Klausur zu Systemnahe Software
03. Juli 2004 (SS 2004)

Bearbeitungszeit: 120 Minuten

NICHT MIT BLEISTIFT SCHREIBEN!

Name:	
Vorname:	
Matrikelnummer:	

Nr	Max	Bewertung	
1	**6**	xxxxx	
2	**12**	xxxxx	
3	**10**	xxxxx	
4	**10**	xxxxx	
(a)	6		xxxxx
(b)	4		xxxxx
5	**10**	xxxxx	
6	**12**	xxxxx	

Nr	Max	Bewertung	
7	**8**	xxxxx	
(a)	6		xxxxx
(b)	2		xxxxx
8	**14**	xxxxx	
(a)	6		xxxxx
(b)	6		xxxxx
(c)	2		xxxxx
9	**8**	xxxxx	
Summe	**90**		

Bitte benutzen Sie für die Lösungen den freigelassenen Platz nach der jeweiligen Angabe oder die Rückseite unter Angabe der Aufgabe. Nennen Sie möglichst alle Annahmen, die Sie für die Lösung einer Aufgabe treffen! Bei den vorgegebenen Programmen wurden die `include`-Zeilen der Übersicht halber weggelassen.

Viel Erfolg!

Aufgabe 1 **(6 Punkte)**

Gegeben sei das folgende Programm:

```
int main ()
{
        int count = 0;
        while (1) {
                int pid = fork ();
                if (pid < 0)
                        break;
                if (pid) {
                        count++;
                        continue;
                } else {
                        exit (count % 100);
                }
        }
        printf ("%d\n", count);
        return 0;
}
```

Beschreiben Sie genau, was beim Ausführen dieses Programms passiert. Terminiert die `while`-Schleife? Falls ja, was für einen Wert hat dann `count` (ungefähr)? Wieviele Einträge in der Prozeßtabelle bleiben nach dem Ende des Programms ungefähr übrig und was passiert ggf. mit Ihnen? Begründen Sie Ihre Antworten!

```
Es werden Kindprozesse in einer Endlosschleife erzeugt,
die sofort wieder terminieren. Allerdings wartet der
Vater nicht mit wait auf die Kinder, deshalb belegt
solange der Vater lebt jedes der Kinder einen Eintrag
in der Prozesstabelle (Zombies). Sobald diese Eintraege
ausgehen terminiert die While-Schleife, weil fork
einen Fehler liefert. Je nach Systemumgebung passiert
dies irgendwo zwischen 100 und 10000 Prozessen (bis
100000 wird bei geeigneter Begr"undung akzeptiert).
Nach dem Tod des Vaters erbt der init-Prozess die
Kinder, dieser beseitigt sie dann, indem er f"ur jeden
Prozess wait() aufruft. Es bleiben also keine Eintraege
in der Prozesstabelle uebrig.
```

Aufgabe 2 **(12 Punkte)**

Gegeben seien die folgenden zwei Shell-Kommandos:

```
bzcat somefile.tar.bz2 | tar -tvzf - 2>tmpfile
<tmpfile wc -l
```

Leider wird von diesen beiden Kommandos unnötiger Weise eine Datei tmpfile (für die Fehlerausgabe von tar) angelegt (aus der wc später wieder liest). Schreiben Sie ein C-Programm, das das selbe erledigt, wie diese beiden Kommando, allerdings **ohne** dazu eine Datei anzulegen.

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>

void die (char * s)
{
        perror (s);
        exit (1);
}

int main ()
{
    int p1[2], p2[2];
    int pid1, pid2;
    if (pipe (p1) < 0) die ("pipe");
    pid1 = fork ();
    if (pid1 < 0) die ("fork");
    if (pid1 > 0) { /* Vater fuehrt bzcat aus.  */
        close (p1[0]);
        if (dup2 (p1[1], 1) < 0) die ("dup2");
        close (p1[1]);
        execlp ("bzcat", "bzcat", "somefile.tar.bz2", 0);
        die ("execlp");
    }
    /* 1. Kind ... */
    close (p1[1]);
    if (dup2 (p1[0], 0) < 0) die ("dup2");
    close (p1[0]);
    if (pipe (p2) < 0) die ("pipe");
```

