Systemnahe Programmierung in C (SPiC)

19 Programme und Prozesse

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

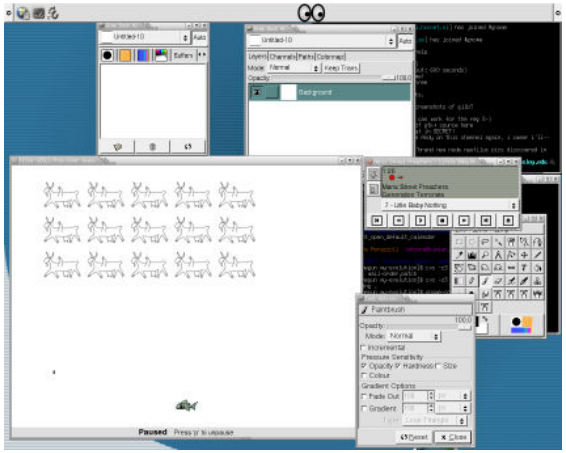
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2019

http://www4.cs.fau.de/Lehre/SS19/V_SPIC

Überblick

- **Mehrere** Programme, die
- **nebenläufig**,
- **dynamisch** gestartet/beendet
- über **definierte E/A-Funktionen**
- ihre Umgebung steuern.



Quelle: www.wikipedia.org

Jedes laufende Programm bekommt Hardware zugeteilt:

- CPU (Zeitanteile)
- Speicher (Teil des Gesamtspeichers)

und kann Betriebssystem-Kern-Funktionen aufrufen.

Definitionen

Programm: Folge von Anweisungen

Prozess: laufendes Programm mit seinen Daten

Hinweis: ein Programm kann sich mehrfach in Ausführung befinden!

Prozesse

- Definition „Prozess”: laufendes Programm mit seinen Daten
- eine etwas andere Sicht:

Prozessor	Zeitanteile am echten Prozessor
Speicher	virtueller Speicher
Interrupts	Signale
E/A-Geräte	E/A-Betriebssystem-Funktionen

Prozesse (2)

- Mehrprogrammbetrieb („Multitasking”)
  - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
  - stehen weniger Prozessoren zur Verfügung, als Prozesse ausgeführt werden sollen, werden Zeitanteile der Rechenzeit an die Prozesse vergeben (**Time Sharing System**)
  - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit bekommt, trifft der Betriebssystem-Kern (**Scheduler**)
  - die Umschaltung zwischen Prozessen erfolgt durch den Betriebssystem-Kern (**Dispatcher**)
  - laufende Prozesse wissen nicht, an welchen Stellen auf andere Prozesse umgeschaltet wird

Prozesszustände

Ein Prozess befindet sich in einem der folgenden Zustände

Erzeugt: (New)
Prozess wurde erzeugt, besitzt aber noch nicht alle zum Laufen notwendigen Betriebsmittel

