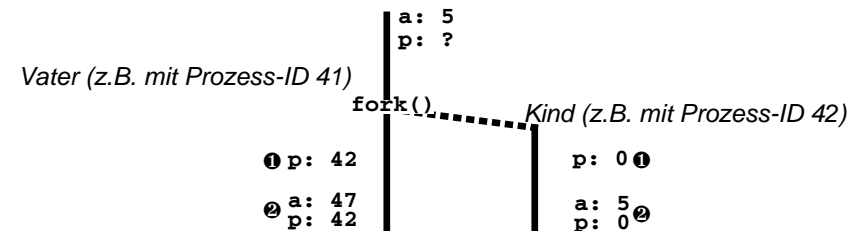
```
1 pid_t fork(void);
```

- Erzeugt einen neuen Kindprozess
- Exakte **Kopie** des Vaters...
  - Datensegment (neue Kopie, gleiche Daten)
  - Stacksegment (neue Kopie, gleiche Daten)
  - Textsegment (gemeinsam genutzt, da nur lesbar)
  - Filedeskriptoren (geöffnete Dateien)
  - ...mit Ausnahme der Prozess-ID
- Kind startet Ausführung hinter dem `fork()` mit geerbtem Zustand
- Vater-/Kindprozess am **Rückgabewert** von `fork(2)` unterscheidbar
  - Vater: PID des Kindes
  - Kind: 0
  - Fehler: -1



## Kindprozess erzeugen – fork

```
1 int a=5;
2 pid_t p = fork(); // (1)
3 a += p; // (2)
4 switch(p) {
5     case -1: // Fehler - kein Kind
6         ...
7     case 0: // Kind
8         ...
9     default: // Vater
10        ...
11 }
```



## Programm ausführen – exec

- Mit Angabe des vollen Pfads der Programm-Datei in `path`

```
1 int execl(const char *path, const char *arg0, ... /*, NULL */);
2 int execv(const char *path, char *const argv[]);
```

- Zum Suchen von `file` wird die Umgebungsvariable `PATH` verwendet

```
1 int execlp(const char *file, const char *arg0, ... /*, NULL */);
2 int execvp(const char *file, char *const argv[]);
```

- Lädt **Programm** zur Ausführung in den **aktuellen Prozess**

- aktuell ausgeführtes Programm wird ersetzt:
  - Text-, Daten- und Stacksegment
- erhalten bleiben:
  - Filedeskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...

- Aufrufparameter für `exec(3)`

- Pfad bzw. Dateiname des neuen Programmes
- Argumente für die `main`-Funktion



## Beispiele zu exec

- Mit absolutem Pfad und einer statischen Liste

```
1 execl("/bin/cp", "/bin/cp", "x.txt", "y.txt", NULL);
```

- Mit Suche in `PATH` und einer statischen Liste

```
1 execlp("cp", "cp", "x.txt", "y.txt", NULL);
```

- Mit Suche in `PATH` und einer veränderbar großen Liste

```
1 char *args[4];
2 args[0] = "cp";
3 args[1] = "x.txt";
4 args[2] = "y.txt";
5 args[3] = NULL;
6 execvp(args[0], args);
```

- Anmerkungen

- Alle Varianten von `exec(3)` erwarten als letzten Eintrag in der Argumentenliste einen `NULL`-Zeiger
- Alle Varianten von `exec(3)` kehren nur im Fehlerfall zurück



## PATH Umgebungsvariable

- Enthält Pfade zu ausführbaren Programmen

- Ausführen von echo:

```
1 $ echo "Foo"
2 Foo
```

- Aber Ausführen von hello\_world:

```
1 $ ./hello_world
2 Hello World
```

- PATH Variable ausgeben:

```
1 $ echo $PATH
2 /proj/i4/bin:/bin:/usr/bin:/usr/local/bin:/sbin:/local/bin
3 $ which -a echo
4 /bin/echo
5 /local/bin/echo
```

