

Laboratorio 3 - Procesos

Versión original por	Modificado por
<ul style="list-style-type: none">● Henry Arcila● Danny Múnera	<ul style="list-style-type: none">● Juan Jaiber Yepes Zapata● Juan Guillermo Restrepo Pineda

1. Llamados al sistema

Los procesos solo pueden ejecutar algunas operaciones básicas en modo de usuario. Muchas de las operaciones privilegiadas relacionadas principalmente con los servicios del sistema operativo tienen que ser solicitadas al Kernel mediante **llamadas al sistema**.

Una llamada al sistema es entonces la interfaz por la cual los procesos acceden a las funciones del Sistema Operativo. Los llamados al sistema se encuentran implementados en el kernel y se ejecutan en modo privilegiado. En la escritura de programas, un llamado al sistema **se evoca como una función**, sin embargo a diferencia de una función ordinaria, cuando un programa llama a una función del sistema, los argumentos son empaquetados y manejados por el kernel, el cual toma el control de la ejecución hasta que la llamada se completa.

Una llamada al sistema **no es una llamada a una función ordinaria, y requiere un procedimiento especial para transferir el control al núcleo.** Básicamente el compilador genera una instrucción especial de máquina que, al ejecutarse, produce una interrupción de software ("trap" o excepción sincronica) para que se realice un cambio de modo en el procesador (de "usuario" a "privilegiado") y el kernel del sistema operativo realice las acciones restringidas, como las de acceso al hardware en general, necesarias para prestar el servicio pedido.

En el caso de Unix y Linux, los llamados al sistema se encuentran inmersos en funciones de librerías apropiados para cada uno de ellos. Por ejemplo, la función "time", de la librería "time.h", contiene la llamada para consultar el reloj del sistema; y la función "close", de la librería "unistd.h", contiene la llamada para cerrar un archivo.

En este enlace se encuentra una tabla de referencias con los principales llamados al sistema del sistema operativo Linux¹, esta tabla es de gran utilidad para el desarrollo de todas las prácticas siguientes, así que se recomienda su estudio.

1.1. Ejemplo de llamada al sistema

Uso de los llamados al sistema a través de dos métodos:

La función “syscall” y la función de la librería correspondiente:

```
1  #include <syscall.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <sys/types.h>
5
6  int main(void) {
7      long ID1, ID2;
8      /*-----*/
9      /* DIRECT SYSTEM CALL          */
10     /* SYS_getpid(func no. is 20)   */
11     /*-----*/
12     ID1 = syscall(SYS_getpid);
13     printf("syscall(SYS_getpid) = %1d\n", ID1);
14
15     /*-----*/
16     /* "libc" WRAPPED SYSTEM CALL   */
17     /* SYS_getpid(func no. is 20)   */
18     /*-----*/
19     ID2 = getpid();
20     printf("getpid() = %1d\n", ID2);
21     return 0;
22 }
```

Código 1. Ejemplo de llamados al sistema

Como habrá notado, la llamada del sistema invocada en el ejemplo fue “getpid”. En el primer caso se empleó la llamada indirecta con la función “syscall”², mientras que en el segundo caso se hizo la llamada al sistema con la función propiamente reservada para ello, “getpid”³.

¹ Tabla de referencias: <http://pubs.opengroup.org/onlinepubs/000095399/functions/exec.html>

² Sintaxis: syscall(int number, ...); Para mas informacion puede consultar: <http://man7.org/linux/man-pages/man2/syscall.2.html>

³ Para mas informacion puede consultar: <http://linux.die.net/man/2/getpid>

2. Procesos

Un proceso se puede definir como una **instancia en ejecución de un programa**, igualmente, concebimos un proceso como una **unidad básica de abstracción en el recurso de procesamiento**. En general los procesos son **componentes fundamentales de un sistema operativo**, por ende es necesario para nosotros manejar las herramientas que este nos ofrece para gestionar su funcionamiento.

El sistema operativo lleva el control de los procesos a través de la tabla de procesos, donde se almacena una estructura de datos llamada el *bloque control de procesos BCP* (Process Control Block PCB). En el BCP se encuentra información relevante del proceso como su identificador, su proceso padre, su estado, su program counter (PC), su stack pointer (SP), las referencias a su imagen de memoria, en general, todo aquello que se debe conocer del proceso para que pueda iniciarse nuevamente desde el punto donde este se detuvo para ceder la CPU a otro proceso.

Process State
Process Number
Program Counter
Registers
Memory Limits
List of Open files
...
Figura 1. Proceso

En los sistemas Linux, la estructura “task_struct” es la que contiene esta información. La declaración de la estructura puede verse en el archivo “sched.h” del código del kernel en su versión 3.0.4.

2.1. Servicios POSIX para la gestión de procesos

A continuación se realizará una revisión de los principales servicios que ofrece POSIX⁴ para la administración de procesos.

