Technische Informatik 2
Tutor: Marc/Bingbin

WS 2016/17

C02

Rene Engel
Dennis Jacob
Jan Schoneberg

Übungsblatt 3

1	2	3	$\sum$

Lösungsvorschlag
Abgabe: 21.11.2016

Aufgabe 1

> Beschreibt kurz an welchen Stellen Traps, Interrupts und Signale ausgelöst werden. Welche Zustandsänderungen werden dadurch für die hier beteiligten Prozesse bewirkt?

Mit dem genannten Befehl werden alle nicht versteckten Dateien und Verzeichnisse des gesamten Systems in Langform an die Datei `ausgabe` gesendet. Mit der Option `-l` werden die Informationen in Langform angegeben. Mit der Option `-R` werden auch die Inhalte aller Unterverzeichnisse ausgegeben. Mit `>` wird die Ausgabe an die Datei `ausgabe` weitergeleitet. Falls diese noch nicht existiert wird diese durch *STDOUT* erstellt.

Traps werden durch Prozesse selbst ausgelöst und sorgen dafür, dass die normale Abarbeitung des Prozesses für die Bearbeitung eines anderen Prozesses aussetzt. Dabei wird temporär der Programmablauf verlassen. Auch bei **Interrupts** wird die Bearbeitung von Prozessen kurzfristig ausgesetzt, allerdings erfolgen Interrupts ausgehend von der Prozessbehandlung des Kernels, der die CPU je nach Bedarf den nötigen Prozessen zum weiteren Ablauf zuteilt.

Im wesentlich ergeben sich durch das genannte Szenario drei Hauptprozesse die in allen Szenarioschritten beachtet werden müssen: Der **Init**-Prozess, der **SHELL**-Prozess und davon erzeugte Kindprozess **ls**. Es werden nun im zeitlichen Verlauf die wesentlichen Schritte behandelt, die für den Ablauf des Szenarios wichtig sind.

1. **Befehlseingabe `ls -lR / >ausgabe`**
   Durch die Befehlseingabe mittels Tastatur erfolgen schon erste **Interrupts** anderer nebenläufiger Prozesse. Dies hat auf den **SHELL**- und **Init**-prozess keinen Einfluss. Der Shell Prozess befindet sich im Zustand *user running*. Die Bearbeitung der Tastatureingaben erfolgt über die Standardeingabe *STDIN*

2. ***ENTER* bzw. `\n`**
   Durch das Drücken der *ENTER*-Taste und der damit verbundenem Eingabe `\n` führt die **SHELL** den Befehl aus und der **SHELL**-Prozess erzeugt so ein **ls**-Kindprozess (Zustand: zunächst *kernel running*, da Kopie; später *user running*)mittels dem Systemaufruf `fork()`. Der Shell Prozess wechselt in den Zustand *kernel running*. Da der Prozess im Vordergrund gestartet wird handelt es sich um ein **Trap** durch den der **SHELL**-Prozess aussetzt und der Benutzer keine gewöhnlich Eingabe mehr an die **SHELL** direkt geben kann. Der *User-Mode* wird verlassen und der *Kernel-Mode* wird betreten. Dabei wird der User-Stackpointer und das dazugehörige Register auf den Kernel-Stack gerettet. Der **SHELL**-Prozess wird mittels der Funktion `sleep()` angehalten und kommt in die Sleep-Queue. Ab diesem

Zeitpunkt wartet der **SHELL**-Prozess auf die Terminierung des Kindprozesses **ls**. Der Shell Prozess hat den Zustand *asleep*

3. **Ablauf des ls-Prozesses im Vordergrund**
   Während des Programmablaufs bei dem es sich um ein **Trap** handelt gibt es wieder permanent (z.B. clock) vom Benutzer nicht bemerkbare **Interrupts** innerhalb des Kernels.

4. **Strg-Z**
   Durch das Drücken dieser Tastenkombination wird vom Interrupthandler ein *SIGSTOP*-**Signal** ausgelöst. Der Interrupthandler sorgt mittels der Funktion `psignal()` dafür, dass der Prozess **ls** das Signal behandelt, welches einen Ausnahmezustand des Prozesses darstellt. Der **ls** Prozess wechselt nicht in den Zustand *stopped*. Durch das **Signal** erfolgt ein **Trap** durch welches der **ls**-Prozess gestoppt wird (**ls** ist *kernel running*). Durch `sleep()` wird der Prozess angehalten (Zustand *asleep*) und durch `qswitch()` an die Sleep-Queue gegeben. Dahingegen wird der **SHELL**-Prozess mittels `wakeup()` wieder aufgeweckt (Zustand jetzt *ready to run*) und ist wieder Aktiv. Er wechselt mittels `qswitch()` wieder in die Run-Queue. (Zustand danach *kernel running/user running*)

5. **Eingabe und *ENTER* des Befehls `bg`**
   Durch die Eingabe des Befehls erfolgen **Interrupts** anderer Prozesse. Durch das Drücken der *ENTER*-Taste und der damit verbundenem Eingabe `\n` führt die **SHELL** den Befehl aus und der Kindprozess **ls** wird im Hintergrund fortgesetzt. (Zustand *user runnig*) Es erfolgt hier wieder im Kerner des Wechsel in die Run-Queue.

