LinuxFocus article number 272
http://linuxfocus.org

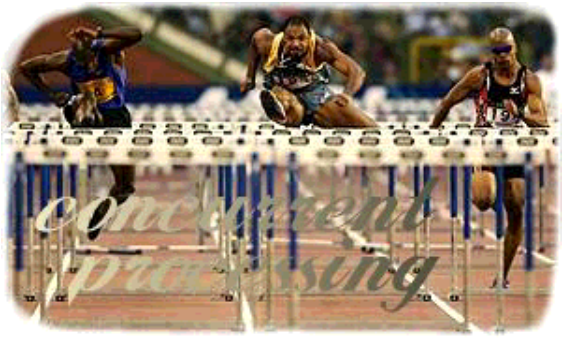
Concurrent programming - Principles and introduction to processes

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 theorical 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)

All references to manual pages are placed between parenthesis after the command name. All the glibc functions are documented in gnu info pages (info Libc, or type info:/libc/Top in konqueror).

Introduction

One of the most important turning points in the history of operating systems was the concept of multiprogramming, a technique for interlacing the execution of several programs in order to gain a more constant use of the system's resources. Let's think of a simply workstation, where a user can execute at the same time a wordprocessor, an audio player, a print queue, a web browser and more. It's an important concept for modern operating systems. As we will discover this little list is only a minimal part of the set of programs that are currently executing on our machine, even though the most "visual-striking".

The concept of process

In order to interlace programs a remarkable complication of the operating system is necessary; in order to avoid conflicts between running programs an unavoidable choice is to encapsulate each of them with all the information needed for their execution.

Before we explore what happens in our Linux box, let's give some technical nomenclature: given a running **PROGRAM**, at a given time the **CODE** is the set of instructions which it's made of, the **MEMORY SPACE** is the part of machine memory taken up by its data and the **PROCESSOR STATUS** is the value of the microprocessor's parameters, such as the flags or the Program Counter (the address of the next instruction to be executed).

We define the term **RUNNING PROGRAM** as a number of objects made of CODE, MEMORY SPACE and PROCESSOR STATUS. If at a certain time during the operation of the machine we will save this informations and replace them with the same set of information taken from another running program, the flow of the latter will continue from the point at which it was stopped: doing this once with the first program and once with the second provides for the interlacing we described before. The term **PROCESS** (or **TASK**) is used to describe such a running program.

Let's explain what was happening to the workstation we spoke about in the introduction: at each moment only a task is in execution (there is only a microprocessor and it cannot do two things at the same time), and the machine executes part of its code; after a certain amount of time named QUANTUM the running process is suspended, its informations are saved and replaced by those of another waiting process, whose code will be executed for a quantum of time, and so on. This is what we call multitasking.

As stated before the introduction of multitasking causes a set of problems, most of which are not trivial, such as the waiting processes queues management (SCHEDULING); nevertheless they have to do with the architecture of each operating system: perhaps this will be the main topic of a further article, maybe introducing some parts of the Linux kernel code.

Processes in Linux and Unix

Let's discover something about the processes running on our machine. The command which gives us such informations is **ps(1)** which is an acronym for "process status". Opening a normal text shell and typing the ps command we will obtain an output such as

```
  PID TTY          TIME CMD
 2241 ttyp4    00:00:00 bash
 2346 ttyp4    00:00:00 ps
```

I state in before that this list is not complete, but let's concentrate on this for the moment: ps has given us the list of each process running on the current terminal. We recognize in the last column the name by which the process is started (such as "mozilla" for Mozilla Web Browser and "gcc" for the GNU Compiler Collection). Obviously "ps" appears in the list becouse it was running when the list of running processes was printed. The other listed process is the Bourne Again Shell, the shell running on my terminals.

Let’s leave out (for the moment) the information about TIME and TTY and let’s look at PID, the Process IDentifier. The pid is a unique positive number (not zero) which is assigned to each running process; once the process has been terminated the pid can be reused, but we are guaranteed that during the execution of a process its pid remains the same. All this implies is that the output each of you will obtain from the ps command will probably be different from that in the example above. To test that I am saying the truth, let’s open another shell without closing the first one and type the ps command: this time the output gives the same list of processes but with different pid numbers, testifying that they are two different processes even if the program is the same.