Bereit: (Ready)
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zu laufen

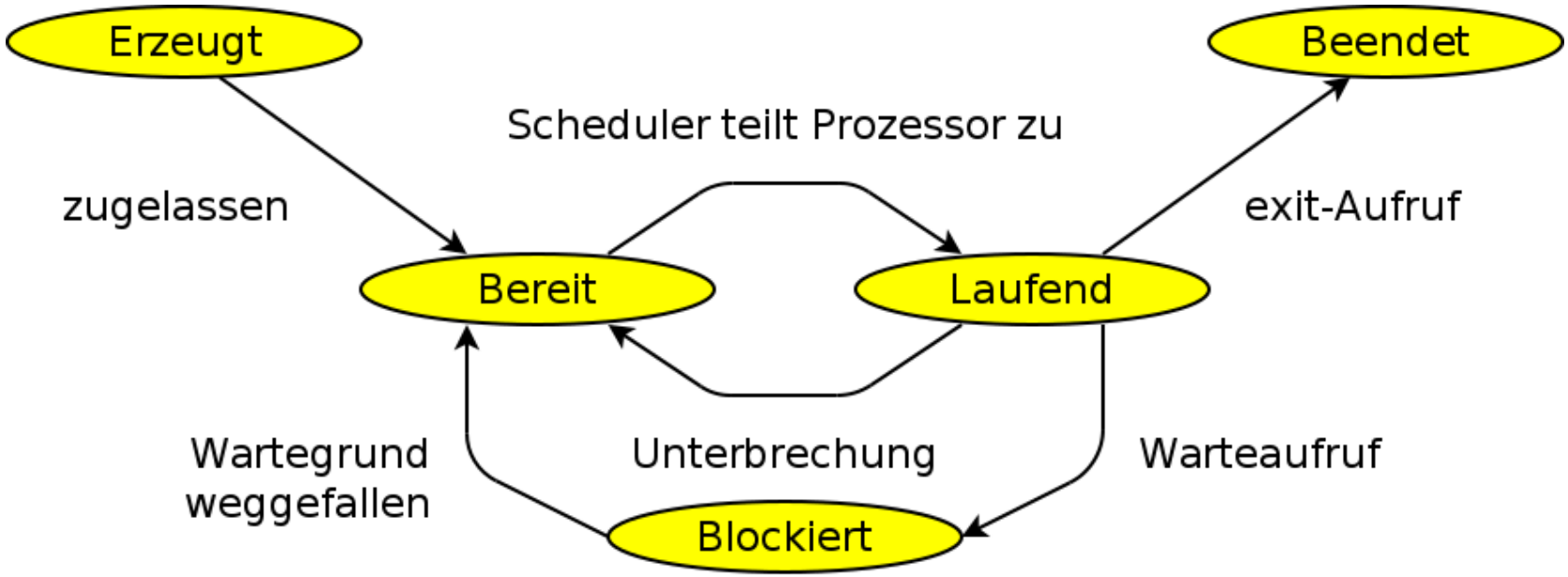
Laufend: (Running)
Prozess wird vom realen Prozessor ausgeführt

Blockiert: (Blocked)
Prozess wartet auf ein Ereignis (Fertigstellung einer Ein- oder Ausgabeoperation)

Beendet: (Terminated)
Prozess ist beendet, seine Betriebsmittel sind noch nicht alle freigegeben

Prozesszustände (2)

- Zustandsdiagramm mit Übergängen:



