ulm university universität

uulm

Probeklausur zu
„Systemnahe Software II“

SS 2012

Dr. Andreas Borchert mit Markus Schnalke

Aufgabe 1 **(15 Punkte)** Prozesse, Signale und Interprozesskommunikation

(a) 3 Punkte
Was wird von dem folgenden Programm ausgegeben, wenn alle Systemaufrufe erfolgreich verlaufen?

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
   pid_t pid1 = fork();
   pid_t pid2 = fork();
   if (pid2 == 0) printf("!\n");
}
```

Zwei Ausrufezeichen werden ausgegeben.

(b) 3 Punkte
Was wird von dem folgenden Programm ausgegeben, wenn alle Systemaufrufe erfolgreich verlaufen?

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t pid = getpid();
   if (fork() == 0) {
      kill(pid, SIGKILL);
      sleep(1);
      printf("my parent is %d\n", getppid());
   } else {
      pause();
   }
}
```

Die Ausgabe ist

```
my parent is 1
```

(c) 4 Punkte
Was wird von dem folgenden Programm ausgegeben, wenn alle Systemaufrufe erfolgreich verlaufen? Ist die Reihenfolge der Ausgabe zwingend oder sind mehrere Varianten denkbar? Begründen Sie Ihre Antwort bitte kurz.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
   int fds[2];
   pipe(fds);
   if (fork() == 0) {
      close(fds[1]);
      char ch; read(fds[0], &ch, sizeof ch);
      printf("child\n");
      exit(0);
   }
   printf("parent 1\n");
   close(fds[0]);
   close(fds[1]);
   int wstat; wait(&wstat);
   printf("parent 2\n");
}
```

Die Ausgabe ist:

```
parent 1
child
parent 2
```

Die Reihenfolge ist zwingend, da die Ausgabe „child“ erst dann erfolgt, wenn der Elternprozess beide Dateideskriptoren der Pipe schließt. Die Ausgabe „parent 2“ folgt wegen dem Aufruf von *wait* erst, sobald der Kindprozess terminiert ist.

(d) 5 Punkte

Bitte kreuzen Sie bei den folgenden Behauptungen an, ob sie zutreffen oder inkorrekt sind:

Behauptung	trifft zu	ist falsch
Ein Prozess erhält immer den gleichen Wert bei einem Aufruf von *getpid()*.	☐	☐
Ein Prozess erhält immer den gleichen Wert bei einem Aufruf von *getppid()*.	☐	☐
Zombies können mit *kill()* zum Verschwinden gebracht werden.	☐	☐
Das Signal `SIGTERM` kann abgefangen werden.	☐	☐
Ein Aufruf von *read()* kann unmittelbar zu einem `SIGPIPE`-Signal führen.	☐	☐

Ja, nein, nein, ja, nein.

Aufgabe 2 **(15 Punkte)** Prozesse

Schreiben Sie ein kleines Kommando namens *not*, das den Exit-Status eines ihm übergebenen Kommandos negiert:

```
clonard$ not
Usage: not command...
clonard$ true
clonard$ echo $?
0
clonard$ not true
clonard$ echo $?
1
clonard$ false
clonard$ echo $?
1
clonard$ not false
clonard$ echo $?
0
clonard$ not not false
clonard$ echo $?
1
clonard$
```

Im einzelnen sind folgende Punkte zu beachten:

- Wenn kein Argument angegeben wird, ist eine Usage-Meldung auf der Standardfehlerausgabe auszugeben.
- Wenn Systemaufrufe fehlschlagen, ist ein Exit-Status von 255 zurückzugeben.
- Wenn das aufgerufene Kommando mit einem Exit-Status von 0 endet, ist ein Exit-Status von 1 zurückzugeben. Bei anderen Exit-Werten ist hingegen 0 zurückzuliefern.
- Wenn das aufgerufene Kommando nicht mit einem Exit-Status terminiert, sollte der Exit-Status ebenfalls 255 sein.

Für die Untersuchung eines Status, den *wait()* hinterlässt, sollten die Makros *WIFEXITED* und *WEXITSTATUS* verwendet werden.

