2A Basistechniken: Weitere Aufgaben

2A.3 Programmierung unter UNIX/Linux

1. Gegeben sind einige Ausschnitte von C-Programmen, die unter UNIX/Linux ausgeführt werden sollen. Beantworten Sie die zugehörigen Fragen!

```
int status;
if (fork()==0) {
  sleep(2);
  exit(0); }
wait(&status);
printf("Hallo");
```

Nach wieviel Sekunden erscheint der Text "Hallo" frühestens auf dem Bildschirm? Kurze Begründung!

Frühestens nach 2 Sekunden. Der Sohn schläft 2 Sekunden und beendet sich dann. Der Vater wartet auf das Ende des Sohns und gibt dann erst den Text aus.

```
int i,pid;
if ((pid=fork())==0) {
  ... Sohn ... }
sleep(5);
kill(pid,SIGKILL);
```

Was macht der Vaterprozess in diesem Programm?

Schläft zunächst 5 Sekunden und beendet dann seinen Sohnprozess.

```
int pid;
if ((pid=fork())==0) {
  ... Sohn ... }
printf("%d",pid);
Ist der Wert von pid, der auf den Bildschirm ausgegeben wird, gleich oder größer als
0? Kurze Begründung!
```

Größer als 0, denn fork() liefert an den Vaterprozess die PID des Sohns zurück, und die ist stets größer als 0.

2. Skizzieren Sie den Aufbau eines Unix-C-Programmstücks, bei dem ein Vaterprozess einen Sohnprozess erzeugt und dieser Sohnprozess dann seinerseits zwei weitere Sohnprozesse (also gewissermaßen zwei "Enkel") erzeugt.

```
... Vater ...
if (fork()==0) {
  ... Sohn ...
  if (fork()==0) {
    ... Enkel 1 ...
   exit(0); }
  if (fork()==0) {
    ... Enkel 2 ...
```

```
   exit(0); }
 exit(0);
}
```

Die folgende Tabelle soll sich auf die Prozesse aus dem neu erstellten Programm beziehen. Ergänzen Sie sie durch sinnvolle Werte für die jeweilige PID und PPID.

	PID	PPID
Vater		
Sohn		
Enkel 1		
Enkel 2		

PIDs, PPIDs:

	PID	*PPID*
Vater	*4712*	*4711*
Sohn	*4713*	*4712*
Enkel 1	*4714*	*4713*
Enkel 2	*4715*	*4713*

3. Betrachten Sie den folgenden Ausschnitt eines Unix-C-Programms:

```
int status = 33;
if (fork()==0) {
 ... Sohn 1 tut hier etwas ...
 exit(0); }
wait(&status);
if (fork()==0) {
 ... Sohn 2 tut hier etwas ...
 exit(-1); }
wait(&status);
... Vater räumt auf ...
```

Warum laufen hier die beiden Söhne nicht "gleichzeitig", sondern hintereinander ab? Wie könnte man erreichen, dass sie doch gleichzeitig aktiv sind?

Der Vater wartet im ersten wait(), bis der erste Sohn fertig wird, und startet dann erst den zweiten. Man müsste das erste wait() hinter den zweiten Sohn verschieben.

An welcher/welchen Stelle(n) ändert die Variable status des Vaters ihren Wert? Wodurch wird der neue Wert bestimmt?

Bei beiden wait()-Aufrufen. Der neue Wert ergibt sich jeweils aus dem Wert, den der Sohn, auf den gewartet wurde, als Parameter in seinem exit()-Aufruf übergeben hat.

4. Betrachten Sie das folgende Programm:

```
#include <stdio.h>
```

```
#include <stdlib.h>
main() {
 int i, err;
 for (i=0;i<3;i++)
  if (fork()==0) {
   sleep(i+1);
   printf("Sohn Nr. %d wird jetzt fertig.\n",i);
   exit(0);
  }
 wait(&err);
 printf("Hier ist der Vater.\n");
 printf("Alle meine Söhne sind fertig\n");
}
```

Welchen Effekt hat die for-Schleife?

Vaterprozess erzeugt drei Sohnprozesse, die eine, zwei bzw. drei Sekunden schlafen, dann eine Meldung ausgeben und terminieren.

Ist die Aussage der printf-Meldungen so korrekt? Wenn nein, wie müßte man das Programm korrigieren?

Nicht korrekt, da wait() nur auf die Terminierung eines Sohns wartet. Nötig wären hier drei wait()-Aufrufe hintereinander.

5. Betrachten Sie den folgenden Programmausschnitt:

```
int p;
if ((p=fork())!=0) {
  printf("Ich bin der Sohn\n");
  printf("Meine PID ist %d\n",p);
  }
 else {
  printf("Ich bin der Vater\n");
  printf("PID meines Sohns ist %d\n",p);
  }
```

Sind die Aussagen "Ich bin der Sohn" bzw. "Ich bin der Vater" im ersten bzw. dritten printf-Aufruf so korrekt? Begründung!

