



**INSTITUTO POLITÉCNICO NACIONAL**  
**ESCUELA SUPERIOR DE CÓMPUTO**



**SISTEMAS OPERATIVOS**

**PRÁCTICA 3:**

Creación de procesos

**ALUMNOS:**

Martínez Ramírez Sergi Alberto – 2020630260 - 2CV15

Meza Vargas Brandon David – 2020630288 - 2CM15

Peña Atanasio Alberto – 2020630367 - 2CV15

Sarmiento Gutiérrez Juan Carlos – 2020630386 - 2CV15

**PROFESOR:**

José Alfredo Jiménez Benítez

# Índice de contenido

<b>Glosario de términos .....</b>	<b>4</b>
<b>Contenido .....</b>	<b>5</b>
Creación de procesos con la llamada al sistema fork .....	5
<b>Desarrollo de la práctica.....</b>	<b>6</b>
Programa 31 .....	6
Programa 32 .....	8
Programa 33 .....	14
<b>Conclusiones .....</b>	<b>16</b>
<b>Bibliografía .....</b>	<b>17</b>
<b>Anexos .....</b>	<b>18</b>
Código programa 31.....	18
Código programa 32.....	19

## Índice de figuras

Imagen 1. Creación básica de procesos con fork().....	6
Imagen 2. Algoritmo de la creación de los procesos hijos del programa 31.c .....	6
Imagen 3. Uso del wait() dentro del programa 31.c .....	7
Imagen 4. Compilación del programa 31.c .....	7
Imagen 5. Ejecución del programa 31.c .....	8
Imagen 6. Finalización de los procesos hijos y padre .....	8
Imagen 7. Procesos del programa 32. ....	9
Imagen 8. Función codigoProceso (int id). ....	9
Imagen 9. Creación del primer proceso con fork(). ....	9
Imagen 10. Proceso bisabuelo.....	10
Imagen 11. Creación de procesos nietos. ....	10
Imagen 12. Creación de procesos bisnietos.....	10
Imagen 13. For para esperar el fin de los procesos bisnietos. ....	11
Imagen 14. For para esperar el fin de los procesos nietos.....	11
Imagen 15. Ejecución del programa32.....	12
Imagen 16. Procesos de nuestro programa32.....	13
Imagen 17. Código del programa 33.....	14
Imagen 18. Contenido del archivo que se modifica en el código.....	15
Imagen 19. Compilación y ejecución del programa 33.....	15
Imagen 20. Contenido del archivo después de la ejecución del programa.....	15
Imagen 21. Código del programa 31 .....	18
Imagen 22. Código del programa 32,.....	20

## Glosario de términos

**Proceso:** programa en ejecución.

**Fork:** es una llamada al sistema que hace referencia a la creación de una copia de si mismo por parte de un programa.

**PID:** abreviación para process ID, es el identificador de procesos.

**PPID:** abreviación de parent process ID, es el identificar del proceso padre de un proceso.

## Contenido

### Creación de procesos con la llamada al sistema fork

Para crear nuevos procesos se usa la llamada al sistema `fork()`.

Un procesos que hace una llamada a `fork()` hará que el sistema cree una copia del proceso original (padre) para que los dos procesos sigan sus caminos independientemente. El nuevo proceso, llamado hijo, tiene un nuevo PID. El PPID (parent's PID) será el proceso que ha llamado a `fork()`: el proceso padre.

Para que los procesos identifiquen quien es el proceso padre y quien el hijo, al código de retorno de `fork()` es diferente del padre que del hijo. En el caso del proceso padre, se le devuelve:

- PID del proceso hijo si se ha creado
- -1 si no se ha podido crear el proceso hijo

El proceso hijo, siempre se le devolverá el valor 0, por lo que sabrá que es el hijo. Por esto podemos ver en el código de muchos programas un `if` acompañado del `fork()`. El proceso hijo hereda la mayoría de los atributos del proceso padre, ya que se copian de su segmento de datos del sistema.

La llamada `fork()` es un proceso relativamente costoso para el sistema, por lo que puede ser aprovechado por un atacante para crear las llamadas `fork bombs`. Por esto lo ideal es restringir el número de procesos que pueden tener los usuarios no privilegiados.

Para terminar un proceso se usa `exit()`, esta finaliza al proceso que la llamó. Todos los descriptores de archivo abiertos son cerrados y sus buffers sincronizados. Si hay procesos hijo cuando el padre ejecuta un `exit`, el PPID de los hijos se cambia 1 (proceso `init`). Es la única llamada al sistema que nunca retorna.

