Technische Informatik 2 | WS 2016/17 | C-01
Tutor: Marc/Bingbin

Habermann, Paul
Köster, Joschka
Rohde, Florian

Übungsblatt 3

1	2	3	4	5	$\sum$

Lösungsvorschlag
Abgabe: 21.11.2016

1 Aufgabe 1

- Der Befehl CTRL-Z loest einen Interrupt aus und
  - sendet das Signal SIGSTP (20) an das Programm, um es zu pausieren
- Der Befehl bg sendet SIGCONT (18), um das Programm fortzusetzen
- Falls die maximale Dateigroesse ueberschritten wird, wird ein SIGXFSZ ausgelöst
- Bei der Eingabe des Befehls ls -lR in die Shell loest diese einen Trap aus (Systemaufruf)

2 Aufgabe 2

2.1 Signalhandler

Listing 1: sigserver.cc.sh

```
6  // extract signals id, sender-pid und uid and print it to console
7  static void handleSignal(int sig, siginfo_t *sigt, void *v)
8  {
9  ___printf("Received signal #%d from PID %ld, running on UID %ld \n",
10 _____sig, (long)sigt->si_pid, (long)sigt->si_uid);
11 }
```

Wir haben einen Signalhandler gebaut, der als Parameter die drei Rueckgaben vom `sigaction` bekommt und daraus eine triviale Konsolenausgabe gemaess Aufgabenstellung baut.

2.2 main

Listing 2: sigserver.cc.sh

```
15 // initiate sigaction-listener and wait
16 int main(int argc, char *argv[])
17 {
18
19 ___printf("My PID is: %lu \n", (unsigned long)getpid());
20
21 ___struct sigaction sa;
22
23 ___sa.sa_flags = SA_SIGINFO;
24 ___sa.sa_sigaction = handleSignal;
25
26 ___sigaction(SIGUSR1, &sa, &sa); //&sa, NULL);
27
28 _____pause();
29
30 ___return 0;
31 }
```

Unsere main-Methode liefert zuerst die eigene PID zurueck, damit wir direkt den `kill`-Befehl darauf absetzen koennen. Dann initialisieren wir unser `sigaction`, setzen die Flags und den Namen der Methode, die ein Signal verarbeiten soll und registrieren zuletzt den "Listener"fuer das Signal SIGUSR1. Die Methode pause() laesst das Programm so lange warten, bis ein Signal gesendet wird. Danach beendet es sich mit dem Exit-Code 0.

2.3 Tests

2.3.1 Standard-Aufruf

Wir starten unser Programm und lassen es mittels `bg` im Hintergrund laufen. Dann senden wir das Signal SIGUSR1 an den Prozess:

```
x13->./sigserver
My PID is: 8767
^Z
[1]+ Stopped ./sigserver

x13->bg
[1]+ ./sigserver &

x13->kill -SIGUSR1 8767
Received signal #10 from PID 24655, running on UID 14104
```

2.3.2 Aufruf mit relativer Prozess-ID

Wir starten unser Programm und lassen es mittels `bg` im Hintergrund laufen. Dann senden wir das Signal SIGUSR1 an die relative Prozess-ID:

```
x13->./sigserver
My PID is: 9597
^Z
[1]+ Stopped ./sigserver

x13->bg
[1]+ ./sigserver &

x13->kill -SIGUSR1 %1
Received signal #10 from PID 24655, running on UID 14104
```

2.3.3 Direkter Aufruf im Hintergrund

Wir starten unser Programm und lassen es mittels `&` im Hintergrund laufen. Dann senden wir das Signal SIGUSR1 an die Prozess-ID:

```
x13->./sigserver &
[1] 9764
x13->My PID is: 9764
kill -SIGUSR1 9764
Received signal #10 from PID 24655, running on UID 14104
```

2.3.4 Verwendung von Signal-ID 10

Da `SIGUSR1` in der Dezimaldarstellung eine 10 ist, ist es eigendlich unnoetig - dennoch haben wir alle Tests mit dem Aufruf `kill -10 (PID)` wiederholt.
Da die Ergebnisse absolut gleich sind, verzichten wir hier auf eine doppelte Dokumentation.

