Concurrent programming – Prinzipien und Einführung in Prozesse

by Leonardo Giordani
<leo.giordani(at)libero.it>

About the author:

Student an der Fakultät für Telecommunication Engineering, Politecnico Milan. Arbeitet als Netzwerkadministrator und interessiert sich für Programmieren (meist Assembler oder C/C++). Seit 1999 arbeitet er fast ausschließlich mit Linux/Unix.

Translated to English by:
Leonardo Giordani
<leo.giordani(at)libero.it>

Abstract:

Diese Serie von Artikeln hat die Absicht, dem Leser das Konzept von Multitasking und seine Umsetzung unter Linux beizubringen. Angefangen mit theoretischen Konzepten werden wir komplette Applikationen schreiben und zeigen, wie man zwischen Prozessen mit einem einfachen, aber effizienten Protokoll kommunizieren kann.
Voraussetzung für das Verständnis dieses Artikels ist:

- Minimale Kenntnisse der Shell
- Basiswissen von C (Syntax, Schleifen, Bibliotheken)

Alle Referenzen auf man–Pages werden mit Klammern hinter dem Befehl angegeben. Alle glibc Funktionen sind mit gnu info Seiten dokumentiert (info Libc, oder info:/libc/Top in konqueror eingeben).

Einführung

Eines der wichtigsten Wendepunkte in der Geschichte der Betriebssysteme war das Konzept Multiprogramming, eine Technik, um die Ausführung von Programmen zeitlich ineinander zu verschachteln. Dadurch wird die CPU konstanter ausgelastet. Denken wir an eine einfache Workstation, wo der Benutzer gleichzeitig eine Textverarbeitung, einen Audiospieler, eine Print–queue, einen Webbrowser und vieles mehr laufen hat. Alles scheint parallel zu laufen. Es ist ein wichtiges Konzept für moderne Betriebssysteme. Wir werden sehen, dass diese kurze Liste von Programmen nur ein kleiner Teil von dem ist, was wirklich läuft, auch wenn es die offensichtlichsten Programme sind.

Das Konzept der Prozesse

Um Programm so laufen zu lassen, als würden sie gleichzeitig laufen, wird eine erstaunliche Arbeit vom Betriebssystem geleistet. Um Konflikte zwischen den Programmen zu vermeiden, laufen sie getrennt und abgekapselt voneinander. Die Kapsel enthält alle Informationen, die zu ihrer Ausführung benötigt werden.

Bevor wir untersuchen, was in unserer Linuxkiste passiert, wollen wir einige technische Begriffe definieren. Nehmen wir ein laufendes **PROGRAMM** zu einer bestimmten Zeit, ist der **CODE** eine Menge von Anweisungen, aus dem es gemacht ist, der **MEMORY SPACE** ist der des Speichers, der mit Daten belegt ist und der **PROCESSOR STATUS** ist der Wert der Parameter des Microprocessors beschreibt wie z.B. Flags oder die Program Counter (die Adresse der nächsten Anweisung, die ausgeführt werden soll).

Wir definieren den Term **RUNNING PROGRAM** (laufendes Programm) als eine Anzahl von Objekten, die aus CODE, MEMORY SPACE und PROCESSOR STATUS bestehen. Wenn zu einem bestimmten Zeitpunkt diese Informationen (CODE, MEMORY SPACE, PROCESSOR STATUS) abgespeichert werden und durch die Daten eines anderen Programmes ersetzt werden, dann wird der Progammfluß des letzteren dort fortgesetzt, wo er vorher angehalten wurde. Macht man das abwechselnd mit dem einen und dem anderen Programm, so sieht es aus, als liefen sie gleichzeitig. Der Ausdruck **PROCESS** (oder **TASK**) wird benutzt, um solch ein laufendes Programm zu beschreiben.

Was passierte auf der Workstation, die wir oben beschrieben haben? Zu jedem Zeitpunkt lief nur ein einziges Programm. Es gibt nur einen Microprocessor und der kann immer nur ein Programm bearbeiten. Nach einem bestimmten Zeitinterval, das man QUANTUM nennt wird der laufende PROCESS geparkt (suspended). Seine Informationen werden gespeichert und ein anderer PROCESS wird wieder zum Leben erweckt. Dieser läuft wieder nur für ein QUANTUM und wird dann geparkt. Das nennt man Multitasking.

Wie schon gesagt, führt Multitasking zu einem Satz von Problemen, die nicht trivial zu lösen sind. Probleme, wie das Verwalten der wartenden Prozesse (Queue Management und SCHEDULING). Diese haben zu tun mit der Architektur des Betriebssystems. Vielleicht wird das das Thema eines zukünftigen Artikels. Vielleicht eine Einführung zu einigen Teilen des Linuxkernel.

