

U3 UNIX-Signale

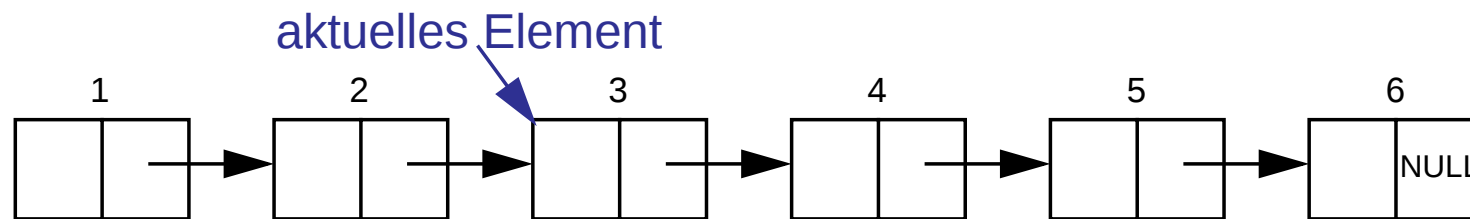
- Besprechung der Aufgabe 2: sister
- Nebenläufigkeit durch Signale
- Aktives Warten auf Ereignisse
- Probleme beim passiven Warten (auf Signale)
- Nachtrag zur Signalbehandlungsschnittstelle
- Duplizieren von Filedeskriptoren
- Nachtrag zu wait/waitpid
- waitpid und SIGCHLD
- Ziele der Aufgabe 3: josh

U3-1 Nebenläufigkeit durch Signale

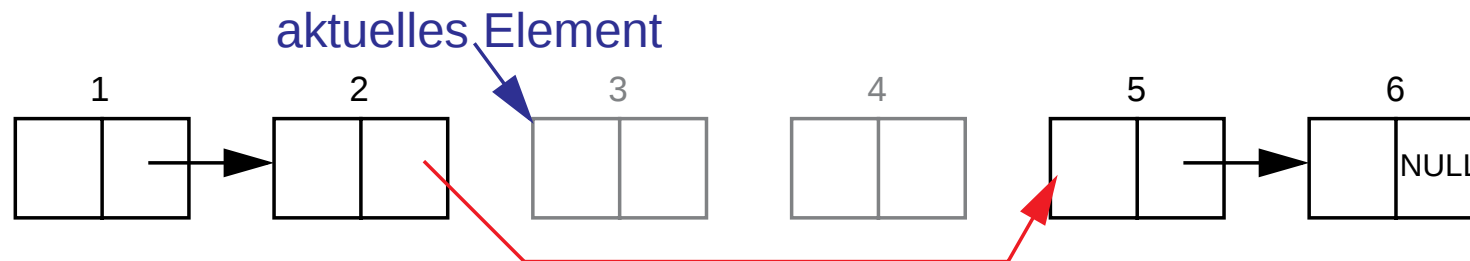
- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses (vgl. Nebenläufigkeit durch Interrupts, Vorlesung *B | V-4*, Seite 26ff. und *B | VI-2*, Seite 18 ff)

- 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 Problem

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

1 Problem

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

2 Synchronisation

- 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 `0xabadbabe` 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 Signalhandlern nur Funktionen aufrufen, die in SuSv3 nicht als *non-reentrant* gekennzeichnet sind
 - ◆ oder Signal während Ausführung der Funktion im Hauptprogramm blockieren
- 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 Aktives Warten auf Ereignisse

```
static int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    while (event == 0) ;

}
```

- Testen des Programm ohne (-O0) und mit (-O3) Compiler-Optimierungen
- Welches Verhalten können Sie beobachten?

U3-2 Aktives Warten auf Ereignisse

- **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. Gerätereister)
- Zugriffe auf volatile-Variablen werden vom Compiler nicht optimiert

U3-2 Aktives Warten auf Ereignisse

```
static volatile int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    while (event == 0) ;

}
```

- Erzwingt erneutes Laden von `event` in jedem Schleifendurchlauf

U3-3 Probleme beim passiven Warten

```
static volatile int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    while (event == 0) {

        SUSPEND();

    }

}
```

■ Nebenläufigkeitsproblem?

