

# U8 8. Übung

## U8-1 Überblick

- Besprechung Aufgabe 5 (printdir)
- Online-Evaluation
- Signale

## U8-2 Evaluation

- Online-Evaluation von Vorlesung und Übung SOS
- zwei TANS, zwei Fragebogen
  - TANS werden in den Tafelübungen verteilt
- Fragebogen vom 13. Juni bis 1. Juli auszufüllen
- Ergebnisse werden ab 4. Juli an Dozenten verschickt
  - Diskussion in Vorlesung und Übungen
  - Veröffentlichung auf den Web-Seiten der Lehrveranstaltung
- Ergebnisse der Evaluation vom letzten Jahr stehen im Netz
- bitte unbedingt teilnehmen - das Feedback ist für die Dozenten und Organisatoren sehr wichtig (wir wollen unseren Job so gut wie möglich machen!)
  - uns interessiert natürlich vor allem, was wir seit dem letzten Sommer besser (oder schlechter) gemacht haben!

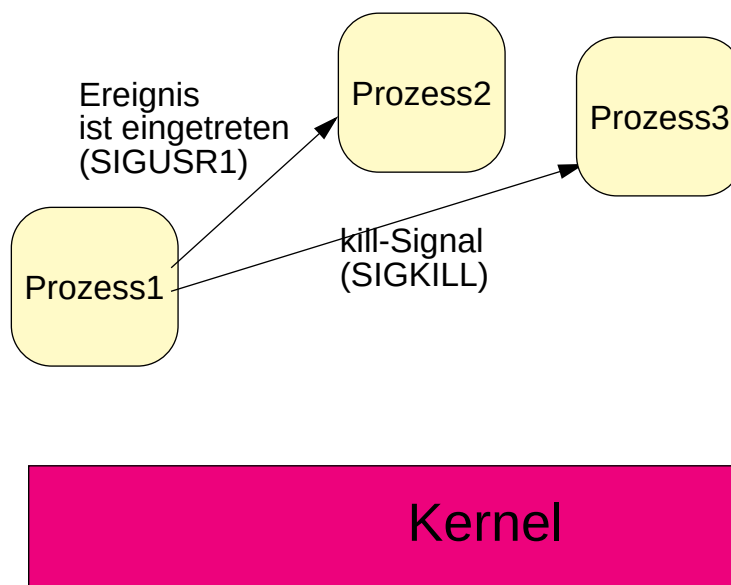
## U8-3 Aufgabe 7: jsh

### Ziele der Aufgabe

- Signale unter UNIX bilden die Konzepte "Trap" und "Interrupt" für eine Interaktion zwischen Betriebssystem und Anwendung nach
  - praktischer Umgang mit diesen Konzepten
- Signalbehandlung führt zu asynchronen Funktionsaufrufen
  - Nebenläufigkeit
  - kritische Abschnitte wo es zu Race-Conditions kommen kann, müssen beim Softwareentwurf erkannt werden
  - Koordinierungsmaßnahmen / unteilbare Abschnitte sind erforderlich
  - Aufgabe macht diese Probleme praktisch deutlich, Umgang mit ersten Koordinierungsmechanismen

