

# Übungen zu Systemprogrammierung 2 (SP2)

## Ü3 – UNIX-Signale

**Christian Eichler, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

WS 2017 – 27. November bis 01. Dezember 2017

[http://www4.cs.fau.de/Lehre/WS17/V\\_SP2](http://www4.cs.fau.de/Lehre/WS17/V_SP2)



# Agenda

---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal
- 3.4 Mehr Details zu UNIX-Signalen
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden



# Agenda

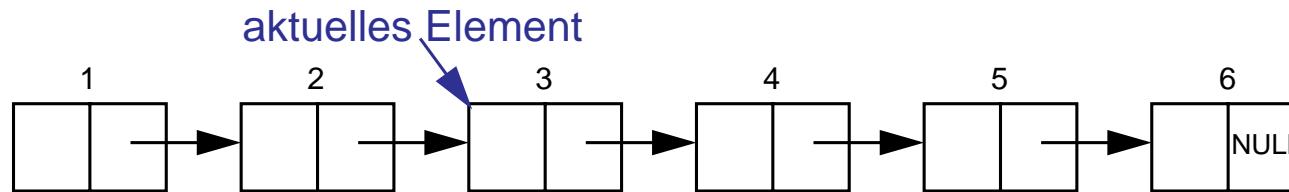
---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal
- 3.4 Mehr Details zu UNIX-Signalen
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden

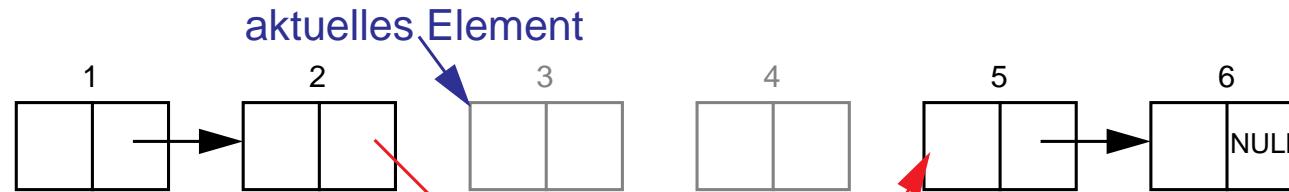


# Nebenläufigkeit durch Signale

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses (vgl. Nebenläufigkeit durch Interrupts, Vorlesung B | V.3, Seite 24 ff.)
- Während der Ausführung eines Programms können Teile seines Zustands vorübergehend inkonsistent sein
- Unterbrechung durch eine Signalbehandlungsfunktion ist problematisch, falls diese auf den selben Zustand zugreift
- 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



? **Welche Art von Nebenläufigkeit liegt vor?**

- Symmetrische, gleichberechtigte Kontrollflüsse*
- Asymmetrische, nicht-gleichberechtigte Kontrollflüsse*

? **Welche Art von Synchronisation sollte verwendet werden?**

(→ Vorlesung C | X-1, Seite 22 ff.)

- Mehrseitige Synchronisation*
- Einseitige Synchronisation*



## ? Welche Art von Nebenläufigkeit liegt vor?

- Symmetrische, gleichberechtigte Kontrollflüsse
- Asymmetrische, nicht-gleichberechtigte Kontrollflüsse:
  1. Hauptprogramm (jederzeit unterbrechbar)
  2. Signalbehandlung (nicht unterbrechbar, *Run-to-Completion-Semantik*)

## ? Welche Art von Synchronisation sollte verwendet werden?

(→ Vorlesung C | X-1, Seite 22 ff.)

- Mehrseitige Synchronisation
- Einseitige Synchronisation:
  - Signal während der Ausführung des kritischen Abschnitts blockieren
  - Nur kritische Signale blockieren
  - Kritische Abschnitte so kurz wie möglich halten (Risiko: Verlust von Signalen)



- Die prozessweite Signalmaske enthält die aktuell blockierten Signale
  - Diese werden erst behandelt, sobald sie wieder deblockiert wurden
- Ändern der Maske mittels **sigprocmask(2)**:  
`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
  - **how**: Verknüpfungsmodus
    - **SIG\_BLOCK**: setzt Vereinigungsmenge aus alter Maske und **set**
    - **SIG\_UNBLOCK**: setzt Schnittmenge aus alter Maske und invertiertem **set**
    - **SIG\_SETMASK**: setzt **set** als neue prozessweite Maske
  - **oldset**: bisherige prozessweite Signalmaske (Ausgabeparameter); bei Desinteresse **NULL** übergeben
- **Beispiel:** Blockieren von **SIGUSR1** zusätzlich zu bereits blockierten Signalen

