

# Übungen zu Systemprogrammierung 2 (SP2)

## Ü2 – IPC mit Sockets, Signale

Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

WS 2017 – 13. bis 17. November 2017

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



## Agenda

- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: *sister*
- 2.7 Gelerntes anwenden



## Agenda

- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: *sister*
- 2.7 Gelerntes anwenden



## IPC-Schnittstelle: Server

- **Ausgangssituation:** Socket wurde bereits erstellt (`socket(2)`)
- Nach seiner Erzeugung muss der Socket zunächst an eine Adresse *gebunden* werden, bevor er verwendet werden kann
- `bind(2)` stellt eine generische Schnittstelle zum Binden von Sockets in unterschiedlichen Domänen bereit:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- `sockfd`: Socket-Deskriptor
- `addr`: protokollspezifische Adresse
  - Socket-Interface (`<sys/socket.h>`) ist zunächst protokollunabhängig:

```
struct sockaddr {  
    sa_family_t sa_family; // Adressfamilie  
    char sa_data[14];      // Platzhalter für Adresse  
};
```

- „Klassenvererbung für Arme“; i. d. R. Cast notwendig
- `addrlen`: Länge der konkret übergebenen Struktur in Bytes



- Name durch IPv4-Adresse und Port-Nummer definiert:

```
struct sockaddr_in {
    sa_family_t    sin_family; // = AF_INET
    in_port_t      sin_port;    // Port
    struct in_addr sin_addr;    // Internet-Adresse
};
```

- `sin_port`: Port-Nummer
  - `sin_addr`: IPv4-Adresse
    - `INADDR_ANY`: wenn Socket auf allen lokalen Adressen (z. B. allen Netzwerkschnittstellen) Verbindungen akzeptieren soll
- `sin_port` und `sin_addr` müssen in Netzwerk-Byteorder vorliegen!
  - Umwandlung mittels `htons(3)`, `htonl(3)`: konvertiert Datenwort von Host-spezifischer Byteorder in Netzwerk-Byteorder – bzw. zurück:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```



- Name durch IPv6-Adresse und Port-Nummer definiert:

```
struct sockaddr_in6 {
    sa_family_t    sin6_family; // = AF_INET6
    in_port_t      sin6_port;    // Port-Nummer
    uint32_t       sin6_flowinfo; // = 0
    struct in6_addr sin6_addr;    // IPv6-Adresse
    uint32_t       sin6_scope_id; // = 0
};
```

```
struct in6_addr {
    unsigned char  s6_addr[16];
};
```

- `sin6_port`: Port-Nummer
  - `sin6_addr`: IPv6-Adresse
    - `in6addr_any`: auf allen lokalen Adressen Verbindungen akzeptieren
- `sin6_port` muss in Netzwerk-Byteorder vorliegen (`htons(3)`)
- `in6_addr`-Struktur ist byteweise definiert, deswegen keine Konvertierung nötig



## IPC-Schnittstelle: Server

- Verbindungsannahme vorbereiten mit `listen(2)`:

```
int listen(int sockfd, int backlog);
```

- `backlog`: (Unverbindliche) Größe der Warteschlange, in der eingehende Verbindungswünsche zwischengepuffert werden
    - Bei voller Warteschlange werden Verbindungsanfragen zurückgewiesen
    - Maximal mögliche Größe: `SOMAXCONN`