```
    pid2 = fork ();
    if (pid2 < 0) die ("fork");
    if (pid2 > 0) {
        /* Vater des 2. Kindes (== 1. Kind) */
        close (p2[0]);
        if (dup2 (p2[1], 2) < 0) die ("dup2");
        close (p2[1]);
        execlp ("tar", "tar", "-tvzf", "-");
        die ("execlp");
    }
    /* 2. Kind.  */
    /* WENN das zweite Kind ein Kind des ersten Kindes
     * ist (und nicht ein Kind des, (Ur)vaters, dann
     * ist es hier nicht notwendig, Filedeskriptoren,
     * die zur ersten Pipe gehoeren zu schliessen.
     */
    close (p2[1]);
    if (dup2 (p2[0], 0) < 0) die ("dup2");
    close (p2[0]);
    execlp ("wc", "wc", "-l", 0);
    die ("execlp");
    return 0;
}
```

Aufgabe 3 **(10 Punkte)**

Sie haben in der Vorlesung und anderswo schon viel von **SIGPIPE** gehört. Leider haben Sie noch nie eines bekommen. Schreiben Sie ein Programm, das **ohne Verwendung von kill** solange dafür sorgt, daß es selbst ein **SIGPIPE**-Signal erhält, bis 10 solche Signale eingegangen sind. Erst dann soll Ihr Programm terminieren.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

volatile sig_atomic_t count = 0;

void handler (int sig)
{
        count++;
        write (1, "Got sig\n", 8);
}

void getsigpipe ()
{
        int fds[2];
        if (pipe (fds) < 0)
                exit (1);
        close (fds[0]);
        write (fds[1], "A", 1);
}

int main ()
{
        struct sigaction act;
        sigemptyset (&act.sa_mask);
        act.sa_handler = handler;
        act.sa_flags = 0;
        if (sigaction (SIGPIPE, &act, 0) < 0)
                return 1;
        while (count < 10)
                getsigpipe ();
        return 0;
}
```

Aufgabe 4 **(10 Punkte)**

(a) 6 Punkte
Gegeben sei das folgende Programm:

```
int main ()
{
        printf ("Guten Morgen!\n");
        printf ("Hier ist ");
        if (fork () > 0) {
                printf ("der Vater\n");
        } else {
                printf ("das Kind\n");
        }
        return 0;
}
```

Sie starten das Programm ohne weitere IO-Umlenkungen an einem Terminal und erwarten von diesem Programm die folgende Ausgabe, die Sie aber nicht erhalten:

```
Guten Morgen!
Hier ist der Vater
das Kind
```

Welche Ausgabe produziert das Programm tatsächlich? Wie muß das Programm korrigiert werden, damit es die oben angegebene Ausgabe produziert?
Begründen Sie Ihre Antworten!

```
Tats"achliche Ausgabe:
-------------------------
Guten Morgen!
Hier ist der Vater
Hier ist das Kind
-------------------------
Begruendung: Auf einem Terminal erfolgt die
Ausgabe zeilengepuffert.  Daher wird vor dem fork nur
die erste Zeile ausgeben, der Text "Hier ist" wird
gepuffert und beim fork dupliziert. Die Ausgabe
erfolgt dann sowohl im Vater als auch im Kind.
Korrektur: In das Programm muss for dem fork ein
Aufruf von fflush(0) eingefuegt werden.
```

(b) 4 Punkte

Beschreiben Sie genau, unter welchen Umständen das Schreiben in eine Pipe blockiert. Wie verhält es sich bei Sockets?

```
Das Schreiben in eine pipe blockiert, wenn mindestens
ein Prozess die pipe noch zum Lesen geoeffnet hat und
der Puffer fuer die geschriebenen aber noch nicht
gelesenen Daten voll ist.  Bei Sockets blockiert das
schreiben, wenn die Verbindung noch besteht, aber der
Kommunikationspartner schon so lange keine Daten mehr
abgenommen hat, dass der Sendepuffer auf dem lokalen
Rechner voll ist.
```

Aufgabe 5 **(10 Punkte)**

Schreiben Sie ein Programm, das zählt, wie oft *direkt* auf ein SIGUSR1-Signal das Signal SIGUSR2 folgt. Ein SIGTERM-Signal soll das Programm beenden, allerdings nicht ohne daß vorher die ermittelte Anzahl ausgeben wird. Sie können davon ausgehen, daß das Programm außer den drei genannten Signalen keine anderen Signale erhält. **Achten Sie darauf, daß im Signalhandler jeweils alle drei Signale blockiert sind!**

```
volatile sig_atomic_t count = 0;
volatile sig_atomic_t lastsig = -1;
volatile sig_atomic_t done = 0;

void termhandler (int sig)
{
    done = 1;
}

void usrhandler (int sig)
{
    if ((lastsig == SIGUSR2) && (sig ==  SIGUSR1))
        count++;
    lastsig = sig;
}

int main ()
{
    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = usrhandler;
    sigemptyset (&act.sa_mask);
    sigaddset (&act.sa_mask, SIGUSR1);
    sigaddset (&act.sa_mask, SIGUSR2);
    sigaddset (&act.sa_mask, SIGTERM);
    if (sigaction (SIGUSR1, &act, 0) < 0) {
        perror ("sigaction");
        exit (1);
    }
    if (sigaction (SIGUSR2, &act, 0) < 0) {
        perror ("sigaction");
        exit (1);
    }
    act.sa_handler = termhandler;
```

```
    if (sigaction (SIGTERM, &act, 0) < 0) {
        perror ("sigaction");
        exit (1);
    }
    while (!done)
        sleep (10);
    printf ("%d\n", count);
    return 0;
}
```

