U3 UNIX-Signale

- Aufgabe 8: josh
- UNIX-Signale: Funktionsweise und Behandlung
- UNIX-API zur Signalbehandlung
- Unterbrechung blockierender Systemaufrufe
- Nebenläufigkeit durch Signalbehandlung
  - Probleme, Beispiel: Warten auf Signale
  - Synchronisation durch Blockieren von Signalen
  - das `volatile`-Schlüsselwort
- Filedeskriptoren duplizieren

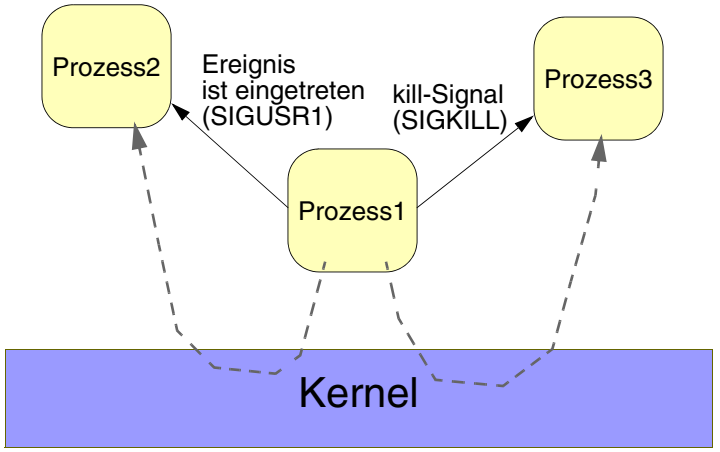
U3-1 Aufgabe 8: josh

1 Ziele der Aufgabe

- Signale unter UNIX bilden die Konzepte *Trap* und *Interrupt* für die Interaktion zwischen Betriebssystem und Prozessen nach
  - ➤ praktischer Umgang mit diesen Konzepten
- Signalbehandlung führt zu asynchronen Funktionsaufrufen
  - ➤ Nebenläufigkeit
  - ➤ kritische Abschnitte, in denen es zu Race-Conditions kommen kann, müssen beim Softwareentwurf erkannt werden
  - ➤ Koordinierungsmaßnahmen sind erforderlich
  - ➤ Aufgabe macht diese Probleme praktisch deutlich, Umgang mit weiteren Koordinierungsmechanismen

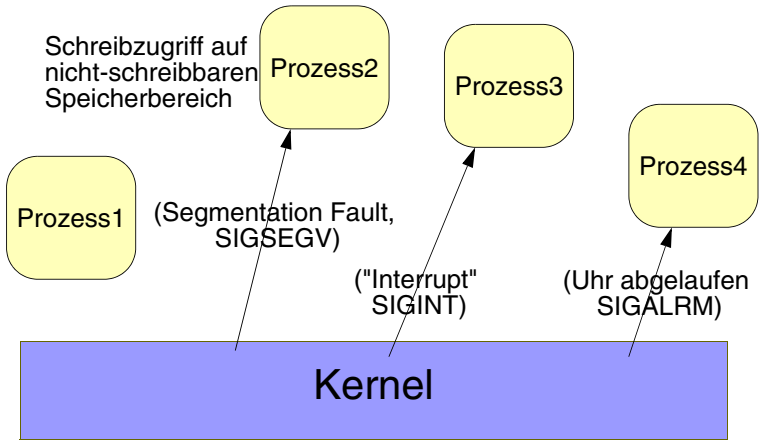
U3-2 Signale

1 Kommunikation zwischen Prozessen



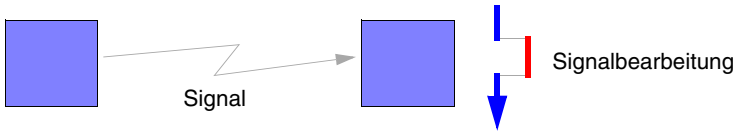
2 Signalisierung des Systemkerns

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



3 Reaktion auf Signale

- abort
  - ◆ erzeugt Core-Dump (Segmente + Registerkontext) und beendet Prozess
- exit (Standardreaktion für die meisten Signale)
  - ◆ beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
  - ◆ ignoriert Signal
- Signal-Behandlungsfunktion
  - ◆ Aufruf der Signalbehandlungsfunktion, danach Fortsetzung des Prozesses



