

# U3 UNIX-Signale

---

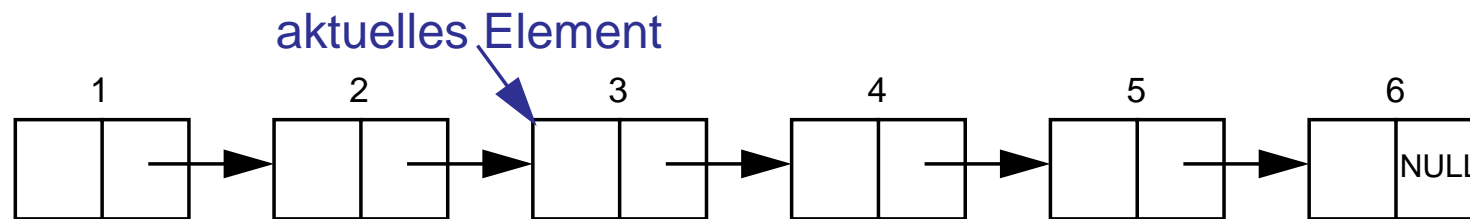
- Besprechung der Aufgabe 2: sister
- Nebenläufigkeit durch Signale
- Nebenläufiger Zugriff auf Variablen
- Passives Warten auf ein Signal
- Nachtrag zur Signalbehandlungsschnittstelle
- Duplizieren von Filedeskriptoren
- Nachtrag zu wait/waitpid
- waitpid und SIGCHLD
- Ziele der Aufgabe 3: josh

# U3-1 Nebenläufigkeit durch Signale

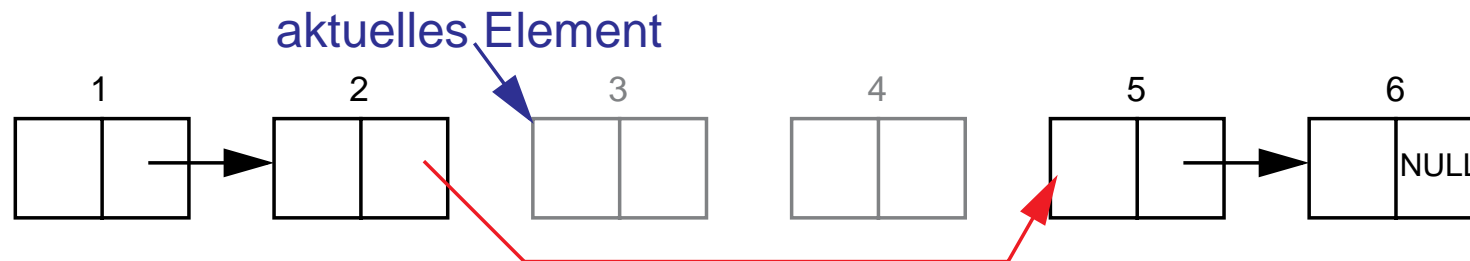
- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses (vgl. Nebenläufigkeit durch Interrupts, Vorlesung *B / V-4*, Seite 26 ff. und *B / VI-2*, Seite 18 ff.)

- 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 Grundsätzliche Fragestellung

---

- Welche Art von Nebenläufigkeit liegt vor?
- Welche Art der Synchronisation sollte verwendet werden?

# 1 Grundsätzliche Fragestellung

---

- Welche Art von Nebenläufigkeit liegt vor?
- Welche Art der Synchronisation sollte verwendet werden?

## 2 Lösung

---

- Asymmetrische, nicht-gleichberechtigte Kontrollflüsse:
  - ◆ Hauptprogramm (jederzeit unterbrechbar)
  - ◆ Signalbehandlung (nicht unterbrechbar, *Run-to-Completion*-Semantik)
- Meist Verwendung einseitiger Synchronisation:
  - ◆ Signal während der Ausführung des kritischen Abschnitts blockieren
    - nur kritische Signale blockieren
    - kritische Abschnitte so kurz wie möglich halten (Risiko: Verlust von Signalen)

### 3 Bibliotheksfunktionen

- Während der Ausführung einer Bibliotheksfunktion kann der dazugehörige interne Zustand inkonsistent sein
  - Beispiel `halde`: `fsp->next` wird auf `0xbaadf00d` gesetzt, danach wird `fsp` auf die neue Verwaltungsstruktur umgesetzt
- Eine Unterbrechung durch eine Signalbehandlungsfunktion ist unproblematisch, solange diese nicht auf den selben Zustand zugreift
- Wird auf den selben Zustand zugegriffen, müssen geeignete Maßnahmen ergriffen werden:
  - ◆ in Signal-Handlern keine Funktionen aufrufen, die in SuSv3 als *non-reentrant* gekennzeichnet sind
  - ◆ oder Signal während Ausführung der Funktion im Hauptprogramm blockieren
- Vorsicht: Auf den selben Zustand können u. U. auch verschiedene Funktionen zugreifen, z. B. `malloc()` und `free()`

