U3 3. Übung

- Besprechung der 1. Aufgabe: simail
- Sockets: Server
- UNIX-Signale: Funktionsweise und Behandlung
- UNIX-API zur Signalbehandlung
- Aufgabe 2: sister
- Besprechung der Miniklausur

1 Verbindungsannahme durch Server

- Server:
  - ***listen(2)*** stellt ein, wie viele ankommende Verbindungswünsche gepuffert werden können (d.h. auf ein *accept* wartend)
    - Maximal möglich: **`SOMAXCONN`**
  - ***accept(2)*** nimmt Verbindung an:
    - *accept* blockiert solange, bis ein Verbindungswunsch ankommt
    - es wird ein neuer Socket erzeugt und an remote Adresse + Port (Parameter **`from`**) gebunden
      lokale Adresse + Port bleiben unverändert
    - dieser Socket wird für die Kommunikation benutzt
    - der ursprüngliche Socket kann für die Annahme weiterer Verbindungen genutzt werden

1 Verbindungsannahme durch Server - Beispiel

- Beispiel: Server, der alle Eingaben wieder zurückschickt (ohne Fehlerbehandlungen)

```
int fd;
fd = socket(PF_INET6, SOCK_STREAM, 0);

struct sockaddr_in6 name;
memset(&name, 0, sizeof(name));
name.sin6_family = AF_INET6;
name.sin6_port = htons(1112);
name.sin6_addr = in6addr_any;

bind(fd, (const struct sockaddr *)&name, sizeof(name));

listen(fd, SOMAXCONN);

int in_fd;
while ( (in_fd = accept(fd, NULL, NULL)) != -1 ) {
   char buf[1024];
   int n;
   while ( (n = read(in_fd, buf, sizeof(buf))) > 0 ) {
      write(in_fd, buf, n);
   }
   close(in_fd);
}

close(fd);
```

2 Abarbeiten von Anfragen

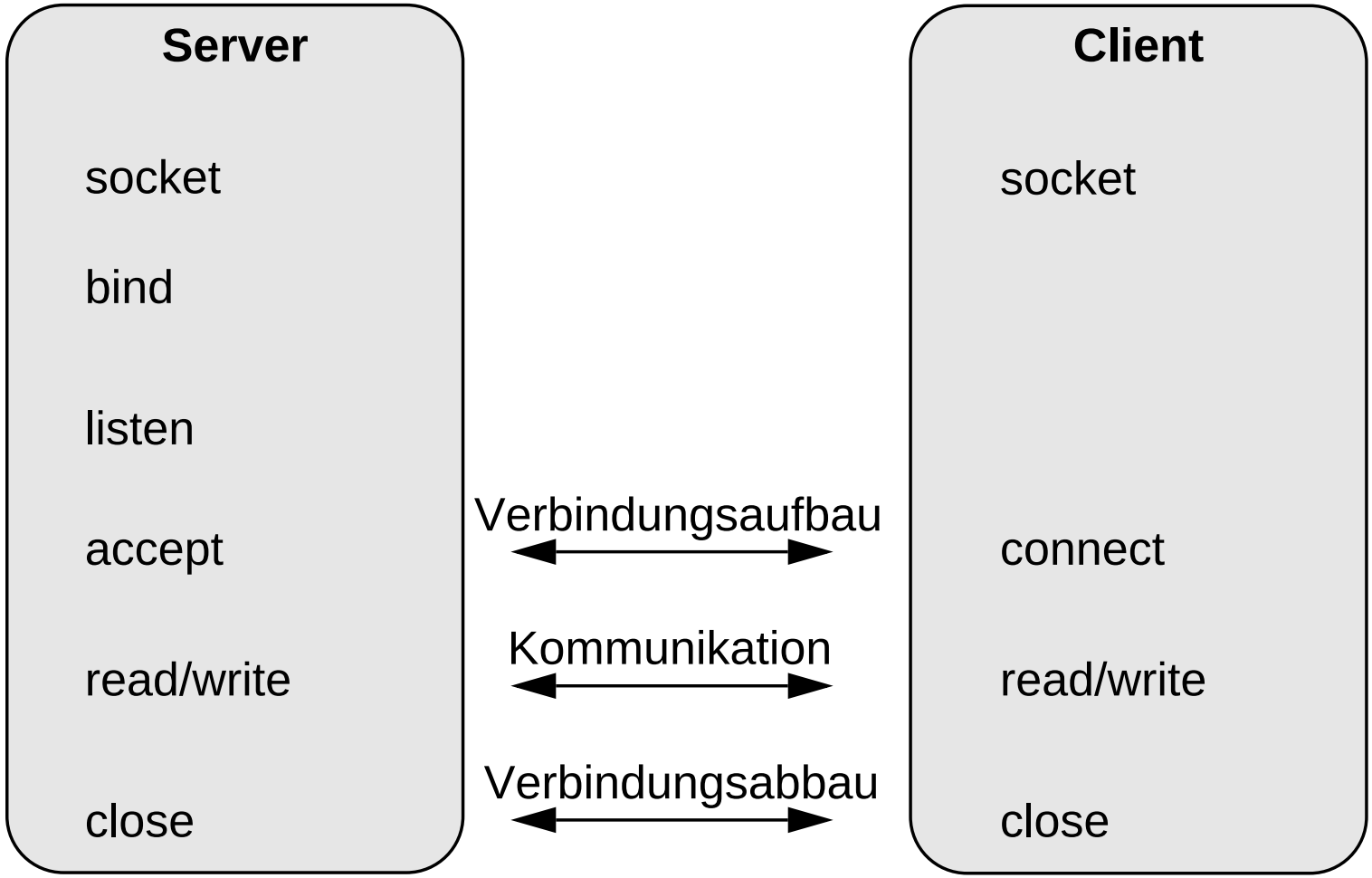
- Auszug aus Echo-Server

```
while ( (in_fd = accept(fd, NULL, NULL)) != -1 ) {
   char buf[1024];
   int n;
   while ( (n = read(in_fd, buf, sizeof(buf))) > 0 ) {
      write(in_fd, buf, n);
   }
   close(in_fd);
}
```

