ulm university universität ulm

Klausurdeckblatt

Name der Prüfung: Systemnahe Software II

Datum und Uhrzeit:	31. Juli 2014, 14-16 Uhr	Bearbeitungszeit:	120 Min.
Institut:	Angewandte Informationsverarbeitung	Prüfer:	Dr. Andreas F. Borchert

Vom Prüfungsteilnehmer auszufüllen:

Name:	_____	Vorname:	_____	Matrikelnummer:
Studiengang:	_____	Abschluss:	_____	_____

Hiermit erkläre ich, dass ich prüfungsfähig bin. Sollte ich nicht auf der Liste der angemeldeten Studierenden aufgeführt sein, dann nehme ich hiermit zur Kenntnis, dass diese Prüfung nicht gewertet werden wird.

_____

Datum, Unterschrift des Prüfungsteilnehmers

Bitte dieses Feld für den Barcode freilassen!

Hinweise zur Prüfung: siehe nächstes Blatt

Erlaubte Hilfsmittel: Bis zu fünf handgeschriebene Blätter.

Viel Erfolg!

Vom Prüfer auszufüllen:

Note: _____ Erreichte Punkte: _____	_____ Dr. Andreas F. Borchert

Nr	Max	Bewertung	
1	**17**	xxxxx	
(a)	3		xxxxx
(b)	5		xxxxx
(c)	5		xxxxx
(d)	4		xxxxx
2	**20**	xxxxx	
3	**15**	xxxxx	
4	**17**	xxxxx	
(a)	3		xxxxx
(b)	3		xxxxx
(c)	3		xxxxx
(d)	8		xxxxx

Nr	Max	Bewertung	
5	**15**	xxxxx	
(a)	2		xxxxx
(b)	2		xxxxx
(c)	2		xxxxx
(d)	2		xxxxx
(e)	2		xxxxx
(f)	5		xxxxx
6	**16**	xxxxx	
(a)	2		xxxxx
(b)	3		xxxxx
(c)	3		xxxxx
(d)	3		xxxxx
(e)	3		xxxxx
(f)	2		xxxxx
Summe	**100**		

- Prüfen Sie zu Beginn, ob Ihre Klausur ab der Aufgabe 1 aus 14 durchnummerierten Seiten besteht.
- Für Ihre Lösungen verwenden Sie bitte den freigelassenen Platz nach der Aufgabenstellung, die Rückseite der jeweiligen Aufgabe oder die angehängte leere Seite unter Angabe der Aufgabennummer.
- Nennen Sie möglichst alle Annahmen, die Sie gegebenenfalls für die Lösung einer Aufgabe treffen!
- Sofern nichts anderes angegeben ist, können Sie bei den Programmier-Aufgaben auf die Angabe der notwendigen *#include*-Anweisungen verzichten.
- Wenn es bezüglich der Aufgabenstellung Unklarheiten gibt, dann scheuen Sie sich bitte nicht, jemanden von der Aufsicht zu befragen.
- Wenn wir während der Klausur feststellen, dass eine Aufgabenstellung missverständlich ist, werden wir an der Tafel einen klärenden Hinweis für alle sichtbar hinschreiben.

Aufgabe 1 **(17 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 (pid1 || pid2) printf("!\n");
}
```

Drei Ausrufezeichen werden ausgegeben.

(b) 5 Punkte

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

Behauptung	trifft zu	ist falsch
Ein Prozess kann sich selbst mit *kill(getpid(), SIGKILL)* beenden.	☐	☐
Der Exit-Status (Wert bei *exit()*) umfasst 16 Bits.	☐	☐
Zombies können mit *wait()* zum Verschwinden gebracht werden.	☐	☐
Das Signal `SIGKILL` kann abgefangen werden.	☐	☐
Ein Aufruf von *write()* kann unmittelbar zu einem `SIGPIPE`-Signal führen.	☐	☐

Ja, nein, ja, nein, ja.

(c) 5 Punkte
Was wird von dem folgenden Programm ausgegeben, wenn alle Systemaufrufe erfolgreich verlaufen?

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
   if (fork() == 0) {
      pid_t pid = getpid();
      if (fork() == 0) {
         sleep(1);
         printf("pid == getppid(): %d\n", pid == getppid());
         exit(1);
      } else {
         exit(2);
      }
   } else {
      int wstat;
      wait(&wstat);
      if (WIFEXITED(wstat)) {
         printf("exit status = %d\n", WEXITSTATUS(wstat));
      }
   }
}
```

```
exit status = 2
pid == getppid(): 0
```