1 Beispiel: Warten auf Signal

```
static volatile int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

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

- ◆ das Prüfen der Wartebedingung und das Schlafenlegen ist in sich bereits ein kritischer Abschnitt!

- Nebenläufigkeitsproblem (Lost-Wakeup-Problem) gelöst?

1 Beispiel: Warten auf Signal

```
static volatile int event = 0;

static void sigHandler() { event=1; }

void wait4event() {

    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,          // 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);
```

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 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

U3-5 Duplizieren von Filedeskriptoren

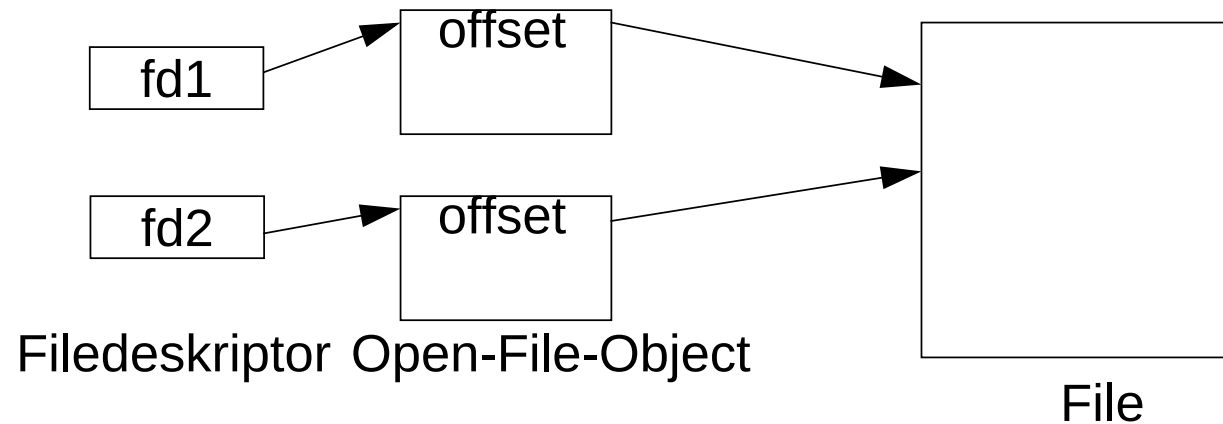
- Ziel: geöffnete Datei soll als stdout/stdin verwendet werden
- `newfd = dup(fd)`: Dupliziert Filedeskriptor `fd`, d.h. Lesen/Schreiben auf `newfd` ist wie Lesen/Schreiben auf `fd`
- `dup2(fd, newfd)`: Dupliziert Filedeskriptor `fd` in anderen Filedeskriptor (`newfd`), falls `newfd` schon geöffnet ist,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

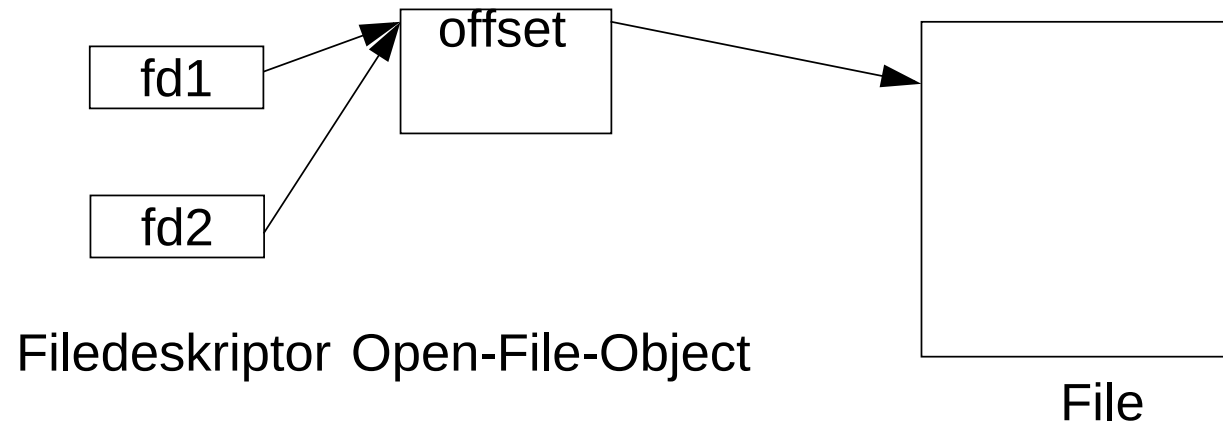
- Erinnerung: offene Filedeskriptoren werden bei `fork(2)` und `exec(2)` vererbt

U3-5 Duplizieren von Filedeskriptoren

- erneutes Öffnen eines Files



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



U3-6 Nachtrag zu wait/waitpid

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

U3-7 wait/waitpid und SIGCHLD

- Szenario: `waitpid`-Aufruf sowohl im Hauptprogramm als auch im Signalhandler für `SIGCHLD`
 - ◆ Betriebssystemspezifisches Verhalten: Welcher der beiden `waitpid`-Aufrufe räumt den Zombie ab und erhält dessen Status?
 - Daher sollte `waitpid` nur vom Signalhandler aufgerufen 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