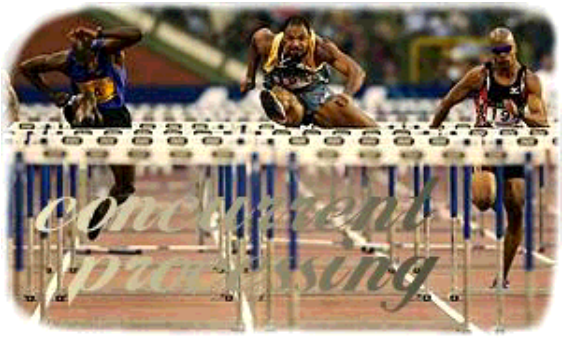
LinuxFocus article number 296
http://linuxfocus.org

Concurrent programming - Message queues (2)



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

About the author:

I just received my diploma from the Faculty of Telecommunication Engineering in Politecnico of Milan. Interested in programming (mostly in Assembly and C/C++). Since 1999 works almost only with Linux/Unix.

Abstract:

This series of articles has the purpose of introducing the reader to the concept of multitasking and to its implementation in the Linux operating system. Starting from the theorical concepts at the base of multitasking we will end up writing a complete application demonstrating the communication between processes, with a simply but efficient communication protocol. Prerequisites for the understanding of the article are:

- Minimal knowledge of the shell use
- Basic knowledege of C language (sintax, loops, libraries)

All references to manual pages are placed between parenthesis after the command name. All the glibc functions are documented through "info Libc".

It might be also a good idea to read some of the previous article in this series first:

- Concurrent programming - Principles and introduction to processes
- Concurrent programming - Communications between processes
- Concurrent programming - Message queues (1)

Introduction

In the last article of this little series we learned how to let two (or more) processes sunchronize and work together through the use of message queues. In this one we will go further and begin creating a simple protocol for our message exchange.

We already said that a protocol is a set of rules that let people or machines talk, even if they are different. For example the use of language English is a protocol, because it allows me to speak to my indian readers (who are always very interested in what I write). Speaking about something more Linux-related, if you recompile your kernel (do not be afraid, it is not so difficult), you will surely notice the Networking section, in which you can let your kernel understand several network protocols, such as TCP/IP.

In order to create a protocol we have to decide what type of application we will develop. This time we will write a simple telephone switch simulator. A main process will be the telephone switch, and the son processes will act as users: we will let users send messages to each other through the switch.

The protocol will cover three different situations: the birth of a user (i.e. the user exists and is connected), the user's ordinary work, and the death of a user (he is no more connected). Let's talk about these three cases:

When a user connects to the system he creates his own message queue (do not forget we are speaking about processes), its identifiers should be sent to the switch in order to let it know how to reach the user. Here it has time to initialize some structures or data if it need them. It receives from the switch the identifier of a queue where it can write the messages that should be delivered to other users through the switch .

The user can send and receive messages. When it sends a message to another user we can encounter two different cases: the receiver is connected or not. We decide that in both cases an aknowledgement should be delivered to the sender, to let it know what happened to its message. This implies no actions by the receiver itself, the switch should do this work.

When a user disconnects from the system he should notify the switch but no more actions are needed. The metacode that describe this way of working is the following

```
/* 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
```

Now we have to define the behaviour of our telephone switch: when a user connects it sends us a message containing the identifier of its message queue; thus, we have to store it, in order to deliver messages sent to this user, and answer sending it the identifier of a queue where it can write the message that we have to deliver to other users. Than we have to analyze all messages received from the users and to check if the receivers are alive: if the receiver is connected we should deliver the message, if the receiver is not connected we have to discard the message; in both cases we should aknowledge the sender. When a user dies we simply remove the identifier of its queue, so that it becomes unreachable.

Again, the metacode implementation is

```
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
 }
}
```

Error handling

Handle error conditions is one of the most difficult and important things to do in a project. Moreover, a good and complete error checking subsystem takes up to 50% of the code we write. I will not explain in this article how to do develop good error checking routines, because the matter is too complex, but from now on I will always check and manage error conditions. A good introduction in error checking comes from the reading of the glibc manual (www.gnu.org) but, if you are interested, I will write an article about this.

Protocol implementation - Layer 1

Our little protocol has two layers: the first one (the lowest) consists of functions to manage queues and to prepare and send messages, while the higher layer implements the protocol as functions similar to the metacode we used to describe the behaviour of the switch and the users.

The first thing to do is to define a structure for our message using the kernel prototype of msgbuf

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

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

This is something general we can later extend: the sender and the receiver fields contain a user identifier and the data field contains general data, while the service field is used to request a service to the switch. For example we could imagine we have two services: one for immediate and one for delayed delivering, in which case the data field could transport the number of seconds of delay. This is only an example, but let us understand that the service field gives us many possibilities.

Now we can implement some functions to manage our data structures, particularly to set and get the fields of the messages. These functions are more or less all the same, so I give you only two of them, and you will find the others in the .h files

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

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

The aim of these function is not that of compress the code (they consist of only one line of code): they are simply to remember their meaning and let the protocol become nearer to human language, and thus simplier to use.

Now we have to write functions to generate IPC keys, create and remove message queues, send and receive massages: biuld an IPC key is simple

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

Then the function to create a queue

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

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

```
    exit(1);
  }

  return(qid);
}
```

as you can see error handling is in this case very simply. The following function destroys a queue

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

And last the functions to get and send messages: sending a message means for us writing it on a particular queue, i.e. the one given to us from the switch.

```
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);
}
```

That’s all. You will find the functions in the file layer1.h: try to create some program (e.g. that of the past article) using them. In the next article we will speak about layer 2 of the protocol and implement it.

Recommended readings

- 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/**

As always you can send me comments, corrections and questions at my mail address (leo.giordani(at)libero.it) or through the Talkback page. Plese write me in english, german or italian.

2005-01-14, generated by lfparser_pdf version 2.51