  - Neue eingehende Verbindung kann erst nach vollständiger Abarbeitung der vorherigen Anfrage angenommen werden.
  - Monopolisierung des Dienstes möglich (Denial-of-Service)

- Lösung: Abarbeitung der Anfrage in eigenen Aktivitätsträger auslagern
  - Ansatz 1: Mehrere Prozesse
    - Anfrage wird durch Kindprozess bearbeitet
  - Ansatz 2: Mehrere Threads
    - Anfrage wird durch einen Thread im gleichen Prozess bearbeitet

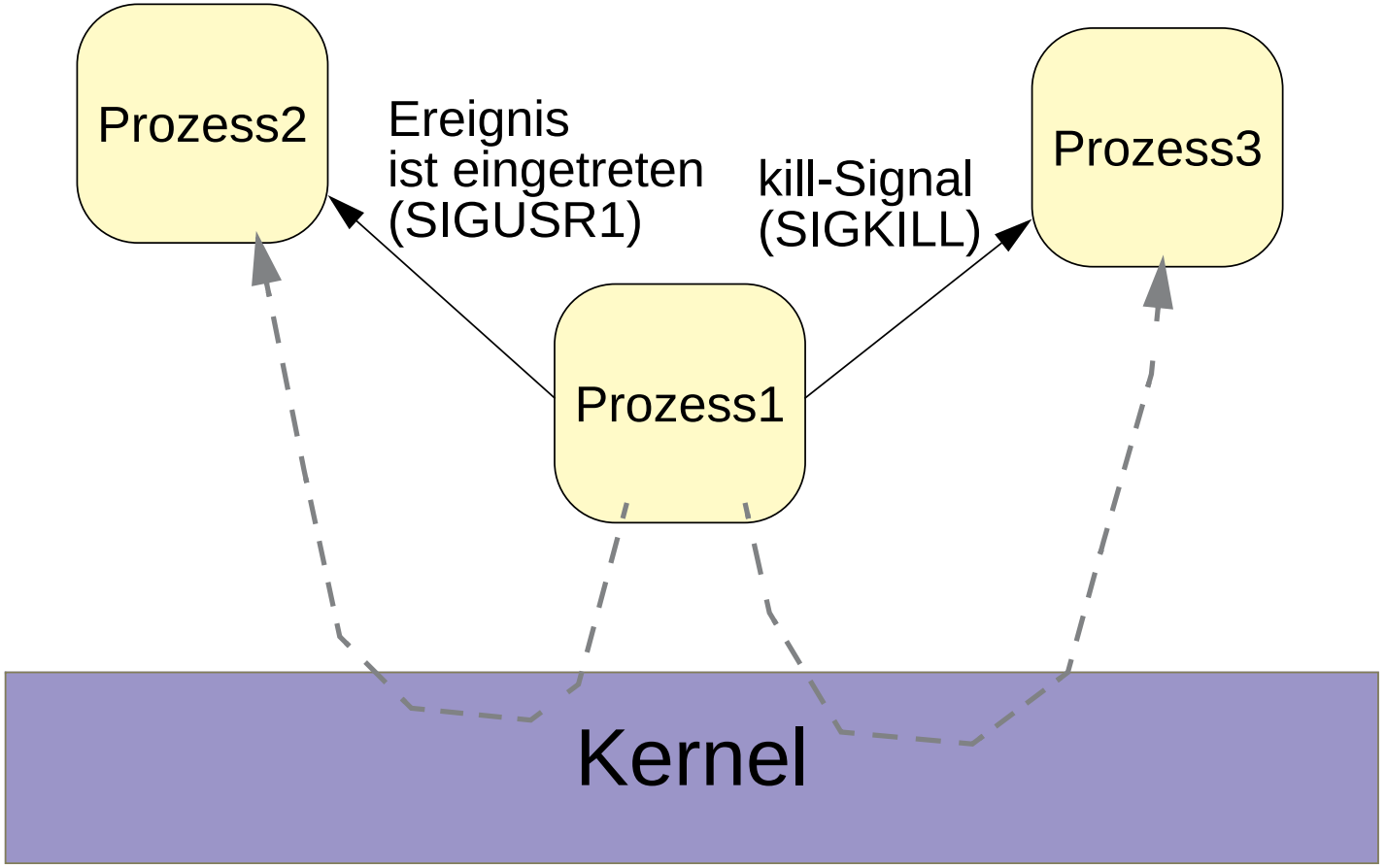
3 TCP-Sockets: Zusammenfassung



- Statt der Systemaufrufe **`read(2)`**/**`write(2)`** können auch Bibliotheksfunktionen wie zum Beispiel **`fgets(3)`**, **`fputs(3)`** oder **`fprintf(3)`** verwendet werden.