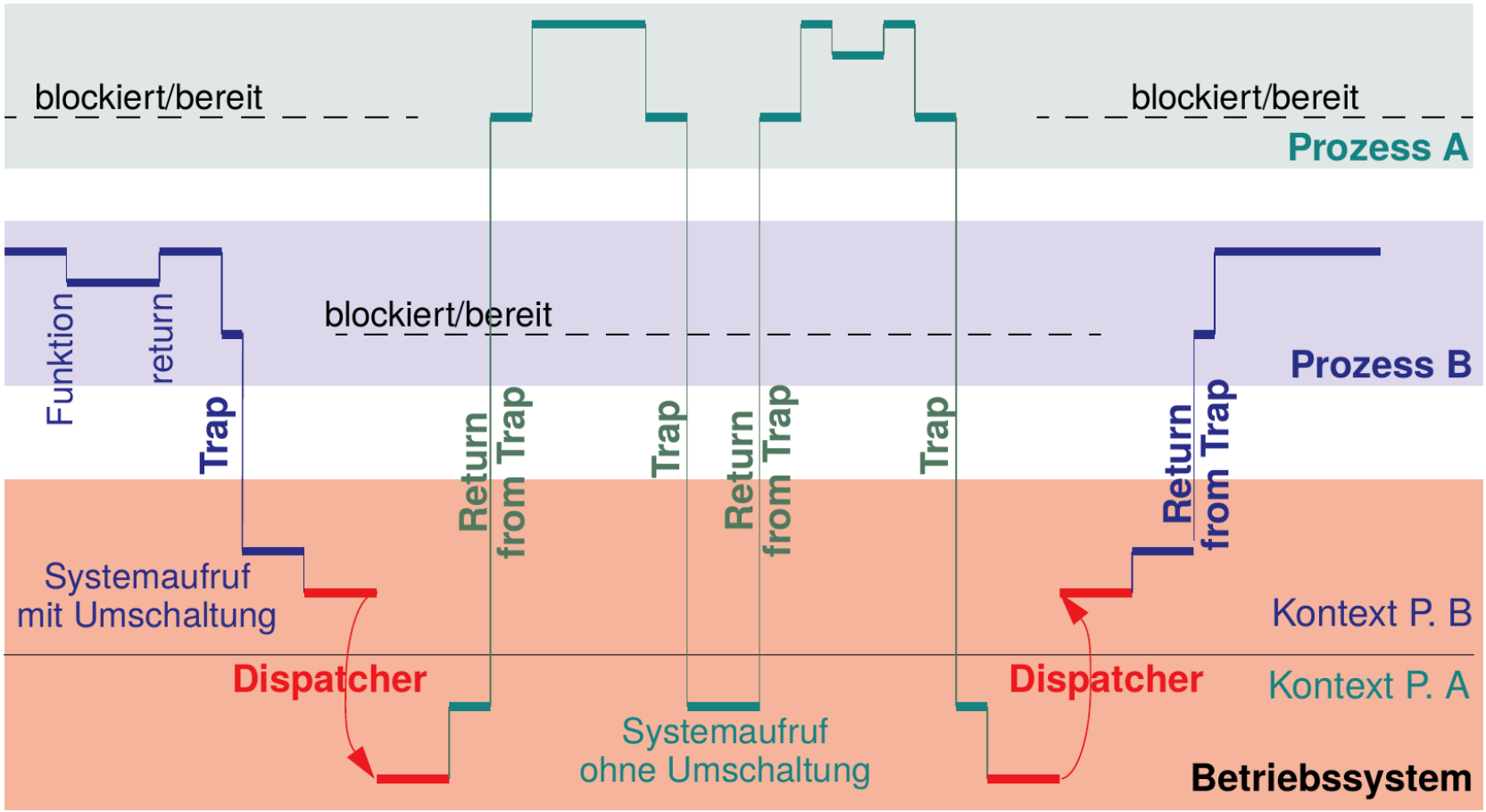
Nach Silberschatz, 1994

Prozesswechsel

- Jeder Prozess hat Zustand/Kontext
  - Registerinhalte des Prozessors
  - Inhalte der Speicherbereiche
  - offene Dateien, aktuelles Verzeichnis, ...
- Beim Prozesswechsel (Context Switch)
  - wird der Inhalt der Prozessorregister abgespeichert,
  - ein neuer Prozess ausgewählt,
  - die Ablaufumgebung des neuen Prozesses hergestellt
    - Umprogrammierung der MMU
    - Wechsel der offenen Dateien, des aktuellen Verzeichnisses, ...
  - werden die gesicherten Register des neuen Prozesses geladen.

Prozesswechsel

- Ablauf von zwei Prozessen in Benutzermodus und Kern mit Umschaltung



Prozesskontrollblock

- Prozesskontrollblock (Process Control Block – PCB)

Datenstruktur des Betriebssystem-Kerns, die alle notwendigen Daten für einen Prozess enthält.

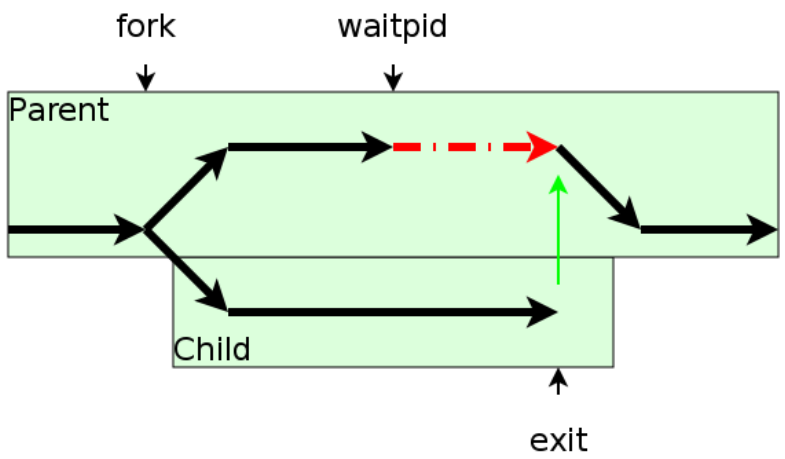
Beispiel UNIX:

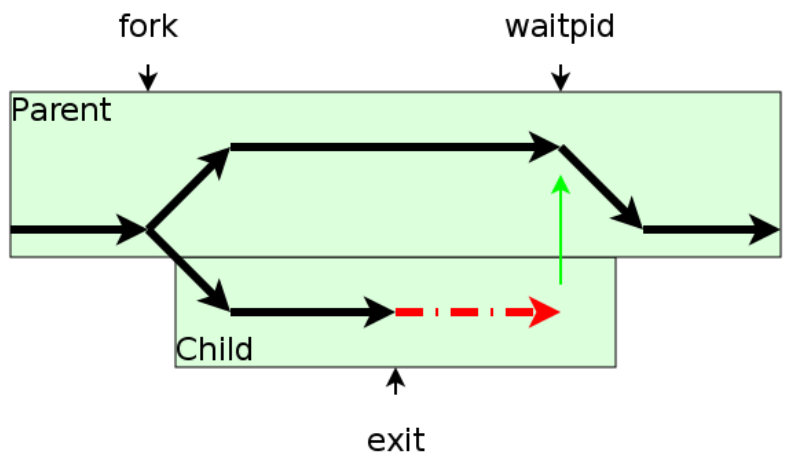
- Prozess-ID (PID)
- Prozesszustand (Laufend, Bereit, ...)
- Register
- Speicherabbildung
- Eigentümer (UID, GID)
- Wurzelverzeichnis, aktuelles Verzeichnis
- offene Dateien
- ...

Beispiel: UNIX

- Überblick:

fork: Erzeugung von neuen Prozessen
exit: Terminieren von Prozessen
waitpid: Warten auf das Terminieren von Prozessen





Programmierschnittstelle

- Duplizieren des gerade laufenden Prozesses

```
#include <unistd.h>

pid_t fork(void);
```

- Terminieren des aktuellen Prozesses

```
#include <stdlib.h>

void exit(int status);
```

- Warten auf das Terminieren eines anderen Prozesses

```
#include <sys/types.h>
#include <sys/wait.h>

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


Beispiel

```c
pid_t pid, ret;
int status;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(1);

case 0: /* Child */
    do_child_work();
    exit(13);

default: /* Parent */
    do_parent_work();
    ret = waitpid(pid, &status, 0);
    /*
     * In case of no error:
     * ret == pid
     * WIFEXITED(status) == 1
     * WEXITSTATUS(status) == 13
     */
    break;
}
```

Prozesserzeugung

- Der Kind-Prozess ist Kopie des Vater-Prozesses
  - gleiches Programm
  - gleiche Daten (Variablen-Inhalte)
  - gleicher Programmzähler
  - gleiches aktuelles Verzeichnis, Wurzelverzeichnis
  - gleiche geöffnete Dateien
- einzige Unterschiede
  - verschiedene Prozess-IDs
  - Rückgabewert von `fork`

Ausführung von Programmen

- Das von einem Prozess ausgeführte Programm kann durch ein neues Programm ersetzt werden:

```
#include <unistd.h>

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

Beispiel:

```
...                 /* Process A */
argv[0] = "ls";
argv[1] = "-l";
argv[2] = NULL;
execv("/bin/ls", argv);
/* Should not be reached. */
```

$\Rightarrow$

```
...                 /* Process A */
int
main(int argc, char *argv[])
{
    ...
}
```

Das zuvor laufende Programm wird beendet, das neue gestartet.

Es wird nur das Programm ausgetauscht.

Der Prozess läuft weiter!

Start eines Programms

- Beispiel: Start des Programms `./prog` mit Parametern `-a` und `-b`

… als Vordergrund-Prozess:

```
pid_t pid;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(EXIT_FAILURE);

case 0: /* Child */
    execl("./prog", "prog",
          "-a", "-b", NULL);
    perror("./prog");
    exit(EXIT_FAILURE);

default: /* Parent */
    waitpid(pid, NULL, 0);
    break;
}
```

… als Hintergrund-Prozess:

```
pid_t pid;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(EXIT_FAILURE);

case 0: /* Child */
    execl("./prog", "prog",
          "-a", "-b", NULL);
    perror("./prog");
    exit(EXIT_FAILURE);

default: /* Parent */
    /* No "waitpid" here! */
    break;
}
```

Signale

- Mikrocontroller kann auf nebenläufige Ereignisse (Interrupts) mit Interrupt-Service-Routinen reagieren.
- Ähnliches Konzept auf Prozess-Ebene: Signale

Signale (2)

Interrupt: asynchrones Signal aufgrund eines „externen” Ereignisses

- CTRL-C auf der Tastatur gedrückt
- Timer abgelaufen
- Kind-Prozess terminiert
- ...

