Institut für Informatik

der Ludwig-Maximilians-Universität München

Systempraktikum – Wintersemester 2011/2012

Prof. Dr. Dieter Kranzlmüller
Dr. Nils gentschen Felde, Stephan Reiter, Johannes Watzl

Blatt 4— Grundlagen III: Prozesserzeugung, Prozessüberlagerung, Pipes

Abgabedatum theor. Aufgaben	Abgabedatum prakt. Aufgaben	Deadline Projektaufgaben
13.11.	13.11.	—

Hinweise

- Am 14.11. findet eine Einheit zur freiwilligen Wiederholung des Stoffes statt.
- Am Montag, 21.11., um 18.00 Uhr s.t. findet die schriftliche Zwischenklausur in den Räumen A140 und M218 im LMU-Hauptgebäude statt.
- In der Vorlesung am Montag, 28.11., erfolgt die Einteilung der Programmierteams. Es besteht also Anwesenheitspflicht.

Theoretische Aufgaben (Blatt 4)

Aufgabe T-4-1

Zeichnen Sie das 7-Zustands-Prozessmodell in Form eines endlichen Automaten (d.h. Zustände als Knoten, Übergänge als gerichtete Kanten/Pfeile). Nummerieren Sie alle Übergänge, und erstellen Sie eine vollständige Tabelle der Form:

Nr.	Übergang	Beschreibung	Beispiel
...	...	...	...
2	Ready → Running	Scheduler wählt Prozess	Der Prozess mit der höchsten Priorität wird zur Ausführung ausgewählt (Priority-Scheduling)
...	...	...	...

Aufgabe T-4-2

Im Zusammenhang mit untereinander verwandten Unix-Prozessen unterscheidet man spezielle Konfigurationen:

a. Was ist ein Zombie-Prozess, und wie kann er entstehen?

b. Wann spricht man im Kontext von Prozessen von einem Waisen? Was geschieht mit ihm, wenn er entstanden ist?

c. Analysieren Sie die folgenden vier Programme. Erklären Sie die Unterschiede, und beschreiben Sie die Folgen daraus. Finden Sie heraus, wo ein Zombie und wo ein Waise deutlich sichtbar wird. Überprüfen Sie

Ihre Vermutungen durch Übersetzen und Testen der Programme! Hierbei hilft Ihnen ein mehrfaches `ps l | grep <progname>` in einem anderen Fenster. Dokumentieren Sie Ihre Ergebnisse ausführlich!

d. Beantworten Sie die folgenden Fragen zu den einzelnen Programmen:

   (a) Zu Programm A: Was geschieht in Zeile 27 (`kill(childPID, SIGINT);`) genau? Gibt es einen Zusammenhang mit der Entstehung oder Vermeidung von Zombies und/oder Waisen?

   (b) Zu Programm B: Inwiefern tragen die Zeilen 17 und 25 (`sleep()`-Aufrufe) zum Entstehen von Zombies und/oder Waisen bei?

   (c) Zu Programm C: Was geschieht in Zeile 23 (`wait(&status);`) genau? Gehen Sie insbesondere auf die Bedeutung der Variablen `status` ein. Informationen hierzu finden Sie in der Manpage der Funktion `wait()`.

Programm A:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
  pid_t childPID;
  int a;

  childPID = fork();

  if (childPID == 0 ) {
    for(a=0; a<10; a++) {
      printf("for loop: child PID %d, %i \n", getpid(), a);
      sleep(2);
    }
    printf("child done: child PID %d \n", getpid());
    exit(0);
  }

  for(a=0; a<10; a++) {
    printf( "\t\t\t parent: child PID %d, %i\n", childPID, a);
    sleep(1);
  }
  kill(childPID, SIGINT);
  printf("\t\t\t parent done: child PID %d, %i\n", childPID, a);
  exit(0);
}
```

Programm B:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
  pid_t childPID;
  int a;

  childPID = fork();
```

```
  if(childPID == 0) {
    for(a=0; a<10; a++) {
      printf("for loop: child PID %d, %i \n", getpid(), a);
      sleep(1);
    }
    printf("child done: child PID %d \n", getpid());
    exit(0);
  }

  for(a=0; a<10; a++) {
    printf("\t\t\t\t parent: child PID %d, %i\n", childPID, a);
    sleep(2);
  }
  printf("\t\t\t\t parent done: child PID %d, %i\n", childPID, a);
  exit(0);
}
```