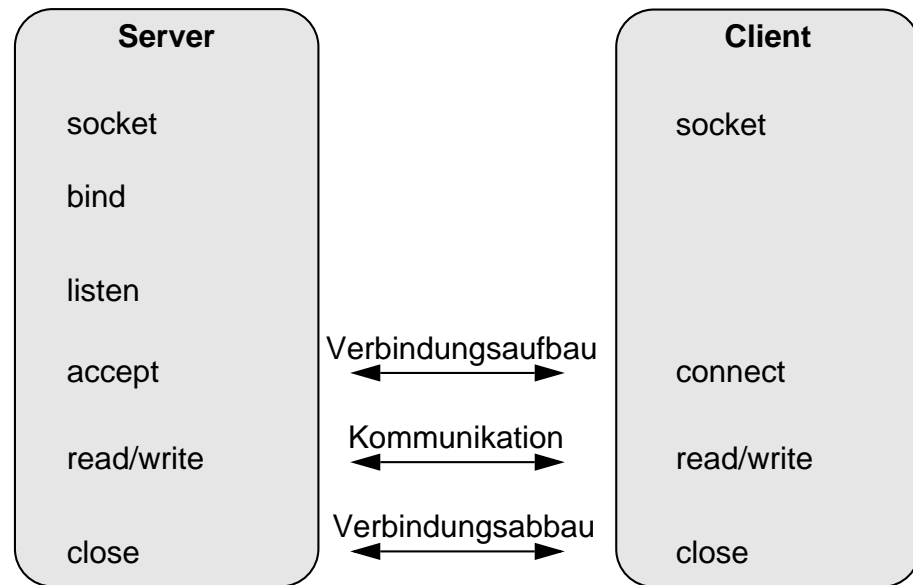
## IPC-Schnittstelle: Server

- Verbindung annehmen mit `accept(2)`:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- `addr`, `addrlen`: Ausgabeparameter zum Ermitteln der Adresse des Clients
    - Bei Desinteresse zweimal `NULL` übergeben
- Entnimmt die vorderste Verbindungsanfrage aus der Warteschlange
  - Blockiert bei leerer Warteschlange
- Erzeugt einen neuen Socket und liefert ihn als Rückgabewert
  - Kommunikation mit dem Client über diesen neuen Socket
  - Annahme weiterer Verbindungen über den ursprünglichen Socket





! Fehlerabfragen nicht vergessen

```

int listenSock = socket(AF_INET6, SOCK_STREAM, 0);

struct sockaddr_in6 name = {
    .sin6_family = AF_INET6,
    .sin6_port = htons(1112),
    .sin6_addr = in6addr_any,
};
bind(listenSock, (struct sockaddr *) &name, sizeof(name));

listen(listenSock, SOMAXCONN);

for (;;) {
    int clientSock = accept(listenSock, NULL, NULL);
    char buf[1024];
    ssize_t n;
    while ((n = read(clientSock, buf, sizeof(buf))) > 0) {
        write(clientSock, buf, n);
    }
    close(clientSock);
}
  
```

```

for (;;) {
    int clientSock = accept(listenSock, NULL, NULL);
    char buf[1024];
    ssize_t n;
    while ((n = read(clientSock, buf, sizeof(buf))) > 0) {
        write(clientSock, buf, n);
    }
    close(clientSock);
}
  
```

- Limitierungen:
  - Neue Verbindung kann erst nach vollständiger Abarbeitung der vorherigen Anfrage angenommen werden
  - Monopolisierung des Dienstes möglich (*Denial of Service*)!
- Mögliche Ansätze zur Abhilfe:
  1. Mehrere Prozesse
    - Anfrage wird durch Kindprozess bearbeitet
  2. Mehrere Threads
    - Anfrage wird durch einen Thread im gleichen Prozess bearbeitet

- Nach Beendigung des Server-Prozesses erlaubt das Betriebssystem kein sofortiges `bind(2)` an den selben Port
  - Erst nach Timeout erneut möglich
  - Grund: es könnten sich noch Datenpakete für den alten Prozess auf der Leitung befinden
- Testen und Debuggen eines Server-Programms dadurch stark erschwert
- Lösungsmöglichkeiten:
  1. Bei jedem Start einen anderen Port verwenden – doof!
  2. Sofortige Wiederverwendung des Ports forcieren:
 

```

int sock = socket(...);
...
int flag = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &flag, sizeof(flag));
...
bind(sock, ...);
          
```