Exception: synchrones Signal, ausgelöst durch die Aktivität des Prozesses

- Zugriff auf ungültige Speicheradresse
- Illegaler Maschinenbefehl
- Division durch 0
- Schreiben auf eine geschlossene Kommunikationsverbindung
- ...

Kommunikation: ein Prozess will einem anderen ein Ereignis signalisieren

Signale (3)

CTRL-C:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
^C
~>
```

Illegaler Speicherzugriff:

```
int main(void)
{
    *(int *) 0 = 0;
    return 0;
}
```

```
~> ./test
Segmentation fault
~>
```

Inter-Prozess-Kommunikation:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
Terminated
~>
```

```
~> killall test
~>
```

Reaktion auf Signale

abort:

erzeugt Core-Dump (Speicher- und Registerinhalte werden in Datei `./core` geschrieben) und beendet Prozess
Standardeinstellung für alle Exceptions (zum nachträglichen Debuggen)

exit:

beendet Prozess (ohne Core-Dump)
Standardeinstellung für z.B. CTRL-C, Kill-Signal

ignore:

Signal wird ignoriert
Standardeinstellung für alle „unwichtigen" Signale (z.B. Kindprozess terminiert, Größe des Terminal-Fensters hat sich geändert)

...

Reaktion auf Signale (2)

...

handler:
: Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses
: nie Standardeinstellung, da Programm-abhängig

stop:
: stoppt Prozess
: Standardeinstellung für Stop-Signal

continue:
: setzt Prozess fort
: Standardeinstellung für Continue-Signal

Reaktion über System-Aufruf (`sigaction`) änderbar

Programmierschnittstelle

- Einstellen der Signalbehandlungsfunktion
  (entspricht dem Setzen der ISR-Funktion)

```
#include <signal.h>

int sigaction(int sig, struct sigaction *new, struct sigaction *old);
```

`struct sigaction` enthält:

```
void (*sa_handler)(int sig); /* handler function
                                or SIG_DFL or SIG_IGN */
sigset_t sa_mask;            /* list of blocked signals while
                                handler is executed */
int sa_flags;                /* ... */
```

- ...

Programmierschnittstelle

- ...
- Blockieren/Freigeben von Signalen
  (entsprich `cli()`, `sei()`)

```
#include <signal.h>

int sigprocmask(int how, sigset_t *nmask, sigset_t *omask);
```

  - `SIG_BLOCK`: angegebene Signale blockieren
  - `SIG_UNBLOCK`: angegebene Signale deblockieren
  - `SIG_SETMASK`: Signalmaske setzen

- Freigeben + Passives Warten auf Signal + wieder Blockieren
  (entspricht `sei(); sleep_cpu(); cli();`)

```
#include <signal.h>

int sigsuspend(sigset_t *mask);
```

- ...

Programmierschnittstelle

- ...
- Erstellen einer leeren Signal-Liste

```
#include <signal.h>

int sigemptyset(sigset_t *mask);
```

- Erstellen einer vollen Signal-Liste

```
int sigfillset(sigset_t *mask);
```

- Hinzufügen eines Signals zu einer Signal-Liste

```
int sigaddset(sigset_t *mask, int sig);
```

- Entfernen eines Signals aus einer Signal-Liste

```
int sigdelset(sigset_t *mask, int sig);
```


Programmierschnittstelle

- Typische Signale:

SIGSEGV: „Segmentation Fault” (ungültiger Speicherzugriff)
SIGINT: „Interrupt” (CTRL-C)
SIGALRM: „Alarm” (Timer abgelaufen)
SIGCHLD: „Child” (Kindprozess terminiert)
SIGTERM: „Terminate” (Abbruch des Prozesses; abfangbar)
SIGKILL: „Kill” (Abbruch des Prozesses; nicht abfangbar)

Signal-Beispiel 1

```c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // CTRL-C signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGINT, &sa, NULL);

  for (int i = 0; ; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

```c
static void handler(int sig)
{
    char s[] = "CTRL-C!\n";
    write(STDOUT_FILENO,
          s, strlen(s));
}
```

(Fehlerbehandlung weggelassen...)

