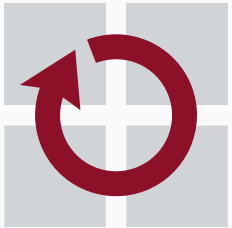
Übungen zu Systemprogrammierung 2

Ü3 – UNIX-Signale

Wintersemester 2025/26

Jürgen Kleinöder, Thomas Preisner, Tobias Häberlein, Ole Wiedemann

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Informatik 4
Systemsoftware

FAU
Friedrich-Alexander-Universität
Technische Fakultät

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

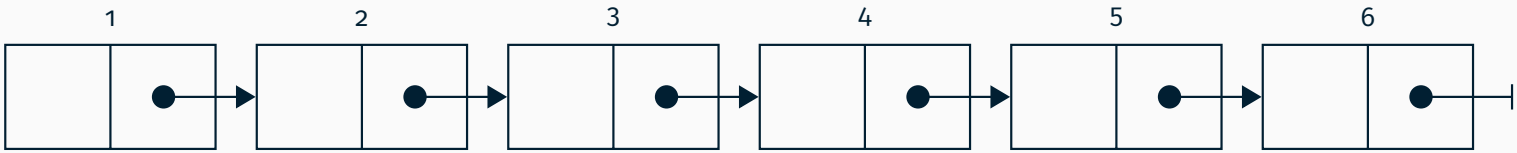
3.5 Umleiten von Dateien

3.6 Gelerntes anwenden

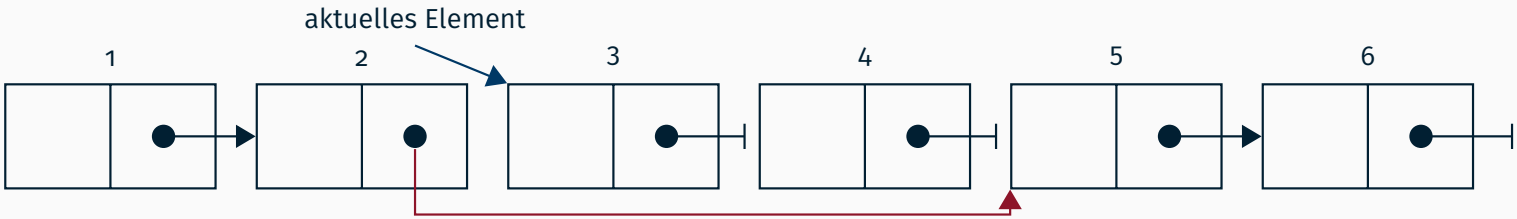
Agenda

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 denselben 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



Grundsätzliche Fragestellung

? 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

Grundsätzliche Fragestellung

? 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: Große Verzögerung von Signalen)

Signale blockieren

- Die prozessweite Signalmaske enthält die aktuell blockierten Signale
  - Diese werden erst behandelt, sobald sie wieder deblockiert wurden
- Ändern der Maske mittels `sigprocmask(3p)`:

```
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)
- **Beispiel:** Zusätzliches blockieren von `SIGUSR1`

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

Vorsicht bei Bibliotheksfunktionen!

- 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 → `signal-safety(7)`

Weitere Fallstricke

- 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

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


```c
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
```

Lade Variable event

Prüfen auf 0

Sprung, falls gleich 0

Sprung, falls ungleich 0

```
; 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

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?

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

Signale und fork(3p)/exec(3p)

- Kindprozess erzeugen mit fork(3p):
  - Kindprozess erbt Signalbehandlung und Signalmaske vom Elternprozess
- Anderes Programm laden mit exec(3p):
  - Signalmaske wird beibehalten
  - Signalbehandlung wird beibehalten, falls SIG_DFL oder SIG_IGN
  - Benutzerdefinierte Signalbehandlung wird auf SIG_DFL zurückgesetzt ($\rightarrow$ warum?)

Signale und fork(3p)/exec(3p)

- Kindprozess erzeugen mit fork(3p):
  - Kindprozess erbt Signalbehandlung und Signalmaske vom Elternprozess
- Anderes Programm laden mit exec(3p):
  - Signalmaske wird beibehalten
  - Signalbehandlung wird beibehalten, falls SIG_DFL oder SIG_IGN
  - Benutzerdefinierte Signalbehandlung wird auf SIG_DFL zurückgesetzt
    ($\rightarrow$ nach dem Laden des neuen Programms existiert die alte Signalbehandlungsfunktion nicht mehr)

