Concurrent programming – Message Queues (2)

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

About the author:

Vor kurzem erhielt ich mein Diplom von der Fakultät für Telekommunikations – Engineering am Politecnico in Milano. Mein Hauptinteresse liegt im Programmieren (Assembler und C/C++). Seit 1999 arbeite ich fast ausschliesslich mit Linux/Unix.

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 Protokolls schreiben, welches die Kommunikation zwischen Prozessen demonstriert. Voraussetzungen, um diesem Artikel zu folgen:

- Einige Kenntnisse der Shell
- Grundkenntnisse in C (Syntax, Schleifen, Bibliotheken)

Alle Referenzen auf Manual Pages sind in Klammern hinter den Befehlen zu finden. Alle *glibc*–Funktionen sind in "info Libc" dokumentiert.

Es ist sicher auch eine gute Idee, einige der vorherigen Artikel aus dieser Serie gelesen zu haben:

- Concurrent programming – Prinzipien und Einführung in Prozesse
- Concurrent programming – Kommunikation zwischen Prozessen
- Concurrent programming – Message Queues (1)

Einführung

Im letzten Artikel dieser Serie haben wir gelernt, wie wir zwei (oder mehr) Prozesse synchronisieren und durch die Anwendung von Message Queues zusammenarbeiten lassen können. In diesem Artikel werden wir

weitergehen und ein einfaches Protokoll für unseren Message−Exchange entwickeln.

Wie ich schon erklärte, ist ein Protokoll eine Reihe von Regeln, die es Menschen oder Maschinen erlaubt, miteinander zu kommunizieren − selbst wenn diese unterschiedlich sind. So ist z.B. ist die Anwendung einer Sprache (wie Englisch) ein Protokoll, durch dieses ist es möglich, mich an meine indischen Leser zu wenden (die immer sehr daran interessiert sind, was ich mitzuteilen habe). Um wieder auf Linux zurückzukommen: wenn wir unseren Kernel rekompilieren (keine Angst, das ist nicht so schwierig), bemerkst du sicher den Networking−Teil, in dem wir unserem Kernel beibringen können, mehrere Netzwerkprotokolle zu verstehen, wie z.B. TCP/IP.

Um ein Protokoll zu entwerfen, müssen wir uns entscheiden, welche Art von Anwendungen wir entwickeln wollen. Dieses Mal schreiben wir einen einfachen Telefon−Vermittlungs−Simulator. Der Hauptprozess wird die Telefon−Vermittlung sein, während der Sohn−Prozess als User auftritt: wir werden den Usern ermöglichen, Messages über die Vermittlungsstelle auszutauschen.

Das Protokoll wird drei verschiedene Situationen abdecken: die Einführung eines Users (d.h. der User existiert und ist verbunden), die gewöhnliche Arbeit des Users und das Entfernen des Users (er ist nicht mehr verbunden). Gehen wir also näher auf diese drei Fälle ein:

Wenn sich ein User mit dem System verbindet, erzeugt er seine eigene Message Queue (nicht vergessen: wir sprechen über Prozesse), seine Identifier müssen der Vermittlungsstelle mitgeteilt werden, damit dieser weiss, wie er den User erreichen kann. Hier werden auch einige Strukturen oder Daten initialisiert. Von der Vermittlungsstelle erhält der User den Identifier einer Queue, in welcher er die Meldungen schreibt, die durch die Vermittlungsstelle zu anderen Benutzern übertragen werden sollen.

Der User kann Meldungen senden und empfangen. Wenn er Meldungen zu einem anderen User schickt, können zwei verschiedene Fälle auftreten: entweder ist der Empfänger verbunden oder nicht. Wir entscheiden, dass in beiden Fällen eine Bestätigung an den Absender geschickt werden soll, um ihn wissen zu lassen, was mit der Meldung geschehen ist. Das erfordert keine Aktion des Empfängers, die Vermittlungsstelle sollte diese Tätigkeit ausführen.

Wenn sich ein User vom System trennt, sollte er dieses der Vermittlungsstelle mitteilen, aber sonst sind keine weiteren Schritte nötig. Der Metacode für diese Arbeitsweise sieht folgendermassen aus

```
/* Birth */
create_queue
init
send_alive
send_queue_id
get_switch_queue_id

/* Work */
while(!leaving){
 receive_all
 if(<send condition>){
  send_message
 }
 if(<leave condition>){
  leaving = 1
 }
}

/* Death */
send_dead
```

Jetzt müssen wir noch das Verhalten unserere Teefon−Vermittlungsstelle (englisch Switch) bestimmen: verbindet sich ein User, schickt er diesem eine Meldung mit dem Identifier seiner Message Queue. Wir müssen diesen also speichern, um Meldungen an diesen User übertragen zu können und um einen Identifier der Queue für die Meldungen an die anderen User zu übertragen. Für alle Meldungen, die wir von den Usern erhalten, müssen wir überprüfen, ob die Empfänger verbunden sind: ist das der Fall, wird die Meldung geliefert. Ist der Empfänger nicht verbunden, wird die Meldung weggeworfen – in beiden Fällen soll das dem Absender mitgeteilt werden. Wird ein User abgehängt, wird einfach der Identifier der Queue entfernt, d.h. er wird unerreichbar.

