Concurrent programming – Message Queue (1)

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

About the author:

Student an der Fakultät für Telecommunications Engineering am Polytechnikum in Milano, arbeitet als Netzwerk – Administrator und beschäftigt sich mit Programmieren (meistens in Assembler und CC++). Arbeitet seit 1999 fast ausschliesslich mit Linux / Unix.

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

Abstract:

Diese Artikelserie möchte den Leser in das Konzept des Multitasking und dessen Implementation in Linux einführen. Beginnend mit den theoretischen Konzepten, die dem Multitasking zugrunde liegen, werden wir zum Abschluss eine vollständige Anwendung in Form eines einfachen, jedoch wirkungsvollen Protokolls schreiben, welches die Kommunikation zwischen Prozessen demonstriert.

Voraussetzungen, um diesem Artikel folgen zu können:

- Einige (minimale) Kenntnisse der Shell
- Grundkenntnisse der C–Sprache (Syntax, Schleifen, Bibliotheken)

Es ist auch hilfreich, die ersten beiden Artikel dieser Serie gelesen zu haben, sie erschienen im LinusFocus vom November 2002 (Artikel Nr. 272) und Januar 2003 (Artikel Nr. 281).

Einführung

In den vorangegangenen zwei Artikeln führten wir in das Konzept des 'concurrent programming' ein und erläuterten eine erste Lösung zum Problem der Kommunikation zwischen Prozessen: die Semaphoren. Wie wir sahen, erlaubt uns die Anwendung von Semaphoren, den Zugang zu gemeinsam genutzten Ressourcen so zu kontrollieren, dass mindestens zwei oder mehr Prozesse synchronisiert werden.

Prozesse synchronisieren bedeutet, deren Ausführung zeitmässig festzulegen, d.h. nicht in einem absoluten Zeitsystem (das würde bedeuten, einen genauen Zeitpunkt für den Beginn des Prozessablaufs festzulegen),

sondern in einer geplanten Reihenfolge zu bestimmen, welcher Prozess zuerst beginnt und welcher als zweiter.

Hierfür Semaphoren zu benutzen, erweist sich als zu komplex und zu begrenzt: komplex, weil jeder Prozess sein Semaphore mit denen der anderen Prozesse synchronisieren müsste. Eingeschränkt, da es uns nicht den Austausch von Parametern zwischen den Prozessen gestattet. Nehmen wir das Beispiel des Aufbaus eines neuen Prozesses: dieses Ereignis sollte jedem beteiligten Prozess mitgeteilt werden, die Semaphoren erlauben einem Prozess nicht solche Informationen auszutauschen.

Die 'concurrency' – Regelung des Zugriffs zu den gemeinsamen Ressourcen durch die Semaphoren kann zu einer andauernden Blockierung eines Prozesses führen, wenn ein weiterer beteiligter Prozess die Ressource freigibt und wieder belegt, bevor ein anderer diese benutzen kann: wie wir erfuhren, ist es in der Welt des "concurrency programming" nicht möglich, im voraus zu wissen, welcher Prozess wann ausgeführt wird.

Hiermit wird klar, dass die Semaphoren ein ungeeignetes Werkzeug zur Behandlung von komplexen Synchronisationsproblemen sind. Eine elegantere Lösung für diese Aufgabe erhalten wir durch die Anwendung von 'Message Queues': in diesem Artikel lernen wir etwas Theorie über diese Vorrichtung zur Kommunikation zwischen Prozessen, ausserdem werden wir ein kleines Programm mit SysV– Primitiven schreiben.

Die Theorie der Message Queues

Jeder Prozess kann eine oder mehrere Datenstrukturen aufbauen, sie werden 'Queue' genannt: jede Datenstruktur kann eine oder mehrere Messages unterschiedlicher Art von verschiedenen Quellen enthalten. Jeder beteiligte Prozess kann Messages in die 'Queues' schicken, vorausgesetzt er kennt deren Identifikatoren. Der Prozess hat sequentiellen Zugriff (von der ältesten, zuerst angekommenen bis zur jüngsten, zuletzt eingetroffenen) auf die Queue, er liest die Messages in chronologischer Reihenfolge selektiv , d.h. nur Messages von einem gewissen Typ werden erwogen: diese letztere Eigenschaft ermöglicht uns eine gewisse Kontrolle über die Priorität der zu lesenden Messages.