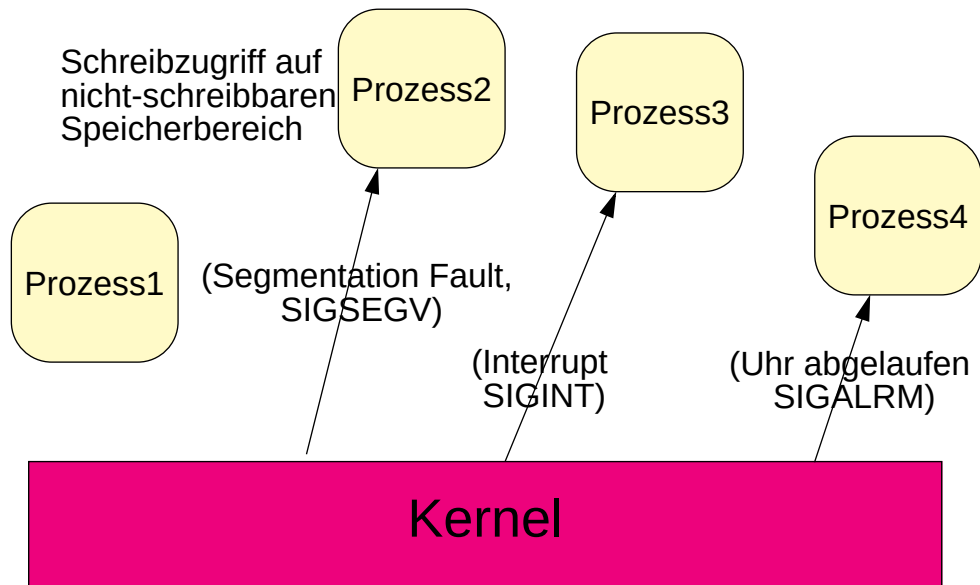
## U8-4 Signale

### 1 Kommunikation zwischen Prozessen



## 2 Signalisierung des Systemkerns

- synchrone Signale: werden durch Aktivität des Prozesses ausgelöst
- asynchrone Signale: werden "von außen" ausgelöst

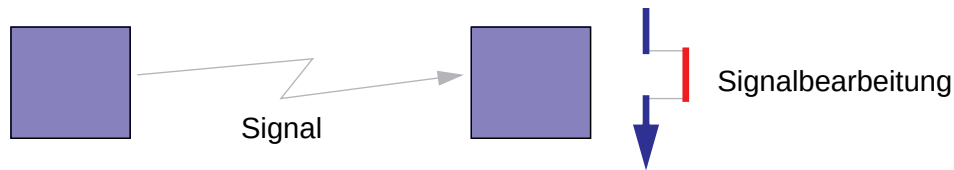


## 3 Reaktion auf Signale

- abort
  - ◆ erzeugt Core-Dump (Segmente + Registercontext) und beendet Prozess
- exit
  - ◆ beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
  - ◆ ignoriert Signal
- stop
  - ◆ stoppt Prozess
- continue
  - ◆ setzt gestoppten Prozess fort
- signal handler
  - ◆ Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

## 4 Posix Signalbehandlung

- Signal bewirkt Aufruf einer Funktion



- ◆ nach der Behandlung läuft Prozess an unterbrochener Stelle weiter

- Systemschnittstelle

- ◆ sigaction
- ◆ sigprocmask
- ◆ sigsuspend
- ◆ sigpending
- ◆ kill

## 5 Signalhandler installieren: sigaction

- Prototyp

```
#include <signal.h>

int sigaction(int sig, /* Signal */
              const struct sigaction *act, /* Handler */
              struct sigaction *oact /* Alter Handler */ );
```

- Handler bleibt solange installiert, bis neuer Handler mit **sigaction** installiert wird

- sigaction Struktur

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
};
```

## 5 Signalhandler installieren: sigaction Handler (sa\_handler)

- Signalbehandlung kann über **sa\_handler** eingestellt werden:
  - **SIG\_IGN**                      Signal ignorieren
  - **SIG\_DFL**                      Default Signalbehandlung einstellen
  - *Funktionsadresse*              Funktion wird in der Signalbehandlung aufgerufen und ausgeführt

## 5 Signalhandler installieren: sigaction Maske (sa\_mask)

- verzögerte Signale
  - ◆ während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
  - ◆ bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - ◆ es wird maximal ein Signal zwischengespeichert
- mit **sa\_mask** in der **struct sigaction** kann man zusätzliche Signale blockieren
- Auslesen und Modifikation der Signal-Maske vom Typ **sigset\_t** mit:
  - ◆ **sigaddset()**: Signal zur Maske hinzufügen
  - ◆ **sigdelset()**: Signal aus Maske entfernen
  - ◆ **sigemptyset()**: Alle Signale aus Maske entfernen
  - ◆ **sigfillset()**: Alle Signale in Maske aufnehmen
  - ◆ **sigismember()**: Abfrage, ob Signal in Maske enthalten ist

## 5 Signalhandler installieren: sigaction Flags (sa\_flags)

- Durch **sa\_flags** lässt sich das Verhalten beim Signalempfang beeinflussen.
  - kann für jedes Signal gesondert gesetzt werden.
- **SA\_NOCLDSTOP**: SIGCHLD wird nur erzeugt, wenn Kind terminiert, nicht wenn es gestoppt wird (POSIX, SystemV, BSD)
- **SA\_RESTART**: durch das Signal unterbrochene Systemaufrufe werden automatisch neu aufgesetzt (kein `errno=EINTR`) (nur SystemV und BSD) (siehe Seite 17)
- **SA\_SIGINFO**: Signalhandler bekommt zusätzliche Informationen übergeben (nur SystemV)
 