Programm C:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
  pid_t childPID;
  int a, status;

  childPID = fork();

  if (childPID == 0 ) {
    for(a=0; a<10; a++) {
      printf("for loop: child PID %d, %i \n", getpid(), a);
      sleep(2);
    }
    printf("child done: child PID %d \n", getpid());
    exit(0);
  }

  wait(&status);
  for(a=0; a<10; a++) {
    printf("\t\t\t\t parent: child PID %d, %i\n", childPID, a);
    sleep(1);
  }
  wait(NULL);
  printf("\t\t\t\t parent done: child PID %d, %i\n", childPID, a);
  exit(0);
}
```

Programm D:

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
  pid_t childPID;
  int a;

  childPID = fork();

  if (childPID == 0 ) {
    for(a=0; a<10; a++) {
      printf("for loop: child PID %d, %i \n", getpid(), a);
      sleep(2);
    }
    printf("child done: child PID %d \n", getpid());
    exit(0);
  }

  for(a=0; a<10; a++) {
    printf("\t\t\t\t parent:   child PID %d, %i\n", childPID, a);
    sleep(1);
  }
  printf("\t\t\t\t parent done: child PID %d, %i\n", childPID, a);
  exit(0);
}
```

Aufgabe T-4-3

a. Überlegen Sie sich vier sinnvolle Anforderungen an CPU-Zuteilungsstrategien (Scheduling-Strategien).

b. Nennen Sie drei preemptive Scheduling-Strategien, und erklären Sie in kurzen Stichworten ihre Arbeitsweise.

c. Versuchen Sie, die in b) gewählten Strategien hinsichtlich der vier Anforderungen aus a) zu beurteilen. Beurteilen Sie außerdem den Umgang der Strategien mit besonders kurzen sowie mit besonders langen Prozessen.

Praktische Aufgaben (Blatt 4)

Aufgabe P-4-1

a. Schreiben Sie ein einfaches C-Programm, das in einer Schleife `fork()`-Aufrufe tätigt. Die Anzahl n der Schleifendurchläufe soll als Aufrufparameter über die Konsole übergeben werden. Jeder erzeugte Kindprozess (sowie der ursprüngliche Vaterprozess) soll genau einmal seine Parent-Prozess-ID (PPID) ausgeben.

b. Testen Sie Ihr Programm mit verschiedenen Werten für n. Dokumentieren Sie die Ausgabe.

c. Zeichnen Sie den Prozess-Baum, der bei $n := 4$ entsteht. Lesen Sie daraus ab, wie viele **unterschiedliche** PPIDs jeweils in welcher Häufigkeit in der Ausgabe vorkommen können. Überprüfen Sie anhand der tatsächlichen Ausgabe. Wie viele direkte Kinder, Enkel, Urenkel usw. wurden also jeweils erzeugt?

d. Geben Sie eine allgemeine Formel für die Zahl der erzeugten Kindprozesse in Abhängigkeit von n an.

e. Schreiben Sie jetzt ein weiteres C-Programm, das durch einen `fork()`-Aufruf zunächst einen ersten Kindprozess und anschließend durch einen zweiten `fork()`-Aufruf einen zweiten Kindprozess erzeugt. Achten Sie aber darauf, dass keiner der beiden Kindprozesse seinerseits Kindprozesse erzeugt! (Es sollen also insgesamt genau zwei direkte Kindprozesse vom ursprünglichen Prozess angelegt werden.) Jeder der beiden Kindprozesse soll nun das Programm aus Teilaufgabe a) einmal ausführen (Aufrufparameter n beliebig). Verwenden Sie einen passenden Aufruf aus der Familie der `exec*()`-Familie, um die Überlagerung der Kindprozesse zu realisieren.