3 Aufgabe 3

3.1 A)

Man erkennt, dass hier jeweils immer nur abwechselnd Sheduled wird. Dadurch, dass nicht geteilt wird, werden Kontostände einfach immer nur um 100 „Punkte“ erhöht. Dadruch ist es natürlich logisch, dass die Prozesse immer abwechselnd laufen. Dies variiert auch keineswegs.

Konto A	akt. Wert	0	0	0	100	100
Nutzung A	100o0	0	0	100	0	100
Konto A’	trunc(konto+nutz/2)	0	0	100	100	200
Prio A	basisprio+konto’	56	56	156	156	256
Konto B	akt Wert	0	0	100	100	200
Nutzung	100o0	0	100	0	100	0
Konto B’	trunc(konto+nutz/2)	0	100	100	200	200
Prio B	bprio+kont’	10	110	110	210	210

3.2 B)

Konto A		0	0	0	50	25	12	56	28	14	57	28	14	57	...
Nutzung		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	...
ab hier: wiederkehrendes Muster(3f. auf 10ter Takt)															
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	82	95	52	82	...

3.3 C)

Konto A	0	0	50	75	87	93	96	98	99	99
Nutzung A	0	100	100	100	100	100	100	100	100	199
Konto A’	0	50	75	87	93	96	98	99	99	99
Prio A	0	50	75	87	93	96	98	99	99	99
Konto B	0	0	0	0	0	0	0	0	0	0
Nutzung B	0	0	0	0	0	0	0	0	0	0
Konto B’	0	0	0	0	0	0	0	0	0	0
Prio B	99	99	99	99	99	99	99	99	99	99

Wir kommen in eine Endlosschleife wo immer A gewählt wird, da wir immer bei 99,5 die 5 wegschneiden. Wir wählen A aus, da wir alphabetisch vorgehen(Tutorium). Am Ende sieht man, dass bei 99 immer wieder alles gleich aussieht es wird sich also in der Matrix nichts verändern. Desweiteren wird nun 98 untersucht, Hier nehmen wir an, dass wir ein Wechsel bekommen. Dadurch ist bewiesen, dass es keine niendiger Zahl mehr gibt, wo B komplett ignoriert wird.

Prio99:

Konto A	0	0	50	75	87	93	96	98	99
Nutzung A	0	100	100	100	100	100	100	100	0
Konto A'	0	50	75	87	93	96	98	99	49
Prio A	0	50	75	87	93	96	98	99	49
Konto B	0	0	0	0	0	0	0	0	0
Nutzung B	0	0	0	0	0	0	0	0	100
Konto B'	0	0	0	0	0	0	0	0	50
Prio B	98	98	98	98	98	98	98	98	98

Man erkennt, dass hier der Scheduler einen Lauf an b abgibt. Deswegen stimmt es dass die Scheduling Zahl(nice-Wert[BasisPrio]) immer <99 sein muss in diesem Algorithmus.

3.4 D)

Da wir hier nur 55 ms des Takts laufen, liegt die Nutzung bei $\frac{55}{250}$ also 0,22 was 2% ergibt.

Konto A		0	0	57	67	83	80	90	84	92	85	92	85	92	...
Nutzung			100	78	100	78	100	78	100	78	100	78	100	78	...
Konto A'		0	57	67	83	80	90	84	92	85	92	85	92	85	...
Prio A		15	72	82	98	95	105	99	107	100	107	100	107	100	...
Konto B		0	0	0	11	5	13	6	14	7	14	7	14	7	...
Nutzung B		0	0	22	0	22	0	22	0	22	0	22	0	22	...
Konto B'		0	0	11	5	13	6	14	7	14	7	14	7	14	...
Prio B		0	0	11	5	13	6	14	7	14	7	14	7	14	...