Die Metacode−Implementation sieht wieder folgendermassen aus

```
while(1){
 /* New user */
 if (<birth of a user>){
  get_queue_id
  send switch_queue_id
 }

 /* User dies */
 if (<death of a user>){
  remove_user
 }

 /* Messages delivering */
 check_message
 if (<user alive>){
  send_message
  ack_sender_ok
 }
 else{
  ack_sender_error
 }
}
```

Fehlerbehandlung

Die Fehlerbehandlung ist eine der schwierigsten und wichtigsten Massnahmen in einem Projekt. Darüberhinaus stellt ein gutes, vollständiges Fehler− Prüfsystem bis zu 50% des Codes dar. Ich werde in diesem Artikel nicht darauf eingehen, wie man gute Routinen zur Fehlerprüfung entwickelt – das Gebiet ist zu komplex, aber von jetzt an werde ich stets nach Fehlern suchen und Fehlerzustände verwalten. Eine gute Einführung in Fehlerprüfung enthält die Anleitung zu *glibc* (www.gnu.org) – falls Interesse besteht, werde ich einen Artikel darüber schreiben.

Protokol−Implementation – Layer 1

Unser kleines Protokoll hat zwei Schichten: die erste (unterste) besteht aus Funktionen, welche die Queues verwalten und um Meldungen vorzubereiten und zu senden. Die darüberliegende Schicht implementiert das Protokoll in ähnlicher Weise wie der Metacode, den wir zur Beschreibung des Verhaltens von Vermittlungsstelle und User benutzten.

Zuerst definieren wir eine Struktur für unsere Meldung, indem wir den Kernel−Prototyp von *msgbuf* benutzen

```
typedef struct
{
  int service;
  int sender;
  int receiver;
  int data;
} messg_t;

typedef struct
{
  long mtype; /* Tipo del messaggio */
  messg_t messaggio;
} mymsgbuf_t;
```

Hier ist etwas in genereller Ausführung, wir können es später erweitern: die Absender– und Empfänger–Felder enthalten einen User–identifier. Das Datenfeld beinhaltet allgemeine Daten, während das Service–Feld benutzt wird, um Dienste anzufordern. Wir können uns zum Beispiel zwei Dienste vorstellen: einen für unmittelbare, den anderen für verzögerte Zustellung – in diesem Fall würde das Datenfeld die Verzögerung in Sekunden enthalten. Das ist nur ein Beispiel, aber es läßt uns verstehen, das es viele Möglichkeiten gibt, das Service–Feld zu benutzen.

Nun können wir einige Funktionen zum Verwalten unserer Datenstrukturen implementieren, vor allem, um die Felder für die Meldungen herzustellen. Diese Funktionen sind mehr oder weniger gleich, ich werde also nur zwei davon beschreiben, die anderen sind in den .h files zu finden.

```
void set_sender(mymsgbuf_t * buf, int sender)
{
  buf->message.sender = sender;
}

int get_sender(mymsgbuf_t * buf)
{
  return(buf->message.sender);
}
```

Diese Funktion (sie besteht nur aus einer Zeile) hat nicht die Aufgabe, den Code zu komprimieren: wir können uns leichter an ihre Bedeutung erinnern, das Protokoll ist der menschlichen Sprache ähnlicher und daher einfacher zu benutzen.

Jetzt schreiben wir noch die Funktionen, die den IPC–Schlüssel erzeugen, Message Queues erzeugen und entfernen, Meldungen abschicken und empfangen. Einen IPC–Schlüssel zu bauen ist einfach:

```
key_t build_key(char c)
{
  key_t key;
  key = ftok(".", c);
  return(key);
}
```

Eine Queue erzeugen wir mit der folgenden Funktion

```
int create_queue(key_t key)
{
  int qid;

  if((qid = msgget(key, IPC_CREAT | 0660)) == -1){
    perror("msgget");
    exit(1);
  }
```

```
  return(qid);
}
```

wie wir sehen, ist die Fehlerbehandlung in diesem Fall sehr einfach. Die folgende Funktion beseitigt eine Queue

```
int remove_queue(int qid)
{
  if(msgctl(qid, IPC_RMID, 0) == -1)
  {
    perror("msgctl");
    exit(1);
  }
  return(0);
}
```

Zuletzt die Funktion, um Meldungen zu empfangen und zu senden: eine Meldung senden, heisst für uns, diese zu einer bestimmten Queue zu schreiben, d.h. jene, die wir von Vermittlungsstelle haben.

```
int send_message(int qid, mymsgbuf_t *qbuf)
{
  int result, lenght;
  lenght = sizeof(mymsgbuf_t) - sizeof(long);
  if ((result = msgsnd(qid, qbuf, lenght, 0)) == -1){
    perror("msgsnd");
    exit(1);
  }

  return(result);
}

int receive_message(int qid, long type, mymsgbuf_t *qbuf)
{
  int result, length;
  length = sizeof(mymsgbuf_t) - sizeof(long);

  if((result = msgrcv(qid, (struct msgbuf *)qbuf,
      length, type, IPC_NOWAIT)) == -1){
    if(errno == ENOMSG){
      return(0);
    }
    else{
      perror("msgrcv");
      exit(1);
    }
  }

  return(result);
}
```

Das wär's. Wir finden die Funktionen in der Datei layer1.h: versuchen wir doch mal, damit ein Programm zu entwickeln (z.B. das aus dem vorangegangenen Artikel). Im nächsten Artikel beschäftigen wir uns mit der zweiten Schicht des Protokolls und dessen Implementation.

Empfohlener Lesestoff

- 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/**

Wie immer könnt ihr mir Kommentare, Berichtigungen und Fragen an meine E−mail−Adresse (leo.giordani(at)libero.it) oder durch die Talkback− Seite zukommen lassen. Bitte schreibt in Englisch, Deutsch oder Italienisch.

2005−01−11, generated by lfparser_pdf version 2.51