● Identificación de procesos

Se identifica a cada proceso por medio de un número único con representación entera. Este es conocido como el *identificador* del proceso (que es de tipo “pid_t”). La siguiente tabla muestra las principales funciones empleadas para este fin:

Función	Descripción
pid_t getpid(void)	Este servicio devuelve el identificador del proceso que realiza la llamada.
pid_t getppid(void)	Devuelve el identificador del proceso padre
uid_t getuid(void)	Devuelve el identificador del usuario real (usuario que ejecuta el proceso).
uid_t geteuid(void)	Devuelve el identificador del usuario efectivo (usuario en el que se ejecuta el proceso).
gid_t getgid(void)	Devuelve el identificador del grupo real.
gid_t getegid(void)	Devuelve el identificador del grupo efectivo.
Tabla 1. Servicios POSIX para identificar procesos	

● Gestión de procesos

Este grupo de funciones permite la creación y manipulación del estado de los procesos. La siguiente tabla describe brevemente estas funciones:

Función	Descripción
pid_t fork(void)	Permite crear procesos: Se realiza una clonación del proceso que lo solicita (conocido como proceso padre). El nuevo proceso se conoce como proceso hijo.
int execl (char *path, char *arg, ...)	Familia de funciones que permiten cambiar el programa que está ejecutando el proceso.
void exit (int status)	Termina la ejecución de un proceso y retorna “status”. Es similar a “return” de la función “main”.
pid_t wait (int *status)	Permite que un proceso padre espere hasta que finalice la ejecución de un proceso hijo. El proceso padre se queda

⁴ Siglas de “Portable Operating System Interface for uniX”.

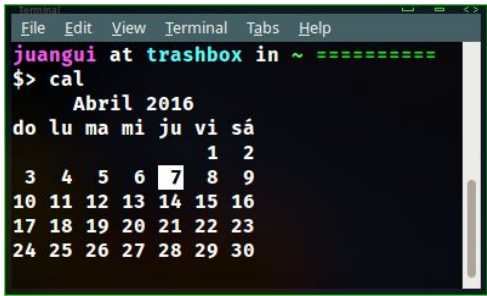
<code>pid_t waitpid (pid_t pid, int *status, int options)</code>	bloqueado hasta que termina el proceso hijo. Ambas llamadas permiten obtener información sobre el estado de terminación.
<code>int sleep (unsigned int seconds)</code>	Suspende el proceso durante un número de segundos. El proceso despierta cuando ha transcurrido el tiempo o cuando el proceso recibe una señal.
Tabla 2. Servicios POSIX para gestionar procesos	

2.2. Ejemplos de identificación y gestión de procesos

2.2.1. Creación de procesos

a. Comando del shell:

La función “system” permite la ejecución de comando en el shell dentro de un programa tal y como si estuviera digitando estos comandos en consola. La siguiente tabla resume los aspectos claves de esta función:

Función	system
Uso	<pre>#include <stdlib.h> int system(const char *command);</pre>
Descripción	La función “system()” ejecuta el comando especificado en “command” al llamar el comando “/bin/sh”, y retorna después de que el comando ha sido completado. El valor retornado será -1 en caso de error. En otro caso se retornará el valor del estado del comando.
Ejecución	<code>system(“cal”);</code>
Resultado	
Tabla 3. Función “system”	

Compile y ejecute el siguiente código:

1	<code>#include <unistd.h></code>
2	<code>#include <stdio.h></code>
3	
4	<code>int main(int argc, char *argv[]) {</code>
5	<code>int valor_retornado;</code>
6	<code>printf("ID del proceso: %d\n", (int)getpid());</code>
7	<code>printf("ID del proceso padre: %d\n", (int)getppid());</code>
8	<code>valor_retornado = system("cal");</code>
9	<code>valor_retornado = system("ls -l");</code>
10	<code>return valor_retornado;</code>
11	<code>}</code>
Código 2. Uso de “system”	

Responda:

- ¿Qué hace el programa anterior? ¿Cuál es su salida?
- ¿Qué hacen las funciones “getpid” y “getppid”?
- ¿Cuáles son los comandos del shell invocados en el programa?

