Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 3: Prozesse und Threads

zum 5. November 2018

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 11.11.2018 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt03.tar zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie statt `./xyz_test` wie auf dem vorherigen Aufgabenblatt nun `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich hierfür nach wie vor im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `pid_t fork ();`
- `pid_t wait (int *status);`
- `pid_t getpid ();`
- `pid_t getppid ();`
- `void perror (const char *str);`
- `unsigned int sleep (unsigned int);`
- `int gettimeofday (struct timeval *restrict tp, void *restrict tzp);`
- `char *ctime (const time_t *t);`
- `void srand (unsigned int seed);`
- `unsigned int rand ();`

2 Tutoraufgaben

2.1 Betriebssysteme allgemein

Welche der folgenden Aussagen kennzeichnen den Begriff **Betriebssystem**, bzw. die Aufgabe oder Funktion eines Betriebssystems?

a) ... die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

b) ... ein System zur Programmierung und Wartung systemnaher Software.

c) ... eine Software-Schicht, die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle oder eine virtuelle Maschine anbietet, die einfacher zu verstehen und zu programmieren ist (als die nackte Hardware).

d) ... ein System zur Manipulation und Darstellung von Daten/Informationen, ihre Anordnung in Listen, Bäumen usw., und ihre Weiterverarbeitung.

e) ... ein Programm, das als Vermittler zwischen Rechnernutzer und Rechner-Hardware fungiert. Sein Zweck ist, eine Umgebung bereitzustellen, in der ein Benutzer bequem und effizient Programme ausführen kann.

f) ... ein System, das alle zur Programmerstellung notwendigen Werkzeuge zusammenführt.

Falls eine Aussage nicht die Eigenschaften von Betriebssystemen beschreibt, begründen Sie das!

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

2.2 Eigenschaften von Betriebssystemen

Beantworten Sie die folgenden Fragen:

a) Was ist ein Prozess?

b) Erklären Sie die Unterschiede zwischen einem Prozess und einem Thread!

c) Was sind Betriebsmittel? Wie können Sie sich vorstellen, dass diese sinnvoll klassifiziert werden? Begründen Sie Ihre Antwort!

d) Welche Ausführungsmodi gibt es bei Betriebssystemen? Wie unterscheiden sich diese und warum sind diese Unterscheidungen sinnvoll?

e) Welche der folgenden Instruktionen sollte nur im System-Mode ausgeführt werden?

- Alle Interrupts deaktivieren
- Die Zeit der System-Clock lesen
- Die Zeit der System-Clock setzen
- Die Memory Map ändern

f) Was ist eine virtuelle Maschine?

2.3 Operatorpräzedenz in C (Tutoraufgabe)

In C gelten die folgenden Präzedenzregeln (nicht vollständig):

[] (Array) und **()** (Funktion) werden von links nach rechts abgearbeitet und haben Vorrang vor * (Pointer/Dereferenzierung) und **&** (Adresse), welche von rechts nach links abgearbeitet werden. Bestimmen Sie davon ausgehend, wie die folgenden Ausdrücke gelesen werden:

```
long **foo[7];
char *(*(**foo [][8])())[];
```

2.4 Rekursion (Tutoraufgabe)

Wir betrachten nun **rekursive Funktionen**. Rekursion ist eine sehr mächtige (doch für Studenten leider oft eine verwirrende) mathematische Methode, die in der Lage ist, komplexe Probleme zu lösen. Die grundlegende Idee besteht darin, ein Problem zunächst in kleinere, einfacher zu lösende Probleme zu unterteilen. Das Problem wird so lange unterteilt, bis man den sogenannten **Trivialfall** erreicht hat. Dieser löst das Teilproblem und stoppt die weitere Unterteilung.

Jede rekursive Funktion besteht aus zwei wichtigen Teilen:

a) **Trivialfall:** Der kleinste Bestandteil eines Problems, der die Rekursion stoppt.

b) **Reduktionsschritt:** Teilt ein Problem solange in Unterprobleme auf, bis der Trivialfall erreicht wurde. Dies geschieht durch den erneuten Aufruf derselben Funktion.

Beide Teile sind von großer Bedeutung für die rekursive Programmierung, denn sie sind dafür verantwortlich, dass das Programm sowohl richtig berechnet wird, als auch terminiert.

Beispiel: Fakultät

Gegeben sei die iterative Version der Fakultätsfunktion `factorial`:

```
unsigned int factorial(unsigned int n)
{
    unsigned int p = 1;

    while ( n != 0 ) {
        p = p * n;
        n = n - 1;
    }

    return p;
}
```

Der folgende Codeabschnitt implementiert eine rekursive Version der Funktion `factorial`:

```
unsigned int factorial_rec(unsigned int n)
{
    if (n == 0)
        return 1;

    return n * factorial_rec(n - 1);
}
```

a) Identifizieren Sie den Trivialfall und die Rekursionsschritte der gegebenen Funktion.

b) Was passiert, wenn der Trivialfall entfernt wird?

c) Was passiert, wenn die Funktion `factorial_rec` mit dem Argument -1 aufgerufen wird?

d) Wie kann man das Problem lösen?

2.5 Prozesse, Waisen, Zombies und Dämonen (Tutoraufgabe)

In der Vorlesung wurde bereits das Konzept eines Prozesses eingeführt. In dieser Aufgabe möchten wir dieses Konzept erweitern und vertiefen. Die folgenden Aufgaben werden unter anderem auf diesem Konzept aufbauen.