- PATH Variable verändern:

```
1 export PATH=/path/to/executables:$PATH
```



## Prozess beenden – exit

```
1 void exit(int status);
```

- Beendet aktuellen Prozess mit angegebenem Exitstatus
- Gibt alle Ressourcen frei, die der Prozess belegt hat
  - Speicher
  - Filedeskriptoren (schließt alle offenen Dateien)
  - Kerndaten, die für die Prozessverwaltung verwendet wurden
  - ...
- Prozess geht in den *Zombie*-Zustand über
  - ermöglicht Vater auf Terminieren des Kindes zu reagieren
  - Zombie-Prozesse belegen Ressourcen & sollten zeitnah beseitigt werden
  - ist der Vater schon vor dem Kind terminiert
    1. Zombie-Prozess wird an init-Prozess (PID 1) weitergereicht
    2. init-Prozess beseitigt Zombie sofort



## Auf Kindprozess warten – wait

- Warten auf die Beendigung von Kind-Prozessen (Rückgabe: PID)

```
1 pid_t wait(int *statusbits);
```

- Beispiel

```
1 int main(int argc, char *argv[]) {
2     pid_t pid;
3     pid = fork();
4     if (pid > 0) { /* Vater */
5         int stbits;
6         wait(&stbits); /* Fehlerbehandlung nicht vergessen! */
7         printf("Kindstatus: %x", stbits); /* nackte Status-Bits */
8     } else if (pid == 0) { /* Kind */
9         execl("/bin/cp", "/bin/cp", "x.txt", "y.txt", NULL);
10        /* diese Stelle wird nur im Fehlerfall erreicht */
11        perror("exec /bin/cp"); exit(EXIT_FAILURE);
12    } else {
13        /* pid == -1 --> Fehler bei fork */
14    }
15 }
```



## Auf Kindprozess warten – wait

- wait(2) blockiert, bis ein Kind-Prozess terminiert wird
- PID dieses Kind-Prozesses wird als Rückgabewert geliefert
- als Parameter kann ein Zeiger auf einen int-Wert mitgegeben werden, in dem u.a. der Exitstatus des Kind-Prozesses abgelegt wird
- Status-Bits enthalten Grund des Terminierens
- Makros erleichtern Abfrage
  - Prozess mit exit(3) terminiert: WIFEXITED(stbits)  
⇒ Exitstatus: WEXITSTATUS(stbits)
  - Prozess durch Signal abgebrochen: WIFSIGNALED(stbits)  
⇒ Nummer des Signals: WTERMSIG(stbits)
- Weitere Makros: siehe Dokumentation wait(2)



## Auf Kindprozess warten – waitpid

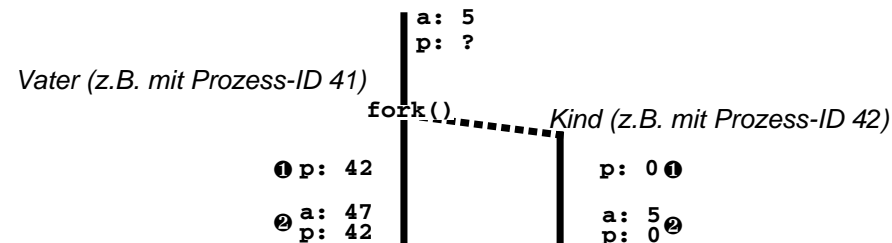
```
1 pid_t waitpid(pid_t pid, int *status, int options);
```

- warten auf bestimmten Prozess
  - `pid > 0` Kindprozess mit Prozess-ID `pid`
  - `pid = -1` Beliebige Kindprozesse
  - ...
- Optionen:
  - `WNOHANG` sofort zurückkehren, wenn kein Kind beendet wurde (nicht blockieren)
  - ...
- Rückgabewert
  - `> 0` Prozess-ID des Kindprozesses
  - `0` kein Prozess beendet (bei Verwendung von `WNOHANG`)
  - `-1` Fehler – Details siehe `waitpid(2)`