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



- Während der Ausführung einer Bibliotheksfunktion kann der dazugehörige interne Zustand inkonsistent sein
  - Beispiel halde:
    - Suche nach passendem freiem Block in der Freispeicherliste; anschließend Entfernen des gefundenen Blocks aus der Liste
    - Falls `malloc()` zwischen diesen beiden Schritten unterbrochen wird und die Signalbehandlungsfunktion ebenfalls `malloc()` aufruft, wird u. U. derselbe Block zweifach vergeben!
- Greift man im Rahmen der Signalbehandlung auf denselben Zustand zu, müssen geeignete Maßnahmen ergriffen werden:
  - Signal während Ausführung der betreffenden Funktionen im Hauptprogramm blockieren
- **Vorsicht:** Auf den selben Zustand können u. U. auch verschiedene Funktionen zugreifen, z. B. `malloc()` und `free()`
- Funktionen, die in SUSv4 als *async-signal-safe* gekennzeichnet sind, müssen **nicht** geschützt werden



- Die meisten Bibliotheksfunktionen teilen sich als gemeinsamen Zustand die **errno**-Variable
  - Änderungen der **errno** im Signal-Handler können die Fehlerbehandlung im Hauptprogramm durcheinander bringen
  - **Lösung:** Kontext-Sicherung
    - Beim Betreten der Signalhandler-Funktion die **errno** sichern und vor dem Verlassen wiederherstellen
- Ein-/Ausgabeoperationen auf **FILE\*** schützen möglicherweise den Stream mit Hilfe eines Locks vor mehrfädigem Zugriff
  - Deadlock, falls eine E/A-Operation unterbrochen wird und im Signal-Handler auf den selben **FILE\*** zugegriffen wird
  - **Lösung:** keine Ein-/Ausgabe mit **FILE\*** in Signal-Handlern betreiben



# Agenda

---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal
- 3.4 Mehr Details zu UNIX-Signalen
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden



# Nebenläufiger Zugriff auf Variablen

---

```
static int event = 0;

static void sigHandler() {
    event = 1;
}

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

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



# Nebenläufiger Zugriff auf Variablen

```
static int event = 0;

static void sigHandler() {
    event = 1;
}

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

*; Ohne Optimierungen*

```
waitForEvent:
    nop
.L3:
    movl event, %eax
    testl %eax, %eax
    je .L3
    ret
```

*; Mit Optimierungen*

```
waitForEvent:
    movl event, %eax
    testl %eax, %eax
    jne .L2
.L5:
    jmp .L5
.L2:
    rep
    ret
```



# 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: Schlüsselwort **volatile** zur Kennzeichnung von Variablen, die extern verändert werden
  - durch andere Kontrollflüsse
  - durch die Hardware (z. B. in den Adressraum eingeblendete Geräteregister)
- Zugriffe auf **volatile**-Variablen werden vom Compiler nicht optimiert



# Nebenläufiger Zugriff auf Variablen

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

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

- Deklaration als **volatile** erzwingt erneutes Laden von `event` in jedem Schleifendurchlauf
- **Randnotiz:** Semantik von **volatile** ist in C/C++ schwächer als in Java (keine Speicherbarriere)



# Agenda

---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal**
- 3.4 Mehr Details zu UNIX-Signalen
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden



# Passives Warten auf ein Signal

```
static volatile int event = 0;  
  
static void sigHandler() {  
    event = 1;  
}  
  
void waitForEvent(void) {  
  
    while (event == 0) {  
  
        SUSPEND(); // Schlafen, bis ein Signal eintrifft  
    }  
}
```

- Nebenläufigkeitsproblem?



# Passives Warten auf ein Signal

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

void waitForEvent(void) {
    BLOCK_SIGNAL();
    while (event == 0) {
        UNBLOCK_SIGNAL();
        SUSPEND(); // Schlafen, bis ein Signal eintrifft
        BLOCK_SIGNAL();
    }
    UNBLOCK_SIGNAL();
}
```

- Nebenläufigkeitsproblem: Prüfen der Wartebedingung + Schlafenlegen ist ein kritischer Abschnitt!
- Nebenläufigkeitsproblem (*Lost Wakeup*) jetzt gelöst?



# Passives Warten auf ein Signal