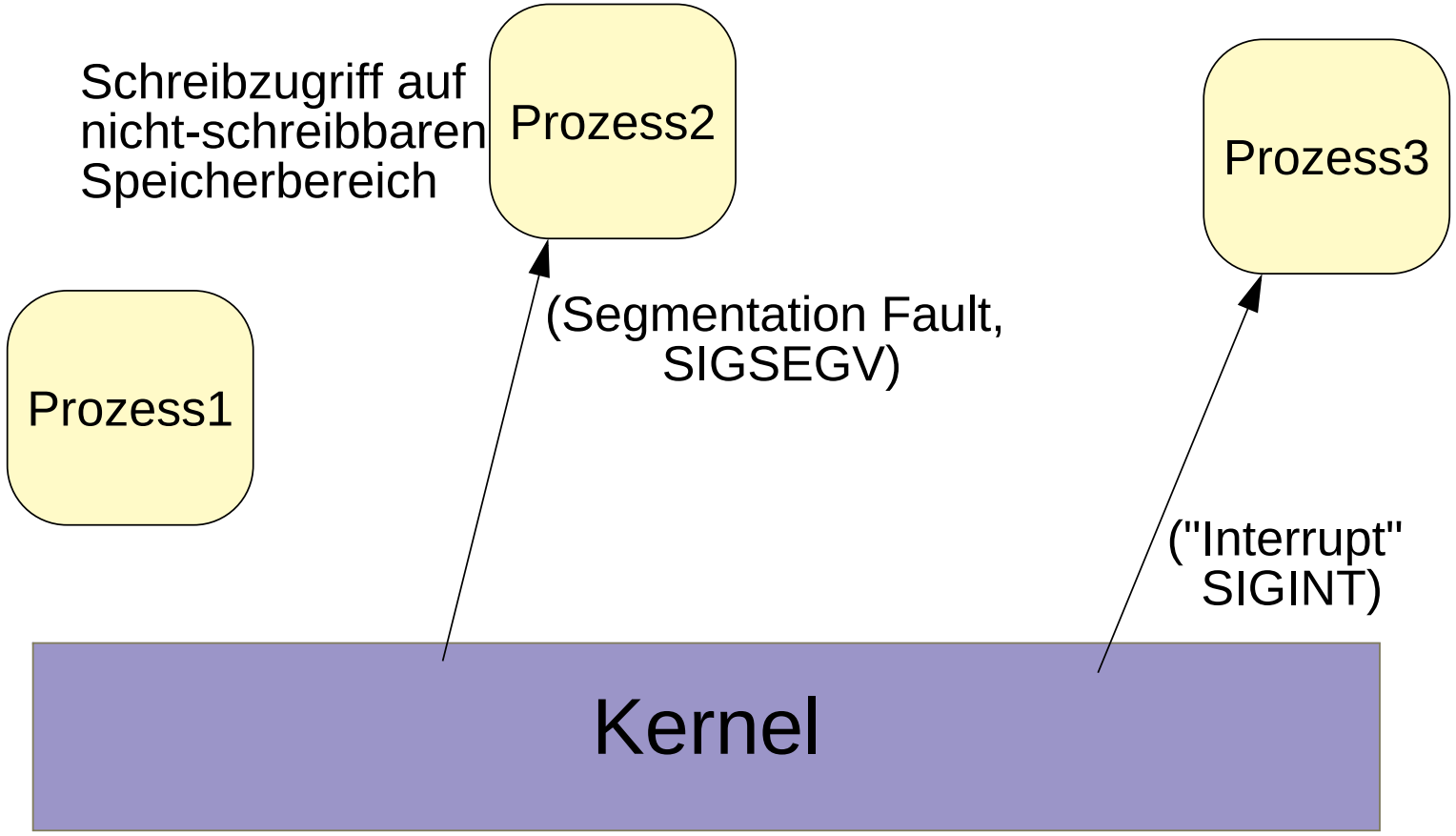
U3-1 Signale

1 Kommunikation zwischen Prozessen



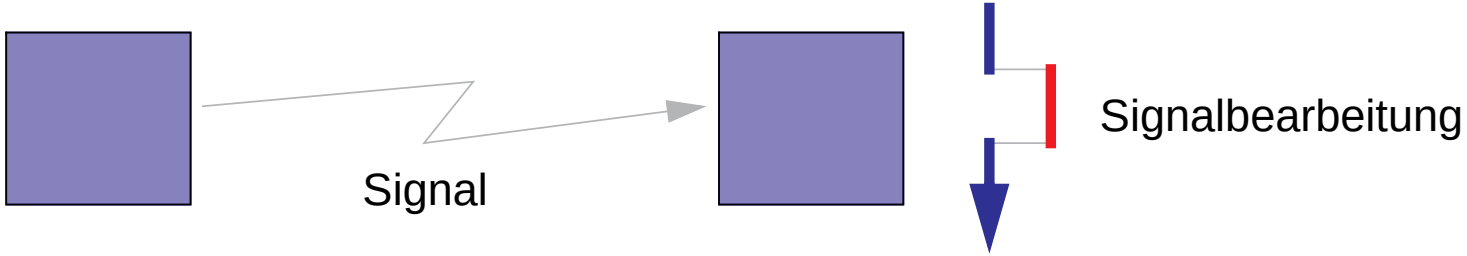
2 Signalisierung des Systemkerns

- 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()`**
- SIGFPE (abort): Floating Point Exception (Division durch 0, Überlauf, ...)
- 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
- SIGCHLD (ignore): Status eines Kindprozesses hat sich geändert
- SIGTERM (exit): Standardsignal von **`kill(1)`**

U3-2 UNIX-API zur Signalbehandlung

- ANSI-C definiert die **signal(2)**-Funktion zum Einrichten von Signalbehandlungen
  - Problem: ungenaue Spezifikation
  - **signal sollte in neuen Programmen nicht mehr benutzt werden**
- Wir verwenden folgende Funktionen der Systemschnittstelle
  - sigaction
  - kill
  - und weitere (siehe nächste Übung)