```
void func(int signo, siginfo_t *info, void *context);
```
- **SA\_NODEFER**: Signal wird während der Signalbehandlung nicht blockiert (nur SystemV)

## 5 Signalhandler installieren: Beispiel

- Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL); /* return abfragen ! */
```

## 6 Signal zustellen

- Systemaufruf

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

- Kommando **kill** aus der Shell (z. B. **kill -USR1 <pid>**)

## 7 POSIX Signale

Das Defaultverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps.

- SIGABRT (core): Abort Signal; entsteht z.B. durch Aufruf von **abort()**
- SIGALRM: Timer abgelaufen (**alarm()**, **setitimer()**)
- SIGFPE (core): Floating Point Exception; z.B. Division durch 0 oder Overflow
- SIGHUP: Terminalverbindung wird beendet (Hangup)
- SIGILL (core): Illegal Instruction; z.B. privilegierte Operation, privilegiertes Register
- SIGINT: Interrupt; (Shell: CTRL-C)
- SIGKILL (nicht abfangbar): beendet den Prozess

## 7 POSIX Signale (2)

- SIGPIPE: Schreiben auf Pipe oder Socket nachdem der lesende terminiert ist
- SIGQUIT (core): Quit; (Shell: CTRL-\)
- SIGSEGV (core): Segmentation violation; inkorrektter Zugriff auf Segment, z.B. Schreiben auf Textsegment
- SIGTERM: Termination; Default-Signal für **kill(1)**
- SIGUSR1, SIGUSR2: Benutzerdefinierte Signale

## 8 Jobcontrol-Signale

Diese Signale existieren in einem POSIX-konformen System nur, wenn das System Jobkontrolle unterstützt (`_POSIX_JOB_CONTROL` ist definiert).

- **SIGCHLD** (Default-Aktion = ignorieren): Status eines Kindprozesses hat sich geändert
- **SIGCONT**: setzt den gestoppten Prozess fort
- **SIGSTOP** (nicht abfangbar): stoppt den Prozess
- **SIGTSTP**: stoppt den Prozess (Shell: CTRL-Z)
- **SIGTTIN**, **SIGTTOU**: Hintergrundprozess wollte vom Terminal lesen bzw. darauf schreiben

## 9 Jobcontrol und wait

- **wait(int \*stat)** kehrt auch zurück, wenn Kind gestoppt wird
- erkennbar an Wert von **\*stat**
- Auswertung mit Macros
  - ◆ **WIFEXITED(\*stat)**: Kind normal terminiert
  - ◆ **WIFSIGNALED(\*stat)**: Kind durch Signal terminiert
  - ◆ **WIFSTOPPED(\*stat)**: Kind gestoppt
  - ◆ **WIFCONTINUED(\*stat)**: gestopptes Kind fortgesetzt