Aufgabe 6 **(12 Punkte)**

Schreiben Sie einen *forwarding*-Server. Dieser Server bekommt beim Start 3 Kommandozeilenargumente: `inputport, targethost` und `targetport`.
Beim Programmstart soll der Server *eine* eingehende Verbindung auf `inputport` annehmen und selbst eine Verbindung zu Port `targetport` auf `targethost` eröffnen. Über diese Verbindung sollen alle Daten, die auf `inputport` ankommen, unverändert an `targethost` weitergegeben werden.
Sie können davon ausgehen, daß über die Verbindung zu `targethost` keine Daten ankommen, es handelt sich also um eine unidirektionale Kommunikation! **Bei den Funktionen der Socket-Schnittstelle kann in dieser Aufgabe auf die Fehlerbehandlung verzichtet werden.**

```
void usage (char * p)
{
    fprintf (stderr,
      "usage: %s inputport targethost targetport\n", p);
    exit (1);
}

int main (int argc, char ** argv)
{
    char * host;
    int iport, oport;
    int sin, sin2, sout;
    struct sockaddr_in iaddr, oaddr;
    struct hostent * hent;

    if (argc != 4)
        usage (argv[0]);
    iport = atoi (argv[1]);
    host = argv[2];
    oport = atoi (argv[3]);

    if ((sin = socket (PF_INET, SOCK_STREAM, 0)) < 0) {
        perror ("socket");
        exit (1);
    }
    if ((sout = socket (PF_INET, SOCK_STREAM, 0)) < 0) {
        perror ("socket");
        exit (1);
    }
    bzero (&iaddr, sizeof (iaddr));
```

```
bzero (&oaddr, sizeof (oaddr));
iaddr.sin_family = AF_INET;
oaddr.sin_family = AF_INET;
iaddr.sin_port = htons (iport);
oaddr.sin_port = htons (oport);
hent = gethostbyname (host);
if (hent == 0) {
    fprintf (stderr, "Can't find host %s\n", host);
    exit (1);
}
memcpy (&oaddr.sin_addr, hent->h_addr,
        hent->h_length);
if (bind (sin, (struct sockaddr*)&iaddr,
                   sizeof (iaddr)) < 0) {
    perror ("bind");
    exit (1);
}
if (connect (sout, (struct sockaddr*)&oaddr,
                       sizeof (oaddr)) < 0) {
    perror ("connect");
    exit (1);
}
if (listen (sin, 5) < 0) {
    perror ("listen");
    exit (1);
}
sin2 = accept (sin, 0, 0);
if (sin2 < 0) {
    perror ("accept");
    exit (1);
}
close (sin);
sin = sin2;
while (1) {
    char buf[1024];
    char * p;
    int len;
    len = read (sin, buf, 1024);
    if (len < 0) {
        perror ("read");
        exit (1);
    }
    if (len == 0)
```

```
            break;
        p = buf;
        while (len) {
            int ret = write (sout, p, len);
            if (ret <= 0) {
                perror ("write");
                exit (1);
            }
            p += ret;
            len -= ret;
        }
    }
    return 0;
}
```

Aufgabe 7 **(8 Punkte)**

(a) 6 Punkte

Stellen Sie sich vor, es gäbe eine Implementierung der Socket-Schnittstelle für herkömmliche Telephonverbindungen. Geben Sie an, welche System-Calls sie dann der Reihe nach ausführen müssen, um eingehende Anrufe annehmen zu können. Dabei müssen sämtliche Parameter und die benötigten Variablendeklarationen mit angegeben werden.
Erfinden Sie ggf. sinnvolle(!) Namen für Konstanten und Datenstrukturen, die von der socket-Implementierung über Headerdateien bereitgestellt werden.

```
struct sockaddr_phon paddr;
int s, s2;
bzero (&paddr, sizeof (paddr));
s = socket (PF_PHONE, SOCK_STREAM, 0);
paddr.phon_family = AF_PHONE;
bind (s, (struct sockaddr*)&paddr, sizeof (paddr));
listen (s, 5);
s2 = accept (s, 0, 0);
```