## Funktionsweise einer minimalen Shell

- Auf Eingaben vom Benutzer warten
- Neuen Prozess erzeugen
- Kind: Startet Programm
- Vater: Wartet auf die Beendigung des Kindes
- Ausgabe der Kindzustands



## Einlesen von der Standard-Eingabe mit fgets

```
1 char *fgets(char *s, int size, FILE *stream);
```

- `fgets(3)` liest eine Zeile vom übergebenen Eingabe-Kanal und schreibt diese in einen vorher angelegten Speicherbereich
- Maximal `size-1` Zeichen werden gelesen und mit `'\0'` abgeschlossen
- Das `'\n'` am Ende der Zeile wird auch gespeichert
- Rückgabewert ist der Zeiger auf den übergebenen Speicherbereich; oder `NULL` am Ende der Eingabe oder im Fehlerfall
  - ⇒ Unterscheidung End-Of-File und Fehler mit `feof(3)` oder `ferror(3)`
- Beispiel

```
1 char buf[23];  
2 while (fgets(buf, 23, stdin) != NULL) { /* Fehlerüberprüfung! */  
3     /* buf enthält die eingelesene Zeile */  
4 }
```



## Stringmanipulation mit strtok

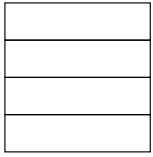
```
1 char *strtok(char *str, const char *delim);
```

- `strtok(3)` teilt einen String in Tokens auf, die durch bestimmte Trennzeichen (engl. delimiter) getrennt sind
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token
  - `str` ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen `NULL`
  - `delim` ist ein String, der alle Trennzeichen enthält, z.B. `" \t\n"`
- Direkt aufeinanderfolgende Trennzeichen in `str` werden übersprungen
- Trennzeichen nach Token werden durch `'\0'` ersetzt
- Ist das Ende des Strings erreicht, gibt `strtok(3)` `NULL` zurück



## Stringmanipulation mit strtok

cmdline → ls -l /tmp\0

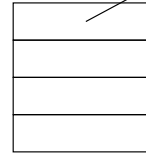


```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```



## Stringmanipulation mit strtok

cmdline → ls\0-l /tmp\0

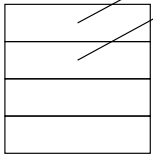


```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```



## Stringmanipulation mit strtok

cmdline → ls\0-l\0/tmp\0

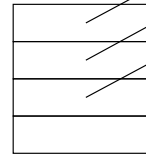


```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```



## Stringmanipulation mit strtok

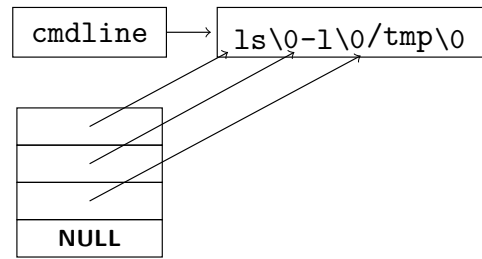
cmdline → ls\0-l\0/tmp\0



```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```



## Stringmanipulation mit strtok



```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```

## Inhalt

Prozesse

Signale

- Allgemein
- Signale zustellen
- Signale maskieren
- Signale behandeln
- Auf Signale warten
- Signale vs. Interrupts

Aufgabe: mish

Hands-on: run

## Signale

- Vergleichbar mit Interrupts beim AVR
- Standardbehandlungen für Signale bereits vorhanden
- Verwendung von Signalen
  - Ereignissignalisierung des Betriebssystemkerns an einen Prozess
  - Ereignissignalisierung zwischen Prozessen
- Zwei Arten von Signalen
  - synchrone Signale: durch Prozessaktivität ausgelöst (Trap / Falle)
    - ⇒ Zugriff auf ungültigen Speicher, ungültiger Befehl
  - asynchrone Signale: "von außen" ausgelöst (Interrupts / Unterbrechung)
    - ⇒ Timer, Tastatureingabe

## Ausgewählte POSIX-Signale

- Das Standardverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps
  - SIGALRM (Term): Timer abgelaufen (alarm(2), setitimer(2))
  - SIGCHLD (Ign): Statusänderung eines Kindprozesses
  - SIGINT (Term): Interrupt (Shell: CTRL-C)
  - SIGQUIT (Core): Quit (Shell: CTRL-@)
  - SIGKILL (nicht behandelbar): beendet den Prozess
  - SIGTERM (Term): Terminierung; Standardsignal für kill(1)
  - SIGSEGV (Core): Speicherschutzverletzung
  - SIGUSR1, SIGUSR2 (Term): Benutzerdefinierte Signale
- Siehe auch signal(7)

## Signal zustellen

- Kommando `kill(1)` aus der Shell

