Concurrent programming – Kommunikation zwischen Prozessen

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.

Voraussetzungen für das Verstehen des Artikels sind:

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

Du solltest den ersten Artikel in dieser Serie gelesen haben, denn er ist die Basis für diesen Artikel: November 2002, Artikel 272.

Enführung

Wie wir in dem vorangegangen Artikel gesehen haben, braucht man zum Erzeugen eines neuen Prozesses (Funktion fork) nur wenige Zeilen Code, da das Betriebssytem die Initialisierung und das Verwalten der Prozesse übernimmt.

Diese ist eine grundlegende Funktion des Betriebssystems. Es ist 'der Überwacher aller Prozesse'. Prozesse werden daher in einer eigenen Umgebung ausgeführt. Dadurch, dass das Betriebssystem die Kontrolle hat, entsteht für den Entwickler ein Problem: Wie ist es möglich, dass unabhängige Prozesse zusammenarbeiten?

Das Problem ist komplexer als es zunächst aussieht. Es ist nicht nur eine Frage der Synchronisation von Prozessen, sondern auch, wie man Daten teilen kann und gemeinsam lesen und schreiben.

Wir wollen zunächst mal das klassische Problem des gleichzeitigen Datenzugriffs von zwei Prozessen diskutieren. Wenn zwei Prozesse dieselben Daten nur lesen, dann ist das offensichtlich kein Problem. Nun soll einer der beiden Prozesse die Daten modifizieren: Der andere wird unterschiedliche Daten sehen in Abhängigkeit davon, wann er auf die Daten zugreift, vor oder nach dem Schreiben des anderen Prozesses. Nehmen wir z.B. die zwei Prozesse "A" und "B" und die ganze Zahl "d". Prozess A erhöht sie um eins und Prozess B druckt sie aus. In einer symbolischen Schreibweise kann man das so ausdrücken:

A { d−>d+1 } & B { d−>output }

das "&" beschreibt hier gleichzeitige Ausführung. Eine mögliche Lösung ist:

(−) d = 5 (A) d = 6 (B) output = 6

aber wenn B zuerst ausgeführt wird, dann erhalten wir:

(−) d = 5 (B) output = 5 (A) d = 6

Man versteht sofort, wie wichtig es ist, diese Situationen richtig handzuhaben: Das Risiko, inkonsistente Daten zu erhalten, ist groß und nicht akzeptabel. Man denke nur daran, dass die Daten auch das Geld auf einem Bankkonto darstellen könnte und man wird das Problem nie unterschätzen.

Im vorangegangen Artikel sprachen wir über die einfachste Form der Synchronisation, die waitpid(2) Funktion. Mit ihr kann ein Prozess auf den anderen warten, bis er fertig ist. Damit kann man schon einige der Konflikte beim Datenzugriff lösen: Prozess P2 kann warten, bis P1 mit dem Modifizieren der Daten fertig ist und dann selbst fortfahren.

Das ist sicher eine Lösung, aber nicht die Beste, weil P2 warten muss, bis P1 fertig ist, selbst wenn P1 nicht mehr an gemeinsamen Daten arbeitet, sondern etwas anderes macht. Wir müssen daher die Granularität erhöhen und eine Regel einführen, um den konfliktfreien Zugrifff auf einzelne Daten zu kontrollieren. Die Standard Library bietet hierzu etwas namens SysV IPC (System V InterProcess Communication).

SysV keys

Bevor wir uns mit der Theorie und ihrer Implementation beschäftigen, wollen wir uns eine Datenstruktur ansehen: IPC keys. Ein IPC key ist eine Zahl, die eindeutig eine IPC Kontrollstruktur beschreibt, aber es kann auch verwendet werden, um allgemeine Identifikationsnummern zu erzeugen. Ein IPC key wird mit der ftok(3) Funktion erzeugt:

key_t ftok(const char *pathname, int proj_id);

Als Argument braucht die Funktion den Pfad zu einer existierenden Datei (pathname) und eine Integerzahl. Es ist nicht garantiert, dass der Key eine global eindeutige Zahl ist, da sie aus der i−node und device Nummer der Datei, die als Argument gegeben wurde, gebildet wird. Eine gute Lösung ist eine kleine Routine zu schreiben, die die bisher vergebenen Keys speichert und damit feststellen kann, ob es ein Duplikat oder eine neue Zahl ist.

