LinuxFocus article number 281
http://linuxfocus.org

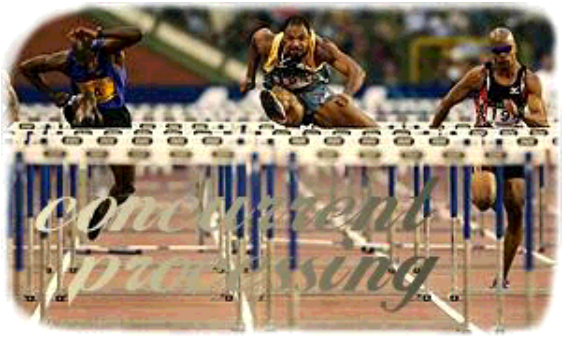
Concurrent programming - communication between processes

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 knowledege of C language (syntax, loops, libraries)

You should read the first article in this series because it is a base for this one: November 2002, article 272.

Introduction

Here we are again struggling with Linux multitasking. As we saw in the preceeding article forking the execution of a program needs just few code lines, because the operating system takes care of initialization, managing and timing of the processes we create.

This service provided by the operating system is fundamental, it is 'the supervisory of processes' execution; thus, processes are executed in a dedicated environment. Losing the control on execution of the processes brings to the developer a synchronization problem, summarized by this question: how is it

possible to let two independent processes work together?

The problem is more complex than it seems: it is not only a question of synchronisation of the execution of the processes, but also of sharing data, both in read- and in write-mode.

Let’s speak about some classical problems of concurrent data access; if two processes read the same dataset this is obviously not a problem, and the execution is CONSISTENT. Now let one of the two processes modify the dataset: the other one will return different results according to the time at which it reads the dataset, before or after the writing by the first process. For example: we have two processes "A" and "B" and an integer "d". The process A increases d by 1, the process B prints it out. Writing it in a meta language we can expess it in this way

A { d->d+1 } & B { d->output }

where the "&" indentifies a concurrent execution. A first possible execution is

(-) d = 5 (A) d = 6 (B) output = 6

but if the process B is executed first we will obtain

(-) d = 5 (B) output = 5 (A) d = 6

You understand immediately how important it is to manage correctly these situations: the risk of INCONSISTENCY of data is big and inacceptable. Try to think that the datasets represent your bank account and you will never underestimate this problem.

In the preceeding article we already spoke about a first form of synchronisation through the use of the waitpid(2) function, which let a process wait for the termination of another one before going on. In fact this allow us to solve some of the conflicts raised about data read and write: once the dataset on which a process P1 will work has been defined, a process P2 which works on the same dataset or on a subset of it shall wait for the termination of P1 before if can proceed with its own execution.

Clearly this method represent a first solution, but is far off from the best, because P2 have to stay idle for a time which can be very long, waiting that P1 terminates its execution, even if it is no more working on common data. Thus, we must increase the granularity of our control, i.e. rule the access to single data or data set. The solution to this problem is given by a set of primitives of the standard library known as SysV IPC (System V InterProcess Communication).

SysV keys

Before we face the arguments strictly related to concurrency theory and its implementations let us introduce a typical SysV structure: IPC keys. An IPC key is a number used to identify unequivocally an IPC control structure (described further), but it can also be used in order to generate generic identifiers, i.e. to organize no-IPC structures. A key can be created with the ftok(3) function

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

which uses the name of an existing file (pathname) and an integer. It is not assured that the key is

unique, because the parameters taken from the file (i-node number and device number) can create identical combinations. A good solution is to create a little library which traces the assigned keys and avoids duplicates.

Semaphores

The idea of semaphore for the car traffic control can be used without great modifications for data access control. A semaphore is a particular structure containing a value grater or equal to zero and that manages a queue of processes waiting for a particular condition on the semaphore itself. Even if it seems simple semaphores are very powerful and consequently complications increase. Let us start (as always) leaving error control out: we will put it in our code when we will face a more complex program.

Semaphores can be used to control resource access: the value of the semaphore represents the number of processes which can access the resource; any time a process accesses the resource the value of the semaphore shall be decremented and incremented again when the resource is released. If the resource is exclusive (i.e. only one process can access it) the initial value of the semaphore will be 1.

A different task can be accomplished by the semaphore, the resource counter: the value it represents, in this case, the number of resources available (for example the number of free memory cells).

Let’s consider a practical case, in which the semaphore types will be used: imagine we have a buffer in which several processes S1,...,Sn can write but from which only a process L can read; moreover, operations cannot be accomplished at the same time (i.e. at a given time only one process is operating on the buffer). Obviously S processes can always write except when the buffer is full, while the process L can read only if the buffer is not empty. Thus, we need three semaphores: the first will manage the access to the resource, the second and the third will keep track of how many elements are in the buffer (we will see later why two semaphores are not sufficient).

Considering that the access to the buffer is exclusive the first semaphore will be a binary one (its value will be 0 or 1), while the second and the third will assume values related to the dimension of the buffer.

Let’s learn how semaphores are implemented in C using SysV primitives. The function that creates a semaphore is semget(2)

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

where key is an IPC key, nsems is the number of semaphores we want to create and semflg is the access control implemented with 12 bits, the first 3 being related to creation policies and the other 9 to read and write access by user, group and other (notice the similarity to the Unix filesystem); for a complete description read the man page of ipc(5). As you can notice SysV manage set of semaphores instead of single ones, resulting in a more compact code.