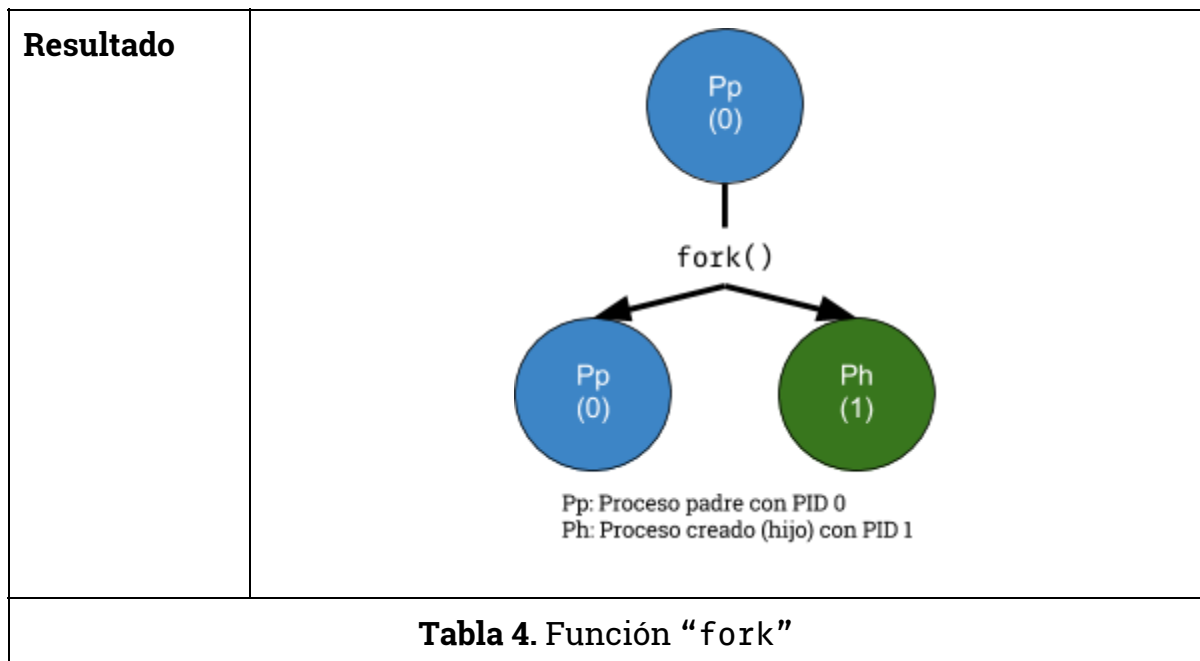
b. Usando fork

En la creación de procesos hay dos elementos claves:

- El proceso padre
- El proceso hijo

Crear un proceso consiste en realizar una copia en memoria de un proceso (*el padre*) por medio de una llamada al sistema. Dicha copia (*el hijo*) tiene toda la información del padre excepto que posee su propio **PID** (Identificador del proceso) y **PPID** (Identificador del proceso padre).

Función	fork
Uso	<code>#include <unistd.h></code> <code>pid_t fork(void);</code>
Descripción	La función “fork()” crea un nuevo proceso hijo como duplicado del proceso que la invoca (<i>padre</i>). Cuando la ejecución de la función es exitosa se retorna un valor diferente para el <i>padre</i> y para el <i>hijo</i> : <ul style="list-style-type: none"> ● Al padre se le retorna el PID del hijo. ● Al hijo se le retorna 0. El valor retornado será -1 en caso de que la llamada falle y no se pueda crear un proceso hijo.
Ejecución	<code>fork();</code>



Compile y ejecute el siguiente código:

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	<pre> #include <unistd.h> #include <stdio.h> int main(int argc, char *argv[]) { pid_t valor_retornado; printf("Ejemplo de fork. Este proceso va a crear otro proceso\n"); printf("El PID del programa principal es: %d\n", (int)getpid()); switch(valor_retornado = fork()) { case -1: // Caso de error printf("Error al crear el proceso"); return -1; case 0: // Código ejecutado por el hijo printf("PROCESO HIJO:\n"); printf("Mi PID es:%d\n", (int)valor_retornado); break; default: // Código ejecutado por el padre printf("PROCESO PADRE:\n"); printf("El PID de mi hijo es:%d\n", (int)valor_retornado); } // Código ejecutado tanto por el padre como el hijo printf("Finalizando el programa...\n"); return 0; } </pre>
Código 3. Uso de “fork”	

Responda:

- ¿Cuál es la salida del programa anterior y por qué?

c. Múltiples hijos

Un proceso puede crear o tener múltiples hijos llamando repetidamente la función `fork`. Y estos hijos pueden tener o crear otros procesos (*nietos*). La recomendación para este laboratorio es tener un proceso padre que cree a todos los hijos que se necesiten. A continuación se muestran algunas formas de trabajar con varios procesos:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  int main () {
5      pid_t pid_hijo1, pid_t pid_hijo2, pid_t pid_hijo3;
6
7      pid_hijo1 = fork(); // Creo el primer hijo
8
9      if (pid_hijo1 == 0) { // Hijo 1
10         printf("Soy el hijo 1");
11         sleep (5);
12
13     } else { // Padre
14
15         pid_hijo2 = fork(); // Creo al segundo hijo
16
17         if (pid_hijo2 == 0) { // Hijo 2
18             printf("Soy el hijo 2");
19             sleep (5);
20
21         } else { // Padre
22
23             pid_hijo3 = fork(); // Creo al tercer hijo
24
25             if (pid_hijo3 == 0) { // Hijo 3
26                 printf("Soy el hijo 3");
27                 sleep (5);
28
29             } else { // Padre
30                 printf("Soy el padre");
31                 sleep (5);
32             }
33         }
34     }
35     return 0;
36 }
```

Código 4. Múltiples llamados a “fork”