Ein **Prozess** ist eine Instanz eines ausgeführten Programms (bzw. ein Programm in Ausführung). Ein **Programm** ist eine Datei mit Informationen darüber, wie ein Programm zur Laufzeit erstellt werden soll. Diese Informationen beinhalten unter anderem (nicht vollständig):

- **Binärformat**: Jedes Programm beinhaltet zusätzliche Meta-Informationen, die sich zwischen den verschiedenen Binärformaten unterscheiden. Unter Linux wird das **Executable and Linking Format** (ELF) und unter Mac OS X/MacOS das **Mach-O** Binärformat eingesetzt.
- **Machinencode**: Der Maschinencode kann als eine Repräsentation des Programms aufgefasst werden, die von der CPU ausgeführt werden kann.
- **Daten**: Das Programm beinhaltet Daten, die vom Prozess während der Laufzeit benötigt werden. Darunter fallen unter anderem Strings.
- **Symbol-Informationen**: Beschreibung der Positionen und Namen der im Programm verwendeten Funktionen und Variablen. Diese Informationen werden in Tabellen gespeichert, die unter anderem beim Debuggen oder Linken des Programms benötigt werden.

- **Shared-Library / Dynamic-Linking Informationen**: Weitere Informationen, die die verwendeten Libraries auflisten. Diese Informationen sind notwendig, sodass während der Laufzeit alle benötigten Libraries in den Speicher geladen und dem Programm zur Verfügung gestellt werden können.

Prozesse werden hierarchisch verwaltet. Jeder Prozess hat eine eindeutige **Prozess ID** (PID), die den entsprechenden Prozess identifiziert; Die PID wird jedem neuen Prozess zugewiesen. Z.B. der **Init** Prozess ist der erste Prozess des Systems und besitzt die $PID = 1$. Davon abgesehen besitzt jeder Prozess (bis auf Init) einen Vaterprozess, der durch die **Parent PID** (PPID) identifiziert wird. Im Folgenden werden wir uns etwas genauer mit der Prozesserstellung beschäftigen.

a) Wie wird ein neuer Prozess erzeugt?

b) Wie kann man Eltern- von Kindprozess unterscheiden?

c) Welche Ressourcen teilen sich Kind- und Elternprozess?

d) Wie erfährt der Elternprozess, wann und wie der Kindprozess terminiert?

e) Was versteht man (unter Linux) unter der Prozesshierarchie?

f) Was passiert, wenn der Kindprozess zuerst terminiert?

g) Was passiert, wenn der Elternprozess zuerst terminieert?

3 Hausaufgaben

3.1 Erzeugen eines Kindprozesses: forkone (3 Punkte)

Schreiben Sie ein Programm `forkone`, dass einen Kind-Prozess erzeugt und dann auf dessen Beendigung wartet. Der Kind-Prozess soll jeweils im Sekundentakt bis zu einer Zahl K hochzählen. K wird auf der Kommandozeile übergeben.

Zu Beginn und kurz vor dem Beenden soll der Elternprozess die jeweils aktuelle Uhrzeit ausgeben. Beim Hochzählen sollen die Prozess-ID des jeweiligen Kindprozesses und die des Elternprozesses vorangestellt werden, so dass sich folgendes Ausgabeformat ergibt.

Ausserdem soll der Kindprozess den Wert K zu seiner eigenen Prozess-ID addieren und das Ergebnis modulo 100 (also nur die Einer- und Zehnerstellen) als Rückgabewert (Exit-Code) liefern. Hinweis: C kennt den Modulo-Operator `%`. Diesen Wert soll der Elternprozess auslesen und als Ergebnis ausgeben.

```
$ ./forkone 10
Start: Tue Oct  9 22:16:22 2018
68333 68332 1
68333 68332 2
68333 68332 3
68333 68332 4
68333 68332 5
68333 68332 6
68333 68332 7
68333 68332 8
68333 68332 9
68333 68332 10
Exit-Code: 43
Ende: Tue Oct  9 22:16:32 2018
```

Syntax: `forkone <K>`

Quellen: `forkone.c`

Executable: `forkone`

3.2 Erzeugen mehrerer Kindprozesse: forkmany (7 Punkte)

Erweitern Sie das Programm von Abschnitt 3.1, dass es N Kindprozess erzeugt, die alle jeweils bis K zählen. Der Elternprozess soll sich alle erzeugten Kind-Prozesse in einer Liste merken und warten, bis sie alle beendet sind.

Schließlich sollen die Kind-Prozesse nicht alle bis K zählen, sondern bis zu einer zufälligen ganzen Zahl in $[K/2; 1,5 \times K]$, wenn der optionale Parameter "-r" auf der Kommandozeile angegeben wird. Hierfür können Sie Integer-Arithmetik verwenden. Nehmen Sie als Default-Parameter $K = 10$ und $N = 1$ an, falls der jeweilige Parameter nicht auf der Kommandozeile übergeben wird.

Das Ausgabeformat bleibt unverändert. Führen Sie das Programm mehrfach mit vielen Prozessen und $K \geq 10$ aus und beobachten Sie, ob/wie sich die Reihenfolge der Ausgaben der Prozesse bei den einzelnen Schritte aufgrund des Schedulings ändert.

Hinweis: die eckigen Klammern in der folgenden Syntax drücken optionale Parameter aus. Die Reihenfolge der Argumente kann prinzipiell beliebig sein. Verwenden Sie `getopt (3)` zum Parsieren der Kommandozeilenparameter.

Syntax: `forkmany [-k <K>] [-n <N>] [-r]`

Quellen: `forkmany.c list.c list.h`

Executable: `forkmany`