Si hay varios procesos hijos, `wait(int *statusp)` espera hasta que uno de ellos termina. No es posible especificar por qué hijo se espera, `wait(int *statusp)` retorna el PID del hijo que termina (o -1 si no se crearon hijos) y almacena el código del estado de finalización del proceso hijo en la dirección apuntada por el parámetro `statusp`.

Un proceso puede terminar en un momento en el que su padre no le esté esperando. Como el kernel debe asegurar que el padre pueda esperar por cada proceso, los procesos hijos por los que el padre no espera se convierten en procesos zombie, es decir, se descartan sus segmentos, pero siguen ocupando una entrada en la tabla de procesos del kernel. Cuando el padre realiza una llamada `wait`, el proceso hijo es eliminado de la tabla de procesos.

En la imagen 1 se muestra un esquema básico de la creación de procesos con la llamada al sistema `fork()`.

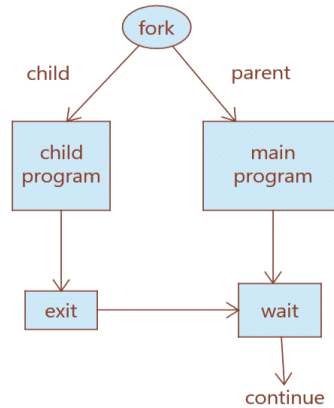


Imagen 1. Creación básica de procesos con fork().

## Desarrollo de la práctica

### Programa 31

Realizar un programa que cree diez procesos hijos del mismo padre y cada uno muestre el mensaje “Hola soy el proceso con pid XXXX y mi padre es XXXX” y el conteo del uno al diez. Al final el padre espera a los hijos y termina. El código completo se puede ver en los anexos.

Para poder realizar el programa se implementó un algoritmo similar a los vistos en clase, a través de un llamada al sistema **fork()** y un ciclo for, se fueron creando los 10 procesos hijos y con la llamadas **getpid()** y **getppid()** se imprimieron los identificadores (ID) del proceso hijo y padre, respectivamente. A continuación, la imagen 2 lo muestra:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <wait.h>
5 #include <unistd.h>
6 #define procesos 10
7
8
9 void main()
10 {
11     int x,salida,p;
12     int pid;
13
14     for(x=0;x<procesos;x++)
15     {
16
17         pid= fork();
18         if(pid==-1)
19         {
20             perror("\nError al crear el proceso\n");
21             exit(-1);
22         }
23         if(pid==0)
24         {
25             printf("\n%d.Hola soy el proceso con pid %d y mi
26 padre es %d\n",x+1,getpid(),getppid());
27             exit(0);
28         }
29     }
30 }

```

Imagen 2. Algoritmo de la creación de los procesos hijos del programa 31.c

Se puede notar que el algoritmo no es desconocido, debido a que ya se habían analizado unos similares.

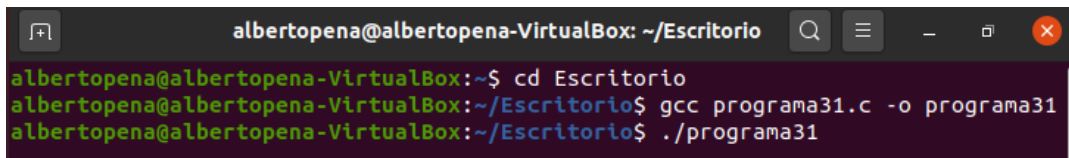
En otro parte del código se utiliza la llamada **wait()**, pero en esta ocasión para esperar a que terminen todos los procesos hijos. Esto lo podemos ver en la imagen 3.

```
26         exit(0);
27     }
28
29 }
30
31 sleep(2);
32 for(p=0;p<procesos;p++)
33 {
34     pid=wait(&salida);
35     printf("\nProceso %d terminado\n",pid);
36     sleep(1);
37 }
38
39
40 printf("\nEl proceso padre con PID: %d ha terminado\n",
getpid());
41
42
43
44 }
```

Imagen 3. Uso del wait() dentro del programa 31.c

Posteriormente se verá, de mejor manera, cómo el **wait()** funciona.

Luego se compilo para descartar la presencia de warnings o errores. Esta compilación la podemos ver en la imagen 4.



```
albertopena@albertopena-VirtualBox: ~/Escritorio
albertopena@albertopena-VirtualBox:~$ cd Escritorio
albertopena@albertopena-VirtualBox:~/Escritorio$ gcc programa31.c -o programa31
albertopena@albertopena-VirtualBox:~/Escritorio$ ./programa31
```

Imagen 4. Compilación del programa 31.c

Se nota la no presencia de errores.

Se ejecuta el programa y se puede notar que se cumple lo que se planteó debido a que se imprimen los 10 procesos hijos junto con su identificador y el de su padre, además de un contador, cabe mencionar que los procesos no se imprimen en orden porque todos los procesos hijos se ejecutan al mismo tiempo. Esto lo vemos en la imagen 5.

