Technische Informatik 2
Tutor: Marc Hildebrandt

WS 2016/17

C08

Timo Jasper (Inf, 3.FS.)
Thomas Tannous (Inf, 3.FS.)
Oliver Hilbrecht (Inf, 3.FS.)
Moritz Gerken (Inf, 3.FS.)

Übungsblatt 3

1	2	3	4	5	$\sum$

Lösungsvorschlag
Abgabe: 21.11.2016

Aufgabe 1

Der Prozess 'ls -lR / >ausgabe', welcher im foreground im User-Mode läuft, wird durch das Drücken von Strg+z aufgehalten, indem das Signal SIGSTOP gesendet wird. Im gleichen Moment wird durch das drücken der Tasten, der aktuelle Prozess unterbrochen (interrupted), da der Computer den input der Tastatur verarbeiten muss. ls listet aus dem root Verzeichnis rekursiv alle Dateiein und ruft somit read auf, was einen Systemaufruf darstellt und den Prozess in die Kernel-Routine bringt (Trap). Durch die Rekursion werden Verzeichnisse aufgerufen, so kommt es auch zum Systemaufruf der funktion open bzw. readdir (trap). Falls die Datei ausgabe noch nicht erstellt wird sie erstellt durch den Systemaufruf open (trap). In die Datei wird geschrieben mit dem Systemaufruf write (trap).

Aufgabe 2

Die Aufgabe war es einen Handler für das Signal USR1 zu erstellen. Die Funktion *handle_signal()* wird aufgerufen, sobald ein USR1 Signal empfangen wird. Diese kriegt dann Informationen über den Sender des Signals über *siginfo_t*.

```
#include <stdio.h>
#include <string.h>
#include "signal.h"
#include <unistd.h>
```

```
void handle_signal(int signum, siginfo_t *siginfo, void *context );

int main() {

    struct sigaction sigact;

    sigact.sa_flags = SA_SIGINFO;
    sigact.sa_sigaction = handle_signal;
```

```
    if( sigaction(SIGUSR1,&sigact, NULL) == -1) {
        perror("Error, cannot handle SIGUSR1");
    }
    pause();

    return 0;
}


void handle_signal(int signum, siginfo_t *siginfo, void *context) {
    printf("SIGNAL NUMBER: %i \n",siginfo->si_signo);
    printf("Process ID: %i \n", siginfo->si_pid);
    printf("User ID: %i \n",siginfo->si_uid);

}
```

Aufgabe 3

3.a)

Große Prozesse mit langer Laufzeit würden ihren Kontostand trotz möglicherweise Hoher Priorität schnell erhöhen und dann 'verhungern', da ihr Kontostand sie daran hindern würde einen niedrigen Wert bei der Effektiven Priorität zu erlangen. Prozesse mit kurzer laufzeit würden davon profitieren, da sie nur wenige Zeitscheiben benötigen um abgearbeitet zu sein.

3.b)

Prozess A (nice = 56)												
Voriges CPU-Konto		0	0	50	25	63	32	16	58	29	15	58
⇓ + Aufschlag (%CPU)		0	100	50	125	63	32	116	58	29	115	58
⇓ Veraltern (/2)												
Aktuelles CPU-Konto		0	50	25	63	32	16	58	29	15	58	29
⇓ + Basispriorität												
Effektive Priorität	56	56	105	81	119	88	72	114	85	71	113	85
Prozess B (nice = 10)												
Voriges CPU-Konto		0	50	25	63	32	66	83	42	71	86	43
⇓ + Aufschlag (%CPU)		100	50	125	63	132	166	83	142	171	86	143
⇓ Veraltern (/2)												
Aktuelles CPU-Konto		50	25	63	32	66	83	42	71	86	43	72
⇓ + Basispriorität												
Effektive Priorität	10	60	35	98	42	76	93	52	81	96	53	82

Nach der einfädelung in den ersten 2 Zeitscheiben, sieht es so aus, als würde Prozess B doppelt so oft den Prozessor zugesprochen bekommen, wie Prozess A.

Es wurde auf ganze zahlen gerundet, da wir ansonsten unnötig große Nachkommazahlen erhalten würden.

3.c)

Bei dem Algorithmus dieser Aufgabenstellung ist in der unten stehenden Tabelle leicht erkennbar, dass sich bei Prozess A (ggf. er läuft ununterbrochen) die 100 als CPU-Konto-Wert einpendelt. Wenn Prozess B die Basisprioriät 100 zugeteilt würde, dann kähme er von Beginn an niemals an eine Zeitscheibe heran, da bei gleicher Effektiven Priorität aufgrund seines Namens (Alphabetische Ordnung) Prozess A immernoch bevorzugt werden würde. Bei 99 und weniger Prioritäts-Wert hätte Prozess B den Prozessor wenigstens einmal im 8. Durchlauf (Prozess A stand zum erstenmal auf 100) erhalten. Eine geringere Basispriorität als die von 100 wird nach Konvention nicht vergeben.

Prozess A (nice = 0)												
Voriges CPU-Konto		0	50	75	88	94	97	99	100	100	100	100	...
⇓ + Aufschlag (%CPU)		100	150	175	188	194	197	199	200	200	200	200	...
⇓ Veraltern (/2)													
Aktuelles CPU-Konto		50	75	88	94	97	99	100	100	100	100	100	...
⇓ + Basispriorität													
Effektive Priorität	0	50	75	88	94	97	99	100	100	100	100	100	...

Es wurde auf ganze zahlen gerundet, da wir ansonsten unnötig große Nachkommazahlen erhalten würden.

3.d)

Prozess A (nice = 0)												
Voriges CPU-Konto		0	50	61	81	77	89	81	91	82	91	82
⇓ + Aufschlag (%CPU)		100	122	161	153	177	161	181	163	182	163	182
⇓ Veraltern (/2)												
Aktuelles CPU-Konto		50	61	81	77	89	81	91	82	91	82	91
⇓ + Basispriorität												
Effektive Priorität	0	50	61	81	77	89	81	91	82	91	82	91
Prozess B (nice = 15)												
Voriges CPU-Konto		0	0	14	7	33	17	23	12	20	10	19
⇓ + Aufschlag (%CPU)		0	28	14	35	33	45	23	40	20	38	19
⇓ Veraltern (/2)												
Aktuelles CPU-Konto		0	14	7	18	17	23	12	20	10	19	10
⇓ + Basispriorität												
Effektive Priorität	15	15	29	22	33	32	38	27	35	25	34	38

Wenn Prozess B die Zeitscheibe erhält, arbeitet er 55 ms und gibt die Zeitscheibe ab (wir gehen in dieser Rechung davon aus, dass die neue Verteilung 0 ms in Anspruch nimmt), daher verliert Prozess B bereits in der selben Zeitscheibe 145 ms seiner Sperrzeit von 250 ms Vorrausgesetzt Prozess B hat nur mit Prozessen zu tun, die ihre Zeitscheiben voll ausnutzen ist er effektiv für eine weitere Zeitscheibe blockiert, wenn er write() aufgerufen hat. Prozess B nimmt pro erhaltener Zeittafel 28% von dieser ein, die restlichen 72% bekommt der nächst Priorisierte Prozess

zugesprochen in diesem Fall immer Prozess A.

Es wurde auf ganze zahlen gerundet, da wir ansonsten unnötig große Nachkommazahlen erhalten würden.

Weitere Aufgaben