4 UNIX-Signale

- SIGABRT (abort): Abort-Signal; entsteht z.B. durch Aufruf von `abort()`
- SIGALRM: Timer abgelaufen (`alarm()`, `setitimer()`)
- SIGFPE (abort): Floating Point Exception (Division durch 0, Überlauf, ...)
- SIGILL (abort): Illegal Instruction; z.B. privilegierte Operation
- SIGINT: Interrupt; (Shell: CTRL-C)
- SIGKILL: beendet den Prozess (nicht abfangbar)
- SIGPIPE: Schreiben auf Pipe oder Socket nachdem die Gegenseite geschlossen wurde
- SIGSEGV (abort): Segmentation Violation; illegaler Speicherzugriff
- SIGTERM: Termination; Default-Signal für `kill(1)`
- SIGUSR1, SIGUSR2: Benutzerdefinierte Signale

5 Jobcontrol-Signale

- SIGCHLD (ignore): Status eines Kindprozesses hat sich geändert
- SIGCONT (continue): setzt den gestoppten Prozess fort
- SIGSTOP (stop, nicht abfangbar): stoppt den Prozess
- SIGTSTP (stop): stoppt den Prozess (Shell: CTRL-Z)
- SIGTTIN: Hintergrundprozess wollte vom Terminal lesen
- SIGTTOU: Hintergrundprozess wollte auf Terminal schreiben

6 Problem: asynchrone Signale und abort/exit

- Beispiel: CTRL-C von der Tastatur
  - ➤ Beendigungsmodell (vgl. Vorlesung *B | V Betriebssystemebene, Seite 17*)
- Widerspruch: ein Interrupt darf niemals nach dem Beendigungsmodell behandelt werden
  - ➤ Grund: der Prozess könnte gerade einen Systemaufruf ausführen (Ebene-2-Code) und dabei komplexe Datenstrukturen des Systemkerns manipulieren (z. B. verkettete Liste)
- Lösung: Prozess wird nicht beendet, sondern über das Signal informiert
  - ➤ Eintragen der Information in Prozessverwaltungsstruktur (Prozesskontrollblock)
  - ➤ vor der nächsten Rückkehr aus dem Betriebssystemkern (Ebene 2, vgl. Vorlesung *B | V Betriebssystemebene, Seite 12*) oder vor einem Übergang in den Zustand "blockiert" erkennt der Prozess das Signal und terminiert selbst

U3-3 UNIX-API zur Signalbehandlung

- Systemschnittstelle
  - sigaction
  - sigprocmask
  - sigsuspend
  - sigpending
  - kill

1 Signalbehandlung installieren

- Prototyp

```
#include <signal.h>

int sigaction(int sig,                  // Signalnummer
       const struct sigaction *act,     // neue Behandlung
       struct sigaction *oact           // vorherige Behandlung
);
```

- Behandlung bleibt solange aktiv, bis eine neue mit `sigaction` installiert wird
- `sigaction`-Struktur

```
struct sigaction {
   void (*sa_handler)(int); // Behandlungsfunktion
   sigset_t sa_mask;        // blockierte Signale
   int sa_flags;            // Optionen
};
```

1 Signalbehandlung installieren: `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

1 Signalbehandlung installieren: `sa_mask`

- blockierte Signale
  - während einer Signalbehandlung wird das auslösende Signal blockiert
  - bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - es wird maximal ein Signal zwischengespeichert
  - Blockieren von Signalen nicht gleich Ignorieren von Signalen:
    ignorierte Signale werden *verworfen*, blockierte nur *verzögert*
- mit `sa_mask` kann man *weitere* Signale blockieren
- Auslesen und Modifikation einer 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