Prozesse unter Linux und Unix

Nun wollen wir einige Prozesse, die auf unserer Maschine laufen, entdecken. Der Befehl dafür ist **ps(1)**. Das ist eine Abkürzung für "process status". In einem normalen Text−Shellfenster tippt man ps und erhält folgende Ausgabe:

```
  PID TTY          TIME CMD
 2241 ttyp4    00:00:00 bash
 2346 ttyp4    00:00:00 ps
```

Ich sagte schon, dass diese Liste nicht vollständig ist, aber laßt uns im Moment auf folgendes konzentrieren. ps gab uns eine Liste der Prozesse, die im Moment in diesem Fenster laufen. In der letzten Spalte sehen wir den Namen unter dem der Prozess läuft (Namen wie "mozilla" für den Mozilla Webbrowser und "gcc" für GNU Compiler Collection). "ps" erscheint auch in der Liste, da es am laufen war, als die Liste gedruckt wurde. Der andere Prozess ist hier die Bourne Again Shell, die Shell, die in meinen Terminals läuft.

Im Moment wollen wir die Information über TIME und TTY auslassen und uns PID, Process IDentifier, ansehen. Die pid ist eine eindeutige positive Nummer (nicht null), die einem laufenden Prozess zugewiesen wird. Wenn der Prozess einmal beendet wurde, könnte sie wieder für einen neuen Prozess verwendet werden. Während der Ausführung eines Programmes bleibt die Pid gleich. Das bedeutet, daß die Ausgabe von ps, die du auf deinem Rechner erhalten wirst, vermutlich anders sein wird. Um das zu prüfen, öffnen wir noch ein

Terminalfenster und geben ps ein. Dieses Mal sehen wir wieder die gleichen Prozesse, aber mit anderer Pid.

Wir können auch eine Liste aller Prozesse, die auf unserer Linuxmaschine laufen, erhalten. Die man–Pages von ps sagen, dass die Option –e alle Prozesse ausgibt. Also tippen wir "ps –e" und leiten die Ausgabe in eine Datei (ps.log) um, um sie besser betrachten zu können.

```
ps -e > ps.log
```

Nun können wir diese Datei mit unserem Lieblingseditor anschauen (oder einfach mit dem Befehl less). Wie schon gesagt, ist die Anzahl der laufenden Prozesse viel höher als wir erwarten würden. Wir stellen fest, dass es dort auch Prozesse gibt, die nicht von uns gestartet wurden. Unabhängig davon, was für ein System du hast, wird es einen Prozess mit der Pid 1 geben, der init heißt. Init ist der Vater aller Prozesse und hat die Pid 1, weil er zuerst gestartet wird. Des weiteren bemerken wir eine Anzahl von Prozessen, deren Namen in "d" enden. Das sind sogenannte "daemons" und sie sind eines der wichtigsten Prozesse. Wir werden sie in einem späteren Artikel besprechen.

Mutitasking in der libc

Wir verstehen nun das Konzept Prozess und sehen, wie wichtig es für das Betriebssystem ist. Wir werden nun Programmcode für Mutitasking schreiben. Von der trivialen Ausführung zweier Prozesse werden wir zur Kommunikation und Syncronisation kommen. Wir werden zwei elegante Lösungen dieser Probleme kennen lernen: Messages und semaphores, aber das wird noch in einem späteren Artikel über threads erklärt werden.

Die Standard C Bibliothk (libc, unter Linux glibc) benutzt die Unix System V Multitasking Möglichkeiten. Das Unix System V (ab jetzt einfach SysV genannt) ist eine kommerzielle Uniximplementation und der Begründer einer der zwei wichtigsten Unixfamilien. Der andere Zweig ist BSD Unix.

In der libc ist der Datentyp pid_t definiert. Das ist eigentlich das selbe wie ein Integer, aber man benutzt pid_t um den Zweck klar zu machen.

Hier ist die Funktion, die die Pid des Prozesses unseres Programmes liefert.

```
pid_t getpid (void)
```

Definiert ist getpid mit pid_t in unistd.h und sys/types.h. Jetzt schreiben wir ein Programm, das seine Pid ausdruckt.