Semaphoren

Die Idee einer Semaphore (Flügelsignal, z.B. im Schienenverkehr) kann ohne große Modifikation für die Zugriffskontrolle auf Daten verwendet werden. Eine Semaphore ist eine Struktur mit einer Zahl größer oder gleich Null und damit werden eine Reihe von Prozessen verwaltet, die auf eine Zugriffserlaubnis warten. Semaphoren sind sehr leistungsfähig und damit auch komplexer. Wir fangen erst einmal an, ein Programm zu

schreiben, dass keine weitere Fehlerbehandlung enthält und kümmern uns um die Fehlerbehandlung erst bei einem richtigen Programm.

Semaphoren kann man verwenden, um Zugriffe zu verwalten: Der Wert der Semaphore beschreibt die Zahl der Prozesse, die Zugriff auf Daten oder andere Dinge haben. Jedes Mal, wenn ein Prozess Zugriff erhält, wird die Semaphore heruntergezählt und wenn der Zugriff wieder freigegeben wird, zählt man hoch. Sollen exklusive Zugriffe vergeben werden, so ist der Anfangswert 1 (nur ein Prozess kann zu jeder Zeit zugreifen).

Die Semaphore kann man auch als Zähler für irgendwelche Dinge verwenden. In diesem Fall repräsentiert der Wert der Semaphore die Anzahl der verfügbaren Dinge (z.B. Anzahl freier Speicherzellen).

Nun zu einer praktischen Anwendung: Wir haben einen Pufferspeicher, in den mehrere Prozesse (S1,...,Sn) schreiben können und der von nur einem Prozess gelesen wird. Zu jedem Zeitpunkt darf immer nur ein Prozess den Wert des Speichers ändern. Offensichtlich können die Prozesse S immer schreiben, solange der Speicher nicht voll ist und der Leseprozess L kann nur lesen, wenn der Speicher nicht leer ist. Wir brauchen daher 3 Semaphoren: eine zum Verwalten der Zugriffsrechte und zwei, um mitzuhalten, wieviele Elemente im Puffer sind (wir werden später sehen, warum nur zwei Semaphoren nicht ausreichen).

Da der Zugriff exklusiv sein soll, muss die erste Semaphore eine binäre Semaphore sein. Die Werte der beiden anderen Semaphoren hängen von der Größe des Pufferspeichers ab.

Nun wollen wir lernen, wie man Semaphoren in C mit SysV Funktion benutzt. Mit semget(2) erzeugt man die Semaphoren:

int semget(key_t key, int nsems, int semflg);

Hier ist key ein IPC Key, nsems ist die Zahl der Semaphoren, die wir haben wollen und semflg beschreibt Zugriffsrecht mit 12 Bits. Die ersten 3 hängen mit dem Erzeugen der Semaphoren zusammen und die anderen 9 mit Lese– und Schreibrechten für user, group und other (beachte den Zusammenhang mit chmod und dem Unix Dateisystem). Eine vollständige Beschreibung findet sich in der man–Page ipc(5). Wie man sieht, verwaltet SysV einen Satz Semaphoren und nicht einzelne, was zu kompakterem Code führt.

Nun wollen wir unsere erste Semaphore erzeugen:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(void)
{
  key_t key;
  int semid;

  key = ftok("/etc/fstab", getpid());

  /* create a semaphore set with only 1 semaphore: */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  return 0;
}
```

Nun müssen wir lernen, wie man sie verwaltet und entfernt. Das macht man mit semctl(2):

int semctl(int semid, int semnum, int cmd, ...)

Diese Funktion arbeitet gemäß der Aktion, die in cmd definiert ist, auf dem Satz Semaphoren, semid, und der einzelnen Semaphore semnum. Die man−Page erklärt die Funktion genauer. Für einige Aktionen braucht man noch ein weiteres Argument, dessen Datentyp wie folgt aussieht:

```
union semun {
 int val;                  /* value for SETVAL */
 struct semid_ds *buf;     /* buffer for IPC_STAT, IPC_SET */
 unsigned short *array;    /* array for GETALL, SETALL */
                           /* Linux specific part: */
 struct seminfo *__buf;    /* buffer for IPC_INFO */
};
```

Um den Wert einer Semaphore zu setzen, benutzt man die SETVAL Direktive und der Wert steht in der union semun. Lasst uns das Programm erweitern und die Semaphore auf 1 setzen:

```
[...]

  /* create a semaphore set with only 1 semaphore */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

[...]
```

Später müssen wir die Semaphore wieder freigeben. Das macht man mit IPC_RMID und semctl. Das löscht die Semaphore und schickt eine Nachricht an alle Prozesse, die auf Zugriff zu der Semaphore warten. Eine letzte Änderung des Programms ist:

```
[...]