1 Signalbehandlung installieren: `sa_flags`

- Beeinflussung des Verhaltens bei Signalempfang durch `sa_flags`
  - ➤ kann für jedes Signal gesondert gesetzt werden
- `SA_NOCLDSTOP`
  - ◆ SIGCHLD wird nur zugestellt, wenn ein Kindprozess terminiert, nicht wenn er gestoppt wird
- `SA_RESTART`
  - ◆ durch das Signal unterbrochene Systemaufrufe werden automatisch neu aufgesetzt (statt `errno=EINTR` Fehler, s. Folie 17)
- weitere Flags siehe **sigaction(2)**

1 Signalhandler installieren: Beispiel

```
#include <signal.h>
void ghostbuster(int sig) { ... }
int main() {
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_NOCLDSTOP | SA_RESTART;
    action.sa_handler = ghostbuster;
    sigaction(SIGCHLD, &action, NULL);
    ...
```

2 Signal zustellen

- Systemaufruf `kill(2)`

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

- Kommando `kill(1)` aus der Shell (z. B. `kill -USR1 <pid>`)

3 Jobcontrol und wait

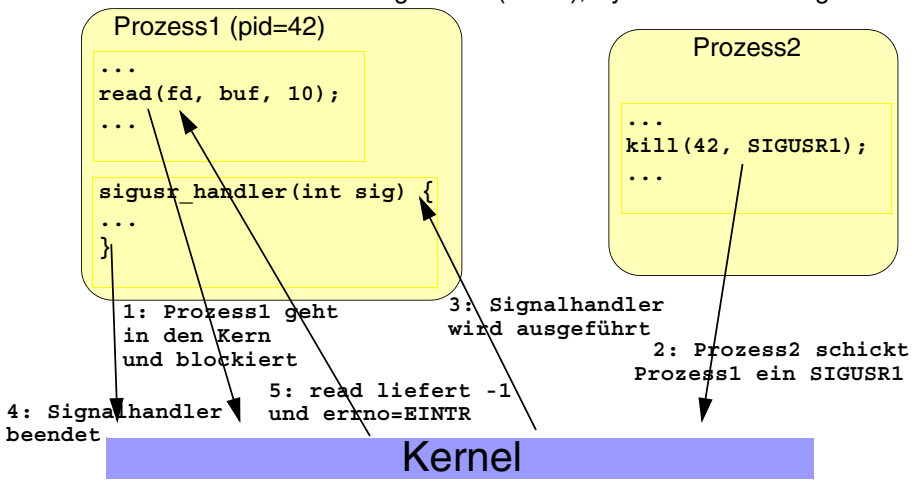
- `pid_t wait(int *status)`
  - kehrt auch zurück, wenn Kind gestoppt wird
  - optional auch, wenn Kind fortgesetzt wird (waitpid-Option `WCONTINUED`)
- erkennbar an Wert von `*status`
- Auswertung mit Macros
  - `WIFEXITED(*status)`: Kind normal terminiert
  - `WIFSIGNALED(*status)`: Kind durch Signal terminiert
  - `WIFSTOPPED(*status)`: Kind gestoppt
  - `WIFCONTINUED(*status)`: gestopptes Kind fortgesetzt

4 signal()-Funktion

- ANSI-C definiert die **signal(2)**-Funktion zum Einrichten von Signalbehandlungen
  - Problem: ungenaue Spezifikation, da Prozesskonzept in ANSI-C nicht definiert
- BSD- und SystemV-UNIX-Systeme enthalten die signal-Funktion
  - Problem: Verhalten implementierungsspezifisch, entweder
    - Verhalten wie bei **sigaction**, mit Blockade von *mindestens* dem Auslösesignal während der Signalbehandlung, oder
    - *unreliable signals*: Signalbehandlung wird vor Ausführung der Behandlungsfunktion auf `SIG_DFL` zurückgesetzt
- **signal sollte in neuen Programmen nicht mehr benutzt werden**
  - ➤ nur sigaction verwenden!