(b) 2 Punkte

Sie wollen zusätzlich bei jedem Anruf, den Sie annehmen auch die Rufnummer des **Anrufers** feststellen. Beschreiben Sie, wie Sie dazu vorgehen und woher Sie diese Information bekommen.

```
Bei accept koennen statt den beiden Nullen weitere
Parameter mit uebergeben werden, ein Zeiger auf eine
sockaddr_phon Struktur und deren Groesse.  In diesem
Falle wird dann accept die Datenstruktur mit der
Adresse des anrufenden fuellen.
```

Aufgabe 8 **(14 Punkte)**

(a) 6 Punkte
Geben Sie je ein Anwendungsbeispiel an in dem Sie

- einem SOCK_STREAM-Socket den Vorzug vor einem SOCK_DGRAM-Socket bzw.
- einem SOCK_DGRAM-Socket den Vorzug vor einem SOCK_STREAM-Socket

geben würden. Begründen Sie Ihre Wahl jeweils ausführlich anhand der Kommunikationsemantik des jeweiligen Socket-Typs.

```
SOCK_STREAM: Bei der Uebertragung einer Datei an
einen anderen Rechner ist es wichtig, dass die
Datei exakt so ankommt, wie sie gesendet wurde.
Daher wird hier in der Regel ein STREAM-Socket
verwendet werden. Der zusaetzlich durch die sichere
Uebertragung erzeugte Aufwand faellt in diesem Fall
nicht besonders ins Gewicht.
SOCK_DGRAM: Bei einer Sprachverbindung ist es
wichtig, dass die Daten schnell beim
Gespraechspartner ankommen. Mit verspaetet
ankommenden Bruchstuecken des Gespraechs kann der
Empfaenger dann nichts mehr anfangen. In diesem Fall
ist es also besser, wenn das Gespraech kurz durch ein
Knacken oder aehliches unterbrochen wird. Hier wuerde
man also einem SOCK_DGRAM-Socket den Vorzug geben.
```

(b) 6 Punkte

Über eine SOCK_DGRAM-Verbindung werden zwei Nachrichten an einen Empfänger verschickt (z.B. mit sendmsg). Die erste Nachricht enthält die beiden Buchstaben “AB”, die zweite Nachricht enthält nur den Buchstaben “C”, weitere Nachrichten wurden nicht verschickt. Der Empfänger gibt sämtliche Nachrichten in der Reihenfolge, in der er sie erhält aus. Geben Sie für jede der folgenden Ausgaben an, ob diese möglich ist.

	Möglich?				Möglich?	
Ausgabe	Ja	Nein		Ausgabe	Ja	Nein
ABC	X			ACB		X
BCA		X		BAC		X
CAB	X			CBA		X
ABCAB	X			CCAB	X	
AACBB		X		BABAC		X
AB	X			CC	X	

(c) 2 Punkte

Welche der angegebenen Ausgaben sind bei einer SOCK_STREAM-Verbindung möglich?

```
Nur die Ausgabe "ABC" ist bei einer SOCK_STREAM
Verbindung moeglich.
```

Aufgabe 9 **(8 Punkte)**

Gegeben sei folgender Prozedurkopf:

```
void send2 (int fd1, int fd2, char * message);
```

Dabei sind `fd1` und `fd2` Filedeskriptoren, die (mindestens) zum Schreiben geöffnet sind. Die Prozedur soll dann den kompletten String `message` in denjenigen der beiden Filedeskriptoren schreiben, der als erstes zum Schreiben bereit ist ohne daß dies blockiert. Implementieren Sie diese Prozedur.
Achtung: Wenn einmal ein Filedeskriptor ausgewählt wurde, dann soll der gesamte String in diesen Filedeskriptor geschrieben werden, auch dann, wenn dies blockieren sollte.

```
void send2 (int fd1, int fd2, char * message)
{
    struct pollfd pfds[2];
    int fd = -1;
    pfds[0].fd = fd1; pfds[0].events = POLLOUT;
    pfds[1].fd = fd2; pfds[1].events = POLLOUT;
    while ((pfds[0].fd >= 0) || (pfds[1].fd >= 0)) {
        fd = -1;
        if (poll (pfds, 2, -1) < 0) {
            perror ("poll");
            exit (1);
        }
        if (pfds[0].revents & POLLOUT) {
            fd = fd1;
        } else if (pfds[0].revents) {
            pfds[0].fd = -1;
        }
        if (pfds[1].revents & POLLOUT) {
            fd = fd2;
        else if (pfds[1].revents) {
            pfds[1].fd = -1;
        }
    }
    while (*message) {
        int ret = write (fd, message, strlen (message));
        if (ret < 0) {
            perror ("write");
            exit (1);
```

```
            }
            message += ret;
        }
    }
```