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

int main()
{
  pid_t pid;

  pid = getpid();
  printf("The pid assigned to the process is %d\n", pid);

  return 0;
}
```

Speichere das Programm als print_pid.c und kompiliere es mit:

```
gcc -Wall -o print_pid print_pid.c
```

Das wird eine ausführbare Datei namens print_pid erzeugen. Erinnere dich daran, dass es nötig ist, das Programm als "./print_pid" zu starten, falls das augenblickliche Verzeichnis nicht in deinem Pfad enthalten ist. Die Ausführung dieses Programmes bringt keine Überraschungen. Es druckt eine positive Zahl, die sich bei jeder neuen Ausführung um eins erhöht. Es ist jedoch nicht immer so, dass sich die Zahl um eins erhöht, weil es sein kann, dass in der Zwischenzeit noch andere Prozesse gestartet wurden. Tippt man zwischendurch z.B. ps oder ls so erhöht sich die Zahl mit Sicherheit nicht nur um eins beim nächsten Ausführen von print_pid.

Nun wird es Zeit zu lernen, wie man Prozesse erzeugt, aber ich muß erst erklären, was wir eigentlich dabei wirklich machen. Wenn ein Programm, das als Prozess A läuft, einen anderen Prozess (B) erzeugt, dann sind A und B zunächt identisch. A und B bestehen aus dem gleichen Code im Speicher. Danach können sie sich in unterschiedliche Richtungen entwickeln. Das kann z.B in Abhängigkeit von Benutzerdaten sein. Prozess A ist der Vater und B ist der Sohn. Nun verstehen wir auch besser den Ausdruck "Vater aller Prozesse", den man oft für init benutzt. Die Funktion, die neue Prozesse erzeugt, ist

```
pid_t fork(void)
```

Sein Name fork (=Gabel im Englischen) kommt von der Gabelung der Prozesse. Die zurückgegebene Nummer ist die Pid, aber man muß hier etwas beachten. Wir sagten, dass die Prozesse dupliziert werden (Vater und Sohn), aber direkt nach der Duplizierung muß man im allgemeinen wissen, wer Vater und wer Sohn ist. Die zwei sind ja zunächt identisch. Welcher wird direkt nach der Duplizierung zuerst ausgeführt? Nun die Antwort ist einfach: Einer von beiden und die Entscheidung wird vom Betriebssystem getroffen.

In jedem Fall ist es aber wichtig zu wissen, wer Vater und wer Sohn ist, denn normalerweise wird man etwa folgendes (symbolischer Code) programmieren:

```
- FORK
- WENN DU SOHN BIST DANN MACHE .....
- WENN DU VATER BIST DANN MACHE .....
```

Laßt uns das Geheimnis aufdecken. Fork gibt '0' an den Sohn zurück und die Pid des Sohns an den Vater. Man braucht also nur zu testen, ob die Pid Null ist, um zu wissen welcher Prozess man ist. In C ist das:

```
int main()
{
  pid_t pid;

  pid = fork();
  if (pid == 0)
  {
    CODE OF THE SON PROCESS
  }
  CODE OF THE FATHER PROCESS
}
```

Jetzt ist es Zeit, den ersten richtigen Multitasking−Code zu schreiben. Speichere die folgenden Zeilen als fork_demo.c und kompiliere sie. Ich habe Zeilennummern vor den Code gestellt, um ihn besser besprechen zu können. Das Programm wird fork ausführen und Vater und Sohn werden etwas auf den Bildschirm schreiben. Wenn alles nach Plan läuft, wird mal der Sohn und mal der Vater schreiben, so dass die Ausgabezeilen gemischt sind. Die Entscheidung, wer wann dran ist, liegt jedoch beim Betriebssystem.

```
(01) #include <unistd.h>
(02) #include <sys/types.h>
(03) #include <stdio.h>

(04) int main()
```

```
(05) {
(05)   pid_t pid;
(06)   int i;

(07)   pid = fork();

(08)   if (pid == 0){
(09)     for (i = 0; i  < 8; i++){
(10)       printf("-SON-\n");
(11)     }
(12)     return(0);
(13)   }