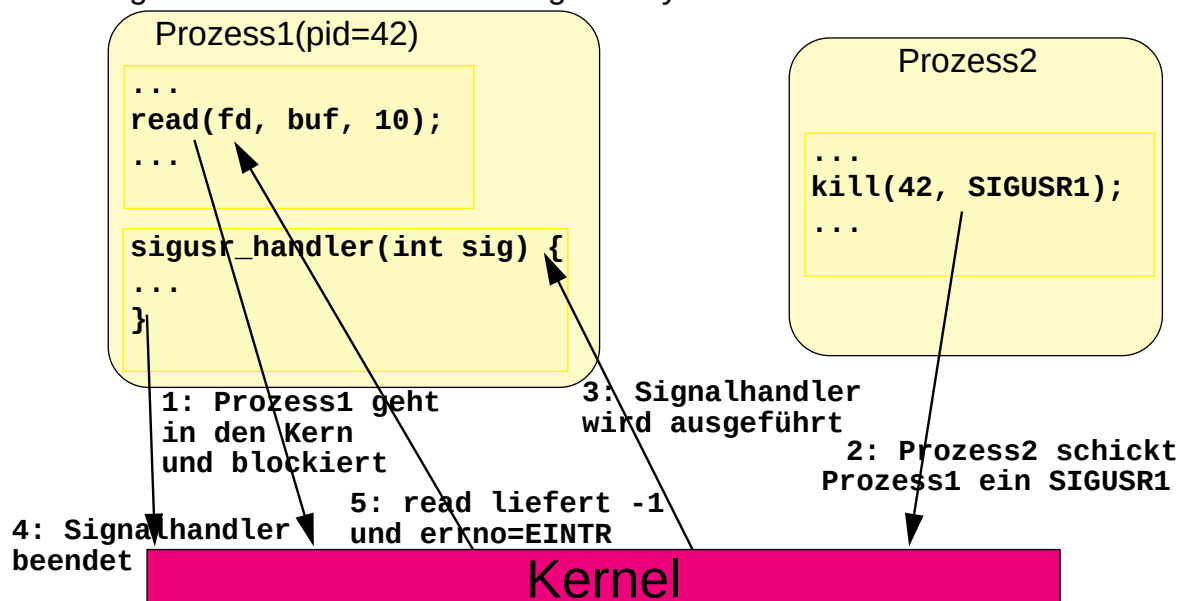


## 10 signal()-Funktion

- ANSI-C definiert die signal()-Funktion zum Installieren von Signalhandlern
  - ◆ Problem: sehr ungenaue Spezifikation, da Prozesskonzept in ANSI-C nicht definiert
- BSD- und SystemV-Unix Systeme enthalten die signal-Funktion
  - ◆ Problem: Prozesskonzept jetzt definiert, aber signal-Semantik ist von Unix Version 7 abgeleitet und unzuverlässig (*unreliable signals*) (Signalhandler bleibt nicht installiert, Signale können nicht blockiert werden)
- **signal() ist deshalb in POSIX.1 nicht enthalten und sollte auch nicht mehr benutzt werden**

## 11 Unterbrechen von Systemcalls

- Signale können die Ausführung von Systemaufrufen unterbrechen



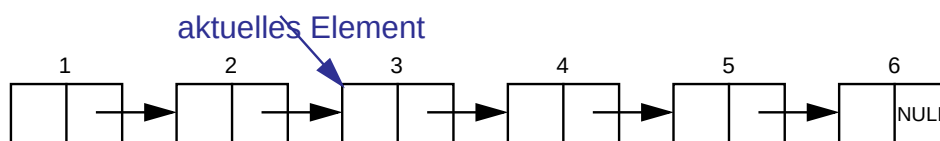
## 11 Unterbrechen von Systemcalls (2)

- dies betrifft nur "langsame Systemcalls" (welche sich über einen längeren Zeitraum blockieren können, z.B. `wait()`, `waitpid()` oder `read()` von einem Socket oder einer Pipe)
- der Systemcall setzt dann `errno` auf `EINTR`
- in einigen UNIXen (z.B. 4.2BSD) werden unterbrochene Systemcalls automatisch neu aufgesetzt
- bei einigen UNIXen (SystemV R4, 4.3BSD), kann man für jedes Signal einstellen (`SA_RESTART`), ob ein Systemcall automatisch neu aufgesetzt werden soll
- POSIX.1 lässt dies unspezifiziert
- die Systemaufrufe `pause()` und `sigsuspend()` werden in keinem Fall fortgesetzt