```
4.Hola soy el proceso con pid 4754 y mi padre es 4750
3.Hola soy el proceso con pid 4753 y mi padre es 4750
2.Hola soy el proceso con pid 4752 y mi padre es 4750
5.Hola soy el proceso con pid 4755 y mi padre es 4750

6.Hola soy el proceso con pid 4756 y mi padre es 4750
1.Hola soy el proceso con pid 4751 y mi padre es 4750
10.Hola soy el proceso con pid 4760 y mi padre es 4750
8.Hola soy el proceso con pid 4758 y mi padre es 4750
9.Hola soy el proceso con pid 4759 y mi padre es 4750
7.Hola soy el proceso con pid 4757 y mi padre es 4750
```

Imagen 5. Ejecución del programa 31.c

Se puede notar que no hay falla alguna y que los procesos hijos fueron terminando correctamente y al final el proceso padre también termina. Todo lo anterior con la llamada al sistema **wait()**. Esto lo vemos en la imagen 6.

```
Proceso 4751 terminado
Proceso 4752 terminado
Proceso 4753 terminado
Proceso 4754 terminado
Proceso 4755 terminado
Proceso 4756 terminado
Proceso 4757 terminado
Proceso 4758 terminado
Proceso 4759 terminado
Proceso 4760 terminado

El proceso padre con PID: 4750 ha terminado
albertopena@albertopena-VirtualBox:~/Escritorio$
```

Imagen 6. Finalización de los procesos hijos y padre

Esta es la última parte de ejecución del programa y se aprecia que se ejecutó correctamente el programa 31.

## Programa 32

Realice un programa que cree un proceso hijo a partir de un proceso padre, el hijo creado a su vez creará tres procesos hijos más. A su vez cada uno de los tres procesos creará dos procesos más. Cada uno de los procesos creados imprimirá en pantalla el PID de su padre y su propio PID.

Antes de empezar a codificar se realizó un esquema de cómo sería la creación de los procesos, esto lo vemos en la imagen 7.



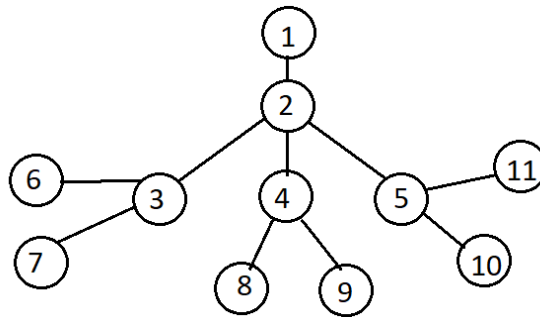


Imagen 7. Procesos del programa 32.

Como vemos en la imagen, existen 11 procesos, clasifiquémoslos:

- 1. Primer proceso, lo llamaremos el proceso bisabuelo
- 2. Proceso hijo de 1, lo llamaremos el proceso abuelo
- 3-5. Son los procesos hijos del abuelo, a estos los llamaremos nietos
- 6-11. Son los procesos hijos de cada nieto, a estos los llamaremos bisnietos

Una vez clasificados, procedemos a explicar el código del programa, este se encontrará completo en los anexos.

En la imagen 8, podemos ver la función `codigoProceso(int id)`.

```

void codigoProceso(int id) /*Esta función imprimira el identificador de un proceso hijo
                             y el identificador de su padre cada vez que sea creado*/
{
    printf("\nHola soy un proceso hijo\n");
    printf("Mi identificador es: %d\n", getpid()); //imprime el PID del proceso hijo
    printf("Mi proceso padre es: %d\n", getppid()); //imprime el PID del proceso padre
    exit(id); //Termina el proceso hijo
}
  
```

Imagen 8. Función `codigoProceso (int id)`.

Como vemos, esta función imprime en pantalla el PID del proceso hijo que llamo a esta función, así como el PID del proceso padre.

En la imagen 9 podemos ver la creación del primer proceso, es decir, el proceso bisabuelo según nuestra clasificación.

```

pid1 = fork(); //creamos el primer proceso

if (pid1 == -1)
{
    perror("\nError al crear el proceso\n"); //Verifica si se puede crear el proceso
    exit(-1);
}
else if(pid1==0) //este es el hijo del primer proceso
{
  
```

Imagen 9. Creación del primer proceso con `fork()`.

Como vemos, creamos el primer proceso con la llamada a `fork()` y con un `if` verificamos si hay algún error al crear este proceso, posteriormente, en el `else if`, comienza el código del hijo de este proceso, es decir del abuelo.

Afuera del código del proceso abuelo se encuentra el código de su padre, es decir del proceso bisabuelo, antes de entrar en el código del abuelo, veamos el código del bisabuelo, este se muestra en la imagen 10.