Nachtrag zu `waitpid(3p)`

```
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(3p)` 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

`sleep(3p)` und Signale

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

Agenda

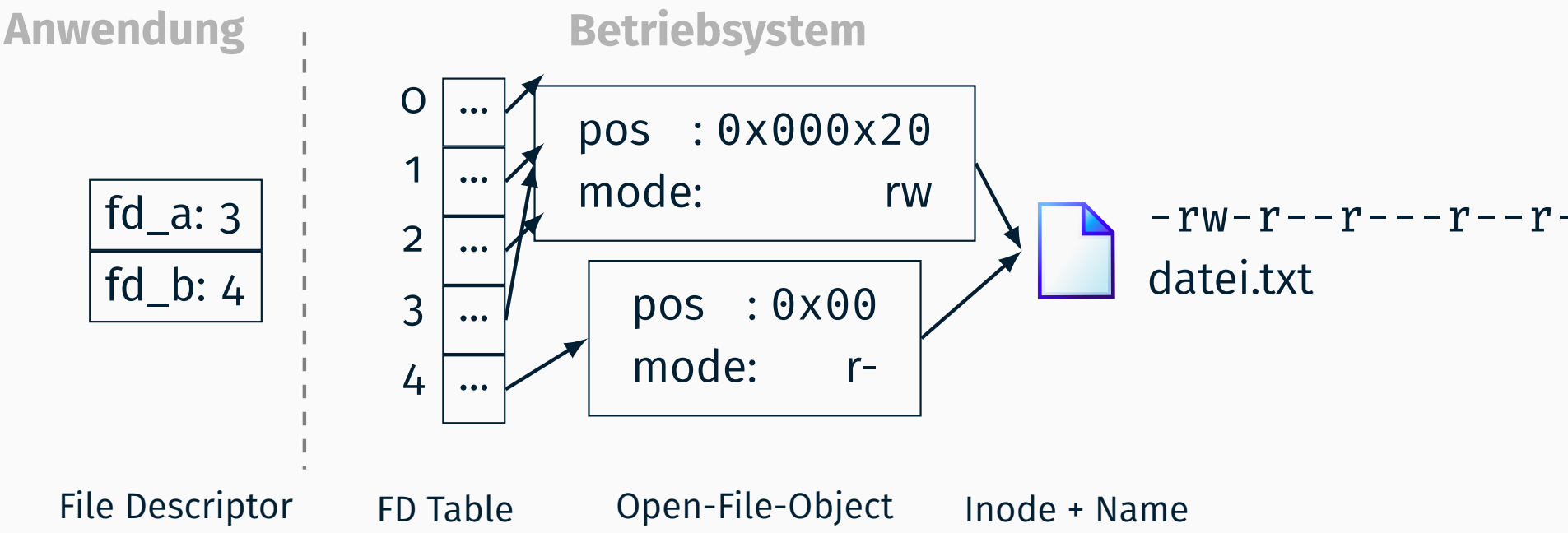
Umleiten von Dateien

- 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:

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

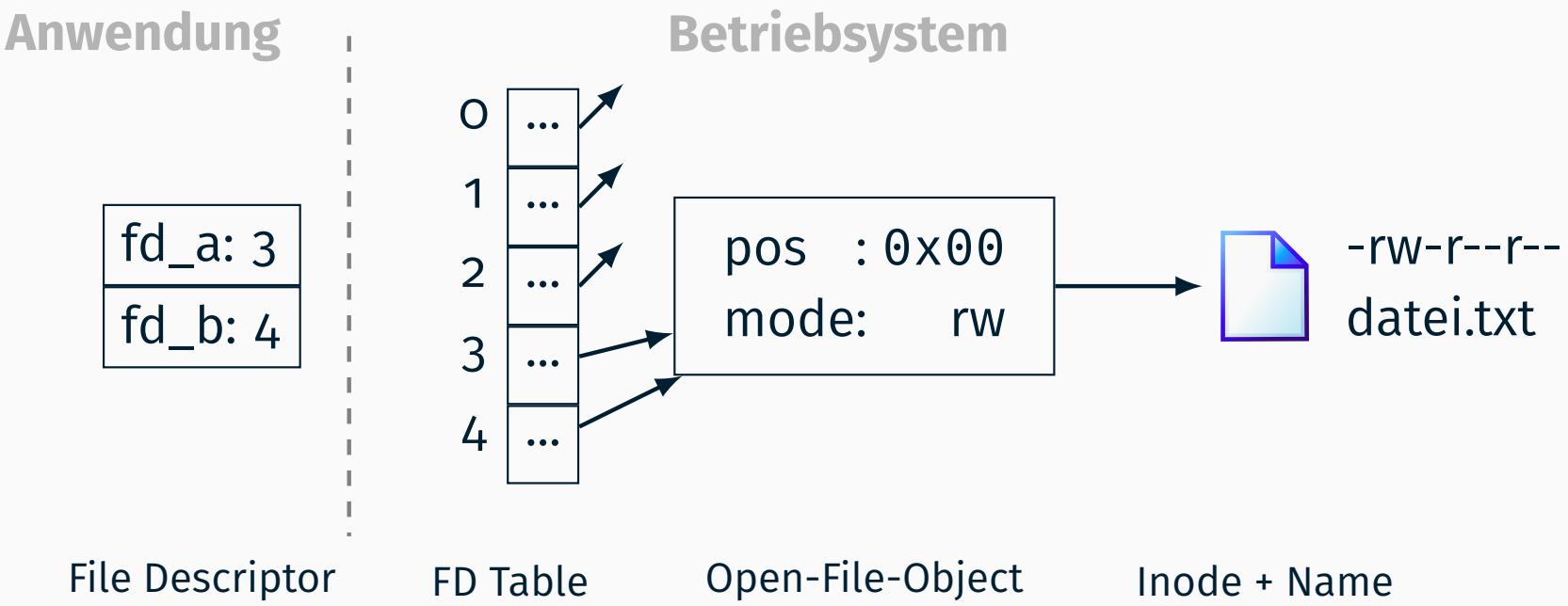
- Erinnerung: offene Dateideskriptoren werden bei `fork(3p)` vererbt und bei `exec(3p)` beibehalten

Dateideskriptoren: Arbeiten mit Dateien