Der Gebrauch von Queues ist insofern die einfache Anwendung eines Mailsystems zwischen Prozessen: jeder Prozess hat eine Adresse und er kann mit anderen Prozessen korrespondieren. Ein Prozess kann also Messages, die an ihn gesandt werden, in einer bestimmten Reihenfolge lesen und entsprechend den vorgefundenen Anforderungen handeln.

Die Synchronisation zweier Prozesse kann infolgedessen einfach durch Messages zwischen ihnen erfolgen: Ressourcen besitzen ausserdem Semaphoren, welche die Prozesse über ihren Status informieren, der zeitliche Ablauf zwischen den Prozessen wird jedoch direkt durchgeführt. Hier wird sofort verständlich, dass der Gebrauch von Message Queues sehr vereinfacht, was anfangs als ein äusserst komplexes Problem erschien.

Bevor wir die Message Queues in der C–Sprache implementieren, müssen wir noch ein anderes Problem erwähnen, das sich auf die Synchronisation bezieht: die Notwendigkeit eines Kommunikationsprotokolls.

Ein Protokoll zusammenstellen

Ein Protokoll ist eine Reihe von Regeln, welche die Zusammenarbeit von Elementen in einem Set behandeln. Im vorhergehenden Artikel haben wir eines der einfachsten Protokolle implementiert, indem wir ein Semaphore schufen und zwei Prozessen anordneten, gemäss ihrem Status zu handeln. Die Anwendung von

Message Queues erlaubt uns komplexere Protokolle anzuwenden: wir können uns einfach vorstellen, dass jedes Netzwerkprotokoll (TCP/IP, DNS, SMTP,...) auf einer "message exchange architektur" aufgebaut ist, selbst wenn die Kommunikation zwischen Computern und nicht innerhalb ihrer Systeme stattfindet. Der Vergleich ist zwingend: es besteht kein grundlegender Unterschied zwischen der Kommunikation von Prozessen in einer Maschine oder zwischen verschiedenen Maschinen. In einem zukünftigen Artikel werden wir sehen, dass die Übertragung des Konzepts – mit dem wir uns hier beschäftigen – auf eine erweiterte Anwendung (mehrere Computer miteinander verbunden) eine ziemlich einfache Angelegenheit ist.

Hier ist das einfache Beispiel eines Protokolls, welches auf dem Austausch von Messages basiert: die Prozesse A und B werden parallel ausgeführt und verarbeiten verschiedene Daten – sobald sie das abgeschlossen haben, müssen sie das Ergebnis kombinieren. Ein einfaches Protokoll für diesen Austausch könnte folgendermassen aussehen:

PROZESS B:

- Verarbeite deine Daten
- Wenn du damit fertig bist, schicke eine Message zu A
- Wenn A antwortet, beginne deine Resultate zu schicken

PROZESS A:

- Verarbeite deine Daten
- Warte auf eine Message von B
- Antworte auf die Message
- Empfange Daten und kombiniere sie mit deinen Daten

Die Bestimmung, welcher Prozess die Daten kombinieren muss, ist in diesem Fall willkürlich, normalerweise ergibt sich das aus der Funktion der beteiligten Prozesse (z.B. Client/Server). Näheres darüber sollte in einem eigenen Artikel behandelt werden.

Dieses Protokoll kann einfach auf 'n' Prozesse ausgedehnt werden: jeder Prozess, ausser A, bearbeitet seine eigenen Daten und schickt eine Message an A. A antwortet und jeder Prozess sendet seine Ergebnisse: die Struktur der individuellen Prozesse – ausser A – bleibt unverändert.

System V Message Queues

Jetzt werden wir auf die Anwendung dieser Konzepte im Linuxsystem eingehen. Wie wir schon bemerkten, haben wir einen Set von Primitiven, der uns erlaubt, mit den Strukturen der Message Queues und mit denen, die zu den Semaphoren gehören, zu arbeiten. Ich setze voraus, dass die Leser mit den grundsätzlichen Konzepten der Erstellung von Prozessen, der Anwendung von Systemaufrufen und IPC–Schlüsseln vertraut sind.

