Creación y duplicación de procesos en C (Parte I) – Linux

[Rodrigo Paszniuk](https://www.programacion.com.py/author/raps) | 2013-08-17 | [1 Comment](https://www.programacion.com.py/escritorio/c/creacion-y-duplicacion-de-procesos-en-c-linux#disqus_thread)

Una situación muy habitual dentro de un programa es la de crear un nuevo proceso que se encargue de una tarea concreta, descargando al proceso principal de tareas secundarias que pueden realizarse asíncronamente o en paralelo. Linux ofrece varias funciones para realizar esto: system(), fork() y exec().

**Con system() nuestro programa consigue detener su ejecución para llamar a un comando de la shell (“/bin/sh” típicamente) y retornar cuando éste haya acabado. Si la shell no está disponible, retorna el valor 127, o –1 si se produce un error de otro tipo. Si todo ha ido bien, system() devuelve el valor de retorno del comando ejecutado. Su prototipo es el siguiente:**

int system(const char \*string);

Donde “string” es la cadena que contiene el comando que queremos ejecutar, por ejemplo:

system(“clear”);

Esta llamada limpiaría de caracteres la terminal, llamando al comando “clear”. Este tipo de llamadas a system() son muy peligrosas, ya que si no indicamos el PATH completo (“/usr/bin/clear”), alguien que conozca nuestra llamada (bien porque analiza el comportamiento del programa, bien por usar el comando strings, bien porque es muy muy muy sagaz), podría modificar el PATH para que apunte a su comando clear y no al del sistema (imaginemos que el programa en cuestión tiene privilegios de root y ese clear se cambia por una copia de /bin/sh: el intruso conseguiría una shell de root).

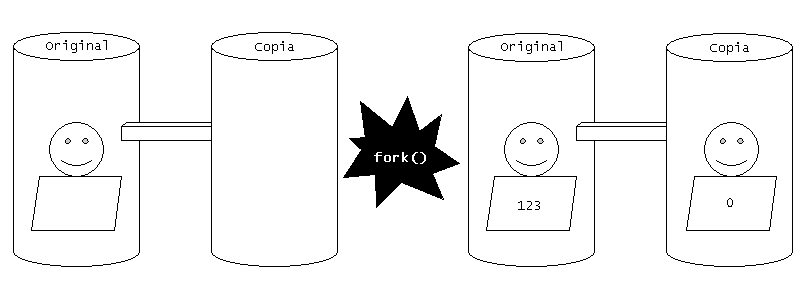
La función system() bloquea el programa hasta que retorna, y además tiene problemas de seguridad implícitos, por lo que desaconsejo su uso más allá de programas simples y sin importancia.

**La segunda manera de crear nuevos procesos es mediante fork(). Esta función crea un proceso nuevo o “proceso hijo” que es exactamente igual que el “proceso padre”. Si fork() se ejecuta con éxito devuelve:**

* Al padre: el PID del proceso hijo creado.
* Al hijo: el valor 0.

Para entendernos, fork() clona los procesos (bueno, realmente es clone() quien clona los procesos, pero fork() hace algo bastante similar). Es como una máquina para replicar personas: en una de las dos cabinas de nuestra máquina entra una persona con una pizarra en la mano. Se activa la máquina y esa persona es clonada. En la cabina contigua hay una persona idéntica a la primera, con sus mismos recuerdos, misma edad, mismo aspecto, etc. pero al salir de la máquina, las dos copias miran sus pizarras y en la de la persona original está el número de copia de la persona copiada y en l

a de la “persona copia” hay un cero

[](http://www.programacion.com.py/wp-content/uploads/2013/08/cursoc03.gif)

**Duplicación de procesos mediante fork().**

En la anterior figura vemos como nuestro incauto voluntario entra en la máquina replicadora con la pizarra en blanco. Cuando la activamos, tras una descarga de neutrinos capaz de provocarle anginas a Radiactivoman, obtenemos una copia exacta en la otra cabina, sólo que en cada una de las pizarras la máquina ha impreso valores diferentes: “123”, es decir, el identificativo de la copia, en la pizarra del original, y un “0” en la pizarra de la copia. No hace falta decir que suele ser bastante traumático salir de una máquina como esta y comprobar que tu pizarra tiene un “0”, darte cuenta que no eres más que una vulgar copia en este mundo. Por suerte, los procesos no se deprimen y siguen funcionando correctamente.

Veamos el uso de fork() con un sencillo ejemplo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <sys/types.h>  #include <unistd.h>  #include <stdio.h>    int main(int argc, char \*argv[])  {      pid\_t pid;        if ( (pid=fork()) == 0 )      { /\* hijo \*/          printf("Soy el hijo (%d, hijo de %d)\n", getpid(),          getppid());      }      else      { /\* padre \*/          printf("Soy el padre (%d, hijo de %d)\n", getpid(),          getppid());      }        return 0;  } |

Guardamos en la variable “pid” el resultado de fork(). Si es 0, resulta que estamos en el proceso hijo, por lo que haremos lo que tenga que hacer el hijo. Si es distinto de cero, estamos dentro del proceso padre, por lo tanto todo el código que vaya en la parte “else” de esa condicional sólo se ejecutará en el proceso padre. La salida de la ejecución de este programa es la siguiente:

txipi@neon:~$ gcc fork.c –o fork

txipi@neon:~$ ./fork

Soy el padre (569, hijo de 314)

Soy el hijo (570, hijo de 569)

txipi@neon:~$ pgrep bash

314

La salida de las dos llamadas a printf(), la del padre y la del hijo, son asíncronas, es decir, podría haber salido primero la del hijo, ya que está corriendo en un proceso separado, que puede ejecutarse antes en un entorno multiprogramado. El hijo, 570, afirma ser hijo de 569, y su padre, 569, es a su vez hijo de la shell en la que nos encontramos, 314. Si quisiéramos que el padre esperara a alguno de sus hijos deberemos dotar de sincronismo a este programa, utilizando las siguientes funciones:

pid\_t wait(int \*status)

pid\_t waitpid(pid\_t pid, int \*status, int options);

La primera de ellas espera a cualquiera de los hijos y devuelve en la variable entera “status” el estado de salida del hijo (si el hijo ha acabado su ejecución sin error, lo normal es que haya devuelto cero). La segunda función, waitpid(), espera a un hijo en concreto, el que especifiquemos en “pid”. Ese PID o identificativo de proceso lo obtendremos al hacer la llamada a fork() para ese hijo en concreto, por lo que conviene guardar el valor devuelto por fork(). En el siguiente ejemplo combinaremos la llamada a waitpid() con la creación de un árbol de procesos más complejo, con un padre y dos hijos:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #include <sys/types.h>  #include <unistd.h>  #include <stdio.h>    int main(int argc, char \*argv[])  {      pid\_t pid1, pid2;      int status1, status2;        if ( (pid1=fork()) == 0 )      { /\* hijo \*/          printf("Soy el primer hijo (%d, hijo de %d)\n",  getpid(), getppid());      }      else      { /\*  padre \*/          if ( (pid2=fork()) == 0 )          { /\* segundo hijo  \*/              printf("Soy el segundo hijo (%d, hijo de %d)\n",  getpid(), getppid());          }          else          { /\* padre \*/  /\* Esperamos al primer hijo \*/              waitpid(pid1, &status1, 0);  /\* Esperamos al segundo hijo \*/              waitpid(pid2, &status2, 0);              printf("Soy el padre (%d, hijo de %d)\n", getpid(), getppid());          }      }        return 0;  } |

El resultado de la ejecución de este programa es este:

txipi@neon:~$ gcc doshijos.c –o doshijos

txipi@neon:~$ ./ doshijos

Soy el primer hijo (15503, hijo de 15502)

Soy el segundo hijo (15504, hijo de 15502)

Soy el padre (15502, hijo de 15471)

txipi@neon:~$ pgrep bash

15471

Con waitpid() aseguramos que el padre va a esperar a sus dos hijos antes de continuar, por lo que el mensaje de “Soy el padre…” siempre saldrá el último.