## 4 `errno` - gemeinsamer Zustand

- Die meisten Bibliotheksfunktionen teilen sich als gemeinsamen Zustand die `errno`-Variable
  - ◆ Änderungen der `errno` im Signalhandler können die Fehlerbehandlung im Hauptprogramm durcheinander bringen
  - ◆ Lösung: Kontext-Sicherung
    - Beim Betreten der Signalhandler-Funktion die `errno` sichern und vor dem Verlassen wieder restaurieren

## U3-2 Nebenläufiger Zugriff auf Variablen

```
static int event = 0;

static void sigHandler() {
    event = 1;
}

void wait4event(void) {
    while (event == 0);
}
```

- Testen des Programms ohne (-O0) und mit (-O3) Compiler-Optimierungen
- Welches Verhalten lässt sich beobachten?

## U3-2 Nebenläufiger Zugriff auf Variablen

```
static int event = 0;

static void sigHandler() {
    event = 1;
}

void wait4event(void) {
    while (event == 0);
}
```

```
; Ohne Optimierungen
wait4event:
.L4:
    movl event, %eax
    testl %eax, %eax
    je .L4
    ret
```

```
; Mit Optimierungen
wait4event:
    movl event, %eax
    testl %eax, %eax
    je .L7
    rep
    ret
.L7:
    jmp .L7
```



## U3-2 Nebenläufiger Zugriff auf Variablen

- `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
- Abhilfe: `volatile` zur Kennzeichnung von Variablen, die extern verändert werden
  - ◆ durch andere Kontrollflüsse
  - ◆ durch die Hardware (z. B. Gerätereister)
- Zugriffe auf volatile-Variablen werden vom Compiler nicht optimiert

## U3-2 Nebenläufiger Zugriff auf Variablen

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

void wait4event(void) {
    while (event == 0);
}
```

- Erzwingt erneutes Laden von `event` in jedem Schleifendurchlauf

## U3-3 Passives Warten auf ein Signal

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

void wait4event(void) {

    while (event == 0) {

        SUSPEND();

    }

}
```

■ Nebenläufigkeitsproblem?

## U3-3 Passives Warten auf ein Signal

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

void wait4event(void) {
    BLOCK_SIGNAL();
    while (event == 0) {
        UNBLOCK_SIGNAL();
        SUSPEND();
        BLOCK_SIGNAL();
    }
    UNBLOCK_SIGNAL();
}
```

◆ Prüfen der Wartebedingung + Schlafenlegen ist ein kritischer Abschnitt!

■ Nebenläufigkeitsproblem (Lost-Wakeup-Problem) gelöst?

# U3-3 Passives Warten auf ein Signal

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

void wait4event(void) {
    BLOCK_SIGNAL();
    while (event == 0) {
        UNBLOCK_SIGNAL();
        SUSPEND();
        BLOCK_SIGNAL();
    }
    UNBLOCK_SIGNAL();
}
```

◆ Deblockieren und Schlafenlegen müssen atomar erfolgen

■ `sigsuspend(2)` gewährleistet dies

# U3-4 Nachtrag zur Signalbehandlungsschnittstelle

## 1 Ändern der prozessweiten Signal-Maske

- Prozessweite Signal-Maske enthält die aktuell blockierten Signale

```
int sigprocmask(int how,          // Verknuepfung der Masken
                const sigset_t *set, // Neue Maske
                sigset_t *oset);    // Alte Maske (Ausgabe)
```