  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* deallocate semaphore */
  semctl(semid, 0, IPC_RMID);

[...]
```

Wie wir gesehen haben, ist das Erzeugen und Verwalten einer Kontrollstruktur für parallel arbeitende Prozesse übrhaupt nicht schwierig. Wenn noch Fehlerbehandlung eingeführt wird, wird der Code etwas länger, aber nicht komplexer.

Die Semaphore kann nun mit der Funktion semop(2) benutzt werden:

int semop(int semid, struct sembuf *sops, unsigned nsops);

Hier ist semid, die Nummer, die einen Satz Semaphoren identifiziert, sops ein Array mit Operationen und nsops die Anzahl dieser Operationen. Jede Operation wird durch die sembuf struct dargestellt:

unsigned short sem_num; short sem_op; short sem_flg;

sem_num ist die Semaphoren Nummer, sem_op die Operation, und sem_flg ein Flag, das die Wartepolitik beschreibt. Wir benutzen sem_flg=0. Die Operationen werden als Zahlen mit folgenden Werten dargestellt:

1. sem_op < 0
   Wenn der Absolutwert der Semaphore größer oder gleich dem vom sem_op ist, dann wird die Operation ausgeführt und sem_op wird zum Wert der Semaphore addiert (eigentlich subtrahiert, negative Zahl). Wenn der Absolutwert der Semaphore kleiner ist, dann schläft der Prozess bis eine solche Nummer verfügbar ist.
2. sem_op = 0
   Der Prozess schläft bis die Semaphore 0 ist.
3. sem_op > 0
   Der Wert von sem_op wird zur Semaphore addiert.

Im folgenden Programm implementieren wir das Pufferspeicher Beispiel. Wir erzeugen 5 Prozesse namens W (writer) und einen R (reader) Prozess. Jeder W Prozess versucht, Zugriff zu erhalten und speichert ein Element, falls der Puffer nicht voll ist. Danach gibt er den Zugriff wieder frei. R versucht auch exklusiven Zugriff zu erhalten und nimmt dann ein Element aus dem Puffer, falls er nicht leer ist. Danach gibt er den Zugriff ebenfalls wieder frei.

Lesen und Scheiben ist nur virtuell. Wie wir im vorangegangenen Artikel gesehen haben, hat jeder Prozess seinen eigenen Speicherbereich und kann nicht einfach auf Speicher anderer Prozesse zugreifen. Jeder Prozess sieht hier also eine Kopie des eigenen Speichers. Das ändert sich, wenn wir über shared memory sprechen, aber es ist besser einen Schritt nach dem anderen zu tun.

Warum brauchen wir 3 Semaphoren? Die erste stellt die Zugriffsrechte dar, und die anderen beiden brauchen wir für Inhalt und Überlauf (voll).

Wir haben also eine Semaphore (O), die die Anzahl der freien Plätze darstellt. Jedesmal, wenn W etwas in den Puffer legt, dann wird der Wert um eins erniedrigt bis Null erreicht ist (Puffer voll). R entnimmt dem Puffer die Werte und erhöht die Semaphore. Hier gibt es aber keine Grenze. R könnte sie unendlich erhöhen. Wir brauchen also noch eine Semaphore (U) für das obere Limit. Der Prozess W wird also O um eins erniedrigen und U um eins erhöhen. Im Gegensatz dazu wird R die Semaphore O um eins erhöhen und U erniedrigen.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(int argc, char *argv[])
{
  /* IPC */
  pid_t pid;
  key_t key;
  int semid;
  union semun arg;
  struct sembuf lock_res = {0, -1, 0};
  struct sembuf rel_res = {0, 1, 0};
  struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
  struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};

  /* Other */
  int i;

  if(argc < 2){
    printf("Usage: bufdemo [dimensione]\n");
    exit(0);
  }

  /* Semaphores */
```

```
  key = ftok("/etc/fstab", getpid());

  /* Create a semaphore set with 3 semaphore */
  semid = semget(key, 3, 0666 | IPC_CREAT);

  /* Initialize semaphore #0 to 1 - Resource controller */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* Initialize semaphore #1 to buf_length - Overflow controller */
  /* Sem value is 'free space in buffer' */
  arg.val = atol(argv[1]);
  semctl(semid, 1, SETVAL, arg);

  /* Initialize semaphore #2 to buf_length - Underflow controller */
  /* Sem value is 'elements in buffer' */
  arg.val = 0;
  semctl(semid, 2, SETVAL, arg);

  /* Fork */
  for (i = 0; i < 5; i++){
    pid = fork();
    if (!pid){
      for (i = 0; i < 20; i++){
        sleep(rand()%6);
        /* Try to lock resource - sem #0 */
        if (semop(semid, &lock_res, 1) == -1){
          perror("semop:lock_res");
        }
        /* Lock a free space - sem #1 / Put an element - sem #2*/
        if (semop(semid, &push, 2) != -1){
          printf("---> Process:%d\n", getpid());
        }
        else{
          printf("---> Process:%d  BUFFER FULL\n", getpid());
        }
        /* Release resource */
        semop(semid, &rel_res, 1);
      }
      exit(0);
    }
  }

  for (i = 0;i < 100; i++){
    sleep(rand()%3);
    /* Try to lock resource - sem #0 */
    if (semop(semid, &lock_res, 1) == -1){
      perror("semop:lock_res");
    }
    /* Unlock a free space - sem #1 / Get an element - sem #2 */
    if (semop(semid, &pop, 2) != -1){
      printf("<--- Process:%d\n", getpid());
    }
    else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
    /* Release resource */
    semop(semid, &rel_res, 1);
  }

  /* Destroy semaphores */
  semctl(semid, 0, IPC_RMID);

  return 0;
}
```