```
1 kill -USR1 <pid>
```

- Parameter: Signalnummer oder Signal ohne "SIG"

- Systemaufruf `kill(2)`

```
1 int kill(pid_t pid, int signo);
```



## Setzen der prozessweiten Signal-Maske

- Konfiguration mit Hilfe einer Variablen vom Typ `sigset_t`
- Hilfsfunktionen konfigurieren das Signalset
  - `sigemptyset(3)`: alle Signale aus Maske entfernen
  - `sigfillset(3)`: alle Signale in Maske aufnehmen
  - `sigaddset(3)`: Signal zur Maske hinzufügen
  - `sigdelset(3)`: Signal aus Maske entfernen
  - `sigismember(3)`: Abfrage, ob Signal in Maske enthalten ist
- Analogie zum AVR: EIMSK-Register



## Setzen der prozessweiten Signal-Maske

- Setzen einer Maske mit

```
1 int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

- `how`: Operation
  - `SIG_SETMASK`: setzt `set`
  - `SIG_BLOCK`: blockiert zusätzlich die in `set` gesetzten Signale
  - `SIG_UNBLOCK`: deblockiert die in `set` gesetzten Signale
- `set`: Parameter für die Operation
- `oset`: Speicher für aktuell installierte Maske

- Beispiel

```
1 sigset_t set;  
2 sigemptyset(&set);  
3 sigaddset(&set, SIGUSR1);  
4 sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
```

- Anwendung: z.B. kritische Abschnitte (vgl. `cli()`, `sei()`)

!! Die prozessweite Signal-Maske wird über `exec(3)` vererbt !!



## sigaction - Signalhandler

- AVR-Analogie: `ISR(..)`, Alarmhandler
- Konfiguration mit Hilfe der Struktur `sigaction`

```
1 struct sigaction {  
2     void (*sa_handler)(int); /* Behandlungsfunktion */  
3     sigset_t sa_mask; /* Signalmaske während der Behandlung */  
4     int sa_flags; /* Diverse Einstellungen */  
5 }
```

- Signalbehandlung kann über `sa_handler` konfiguriert werden:
  - `SIG_IGN`: Signal ignorieren
  - `SIG_DFL`: Default-Signalbehandlung einstellen
  - Funktionsadresse: Funktion wird in der Signalbehandlung aufgerufen  
Als Parameter wird die Signalnummer übergeben
- `SIG_IGN` und `SIG_DFL` werden über `exec(3)` vererbt, nicht aber eine Behandlungsfunktion (nicht möglich, warum?)



## sigaction - Maske

- Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {
2     void (*sa_handler)(int); /* Behandlungsfunktion */
3     sigset_t sa_mask; /* Signalmaske während der Behandlung */
4     int sa_flags; /* Diverse Einstellungen */
5 }
```

- sa\_mask wird während der Ausführung des Signalhandlers gesetzt  
⇒ sigprocmask()



## sigaction - Flags

- Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {
2     void (*sa_handler)(int); /* Behandlungsfunktion */
3     sigset_t sa_mask; /* Signalmaske während der Behandlung */
4     int sa_flags; /* Diverse Einstellungen */
5 }
```