```
#include <unistd.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <sys/wait.h>

int main(int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc == 0) {
      fprintf(stderr, "Usage: %s command...\n", cmdname);
      exit(255);
   }
   pid_t pid = fork();
   if (pid < 0) {
      perror("fork"); exit(255);
   }
   if (pid == 0) {
      execvp(argv[0], argv);
      perror(argv[0]); exit(255);
   }
   int wstat; wait(&wstat);
   if (WIFEXITED(wstat)) {
      if (WEXITSTATUS(wstat)) {
         exit(0);
      } else {
         exit(1);
      }
   } else {
      exit(255);
   }
}
```

Aufgabe 3 **(15 Punkte)** Pipes

Manche Kommandos erzeugen sowohl Ausgabe auf *stdout* als auch auf *stderr*. Im Rahmen der gewohnten Bourne-Shell können jedoch nicht beide Ausgabeströme in getrennte Pipelines geleitet werden. Hier könnte ein kleines Werkzeug namens *pipe2* recht nützlich sein, dass diese zusätzliche Pipeline einrichtet. Folgendes Beispiel könnte dann beispielsweise die Warnungen des *gcc* wegfiltern:

```
clonard$ pipe2 gcc -c x.c -- fgrep -v warning:
```

Schreiben Sie eine Funktion *pipe2()*, die nach der Kommandozeilenbehandlung die eigentliche Funktionalität dieses Kommandos herstellt:

```
int pipe2(char** cmd_argv, char** log_argv) { // ...
```

Hierbei ist *cmd_argv* die Argumentliste für das Kommando, bei dem die Standardfehlerausgabe über eine Pipeline in das zweite Kommando zu filtern ist. Die Argumentliste für das filternde Kommando findet sich in *log_argv*. Wenn Systemaufrufe fehlschlagen, sollte der jeweilige Prozess mit *exit()* terminiert werden. Im Erfolgsfalle sollte *pipe2()* den von *wait()* zurückgelieferten Status des ersten Kommandos zurückgeben.

```
int pipe2(char** cmd_argv, char** log_argv) {
   pid_t pid = fork();
   if (pid < 0) {
      perror("fork"); exit(255);
   }
   if (pid == 0) {
      int logfds[2];
      pipe(logfds);
      pid_t logpid = fork();
      if (logpid < 0) {
         perror("fork"); exit(255);
      }
      if (logpid == 0) {
         close(logfds[1]);
         dup2(logfds[0], 0);
         close(logfds[0]);
         execvp(log_argv[0], log_argv);
         perror(log_argv[0]); exit(255);
      }
      close(logfds[0]);
```

```
      dup2(logfds[1], 2);
      close(logfds[1]);
      execvp(cmd_argv[0], cmd_argv);
      perror(cmd_argv[0]); exit(255);
   }
   int wstat; wait(&wstat);
   return wstat;
}
```

Aufgabe 4 **(15 Punkte)** Signale

(a) 3 Punkte
Welche drei prinzipiellen Möglichkeiten der Signalbehandlung stehen zur Verfügung?

Signale können zur Terminierung eines Prozesses führen (*SIG_DFL*), ignoriert werden (*SIG_IGN*) oder von einem Signalbehandler abgefangen werden.

(b) 3 Punkte
Wozu dient das Signal *SIG_CHLD*?

Damit kann auf die Terminierung erzeugter Prozesse reagiert werden.

(c) 3 Punkte
Welche Bedeutung hat das Schlüsselwort *volatile*?

Mit *volatile* gekennzeichnete Variablen werden bei jedem Zugriff aus dem Speicher erneut geladen. Globale Variablen, die von einem Signalbehandler verändert werden, sollten so gekennzeichnet werden, damit ein optimierender Übersetzer solche asynchronen Veränderungen berücksichtigt.

(d) 6 Punkte
Schreiben Sie ein kleines Programm, das nichts tut, abgesehen davon, dass es genau nach dem zweiten Empfang eines *SIGINT*-Signals terminiert.