```
pid1=wait(&status);
printf("\nHola soy el proceso bisabuelo (primer padre)\n"); //PID del primer proceso padre y de su padre
printf("Mi identificador es: %d\n", getpid());
printf("Mi proceso padre es: %d\n", getppid());
```

Imagen 10. Proceso bisabuelo.

Como vemos en la imagen esperamos a que el proceso abuelo termine para que el bisabuelo realice sus acciones, que son mostrar su PID y el de su padre.

Ahora sí, expliquemos el código del proceso abuelo, en la imagen 11 vemos como se hace un `for` para que este proceso tenga 3 hijos que son los que se solicitan en el programa, por cada iteración del `for` se va creando un proceso nieto con la llamada a `fork()`.

```
else if(pid1==0) //este es el hijo del primer proceso
{
    for ( i = 0; i < 3; i++) //aquí hacemos un for para crear 3 hijos del hijo previamente creado
    {
        pid2=fork(); //por cada iteración se crea un proceso
        if (pid2==-1)
        {
            perror("Error al crear el proceso"); //Verifica si se puede crear el proceso
            exit(-1);
        }
        else if(pid2==0) //empieza el código de los hijos del primer hijo (nietos)
        {
```

Imagen 11. Creación de procesos nietos.

Dentro de cada proceso nieto, creamos dos procesos bisnietos, esto lo vemos en la imagen 12.

```
for ( i = 0; i < 2; i++) //de los 3 hijos del primer hijo se crearan dos hijos más por cada uno
{
    pid3=fork(); //por cada iteración se crea un proceso
    if(pid3==-1)
    {
        perror("Error al crear el proceso"); //Verifica si se puede crear el proceso
        exit(-1);
    }
    else if(pid3==0) //código de los procesos hijos
    {
        codigoProceso(id[i]); /*manda a llamar a la función que imprime el
                                PID del hijo y de su proceso padre*/
    }
}
```

Imagen 12. Creación de procesos bisnietos.

Como se ve en la imagen anterior, usamos un for que va de i igual a cero hasta i menor a dos, esto porque queremos que cada proceso nieto tenga dos procesos bisnietos. Por cada iteración se va creando un proceso gracias a fork(). Dentro del código del proceso bisnieto, mandamos a llamar a la función códigoProceso(id[i]), a esta se le manda el id de este proceso de acuerdo con la iteración en la que vaya el ciclo for.

Posteriormente a la creación de los procesos bisnietos, tenemos un ciclo for que espera a que terminen de ejecutarse esos procesos. Después de este for se manda a llamar a la función códigoProceso para que cada proceso nieto termine e imprima su PID y PPID. Esto lo vemos en la imagen 13.

```
for ( i = 0; i < 2 ;i++)
{
    pid3=wait(&status); //este for es para esperar a que terminen de ejecutarse los procesos hijos
}

códigoProceso(id[i]);/*manda a llamar a la función que imprime el
                        PID del hijo y de su proceso padre*/
} //termina el código de los procesos nietos
```

Imagen 13. For para esperar el fin de los procesos bisnietos.

En la imagen 14, vemos el for para esperar a que terminen los procesos nietos y las líneas que imprimen el PID del proceso abuelo y su PPID.

```
for ( i = 0; i < 3; i++)
{
    pid2=wait(&status); //este for es para esperar a que terminen de ejecutarse los procesos hijos
                        //para esperar se usa wait()
}

printf("\nHola soy el proceso abuelo (segundo padre)\n"); //PID del primer hijo y el de su padre
printf("Mi identificador es: %d\n", getpid());
printf("Mi proceso padre es: %d\n", getppid());

exit(0);
```

Imagen 14. For para esperar el fin de los procesos nietos.

En la imagen 15, vemos la ejecución del programa.