- Modus (**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);
```

## 2 Passiv auf Signale warten

### ■ 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 den Prozess
- ◆ Ein Signal, das nicht in der Maske enthalten ist, führt zur Ausführung der vorher festgelegten Signalbehandlung
- ◆ **sigsuspend** kehrt nach Ende der Signalbehandlung mit Fehler **EINTR** zurück und restauriert gleichzeitig die ursprüngliche Signal-Maske

## U3-5 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 Filedeskriptor `fd` in anderen Filedeskriptor (`newfd`), falls `newfd` schon geöffnet ist, wird `newfd` erst geschlossen
- Verwenden von `dup2`, um stdout umzuleiten:

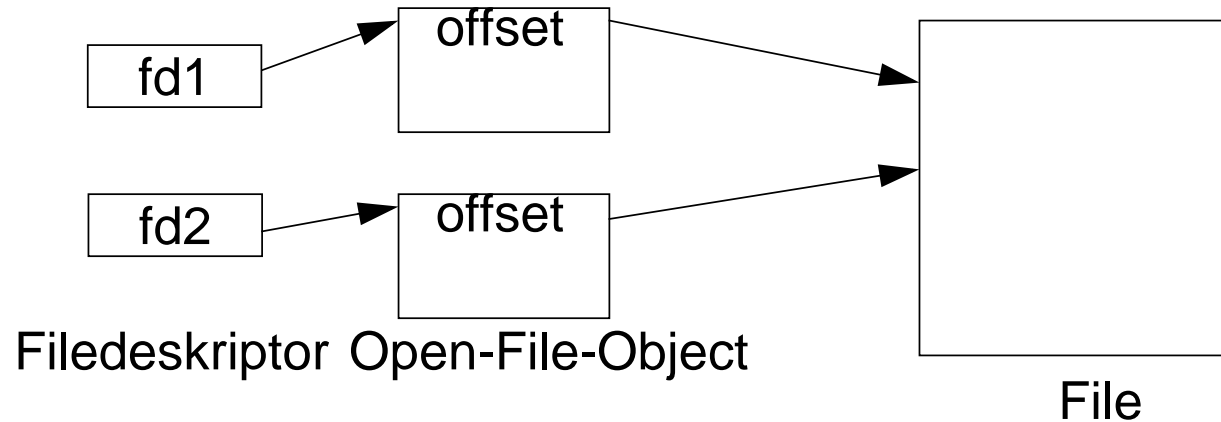
```
int fd = open("/dev/null", O_WRONLY);  
dup2(fd, STDOUT_FILENO);  
printf("Hallo\n"); // wird nach /dev/null geschrieben
```

- Erinnerung: offene Filedeskriptoren werden bei `fork(2)` und `exec(2)` vererbt

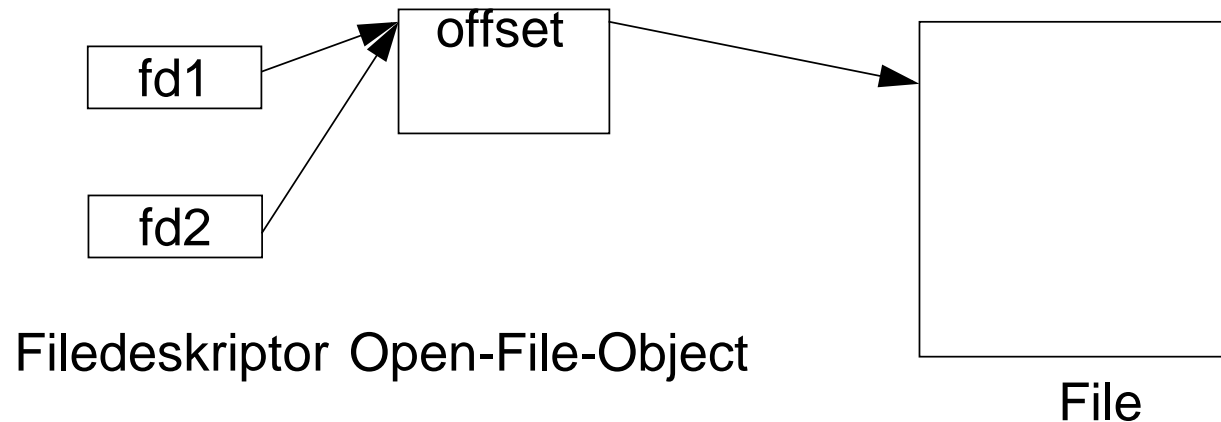


## U3-5 Duplizieren von Filedeskriptoren

- erneutes Öffnen einer Datei



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



## U3-6 Nachtrag zu wait/waitpid

### ■ `pid_t waitpid(pid_t pid, int *status, int options)`

◆ kehrt optional auch zurück, wenn ein Kind ...

- ... gestoppt wird (Option **WUNTRACED**)
- ... fortgesetzt wird (Option **WCONTINUED**)

### ■ Auswertung von `status` mit Makros:

◆ **WIFEXITED(status)**: Kind hat sich normal beendet

- Ermitteln des Exitstatus mit **WEXITSTATUS(status)**

◆ **WIFSIGNALED(status)**: Kind wurde durch ein Signal terminiert

- Ermitteln des Signals mit **WTERMSIG(status)**

◆ **WIFSTOPPED(status)**: Kind wurde gestoppt

- Ermitteln des Signals mit **WSTOPSIG(status)**

◆ **WIFCONTINUED(status)**: gestopptes Kind wurde fortgesetzt

## U3-7 wait/waitpid und SIGCHLD

---

- Szenario: **waitpid**-Aufruf sowohl im Hauptprogramm als auch im Signalhandler für SIGCHLD
  - ◆ Welcher der beiden **waitpid**-Aufrufe räumt den Zombie ab und erhält dessen Status?
    - Das Verhalten in diesem Fall ist betriebssystemsspezifisch - es existiert keine portable Lösung!
  - ◆ Daher darf **waitpid** nur im Signalhandler aufgerufen werden
    - Das Warten auf Vordergrundprozesse muss mit Hilfe von **sigsuspend** realisiert werden

## U3-8 Ziele der Aufgabe 3: josh

---

- Signale unter UNIX bilden die Konzepte *Trap* und *Interrupt* für die Interaktion zwischen Betriebssystem und Prozessen nach
  - ◆ praktischer Umgang mit diesen Konzepten
- Erkennen von Nebenläufigkeitsproblemen
- Beheben der Nebenläufigkeitsprobleme durch geeignete Koordinierungsmaßnahmen