```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main () {
6      int i;
7      int numHijos = 3;
8      pid_t pid;
9      for (i = 0; i < numHijos; i++) {
10         pid = fork();
11         if (pid == -1) {
12             /* Error */
13             printf("No fue posible crear un hijo\n");
14             return -1;
15         }
16         if (pid == 0) {
17             printf("Soy el hijo #%d con PID: %d\n", i+1, getpid());
18             sleep (5);
19         }
20     }
21     return 0;
22 }

```

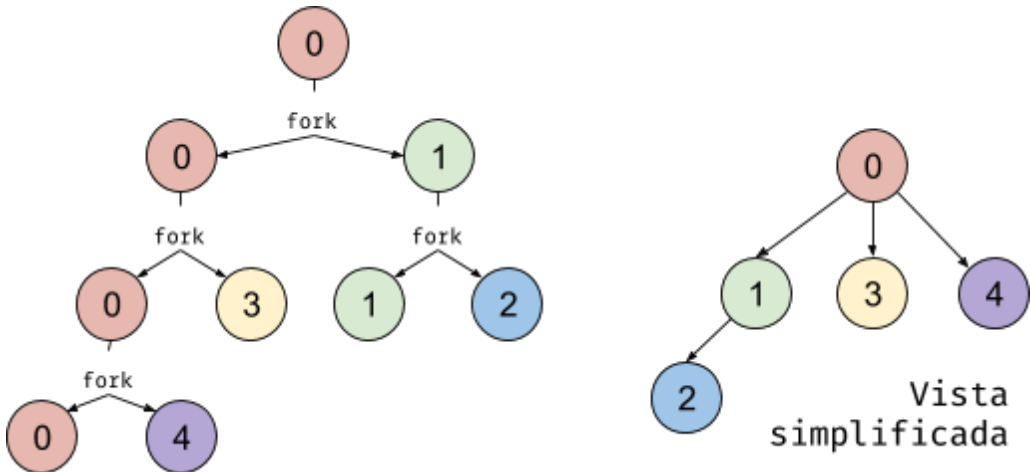
Código 5. Llamando a “fork” en un ciclo.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main () {
6
7      int pid;
8      int numHijos = 5;
9      int numProceso;
10
11     for(numProceso = 0; numProceso < numHijos; numProceso++) {
12         pid = fork();
13         if (pid == -1) {
14             // Imprimir algun mensaje de error
15         } else if(pid == 0) {
16             break;
17         }
18     }
19
20     if (pid == 0) {
21         // Logica del hijo
22         printf("Soy el hijo #%d\n", numProceso);
23     }

```

24	<code>else {</code>
25	<code>printf("Soy un padre perezoso\n");</code>
26	<code>}</code>
27	<code>}</code>
Código 6. Llamando a “fork” en un ciclo (otra forma).	

<p>En la siguiente jerarquía de procesos, cada uno de los procesos hijos deberá ir aumentando el valor de una variable y desplegándolo en pantalla:</p>	
 <pre> graph TD P0((0)) -- fork --> H0((0)) P0 -- fork --> H1((1)) H0 -- fork --> H0_0((0)) H0 -- fork --> H0_3((3)) H1 -- fork --> H1_1((1)) H1 -- fork --> H1_2((2)) H0_0 -- fork --> H0_0_0((0)) H0_0 -- fork --> H0_0_4((4)) </pre> <p>Vista simplificada</p> <pre> graph TD P0((0)) --> H1((1)) P0 --> H3((3)) P0 --> H4((4)) H1 --> H2((2)) </pre>	
<p>Note de la figura anterior que el proceso padre (0) tuvo en realidad 3 hijos (1, 3, 4) y un nieto (2). A continuación se muestra el código asociado el problema anterior.</p>	

1	<code>#include <unistd.h></code>
2	<code>#include <stdio.h></code>
3	
4	<code>int main(int argc, char *argv[]) {</code>
5	<code>pid_t pid_h1, pid_h2, pid_h3;</code>
6	<code>pid_t pid_n;</code>
7	<code>int i = 0;</code>
8	<code>pid_h1 = fork();</code>
9	<code>if(pid_h1 == 0) {</code>
10	<code>pid_n = fork();</code>
11	<code>if(pid_n==0) {</code>
12	<code>i++;</code>
13	<code>printf("NIETO: i = %d\n",i);</code>
14	<code>}else {</code>
15	<code>i++;</code>
16	<code>printf("HIJO 1: i = %d\n",i);</code>
17	<code>}</code>
18	<code>}else {</code>
19	<code>pid_h2 = fork();</code>

20	<code>if(pid_h2 == 0) {</code>
21	<code> i++;</code>
22	<code> printf("HIJO 2: i = %d\n",i);</code>
23	<code>}else {</code>
24	<code> pid_h3 = fork();</code>
25	<code> if(pid_h3 == 0) {</code>
26	<code> i++;</code>
27	<code> printf("HIJO 3: i = %d\n",i);</code>
28	<code> }else {</code>
29	<code> i++;</code>
30	<code> printf("PAPA: i = %d\n",i);</code>
31	<code> }</code>
32	<code>}</code>
33	<code>}</code>
34	<code>return 0;</code>
35	<code>}</code>
Código 7. Hijos y Nietos	

Responda:

- ¿Qué diferencias nota entre los tres ejemplos anteriores?
- ¿Cuál es la salida de estos ejemplos y por qué?

Algunas de las “anomalías” presentes en la ejecución de los ejemplos anteriores serán explicados en la siguiente sección.

2.2.2. Gestión de procesos

a. Terminación de procesos con exit

Así cómo es posible crear procesos también es posible culminarlos. Para ello existen dos maneras: por medio del uso de la función “exit” o usando la función “kill”. Veamos primero a “exit”:

Función	exit
Uso	<code>#include <stdlib.h></code> <code>void exit(int status);</code>
Descripción	Esta función causa la terminación normal de un proceso. La variable entera status es empleada para transmitir al proceso padre la forma en que el proceso hijo ha terminado. Por convención este valor suele ser 0 si el programa termina de manera exitosa u otro valor cuando la terminación de este es anormal.