(d) 4 Punkte

Was wird von dem folgenden Programm ausgegeben, wenn die Systemaufrufe *pipe()*, *fork()*, *close()* und *wait()* allesamt erfolgreich verlaufen? Begründen Sie Ihre Antwort bitte kurz.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
   int fds[2];
   pipe(fds);
   if (fork() == 0) {
      close(fds[0]);
      sleep(1);
      char ch = 'x'; write(fds[1], &ch, sizeof ch);
      exit(0);
   }
   close(fds[1]); close(fds[0]);
   int wstat;
   if (wait(&wstat) > 0) {
      if (WIFEXITED(wstat)) {
         printf("exit status of child = %d\n", WEXITSTATUS(wstat));
      } else {
         printf("child aborted\n");
      }
   }
}
```

Die Ausgabe ist:

```
child aborted
```

Der erzeugte Prozess wird nach dem Eintreffen des SIGPIPE-Signals bei der Schreib-Operation terminiert.

Aufgabe 2 **(20 Punkte)** Prozesse

Schreiben Sie ein kleines Werkzeug namens *try*, das ein Kommando immer wieder erneut ausführt, bis es entweder mit den Exit-Codes 0 oder 255 endet oder durch ein Signal ohne Exit-Status terminiert. Dabei spezifiziert das erste Kommandozeilenargument die maximale Anzahl der Versuche.

```
clonard$ try
Usage: try count command
clonard$ try 0 date
clonard$ echo $?
0
clonard$ try 2 date
Tue Jul 10 15:32:16 MEST 2012
clonard$ echo $?
0
clonard$ try 2 cat tmpfile
cat: tmpfile: No such file or directory
cat: tmpfile: No such file or directory
clonard$ echo $?
1
clonard$ (sleep 1; echo tmpfile >tmpfile)&
[1] 19584
clonard$ try 2 cat tmpfile
cat: tmpfile: No such file or directory
tmpfile
[1]+  Done                    ( sleep 1; echo tmpfile > tmpfile )
clonard$
```

Im einzelnen sind folgende Punkte zu beachten:

- Wenn die Anzahl oder das auszuführende Kommando fehlt, ist eine Usage-Meldung auf der Standardfehlerausgabe auszugeben und der Prozess mit einem Exit-Status von 255 zu beenden.
- Wenn Systemaufrufe fehlschlagen, ist ein Exit-Status von 255 zurückzugeben.
- Wenn das aufgerufene Kommando mit einem Exit-Status von 0 terminiert, sollte der Exit-Status ebenfalls 0 sein.
- Wenn das aufgerufene Kommando mit einem Exit-Status von 255 endet oder durch ein Signal ohne Exit-Status terminiert, ist *try* ebenfalls sofort mit einem Exit-Status von 255 zu beenden.
- Bei einem anderen Exit-Status des aufgerufenen Kommandos ist, sofern die maximale Zahl der Versuche noch nicht erreicht worden ist, eine Sekunde zu warten, bevor ein erneuter Versuch unternommen wird.

- Wenn die maximale Zahl der Versuche erschöpft ist, sollte *try* mit einem Exit-Code von 1 enden.
- Wenn die Zahl der Versuche nicht positiv ist, dann ist das Kommando nicht auszuführen und ein Exit-Status von 0 zurückzugeben.

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

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void try(char* argv[]) {
   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;
   if (wait(&wstat) < 0) {
      perror("wait"); exit(1);
   }
   if (WIFEXITED(wstat)) {
      int status = WEXITSTATUS(wstat);
      if (status == 0 || status == 255) exit(status);
   } else {
      exit(255);
   }
}

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   if (argc < 2 || !isdigit(argv[0][0])) {
      fprintf(stderr, "Usage: %s count command\n", cmdname); exit(255);
   }
   unsigned int count = atoi(*argv++); --argc;

   if (count > 0) {
```

```
      try(argv);
      while (--count > 0) {
         sleep(1); try(argv);
      }
   }
   exit(0);
}
```

