# Übungen zu Systemprogrammierung 2

Ü2 – IPC mit Sockets, Signale

Wintersemester 2025/26

Jürgen Kleinöder, Thomas Preisner, Tobias Häberlein, Ole Wiedemann

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







- 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



### 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 (3p))
- Nach seiner Erzeugung muss der Socket zunächst an eine Adresse gebunden werden, bevor er verwendet werden kann
- bind(3p) 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[]; // Adresse (variable Länge)
};
```

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

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

- 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(3p): 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:

- sin6\_port: Port-Nummer
- sin6\_addr: IPv6-Adresse
  - in6addr\_any: auf allen lokalen Adressen Verbindungen akzeptieren
- sin6\_port muss in Netzwerk-Byteorder vorliegen (htons(3p))
- in6\_addr-Struktur ist byteweise definiert, deswegen keine Konvertierung nötig

#### **IPC-Schnittstelle: Server**



Verbindungsannahme vorbereiten mit listen(3p):

```
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**



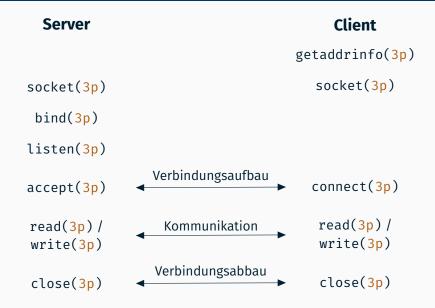
■ Verbindung annehmen mit accept(3p):

```
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

# **TCP/IP-Sockets: Zusammenfassung**





#### **Nicht vergessen**

Fehlerabfragen

```
int listenSock = socket(AF_INET6, SOCK_STREAM, 0);
// AF INET6 akzeptiert auch automatisch IPv4-Verbindungen,
// falls vom OS unterstützt (z.B. Linux)
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);
while (1) {
  int clientSock = accept(listenSock, NULL, NULL);
 // handle connection
  close(clientSock);
```

```
while (1) {
  int clientSock = accept(listenSock, NULL, NULL);
  // handle connection
  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

#### "Wiederverwenden" von Ports



- Nach Beendigung des Server-Prozesses erlaubt das Betriebssystem kein sofortiges bind(3p) an den selben Port
  - Erst nach Timeout erneut möglich
- 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));
// Fehlerbehandlung!
...
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

### **UNIX-Signale**

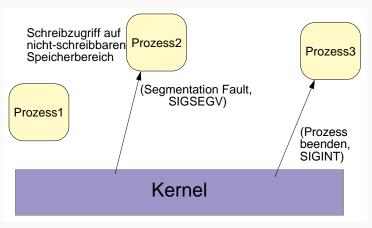


- Essenzielles Betriebssystemkonzept: synchrone/asynchrone
   Programmunterbrechungen (Traps bzw. Interrupts)
  - Zweck: Signalisierung von Ereignissen
  - Abwicklung zwischen Hardware und Betriebssystem
  - Transparent f
    ür die Anwendung
- UNIX-Signale: Nachbildung des Konzepts auf Anwendungsebene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

### **UNIX-Signale**



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

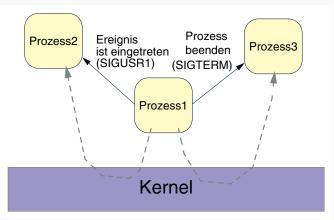


- Synchrone Signale: unmittelbar durch Aktivität des Prozesses ausgelöst
- Asynchrone Signale: "von außen" ausgelöst

### **UNIX-Signale**



■ **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 f
    ür die meisten Signale
- Signal-Behandlungsfunktion
  - Aufruf einer vorher festgelegten Funktion, danach Fortsetzen des Prozesses:



## Gängige Signale (mit Standardverhalten)



#### Standardverhalten: Term

SIGINT Beenden durchs Terminal (Ctrl-C)

SIGTERM Standardsignal von kill(1)

SIGKILL "Tötet" den Prozess; nicht abfangbar

SIGPIPE Schreiben auf Pipe oder Socket, nachdem die Gegenseite

geschlossen wurde

### Standardverhalten: Core Dump

SIGSEGV Segmentation Violation; illegaler Speicherzugriff

SIGABRT Abort-Signal; entsteht z. B. durch Aufruf von abort (3p)
SIGFPE Floating-Point Exception (Division durch 0, Überlauf, ...)