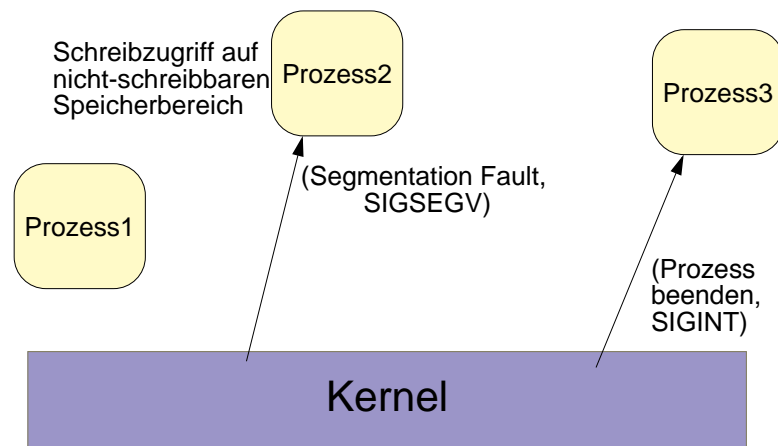
- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: *sister*
- 2.7 Gelerntes anwenden



- Essenzielles Betriebssystemkonzept: synchrone/asynchrone Programmunterbrechungen (*Traps* bzw. *Interrupts*)
  - Zweck: Signalisierung von Ereignissen
  - Abwicklung zwischen Hardware und Betriebssystem
  - Transparent für die Anwendung
- **UNIX-Signale:** Nachbildung des Konzepts auf Anwendungsebene
  - Abwicklung zwischen Betriebssystem und Anwendung
  - Unabhängig von der Hardware



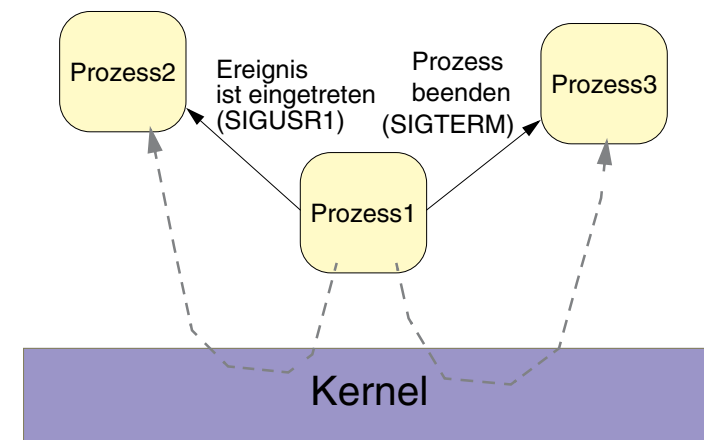
- **Anwendungsfall 1:** Signalisierungen durch den Betriebssystemkern



- Synchrone Signale: unmittelbar durch Aktivität des Prozesses ausgelöst
- Asynchrone Signale: „von außen“ ausgelöst



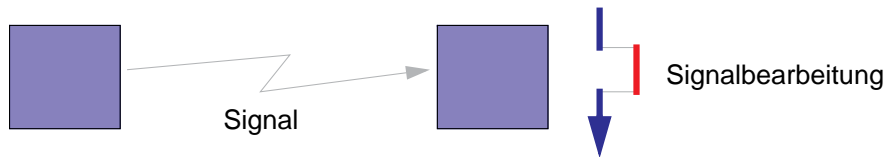
- **Anwendungsfall 2:** primitive „Kommunikation“ zwischen Prozessen



- Asynchron zum eigentlichen Programmablauf



- *Ign*
  - Ignorieren des Signals
- *Core*
  - Erzeugen eines Core-Dumps (Speicherabbild + Registerkontext) und Beenden des Prozesses
- *Term*
  - Beenden des Prozesses, ohne einen Core-Dump zu erzeugen
  - Standardreaktion für die meisten Signale
- Signal-Behandlungsfunktion
  - Aufruf einer vorher festgelegten Funktion, danach Fortsetzen des Prozesses:



<b>SIGINT</b>	<i>Term</i>	Beenden durchs Terminal (Ctrl-C)
<b>SIGABRT</b>	<i>Core</i>	Abort-Signal; entsteht z. B. durch Aufruf von <code>abort(3)</code>
<b>SIGFPE</b>	<i>Core</i>	Floating-Point Exception (Division durch 0, Überlauf, ...)
<b>SIGKILL</b>	<i>Term</i>	„Tötet“ den Prozess; nicht abfangbar
<b>SIGSEGV</b>	<i>Core</i>	Segmentation Violation; illegaler Speicherzugriff
<b>SIGPIPE</b>	<i>Term</i>	Schreiben auf Pipe oder Socket, nachdem die Gegenseite geschlossen wurde
<b>SIGTERM</b>	<i>Term</i>	Standardsignal von <code>kill(1)</code>
<b>SIGCHLD</b>	<i>Ign</i>	Status eines Kindprozesses hat sich geändert

- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: *sister*
- 2.7 Gelerntes anwenden

- Prototyp:
 

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

  - `signum`: Signalnummer
  - `act`: Neue Behandlung für dieses Signal
  - `oldact`: Bisherige Behandlung dieses Signals (Ausgabeparameter)
- Die eingerichtete Behandlung bleibt so lange 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
};
```

■ `sigaction`-Struktur:

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

■ Über `sa_handler` kann die Signalbehandlung eingestellt werden:

- `SIG_IGN`: Signal ignorieren
- `SIG_DFL`: Standard-Signalbehandlung einstellen
- *Funktionsadresse*: Funktion wird in der Signalbehandlung aufgerufen

■ `sigaction`-Struktur:

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

■ Trifft während der Signalbehandlung dasselbe Signal erneut ein, wird dieses bis zum Ende der Behandlung verzögert (*blockiert*)

- Maximal ein Ereignis wird zwischengespeichert
- Mit `sa_mask` kann man **weitere** Signale blockieren

## ■ Hilfsfunktionen zum Auslesen und Modifizieren einer Signal-Maske:

- `sigaddset(3)`: Bestimmtes Signal zur Maske hinzufügen
- `sigdelset(3)`: Bestimmtes Signal aus Maske entfernen
- `sigemptyset(3)`: Alle Signale aus Maske entfernen
- `sigfillset(3)`: Alle Signale in Maske aufnehmen
- `sigismember(3)`: Abfrage, ob bestimmtes Signal in Maske enthalten ist

■ `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` (0 oder Veroderung von Flag-Konstanten):

- `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 nächste Folie)

■ Weitere Flags siehe `sigaction(2)`

## ■ Signalbehandlung muss im Benutzerkontext durchgeführt werden

## ? Was geschieht, wenn ein Prozess ein Signal erhält, während er sich in einem Systemaufruf befindet?

## ■ Nicht-blockierender Systemaufruf:

- Signalbehandlung wird durchgeführt, sobald der Kontrollfluss aus dem Kern zurückkehrt

## ■ Blockierender Systemaufruf:

- **Problem**: Die Blockade kann beliebig lang dauern, z. B. beim Warten auf eingehende Verbindungen mit `accept(2)`
  - Die Signalbehandlung indefinit hinauszuzögern, ist keine gute Idee
- **Lösung**: Systemaufruf wird abgebrochen und kehrt mit `errno = EINTR` zurück, Signal wird sofort behandelt
- **Vereinfachung**: Setzt man das Flag `SA_RESTART`, kehrt der Systemaufruf nicht mit Fehler zurück, sondern wird nach der Signalbehandlung automatisch wiederholt



## ■ Systemaufruf kill(2):

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

## ■ Shell-Kommando kill(1):

- Sendet ein Signal an einen bestimmten Prozess
- z. B. `user@host:~$ kill -USR1 <pid>`

## ■ Shell-Kommando pkill(1):

- Sendet ein Signal an alle Prozesse, die ein bestimmtes Programm ausführen
- z. B. `user@host:~$ pkill -USR1 <programmname>`



- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: sister
- 2.7 Gelerntes anwenden



# 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
  - Aufruf in Schleife notwendig – während der Signalbehandlung könnten weitere Kindprozesse sterben
- **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



# Agenda

- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: sister
- 2.7 Gelerntes anwenden



## Dynamische Makros

- **\$@** Name des Targets (hier: `test`)