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`

- **`sigaction`-Struktur**

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

- 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`

- **`sigaction`-Struktur**

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

- während einer Signalbehandlung ist das auslösende Signal blockiert
  - es wird pro Signalnummer maximal ein Ereignis zwischengespeichert
  - mit **`sa_mask`** kann man *weitere* Signale blockieren
- Anmerkung: Blockieren von Signalen nicht gleich Ignorieren von Signalen; ignorierte Signale werden *verworfen*, blockierte nur *verzögert*

1 Signalbehandlung installieren: `sa_mask`

- **`sigaction`-Struktur**

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

- 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

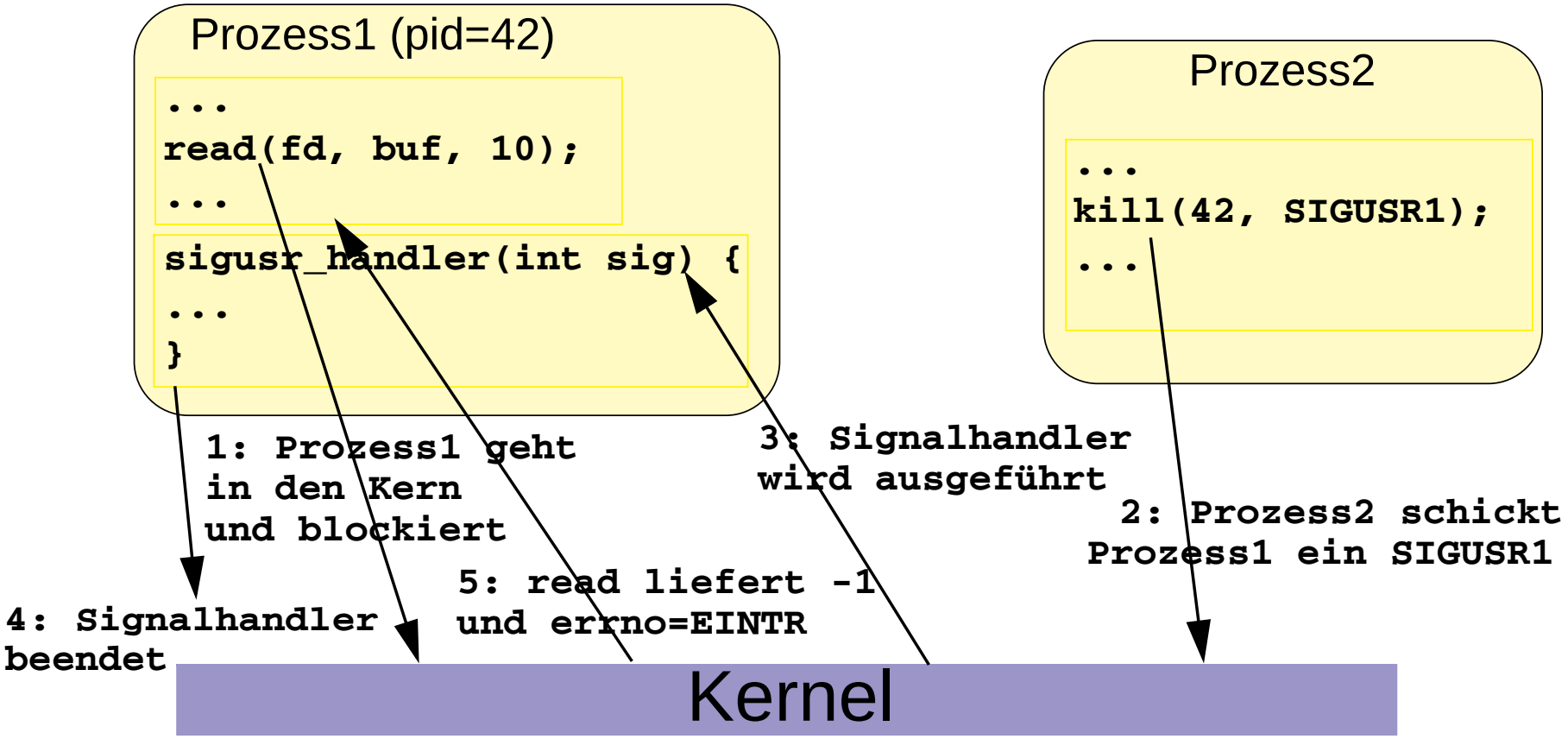
1 Signalbehandlung installieren: `sa_flags`

