Multi-processing

Dinesh Sharma
EE Department, IIT Bombay

February 16, 2023

Multi-processing implies that multiple programs are active at the same time. A scheduler is used to run these programs one after the other, so that they all appear to be running simultaneously. However, when a processor boots, a single program runs. So how do we get multi-processing?

We show how a user level program can launch other programs on a unix platform. Other operating system use similar function calls for this purpose. Launching a new process involves calls to two separate system provided functions in this example, – "fork" and "execlp".

Please review the introduction to processes first. Recall that a process is different from an executable program stored on disk. When a program is loaded in order to run it, memory is allocated for its code, data, stack and bss segments and a process page is created for it. The process is given a unique process id number called pid. The process page contains information about the running process, including its pid, the program counter value etc.

System provided function "fork"

As the name implies, "fork" duplicates a process to give two processes which are all but identical, except for a few details.

Now consider the operation of the function "fork".

At the time of making a call to this function there was only one process. This process made a call to the function "fork", which created a fresh process page with a new process id and with independent data, stack and bss segments. After "fork" has done its job, it should return to the process which called it.

But wait!
At the time of return, there are TWO processes, the original process and its copy. The original process is called the parent process and the copy is called the child process. Both processes are now active, and will be scheduled to run by a scheduler whenever their turn comes to run. Both process are in the same state – they are both waiting for a return from the call to "fork"! So where does the processing resume on return from "fork"?

In fact, "fork" returns to both functions! This function returns a value to the calling program. The calling program will assign the return value to some variable. The parent process as well as the child process will do this, whenever they run. The return value to the parent process when "fork" returns is the process id of the child process. On the other hand, the return value to the freshly created child process is 0. Anticipating this, when the program using "fork" is written, it uses an if then else kind of construct which runs different blocks of code depending on the return value from "fork". If it is non zero, the code block associated with this condition executes code which the parent program was supposed to do. On the other hand, if the return value is 0, the code block associated with this condition does what the child process was supposed to do.

This may be a good time to learn more details about this function on your linux machine by the command:

man -a fork

The following example program illustrates what happens when you "fork" a process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int a = 10; /* global initialized variable in data segment */
int b; /* global uninitialized variable in BSS segment */
int
main (int argc, char *argv[])
{
  int c; /* local variable c in Stack segment or in reg */
  pid_t retval;

