

U2 2. Übung

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

SP -

Systemprogrammierung — Übungen

© Jürgen Kleinröder, Michael Stilkerich, Jens schedel, Christoph Erhardt • Universität Erlangen-Nürnberg • Informatik 4, 2012

U02.fm 2012-05-08 14,11

U2.1

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Verbindungsannahme durch den Server

- Ausgangssituation: Socket wurde bereits erstellt und an einen Namen gebunden
 - ◆ ***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

SP -

Systemprogrammierung — Übungen

© Jürgen Kleinröder, Michael Stilkerich, Jens schedel, Christoph Erhardt • Universität Erlangen-Nürnberg • Informatik 4, 2012

U02.fm 2012-05-08 14,11

U2.2

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Verbindungsannahme durch den 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, (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 dem Echo-Server

```

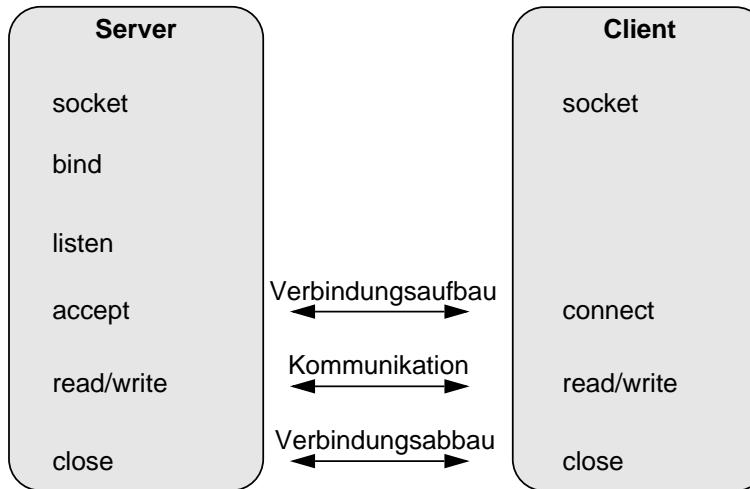
while ( (in_fd = accept(fd, NULL, NULL)) != -1 ) {
    char buf[1024];
    int n;
    while ( (n = read(in_fd, buf, sizeof(buf))) > 0 ) {
        write(in_fd, buf, n);
    }
    close(in_fd);
}