```
~> ./test
...
146431
146432
146433
14^CCTRL-C!
6434
146435
146436
...
~>
```


Signal-Beispiel 2

```c
int main(void)
{
  // Call handler when
  // timer signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGALRM, &sa, NULL);

  // Send timer signal every sec.
  struct itimerval it;
  it.it_value.tv_sec  = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec  = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);

  // Wait for timer ticks.
  sigset_t mask;
  sigemptyset(&mask);
  while (1) sigsuspend(&mask);
}
```

```c
static void handler(int sig)
{
  write(STDOUT_FILENO,
        "Tick\n", 5);
}
```

(Fehlerbehandlung weggelassen...)

```
~> ./test
Tick
Tick
Tick
^C
~>
```

Signal-Beispiel 3

```c
#include <signal.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // I/O signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGIO, &sa, NULL);

  // Send I/O signal when
  // STDIN can be read.
  int flags = fcntl(STDIN_FILENO,
      F_GETFL);
  flags |= O_ASYNC;
  fcntl(STDIN_FILENO, F_SETFL,
      flags);

  while (1) sleep(1);
}
```

```c
static void handler(int sig)
{
  char buf[256];
  int len;

  // Read chars from STDIN.
  len = read(STDIN_FILENO, buf,
          sizeof(buf));

  // Handle chars in buf.
  ...
}
```

(Fehlerbehandlung weggelassen...)

Signale und Nebenläufigkeit

- Signale erzeugen Nebenläufigkeit innerhalb von Prozessen
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller

Nebenläufigkeitsbeispiel

```c
int main(void) {
  struct sigaction sa;
  struct itimerval it;
  /* Setup timer tick handler. */
  sa.sa_handler = tick;
  sa.sa_flags = 0;
  sigfillset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);
  /* Setup timer. */
  it.it_value.tv_sec = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);
  /* Print time while working. */
  while (1) {
    int s = sec, m = min, h = hour;
    printf("%02d:%02d:%02d\n", h, m, s);
    do_work();
  }
}
```

↓ hier Signal

(Fehlerbehandlung weggelassen...)

```c
volatile int hour = 0;
volatile int min = 0;
volatile int sec = 0;

static void tick(int sig) {
  sec++;
  if (60 <= sec) {
    sec = 0; min++;
  }
  if (60 <= min) {
    min = 0; hour++;
  }
  if (24 <= hour) {
    hour = 0;
  }
}
```

```
~> ./test
...
23:59:59
00:59:59
00:00:00
...
```

← hier Problem!

Lösungen Nebenläufigkeitsbeispiel

1. Lösung

```
sigset_t nmask, omask;

/* Block SIGALRM. */
sigemptyset(&nmask);
sigaddset(&nmask, SIGALRM);
sigprocmask(SIG_BLOCK,
            &nmask, &omask);

/* Get current time. */
int s = sec, m = min, h = hour;

/* Restore signal mask. */
sigprocmask(SIG_SETMASK,
            &omask, NULL);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

2. Lösung

```
/* Get current time. */
int s, m, h;
do {
  s = sec;
  m = min;
  h = hour;
} while (s != sec
      || m != min
      || h != hour);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

Weitere Lösungen existieren...

Nebenläufigkeitsprobleme

- Zusätzliches Problem:
  interne Funktionsweise von Bibliotheksfunktionen i.A. unbekannt
- Beispiel 1:
  `printf` fügt Zeichen in Puffer ein
  => Nutzung von `printf` im Hauptprogramm *und* in Signal-Behandlungsfunktion u.U. gefährlich
- Beispiel 2:
  `malloc` durchsucht Liste nach freiem Speicherbereich; `free` fügt Block in Liste ein
  => Nutzung von `malloc`/`free` im Hauptprogramm *und* in Signal-Behandlungsfunktion u.U. gefährlich
- Lösung:
  - Signale während der Ausführung kritischer Bereiche blockieren oder
  - keine unbekannten Bibliotheksfunktionen aus Signal-Behandlungsfunktionen heraus aufrufen