6. Ablauf des **ls**-Prozesses im Hintergrund Während des Ablaufs im Hintergrund erfolgen permanent **Interrupts** des **ls**- und des **SHELL**-Prozesses.

7. Terminierung des **ls**-Prozesses im Hintergrund
   Durch die Terminierung des **ls**-Prozesses wird ein *SIGCHLD*-Signal an den **INIT**-Prozess gesendet. Der **INIT**-Prozess gibt die proc-Struktur des **ls**-Prozesses frei. Der **ls**-Prozess existiert nicht mehr.

Aufgabe 2

Bei der Lösung der Aufgabe 2 wurde sich grob an dem Beispiel aus dem *Linux Programming Blog*[1] gehalten.

```
1  #include <stdio.h>
2  #include <iostream>
3  #include <unistd.h>
4  #include <signal.h>
5  #include <string.h>
6
7  void sig_handler(int signo, siginfo_t *sinfo, void *context)
8  {
9      std::cout << "Signal number: " << signo << "\tProcess ID: " << sinfo->si_pid
           << "\t User ID: " << sinfo->si_uid << std::endl << std::flush;
10 }
11
12 int main()
13 {
14     struct sigaction a;
15     memset(&a,'\0',sizeof(a));
16     a.sa_sigaction = &sig_handler;
17     a.sa_flags = SA_SIGINFO;
18     if (sigaction(SIGUSR1, &a, NULL) < 0) {
19         perror ("Could not register Sighandler");
20         return 1;
21     }
22     while(1)
23     {
24         pause();
25     }
26     return 0;
27 }
```

In der *main()*-Methode wird zuerst der *struct* **sigaction** initiirt und mit nullen gefüllt. Danach wird die *Callback*-Funktion und der Typ in dem *struct* gesetzt. Danach wird das Signal *SIGUSR1* an das *struct* gebunden. Sollte dies Fehlschlagen wird eine Fehlermeldung mit dem Fehler ausgegeben. Danach geht das Programm in eine Dauerschleife, in der es auf ein Signal wartet. In der *Callback*-Funktion **sig_handler** wird die Nummer des empfangenen Signals, die Process ID des Prozesses, der das Signal ausgelöst hat, sowie die User ID des Auslöser-Process-Besitzers.

Aufgabe 3

Aufgabe 3a)

Es könnte zu Effekt des Alterns kommen. Das bedeutet, dass Prozesse die schon lange in Ausführung waren einen immer höheren CPU Kontostand bekommen. Dadurch kann es vorkommen, das diese Prozesse nur sehr selten eine Zeitscheibe erhalten, wenn oft neue Prozesse gestartet werden (was i.d.R. der Fall ist). Dieser Effekt kann auch auftreten, obwohl der Prozess ggf. eine Hohe Priorität (geringer *nice*-Wert) hat.

Aufgabe 3b)

Zu Beginn wird Prozess B ausgewählt. Dafür wird die Berechnung exemplarisch durchgeführt. Für die weiteren Zeitscheiben ist die Berechnung äquivalent.

[1] https://www.linuxprogrammingblog.com/code-examples/sigaction

$$Summe_0A = Konto_0A + Priorität_A = 0 + 56 = 56$$
$$Summe_0B = Konto_0B + Priorität_B = 0 + 10 = 10$$

Prozess B wird ausgewählt und für die Zeitscheibe ausgeführt. Der neue Kontostand berechnet sich wie folgt:

$$Konto_1A = \frac{Konto_0A}{2} = \frac{0}{2} = 0$$
$$Konto_1B = \frac{Konto_0B+100}{2} = \frac{100}{2} = 50$$

Nachdem Prozess B beginnt stellt sich, wie auf folgender Tabelle ersichtlich, das Muster **ABB** ein. Somit hat B für lange Laufzeit (fast) doppelt so lange die CPU wie A (die angegeben Prozesse haben jeweils 100% CPU Zeit). Zur Berechnung der gelten folgende *EXCEL* bzw. *LibreOffice* Formeln (Beispiel für Spalte C, Prozess A):

KONTO(C) = ABRUNDEN(WENN(B11 = „A “; (B3+100)/2; B3/2))
SUMME(C) = ABRUNDEN(SUMME(C3;C4))
PROZESS(C) = WENN(C5 <= C9; „A“;„B“)

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

Aufgabe 3c)

Für den folgenden Aufgabenteil gilt folgende Notation:
$P_{Prozess}$:= Basispriorität; $S(n)_{Prozess}$:= Summe eines Prozesses zum Zeitpunkt n; n := Index/-Zeitpunkt der Zeitscheibe, beginnt bei 0
Die Tabelle aus Aufgabenteil b lässt sich auch hier anwenden. Allerdings haben sich die Basisprioritäten geändert:

Scheibe	0	1	2	3	4	5	6	7	8	9	10	11
Konto A	0	50	75	87	93	96	98	99	99	99	99	99
Priorität A	0	0	0	0	0	0	0	0	0	0	0	0
Summe A	0	50	75	87	93	96	98	99	99	99	99	99
Konto B	0	0	0	0	0	0	0	0	0	0	0	0
Priorität B	99	99	99	99	99	99	99	99	99	99	99	99
Summe B	99	99	99	99	99	99	99	99	99	99	99	99
Prozess	A	A	A	A	A	A	A	A	A	A	A	A

Es lässt sich erkennen, dass B in den ersten 12 Zeitschreiben nie CPU erhält, obwohl $P_B = 99$.

Weiterhin lässt sich erkennen, dass $Summe_A$ gegen 99 strebt, solange nur A die CPU bekommt. Dies lässt sich mithilfe folgender Berechnung beweisen:

Unter der Annahme, dass die Summen nicht abgerundet werden, lässt sich folgende Beobachtung für die Summe von A machen:

$$S(n)_A = \frac{100+S_A(n-1)}{2}$$

Also ergibt sich folgende Folge (mit $S(0)_A = 0$):

$$0; 50; 75; 87,5....$$

Diese Folge lässt sich durch die Summe

$$S(n)_A = 0 + \frac{50}{1} + \frac{50}{2} + \frac{50}{4} + ...$$

$$S(n)_A = \sum_{i=0}^{n} \frac{50}{2^i} = 50 \cdot \sum_{i=0}^{n} \frac{1}{2^i} = 50 \cdot \sum_{i=0}^{n} (\frac{1}{2})^i = 50$$

ausdrücken. Diese Summe kann mit der geometrischen Reihe ausgerechnet werden:

$$a_0 \cdot \sum_{i=0}^{k} q^i = a_0 \cdot \frac{1-q^{k+1}}{1-q}$$

$$S(n)_A = 50 \cdot \sum_{i=0}^{n} (\frac{1}{2})^i = 50 \cdot \frac{1-\frac{1}{2}^{n+1}}{1-\frac{1}{2}}$$

Für sehr große n strebt $S(n)_A$ gegen 100. Da die Nachkommastellen aber abgeschnitten werden erreicht $S(n)_A$ den Wert 99. Somit muss $P_B \geq 99$ damit $S(n)_A \leq S(n)_B \ \forall n$ und Prozess B niemals die CPU bekommt. Es reicht jedoch wenn $P_B = 99$, da dann für große n beide Prozesse die gleiche Summe haben und dann laut Konvention Prozess A die CPU bekommt.

Aufgabe 3d)

Aufgabe 3d ist wesentlich komplexer als die bisherigen. Da Prozess B sich für 250ms blokiert kann es passieren, dass Prozess A die CPU bekommt, obwohl A den höheren *nice* Wert hat bzw. die niedrigere Priorität. Dies kann auch während einer Zeitscheibe passieren da B die Run-Queue verlässt. Weiterhin kann es passieren, dass B sich nach ablauf der 250ms wieder in die Run-Queue einreiht und aufgrund einer höheren Priorität sofort die CPU bekommt.
In den ersten beiden Tabellen sind die ersten 6 Zeitscheiben detailliert aufgelistet.
(Bemerkung: 145ms A und 55ms B = 72,5% A und 27,5% B) Es lässt sich erkennen, dass Prozess B für die ersten 12 Scheiben wahrscheinlich eine bessere Priorität haben wird als A. Dies ist auch logisch, da B immer nur maximal 55ms CPU Zeit hat und A die übrigen 145ms. Deshalb verzichten wir auf die detaillierte Berechnung der Summen/Prioritäten (die nur Fleißarbeit wäre) und stellen im folgenden das Sheduling für die restlichen 6 Scheiben dar (unter der Voraussetzung, dass B immer die höhere Priorität hat)

Scheibe	0	1	2	3	4	5
Konto A	0	50	(72,5+50)/2=61	(72,5+61)/2 = 66	(100+66)/2 = 83	(72,5+83)/2= 77
Priorität A	0	0	0	0	0	0
Summe A	0	50	61	66	83	77
Konto B	0	0	(27,5+0)/2 = 13	(27,5+13)/2=20	(0+20)/2 =10	(27,5+10)/2 = 18
Priorität B	15	15	15	15	15	15
Summe B	15	15	28	35	35	33
Prozess	200ms A	145ms A	105ms A	200ms A	10ms A	115ms A
		55ms B	55ms B		55ms B	55ms B
			40ms A		135ms A	30ms A

Scheibe	Prozess
6	200ms A
7	20ms A, 55ms B, 125ms A
8	145ms A, 55ms B, 20ms A
9	200ms A
10	30ms A, 55ms B, 115ms A
11	125ms A, 55ms B, 20ms A