We can also obtain a list of all processes running on our Linux box: the ps command man page says that the switch -e means "select all processes". Let’s type "ps -e" in a terminal and ps will print out a long list formatted as seen above. In order to confortably analyze this list we can redirect the output of ps in the ps.log file:

```
ps -e > ps.log
```

Now we can read this file editing it with our preferred editor (or simply with the less command); as stated at the beginning of this article the number of running processes is higher than we would expect. We actually note that list contains not only processes started by us (throught the command line or our graphical environment) but also a set of processes, some of which with strange names: the number and the identity of the listed processes depends on the configuration of your system, but there are some common things. First of all, no matter what type of configuration you gave to the system, the process with pid equal to 1 is always "init", the father of all the processes; it owns the pid number 1 because it is always the first process executed by the operating system. Another thing we can easily note is the presence of many processes, whose name ends with a "d": they are the so called "daemons" and are some of the most important processes of the system. We will study in detail init and the daemons in a further article.

Mutitasking in the libc

We understand now the concept of process and how important it is for our operating system: we will go on and begin to write mutitasking code; from the trivial simultaneous execution of processes we will shift towards a new problem: the communication between concurrent processes and their synchronization; we will discover two elegant solutions to this problem, messages and semaphores, but the latters will be deeply explained in a further article about the threads. After the messages it will be the time to begin writing our application based on all these concepts.

The standard C library (libc, implemented in Linux with the glibc) uses the Unix System V multitasking facilities; the Unix System V (from now on SysV) is a commercial Unix implementation, is the founder of one of the two most important families of Unix, the other being BSD Unix.

In the libc the pid_t type is defined as an integer capable of containing a pid. From now on we will use it to bear the value of a pid, but only for clarity’s sake: using an integer is the same thing.

Let’s discover the function which give us the knowledge of the pid of the process containing our program.

```
pid_t getpid (void)
```

(which is defined with pid_t in unistd.h and sys/types.h) and write a program whose aim is to print on the standard output its pid. With an editor of your choice write the following code

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main()
{
  pid_t pid;

  pid = getpid();
  printf("The pid assigned to the process is %d\n", pid);

  return 0;
}
```

Save the program as print_pid.c and compile it

```
gcc -Wall -o print_pid print_pid.c
```

this will build an executable named print_pid. I remind you that if the current directory is not in the path it is necessary to run the program as "./print_pid". Executing the program we will have no great surprises: it prints out a positive number and, if executed more than once, you see that this number will increase one by one; this is not mandatory, because another process can be created from a program between an execution of print_pid and the following. Try, for example, to execute ps between two executions of print_pid...

Now it’s time to learn how to create a process, but I have to spend some more words about what really happens during this action. When a program (contained in the process A) creates another process (B) the two are identical, that is they have the same code, the memory full of the same data (not the same memory) and the same processor status. From this point on the two can execute in two different ways, for example depending on the user’s input or some random data. The process A is the "father process" while the B is the "son process"; now we can better understand the name "father of all the processes" given to init. The function which creates a new process is

```
pid_t fork(void)
```

and its name comes from the property of forking the execution of the process. The number returned is a pid, but deserves a particular attention. We said that the present process duplicates itself in a father and a son, which will execute interlacing themselves with the other running processes, doing different works; but immediately after the duplication which process will be executed, the father or the son? Well, the answer is simply: one of the two. The decision of which process has to be executed is taken by a part of the operating system called scheduler, and it pays no attention if a process is the father or the son, following an algorithm based on other parameters.

Anyway, it is important knowing what process is in execution, because the code is the same. Both processes will contain the father’s code and the son’s one, but each of them has to execute only one of this codes. In order to clarify this concept let’s look at the following algorithm:

```
- FORK
- IF YOU ARE THE SON EXECUTE (...)
- IF YOU ARE THE FATHER EXECUTE (...)
```

which represents in a sort of meta language the code of our program. Let's unveil the mistery: the fork function returns '0' to the son process and the son's pid to the father. So it is sufficient to test if the returned pid is zero and we will know what process is executing that code. Putting it in C language we obtain

```
int main()
{
  pid_t pid;

  pid = fork();
  if (pid == 0)
  {
    CODE OF THE SON PROCESS
  }
  CODE OF THE FATHER PROCESS
}
```

It's time to write the first real example of multitasking code: you can save it in a fork_demo.c file and compile it as done before. I put line numbers only for clarity. The program will fork itself and both the father and the son will write something on the screen; the final output will be the intelacing of the two output (if all goes right).