```c
int fd_a = open("datei.txt", O_RDWR); // = 3
chmod("datei.txt", S_IRUSR | S_IRGRP | S_IROTH);
int fd_b = open("datei.txt", O_RDONLY); // = 4
write(fd_a, buffer, 0x20); // = 0x20
write(fd_b, buffer, 0x20); // = -EACCES
int fd_c = open("datei.txt", O_RDWR); // = -EACCES
```

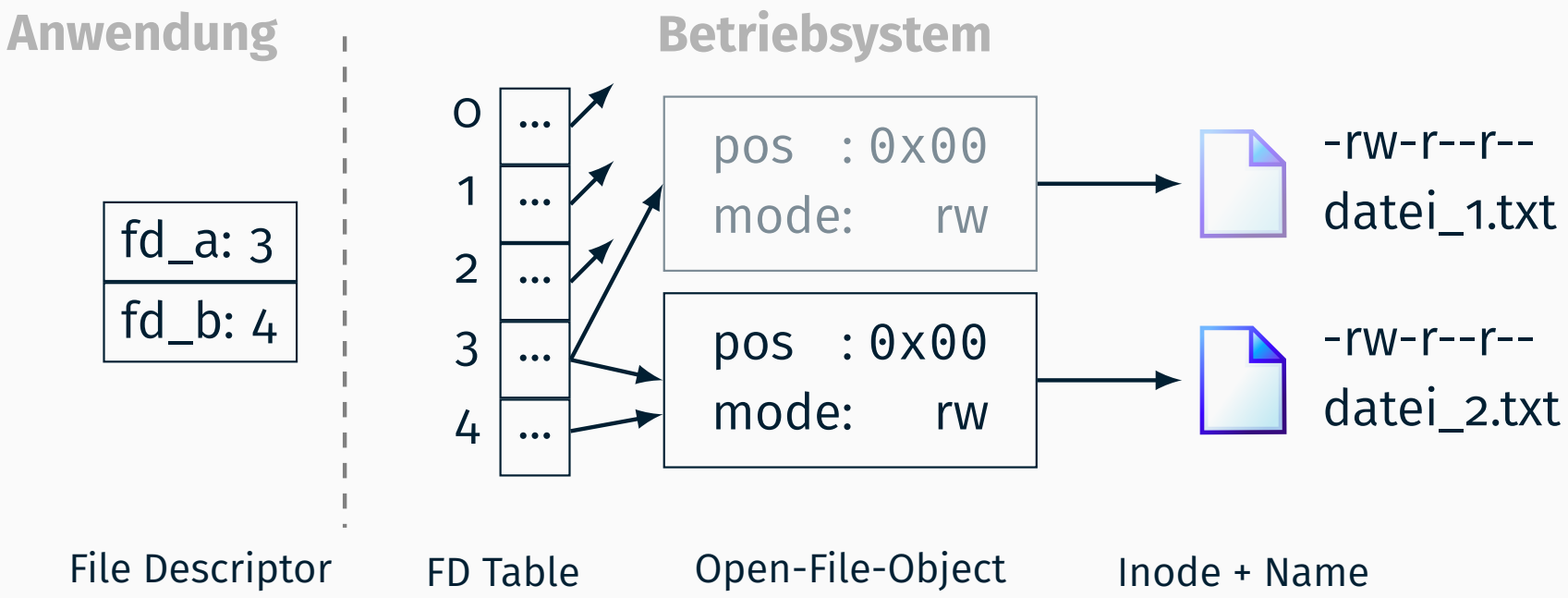
Dateideskriptoren: dup(3p)