```
static volatile int event = 0;

static void sigHandler() {
    event = 1;
}

void waitForEvent(void) {
    BLOCK_SIGNAL();
    while (event == 0) {
        UNBLOCK_SIGNAL();
        SUSPEND(); // Schlafen, bis ein Signal eintrifft
        BLOCK_SIGNAL();
    }
    UNBLOCK_SIGNAL();
}
```

- Prüfen der Wartebedingung + Schlafenlegen ist ein kritischer Abschnitt!
- Deblockieren des Signals und Schlafenlegen müssen atomar erfolgen
- Betriebssystemschnittstelle muss entsprechende Operation anbieten



# Passives Warten auf ein Signal

---

- Die Kombination der Pseudo-Operationen `UNBLOCK_SIGNAL()` + `SUSPEND()` + `BLOCK_SIGNAL()` lässt sich durch Aufruf von `sigsuspend()` realisieren
- Prototyp:  
`int sigsuspend(const sigset_t *mask);`
  - `sigsuspend()` merkt sich die aktuelle prozessweite Signalmaske, setzt `mask` als neue Signalmaske und legt den Prozess schlafen
  - Ein Signal, das nicht in `mask` enthalten ist, führt zur Ausführung der vorher festgelegten Signalbehandlung
  - `sigsuspend()` stellt nach Ende der Signalbehandlung die ursprüngliche Signalmaske wieder her und kehrt zurück
- Es ist garantiert, dass das Setzen der Maske und das Schlafenlegen atomar erfolgen



# Agenda

---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal
- 3.4 Mehr Details zu UNIX-Signalen**
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden



# Signale und fork(2)/exec(2)

---

- Kindprozess erzeugen mit `fork(2)`:
  - Kindprozess erbt Signalbehandlung und Signalmaske vom Vaterprozess
- Anderes Programm laden mit `exec(2)`:
  - Signalmaske wird beibehalten
  - Signalbehandlung wird beibehalten, falls `SIG_DFL` oder `SIG_IGN`
  - Benutzerdefinierte Signalbehandlung wird auf `SIG_DFL` zurückgesetzt  
(→warum? )



# Signale und fork(2)/exec(2)

---

- Kindprozess erzeugen mit `fork(2)`:
  - Kindprozess erbt Signalbehandlung und Signalmaske vom Vaterprozess
- Anderes Programm laden mit `exec(2)`:
  - Signalmaske wird beibehalten
  - Signalbehandlung wird beibehalten, falls `SIG_DFL` oder `SIG_IGN`
  - Benutzerdefinierte Signalbehandlung wird auf `SIG_DFL` zurückgesetzt  
(→ nach dem Laden des neuen Programms existiert die alte Signalbehandlungsfunktion nicht mehr)



# Nachtrag zu waitpid(2)

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Kehrt optional auch zurück, wenn ein Kindprozess
  - ... gestoppt wird (Option **WUNTRACED**)
  - ... fortgesetzt wird (Option **WCONTINUED**)
- Auswertung von **status** mit Makros (**if**-Kaskade notwendig!):
  - **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



# waitpid() und SIGCHLD

---

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



- Prototyp:

```
unsigned int sleep(unsigned int seconds);
```

- Legt den aufrufenden Prozess für `seconds` Sekunden schlafen
- Falls während des Schlafens ein Signal eintrifft, kehrt `sleep()` sofort zurück

- Rückgabewert:

- 0, falls volle Wartezeit absolviert
- Verbleibende Wartezeit, falls zwischendrin durch ein Signal unterbrochen

- Signale, die mit `sigprocmask()` blockiert sind, können **nicht** für ein vorzeitiges Aufwachen sorgen



# Agenda

---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal
- 3.4 Mehr Details zu UNIX-Signalen
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden

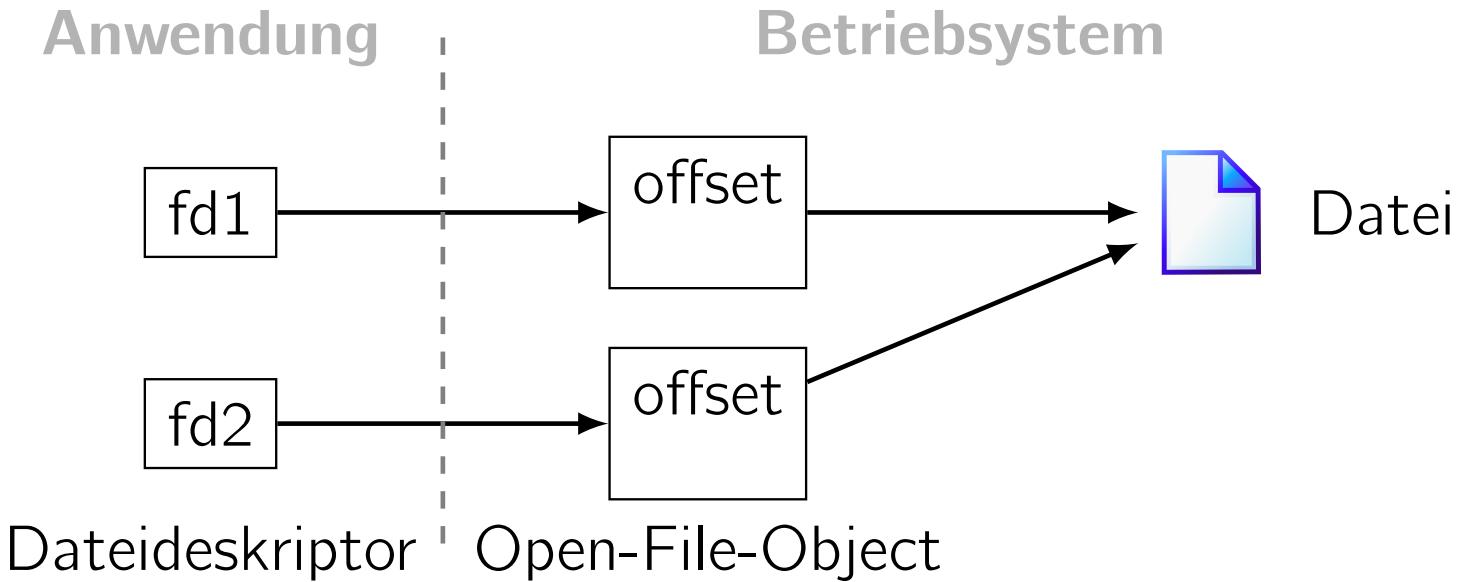


- Ziel: geöffnete Datei soll als **stdout/stdin** verwendet werden
- `newfd = dup(fd)`: Dupliziert Dateideskriptor `fd`, d. h.  
Lesen/Schreiben auf `newfd` ist wie Lesen/Schreiben auf `fd`
  - Die Nummer von `newfd` wird vom System gewählt
- `dup2(fd, newfd)`: Dupliziert Dateideskriptor `fd` in anderen Dateideskriptor (`newfd`); falls `newfd` schon geöffnet ist, wird `newfd` erst geschlossen
  - Die Nummer von `newfd` wird vom Benutzer vorgegeben
- 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 Dateideskriptoren werden bei `fork(2)` vererbt und bei `exec(2)` beibehalten



# Dateideskriptoren: Erneutes Öffnen einer Datei

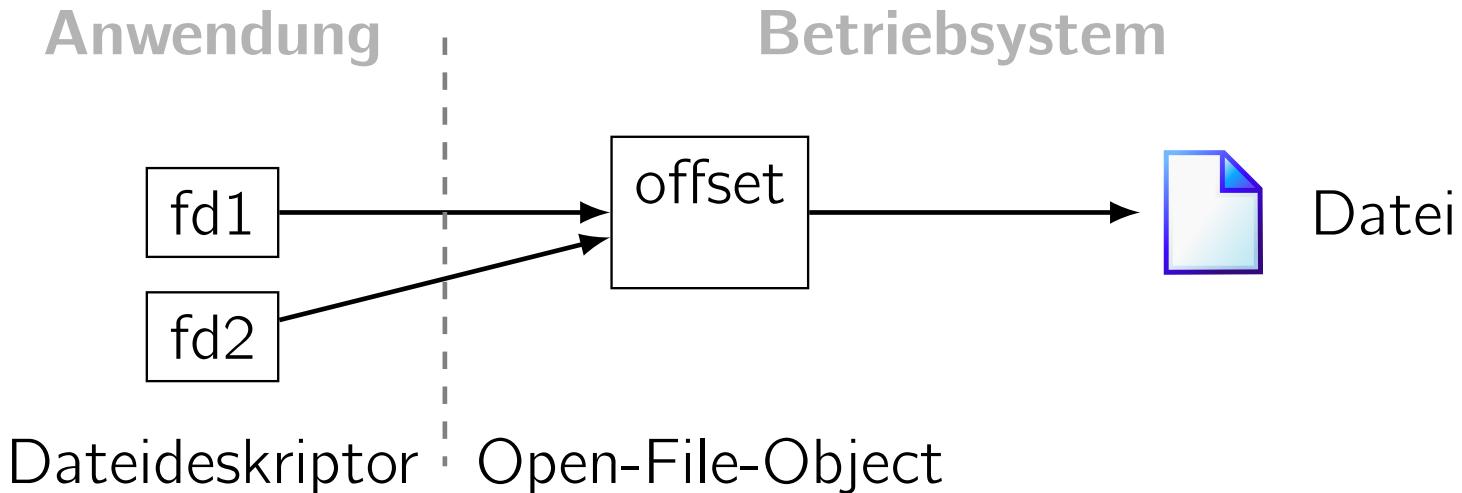