```
(01) #include <unistd.h>
(02) #include <sys/types.h>
(03) #include <stdio.h>

(04) int main()
(05) {
(05)   pid_t pid;
(06)   int i;

(07)   pid = fork();

(08)   if (pid == 0){
(09)     for (i = 0; i  < 8; i++){
(10)       printf("-SON-\n");
(11)     }
(12)     return(0);
(13)   }

(14)   for (i = 0; i < 8; i++){
(15)     printf("+FATHER+\n");
(16)   }

(17)   return(0);
(18) }
```

Lines number (01)-(03) contain the includes for the necessary libraries (standard I/O, multitasking). The main (as always in GNU), returns an integer, which normally is zero if the program reached the end without errors or an error code if something goes wrong; let's state this time all will run without errors (we will add error control when the basic concepts will be clear). Then we define the data type containing a pid (05) and an integer working as counter for loops (06). These two types, as stated before, are identical, but they are here for clarity's sake.
At line (07) we call the fork function which will return zero to the program executed in the son process and the pid of the son process to the father; the test is at line (08). Now the code at lines (09)-(13) will be

executed in the son process, while the rest (14)-(16) will be executed in the father.
The two parts simply write 8 times on the standard output the word "-SON-" or "+FATHER+", depending on which process executes it, and then ends up returning 0. This is really important, because without this last "return" the son process, once the loop has ended, would go further executing the father's code (try it, it does not harm your machine, simply it does not do what we want). Such an error will be really difficult to find, since the execution of a multitasking program (especially a complex one) gives different results at each execution, making debugging based on results simply impossible.

Executing the program you will perhaps be unsatisfied: I cannot assure you that the result will be a real mix between the two strings, and this due to the speed of execution of such a short loop. Probably your output will be a succession of "+FATHER+" strings followed by a "-SON-" one or the contrary. Try however to execute more than once the program and the result may change.

Inserting a random delay before every printf call, we may obtain a more visual multitasking effect: we do this with the sleep and the rand function.

```
sleep(rand()%4)
```

this makes the program sleep for a random number of seconds between 0 and 3 (% returns the remainder of the integer division). Now the code looks as

```
(09)  for (i = 0; i < 8; i++){
(->)    sleep (rand()%4);
(10)    printf("-FIGLIO-\n");
(11)  }
```

and the same for the father's code. Save it as fork_demo2.c, compile and execute it. It is slower now, but we notice a difference in the output order:

```
[leo@mobile ipc2]$ ./fork_demo2
-SON-
+FATHER+
+FATHER+
-SON-
-SON-
+FATHER+
+FATHER+
-SON-
-FIGLIO-
+FATHER+
+FATHER+
-SON-
-SON-
-SON-
+FATHER+
+FATHER+
[leo@mobile ipc2]$
```

Now let us look at the problems we have to face now: we can create a certain number of son processes given a father process, so that they execute operations different from those executed by the father process himself in a concurrent processing environment; often the father needs to communicate with sons or at least to synchronize with them, in order to execute operations at the right time. A first way to obtain such a synchronization between processes is the wait function

```
pid_t waitpid (pid_t PID, int *STATUS_PTR, int OPTIONS)
```

where PID is the PID of the process whose end we are waiting for, STATUS_PTR a pointer to an integer which will contain the status of the son process (NULL if the information is not needed) and OPTIONS a set of options we have not to care about for now. This is an example of a program in which the father creates a son process and waits until it ends

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main()
{
  pid_t pid;
  int i;

  pid = fork();

  if (pid == 0){
    for (i = 0; i < 14; i++){
      sleep (rand()%4);
      printf("-SON-\n");
    }
    return 0;
  }

  sleep (rand()%4);
  printf("+FATHER+ Waiting for son's termination...\n");
  waitpid (pid, NULL, 0);
  printf("+FATHER+ ...ended\n");

  return 0;
}
```

The sleep function in the father's code has been inserted to differentiate executions. Let's save the code as fork_demo3.c, compile it and execute it. We just wrote our first multitasking synchronized application!

In the next article we'll learn more about synchronization and communication between processes; now write your programs using described functions and send me them so that I can use some of them to show good solutions or bad errors. Send me both the .c file with the commented code and a little text file with a description of the program, your name and your e-mail address. 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

Translation information:
it --> -- : Leonardo Giordani <leo.giordani(at)libero.it>
it --> en: Leonardo Giordani <leo.giordani(at)libero.it>

2005-01-14, generated by lfparser_pdf version 2.51