#### Standardverhalten: Ignore

SIGCHLD 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:

- signum: Signalnummer
- act: Neue Behandlung für dieses Signal
- oldact: Bisherige Behandlung dieses Signals (Ausgabeparameter)
- Behandlung bleibt aktiv bis 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

- **Achtung:** SIG\_IGN ≠ Ignorieren als Standardverhalten eines Signals
  - Explizites Setzen von SIG\_IGN für SIGCHLD räumt automatisch Zombie-Prozesse auf (Siehe S. 26)
  - Das Standardverhalten (SIG\_DFL) ignoriert das Signal zwar auch, räumt aber keine Zombie-Prozesse auf

■ 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(3p): Bestimmtes Signal zur Maske hinzufügen
sigdelset(3p): Bestimmtes Signal aus Maske entfernen
sigemptyset(3p): Alle Signale aus Maske entfernen
sigfillset(3p): Alle Signale in Maske aufnehmen
sigismember(3p): 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 NOCLDWAIT und SA NOCLDSTOP: Siehe S. 26
  - SA\_RESTART: durch das Signal unterbrochene Systemaufrufe werden automatisch neu aufgesetzt (siehe n\u00e4chste Folie)
- Weitere Flags siehe sigaction(3p)

- 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(3p)
      - 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(3p):

```
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(3p) 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
  - Das Flag SA\_NOCLDSTOP bewirkt, dass SIGCHLD nur gesendet wird, wenn ein Kindprozess terminiert
  - Signal wird nicht gesendet, wenn Prozess nur gestoppt oder wiederaufgenommen wird
- Variante 3: Signalhandler für SIGCHLD auf SIG\_IGN setzen



- 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



■ \$a Name des Targets (hier: test)

```
test: test.c

$(CC) $(CFLAGS) $(LDFLAGS) -o $0 test.c
```

\$< Name der ersten Abhängigkeit (hier: test.c)</p>

```
test.o: test.c test.h
$(CC) $(CFLAGS) -c $<</pre>
```

- \$^ Namen aller Abhängigkeiten (hier: test.o func.o)
  - Achtung: GNU-Erweiterung, nicht SUSv4-konform!
  - Darf in SP trotzdem verwendet werden

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



- 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) $(CFLAGS) -c $<
```

Regeln ohne Kommandos können Abhängigkeiten ergänzen

```
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 test.h func.h
$(CC) $(CFLAGS) $(LDFLAGS) -DXYZ -c $
```

## Für zukünfige Übungen



#### Hinweis

Die vorgestellten automatischen Variablen und Pattern-Regeln müssen ab der kommenden Übung verwendet werden.

- Warum?
  - Vermeidung von doppeltem Code
  - Pattern-Regeln sind m\u00e4chtiges Werkzeug und werden in vielen echten Projekten verwendet



- 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(3p))
- Modularer Aufbau (vgl. SP1#SS25 A/II 7 Programmstruktur und Module)
  - 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

#### **Achtung**

Es dürfen keine Annahmen über eine Implementierung getroffen werden, die über die Beschreibung in der Header-Datei hinausgehen. Die Verwendung eines Moduls muss ohne Kenntnis der eigentlichen Implementierung möglich sein.

### 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

### 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, i4httools), vorgegeben

- cmdline: Schnittstelle zum Parsen der Befehlszeilenargumente
- i4httools: Hilfsfunktionen zum Implementieren eines HTTP-Servers



- 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

#### **Aktive Mitarbeit!**



### "Aufgabenstellung"

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