Technische Informatik 2 | WS 2016/17 | C/06
Tutor: Marc Hildebrandt

Niklas Koenen
Jan Klüver
Vincent Jankovic

Übungsblatt 3

1	2	3	4	5	$\sum$

Lösungsvorschlag
Abgabe: 21.11.2016

Aufgabe 1

Zuerst werden Signale und Interrupts durch die Eingabe des Befehles $ls - 1R/ > ausgabe$ in der Shell über die Tastatur ausgelöst. Diese werden dann entsprechend vom Betriebssystem behandelt (Interrupthandler). Wird der Befehl ausgeführt (ein Interrupt wird ausgelöst und ein Signal 'erzeugt' den Prozess), startet der Prozess, um die Daten in die *ausgabe*-Datei zu schreiben. Bis zu dem Zeitpunkt der Tastatureingabe von *Strg-Z* können nach belieben prozessinterne Pausen (Traps) und andere Interrupts auftreten. Dies hängt aber von dem allgemeinen Setup des Computers bzw. Betriebssystems und dem aktuellen Zustand ab (Sheduling). Wird nun *Strg-Z* eingegeben, pausiert der Prozess, ausgehend von einem Interrupt der Tastatur (da der Befehl der Tastatur asynchron zum Prozess bearbeitet wird, handelt es sich um ein Interrupt). Desweiteren wird neben dem typischen Tastatur-Interrupt auch ein Signal ausgelöst, welches vom Prozess bearbeitet wird und ihn in einen blockierten Zustand setzt (später dazu mehr). Nun wartet der Prozess auf einen Input. Nach der Eingabe von *bg* wird der Prozess in den Hintergrund verlagert und mit dem Vordergrundprozess (vllt. auch mehr) gesheduled. Dabei interagiert der Prozess natürlich mit dem Betriebssystem, welches über Signale ermöglicht wird. Ist der Prozess fertig, wird er terminiert (evtl. durch einen Trap). Dann wird ein Signal an den Shell-Prozess gesendet und das Betriebssystem und damit auch der Nutzer über das Ergebnis des Prozesses infomiert.

Nun zu den Zustandsänderungen der Prozesse. Wir beschränken uns auf den Hauptprozess des Befehls in der Aufgabe (also keine Nebenprozesse). Nach der Befehleingabe ist der Prozess in einem rechnendem Zustand (Betritt den Kern bei Traps) und geht nach der Eingabe von *Strg Z* in einen blockierten Zustand über. Der Prozess wartet nun auf einen weiteren Befehl. Nach der Eingabe von *bg* geht der Prozess wieder in einen rechenbereiten Modus über und der Sheduler 'übernimmt' die weiteren Zustandsänderungen. Wurde fertig gerechnet, wird der Prozess terminiert und ist in einem beendeten Zustand.

Aufgabe 2

In dieser Aufgabe mussten wir einen Handler für das Signal **SIGUSR1** mittels **sigaction** schreiben. Dazu brauchten wir drei Bibliotheken, nämlich die für die Signale, die Standardausgabe und die für die Methode **pause()**:

```
0  #include <signal.h>
1  #include <iostream>
2  #include <unistd.h>
```

Außerdem brauchten wir eine Handler-Funktion, die ausgeführt wird, wenn unser Signal vorkommt. Diese braucht per Definition drei bestimmte Parameter, damit wir später *sa_saction*

diese Handler-Funktion zuweisen können, wobei die Integer die Signalnummer und *siginfo_t* alle relevanten Informationen über das Signal enthält. Der Handler soll die Nummer, die Prozess-ID und die User-ID ausgeben, was wir mit der Standardausgabe bewältigt haben. Insgesamt sieht die Handler-Funktion wie folgt aus:

```
4   void handler(int sig, siginfo_t *act, void *context)
5   {
6     std::cout<<"Signal gefunden!" << std::endl;
7     std::cout<<"Nummer " << sig << "  PID: " <<  act->si_pid <<  "  UID: "
8           <<  act->si_uid <<std::endl;
9   }
```

In der *main*-Methode wird zunächst die Struktur *sigaction* erstellt und anschließend die zuvor beschriebene Handler-Funktion an *sigaction* weitergegeben, damit diese Funktion bei einem Auftreten eines Signals auch ausgeführt wird. Außerdem wird *sa_flags* auf *SA_SIGINFO* gesetzt. Nun wird mit einer *if*-Anweisung geschaut, ob der Handler *sigaction* für das Signal **SIGUSR1** erstellt werden kann, ansonsten wird ein Fehler ausgegeben. Wenn alles erstellt werden kann, wird eine Endlosschleife mit *while(pause)* gestartet, die immer ausgeführt wird, wenn ein Signal eingeht. Insgesamt sieht die *main* also so aus:

```
10  int main (void)
11  {
12    struct sigaction sa;
13    sa.sa_sigaction= &handler;
14    sa.sa_flags = SA_SIGINFO;
15    if(sigaction(SIGUSR1, &sa, NULL) != 0)
16      {
17        std::cout<< "Error!"<<std::endl;
18        return -1;
19      }
20    while(pause())
21      {
22      }
23      return 0;
24  }
```

