Technische Informatik 2 WS 2016/17 C04
Tutor: Marc Hildebrandt/Bingbin Yu

Michael Schmidt
Stanislav Telis
Dominique Schulz
Norman Lipkow

Übungsblatt 3

1	2	3	4	5	$\sum$

Lösungsvorschlag
Abgabe: 21.11.2016

Aufgabe 1

Die Shell erzeugt zunächst mit `fork()` ein Kindprozess, wodurch ein Trap ausgelöst wird, da fork() ein Systemaufruf ist und die Kontrolle während der Erzeugung an den Kern geht. Der Kindprozess führt dann mit `exec()` das Kommando `ls -1R / >ausgabe` aus (hierbei wird erneut ein Trap ausgelöst (da `exec()` ein systemcall ist) und lässt diesen im Vordergrund laufen und die Shell (Vaterprozess) geht mit `wait()` von `user running` in den `asleep`-Zustand und wartet somit auf Terminierung des Hintergrundprozesses `ls`.

Der Kindprozess `ls` läuft nun im Vordergrund. Zum Lesen der Dateien aus dem `root`-Verzeichnis und den rekursiv darunter liegenden Unterverzeichnisse wird mit `read()` ein systemcall, also ein Trap, ausgeführt und es wird in den Kern-Modus gewechselt. Das Betriebssystem liest nun die benötigten Informationen aus den Dateien. Dies passiert solange bis die Benutzerin die Tastenkombination `Strg-Z` gedrückt hat. Hier wird nun ein Interrupt ausgeführt, der Prozess `ls` wird unterbrochen und die CPU wird vom Interrupt-Handler „ausgeliehen“der im Kern läuft. Der Interrupt-Handler verarbeitet die Tastenkombination des Benutzers. Der Interrupt-Handler liefert nun das Signal `SIGSTOP` an den `ls`-Prozess und stoppt diesen damit. Im Anschluss liefert der `ls`-Prozess seinem Vaterprozess, der Shell, das Signal `SIGCHILD` aus und der Shell-Prozess läuft weiter. Beim Shell-Kommando `bg` erzeugt die Shell einen neuen Prozess. Der Ablauf zur Prozesserzeugung ist der gleiche Ablauf wie beim `ls`-Kommando (gleiche Aufrufe und Traps) und dieser `bg`-Prozess sendet dem `ls`-Prozess ein `SIGCONT` damit dieser im Hintergrund weiter laufen kann und damit kehrt er in den `user running` Zustand zurück. Der `bg`-Prozess ist damit beendet und dieser führt dann `exit()` aus und schickt damit ein `SIGCHILD`-Signal an seinen Vaterprozess und geht von `running` in den `zombie`-Zustand. Nach Ausführung des `ls`-Prozesses, ruft dieser ebenfalls `exit()` auf mit gleichem Ablauf wie vorhin beim `bg`-Prozess beschrieben

Im Folgenden haben wir der Übersicht halber den Verlauf stichpunktartig zusammengefasst:

1. Shell erzeugt Kindprozess (mit `fork()` → Trap, da systemcall) → Zustand von Shell-Prozess von `running` zu `asleep` (da `ls` im Vordergrund) → Shell hat `wait()` aufgerufen und wartet auf Terminierung des Kindprozesses.
2. Kindprozess führt `exec()` auf (Trap, da systemcall) → `ls` will Dateien lesen (`read()` → systemcall → Trap → Kernmodus)
3. Interrupt von Tastaturkombination `Strg-Z` → Signal `SIGSTOP` an `ls`-Prozess → Zustand von `ls`-Prozess von `running` zu `stopped`
4. Signal `SIGCHILD` von ls-Prozess an Shell-Vaterprozess → Zustand von Shell-Prozess geht zurück auf `running`
5. Kindprozesserzeugung wie oben beschrieben für das `bg`-Kommando, das ein `SIGCONT`-Signal an den gestoppten `ls`-Prozess sendet der nun im Hintergrund weiterläuft.

6. Beim Schreiben der Daten in die Ausgabedatei `ausgabe` wird ein weiterer Trap(systemcall `write()`) ausgeführt → Auch hier wird in den Kern-Modus gesprungen.
7. Am Ende wird `exit()` vom `ls`-Prozess aufgerufen → Signal `SIGCHILD` an Shell-Prozess und `ls`-Prozess geht von `running` in den `zombie`-Zustand.

Aufgabe 2

Im Folgenden unsere Implementierung für den `sigserver`

sigserver.cc

```
#include <iostream>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <cerrno>

/*
 *  Der Signalhandler, der das empfangene Signal "behandelt" und
 *  die Signalnummer, die Prozess-ID und die User-ID des sendenden
 *  Prozesses ausgibt.
*/
void sighandler(int sig, siginfo_t *siginfo, void *unused) {
        std::cout << "Signal_number: " << siginfo->si_signo << std::endl;
        std::cout << "Process_ID: " << siginfo->si_pid << std::endl;
        std::cout << "User_ID: " <<  siginfo->si_uid << std::endl;
}

int main() {
        // Die Struktur des sigaction()-Systemaufrufs
        struct sigaction act;

        // Hier wird der Funktion des Signalhandlers gesetzt
        act.sa_sigaction = &sighandler;

        /* Hier wird festgelegt, dass sa_sigaction als
         * Signalhandler gesetzt wird
        */
        act.sa_flags = SA_SIGINFO;

        if (sigaction(SIGUSR1, &act, NULL) == -1) {
                std::cerr << strerror(errno) << std::endl;
                return 1;
        }

        // Endlosschleife... Warten auf Signal
        std::cout << "Waiting for signal..." << std::endl;
        if(pause() == -1) {
          std::cerr << strerror(errno) << std::endl;
        }

        return 0;
 }
```

