LinuxFocus article number 287
http://linuxfocus.org

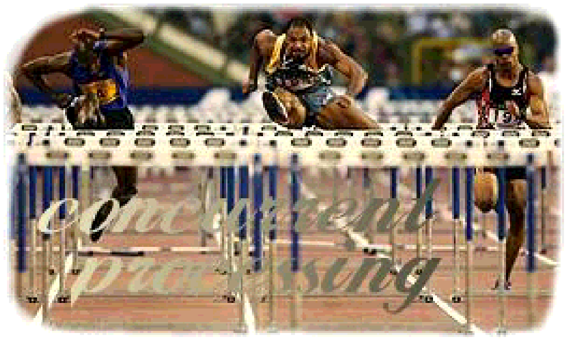
Concurrent programming - Message queues (1)

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

About the author:

Student at the Faculty of Telecommunication Engineering in Politecnico of Milan, works as network administrator and is interested in programming (mostly in Assembly and C/C++). Since 1999 works almost only with Linux/Unix.

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



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 theoretical concepts at the base of multitasking we will end up writing a complete application demonstrating the communication between processes, with a simple but efficient communication protocol.

Prerequisites for the understanding of the article are:

- Minimal knowledge of the shell
- Basic knowledge of C language (syntax, loops, libraries)

It is a good idea to read also the other articles in this series which appeared in the last 2 issues of LinuxFocus (November2002 and January2003).

Introduction

In the past articles we introduced the concept of concurrent programming and studied a first solution to the problem of inter process communication: the semaphores. As we saw, the use of semaphores allows us to manage the access to shared resources, so that it roughly synchronizes two or more processes.

Synchronizing processes means timing their work, not in an absolute reference system (giving a precise time in which the process should begin its operations) but in a relative one, where we can schedule which process should work first and which second.

Using semaphores for this reveals itself as complex and limited: complex because every process should manage a semaphore for every other process that has to synchronize with it. Limited because it does not allow us the exchange parameters between the processes. Let’s consider for example the creation of a new process: this event should be notified to every working process, but semaphores do not allow a process to send such information.

The concurrency control of the access to shared resources through semaphores, moreover, can lead to continuous blocking of a process, when one of the other processes involved release the resource and lock it again before others can use it: as we saw, in the world of concurrency programming it is not possible to know in advance which process will be executed and when.

These brief notes let us immediately understand that semaphores are an inadequate tool for managing complex synchronization problems. An elegant solution to this matter comes with the use of message queues: in this article we will study the theory of this interprocess communication facility and write a little program using SysV primitives.

The Theory of Message Queues

Every process can create one or more structures named queues: Every structure can hold one or more messages of different type, which can originate from different sources and can contain information of every nature; everyone can send a message to the queues provided that he knows its identifier. The process can access sequentially the queue, reading the messages in chronological order (from the oldest, the first, to the most recent, the last arrived), but selectively, that is considering only the messages of a certain type: this last feature gives us a sort of control on the priority of the messages we read.

The use of queues is thus a simple implementation of a mail system between processes: every process has an address with which it can other processes. The process can then read the messages delivered to its box in a preferential order and act accorting to what has been notified.

The synchronization of two processes can thus be performed simply using messages between the two: resources will still own semaphores to let the processes know their status, but timing between processes will be performed directly. Immediately we can understand that the use of message queues simplified very much what at the beginning was a extremely complex problem.

Before we can implement in C language the message queues it is necessary to speak about another problem related to synchronzation: the need for a communication protocol.

Creating a Protocol

A protocol is a set of rules which control the interaction of elements in a set; in the past article we implemented one of the simplest protocols creating a semaphore and ordering two processes to act according to its status. The use of message queues lets us implement more complex protocols: it is sufficient to think that every network protocol (TCP/IP, DNS, SMTP, ...) is built on a message exchange architecture, even if the communication is between computers and not internal to one of them. The comparison is compulsory: there is not a real difference between interprocess communication on the same machine and between machines. As we will see in a future article extending the concepts we are speaking about to a distributed contest (several computers connected) is a very simple matter.