  b = 20;
  c = 30;
  retval = fork ();
  if (retval == 0)
    {
      printf ("\nfork returned %d. So this is the child process.\n", retval);
      printf ("Variables: a = %d, b = %d, c = %d.\n", a, b, c);
      a = 15;
      b = 25;
      c = 35;
      printf ("Changed variables to %d, %d and  %d.\n", a, b, c);
      return (0);
    }
  else
    {
      printf ("\nfork returned %d. So this is the parent process.\n", retval);
      printf ("Variables: a = %d, b = %d, c =%d.\n", a, b, c);
      a = 100;
      b = 200;
      c = 300;
      printf ("Changed variables to %d, %d and  %d.\n", a, b, c);
      return (0);
    }
}
```

Open a terminal and copy the code above in somefile.c. compile this file using cc and then run the resulting executable file. (In Linux, you will run ./a.out if you did not give a -o option to cc).

Clearly, the child inherits all the data stored in the parent. The data structure in the child is as defined in the parent. However, it gets a **copy** of parents data. Parent and child are now independent processes and can change their data in their own data/bss/stack segments, without affecting the other process.

Exec family of functions

We can use fork if we write both the parent and child code, because both must be included in a single C program. What should we do if we want to launch an executable program which is available on the system, but whose source code is not with us?

This can be done by combining fork with a system call to one of the family of system functions which are collectively known as exec.

A call to exec completely replaces a running process with a new one, created from a specified executable binary. Different members of the exec family differ in How the program and the arguments to be passed to it are specified. Since the original calling process is completely overwritten, there is no return value from these functions. (Where will they return? The calling program has been destroyed!) The newly created process runs from its beginning.
Here we shall use the function execlp, which provides the arguments to the new program explicitly in the call itself as null terminated strings.

With this introduction, you can now learn other details about these functions by reading their man page. On a linux machine, you can type the command

man -a execlp

To understand the arguments to the call, you should review the argv, argc mechanism used in 'C' to handle arguments passed to a program.
execlp is called with the following arguments:

1. path to the executable file which will be loaded to run.
2. name by which this program is being invoked. Some programs take different actions depending on the name by which these are called. For example, compress and decompress programs are the same binary, but carry out different actions depending on whether they were called by the name compress or decompress.
3. comma separated list of arguments to the program. Each item in this list is a null terminated string. the number of these arguments can be variable.
4. A NULL. This terminates the list of arguments. Since the number of arguments can be any thing, this NULL signals the end of arguments and permits the evaluation of "argc", which is the count of arguments used in this invocation.

The following program illustrates how execlp is combined with fork to spawn a new process. For this illustration, we have used the program gnuplot which will be spawned. gnuplot is a program which plots data. It receives its commands from a text file, which should be given as one of its arguments.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <sys/wait.h>
FILE *pltfile;
int
main (int argc, char *argv[])
{
  double x[128], y[128];
  double angle;
  int i;
  pid_t new_pid;
  /* We intend to use the spawned process for plotting data which the parent
     program generates. */

  pltfile = fopen ("plt.tmp", "w");

  /* The main program generates data and writes to pltfile:
     The code below is just an example which fills the array with sine values.
  */
  for (i = 0; i < 128; i++)
    {
      angle = 3.1415926 * i / 16;
      x[i] = angle;
      y[i] = sin (angle);
      fprintf (pltfile, "%12g %12g\n", x[i], y[i]);
    }
  fclose (pltfile);

  /*  Now spawn gnuplot - a plotting program */
  /*  The first step is to use fork to generate a copy of parent program. */
  new_pid = fork ();
  if (new_pid == 0)
    { /* Child process */
      i =
execlp ("gnuplot",/* Path to the binary -- will use default path*/
"gnuplot",  /* name of the program (argv[0])*/
"-geometry", "400x300-150+150", /* arguments specifying window size */
"testplot",/* name of the file containing commands */
NULL); /* terminating argument */
    }
  else
    {
      wait (NULL);
      printf("Back from first invocation of gnuplot\n");
      pltfile = fopen ("plt.tmp", "w");
      for (i = 0; i < 128; i++)
{
  angle = 3.1415926 * i / 16;
  x[i] = angle;
  y[i] = exp (-angle/8.0) * cos (angle);
  fprintf (pltfile, "%12g %12g\n", x[i], y[i]);
}
      fclose (pltfile);
      /* spawn gnuplot again */
      new_pid = fork ();
      if (new_pid == 0)
{ /* Child process */
  i =
    execlp ("gnuplot", "gnuplot", "-geometry", "400x300-150+150",
    "testplot", NULL);
}
      else
{
  wait (NULL);
          printf("Back from second invocation of gnuplot\n");
}
    }
}
```

Open a terminal and copy the code above in somefile.c. compile this file using cc, linking with the math library (-lm) and then run the resulting executable file.
(In Linux, you will run ./a.out if you did not give a -o option to cc).

gnuplot expects its commands in a file called testplot. (Notice that the plot data is in another file called plt.tmp). testplot contains the following commands:

```
plot "plt.tmp" with lines
pause -1
```

When gnuplot runs, it reads its commands from the text file called testplot (as given in the arguments used for execl). testplot asks gnuplot to plot the data in plt.tmp with lines and then waits for an end of line from the user (pause command). When the user presses q to exit and then presses the enter key, gnuplot terminates and we are back to the parent program.

In the current example, after plotting the sin(x) data, the parent program computes and plots a damped cosine function.