```
brandon@BDMV:~/programas so/practica 3$ gcc programa32.c -o p32
brandon@BDMV:~/programas so/practica 3$ ./p32

Hola soy un proceso hijo
Mi identificador es: 7899
Mi proceso padre es: 7894

Hola soy un proceso hijo
Mi identificador es: 7898
Mi proceso padre es: 7893

Hola soy un proceso hijo
Mi identificador es: 7896
Mi proceso padre es: 7893
Hola soy un proceso hijo
Mi identificador es: 7897
Mi proceso padre es: 7894

Hola soy un proceso hijo
Mi identificador es: 7893
Mi proceso padre es: 7892

Hola soy un proceso hijo
Mi identificador es: 7894
Mi proceso padre es: 7892

Hola soy un proceso hijo
Mi identificador es: 7901
Mi proceso padre es: 7895

Hola soy un proceso hijo
Mi identificador es: 7900
Mi proceso padre es: 7895

Hola soy un proceso hijo
Mi identificador es: 7895
Mi proceso padre es: 7892

Hola soy el proceso abuelo (segundo padre)
Mi identificador es: 7892
Mi proceso padre es: 7891

Hola soy el proceso bisabuelo (primer padre)
Mi identificador es: 7891
Mi proceso padre es: 6847
brandon@BDMV:~/programas so/practica 3$ █
```

Imagen 15. Ejecución del programa32.

Como vemos en la imagen tenemos los procesos creados, en la imagen 16 podemos observar una clasificación ya con estos valores.

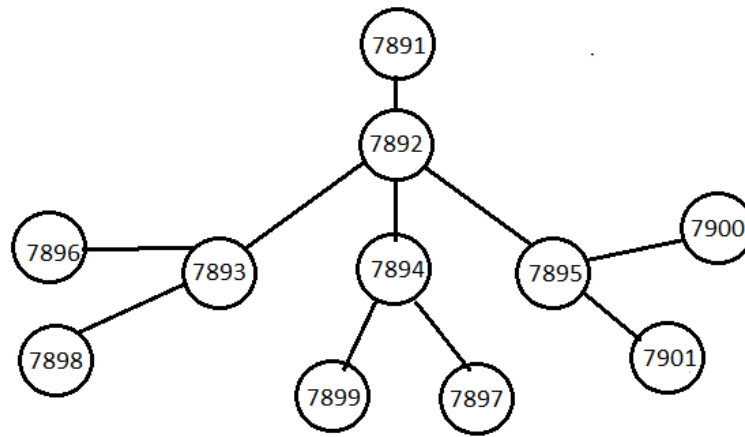


Imagen 16. Procesos de nuestro programa32.

Podemos ver que el proceso bisabuelo es el 7891, el abuelo es el 7892, los procesos nietos son los procesos 7893, 7894, 7895 y los procesos bisnietos son los procesos 7896, 7898, 7899, 7897, 7901, 7900.

## Programa 33

A continuación, se presenta la explicación del programa 33, así como su código que vemos en la imagen 17.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h> //Libreria necesaria para la llamada al sistema "wait()"
4  #include <unistd.h>
5  #include <string.h> //Libreria para el manejo de cadenas
6
7  #define Procesos 5 //Se define una variable Procesos con valor de 5
8
9  void proceso(int id, char cadena[], FILE* fichero){ //Funcion para crear un nuevo proceso
10
11     fputs(cadena, fichero); //Escribe una cadena en el archivo
12     sleep(2); //Duerme al proceso por 2 segundos
13
14     exit(id); //Termina el proceso
15 }
16
17 int main(int argc, char const *argv[]){
18
19     FILE* fichero; //Crea una variable de tipo FILE llamada fichero
20     fichero = fopen("practica33.txt", "wt"); //Se declara la variable fichero, con el archivo
21                                           //que se quiere editar
22
23     char cadenas[9] = {"Hola ", "esta ", "es ", "mi ", "practica "};
24     //Se declara un arreglo bidimensional de caracteres para la cadena "Hola esta es mi practica"
25
26     int p;
27     int id[Procesos] = {0, 1, 2, 3, 4}; //Declaracion de arreglo de id de procesos
28     int pid;
29     int salida;
30
31     for(p = 0; p < Procesos; p++){ //for que itera del 0 al 4
32         pid = fork();
33         //Llamada al sistema fork() que crea un nuevo proceso y retorna un entero indicando si el este fue creado o no
34         if(pid == -1){ //if que verifica si el nuevo proceso no se creo
35             perror("Error al crear el proceso"); //Si el proceso no fue creado, regresa el siguiente error
36             exit(-1); //Salida del proceso con error
37         } else if(pid == 0){ //En otro caso verifica si el proceso fue creado con éxito
38             proceso(id[p], cadenas[p], fichero); //Crea un nuevo proceso
39         }
40     }
41
42     for(p = 0; p < Procesos; p++){ //for que itera del 0 al 4
43         pid = wait(&salida); //Llamada wait() que obliga al proceso padre a esperar a los hijos
44         sleep(2); //el proceso se duerme por 2 segundos
45     }
46
47     fputs("uno", fichero); //El proceso padre escribe la ultima palabra de la cadena
48
49     fclose(fichero); //Se cierra el archivo
50 }
51 }
```

Imagen 17. Código del programa 33.