```

- 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

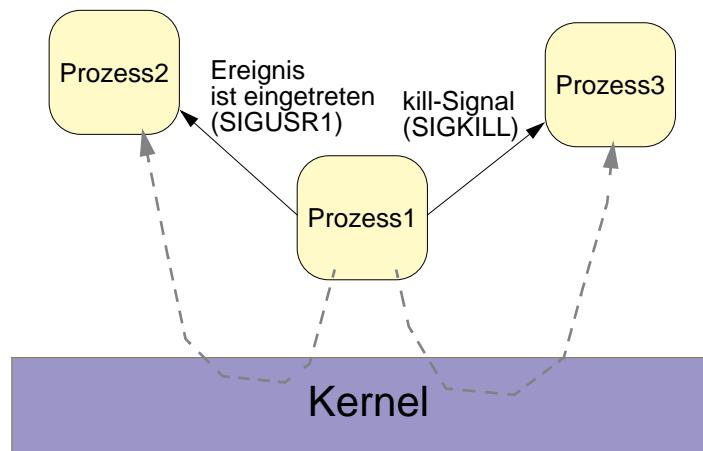
3 TCP-Sockets: Zusammenfassung



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

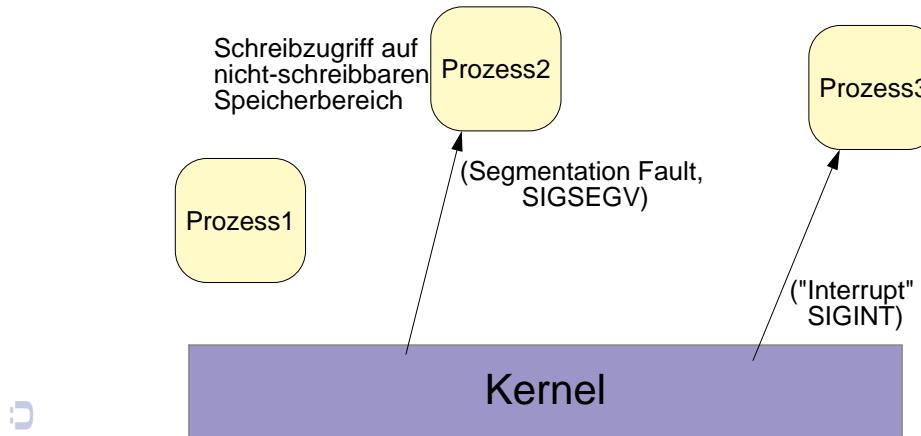
U2-1 Signale

1 "Kommunikation" zwischen Prozessen



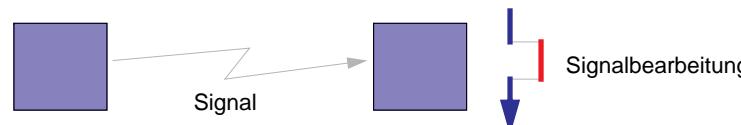
2 Signalisierung des Systemkerns

- synchrone Signale: unmittelbar 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)`

- 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
};
```

- Die 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 automatisch 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`

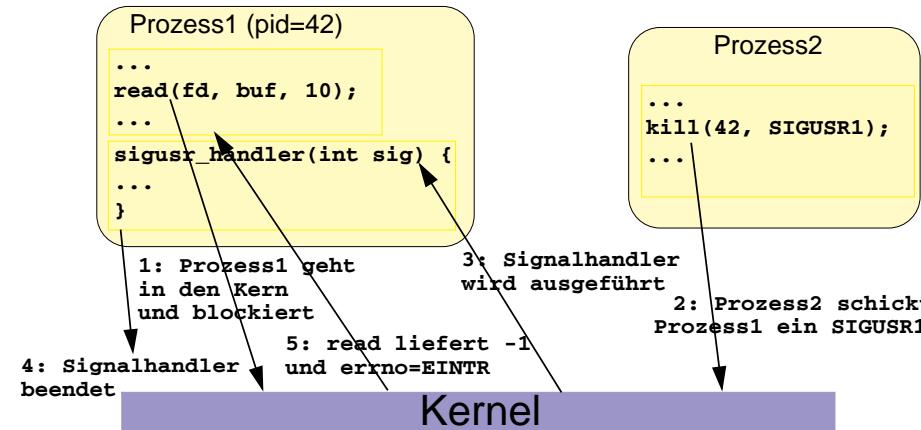
- `SIGCHLD` wird nur zugestellt, wenn ein Kindprozess terminiert, nicht wenn er gestoppt wird

◆ `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
- ◆ Dieses Verhalten kann mit `SA_RESTART` verhindert 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>`)

U2-3 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

U2-4 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: 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

2 Hauptmodul (sister.c)

- Implementiert die `main()`-Funktion
 - ◆ Initialisierung des Verbindungs- und `cmdline`-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 (cmdline, i4httools)

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

6 Statische Bibliotheken: Kurzübersicht

- Statische Bibliothek **libsister.a**:
 - ◆ Archiv, in dem mehrere kompilierte Module (.o-Dateien) zusammengefasst sind
 - ◆ Kann gemeinsam mit anderen Modulen zu einem ausführbaren Programm zusammengebunden werden
 - ◆ Der Linker wählt aus einer Bibliothek nur diejenigen Module aus, die Funktionen bzw. globale Variablen enthalten, welche von anderen Modulen referenziert werden
- Benötigte Linker-Flags:
 - ◆ **-L.** nimmt das Verzeichnis . in die Liste der Suchpfade auf (standardmäßig immer enthalten: **/usr/local/lib, /usr/lib**)
 - ◆ **-lsister** bindet die Bibliothek **libsister.a** ein, die in einem der Suchpfade liegen muss
- Mehr Details zu statischen und dynamischen Bibliotheken in der Übung zu Aufgabe 4