Ejecución	exit(8);
Tabla 5. Función “exit”	

Compile y ejecute el siguiente código:

1	<code>#include <unistd.h></code>
2	<code>#include <stdio.h></code>
3	<code>#include <stdlib.h></code>
4	
5	<code>int main(int argc, char *argv[]) {</code>
6	
7	<code>pid_t pid_hijo, pid_padre;</code>
8	<code>printf("El pid del programa principal es: %d\n", (int) getpid());</code>
9	
10	<code>switch(pid_hijo=fork()) {</code>
11	
12	<code>case -1: /* Código ejecutado en caso de error*/</code>
13	<code>printf("Error al crear el proceso");</code>
14	<code>return -1;</code>
15	<code>case 0: /* Código ejecutado por el hijo */</code>
16	<code>printf("PROCESO HIJO:\n");</code>
17	<code>printf("PID del proceso: %d\n", (int) pid_hijo);</code>
18	<code>printf("PID del padre: %d\n", (int) getppid());</code>
19	<code>exit(0);</code>
20	<code>printf("Esta instrucción nunca se ejecutara en el proceso</code>
21	<code>hijo\n");</code>
22	<code>break;</code>
23	<code>default: /* Código ejecutado por el padre */</code>
24	<code>printf("PROCESO PADRE: Proceso hijo con PID %d</code>
25	<code>creado\n", (int) pid_hijo);</code>
26	<code>printf("PID del proceso hijo: %d\n", (int) pid_hijo);</code>
27	<code>printf("PID del padre: %d\n", (int) getppid());</code>
28	<code>}</code>
	<code>return 0;</code>
	<code>}</code>
Código 8. Uso de “exit”	

Al ejecutar el código anterior notara se espera que salga algo similar a la captura mostrada en la siguiente figura:

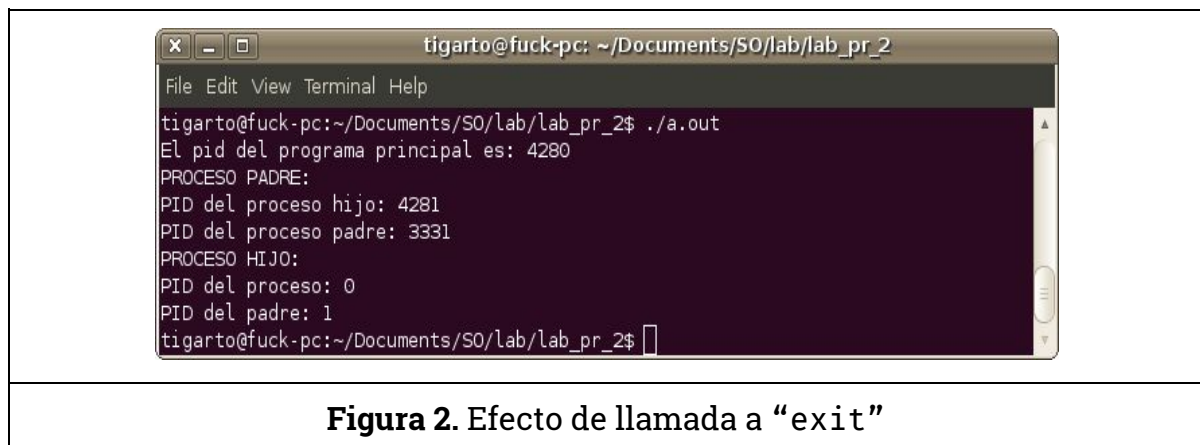


Figura 2. Efecto de llamada a “exit”

Observe bien la figura anterior y notará que una vez invocada la función “exit” el proceso hijo deja de ejecutarse, por eso todas las instrucciones colocadas después de esta llamada nunca se ejecutarán, por ello la salida anteriormente mostrada.

b. Esperar que un hijo termine

Si observa la Figura 2 notará algo interesante; el proceso padre culmina antes de que el hijo lo haga. Pues bien, existen ocasiones en las cuales es deseable que el proceso padre espere a que el proceso hijo culmine y es allí donde entra en juego la función “wait”. Básicamente lo que hace esta función es permitir esperar que la ejecución de un proceso hijo finalice y permitir al padre esperar recuperar información sobre la finalización del hijo. La siguiente tabla resume esta función:

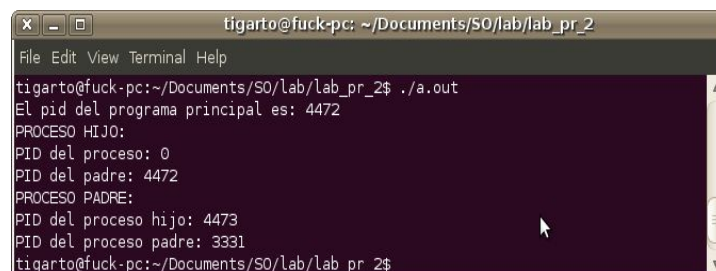
Función	wait
Uso	<pre>#include <sys/types.h> #include <wait.h> pid_t wait(int *status);</pre> <p>Dónde:</p> <ul style="list-style-type: none"> ● Valor retornado: Entero que contiene el PID del proceso hijo que finalizó o -1 si no se crearon hijos o si ya no hay hijos por los cuales esperar. ● Status: Puntero a la dirección donde la llamada al sistema debe almacenar el estado de finalización, o valor de retorno del proceso hijo (parámetro utilizado en la llamada exit).
Descripción	Esta función suspende la ejecución del proceso padre hasta que su hijo termine.
Tabla 6. Función “wait”	