En la imagen anterior se puede apreciar un código que al ejecutarlo inicia 5 procesos y cada uno de estos escriben una palabra de la frase “Hola esta es mi práctica”, en específico esto sucede en la línea 11 del código y teóricamente esta frase debería ser escritas en orden, pero en la práctica esto no sucede, la frase se escribe según el orden en el que el sistema operativo decida ejecutar cada uno de los procesos

Para este código se inicializó una sola función para los procesos hijos y se utilizó un for para que se fueran creando cada uno de los 5 procesos, se declaró una variable fichero tipo apuntador y se le pasó como atributo a la función del proceso, para que así cada proceso hijo pudiese escribir en el archivo.

Ahora, se presenta la compilación y ejecución del programa. Pero antes se presentará el contenido del archivo para dar una visión detallada de la ejecución del programa que se ve en la imagen 18.

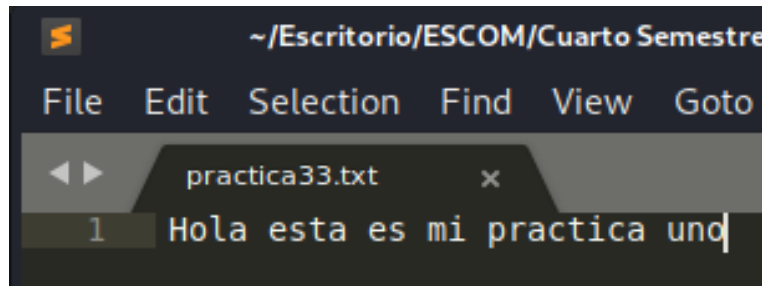


Imagen 18. Contenido del archivo que se modifica en el código.

En la imagen anterior se observa el contenido del archivo a modificar, en este caso su contenido tiene la frase a escribirse esto debido a una ejecución anterior, y como es evidente la frase está escrita en orden, sin embargo, no siempre sucede de esta forma.

En la imagen 19 se puede apreciar la compilación y ejecución del código.

```
(sergi@ sergi-kali)-[~/.../S0/Primer parcial/Unidad 2/Practica 3]
$ ls
practica33.txt  Programa33  Programa33.c

(sergi@ sergi-kali)-[~/.../S0/Primer parcial/Unidad 2/Practica 3]
$ gcc Programa33.c -o Programa33

(sergi@ sergi-kali)-[~/.../S0/Primer parcial/Unidad 2/Practica 3]
$ ./Programa33

(sergi@ sergi-kali)-[~/.../S0/Primer parcial/Unidad 2/Practica 3]
$
```

Imagen 19. Compilación y ejecución del programa 33.

Tal y como se puede observar en la imagen anterior, la ejecución del programa no muestra ninguna salida, ahora se muestra la imagen 20 del contenido del archivo actualizado.

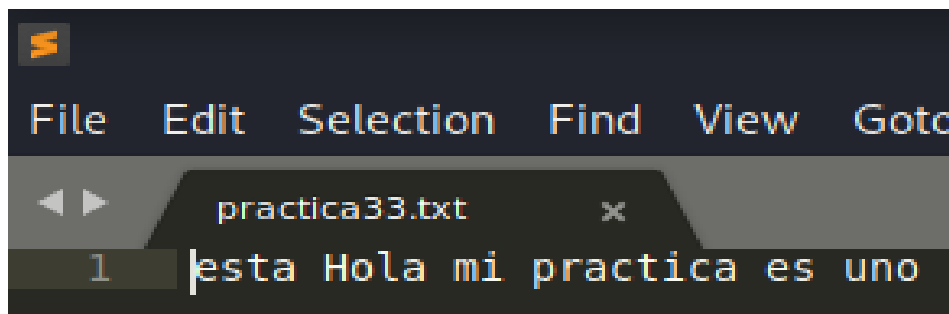


Imagen 20. Contenido del archivo después de la ejecución del programa.

Como se mencionó anteriormente, el orden de las palabras es aleatorio y eso es evidente en la imagen anterior.

## Conclusiones

### Martínez Ramírez Sergi Alberto

En esta práctica se pudo apreciar el manejo de procesos, por tanto, tuvimos la oportunidad de mejorar el conocimiento que teníamos acerca de los procesos. En mi caso, el tema de procesos fue bastante fácil de entender, ya que es similar a la función de un programa de código, debido a esto la implementación de los programas fue relativamente sencilla. He de decir que al principio estaba preocupado ya que al realizar el programa 33 no sabía cómo hacer que la frase se escribiera en orden, pensé en utilizar semáforos o memoria compartida, condicionar la ejecución de los procesos con if's, etcétera. Afortunadamente pude observar que no era necesario que se escribiera en orden la frase, ya que era parte del arte del programa, el ver como el sistema operativo decide ejecutar un proceso antes que otro y eso ocasiona que la frase esté desordenada.

### Meza Vargas Brandon David

La práctica fue sobre la creación de procesos con `fork()` y como se hace en el lenguaje c.

Fue una práctica muy interesante, pues no había tocado este tema en lenguaje c y mucho menos creado uno, como vimos en la práctica `fork()` nos sirve para crear un procesos. En los distintos programas realizados se pudo ver claramente la creación de procesos, pues en cada uno se crearon, así como procesos hijos.

En cada programa se hizo uso de una función llamada `PID()` para saber el id del proceso creado y la función `PPID()` para saber el id del proceso padre de un proceso hijo. Es importante mencionar que podemos crear un tipo de árbol de procesos como más o menos se vio en el programa 32 donde un proceso padre tenía un hijo, y este proceso hijo tenía más hijos. En la práctica se hizo un diagrama de cómo se podría ver esta descendencia.

De igual forma, en el programa 33 observamos claramente como un proceso puede realizar una acción de manera independiente a su proceso padre.

Por último, algo que me pareció curioso es que la creación de procesos no lleva un orden, ya que estos son creados de acuerdo con el sistema operativo y no podemos controlar esta parte.

Sin duda los procesos son una parte importante de cualquier sistema operativo, que complementados con otras herramientas como lo son los hilos, proporcionan un gran poder a un sistema operativo.

### Peña Atanasio Alberto

La práctica se realizó satisfactoriamente y se pudo entender de una mejor manera cómo es que los procesos padre e hijo interactúan. Se pudo ver que un proceso hijo se crea con una llamada al sistema a través de **`fork()`** y que este es una copia exacta del padre pero con variables y espacio de memoria asignados distintos a los del padre, lo único que cambia es el Identificador de procesos (PID) y el valor de retorno de la función **`fork()`**, de la anterior manera se puede diferenciar a ambos.

Se comprendió de una mejor manera cómo es que funciona la llamada al sistema **`wait()`**, esta simplemente espera a que se acaben de ejecutar los procesos hijos a través del padre.



Al principio se complicó un poco la práctica, debido a que el que los procesos se ejecuten de una manera muy rápida y esto hace que se pierda, de cierta manera, el orden y el sentido de la ejecución del programa, pero al final se entendió de buena forma cómo es que funcionan.

También se hizo uso de algunas bibliotecas nuevas, las cuales se desconocían, como la `unistd.h`, `sys/types.h` y la `wait.h`. Estas anteriores son fundamentales para que se ejecuten todas las llamadas utilizadas en la implementación del programa.

Y cómo se observó en los videos y en la práctica, los procesos son una unidad fundamental del Sistema Operativo, ya que este último está organizado en procesos.

### **Sarmiento Gutiérrez Juan Carlos**

En el desarrollo de esta práctica aprendí más sobre la creación de procesos y cómo estos se pueden relacionar en el modelo de padre e hijo, es decir un padre puede crear otros procesos, los cuales adquieren su propio pid. Además me di cuenta que desde el lenguaje C es muy sencillo crearlos, basta con llamar a la función `fork()` y de ahí se pueden obtener estos identificadores como pid o ppid que es de más ayuda ya que por cada pid hay un ppid. Por otro lado, me asombra pensar cómo es que el sistema es capaz de crearlos y administrarlos sabiendo eso de que cada padre tiene distintos hijos ya que como sabemos una computadora tiene miles de procesos activos y serán aún más procesos hijos. Sin duda una práctica buena para entender su manejo desde el lenguaje C.

## **Bibliografía**

- [1] A. Tenenbaum, Sistemas operativos modernos. México: Pearson Educación de México, 2009.
- [2] C. Rodríguez, “La función fork”, 2012. [En línea]. Disponible: [http://sopa.dis.ulpgc.es/ii-dso/leclinux/procesos/fork/LEC7\\_FORK.pdf](http://sopa.dis.ulpgc.es/ii-dso/leclinux/procesos/fork/LEC7_FORK.pdf)

## Anexos

### Código programa 31

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <wait.h>
5  #include <unistd.h>
6  #define procesos 10
7
8  void main()
9  {
10     int x,salida,p;
11     int pid;
12
13     for(x=0;x<procesos;x++)
14     {
15         pid= fork();
16         if(pid==-1)
17         {
18             perror("\nError al crear el proceso\n");
19             exit(-1);
20         }
21         if(pid==0)
22         {
23             printf("\n%d.Hola soy el proceso con pid %d y mi padre es %d\n",x+1,getpid(),getppid());
24             exit(0);
25         }
26     }
27     sleep(2);
28     for(p=0;p<procesos;p++)
29     {
30         pid=wait(&salida);
31         printf("\nProceso %d terminado\n",pid);
32         sleep(1);
33     }
34
35     printf("\nEl proceso padre con PID: %d ha terminado\n", getpid());
36
37 }
```

Imagen 21. Código del programa 31

## Código programa 32

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h> //permite utilizar la llamada al sistema fork()
4  #include <wait.h> //define las declaraciones de espera
5  #include <sys/types.h> //tipos de datos, permite usar pid_t
6
7  /*
8   Programa32.c Realice un programa que cree un proceso hijo a partir de un proceso padre,
9   el hijo creado a su vez creará tres procesos hijos más. A su vez cada uno de los tres
10  procesos creará dos procesos más. Cada uno de los procesos creados
11  imprimirá en pantalla el PID de su padre y su propio PID.
12  */
13
14  void codigoProceso(int id); //constructor de la función
15
16  void codigoProceso(int id) /*Esta función imprimirá el identificador de un proceso hijo
17  y el identificador de su padre cada vez que sea creado*/
18  {
19
20      printf("\nHola soy un proceso hijo\n");
21      printf("Mi identificador es: %d\n", getpid()); //imprime el PID del proceso hijo
22      printf("Mi proceso padre es: %d\n", getppid()); //imprime el PID del proceso padre
23      exit(id); //Termina el proceso hijo
24  }
25
26
27  int main()
28  {
29      int id[3]={1,2,3}, i;
30      pid_t pid1, pid2, pid3; //pid_t es una redefinición de un entero, estas variables son para hacer la llamada a fork()
31      int status;
32
33      pid1 = fork(); //creamos el primer proceso
34
35      if (pid1 == -1)
36      {
37          perror("\nError al crear el proceso\n"); //Verifica si se puede crear el proceso
38          exit(-1);
39      }
40      else if(pid1==0) //este es el hijo del primer proceso
41      {
42
43          for ( i = 0; i < 3; i++)//aquí hacemos un for para crear 3 hijos del hijo previamente creado
44          {
45              pid2=fork(); //por cada iteración se crea un proceso
46              if (pid2== -1)
47              {
48                  perror("Error al crear el proceso");//Verifica si se puede crear el proceso
49                  exit(-1);
50              }
51              else if(pid2==0) //empieza el código de los hijos del primer hijo (nietos)
52              {
53                  for ( i = 0; i < 2; i++) //de los 3 hijos del primer hijo se crearan dos hijos más por cada uno
54                  {
55                      pid3=fork(); //por cada iteración se crea un proceso
56                      if(pid3== -1)
57                      {
58                          perror("Error al crear el proceso");//Verifica si se puede crear el proceso
59                          exit(-1);
60                      }
61                      else if(pid3==0) //código de los procesos hijos
62                      {
63
64                          codigoProceso(id[i]); /*manda a llamar a la función que imprime el
65                          PID del hijo y de su proceso padre*/
66                      }
67                  }
68              }
69
70              for ( i = 0; i < 2 ;i++)
71              {
72                  pid3=wait(&status); //este for es para esperar a que terminen de ejecutarse los procesos hijos
73              }
74
75              codigoProceso(id[i]);/*manda a llamar a la función que imprime el
76              PID del hijo y de su proceso padre*/
77          }
78          //termina el código de los procesos nietos
79      }
80      for ( i = 0; i < 3; i++)
81      {
82          pid2=wait(&status);//este for es para esperar a que terminen de ejecutarse los procesos hijos
83          //para esperar se usa wait()
84      }
85  }
```

```
85 |  
86 |     printf("\nHola soy el proceso abuelo (segundo padre)\n"); //PID del primer hijo y el de su padre  
87 |     printf("Mi identificador es: %d\n", getpid());  
88 |     printf("Mi proceso padre es: %d\n", getppid());  
89 |  
90 |  
91 |     exit(0);  
92 | }//fin proceso abuelo  
93 |  
94 | pid1=wait(&status);  
95 | printf("\nHola soy el proceso bisabuelo (primer padre)\n");//PID del primer proceso padre y de su padre  
96 | printf("Mi identificador es: %d\n", getpid());  
97 | printf("Mi proceso padre es: %d\n", getppid());  
98 |  
99 |  
100 |     return 0;  
101 | }
```

Imagen 22. Código del programa 32