```
#include <signal.h>

volatile sig_atomic_t sigcount = 0;

void handler(int signal) {
   ++sigcount;
}

int main() {
   struct sigaction sigact = {0};
   sigact.sa_handler = handler;
   sigaction(SIGINT, &sigact, 0);
   while (sigcount < 2) {
      pause();
   }
}
```

Aufgabe 5 **(13 Punkte)** Netzwerke

(a) 2 Punkte

Welcher Dienst bildet Rechnernamen wie *theseus.mathematik.uni-ulm.de* in IP-Adressen ab?

DNS (*domain name service*)

(b) 2 Punkte
Welcher Ebene des OSI-Schichtenmodells lässt sich das IP-Protokoll zuordnen?

3

(c) 2 Punkte

In einer Netzwerkanwendung finden Sie folgende Zeile:

```
address.sin_port = htons(port);
```

Wozu dient die Funktion *htons()*?

Konvertierung einer 16-Bit-Zahl in die *network byte order*.

(d) 2 Punkte

Was ist die naheliegendste Ursache, wenn der Systemaufruf *connect()* fehlschlägt und *errno* den Wert *ECONNREFUSED* hat bzw. eine Ausgabe von *perror()* die Fehlermeldung „Connection refused“ liefert?

Auf der Zielmaschine lauscht kein Prozess auf den angegebenen Port.

(e) 5 Punkte

Bitte kreuzen Sie bei den folgenden Behauptungen an, ob sie zutreffen oder inkorrekt sind:

Behauptung	trifft zu	ist falsch
Bei TCP können Pakete doppelt ankommen.	☐	☐
Bei UDP werden Pakete in der Reihenfolge empfangen, wie sie abgeschickt werden.	☐	☐
IPv4-Adressen benötigen 8 Bytes zur Repräsentierung.	☐	☐
Der Systemaufruf *bind()* ist auf der Klientenseite einer Netzwerkverbindung zwingend notwendig.	☐	☐
Ein Rechner kann mehrere IP-Adressen haben.	☐	☐

Nein, nein, nein, nein, ja.

Aufgabe 6 **(15 Punkte)** Netzwerkdienst

Schreiben Sie einen Netzwerkdienst, der Anfragen auf Port 11011 entgegennimmt und für jeweils vier eingelesene Bytes vier pseudo-zufällige bestimmte Bytes zurücksendet. Der Dienst soll beliebig viele Sitzungen gleichzeitig unterstützen.

Hinweise:

- Die Zufallszahlen können mit *srand()* (zum Setzen des Seed-Werts) und *rand()* erzeugt werden. Beide Funktionen arbeiten mit ganzen Zahlen.
- Die ersten eingelesenen vier Bytes sollten an *srand()* weitergegeben werden. Die weiteren sind mit dem Ergebnis von *rand()* zu verknüpfen (etwa mit dem XOR-Operator ^).
- Da die Paketgrößen mit jeweils vier Bytes recht klein sind, können Sie davon ausgehen, dass diese nicht fragmentiert werden.

```
#include <netinet/in.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define PORT 11011

int main () {
   struct sockaddr_in address = {0};
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = htonl(INADDR_ANY);
   address.sin_port = htons(PORT);

   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &address,
             sizeof address) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
```

```
      perror("socket"); exit(1);
   }

   int fd;
   while ((fd = accept(sfd, 0, 0)) >= 0) {
      pid_t pid = fork();
      if (pid == 0) {
         close(sfd);
         long seed;
         bool seeded = false;
         while (read(fd, &seed, sizeof seed) == sizeof seed) {
            long rval;
            if (!seeded) {
               srand(seed);
               rval = (rand() << 16) | rand();
               seeded = true;
            } else {
               rval = seed ^ ((rand() << 16) | rand());
            }
            if (write(fd, &rval, sizeof rval) != sizeof rval) break;
         }
         exit(0);
      }
      close(fd);
   }
}
```