- **`sigaction`-Struktur**

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

- Beeinflussung des Verhaltens bei Signalempfang durch **`sa_flags`**
  - **`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 (siehe Folie 16)
- weitere Flags siehe **sigaction(2)**

2 Unterbrechung blockierender Systemaufrufe



- Signale können blockierende Systemaufrufe (z.B. **`accept`**) unterbrechen
  - Systemaufrufe setzen **`errno`** auf **`EINTR`** und kehren mit Fehler zurück
  - Verhalten kann mit **`SA_RESTART`** geändert werden

3 Signalhandler installieren: Beispiel

```
#include <signal.h>

void handler(int sig) { ... }

int main() {
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_RESTART;
    action.sa_handler = handler;
    sigaction(SIGPIPE, &action, NULL);
```

4 Signal zustellen

- Systemaufruf **`kill(2)`**

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

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

U3-3 Zombies einsammeln mit Hilfe von Signalen

- Stirbt ein Kindprozess, so erhält der Vater das Signal SIGCHLD vom Kernel
  - damit ist sofortiges Aufsammeln von Zombieprozessen möglich
- Variante 1: Aufruf von `waitpid(2)` im Signalhandler

```
void ghostbuster(int sig) {
    int status;
    int retVal;
...
    retVal = waitpid(-1, &status, WNOHANG);
...
```

- Variante 2: Signalhandler für SIGCHLD auf `SIG_DFL` setzen und in den sa_flags den Wert `SA_NOCLDWAIT` setzen
- Variante 3: Signalhandler für SIGCHLD auf `SIG_IGN` setzen

U3-4 Aufgabe 2: sister

- Einfacher HTTP-Webserver zum Ausliefern statischer HTML-Seiten innerhalb eines Verzeichnisbaums (*www-Pfad*)
- Abarbeitung der Anfragen erfolgt in eigenem Prozess (`fork(2)`)
- Modularer Aufbau (vgl. SP1#WS11 A/I 7)
  - Wiederverwendung einzelner Module in der Aufgabe 5: mother

1 Exkurs: Modulare Softwareentwicklung in C

- Wiederholung: Modul besteht aus:
  - öffentlicher Schnittstelle (Header-Datei)
  - konkreter Implementierung dieser Schnittstelle (C-Datei)
- Durch diese Trennung ist es möglich die Implementierung auszutauschen, ohne die Schnittstelle zu verändern
  - ◆ Module, die die öffentliche Schnittstelle verwenden, müssen nicht angepasst werden wenn die konkrete Implementierung geändert wird.

2 Hauptmodul (sister.c)

- Implementiert die `main()`-Funktion
  - Initialisierung des Verbindungs- und `getargs`-Moduls
  - Vorbereiten der Interprozesskommunikation
  - Annehmen von Verbindungen
  - Angenommene Verbindungen werden an das Verbindungsmodul übergeben

3 Verbindungsmodul (connection-fork.c)

- Implementiert die Schnittstelle aus der Header-Datei `connection.h`
  - Initialisierung des Anfragemoduls
  - Erstellt Kind-Prozess zur Abarbeitung der Anfrage
    - Anmerkung: Entstandene Zombie-Prozesse müssen beseitigt werden!
  - Weitergabe der Verbindung an das Anfragemodul

4 Anfragemodul (request-http.c)

- Implementiert die Schnittstelle aus der Header-Datei **`request.h`**
  - Einlesen und Auswerten der Anfrage(-zeile)
  - Suchen der angeforderten Datei im *www-Pfad*
    - Anmerkung: Anfragen auf Dateien jenseits des *www-Pfades* stellen ein Sicherheitsrisiko dar. Sie müssen erkannt und abgelehnt werden!
  - Ausliefern der Datei

5 Hilfsmodule (i4httools, getargs)

- **`i4httools`**: Hilfsfunktionen zum Implementieren eines HTTP-Servers
- **`getargs`**: Aus Aufgabe 1 (simail) bekannte Schnittstelle zum Parsen der Befehlszeilenargumente