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");
}
```

Lösung:

(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();
   }
}
```

Lösung:

(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");
}
```

Lösung:

(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.	☐	☐

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.

Lösung bitte auf nächster Seite!

Lösung:

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.

Lösung:

Aufgabe 4 **(15 Punkte)** Signale

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

Lösung:

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

Lösung:

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

Lösung:

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

Lösung:

Aufgabe 5 **(13 Punkte)** Netzwerke

(a) 2 Punkte

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

Lösung:

(b) 2 Punkte

Welcher Ebene des OSI-Schichtenmodells lässt sich das IP-Protokoll zuordnen?

Lösung:

(c) 2 Punkte

In einer Netzwerkanwendung finden Sie folgende Zeile:

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

Wozu dient die Funktion *htons()*?

Lösung:

(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?

Lösung:

(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.	☐	☐

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.

Lösung: