Práctica 2.
Creación y duplicación de procesos.

Primer programa en C

Si hay un programa obligatorio a la hora de empezar a programar en un lenguaje de programación, ese es el mítico “Hola, mundo!”.

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
printf( "Hola, mundo!\n");

return 0;
}
```

```
txipi@neon:~$ gcc hola.c –o hola
txipi@neon:~$ ./hola
Hola, mundo!
```

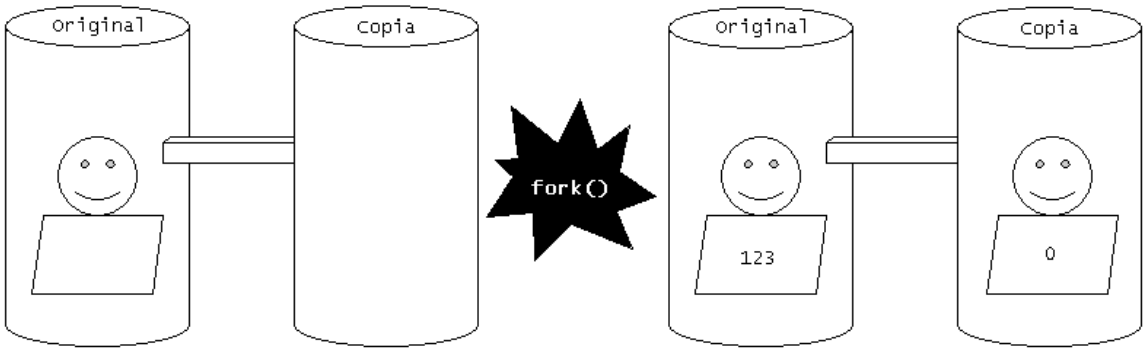
Creación y duplicación de procesos

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().
Nos centramos en primer lugar en 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 la de la “persona copia” hay un cero:



En 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.

```
#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:

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

int main(int argc, char *argv[])
{
pid_t pid1, pid2;
int status1, status2;
pid1=fork();
if ( pid1== 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 termine, status 0*/
    waitpid(pid1, &status1, 0);
            /* Esperamos al segundo hijo termine, status 0 */
    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.

Entregable1

Abuelo ---- fork () ---> Hijo ----- fork() ----> nieto

Realiza un programa llamado tresgeneraciones.c en el que el árbol cubra tres generaciones y la salida debe ser contranatura es decir siempre tendrá el orden: nieto, padre, abuelo, la salida al ejecutarlo será debe ser la siguiente:

```
txipi@neon:~$ ./tresgeneraciones
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
```

Entregable2

Realiza un programa en C que cree un proceso (tendremos 2 procesos uno padre y otro hijo). El programa definirá una variable entera y le dará el valor 6. El proceso padre incrementará dicho valor en 5 y el hijo restará 5. Se deben mostrar los valores en pantalla. A continuación se muestra ejemplo de la ejecución.

```
david@david-OEM ~/pss $ ./actividad2
Soy el Padre.Valor variable 1
Soy el hijo.Valor variable 11
```

Explica el resultado del programa (recuerda los cambios de contextos explicados en la teoría).