Aufgabe 3 **(15 Punkte)** Pipes

Zu implementieren ist eine Funktion *randfeed*, die einen Dateideskriptor zurückgibt, von dem eine pseudo-zufällige Bytesequenz eingelesen werden kann:

```
int randfeed() {
   // ...
}
```

Die Implementierung muss mit Hilfe einer Pipeline und eines neu erzeugten Prozesses erfolgen. Der neue Prozess soll dabei mit *srand* eine pseudo-zufällige Sequenz initiieren (als Seed empfiehlt sich ein XOR aus der Prozess-ID und der aktuellen Zeit) und dann sukzessive die Bytes aus der pseudo-zufälligen Sequenz mit *rand* erzeugen, wobei pro Byte nur die 8 niedrigstwertigen Bits der von *rand()* zurückgelieferten Zahl zu nehmen sind. Um die Zahl der Systemaufrufe zu minimieren, muss *write* jeweils mit einem größeren Puffer aufgerufen werden, der zuvor mit pseudo-zufälligen Bytes gefüllt wurde.
Wenn irgendwelche Systemaufrufe fehlschlagen, hat *randfeed()* einen Wert von -1 zurückzugeben.

```
int randfeed() {
   int pipefds[2];
   if (pipe(pipefds) < 0) return -1;
   pid_t pid = fork();
   if (pid < 0) {
      close(pipefds[0]); close(pipefds[1]); return -1;
   }
   if (pid == 0) {
      close(pipefds[0]);
      srand(getpid() ^ time(0));
      for(;;) {
         char buf[BUFSIZ];
         for (char* cp = buf; cp < buf + BUFSIZ; ++cp) {
            *cp = rand() & 0xff;
         }
         if (write(pipefds[1], buf, sizeof buf) < 0) exit(1);
      }
   }
   close(pipefds[1]);
   return pipefds[0];
}
```

Aufgabe 4 **(17 Punkte)** Signale

(a) 3 Punkte
Wozu dient das Signal *SIG_ALRM* und mit welchem Systemaufruf steht es in Verbindung?

Dient als Wecksignal und wird durch den Systemaufruf *alarm* initiiert.

(b) 3 Punkte
Welche Signalbehandlung ist die Voreinstellung bei *SIGCHLD*?

Das Signal wird ignoriert.

(c) 3 Punkte
Welche Eigenschaften hat der Datentyp *sig_atomic_t*?

Dieser Datentyp ist ganzzahlig und sowohl Lese- als auch Schreibzugriffe darauf erfolgen atomar.

(d) 8 Punkte

Schreiben Sie ein kleines Programm, das über einen Signalbehandler verfügt, der auf *SIGFPE* reagiert und der mit einer Fehlermeldung auf der Standardfehlerausgabe den Prozess beendet. Nachdem der Signalbehandler eingesetzt ist, sollte das Signal durch eine ganzzahlige Division durch 0 ausgelöst werden.

Achten Sie dabei genau darauf, was innerhalb eines Signalbehandlers zulässig ist und was nicht.

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

void fpe(int signal) {
   static const char errmsg[] = "Division by 0. Exiting.\n";
   write(2, errmsg, sizeof errmsg - 1);
   _Exit(1);
}

int main() {
   struct sigaction sigact = {0};
   sigact.sa_handler = fpe;
   sigaction(SIGFPE, &sigact, 0);
   int i = 1; int j = 0;
   i = i / j;
}
```

Aufgabe 5 **(15 Punkte)** Netzwerke

(a) 2 Punkte
Ist 134.60.54.12 ein Beispiel für einen Domainnamen, eine IPv4-Adresse, eine IPv6-Adresse oder eine Portnummer?

IPv4-Adresse

(b) 2 Punkte
Wieviel Bytes benötigt eine IPv4-Adresse?

4

(c) 2 Punkte
Was ist die Aufgabe eines sogenannten Root-Servers?

Root-Server liefern die Adressen der Name-Server der Top-Level-Domains wie z.B. „.de“ oder „.com“.

(d) 2 Punkte
Mit Hilfe welcher Funktion können Sie Domainnamen in IP-Adressen verwandeln? Und welcher Dienst wird dazu in Anspruch genommen?

gethostbyname(), Domain-Server

(e) 2 Punkte
Was ist die naheliegendste Ursache, wenn der Systemaufruf *connect()* fehlschlägt und *errno* den Wert *ETIMEDOUT* hat bzw. eine Ausgabe von *perror()* die Fehlermeldung „Connection timed out“ liefert?

Die Gegenseite reagiert nicht, z.B. weil die Netzwerkverbindung unterbrochen ist oder der entsprechende Rechner zur Zeit nicht läuft.

(f) 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 in einer falschen Reihenfolge ankommen.	☐	☐
Bei UDP können Pakete doppelt ankommen.	☐	☐
IPv6-Adressen benötigen 16 Bytes zur Repräsentierung.	☐	☐
Der Systemaufruf *listen()* ist auf der Server-Seite einer Netzwerkverbindung zwingend notwendig, um Verbindungen entgegenzunehmen.	☐	☐
Die TCP-Schicht ist dem OSI-Layer 3 zuzuordnen.	☐	☐

Nein, ja, ja, ja, nein.

Aufgabe 6 **(16 Punkte)** Netzwerkdienste

(a) 2 Punkte
Bei einem Netzwerkdienst entdecken Sie zu Beginn folgende Zeile:

```
int sfd = socket(PF_INET, SOCK_STREAM, 0);
```

Welche Bedeutung hat hier *SOCK_STREAM*?

Damit wird festgelegt, dass die über diese Socket eröffneten Netzwerkverbindungen verbindungsorientiert sind auf Basis von TCP.

(b) 3 Punkte
Etwas später sehen Sie folgenden Funktionsaufruf. Was wird damit erreicht?

```
int optval = 1;
if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
      &optval, sizeof optval) < 0) {
   // error handling
}
```

Das stellt sicher, dass die verwendete Portnummer nach Schließung der Socket sofort wieder erneut verwendet werden kann.

(c) 3 Punkte
Bei einem Netzwerkdienst mit parallelen Sitzungen auf Basis von *fork()* entdecken Sie folgenden Programmtext. Welche Bedeutung hat dieser?

```
struct sigaction action = {0};
action.sa_handler = SIG_IGN;
action.sa_flags |= SA_NOCLDWAIT;
if (sigaction(SIGCHLD, &action, 0) < 0) return;
```

Damit wird sichergestellt, dass die erzeugten Prozesse nicht als Zombies enden, ohne das jeweils *wait()* erforderlich ist.

(d) 3 Punkte

Bei dem gleichen Netzwerkdienst finden Sie etwas weiter folgendes Konstrukt. Nun stellen Sie fest, dass der Dienst für einige Zeit problemlos läuft, dann aber irgendwann mit der von *perror* erzeugten Meldung „Too many open files“ abbricht. Erklären Sie die Ursache und zeigen Sie, wie das Problem vermieden werden kann.

```
int fd;
while ((fd = accept(sfd, 0, 0)) >= 0) {
   pid_t child = fork();
   if (child == 0) {
      close(sfd);
      handler(fd, argc, argv);
      exit(0);
   }
}
perror("accept");
```

Der erzeugende Prozess sollte jeweils *fd* schließen, bevor *accept* wieder erneut aufgerufen wird. Sonst gehen irgendwann einmal die Dateideskriptoren aus.

(e) 3 Punkte

Angenommen ein Netzwerkdienst erwartet Anfragen, die mit einem Zeilentrenner abgeschlossen werden. In der Implementierung finden Sie dann folgende Funktion, um eine Anfrage einzulesen:

```
int read_request(int fd, stralloc* request) {
   request.len = 0;
   char ch;
```

```
    ssize_t nbytes;
    while ((nbytes = read(fd, &ch, sizeof ch)) > 0 && ch != '\n') {
       stralloc_append(&line, &ch);
    }
    return request.len > 0 || nbytes > 0;
}
```

Ist das eine gute Implementierung? Was sollte ggf. verbessert werden?

Das ist eine sehr ineffiziente Implementierung, da *read* unnötig oft aufgerufen wird. Stattdessen sollte die Eingabe gepuffert werden.

(f) 2 Punkte

Angenommen, Sie haben zwei Dateideskriptoren, die jeweils zu einer *SOCK_STREAM*-Netzwerkverbindung gehören, und Sie wollen von beiden gleichzeitig lesen und jeweils auf die nächste zur Verfügung stehende Eingabe sofort reagieren. Welchen Systemaufruf würden Sie dazu benötigen?

poll()