Se pueden crear árboles de procesos más complejos, veamos un ejemplo de un proceso hijo que tiene a su vez otro hijo, es decir, de un proceso abuelo, otro padre y otro hijo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <sys/types.h>  #include <unistd.h>  #include <stdio.h>    int main(int argc, char \*argv[])  {      pid\_t pid1, pid2;      int status1, status2;        if ( (pid1=fork()) == 0 )      { /\* hijo (1a generacion) = padre \*/          if ( (pid2=fork()) == 0 )          { /\* hijo (2a generacion)  = nieto \*/              printf("Soy el nieto (%d, hijo de %d)\n",  getpid(), getppid());          }          else          { /\* padre (2a generacion) = padre \*/              wait(&status2);              printf("Soy el padre (%d, hijo de %d)\n",  getpid(), getppid());          }      }      else      { /\* padre (1a generacion) = abuelo \*/          wait(&status1);          printf("Soy el abuelo (%d, hijo de %d)\n", getpid(),  getppid());      }        return 0;  } |

Y el resultado de su ejecución sería:

txipi@neon:~$ gcc hijopadrenieto.c -o hijopadrenieto

txipi@neon:~$ ./hijopadrenieto

Soy el nieto (15565, hijo de 15564)

Soy el padre (15564, hijo de 15563)

Soy el abuelo (15563, hijo de 15471)

txipi@neon:~$ pgrep bash

15471

Tal y como hemos dispuesto las llamadas a wait(), paradójicamente el abuelo esperará a que se muera su hijo (es decir, el padre), para terminar, y el padre a que se muera su hijo (es decir, el nieto), por lo que la salida de este programa siempre tendrá el orden: nieto, padre, abuelo. Se pueden hacer árboles de procesos mucho más complejos, pero una vez visto cómo hacer múltiples hijos y cómo hacer múltiples generaciones, el resto es bastante trivial.

**Otra manera de crear nuevos procesos, bueno, más bien de modificar los existentes, es mediante el uso de las funciones exec(). Con estas funciones lo que conseguimos es reemplazar la imagen del proceso actual por la de un comando o programa que invoquemos, de manera similar a como lo hacíamos al llamar a system(). En función de cómo queramos realizar esa llamada, elegiremos una de las siguientes funciones:**

int execl( const char \*path, const char \*arg, ...);

int execlp( const char \*file, const char \*arg, ...);

int execle( const char \* path, const char \*arg , ..., char \* const envp[]);

int execv( const char \* path, char \*const argv[]);

int execvp( const char \*file, char \*const argv[]);

int execve(const char \*filename, char \*const argv [], char \*const envp[]);

El primer argumento es el fichero ejecutable que queremos llamar. Las funciones que contienen puntos suspensivos en su declaración indican que los parámetros del ejecutable se incluirán ahí, en argumentos separados. Las funciones terminadas en “e” ( execle() y execve() ) reciben un último argumento que es un puntero a las variables de entorno. Un ejemplo sencillo nos sacará de dudas:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>    int main(int argc, char \*argv[])  {      char \*args[] = { "/bin/ls", NULL };        execv("/bin/ls", args);        printf("Se ha producido un error al ejecutar execv.\n");        return 0;  } |

La función elegida, execv(), recibe dos argumentos, el path al fichero ejecutable (“/bin/ls”) y un array con los parámetros que queremos pasar. Este array tiene la misma estructura que argv, es decir, su primer elemento es el propio programa que queremos llamar, luego se va rellenando con los argumentos para el programa y por último se finaliza con un puntero nulo (NULL). El printf() final no debería salir nunca, ya que para ese entonces execv() se habrá encargado de reemplazar la imagen del proceso actual con la de la llamada a “/bin/ls”. La salida de este programa es la siguiente:

txipi@neon:~$ gcc execv.c -o execv

txipi@neon:~$./execv

doshijos execv fil2 files.c hijopadrenieto.c

doshijos.c execv.c fil2.c hijopadrenieto