- sa\_flags beeinflussen das Verhalten beim Signalempfang
- Bei uns gilt: sa\_flags=SA\_RESTART



## Setzen der Signalbehandlung

- Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {
2     void (*sa_handler)(int); /* Behandlungsfunktion */
3     sigset_t sa_mask; /* Signalmaske während der Behandlung */
4     int sa_flags; /* Diverse Einstellungen */
5 }
```

- Konfiguration Setzen

```
1 #include <signal.h>
2
3 int sigaction(int sig, const struct sigaction *act,
4              struct sigaction *oact);
```

- sig: Signalnummer
- act: Zu installierende Konfiguration
- oact: Speicher für die aktuell installierte Konfiguration



## sigaction - Beispiel

- sigaction

```
1 struct sigaction {
2     void (*sa_handler)(int); /* Behandlungsfunktion */
3     sigset_t sa_mask; /* Signalmaske während der Behandlung */
4     int sa_flags; /* Diverse Einstellungen */
5 }
```

- Installieren eines Handlers für SIGUSR1

```
1 #include <signal.h>
2
3 void my_handler(int sig) {
4     ...
5 }
6
7 int main(int argv, char * argv[]){
8     struct sigaction action;
9     action.sa_handler = my_handler;
10    sigemptyset(&action.sa_mask);
11    action.sa_flags = SA_RESTART;
12    sigaction(SIGUSR1, &action, NULL);
13    ....
```



# Warten auf Signale

- Problem: In einem kritischen Abschnitt auf ein Signal warten
  1. Signal deblockieren
  2. Passiv auf Signal warten (*schlafen* legen)
  3. Signal blockieren
  4. Kritischen Abschnitt bearbeiten
- Operationen müssen atomar am Stück ausgeführt werden!  
⇒ gleiche Problematik wie bei den Stromsparmodi des AVR-Prozessors

## ■ Sigsuspend

```
1 #include <signal.h>  
2 int sigsuspend(const sigset_t *mask);
```

1. sigsuspend(mask) setzt mask als Signal-Maske
2. Der Prozess blockiert bis zum Eintreffen eines Signals
3. Der Signalhandler wird ausgeführt
4. sigsuspend() setzt die ursprüngliche Signal-Maske und kehrt zurück



# POSIX-Signale vs. AVR-Interrupts

## ■ Vergleich

	Interrupts	Signale
Behandlung installieren	ISR()-Makro	sigaction(2)
Auslösung	Hardware	Prozesse mit kill() oder Betriebssystem
Synchronisation	cli(), sei()	sigprocmask(2)
Warten auf Signale	sei(); sleep_cpu()	sigsuspend(2)

- Signale und Interrupts sind sehr **ähnliche Konzepte** auf unterschiedlichen Ebenen
- Viele Probleme treten in beiden Fällen auf und sind konzeptionell identisch zu lösen



# Inhalt

Prozesse

Signale

Aufgabe: mish

Hands-on: run



# Aufgabe: mish - Teil a

- Einfache Shell ("mini shell") zum Ausführen von Kommandos
  - Ausgabe des Shell-Prompt und Warten auf Eingaben: fgets(3)
  - Zerlegen der Eingaben in Kommandoname und Argumente: strtok(3)
  - Starten des Kommandos in neuem Prozess: fork(2), execvp(3)
  - Warten auf Terminierung & Ausgabe des Exitstatus: wait(2)

```
1 # Reguläre Beendigung durch Exit (Exitstatus = 0)  
2 mish> ls -l  
3 ...  
4 Exit status [2110] = 0  
5  
6 # Beendigung durch Signal (hier SIGINT = 2)  
7 mish> sleep 10  
8 Signal [1302] = 2
```

- Anpassen der Signalbehandlungen für SIGINT (sigaction(2))
  - Vater: Signal ignorieren (SIG\_IGN)
  - Kind: Default-Behandlung (SIG\_DFL)



## Aufgabe: mish - Teil b

- Unterstützung von Hintergrundprozessen
  - Starten im Hintergrund mit "&" am Ende des Befehls
  - Beispiel: ./sleep 10 &
  - Ausgabe der Prozess-ID und des Prompts
  - Anschließend sofort neue Befehle entgegennehmen
- Aufsammeln der Zombie-Prozesse
  - Signal SIGCHLD zeigt Statusänderung von Kindprozessen an
  - Aufsammeln mit waitpid(2) im Hauptprogramm
  - Weitere relevante Bibliotheksfunktionen: sigaction(2), sigprocmask(2), sigsuspend(2)