Compile y ejecute el siguiente código:

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <wait.h>
6
7  int main(int argc, char *argv[]) {
8
9      pid_t pid_hijo, pid_padre;
10     int estado;
11     printf("El pid del programa principal es: %d\n", (int) getpid());
12     switch(pid_hijo=fork()) {
13         case -1: /*Codigo ejecutado en caso de error*/
14             printf("Error al crear el proceso");
15             return -1;
16         case 0: /*Codigo ejecutado por el hijo */
17             printf("PROCESO HIJO:\n");
18             printf("PID del proceso: %d\n", (int) pid_hijo);
19             printf("PID del padre: %d\n", (int) getppid());
20             exit(0);
21             printf("Esta instrucción nunca se ejecutara en el proceso
22 hijo\n");
23             break;
24         default: /*Codigo ejecutado por el padre */
25             wait(&estado);
26             printf("PROCESO PADRE: Proceso hijo con PID %d
27 creado\n", (int) pid_hijo);
28             printf("PID del proceso hijo: %d\n", (int) pid_hijo);
29             printf("PID del padre: %d\n", (int) getppid());
30     }
31     return 0;
32 }
```

Código 9. Uso de “wait”

Al ejecutar el código anterior la salida esperada es algo como la siguiente:



```
tigarto@fuck-pc: ~/Documents/SO/lab/lab_pr_2
File Edit View Terminal Help
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ ./a.out
El pid del programa principal es: 4472
PROCESO HIJO:
PID del proceso: 0
PID del padre: 4472
PROCESO PADRE:
PID del proceso hijo: 4473
PID del proceso padre: 3331
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$
```

Figura 3. Efecto de llamada a “wait”

Si se compara la Figura 2 con la Figura 3 podrá notar que ya hay algo diferente y es que una vez invocado el “wait”, el proceso padre no continua la ejecución de las instrucciones siguientes hasta que el proceso hijo culmine su ejecución.

Ahora bien, si hay varios procesos hijos, el proceso padre queda bloqueado hasta que uno de ellos culmina. Al finalizar uno de ellos, se liberan todos los recursos que tengan asociados, recuperándose el valor de retorno devuelto para que pueda ser accesible desde el proceso que realizó la llamada. El siguiente código clarifica un poco esto:

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <wait.h>
6
7  int main(int argc, char *argv[]) {
8      pid_t pid_h1, pid_h2, pid_h3;
9      int status_h1, status_h2, status_h3;
10     pid_t pid_n;
11     int status_n;
12     int i = 0;
13     pid_h1 = fork();
14     if(pid_h1 == 0) {
15         pid_n = fork();
16         if( pid_n==0 ) {
17             i++;
18             printf("NIETO: i = %d\n",i);
19         }
20         else {
21             wait(&status_n); // Papa (hijo 1) esperando hijo (nieto)
22             i++;
23             printf("HIJO 1: i = %d\n",i);
24         }
25     }
26     else {
27         pid_h2 = fork();
28         if(pid_h2 == 0) {
29             i++;
30             printf("HIJO 2: i = %d\n",i);
31         }
32         else {
33             pid_h3 = fork();
34             if(pid_h3 == 0) {
35                 i++;
36                 printf("HIJO 3: i = %d\n",i);
37             }
38             else {
39                 // El papa decidió esperar todos los hijos al final
```

40	wait(&status_h1); // Papa esperando hijo 1
41	wait(&status_h2); // Papa esperando hijo 2
42	wait(&status_h3); // Papa esperando hijo 3
43	i++;
44	printf("PAPA: i = %d\n",i);
45	}
46	}
47	}
48	
49	return 0;
50	}
Código 10. Uso de “wait” con varios hijos	

Como se puede ver en el código anterior, hay una invocación a “wait” por cada uno de los hijos esperados.

Responda:

- ¿Qué diferencias nota entre el Código 8 y el Código 10?
- ¿Cuál es la salida de estos ejemplos y por qué?

c. Terminación de procesos con kill

Es posible terminar abruptamente con la vida de un proceso, para ello se emplea la función “kill”, la cual a diferencia de la función “exit” termina a la brava dicho proceso.

“kill” también es un comando en consola, el cual se emplea pasando como argumento el PID del proceso que se desea culminar. Este comando trabaja enviando una señal de terminación (*SIGTERM*) la cual causa que el proceso culmine a menos que el programa tenga un *handler* para gestionar esta señal o que *SIGTERM* (la señal) se encuentra enmascarada. En lo que respecta a la función la siguiente tabla resume sus mayores atributos:

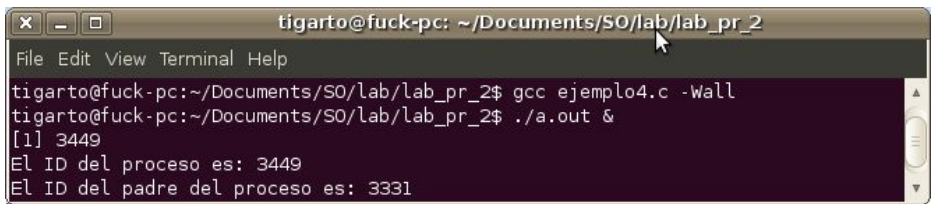
Función	kill
Uso	<pre>#include <sys/types.h> #include <signal.h> int kill(pid_t pid, int sig);</pre>
Descripción	Esta función es empleada para enviar una señal a cualquier proceso o grupo de procesos.