U3-4 Unterbrechung blockierender Systemaufrufe

- Signale können blockierende Systemaufrufe unterbrechen
- blockierter Prozess wird aufgeweckt (*bereit*), Systemaufruf schlägt fehl

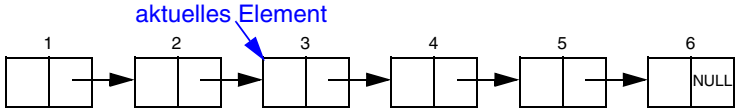


U3-4 Unterbrechung blockierender Systemaufrufe

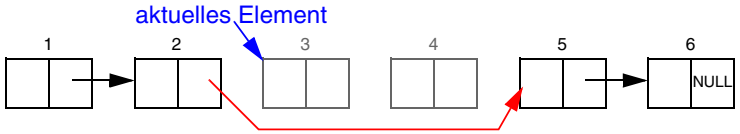
- betrifft nur blockierende Systemaufrufe (z.B. `wait()` oder `read()`)
- Systemaufruf setzt `errno` auf `EINTR` und kehrt mit Fehler zurück
- SUSv3: Behandlungsflag `SA_RESTART`:
  - Unterbricht ein Signal, für dessen Behandlung das `SA_RESTART`-Flag gesetzt wurde, einen blockierten Systemaufruf, so wird nach Ende der Signalbehandlung der Systemaufruf automatisch neu aufgesetzt
  - Ausnahme: `pause()` und `sigsuspend()` werden nie fortgesetzt

U3-5 Signale und Race-Conditions

- Signale erzeugen Nebenläufigkeit (Race-Conditions möglich) innerhalb des Prozesses (vgl. Nebenläufigkeit durch Interrupts, Vorlesung *B | V Betriebssystemebene, Seite 25* und *B | VI Prozesse, Seite 17 ff*)
- Beispiel:
  - Programm durchläuft gerade eine verkettete Liste



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



1 Synchronisation

- Lösung: Signal während Ausführung des kritischen Abschnitts blockieren
  - nur kritische Signale blockieren
  - kritische Abschnitte so kurz wie möglich halten (Risiko: Verlust von Signalen)
- Problem betrifft auch Bibliotheksfunktionen:
  - Aufruf von Bibliotheksfunktionen, z.B. `malloc` wird durch Signal unterbrochen und nach Ausführung des Signalhandlers fortgesetzt
  - Signalhandler ruft auch `malloc` auf ➠ Race-Condition
- Lösung:
  - in Signalhandlern nur Funktionen aufrufen, die in SuSv3 nicht als *non-reentrant* gekennzeichnet sind (`readdir` ist z.B. nicht 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

2 Ändern der prozessweiten Signal-Maske

- Prozessweite Signal-Maske enthält die aktuell blockierten Signale

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

- how:
  - **SIG_BLOCK**: setzt Vereinigungsmenge übergebener und alter Maske
  - **SIG_SETMASK**: setzt übergebene Maske
  - **SIG_UNBLOCK**: setzt Schnittmenge inverser übergebener und alter Maske
- Beispiel: Blockieren von SIGUSR1 zusätzlich zu bereits blockierten Signalen

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


3 Warten auf Signale

- Szenario:
  - Prozess befindet sich in kritischem Abschnitt
  - Fortschritt hängt von der Behandlung eines blockierten Signals ab
    - ➤ Signal muss deblockiert werden
    - ➤ Prozess muss auf Eintreffen und Behandlung des Signals warten
    - ➤ Signal muss wieder blockiert 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 Ausführung der eingestellten Signalbehandlung
  - **sigsuspend** kehrt nach Ende der Signalbehandlung mit Fehler **EINTR** zurück und restauriert gleichzeitig die ursprüngliche Signal-Maske

3 Beispiel: Warten auf Signal

```
static int event = 0;

static void sigHandler() { event=1; }

void wait4event() {



    while (event == 0) {

        SUSPEND();

    }

}
```

- Nebenläufigkeitsproblem?