Bitte denken Sie an ein Makefile!

Aufgabe P-4-2

Schreiben Sie **genau ein** C-Programm, das eine über die Kommandozeile angegebene Anzahl von Kindprozessen erzeugt, die jeweils mit Hilfe der Funktion `getpid()` ihre Prozess-ID auf dem Bildschirm ausgeben. Falls kein Argument über die Kommandozeile übergeben wird, sollen vier Kindprozesse erzeugt werden. Lassen Sie jeden Kindprozess dann noch fünf Sekunden pausieren. Geben sie eine Zeichenkette aus, die besagt, welcher Prozess terminiert, bevor dies passiert. Achten Sie darauf, dass die Kindprozesse ihrerseits keine neuen Kindprozesse erzeugen.

Während die Kindprozesse laufen, soll der Elternprozess durch den Systemaufruf `execlp()` das Unix-Kommando `ps lf` in einer Schleife zehn mal mit jeweils zwei Sekunden Pause dazwischen ausführen. Benutzen Sie die Funktion `sleep()`, um die Wartezeiten zu erzeugen. Schreiben Sie getrennte Funktionen für die Aktionen des Elternprozesses und der Kindprozesse.

Bitte denken Sie an ein Makefile!

Aufgabe P-4-3

Schreiben Sie ein C-Programm, das eine uni-direktionale Kommunikation über eine (namenlose) Pipe zwischen einem Vaterprozess und einem Sohnprozess bereitstellt. Hierbei soll der Vaterprozess den Inhalt einer Datei lesen und an den Sohnprozess schicken, der diesen dann am Bildschirm ausgibt. Der Name der zu lesenden Datei soll dabei als Argument übergeben werden.

Bevor der Sohn beginnt, von der Pipe zu lesen, soll er zunächst 5 Sekunden pausieren. Versehen Sie Vater und Sohn mit Ausgabeanweisungen, die anzeigen, welche Aktion sie gerade ausführen (insb. dass sie gestartet bzw. beendet wurden).

Überlegen Sie sich zur Lösung der Aufgabe die folgenden Teilschritte:

- Wie funktioniert der `pipe()` Systemaufruf?
- Wann muss die Pipe erzeugt werden? Vor oder nach der Erzeugung des Sohns?
- Schreiben Sie getrennte Funktionen für Vater bzw. Sohn.
- Vater:
  - Welche Operation muss der Vater ausführen, sodass er nur in die Pipe schreiben kann?
  - Wie kann aus einer Datei gelesen werden? Belegen Sie den Speicher für den Lesepuffer dynamisch.
  - Mittels welcher Operation kann in die Pipe geschrieben werden?
  - Welche Operation muss auf der Pipe ausgeführt werden, nachdem die Datei vollständig geschrieben wurde?
  - Falls der Vater mit dem Schreiben fertig ist, bevor der Sohn seine Ausgabe abgeschlossen hat: Welche Maßnahmen muss der Vater ergreifen, damit der Sohn seine Arbeit korrekt beenden kann?
- Sohn:
  - Welche Operation muss der Sohn ausführen, sodass er aus der Pipe nur lesen kann?

  - Mittels welcher Operation kann aus der Pipe gelesen werden? Belegen Sie den Speicher für den Lesepuffer dynamisch.
  - Wie wird auf STDOUT ausgegeben?
  - Wie erkennt der Sohn, dass nichts mehr aus der Pipe gelesen werden kann?
- Achten Sie auf eine detaillierte, zuverlässige Fehlerbehandlung.

Bitte denken Sie an ein Makefile!