Let’s create our first semaphore

```
#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;
}
```

Going further we have to learn how to manage and remove semaphores; the management of the semaphore is performed by the primitive semctl(2)

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

which operats according to the action identified by cmd on the set semid and (if requested by the action) on the single semaphore semnum. We will introduce some options when we will need them, but a complete list can be found on the man page. Depending on the cmd action it could be necessary to specify another argument for the function, whose type is

```
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 */
};
```

To set the value of a semaphore the SETVAL directive should be used and the value has to be specified in the union semun; let’s modify the preceeding program setting the semaphore’s value to 1

```
[...]

  /* 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);

[...]
```

Then we have to release the semaphore deallocating the structures used for its management; this task is accomplished by the directive IPC_RMID of semctl. This directive removes the semaphore and sends a message to all the processes waiting to gain access to the resource. A last modification to the program is

```
[...]

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

  /* deallocate semaphore */
```

```
  semctl(semid, 0, IPC_RMID);
[...]
```

As seen before creating and managing a structure for controlling concurrenct execution is not difficult; when we will introduce error management things will become more complex, but only from a code complexity point of view.

The semaphore can now be used through the function semop(2)

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

where semid is the set identifier, sops an array containing operations to be performed and nsops the number of these operations. Every operation is represented by a sembuf struct.

unsigned short sem_num; short sem_op; short sem_flg;

i.e. by the semaphore number in set (sem_num), the operation (sem_op) and a flag setting the wait policy; for now let sem_flg be 0. The operations we can specify are integer numbers and follow these rules:

1. sem_op < 0
   If the absolute value of the semaphore is greater than or equal to that of sem_op the operation goes on and sem_op is added to the value of the semaphore (actually it is subtracted, negative number). If the absolute value of sem_op is greater than the value of the semaphore the process falls in a sleep state until such a number of resources is available.
2. sem_op = 0
   The process sleeps until the value of the semaphore reaches 0.
3. sem_op > 0
   The value of sem_op is added to the value of the semaphore, releasing the resources previously taken.

The following program tries to show how to use semaphores implementing the previous buffer example: we will create 5 processes called W (writers) and a process R (reader). Each W process tries to gain the control of the resource (the buffer) locking it through a semaphore and, if the buffer is not full, puts an element into it and releases the resource. The R process tries to lock the resource, takes an element if the buffer is not empty and unlocks the resource.

Read and write of the buffer are only virtual: this happens because, as seen in the preceeding article, every process has its own memory space and cannot access that of another process. This makes the correct management of the buffer with 5 processes impossible, because each one will see its own copy of the buffer. It will change when we will speak about shared memory but let’s learn things step by step.

Why do we need 3 semaphores? The first (number 0) acts as a buffer access lock and has a maximum value of 1, while the other two manage the overflow and underflow conditions. A single semaphore cannot manage both situations, because semop acts one-way.

Let’s clarify the matter: with one semaphore (called O), which value represents the number of empty spaces in the buffer. Every time an S process puts something in the buffer it decreases the value of the

semaphore by one, until the values reaches zero, i.e. the buffer is full. This semaphore cannot manage the underflow condition: the R process, in fact, can increase its value without limits. We need thus a special semaphore (called U), which value represents the number of elements in the buffer. Every time a W process puts an element in the buffer it will also increase the value of the U semaphore and decrease that of the O semaphore. On the contrary, the R process will decrease the value of the U semaphore and increase that of the O semaphore.

The overflow condition is thus identified by the impossibility of decreasing the O semaphore and the underflow condition by the impossibility of decreasing th U semaphore.

```
#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;
}
```

Let’s comment the more interesting parts of the code:

```
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};
```

These 4 lines are the actions we can perform on our semaphore set: the first two are single actions, while the others are double. The first action, lock_res, tries to lock the resource: it decreases the value of the first semaphore (number 0) by a value of 1 (if the value is not zero) and the policy adopted if the resource is busy is none (i.e. the process waits). The rel_res action is identical to lock_res but the resource is released (the value is positive).

The push and pop actions are a bit special. They are arrays of two actions, the first on the semaphore number 1 and the second on the semaphore number 2; while the first is incremented the second is

decremented and viceversa, but the policy is no more a wait one: IPC_NOWAIT forces the process to continue execution if the resource is busy.

```
/* 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);
```

Here we initialize the value of the semaphores: the first to 1 because it controls the access to an exclusive resource, the second to the length of the buffer (given on the command line) and the third to 0, as said before about over- and underflow.

```
/* 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);
```

The W process tries to lock the resource through the lock_res action; once this is done it performs a push and tells it on the standard output: if the operation cannot be performed it prints that the buffer is full. After that it releases the resource.

```
/* 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);
```

The R process acts more or less as the W process: locks the resource, performs a pop and releases the resource.

In the next article we will speak about message queues, another structure for the InterProcess Communication and synchronisation. As always if you write something simple using what you learned from this article send it to me, with your name and your e-mail address, I will be happy to read it. Good

work!

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

---

2005-01-14, generated by lfparser_pdf version 2.51