Tests:

Wir haben unser Programm auf dem Linux-Rechner x20 im MZH getestet

Vorab lassen wir uns unsere User-ID ausgeben, damit wir diese mit unserer späteren Ausgabe vergleichen können:

bash-Prozess 1

```
1 x20->id -u nlipkow
2 153599
```

Unsere User-ID ist also: `153599`

Im folgenden kompilieren und führen wir unser Programm aus:

bash-Prozess 1

```
3 x20->g++ -o sigserver sigserver.cc
4 x20->./sigserver
5 Waiting for signal...
```

Wir sehen in dem Linux-Programm *System Monitor* nach, welche Prozess-ID der gestartete Prozess `sigserver` hat.

Wir öffnen ein zweites Terminal und terminieren den `sigserver`-Prozess mit dem `kill`-Kommando, senden ein `SIGUSR1`-Signal und lassen es im Hintergrund laufen damit wir auch die Prozess-ID wiedergegeben bekommen um die Richtigkeit unserer Ausgabe vom `sigserver` zu testen:

bash-Prozess 2

```
1 x20->kill -SIGUSR1 31140 &
2 [1] 31167
```

Das heißt, der `kill`-Prozess hatte die ID: `31167`

In der Shell in dem der `sigserver`-Prozess im Vordergrund lief erhalten wir folgende Ausgabe:

bash-Prozess 1

```
6 Signal_number: 10
7 Process_ID: 31167
8 User_ID: 153599
9 Interrupted system call
```

Wir sehen, dass die ausgegebene Prozess-ID mit der ID unseres sendenden kill-Kommando, sowie unsere User-ID mit der Ausgabe übereinstimmt.

Die Fehlermeldung am Ende wurde von der `pause()`-Methode ausgegeben, da diese mit der Terminierung ja unterbrochen wurde.

Aufgabe 3

a)

Langlebige Prozesse hätten einen großen Nachteil wenn sich deren Zeitkonto nicht mehr verringern würde, da der Wert irgendwann so groß sein würde, dass sie kaum bis gar nicht mehr dran kämen. Und neue Prozesse die wohlmöglich eine geringe Priorität haben würden irgendwann immer bevorzugt werden.

b)

Zeitscheibe	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

In den grün markierten Spalten sieht man, welcher Prozess in welcher Zeitscheibe die CPU bekommen hat.

Ab dem 3. Schritt erkennt man, dass die CPU in einem 2:1 Muster verteilt ist. D.h. Prozess A erhält nur in jedem 3.Schritt die CPU, während Prozess B 2 von 3 Zeitscheiben die CPU erhält.

c)

Zeitscheibe	0	1	2	3	4	5	6	7	8	9
Konto(A)	0	0	50	75	87	93	96	98	99	99
Nutzung(A)	0	100	100	100	100	100	100	100	100	100
Konto'(A)	0	50	75	87	93	96	98	99	99	99
Prio(A)	0	50	75	87	93	96	98	99	99	99

Nach der 7. Zeitscheibe ändert sich die Priorität von Prozess A nicht mehr und stagniert auf der Priorität 99. Da bei gleichwertigem Prioritätswert die Prozesse nach dem Alphabet bevorzugt werden, würde Prozess B ab einer Basispriorität von 99 die CPU in diesem Szenario nie erhalten.

d)

Zeitscheibe	0	1	2	3	4	5	6	7	8	9	10	11	12
Konto(A)	0	0	50	61	80	76	88	80	90	81	90	81	90
Nutzung(A)	0	100	73	100	73	100	73	100	73	100	73	100	73
Konto'(A)	0	50	61	80	76	88	80	90	81	90	81	90	81
Prio(A)	0	50	61	80	76	88	80	90	81	90	81	90	81
Konto(B)	0	0	0	13	6	16	8	17	8	17	8	17	8
Nutzung(B)	0	0	27	0	27	0	27	0	27	0	27	0	27
Konto'(B)	0	0	13	6	16	8	17	8	17	8	17	8	17
Prio(B)	15	15	28	21	31	23	32	23	32	23	32	23	32

Hat Prozess B die höchste Priorität, so arbeitet dieser für 55 ms. In einer 200 ms Zeitscheibe sind das 27%(da $\frac{55}{200} = 0,275$). Danach ist dieser für 250 ms blockiert, also für den Rest der aktuellen Zeitscheibe gibt er die CPU an Prozess A ab. Im Laufe der nächsten Zeitscheibe wird Prozess B zwar wieder frei, muss aber bis zum nächsten Scheduling/Zeitscheibe warten und kommt somit für die komplette Zeitscheibe nicht zum Zug (rot markierte Zellen).