	El argumento pid es el PID del proceso que se desea killar mientras que el argumento sig está asociado a la señal que se desea enviar (KILL, TERM, TRAP, ALRM, SIGINT).
Tabla 7. Función “kill”	

Compile y ejecute el siguiente código:

1	<code>#include <unistd.h></code>
2	<code>#include <stdio.h></code>
3	<code>#include <stdlib.h></code>
4	<code>#include <sys/types.h></code>
5	<code>#include <signal.h></code>
6	
7	<code>int main(int argc, char *argv[]) {</code>
8	<code>printf("ID del proceso: %d\n", (int)getpid());</code>
9	<code>printf("ID del padre de este proceso: %d\n", (int)getppid());</code>
10	<code>for(;;) {</code>
11	<code> pause();</code>
12	<code>}</code>
13	<code>return 0;</code>
14	<code>}</code>
Código 11. Programa con ciclo infinito	

Luego compile el código y ejecútalo en **background**⁵ (esto para que no se bloquee la consola hasta que el programa termine y pueda ejecutar otros comandos), a continuación se muestra como:


Figura 4. Salida en pantalla

Si se invoca el comando “ps”, se puede ver la información más relevante de los procesos que actualmente se están ejecutando tal y como se muestra la siguiente figura:

⁵ Para ello use el símbolo ampersand (&) después del nombre del ejecutable

```
tigarto@fuck-pc: ~/Documents/SO/lab/lab_pr_2
File Edit View Terminal Help
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ ps
  PID TTY          TIME CMD
 3331 pts/0    00:00:00 bash
 3449 pts/0    00:00:00 a.out
 3450 pts/0    00:00:00 ps
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$
```

Figura 5. Ejecución del comando “ps”

Como se puede notar de la figura anterior, se despliegan 3 procesos la consola (**bash**), el proceso del comando **ps** y el proceso del ejecutable que se compilo y ejecuto (**a.out**). Como el programa tiene un ciclo infinito vamos a killarlo con el comando kill tal y como se muestra a continuación:

```
tigarto@fuck-pc: ~/Documents/SO/lab/lab_pr_2
File Edit View Terminal Help
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ kill -9 3449
[1]+  Killed                  ./a.out
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$
```

Figura 6. Ejecucion del comando kill

Como se puede notar en la figura anterior, el programa es culminado (si vuelve a ejecutar el comando ps notara que ya no aparece este proceso). Note una cosa importante, el comando kill⁶ se invocó pasando el **pid** del proceso a matar (3449) y el número⁷ de la señal a enviar (9). Una forma alternativa de invocar este comando seria: **kill -KILL 3449**, donde **KILL** es el mismo número 9.

Anteriormente se vio el uso de “kill” como comando, ahora veamos cómo es su empleo como función, para ello codifique y compile el siguiente código fuente:

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
```

⁶ Para más información sobre el comando kill:
<http://lowfatlinux.com/linux-kill-manual.html>
<http://es.wikipedia.org/wiki/Kill>

⁷ Para más información puede consultar las siguientes URLs:
[http://es.wikipedia.org/wiki/Se%C3%B1al_\(inform%C3%A1tica\)](http://es.wikipedia.org/wiki/Se%C3%B1al_(inform%C3%A1tica))
<http://en.wikipedia.org/wiki/Signal>

4	<code>#include <signal.h></code>
5	<code>#include <sys/types.h></code>
6	
7	<code>int main(int argc, char *argv[]) {</code>
8	<code>pid_t pid_hijo;</code>
9	<code>pid_hijo = getpid();</code>
10	<code>printf("El ID del proceso es: %d \n", (int)pid_hijo);</code>
11	<code>printf("El ID del padre del proceso es: %d \n", (int)getppid());</code>
12	<code>printf("Hola mundo. \n");</code>
13	<code>printf("Hola mundo. \n");</code>
14	<code>printf("Hola mundo. \n");</code>
15	<code>printf("Hola mundo. \n");</code>
16	<code>printf("Hasta la vista baby. \n");</code>
17	<code>kill(pid_hijo, 9); // Forma alternativa: kill(pid_hijo, SIGKILL);</code>
18	<code>printf("Hasta la vista baby. \n");</code>
19	<code>printf("Hasta la vista baby. \n");</code>
20	<code>printf("Hasta la vista baby. \n");</code>
21	<code>return 0;</code>
22	<code>}</code>
Código 12. Uso de “kill”	

2.2.3. Procesos Zombies y Procesos Huérfanos

Después de que un proceso hijo es creado por su padre haciendo uso de la función “fork” pueden suceder una de las siguientes cosas:

- Que el proceso padre espere a que el proceso hijo culmine haciendo uso de la función “wait”. En el caso normal, cuando el proceso hijo termina se le notifica su terminación al padre y se le manda el valor a la variable *status*. Ahora bien, por otro lado, puede suceder que el proceso padre se queda a la espera de que el hijo acabe y que este en efecto ya ha culminado, más exactamente, que el proceso hijo finalice antes de que el proceso padre llame la función “wait”. Cuando esto sucede, el proceso padre no puede recoger el código de salida de su hijo y por lo tanto el hijo se volverá un **proceso zombie**.
- Que el proceso padre no espere a que su hijo culmine, de tal manera que si el proceso padre culmina primero el proceso hijo será un **proceso huérfano**. Para este caso el nuevo identificador del padre será 1. (Identificador del proceso *init*).

Compile y ejecute el siguiente código el cual crea un proceso zombie. Elija como nombre del ejecutable *make-zombie*:

1	<code>#include <stdlib.h></code>
2	<code>#include <sys/types.h></code>

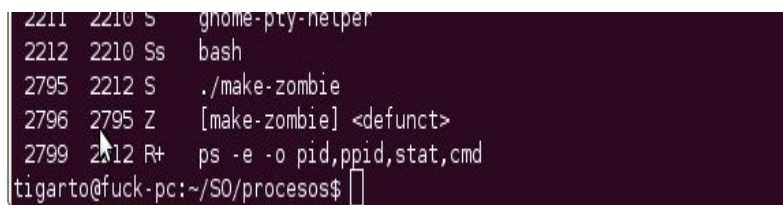
```

