35 Überblick über die 7. Übung

- Besprechung 3. Aufgabe (printdir)
- Rechenzeiterfassung von Prozessen
- POSIX-Signale

!!! Änderung von Aufgabe 5f: SIGQUIT anstelle von SIGKILL senden !!!

36 Rechenzeiterfassung (2)

- Prototyp

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

- tms Datenstruktur

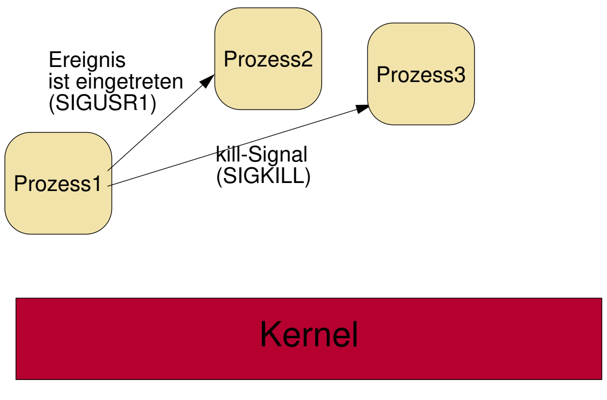
```
struct  tms  {
        clock_t tms_utime;  /* user time */
        clock_t tms_stime;  /* system time */
        clock_t tms_cutime; /* user time of children */
        clock_t tms_cstime; /* system time of children */
}
```


36 Rechenzeiterfassung

- Betriebssystem erfasst die Rechenzeit der Prozesse
  - user time: Rechenzeit im Benutzermodus
  - system time: Rechenzeit im Systemkern (priv. Modus)
- für jeden Prozess wird außerdem die Rechenzeit aller Kind-Prozesse aufsummiert
  - nach Terminieren eines Kind-Prozesses bleiben die Daten in Datenstruktur des ZOMBIE-Prozesses gespeichert
  - bei **`wait()/waitpid()`** werden die Daten in den Vaterprozess übernommen
- Rechenzeiten werden in clock ticks angegeben (meist 10 ms)
  - clock ticks/Sekunde kann durch das Makro **`CLK_TCK`** abgefragt werden

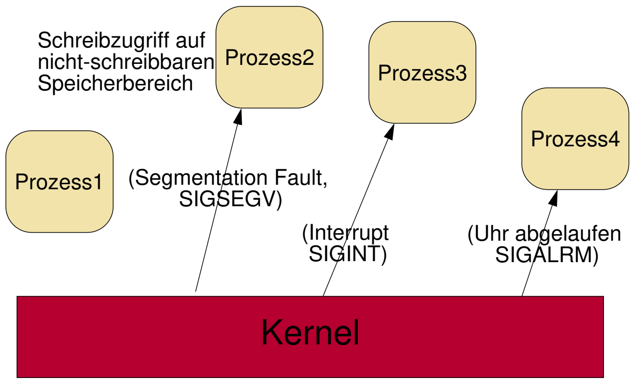
37 Signale

37.1 Kommunikation zwischen Prozessen



37.2 Signalisierung des Systemkerns

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



37.3 Reaktion auf Signale: Default-Aktionen

- abort
  - erzeugt einen 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

37.4 Posix Signalbehandlung

- sigaction
- sigprocmask
- sigsuspend
- sigpending

37.5 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;
}
```

37.5.1 sigaction Handler (sa_handler)

- ist ein Funktionspointer oder einer der vordefinierten Werte
  - **`SIG_DFL`**: Default Signalbehandlung
  - **`SIG_IGN`**: Signal ignorieren

37.5.2 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

37.5.3 sigaction Flags (sa_flags)

- Durch 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, SVID, BSD)
- **`SA_RESTART`**: durch das Signal unterbrochene Systemaufrufe werden automatisch neu aufgesetzt (kein errno=EINTR) (nur SVID und BSD)
- **`SA_SIGINFO`**: Signalhandler bekommt zusätzliche Informationen übergeben (nur SVID)
  `void func(int signo, siginfo_t *info, void *context);`
- **`SA_NODEFER`**: Signal wird während der Signalbehandlung nicht blockiert (nur SVID)

37.5.4 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 ! */
```