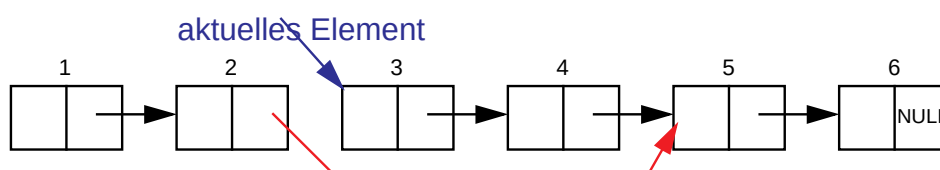
## 12 Signale und Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses (vgl. Nebenläufigkeit durch Interrupts, Vorlesung Seite III-51 und V-78 ff)
- diese Nebenläufigkeit kann zu Race-Conditions führen
- Beispiel:

◆ main-Funktion läuft durch eine verkettete Liste



◆ Prozess erhält Signal; Signalhandler entfernt Elemente 3 und 4 aus der Liste und gibt den Speicher dieser Elemente frei



## 12 Signale und Race Conditions (2)

- Lösung: Signal während Ausführung des kritischen Abschnitts blockieren!
- weiteres Problem:
  - ◆ Aufruf von Bibliotheksfunktionen, z.B. **getpwuid()**, wird durch Signal unterbrochen und nach Ausführung des Signalhandlers fortgesetzt
  - ◆ Signalhandler ruft auch **getpwuid()** auf -> Race Condition!
- Lösung:
  - ◆ in Signalhandlern nur Funktionen aufrufen, die in POSIX.1 als reentrant gekennzeichnet sind (**getpwuid** und **malloc/free** sind z.B. nicht reentrant, **wait** und **waitpid** sind reentrant)
    - Achtung: wenn in einem Signalhandler Funktionen verwendet werden, die **errno** verändern, muss der Wert von **errno** vorher gesichert und vor Beendigung des Signalhandlers wieder zurückgesetzt werden
  - ◆ oder Signal während Ausführung der Funktion blockieren

## 13 Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how, /* Verknüpfung der Masken */
               const sigset_t *set, /* neue Maske */
               sigset_t *oset /* Speicher für alte Maske */ );
```

- how:
  - ◆ **SIG\_BLOCK**: Vereinigungsmenge zwischen übergebener und alter Maske
  - ◆ **SIG\_SETMASK**: Setzen der Maske ohne Beachtung der alten Maske
  - ◆ **SIG\_UNBLOCK**: Schnittmenge zwischen inverser übergebener Maske und alter Maske

- Beispiel

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL);
```

- Anwendung: kritische Abschnitte, die nicht durch ein Signal unterbrochen werden dürfen

## 14 Warten auf Signale

- Problem: Prozess befindet sich in kritischem Abschnitt und will auf ein Signal warten
  - Signal muss deblockiert werden
  - Prozess wartet auf Signal
  - Signal muss wieder blockiert werden
- Operationen müssen atomar am Stück ausgeführt werden!
- Prototyp

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

- ◆ **sigsuspend(mask)** merkt sich die aktuelle Signal-Maske, setzt **mask** als neue Signal-Maske und blockiert Prozess
- ◆ Signal führt zu Aufruf des Signalhandlers (muss vorher installiert werden)
- ◆ **sigsuspend** kehrt nach Bearbeitung des Signalhandlers mit Fehler **EINTR** zurück und restauriert gleichzeitig die ursprüngliche Signal-Maske

## 15 Abfrage blockierter Signale

- Prototyp

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- **sigpending** speichert alle Signale, die blockiert sind, aber empfangen wurden, in **set** ab