UNIVERSIDAD SIGLO 21

MARCO TEÓRICO DE PROCESOS EN C/C++

Algoritmos concurrentes y paralelos

Dr. Miguel Méndez-Garabetti, Ing. Eduardo Piray

25/10/2022

Marco teórico de soporte para la realización de los trabajos prácticos.

Contenido

1. Introducción 3
2. Identificador de proceso en el lenguaje de programación C 3
3. Llamada al sistema fork() 3
3.1. Consideraciones y recomendaciones importantes acerca de fork() 5
3. Árboles de procesos 6
4. Comparación: fork() vs exec() 8
5. Terminación de procesos 8
5.1. Otros ejemplos del uso de wait(): 10
5.2. Uso de exit() en C/C++ para terminar procesos 12
6. Generación de múltiples procesos 13
6.1. Ejemplo de generación de múltiples procesos con fork() en C++ 19
7. Anexo 23
7.1. Compiladores online 23
7.2. Bibliografía 23
7.3. Sitios web de referencia 23

1. Introducción

Recordemos que todo programa en ejecución es un **proceso**. Los procesos son manejados por el sistema operativo.
Tal como se hizo con hilos, vamos a seguir la idea de **concurrencia**, es decir, varios procesos que intentan ejecutarse al mismo tiempo de forma concurrente.
La mayoría del código fuente que se expone en el presente documento referido a procesos con fork, está mayormente codificado en el lenguaje de C, pero su transición a C++ es sencilla, ya existe una gran equivalencia entre ambos lenguajes de programación.

2. Identificador de proceso en el lenguaje de programación C

Todos los procesos son manejados por el sistema operativo, el cual los diferencia entre sí según un **identificador único de proceso**, o **process id (pid)**. El pid se puede consultar y obtener mediante la función **getpid()** de la cabecera **unistd**. Por ejemplo:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    printf("Soy un proceso, mi identificador de proceso es: %d\n", getpid());
}
```

La función **getppid()** es una función que regresa el identificador (pid) del proceso padre.

3. Llamada al sistema fork()

La llamada al sistema **fork()** se usa para crear un nuevo proceso, que se llama **proceso hijo** (child), que se ejecuta simultáneamente con el proceso que realiza la llamada a fork (proceso principal o padre). Este proceso hijo será un "clon" del proceso principal, pero ambos serán independientes entre sí. Después de que se crea un nuevo proceso hijo, ambos procesos ejecutarán la siguiente instrucción después de la llamada al sistema fork(). Un proceso hijo usa el mismo contador de programa (program counter), los mismos registros de CPU, los mismos archivos abiertos que usa el proceso padre.

Los **archivos de cabecera** en donde se encuentra la definición de fork son **sys/types.h** y **unistd.h**. En **unistd.h** está definida la función y en **types.h** el tipo de dato **pid_t** el cuál es un identificador de proceso Linux PID. La ejecución de fork crea un proceso hijo que se diferencia de su creador únicamente por su PID y por su PPID (Parent PID, identificador del proceso padre del proceso actual).

La llamada fork() no recibe parámetros y devuelve un **valor entero**. A continuación se muestran los **valores de retorno de fork()**:

- Valor negativo: la creación de un proceso hijo no tuvo éxito.
- Cero: regresa este valor al proceso hijo recién creado.
- Valor positivo: valor regresado que contiene el identificador de proceso (Process ID) del nuevo proceso hijo recién creado. Es decir, que fork() en este caso regresa al proceso padre el process id del hijo.

Veamos un ejemplo en C:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork(); //aquí se crea un proceso hijo del proceso principal

    printf("Hola!\n");
    return 0;
}
```

La salida en pantalla será con dos mensajes, uno del proceso principal y otro del proceso hijo:

```
Hola!

Hola!
```

En este caso ambos procesos "hacen lo mismo", pero muchas veces, es más útil que el nuevo proceso realice una funcionalidad diferente de su proceso padre. Para lograr esto en el mismo código de un programa en C, es habitual el uso de una sentencia de decisión if-else, por medio del identificador de proceso, se pueda determinar en el código quien es el proceso padre, y quien es el proceso hijo.

Veamos dos ejemplos que representan el manejo de procesos en C.

Primera forma:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int pid = fork();

    // instrucciones que tanto el padre como el hijo realizarán

    if (pid == 0)
    {

         // instrucciones que solo el proceso hijo realizará
    }
```

```
    else
    {
        // instrucciones que solo el proceso padre realizará
    }
}
```

Segunda forma:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int pid = fork();

    // instrucciones que tanto el padre como el hijo realizarán

    if (pid != 0)
    {
        // instrucciones que solo el proceso padre realizará
    }
    else
    {
        // instrucciones que solo el proceso hijo realizará
    }
}
```

3.1. Consideraciones y recomendaciones importantes acerca de fork()

La llamada al sistema fork() es propia de los sistemas operativos derivados de UNIX y de Linux.

En **sistemas operativos Windows no es posible ejecutar la llamada al sistema fork en forma nativa**. Es decir, si se intenta compilar un programa en C/C++ con fork en Windows, se producirá un error porque el sistema operativo no reconocerá dicha instrucción.

Para utilizar fork en Windows se puede recurrir a alternativas como:

- Uso de **compiladores online**.
- La creación de una **máquina virtual**, que prepare un entorno de Linux con un compilador de C/C++.
- La creación de un **contenedor**, de Docker por ejemplo, que prepare un entorno Linux con un compilador de C/C++.
- Probar con alternativas como **Cygwin**. Cygwin es un entorno de ejecución y programación compatible con POSIX que se ejecuta de forma nativa en Microsoft Windows. Bajo Cygwin, el código fuente diseñado para sistemas operativos similares a Unix puede compilarse con modificaciones mínimas y ejecutarse[1] [2].

[1] Sitio oficial del Proyecto Cyqwin: https://www.cygwin.com/
[2] Foto de referencia: https://stackoverflow.com/questions/985281/what-is-the-closest-thing-windows-has-to-fork

- El **Subsistema de Windows para Linux, o Windows Subsystem for Linux (WSL)**, permite la ejecución de un entorno de trabajo GNU/Linux, incluyendo todas sus herramientas, utilidades y aplicaciones, directamente en Windows, sin la necesidad de usar una máquina virtual tradicional o hacer un dualboot (arranque dual).
  Nota: ver el primer marco teórico (Hilos en C-C++), allí se explica la manera de configurar WSL.

Por lo mencionado, en los trabajos prácticos de la materia que involucren la codificación de programas con fork, y si se dispone de una PC con Windows, se recomienda en primera instancia utilizar compiladores online, o recurrir a una máquina virtual con un entorno de Linux.

3. Árboles de procesos

Sucesivas llamadas a fork() van generar un **árbol de procesos**. El número total de procesos generados será el resultado de la siguiente potencia:

$$Cantidad\ de\ procesos = 2^n$$

Donde n es el número de llamadas al sistema fork().

Por ejemplo:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hola\n");
    return 0;
}
```

El código anterior tendrá como salida:

```
Hola

Hola

Hola

Hola

Hola

Hola

Hola

Hola
```

Se realizan n = 3 llamadas al sistema fork(), generando 8 procesos (nuevos procesos hijos y un proceso original), porque $2^3 = 8$.

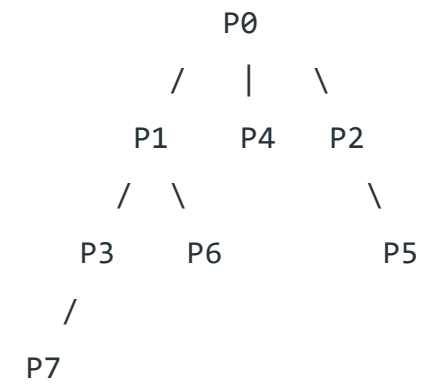
Si queremos representar la relación entre los procesos como una jerarquía de árbol, sería de la siguiente manera:

El proceso principal (padre): P0.

Procesos creados por el primero de los fork(): P1.

Procesos creados por el segundo de los fork(): P2, P3.

Procesos creados por el tercero de los fork(): P4, P5, P6, P7.



Otro ejemplo usando fork() en C:

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

void forkejemplo()
{
    int x = 1;

    if (fork() == 0)
        printf("Hijo tiene x = %d\n", ++x);
    else
        printf("Padre tiene x = %d\n", --x);
}
int main()
{
    forkejemplo();
    return 0;
}
```

La salida del código anterior es:

```
Padre tiene x = 0
Hijo tiene x = 2
```

(ó)

```
Hijo tiene x = 2
Padre tiene x = 0
```

En base al código anterior y su salida en pantalla, podemos decir lo siguiente:

- Se crea un proceso hijo. La llamada al sistema fork() devuelve 0 en el proceso hijo y un entero positivo en el proceso padre o principal.
- Dos salidas son posibles porque el proceso padre y el proceso hijo se ejecutan simultáneamente (concurrentemente). No sabemos si el sistema operativo dará primero el control al proceso padre o al proceso hijo. El cambio de una variable global en un proceso no afecta a los otros procesos porque los datos/estado de ambos procesos son diferentes. El padre y el hijo se ejecutan simultáneamente (concurrentemente), por lo que son posibles dos salidas.
- El sistema operativo asigna diferentes datos y estados para estos dos procesos, y el flujo de control de estos procesos puede ser diferente.

4. Comparación: fork() vs exec()

La llamada al sistema fork() crea un nuevo proceso. El nuevo proceso creado por fork() es una copia del proceso actual excepto por el valor devuelto. La llamada al sistema **exec()** reemplaza el proceso actual con un nuevo programa.

5. Terminación de procesos

Ya vimos anteriormente como se va generando un árbol de procesos cuando hacemos llamadas sucesivas al sistema fork(). Antes de continuar analizando la creación de múltiples procesos, debemos conocer la llamada al sistema wait().

Una llamada a **wait()** bloquea el proceso de llamada hasta que uno de sus procesos hijo finaliza o se recibe una señal del sistema operativo o de otro proceso. Después de que finaliza el proceso hijo, el padre continúa su ejecución después de la instrucción de llamada al sistema **wait()**.

El proceso hijo puede terminar debido a:

- Llama a **exit()**.
- Devuelve un int de main.

- Recibe una señal (del sistema operativo o de otro proceso) cuya acción predeterminada es terminar.

Otras consideraciones acerca de **wait()**:

- Si algún proceso tiene más de un proceso hijo, luego de llamar a wait(), el proceso principal debe estar en estado de espera si no finaliza ningún proceso hijo.
- Si solo se finaliza un proceso hijo, wait() retorna el ID del proceso hijo finalizado.
- Si se finaliza más de un proceso hijo, wait() selecciona arbitrariamente uno de esos procesos hijos terminados, y devuelve un ID de proceso de ese proceso hijo.
- wait() también define el estado de salida (que le dice a nuestro proceso por qué terminó) a través del puntero, si el estado no es NULL.
- Si algún proceso no tiene un proceso hijo, wait() devuelve "-1".

Veamos un ejemplo con uso de wait():

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);           // hijo terminado
    else
        cpid = wait(NULL);
    printf("Padre pid = %d\n", getpid());
    printf("Hijo pid = %d\n", cpid);

    return 0;
}
```

La salida de este programa es:

```
Padre pid = 7998

Hijo pid = 7999
```

En este programa el proceso padre no termina hasta tanto el hijo no haya terminado.

Veamos otro ejemplo de uso de wait():

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

```
int main()
{
    if (fork()== 0)
        printf("HH: Hola desde el hijo\n");
    else
    {
        printf("HP: hola desde el padre\n");
        wait(NULL); //espera a que termine el hijo
        printf("HT: hijo terminado\n");
    }
    printf("Fin\n");
    return 0;
}
```

La salida de este programa (como en muchos otros) va a depender del ambiente o entorno donde se ejecute. Una posible salida es:

```
HP: hola desde el padre

HH: Hola desde el hijo

Fin

HT: hijo terminado

Fin
```

La sentencia HT: hijo terminado no se imprime antes de HH porque está presente la llamada al sistema wait().

En otro ambiente o entorno, la salida puede ser:

```
HH: Hola desde el hijo

Fin

HP: hola desde el padre

HT: hijo terminado
```

5.1. Otros ejemplos del uso de wait():

Ejemplo: Vamos a realizar un programa que deberá imprimir por pantalla los números del 1 al 10 en orden. Lo realizaremos con dos procesos: el proceso hijo debe imprimir los números del 1 al 5, y el proceso padre debe imprimir los números restantes del 6 al 10.

En principio codificamos lo siguiente:

```
#include<stdio.h>
#include<string.h>
```

```
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
#include<time.h>

int main(int argc, char* argv[])
{
	int id = fork();
	int n;
	if(id == 0){ //proceso hijo
		n = 1;
	}
	else { //proceso padre
		n = 6;
	}

	//ciclo for que imprime todos los números
	int i;
	for (int i = n; i < n + 5; i++){
		printf("%d ", i);
		fflush(stdout);
	}

	return 0;
}
```

Nota: **fflush()** se usa generalmente solo para el flujo de salida. Su propósito es borrar (o vaciar) el búfer de salida y mover los datos almacenados en el búfer a la consola (en el caso de la salida estándar) o al disco (en el caso del flujo de salida del archivo).
La salida estándar de este programa puede ser:

```
6 7 8 9 10 1 2 3 4 5
```

Dependiendo del entorno de ejecución, pueden generarse otras salidas tales como:

```
6 1 7 8 2 9 3 10 4 5

6 7 8 9 10
```

En todos los casos la salida no está en orden. En una de las salidas se imprimen únicamente los números del 6 al 10, que corresponden al proceso padre, esto quiere decir que el padre se ejecuta y termina el proceso principal sin que se ejecute el hijo. En otra salida imprime por el lado del padre, luego del hijo, y va alternando (6 1 7 8 2 9 3 10 4 5), digamos que es la salida menos deseada. Y en la otra salida se observa que se ejecuta el proceso padre y luego el hijo (6 7 8 9 10 1 2 3 4 5). Estas ejecuciones varían, porque el sistema operativo es quien está decidiendo el orden de ejecución de los procesos.

Sin embargo, ¿cómo se puede hacer para que se impriman en orden? Por medio del uso de la función wait(). Lo que va a hacer wait() es detener la ejecución del proceso padre hasta tanto el

proceso hijo no termine de imprimir sus números, y así posteriormente continuar la ejecución del proceso padre hasta terminar.

Al código anterior solamente hay que agregar la línea wait(), dentro de un if que pregunte si no es el proceso hijo se debe ejecutar wait():

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
#include<time.h>

int main(int argc, char* argv[])
{
	int id = fork();
	int n;
	if(id == 0){ //proceso hijo
		n = 1;
	}
	else { //proceso padre
		n = 6;
	}
	if(id != 0){
		wait();//espera a que termine el proceso hijo
	}
	//ciclo for que imprime todos los números
	int i;
	for (int i = n; i < n + 5; i++){
		printf("%d ", i);
		fflush(stdout);
	}

	return 0;
}
```

Nota: en algunos compiladores **wait()** requiere como argumento **NULL**, quedando la línea que añadimos como **wait(NULL)**. También es necesario el añadido de la librería **#include<sys/wait.h>**.
Con estos cambios, la salida en pantalla será en el orden pretendido:

```
1 2 3 4 5 6 7 8 9 10
```

5.2. Uso de exit() en C/C++ para terminar procesos

exit() termina el proceso normalmente, sin ejecutar el código restante que se encuentra después de la función **exit()**.

Sintaxis: **void exit (int status);**

status: valor de estado devuelto al proceso padre. Un valor de status de **0** o **EXIT_SUCCESS** indica éxito, y cualquier otro valor o la constante **EXIT_FAILURE** se utilizan para indicar un error.

exit() realiza las siguientes operaciones:

- Vacía los datos almacenados en el búfer no escritos.
- Cierra todos los archivos abiertos.
- Elimina archivos temporales.
- Devuelve un estado de salida entero al sistema operativo.

6. Generación de múltiples procesos

Finalizado el ejemplo anterior, pasaremos a profundizar un poco más en cómo generar correctamente múltiples procesos.

Seguramente para generar 10 procesos te imaginas algo como:

```
for(int i = 0; i < 10; i++){
	fork();
}
```

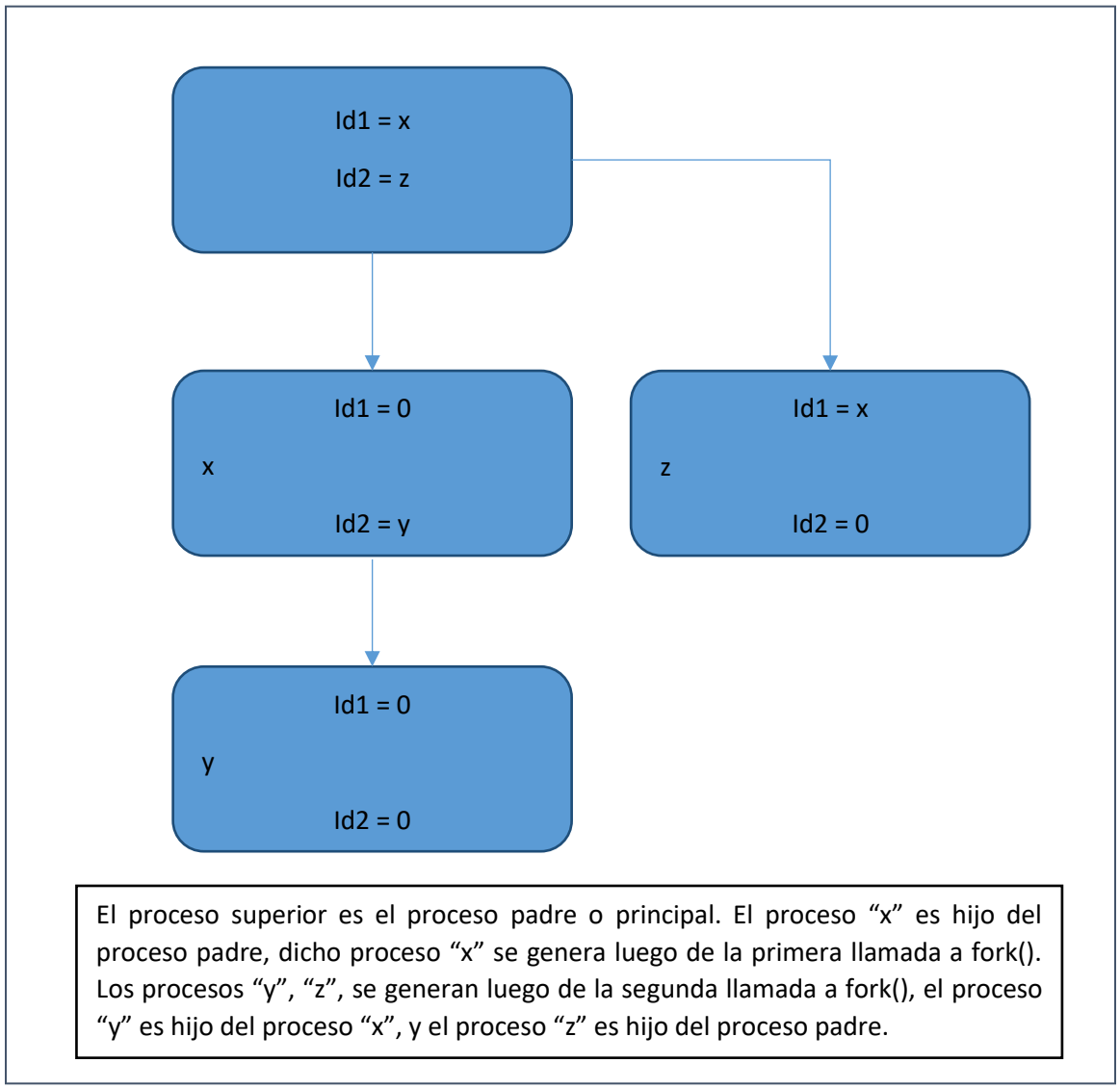
Al analizar este código piensas que se generan diez (10) procesos. La realidad es que **no** se generan diez (10) procesos, sino muchos más procesos que la cantidad 10 (recordar cuando se mencionó las potencias de dos (2) que generan los árboles de procesos). Veamos un ejemplo para aclarar esto.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
	int id1 = fork();
	int id2 = fork();

	return 0;
}
```

En el código anterior se realizan dos llamadas fork(), es decir que se generan 4 procesos, que se detallan en el siguiente diagrama:



El proceso superior es el proceso padre o principal. El proceso “x” es hijo del proceso padre, dicho proceso “x” se genera luego de la primera llamada a fork(). Los procesos “y”, “z”, se generan luego de la segunda llamada a fork(), el proceso “y” es hijo del proceso “x”, y el proceso “z” es hijo del proceso padre.

Codifiquemos la jerarquía de procesos descripta en el diagrama:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
      int id1 = fork();
      int id2 = fork();

      if(id1 == 0){
```

```
            if(id2 == 0){//nieto del proceso padre o hijo del proceso hijo
del padre
                  printf("Soy el proceso y\n");
            } else {
                  printf("Soy el proceso x\n"); //hijo del proceso padre
            }
      } else {
            if(id2 == 0) {
                  printf("Soy el proceso z\n");
            } else {
                  printf("Soy el proceso padre\n");
            }
      }
      return 0;
}
```

Nota importante sobre la ejecución de programas: *dependiendo del entorno o ambiente de ejecución en donde se ejecuten nuestros algoritmos con procesos pueden producirse distintas salidas. Existen muchos entornos, por ejemplo, una computadora con Linux (que incluye compiladores de C y C++), una computadora con Windows con una máquina virtual con un entorno Linux y un compilador de C/C++, compiladores online, entre otros entornos.*

Volviendo al ejemplo que estamos tratando, si ejecutamos en un entorno obtenemos salidas tales como (se presentan seis ejecuciones):

Soy el proceso padre Soy el proceso z Soy el proceso y Soy el proceso x	Soy el proceso padre Soy el proceso x Soy el proceso y Soy el proceso z	Soy el proceso padre Soy el proceso x Soy el proceso y Soy el proceso z	Soy el proceso padre Soy el proceso x Soy el proceso z Soy el proceso y
Soy el proceso padre Soy el proceso z Soy el proceso x Soy el proceso y	Soy el proceso padre Soy el proceso x Soy el proceso z Soy el proceso y		

En otro entorno obtenemos (se presentan dos ejecuciones):

Soy el proceso padre	Soy el proceso padre Soy el proceso x

Este último entorno está mostrando de mejor manera lo que ocurre. El proceso padre se ejecuta, y no espera a que se ejecuten y terminen sus procesos hijos, dicho proceso padre termina, y por ende termina el programa, por supuesto que también finalizan los procesos hijos.

Debemos ver cómo solucionar este inconveniente. Si añadimos al código del algoritmo una instrucción **wait(NULL)** antes de la instrucción **return 0**, la ejecución toma varios estados diferentes:

Soy el proceso padre Soy el proceso z	Soy el proceso padre Soy el proceso x Soy el proceso z
Soy el proceso padre Soy el proceso z Soy el proceso x Soy el proceso y	Soy el proceso padre Soy el proceso x Soy el proceso y Soy el proceso z
Soy el proceso padre Soy el proceso x Soy el proceso z Soy el proceso y	Soy el proceso padre Soy el proceso x Soy el proceso y

Estas salidas no son del todo correctas, el programa sigue terminando antes de que todos los procesos hijos existentes en el árbol de procesos terminen (en la mayoría de las ejecuciones). Para solucionar estas variaciones en la salida, y esperar a que todos los hijos del árbol terminen (recordar que hay tres procesos que tienen un proceso que los antecede en la jerarquía inmediata superior). Hay que un usa un **while** y hacer un manejo de error utilizando la variable global **errno**. En el while se imprimirá tres veces un mensaje, porque se espera a que todos los hijos (son tres en total) terminen antes de terminar el proceso padre raíz. La condición del while es: **`wait (NULL) != -1 || errno != ECHILD`**, que indica que se ejecutará el printf mientras wait() no devuelva -1 ó errno sea distinto de ECHILD. **ECHILD** se refiere a "No child processes", es decir no hay o no quedan procesos hijos en ejecución.
El código con manejo de error es:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <errno.h>

int main(int argc, char *argv[]){
      int id1 = fork();
      int id2 = fork();

      if(id1 == 0){
            if(id2 == 0){//nieto del proceso padre o hijo del proceso hijo
del padre
                  printf("Soy el proceso y\n");
            } else {
                  printf("Soy el proceso x\n"); //hijo del proceso padre
```

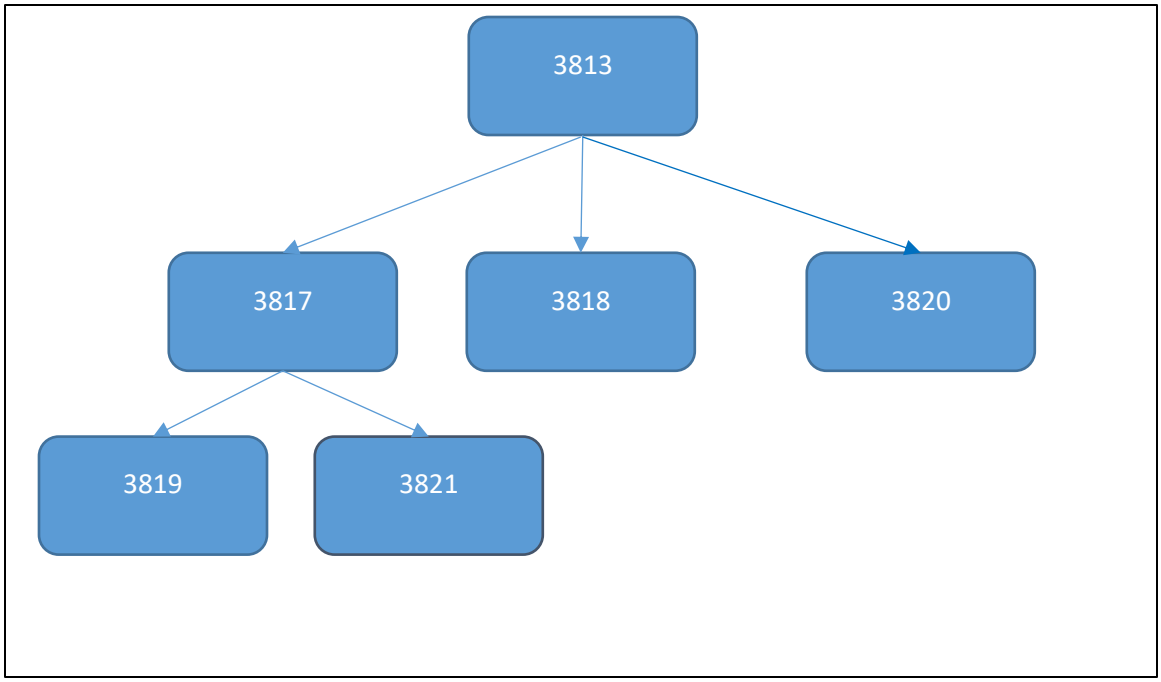
```
            }
      } else {
            if(id2 == 0) {
                  printf("Soy el proceso z\n");
            } else {
                  printf("Soy el proceso padre\n");
            }

      }
      while(wait(NULL) != -1 || errno != ECHILD){
          printf ("Esperando a que un hijo termine\n");
      }
      return 0;
}
```

Se presentan seis posibles salidas, en las cuáles todos los procesos terminan correctamente.

Soy el proceso padre Soy el proceso x Soy el proceso y Soy el proceso z Esperando a que un hijo termine Esperando a que un hijo termine Esperando a que un hijo termine	Soy el proceso padre Soy el proceso z Soy el proceso x Soy el proceso y Esperando a que un hijo termine Esperando a que un hijo termine Esperando a que un hijo termine	Soy el proceso padre Soy el proceso z Esperando a que un hijo termine Soy el proceso x Soy el proceso y Esperando a que un hijo termine Esperando a que un hijo termine
Soy el proceso padre Soy el proceso x Soy el proceso y Esperando a que un hijo termine Esperando a que un hijo termine Soy el proceso z Esperando a que un hijo termine	Soy el proceso padre Soy el proceso z Esperando a que un hijo termine Soy el proceso x Soy el proceso y Esperando a que un hijo termine Esperando a que un hijo termine	Soy el proceso padre Soy el proceso z Esperando a que un hijo termine Soy el proceso x Soy el proceso y Esperando a que un hijo termine Esperando a que un hijo termine

Otro Ejemplo: vamos a realizar un árbol de procesos que tenga el proceso principal (proceso padre), este tenga tres procesos hijos, y uno de sus hijos tiene a su vez otros dos hijos. En el siguiente diagrama se observa el árbol de procesos a realizar:




```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
 pid_t pid;
 int x,y;

 for(x = 1;x <= 3; x++)
 {
      pid=fork();
      if(pid)
      {
            printf("Soy el proceso %d\n",getpid());
            sleep(2);
      }
      else
      {
            printf("Soy el hijo %d, mi padre es %d\n",getpid(),getppid());
            sleep(2);
             if(x==1)
             {
                  for(y = 1;y <= 2;y++)
                  {
                        pid=fork();
                         if(pid)
```

```
                        {
                                printf("Soy el proceso %d\n",getpid());
                                sleep(2);
                        }
                        else
                                {
                                printf("Soy el hijo %d, mi padre es
                                %d\n",getpid(),getppid());
                                sleep(2);
                                exit(0);
                                }
                        }
                }
                exit(0);
        }
 }
 return 0;
}
```

Al ejecutar como salida en pantalla se obtiene:

```
Soy el hijo 3817, mi padre es 3813
Soy el proceso 3813
Soy el hijo 3818, mi padre es 3813
Soy el proceso 3817
Soy el hijo 3819, mi padre es 3817
Soy el proceso 3813
Soy el proceso 3817
Soy el hijo 3820, mi padre es 3813
Soy el hijo 3821, mi padre es 3817
```

Notar en el código el uso de exit(), esto suele tener la funcionalidad de controlar la generación de procesos hijos.

6.1. Ejemplo de generación de múltiples procesos con fork() en C++

En C++ la llamada al sistema fork() es equivalente a como se utiliza en C, se siguen lineamientos muy similares. Veamos un ejemplo en donde se generan n cantidad procesos, para este caso puntual analizamos la creación de ocho (8) procesos hijos que tienen un mismo padre[3]. Este código en C++ se puede trasladar al lenguaje C.

El código fuente es el siguiente:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
```

[3] Fuente: https://gist.github.com/dgacitua/64ff00e90d5e21f9c3f7 por Daniel Gacitúa <danielgacituav@gmail.com>

```
#include <cerrno>
#include <cstdio>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>

using namespace std;

int main (int argc, char** argv) {
      static const int PROC = 8;    // PROC fija cantidad de procesos a crear

      int status, procNum;          // procNum almacena el número del proceso
      pid_t pid;                    // pid almacena el id del proceso

      for (procNum=0; procNum<PROC; procNum++) {
            pid = fork();           // llamada al sistema fork()

            if (pid==0) {       // si el proceso se crea bien, termina el for
                  break;
            }
            else if (pid==-1) {     // si hay error termina la operación
                  perror("ERROR al hacer llamada fork()");
                  exit(1);
                  break;
            }
      }

      if (pid==0) {                 // Lógica del proceso Hijo
            cout << "Soy el proceso " << procNum << " mi id es " << getpid()
<< " mi padre es " << getppid() << endl;
            exit(0);
      }
      else {                              // Lógica del proceso Padre
            for (int i=0; i<PROC; i++) {  // esperar a que todos los hijos
terminen
            if ((pid = wait(NULL)) >= 0) {
                  cout << "Proceso " << pid << " terminado" << endl;
            }
      }

            cout << "Soy el padre " << getpid() << endl;
      }
      return 0;         // Fin
}
```

Analicemos algunos detalles del código y del algoritmo:

- El tipo de datos **pid_t** es un tipo de entero con signo que puede representar un ID de proceso. Es un int.
- Con la constante PROC se define la cantidad de proceso a crear.
- Notar que hay un proceso padre y se crean 8 procesos hijos.

Analicemos la salida del código anterior con varias ejecuciones (cuatro):

```
Soy el proceso 0 mi id es 3514 mi padre es 3510
Soy el proceso 1 mi id es 3515 mi padre es 3510
Soy el proceso 4 mi id es 3518 mi padre es 3510
Proceso 3514 terminado
Proceso 3515 terminado
Soy el proceso 2 mi id es 3516 mi padre es 3510
Soy el proceso 3 mi id es 3517 mi padre es 3510
Proceso 3518 terminado
Soy el proceso 6 mi id es 3520 mi padre es 3510
Proceso 3516 terminado
Proceso 3517 terminado
Proceso 3520 terminado
Soy el proceso 7 mi id es 3521 mi padre es 3510
Soy el proceso 5 mi id es 3519 mi padre es 3510
Proceso 3521 terminado
Proceso 3519 terminado
Soy el padre 3510
```

```
Soy el proceso 0 mi id es 244 mi padre es 240
Soy el proceso 1 mi id es 245 mi padre es 240
Soy el proceso 3 mi id es 247 mi padre es 240
Soy el proceso 4 mi id es 248 mi padre es 240
Proceso 244 terminado
Proceso 245 terminado
Soy el proceso 2 mi id es 246 mi padre es 240
Proceso 247 terminado
Soy el proceso 7 mi id es 251 mi padre es 240
Proceso 248 terminado
Proceso 246 terminado
Proceso 251 terminado
Soy el proceso 5 mi id es 249 mi padre es 240
Proceso 249 terminado
Soy el proceso 6 mi id es 250 mi padre es 240
Proceso 250 terminado
Soy el padre 240
```

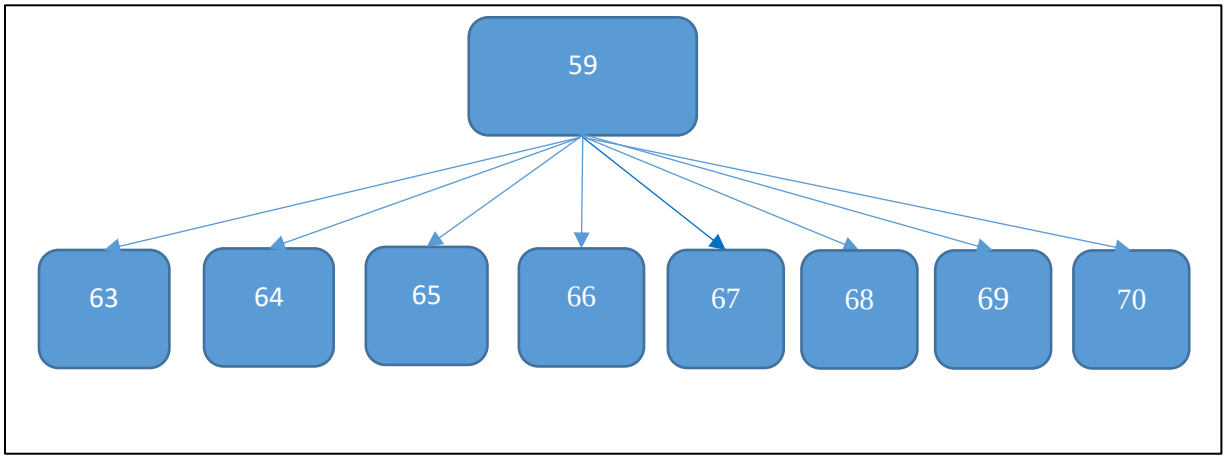
```
Soy el proceso 2 mi id es 48 mi padre es 42
Proceso 48 terminado
Soy el proceso 7 mi id es 53 mi padre es 42
Soy el proceso 5 mi id es 51 mi padre es 42
Soy el proceso 6 mi id es 52 mi padre es 42
Proceso 51 terminado
```

```
Proceso 53 terminado
Proceso 52 terminado
Soy el proceso 4 mi id es 50 mi padre es 42
Proceso 50 terminado
Soy el proceso 3 mi id es 49 mi padre es 42
Proceso 49 terminado
Soy el proceso 1 mi id es 47 mi padre es 42
Proceso 47 terminado
Soy el proceso 0 mi id es 46 mi padre es 42
Proceso 46 terminado
Soy el padre 42
```

```
Soy el proceso 2 mi id es 65 mi padre es 59
Soy el proceso 3 mi id es 66 mi padre es 59
Soy el proceso 1 mi id es 64 mi padre es 59
Soy el proceso 4 mi id es 67 mi padre es 59
Proceso 65 terminado
Proceso 64 terminado
Proceso 66 terminado
Proceso 67 terminado
Soy el proceso 5 mi id es 68 mi padre es 59
Soy el proceso 0 mi id es 63 mi padre es 59
Soy el proceso 6 mi id es 69 mi padre es 59
Proceso 68 terminado
Proceso 69 terminado
Proceso 63 terminado
Soy el proceso 7 mi id es 70 mi padre es 59
Proceso 70 terminado
Soy el padre 59
```

Notar que varían las salidas: cambian los id de procesos, y cambia el orden de ejecución, esto último indica la presencia de concurrencia en la ejecución de los procesos.

Un esquema de la última ejecución define el proceso padre con id 59, y sus 8 hijos con sus id del 63 al 70:



7. Anexo

7.1. Compiladores online

Es posible ejecutar programas en C/C++ por medio de compiladores online. Algunos compiladores online son:

https://wandbox.org/

https://www.onlinegdb.com/online_c_compiler

https://www.programiz.com/c-programming/online-compiler/

https://www.programiz.com/cpp-programming/online-compiler/

7.2. Bibliografía

Bibliografía para tener de referencia para la sintaxis básica de los lenguajes C y C++ (variables, tipos de datos, asignaciones, operadores, estructura de decisión y control, punteros, funciones, objetos en el caso de C++):

- Cómo programar en C++ (9a. ed.): Harvey Deitel; Deitel, Paul. Pearson Educación, 2014.
- Cómo programar en C/C++ (4a. ed.) Libro electrónico. By: Paul J. Deitel; M. Deitel, Harvey. Pearson Educación, 2004.
- C++11 for Programmers. By: Paul J. Deitel; Harvey M. Deitel. Series: Deitel Developer Series. Prentice Hall. 2013.
- C for Programmers with an Introduction to C11. By: Paul J. Deitel; Harvey Deitel. Series: Deitel Developer Series. Prentice Hall. 2013.
- C How to Program, Global Edition. By: Paul Deitel; Harvey Deitel. Series: How to Program Series. Edition: Eighth edition, global edition. Boston: Pearson. 2016.
- C++ How to Program (Early Objects Version), International Edition: (Early Objects Version). By: Harvey Deitel; Paul Deitel. Harlow, United Kingdom: Pearson. 2013.

7.3. Sitios web de referencia

Referencia para la llamada al sistema fork:

- https://man7.org/linux/man-pages/man2/fork.2.html

Documentación de C:

- Estándar de C: recuperado de http://www.open-std.org/jtc1/sc22/wg14/
- https://www.geeksforgeeks.org/c-language-set-1-introduction/?ref=gcse

- POSIX, pthread: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html#:~:text=The%20POSIX%20thread%20libraries%20are,through%20parallel%20or%20distributed%20proc
- Referencia para concurrencia en C: https://en.cppreference.com/w/c/thread

Documentación de C++:

- Referencia para threads: https://www.cplusplus.com/reference/thread/thread/
- Estándar de referencia para concurrencia: https://isocpp.org/wiki/faq/cpp11-library-concurrency
- Referencia para concurrencia en C++: https://en.cppreference.com/w/cpp/thread
- Referencia del lenguaje C++: https://www.w3schools.com/cpp/default.asp

Sitio oficial de GCC, the GNU Compiler Collection:

- https://gcc.gnu.org/

IDE Code::Blocks:

- Instalar y ejecutar programas de C y C++ en Windows: http://www.codeblocks.org/
- Documentación de Code::Blocks: https://wiki.codeblocks.org/index.php/Main_Page

Ejemplos en general:

- https://code-vault.net/