- Signal schicken mit **`kill -USR1 <pid>`** oder mit

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

37.6 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);
```


37.7 Warten auf Signale

- Prototyp

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

- **sigsuspend(mask)** wartet bis ein nicht blockiertes Signal eintrifft, welches durch einen Signalhandler abgefangen wird
- **mask** wird als aktuelle Signal-Maske installiert
- kehrt nach Bearbeitung des Signalhandlers zurück

37.8 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

37.9 POSIX Signale

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

- 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

37.9 POSIX Signale

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

37.10 Jobcontrol-Signale

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

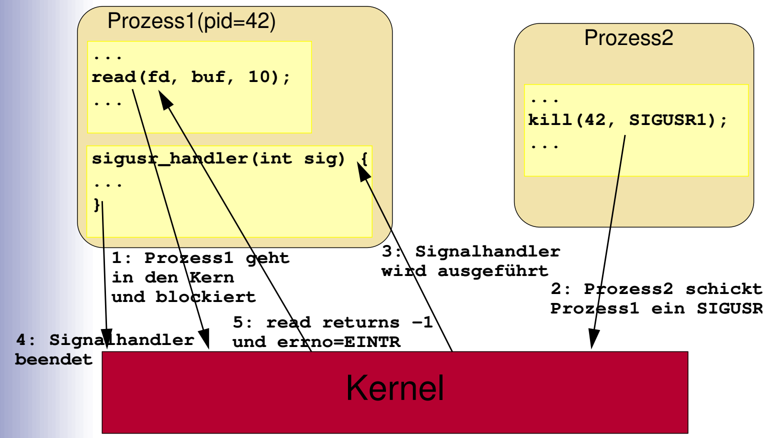
- SIGCHLD (Defaultaktion ist Ignorieren): Status eines Kindprozesses hat sich geändert
- SIGCONT: setzt den gestoppten Prozess fort
- SIGSTOP (nicht abfangbar): stoppt den Prozess
- SIGTSTP: stoppt den Prozess (Shell: CTRL-Z)
- SIGTTIN, SIGTTOU: Hintergrundprozess wollte vom Terminal lesen bzw. darauf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

37.11 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

37.12 Unterbrechen von Systemcalls

- Signale können die Ausführung von Systemaufrufen unterbrechen

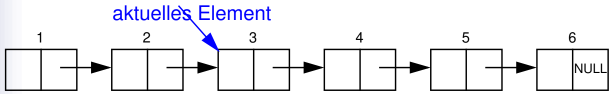


37.12 Unterbrechen von Systemcalls

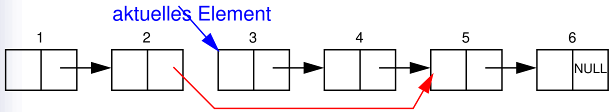
- dies betrifft nur "langsame Systemcalls" (welche sich über einen längeren Zeitraum blockieren können, z.B. **`wait()`**, **`waitpid()`** oder **`read()`** von einem Socket oder einer Pipe)
- der Systemcall setzt dann **`errno`** auf **`EINTR`**
- in einigen UNIXen (z.B. 4.2BSD) werden unterbrochene Systemcalls automatisch neu aufgesetzt
- bei einigen UNIXen (SVR4, 4.3BSD), kann man für jedes Signal einstellen (**`SA_RESTART`**), ob ein Systemcall automatisch neu aufgesetzt werden soll
- POSIX.1 läßt dies unspezifiziert

37.13 Signale und Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
- 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



37.13 Signale und Race Conditions

- 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

37.14 signal()-Funktion

- ANSI-C definiert die signal()-Funktion zum Installieren von Signalhandlern
  - Problem: Die Semantik ist vom Betriebssystem abhängig.
    z.B. bei Unix Version 7 (Signalhandler bleibt nicht installiert,Signale können nicht blockiert werden)
- **signal() ist deshalb in POSIX.1 nicht enthalten und sollte auch nicht mehr benutzt werden**