```
test: test.c
    $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $@ test.c
```

- **\$\*** Basisname des Targets (ohne Dateierdung, hier: `test`)

```
test.o: test.c test.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $*.c
```

- **\$<** Name der ersten Abhängigkeit (hier: `test.c`)

```
test.o: test.c test.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

- **\$^** Namen aller Abhängigkeiten (hier: `test.o func.o`)

- Achtung: GNU-Erweiterung, nicht SUSv4-konform!

```
test: test.o func.o
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^
```



## Pattern-Regeln

- Allgemeine Regel zur Erzeugung einer Datei mit einer bestimmten Endung aus einer gleichnamigen Datei mit einer anderen Endung

- Beispiel: Erzeugung von `.o`-Dateien aus `.c`-Dateien

```
%.o: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

- Regeln ohne Kommandos können Abhängigkeiten überschreiben  
`test.o: test.c test.h func.h`

- Die Pattern-Regel wird weiterhin zur Erzeugung herangezogen

- Explizite Regeln überschreiben die Pattern-Regeln

```
test.o: test.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -DXYZ -c $<
```



## Agenda

- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: **sister**
- 2.7 Gelerntes anwenden



## 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#SS16 A/II 7)
  - Wiederverwendung einzelner Module in Aufgabe 5: `mother`



- **Wiederholung:** Ein 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 deren konkrete Implementierung geändert wird



## Aufgabe 2: sister

### Anfragemodul (request-http.c)

- Implementiert die Schnittstelle aus dem Header `request.h`:
  - Einlesen und Auswerten der Anfragezeile
  - Suchen der angeforderten Datei im WWW-Pfad
    - ! **Vorsicht:** Anfragen auf Dateien jenseits des WWW-Pfades stellen ein Sicherheitsrisiko dar. Sie müssen erkannt und abgelehnt werden!
  - Ausliefern der Datei

### Hilfsmodule (cmdline, i4httpools)

- `cmdline`: Schnittstelle zum Parsen der Befehlszeilenargumente
- `i4httpools`: Hilfsfunktionen zum Implementieren eines HTTP-Servers



## Aufgabe 2: sister

### Hauptmodul (sister.c)

- Implementiert die `main()`-Funktion:
  - Initialisierung des Verbindungs- und `cmdline`-Moduls
  - Vorbereiten der Interprozesskommunikation
  - Annehmen von Verbindungen
  - Übergabe angenommener Verbindungen an das Verbindungsmodul

### Verbindungsmodul (connection-fork.c)

- Implementiert die Schnittstelle aus dem Header `connection.h`:
  - Initialisierung des Anfragemoduls
  - Erstellen eines Kindprozesses zur Abarbeitung der Anfrage
    - Anmerkung: Entstandene Zombie-Prozesse müssen beseitigt werden!
  - Weitergabe der Verbindung an das Anfragemodul



## Agenda

- 2.1 IPC-Schnittstelle: Server
- 2.2 UNIX-Signale
- 2.3 Signal-API von UNIX
- 2.4 Einsammeln von Zombies
- 2.5 Makefiles – Teil 3
- 2.6 Aufgabe 2: sister
- 2.7 Gelerntes anwenden



### „Aufgabenstellung“

- Programm schreiben, welches durch Ctrl-C nicht beendet werden kann