Nicht korrekt, denn fork() gibt dem Vater eine PID ungleich 0 zurück und dem Sohn eine 0. Die printf-Aufrufe sind also vertauscht.

Ist die Verwendung von `p` im zweiten bzw. vierten printf-Aufruf so korrekt? Begründung!

Nur beim Vater korrekt; beim Sohn hat p den Wert 0.

6. Betrachten Sie den folgenden Programmausschnitt:

```
int n=3;
...
for (i=0;i<n;i++)
 if (fork()==0) {
  printf("Hallo, hier bin ich!\n);
  exit(0);
 }
printf("Gute Nacht!\n");
```

```
sleep(3600);
```

Wie oft wird "Hallo, ..." auf den Bildschirm ausgegeben und wie oft "Gute Nacht"?

"Hallo, ..." dreimal und "Gute Nacht" einmal.

Streichen Sie jetzt die exit-Anweisung. Warum werden nun "Hallo, ..." und "Gute Nacht" öfter ausgegeben?

Die Söhne werden nach ihrer Ausgabe nicht mehr durch exit beendet, sondern treten ihrerseits mit dem jeweils aktuellen i-Wert in die for-Schleife ein, wodurch neue Prozesse erzeugt werden. Es kommt also zu einer begrenzten Kettenreaktion.

Warum ist also das `exit()` im Programm aus Teilaufgabe b.) wichtig in Hinblick auf die Arbeitsfähigkeit des gesamten Systems? (Hinweis: Überlegen Sie sich, was für große Werte von `n` passiert!). Was kann das Betriebssystem tun um zu verhindern, dass Benutzer mit solchen Programmen Probleme verursachen?

Wenn das exit fehlte, so würde das Programm, insbesondere für große n, zu einer Lawine von neuen Prozessen führen, die im Extremfall das gesamte System lahmlegen könnten. Das Betriebssystem kann das verhindern, indem es eine Obergrenze für die Zahl der Prozesse festlegt, die für einen Benutzer gleichzeitig aktiv sein dürfen.

7. Gegeben ist das folgende einfache Problem: Ein UNIX-Vaterprozess soll in einer for-Schleife zehn Sohnprozesse erzeugen. Jeder dieser Sohnprozesse soll nur eine einzige Operation ausführen, nämlich den Wert ausgeben, den der Schleifenzähler bei seiner Erzeugung hatte. Die Reihenfolge der Ausgaben ist dabei beliebig.

   Es wird die folgende Lösung vorgeschlagen:

```
...
int i;
for (i=0;i<10;i++)
 if (fork()==0)
  printf("Schleifenzähler = %d\n",i);
...
```

   Ist diese Lösung korrekt? Wenn nein: Wie müßte sie korrigiert werden, um den gewünschten Effekt zu erzielen?

   Nicht korrekt, da das exit() nach dem printf() fehlt, mit dem die Sohnprozesse jeweils gleich wieder terminiert werden.

Allgemeiner Hinweis: Bevor Sie sich ausloggen, sollten Sie mit `ps` prüfen, ob noch Prozesse für Sie aktiv sind und diese (natürlich außer der Shell) mit `kill` löschen.

2A.4 Programmierung in Java

1. Schreiben Sie ein Java-Programm, das eine grafische Oberfläche mit einem Button erzeugt. Beim Anklicken des Buttons soll der Thread, der den zugehörigen Listener ausführt, seine ID auf die Konsole ausgeben. Zudem soll der Thread, der das Hauptprogramm ausführt, seine ID ausgeben. Was beobachten Sie?

   1.)import java.awt.;*
   import java.awt.event.;*
   import javax.swing.;*

```
public class GUIThreads {
 public static void main(String args[]) {
  JFrame frame;
  JButton button ;
  JTextField textfeld;
  class ButtonListener implements ActionListener {
   JTextField tf;
   ButtonListener(JTextField tf) {
    this.tf = tf;
   }
   public void actionPerformed(ActionEvent e) {
    if (tf.getText().equals(""))
      tf.setText("Hallo");
     else
      tf.setText("");
    System.out.println("ThreadID im Listener: "+Thread.currentThread().getId());
   }
  }
  frame = new JFrame();
  frame.getContentPane().setLayout(new GridLayout(2,1));
  textfeld = new JTextField();
  textfeld.setFont(new Font("Arial",Font.BOLD,24));
  frame.getContentPane().add(textfeld);
  button = new JButton("Click");
  button.setFont(new Font("Arial",Font.BOLD,24));
  button.setForeground(Color.black);
  button.addActionListener(new ButtonListener(textfeld));
  frame.getContentPane().add(Box.createVerticalStrut(10));
  frame.getContentPane().add(button);
  frame.pack();
  frame.setVisible(true);
  frame.setLocation(300,200);
  frame.setSize(250,100);
  while (true) {
     System.out.println("ThreadID in main(): "+Thread.currentThread().getId());
     try {
       Thread.currentThread().sleep(1000);
     } catch (InterruptedException e) {}
   }
 }
}
```