```
int fd1 = open("datei.txt", /*... */);  
int fd2 = open("datei.txt", /*... */);
```

- Erzeugen eines neuen Open-File-Objects und eines neuen Dateideskriptors



# Dateideskriptoren: dup(2)

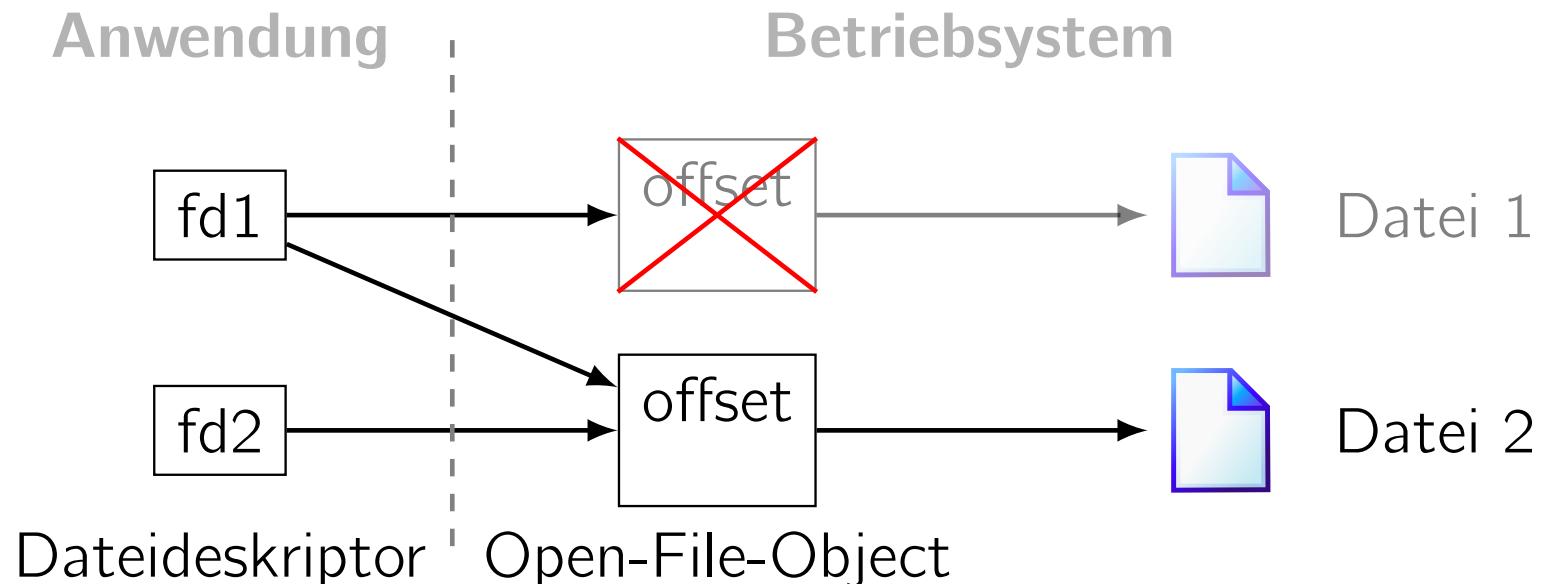


```
int fd1 = open(/*... */);  
int fd2 = dup(fd1);
```

- Der Dateideskriptor wird dupliziert
- aber: die Datei wird **nicht neu geöffnet**



# Dateideskriptoren: dup2(2)



```
int fd1 = open("datei1.txt", /*... */);
int fd2 = open("datei2.txt", /*... */);
//int dup2(int oldfd, int newfd);
dup2(fd2, fd1);
```

- Schließt `newfd` (falls zuvor geöffnet)
- Nutzt `newfd` als Wert für neuen Dateideskriptor



# Agenda

---

- 3.1 Nebenläufigkeit durch Signale
- 3.2 Nebenläufiger Zugriff auf Variablen
- 3.3 Passives Warten auf ein Signal
- 3.4 Mehr Details zu UNIX-Signalen
- 3.5 Umleiten von Dateien
- 3.6 Gelerntes anwenden



## „Aufgabenstellung“

- Programm schreiben, das passiv auf **SIGUSR1** wartet