3 Beispiel: Warten auf Signal

```
static int event = 0;

static void sigHandler() { event=1; }

void wait4event() {



    BLOCK_SIGNAL();
    while (event == 0) {
        UNBLOCK_SIGNAL(); /* sigsuspend in seine drei */
        SUSPEND();        /* Teiloperationen aufgeteilt */
        BLOCK_SIGNAL();
    }
    UNBLOCK_SIGNAL();
}
```

➤ das Warten auf ein Signal ist in sich bereits ein kritischer Abschnitt!

- Lost-Wakeup-Problem gelöst?

3 Beispiel: Warten auf Signal

```
static int event = 0;

static void sigHandler() { event=1; }

void wait4event() {


   BLOCK_SIGNAL();
   while (event == 0) {
       UNBLOCK_SIGNAL(); /* sigsuspend in seine drei */
       SUSPEND();        /* Teiloperationen aufgeteilt */
       BLOCK_SIGNAL();
   }
   UNBLOCK_SIGNAL();
}
```

➤ Deblockieren und Schlafenlegen müssen atomar erfolgen

- **sigsuspend(2)** gewährleistet dies

4 Das volatile-Schlüsselwort

```
static int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    while (event == 0) ;

}
```

- Testen des Programm ohne (-O0) und mit (-O) Compiler-Optimierungen
- Welches Verhalten können Sie beobachten?

4 Das volatile-Schlüsselwort

```
static int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    while (event == 0) ;

}
```

```
; Ohne Optimierungen
0: push   %ebp
1: mov    %esp,%ebp
3: mov    0x80495c4,%eax
8: test   %eax,%eax
a: je     12 <w4e+0x3>
c: pop    %ebp
d: ret
```

```
; Mit Optimierungen
0: push   %ebp
1: mov    %esp,%ebp
3: cmpl   $0x0,0x80495c4
a: jne    1d <w4e+0xe>
c: jmp    1b <w4e+0xc>
e: pop    %ebp
f: ret
```


4 Das volatile-Schlüsselwort

- `event` wird nebenläufig verändert
- der Compiler hat hiervon keine Kenntnis
  - innerhalb der Schleife wird `event` nicht verändert
  - die Schleifenbedingung ist also beim erstmaligen Prüfen wahr oder falsch
  - Bedingung ändert sich aus Sicht des Compilers innerhalb der Schleife nicht
    - ➤ Endlosschleife, wenn Bedingung nicht von vornherein falsch
- **volatile** zur Kennzeichnung von Variablen, die extern verändert werden
  - durch andere Kontrollflüsse
  - durch die Hardware (z.B. Geräteregister)
- Zugriffe auf volatile-Variablen werden vom Compiler nicht optimiert

4 Das volatile-Schlüsselwort

```
static volatile int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    while (event == 0) ;

}
```

- Erzwingt erneutes Laden von **event** in jedem Schleifendurchlauf

U3-6 Duplizieren von Filedeskriptoren

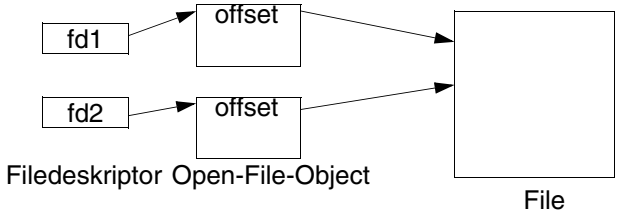
- Ziel: geöffnete Datei soll als stdout/stdin verwendet werden
- `newfd = dup(fd)`: Dupliziert Filedeskriptor fd, d.h. Lesen/Schreiben auf newfd ist wie Lesen/Schreiben auf fd
- `dup2(fd, newfd)`: Dupliziert FD in anderen FD (newfd), falls newfd schon geöffnet ist, wird newfd erst geschlossen
- Verwenden von dup2, um stdout umzuleiten:

```
int fd = open("/dev/null", O_WRONLY);
dup2(fd, STDOUT_FILENO);
printf("Hallo\n"); // wird nach /dev/null geschrieben
```

- Erinnerung: offene Filedeskriptoren werden bei fork und exec vererbt

U3-6 Duplizieren von Filedeskriptoren

- erneutes Öffnen eines Files



- bei dup werden FD dupliziert, aber Files werden nicht neu geöffnet!