```
1 # Starten eines Hintergrundprozesses mit &
2 mish> sleep 10 &
3 Started [2110]
4 mish> ...
5 ...
6 Exit status [2110] = 0
```



## Übersicht der Konzepte (1)

### 1. Neuen Prozess erstellen: fork(2)

```
1 pid_t p = fork();
2 switch(p) {
3     case -1: // Fehler - kein Kind
4         ...
5     case 0: // Kind
6         ...
7     default: // Vater
8         ...
9 }
```

### 2. Programm im aktuellen Prozess starten: exec(3)

```
1 char *args[4];
2 args[0] = "cp";
3 args[1] = "x.txt";
4 args[2] = "y.txt";
5 args[3] = NULL;
6 execvp(args[0], args);
```



## Übersicht der Konzepte (2)

### 3. Auf Prozessbeendigung warten: wait(2)

```
1 int stbits;
2 wait(&stbits); /* Fehlerbehandlung nicht vergessen! */
```

```
1 int stbits;
2 pid_t pid = waitpid(-1, &stbits, WNOHANG);
3 /* beliebiger Kindprozess; nicht blockieren */
```

### 4. Signalhandler installieren: sigaction(2)

```
1 struct sigaction act;
2 act.sa_handler = SIG_DFL; // Handlersignatur: void f(int signum)
3 act.sa_flags = SA_RESTART;
4 sigemptyset(&act.sa_mask);
5 sigaction(SIGINT, &act, NULL);
```



## Übersicht der Konzepte (3)

### 5. Signale blockieren/deblockieren: sigprocmask(2)

```
1 sigset_t set;
2 sigemptyset(&set);
3 sigaddset(&set, SIGUSR1);
4 sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
5 // kritischer Abschnitt
6 sigprocmask(SIG_UNBLOCK, &set, NULL); /* Deblockiert SIGUSR1 */
```

### 6. Auf Signale warten: sigsuspend(2)

```
1 sigprocmask(SIG_BLOCK, &set, &old); /* Blockiert Signale */
2 while(event == 0){
3     sigsuspend(&old); /* Wartet auf Signale */
4 }
5 sigprocmask(SIG_SETMASK, &old, NULL); /* Deblockiert Signale */
```



- Testprogramme unter `/proj/i4spic/pub/aufgabe8`

- `spic-wait`

```
1 /proj/i4spic/pub/aufgabe8/spic-wait
2 My PID: 20746
3 Try
4 kill 20746
5   to terminate without core dump (SIGTERM)
6 kill -QUIT 20746
7   to terminate with core dump (SIGQUIT)
```

- `spic-exit`

```
1 /proj/i4spic/pub/aufgabe8/spic-exit 12
2 Exiting with status 12
```



```
1 ./run <programm> <param0> [<params> ...]
```

- `run` erhält einen Programmnamen und eine Liste mit Parametern
  - erstellt für jeden Parameter einen neuen Prozess
  - führt das angegebene Programm aus und übergibt den zugehörigen Parameter
  - wartet auf dessen Beendigung und behandelt nächsten Parameter
- Aufrufbeispiel: `./run echo Auto Haus Katze`
- Generierte Programmaufrufe:
  - `echo Auto`
  - `echo Haus`
  - `echo Katze`
- Bibliotheksfunktionen: `fork(2)`, `exec(3)`, `wait(2)`
- Fehlerbehandlung beachten