(14)   for (i = 0; i < 8; i++){
(15)     printf("+FATHER+\n");
(16)   }

(17)   return(0);
(18) }
```

Zeilen (01)–(03) enthalten die includes für die Bibliothken (Standard I/O, multitasking).
Die main (wie immer in GNU) gibt einen Integer zurück, was normalerweise Null ist, falls alles wie geplant läuft und eins, falls ein Fehler aufgetreten ist. Unser Programm enthält zur Vereinfachung keine Fehlerbehandlung (Es geht hier um Konzepte, nicht um gute Programme).
Danach definieren wir eine pid (05) und einen Index für die Schleifen (06). Datentyp pid_t und int sind, wie schon gesagt, identisch, aber pid_t wird wegen der Klarheit der Darstellung genommen.
In Zeile (07) rufen wir fork auf. In Zeile (08) testen wir, ob wir Sohn oder Vater sind. Der Code in den Zeilen (09)–(13) wird im Sohn–Prozess ausgeführt und (14)–(16) im Vater–Prozess.
Wir schreiben einfach 8 mal "–SON–" oder "+FATHER+". Es ist wichtig, dass der Sohn mit "return 0" beendet wird, sonst würde er weitermachen und den Code vom Vater ausführen. Solch vergessene "return" sind oft schwer zu finden und können zu merkwürdigen Fehlern führen.

Die Ausführung des Programms wird vielleicht nicht sehr zufriedenstellend sein: Ich kann nicht garantieren, dass die Zeilen "–SON–" oder "+FATHER+" wirklich gemischt sein werden. Das ist so, weil diese kurzen Schleifen sehr schnell ausgeführt werden können. Vermutlich wird die Ausgabe so aussehen, dass erst "+FATHER+" und dann "–SON–" oder umgekehrt kommt. Wenn man es aber mehrmals probiert, kann man Glück haben und es gibt eine Mischung.

Wenn wir eine Zufallsverzögerung vor jedem printf einfügen, werden wir einen sichtbareren Multitasking Effekt erhalten: Wir machen das mit sleep und rand.

```
sleep(rand()%4)
```

Das läßt das Programm zufällig zwischen 0 und 3 Sekunden schlafen. (% ist der Modulo–Operator, Rest beim Teilen durch eine ganze Zahl). Nun sieht der Code so aus:

```
(09)  for (i = 0; i < 8; i++){
(->)    sleep (rand()%4);
(10)    printf("-FIGLIO-\n");
(11)  }
```

und dasselbe für den Code des Vaters. Speichere das als fork_demo2.c, kompiliere es und führe es aus. Das Programm läuft langsamer wegen der sleeps, aber die Ausgabe ist jetzt vermischt:

```
[leo@mobile ipc2]$ ./fork_demo2
-SON-
+FATHER+
+FATHER+
```

```
-SON-
-SON-
+FATHER+
+FATHER+
-SON-
-FIGLIO-
+FATHER+
+FATHER+
-SON-
-SON-
-SON-
+FATHER+
+FATHER+
[leo@mobile ipc2]$
```

Nun ein neues Problem. Wir können beliebig viele Söhne aus dem einen Vater erzeugen, aber oft muß der Vater mit dem Sohn kommunizieren oder zumindest synchronisieren, um bestimmte Dinge zur richtigen Zeit zu machen. Die erste Möglichkeit für eine solche Synchronisation ist die wait Funktion.

```
pid_t waitpid (pid_t PID, int *STATUS_PTR, int OPTIONS)
```

Hier ist PID die PID des Prozesses, auf den wir warten. Warten bis er fertig ist. STATUS_PTR ist ein Pointer auf Status Informationen über den Sohn (NULL, falls wir an den Status Informationen nicht interessiert sind) und OPTIONS ist eine Anzahl von Optionen, um die wir uns im Moment noch keine Gedanken machen wollen.
Das ist ein Beispiel eines Programms, in dem der Vater einen Sohn erzeugt und zu einem bestimmten Zeitpunkt dann auf ihn wartet.

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

int main()
{
  pid_t pid;
  int i;

  pid = fork();

  if (pid == 0){
    for (i = 0; i < 14; i++){
      sleep (rand()%4);
      printf("-SON-\n");
    }
    return 0;
  }

  sleep (rand()%4);
  printf("+FATHER+ Waiting for son's termination...\n");
  waitpid (pid, NULL, 0);
  printf("+FATHER+ ...ended\n");

  return 0;
}
```

Die sleep Funktion wurde wieder eingeführt, um Multitasking sichtbarer zu machen. Laßt uns das als fork_demo3.c speichern, kompilieren und ausführen. Wir haben gerade unsere erste synchronisierte Multitasking Applikation geschrieben!

Im nächsten Artikel werden wir mehr über Synchronisation und Kommunikaton zwischen Prozessen lernen. Nun kannst du deine eigenen Programme schreiben und mir zuschicken. Ich kann dann einige gute Lösungen oder häufige Probleme zeigen. Schick mir beides, den C−Code und ein kleines Textfile mit einer Beschreibung des Programms. Gute Arbeit!

Empfehlungen zum Lesen

- Silberschatz, Galvin, Gagne, **Operating System Concepts – Sixth Edition**, Wiley&Sons, 2001
- Tanenbaum, WoodHull, **Operating Systems: Design and Implementation – Second Edition**, Prentice Hall, 2000
- Stallings, **Operating Systems – Fourth Edition**, Prentice Hall, 2002
- Bovet, Cesati, **Understanding the Linux Kernel**, O'Reilly, 2000