Die grundlegende Struktur des Systems, das eine Message beschreibt, ist mit `msgbuf` bezeichnet; ist in `linux/msg.h` deklariert

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
        long mtype;         /* type of message */
        char mtext[1];      /* message text */
```

```
};
```

Das Feld `mtype` repräsentiert die Art der Message und ist eine streng positive Zahl: der Zusammenhang zwischen den Zahlen und den Arten der Messages muss im voraus eingestellt werden, das ist Teil der Protokolldefinition. Das zweite Feld repräsentiert den Inhalt der Message, welcher jedoch in der Deklaration nicht aufgeführt werden muss. Die Struktur `msgbuf` kann so umdefiniert werden, dass sie komplexe Daten enthält; es ist zum Beispiel möglich zu schreiben

```
struct message {
        long mtype;         /* message type */
        long sender;        /* sender id */
        long receiver;      /* receiver id */
        struct info data;   /* message content */
        ...
};
```

Bevor wir uns mit den Argumenten beschäftigen, die sich direkt auf die Theorie der 'concurrency ' beziehen, müssen wir die Erstellung des Prototyp einer Message mit der fixierten Grösse von maximal 4056 Bytes planen. Es ist natürlich jederzeit möglich, den Kernel zu rekompilieren und diese Grösse zu erhöhen, das macht die Anwendung jedoch nicht portabel (ausserdem wird die Leistung nicht vorteilhaft beeinflusst, deshalb bleiben wir lieber bei der definierten Grösse).

Um eine neue Queue zu erstellen, sollte ein Prozess einen Aufruf an die `msgget()` Funktion machen

```
int msgget(key_t key, int msgflg)
```

welche als Argumente einen IPC−Schlüssel und einige Flags erhält, die nun auf

```
IPC_CREAT | 0660
```

eingestellt werden können (erzeuge die Queue, falls sie noch nicht existiert , erlaube Zugriff für den Besitzer und die Gruppenmitglieder), was den Identifikator für die Queue zurückgibt.

Wie in den vorangegangenen Artikeln nehmen wir an, dass keine Fehler auftreten, so können wir den Code vereinfachen − in einem zukünftigen Artikel werden wir uns mit sicherem IPC−Code beschäftigen.

Um eine Message an eine Queue mit bekanntem Identifikator zu schicken, benutzen wir das `msgsnd()` Primitivum

```
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg)
```

wobei `msqid` der Identifikator der Queue und `msgp` ein Zeiger auf die Message, die wir schicken wollen, ist (deren Typ hier mit `struct msgbuf` identifiziert wird − deren Typ wir jedoch umdefiniert haben), `msgsz` enthält die Dimension der Message (ausschliesslich der Länge des `mtype` Feldes, die 4 Byte beträgt) und `msgflg` −ein Flag, das sich auf die Wait−Richtlinien bezieht. Die Länge der Message kann einfach festgestellt werden durch

```
length = sizeof(struct message) - sizeof(long);
```

Die Wait−Richtlinie bezieht sich auf den Fall einer vollen Queue: falls `msgflg` auf IPC_NOWAIT eingestellt ist, wartet der Absender nicht auf freiwerdenden Platz, sondern er beendigt mit einem Fehlercode −

wir werden weiter darauf eingehen, wenn wir über Fehlermanagement sprechen.

Um die Messages in der Queue zu lesen, benutzen wir den `msgrcv()` –System – Aufruf

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long mtype,
int msgflg)
```

wobei der `msgp` – Zeiger den Puffer identifiziert, in welchen wir die Messages kopieren, die aus der Queue gelesen werden und `mtype` identifiziert eine Untermenge der Messages, die wir betrachten wollen.

Eine Queue kann man entfernen mit dem Primitivum `msgctl()` bei Verwendung des Flag IPC_RMID

```
msgctl(qid, IPC_RMID, 0)
```

Wir können das mit einem einfachen Programm testen, es erzeugt eine Message Queue, sendet und liest diese – auf diese Weise werden wir das korrekte Verhalten des Systems prüfen.