```c
int fd_a = open("datei.txt", O_RDWR); // = 3
int fd_b = dup(fd_a); // = 4
```

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

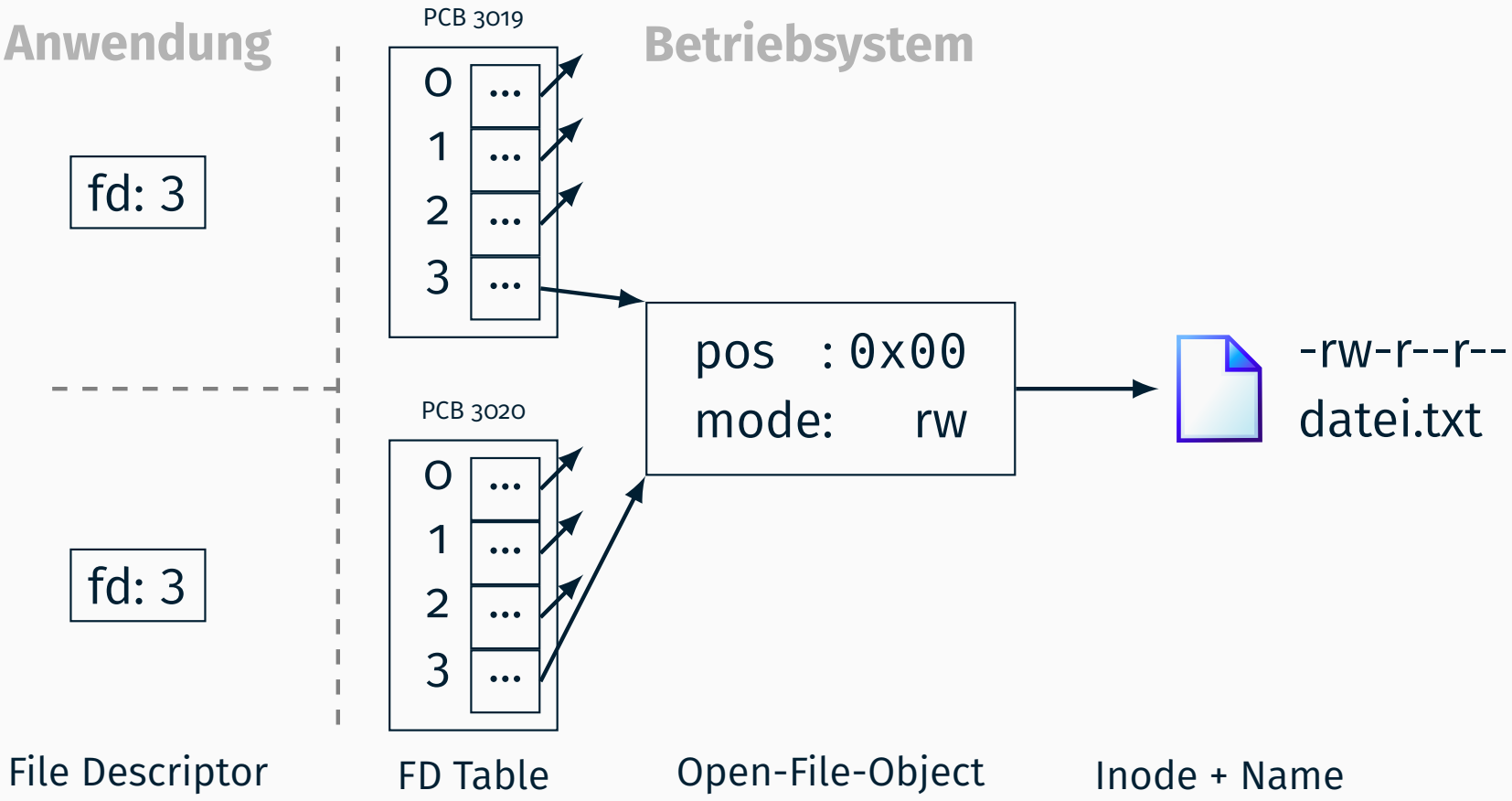
Dateideskriptoren: dup2( )




```c
int fd_a = open("datei_1.txt", O_RDWR); // = 3
int fd_b = open("datei_2.txt", O_RDWR); // = 4
// int dup2(int oldfd, int newfd);
dup2(fd_b, fd_a); // = 3
```

- Schließt `fd_a` (≙ `newfd`) (falls zuvor geöffnet)
- Lässt `fd_a` auf das Open-File-object von `fd_b` zeigen

Dateideskriptoren: fork(3p)




```
getpid(); // = 3019
int fd = open("datei.txt", O_RDWR); // = 3
fork(); // = 3020
```

Agenda

Aktive Mitarbeit!

„Aufgabenstellung“

- Programm schreiben, das passiv auf `SIGUSR1` wartet