This is a simple example of a protocol based on message exchange: two processes A and B are executing concurrently and process different data; once they end their processing the have to merge the results. A simple protocol to rule their interaction could be the following

PROCESS B:

- Work with your data
- When you finish send a message to A
- When A answers begin sending it your results

PROCESS A:

- Work with your data
- Wait for a message from B
- Answer the message
- Receive data and merge them with yours

Choosing which process has to merge data is in this case totally arbitrary; commonly this happens on the basis of the nature of the process involved (client/server) but this discussion deserves a dedicated article.

This protocol is simply extensible to the case of n processes: every process but A works with its own data and then sends a message to A. When A answers every process sends it its results: the structure of the individual processes (except A) has not been modified.

System V Message Queues

Now it is time to speak about implementing these concepts in the Linux operating system. As already said we have a set of primitives that allow us to manage the structures related to message queues and that works as those given to manage semaphores: I will thus assume that the reader knows the basic concepts related to process creation, use of system calls and IPC keys.

The structure at the basis of the system describing a message is called `msgbuf` ;it is declared in `linux/msg.h`

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

The field mtype represents the type of message and is a strictly positive number: the correspondence between numbers and message types has to be set in advance and is part of the protocol definition. The second field represents the content of the message but not have to be considered in the declaration. The structure `msgbuf` can be redefined so that it can contain complex data; for example it is possible to write

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

Before we face the arguments strictly related to concurrency theory we have to consider creating the prototype of a message with the maximum size, fixed to 4056 bytes. Obviously it is always possible to recompile the kernel increasing this dimension, but this makes the application unportable (more over this bound has been fixed to grant good performances and increasing it much certainly is not good).

To create a new queue a process should call the `msgget()` function

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

which receives as arguments an IPC key and some flags, which by now can be set to

```
IPC_CREAT | 0660
```

(create the queue if it does not exist and grant access to the owner and group users), and that returns the queue identifier.

As in the previous articles we will assume that no errors will happen, so that we can simplify the code, even if in a future article we will speak about secure IPC code.

To send a message to a queue of which we know the identifier we have to use the `msgsnd()` primitive

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

where `msqid` is the identifier of the queue, `msgp` is a pointer to the message we have to send (which type is here identified as `struct msgbuf` but which is the type we redefined), `msgsz` the dimension of the message (excluding the length of the `mtype` that is the length of a long, which is commonly 4 bytes) and `msgflg` a flag related to the waiting policy. The length of the message can be easily be found as

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

while the waiting policy refers to the case of full queue: if `msgflg` is set to IPC_NOWAIT the sender process will not wait until some space is available and will exit with an error code; we will speak about such a case when we will talk about error management.

To read the messages contained in a queue we use the `msgrcv()` system call

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

where the `msgp` pointer identifies the buffer where we will copy the message read from the queue and `mtype` identifies the subset of messages we want to consider.

Removing a queue can be performed through the use of the `msgctl()` primitive with the flag IPC_RMID

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

Let’s test what we said with a simple program wich creates a message queue, sends a message and reads it; we will control that way the correct working of the system.

```
#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.ch = ’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("Integer 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);
}
```

Now we can create two processes and let them communicate through a message queue; think a bit about process forking concepts: the value of all variables allocated by the father process is taken to those of the son process (memory copy). This means we should create the queue before the fork the father process and the son will known the queue identifier and thus access it.

The code I wrote creates a queue used by the son process to send its data to the father: the son generates random numbers, sends them to the father and both print them on the standard output.

```
#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("SON - 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 - MESSAGE NUMBER %d: %d\n", cont+1, buf.num);
  }

  msgctl(qid, IPC_RMID, 0);

  return 0;
}
```

We created thus two processes, which can collaborate in an elementary manner through a message exchange system. We didn’t need a (formal) protocol because the operations performed were very simple; in the next article we will speak again about message queues and about managing different message types. We will work moreover on the communication protocol in order to begin the building of our big IPC project (a telephone switch simulator).

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/