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

/* Redefines the struct msgbuf */
typedef struct mymsgbuf
{
  long mtype;
  int int_num;
  float float_num;
  char ch;
} mess_t;

int main()
{
  int qid;
  key_t msgkey;

  mess_t sent;
  mess_t received;

  int length;

  /* Initializes the seed of the pseudo-random number generator */
  srand (time (0));

  /* Length of the message */
  length = sizeof(mess_t) - sizeof(long);

  msgkey = ftok(".",'m');

  /* Creates the Queue*/
  qid = msgget(msgkey, IPC_CREAT | 0660);

  printf("QID = %d\n", qid);

  /* Builds a message */
  sent.mtype = 1;
  sent.int_num = rand();
  sent.float_num = (float)(rand())/3;
  sent.carattere = 'f';
```

```
  /* Sends the message */
  msgsnd(qid, &sent, length, 0);
  printf("MESSAGE SENT...\n");

  /* Receives the message */
  msgrcv(qid, &received, length, sent.mtype, 0);
  printf("MESSAGE RECEIVED...\n");

  /* Controls that received and sent messages are equal */
  printf("Interger number = %d (sent %d) -- ", received.int_num,
         sent.int_num);
  if(received.int_num == sent.int_num) printf(" OK\n");
  else printf("ERROR\n");

  printf("Float numero = %f (sent %f) -- ", received.float_num,
         sent.float_num);
  if(received.float_num == sent.float_num) printf(" OK\n");
  else printf("ERROR\n");

  printf("Char = %c (sent %c) -- ", received.ch, sent.ch);
  if(received.ch == sent.ch) printf(" OK\n");
  else printf("ERROR\n");

  /* Destroys the Queue */
  msgctl(qid, IPC_RMID, 0);
}
```

Jetzt können wir zwei Prozesse anlegen und diese miteinder durch eine Queue kommunizieren lassen; erinnern wir uns an das Prozess−Forking−Konzept: der Wert aller Variablen, der durch den Vater−Prozess zugewiesen wurde, wird auf den Sohn−Prozess übertragen (memory copy). Daraus folgt, wir sollten die Queue vor dem Father−fork−Prozess und bevor der Sohn den Identifikator der Queue erfährt und diesen anspricht, einrichten.

Mein Code erzeugt eine Queue, die vom Sohn−Prozess benutzt wird, um Daten zum Vater zu schicken: der Sohn erzeugt Zufallszahlen, überträgt diese zum Vater und beide drucken diese als Standard−Ausgabe.

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

/* Redefines the message structure */
typedef struct mymsgbuf
{
  long mtype;
  int num;
} mess_t;

int main()
{
  int qid;
  key_t msgkey;
  pid_t pid;

  mess_t buf;

  int length;
  int cont;
```

```
  length = sizeof(mess_t) - sizeof(long);

  msgkey = ftok(".",'m');

  qid = msgget(msgkey, IPC_CREAT | 0660);

  if(!(pid = fork())){
    printf("FATHER - QID = %d\n", qid);

    srand (time (0));

    for(cont = 0; cont < 10; cont++){
      sleep (rand()%4);
      buf.mtype = 1;
      buf.num = rand()%100;
      msgsnd(qid, &buf, length, 0);
      printf("SON - MESSAGE NUMBER %d: %d\n", cont+1, buf.num);
    }

    return 0;
  }

  printf("FATHER - QID = %d\n", qid);

  for(cont = 0; cont < 10; cont++){
    sleep (rand()%4);
    msgrcv(qid, &buf, length, 1, 0);
    printf("FATHER - MESSAGGE NUMBER %d: %d\n", cont+1, buf.num);
  }

  msgctl(qid, IPC_RMID, 0);

  return 0;
}
```

Wir erzeugten also zwei Prozesse, die auf elementarer Basis über ein Message−Exchange−System zusammenarbeiten können. Wir benötigten kein formales Protokoll, weil die ausgeführten Operationen sehr einfach waren. Im nächsten Artikel werden wir uns wieder mit Message Queues und den verschiedenen Messagetypen beschäftigen. Wir werden auch am Kommunikationsprotokoll arbeiten, um mit dem Erstellen unseres grossen IPC−Projekts zu beginnen (ein Telefon−Switch−Simulator).

Empfohlene Literatur

- 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
- Web page of the #kernelnewbies IRC channel http://www.kernelnewbies.org/
- The linux−kernel mailing list FAQ http://www.tux.org/lkml/

Translation information:
it −−> −− : Leonardo Giordani <leo.giordani(at)libero.it>
it −−> en: Leonardo Giordani <leo.giordani(at)libero.it>
en −−> de: Jürgen Pohl <sept.sapins/at/verizon.net>

2005−01−11, generated by lfparser_pdf version 2.51