Nun will ich die interessanten Teile etwas erläutern:

```
struct sembuf lock_res = {0, -1, 0};
struct sembuf rel_res = {0, 1, 0};
struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};
```

In dieses 4 Zeilen sind die Aktionen, die man auf unsere Semaphoren anwenden kann. Die ersten zwei sind Einzelaktionen und die letzten zwei Doppelaktionen. lock_res versucht die Resource zu sperren, also exklusiven Zugriff zu erhalten: Es wird 1 von der ersten Semaphore (nummer 0) subtrahiert, wenn der Wert noch nicht Null ist. Das Verhalten (policy), wenn die Resource schon in Benutzung ist, ist "none" (der Prozess wartet). Die rel_res Aktion ist identisch zu lock_res, aber die Resource wird freigegeben (der Wert der Semaphore wird positiv).

Die Push und Pop Aktionen sind etwas komplexer. Es sind Felder mit zwei Aktionen. Die erste Aktion wird auf Semaphore Nummer 1 ausgeführt und die zweite auf Semaphore Nummer 2. Die erste wird hochgezählt und die zweite Semaphore heruntergezählt. Das Verhalten des Prozesses (policy) ist nicht mehr warten, sondern IPC_NOWAIT. Der Prozess arbeitet weiter und wird nicht blockiert.

```
/* Initialize semaphore #0 to 1 - Resource controller */
arg.val = 1;
semctl(semid, 0, SETVAL, arg);

/* Initialize semaphore #1 to buf_length - Overflow controller */
/* Sem value is 'free space in buffer' */
arg.val = atol(argv[1]);
semctl(semid, 1, SETVAL, arg);

/* Initialize semaphore #2 to buf_length - Underflow controller */
/* Sem value is 'elements in buffer' */
arg.val = 0;
semctl(semid, 2, SETVAL, arg);
```

Hier initialisieren wir den Wert der Semaphoren: Die Erste auf eins (exclusiver Zugriff), die Zweite erhält den Wert der Länge des Pufferspeichers, und die Dritte wird auf 0 gesetzt. Wie schon gesagt, sind 2 und 3 für Überlauf und leer.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
/* Lock a free space - sem #1 / Put an element - sem #2*/
if (semop(semid, &push, 2) != -1){
  printf("---> Process:%d\n", getpid());
}
else{
  printf("---> Process:%d  BUFFER FULL\n", getpid());
}
/* Release resource */
semop(semid, &rel_res, 1);
```

Der W Prozess versucht, Exklusivrechte zu erhalten mit der lock_res Aktion. Wenn das getan ist, wird eine Push Aktion ausgeführt und auf den Bildschirm wird etwas geschrieben, falls die Aktion nicht ausgeführt werden kann, weil der Puffer voll ist. Danach werden die Exklusivrechte wieder abgegeben.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
```

```
/* Unlock a free space - sem #1 / Get an element - sem #2 */
if (semop(semid, &pop, 2) != -1){
  printf("<--- Process:%d\n", getpid());
}
else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
/* Release resource */
semop(semid, &rel_res, 1);
```

Der R Prozess arbeitet wie der W Prozess nur wird ein Pop statt dem Push ausgeführt.

Im nächsten Artikel werden wir über "message queues", eine andere Struktur für InterProcess Communication (Kommunikation zwischen Prozessen) und Synchronisation sprechen. Wie immer, schick mir ein E–Mail, wenn du etwas geschrieben hast mit den Funktionen, die wir hier besprochen haben. Ich freue mich, es zu lesen. 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
- The Linux Programmer's Guide: http://www.tldp.org/LDP/lpg/index.html
- Linux Kernel 2.4 Internals http://www.tldp.org/LDP/lki/lki–5.html