Der gesamte Quellcode sieht also wie folgt aus:

```
0   #include <signal.h>
1   #include <iostream>
2   #include <unistd.h>
3
4   void handler(int sig, siginfo_t *act, void *context)
5   {
6     std::cout<<"Signal gefunden!" << std::endl;
7     std::cout<<"Nummer " << sig << "  PID: " <<  act->si_pid <<  "  UID: " <<  act->si_uid
8   }
9
10  int main (void)
11  {
12    struct sigaction sa;
13    sa.sa_sigaction= &handler;
14    sa.sa_flags = SA_SIGINFO;
```

```
15      if(sigaction(SIGUSR1, &sa, NULL) != 0)
16        {
17          std::cout<< "Error!"<<std::endl;
18          return -1;
19        }
20      while(pause())
21        {
22        }
23        return 0;
24    }
```

Anschließend haben wir das Programm getestet. Dabei haben wir in einem Terminal das Programm kompiliert und ausgeführt. Dann haben wir in einem anderen Terminal mit **ps -e** alle Prozesse ausgegeben und das zu unserem Programm herausgesucht. Nun konnten wir mit **kill -usr1 [PID]** das Signal **SIGUSR1** senden. Unser Programm hat darauf das folgende ausgegeben:

```
0   Signal gefunden!
1   Nummer 10  PID: 8724  UID: 43042
```

Aufgabe 3

a) Wenn der Kontostand nicht halbiert würde, bekämen Prozesse, die schon lange laufen (also einen hohen Kontostand haben), sehr lange nicht die CPU, wenn ein neuer Prozess gestartet wird. Dieser hätte nämlich zuerst den Kontostand 0 und würde die CPU wohl für viele Zeitscheiben hintereinander bekommen und die ggf. noch aktiven Langzeitprozesse wären außen vor. Da immer mal wieder neue Prozesse starten könnten, würden die älteren unter Umständen gar nicht mehr an die Reihe kommen.

b) Basisprioritäten: A=56 , B=10

Zeitscheiben	0	1	2	3	4	5	6	7	8	9	10	11	12
Konto A	0	0	0	50	25	12	56	28	14	57	28	14	57
Nutzung A	0	0	100	0	0	100	0	0	100	0	0	100	0
Konto' A	0	0	50	25	12	56	28	14	57	28	14	57	28
Prio A	56	56	106	81	68	112	84	70	113	84	70	113	84
Konto B	0	0	50	25	62	81	40	70	85	42	71	85	42
Nutzung B	0	100	0	100	100	0	100	100	0	100	100	0	100
Konto' B	0	50	25	62	81	40	70	85	42	71	85	42	71
Prio B	10	60	35	72	91	50	80	95	52	81	95	52	81

Man sieht, dass Prozess B stets zweimal hintereinander die CPU bekommt, woraufhin A diese dann nur für eine Zeitscheibe kriegt. Die Werte für Konto, etc scheinen sich auch bis auf kleine Abweichungen in diesem Dreierzyklus zu wiederholen (erkennbar ab Spalte 4).

c) Sei $(a_n)_{n\in\mathbb{N}} \subset \mathbb{R}$ definiert durch $a_{n+1} := \frac{a_n+100}{2}$ und $a_0 = 0$ die Folge, deren Glieder den Kontostand von A repräsentieren (zunächst ohne Abrunden). Konvergenz ist klar, da

monoton wachsend und nach oben z.B. durch 100 beschränkt. Für den Grenzwert gilt:

$$a* = \frac{a* + 100}{2}$$
$$\Leftrightarrow a* = 100$$

Da $a_n < 100$ für alle $n \in \mathbb{N}$, gilt: $\exists N_0 \in \mathbb{N} \forall n \geq N_0 : \lfloor a_n \rfloor = 99$
Also hat A irgendwann stets den Kontostand 99. Somit würde B die CPU niemals erhalten, wenn es eine Basispriorität von 99 oder höher hätte, da bei Gleichstand alphabetisch vorgegangen wird.

d) Basisprioritäten: A=0 , B=15

Zeitscheiben	0	1	2	3	4	5	6	7	8	9	10	11	12
Konto A	0	0	50	61	80	76	88	80	90	95	97	84	92
Nutzung A	0	100	72,5	100	72,5	100	72,5	100	72,5	100	72,5	100	72,5
Konto' A	0	50	61	80	76	88	80	90	95	97	84	92	82
Prio A	0	50	61	80	76	88	80	90	95	97	84	92	82
Konto B	0	0	0	13	21	24	12	19	9	18	9	18	9
Nutzung B	0	0	27,5	0	27,5	0	27,5	0	27,5	0	27,5	0	27,5
Konto' B	0	0	13	6	24	12	19	9	18	9	18	9	18
Prio B	15	15	28	21	39	27	34	24	33	24	33	24	33
lauffähig?	J	J	N	J	N	J	N	J	N	J	N	J	N