3  #include <unistd.h>
4  int main () {
5      pid_t child_pid;
6      /* Creacion del proceso hijo. */
7      child_pid = fork ();
8      if (child_pid > 0) {
9          /* Este es el proceso padre el cual duerme por 20 segundos. */
10         sleep (20);
11     }
12     else {
13         /* Este es el proceso hijo el cual culmina inmediatamente. */
14         exit (0);
15     }
16     return 0;
17 }

```

Código 13. make-zombie.c

Ejecute el programa anterior y una vez hecho esto, ejecuta en otra pestaña el comando *ps -e -o pid,ppid,stat,cmd*. Notará una salida algo similar como la de la siguiente figura:



```

2211 2210 S  gnome-ptty-helper
2212 2210 Ss  bash
2795 2212 S   ./make-zombie
2796 2795 Z   [make-zombie] <defunct>
2799 2212 R+  ps -e -o pid,ppid,stat,cmd
tigarto@fuck-pc:~/SO/procesos$

```

Figura 7. Salida en pantalla de ps

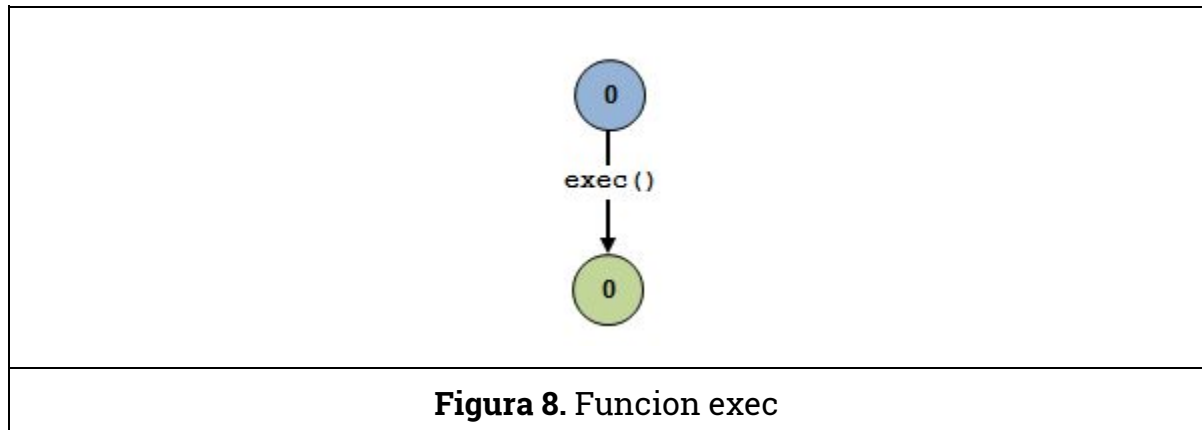
Observe además del proceso *make-zombie* la existencia de otro proceso (el hijo zombie: [make-zombie] <defunct>) cuyo código de estado es Z indicando que es zombie.

2.3. Ejecución de nuevos programas

a. Familia de funciones exec

Con anterioridad se trató la función **fork** la cual permitía la creación de un nuevo proceso el cual era una copia del proceso padre, la limitante al respecto era que al ser el nuevo proceso una copia del padre, lo que en realidad se estaba ejecutando era otra instancia de un mismo programa, esto impone una limitante la cual se traduce en la siguiente pregunta: ¿Es posible realizar la ejecución de nuevos programas?

Pues bien, afortunadamente existe una nueva función con la cual esta limitante puede ser superada, la función **exec**. Esta función reemplaza el programa que se está ejecutando en un proceso por otro programa. Cuando un programa llama una función **exec**, el proceso inmediatamente cesa de ejecutar el programa y empieza ejecutando un nuevo programa desde el principio (asumiendo que la llamada **exec** no encontró un error). La siguiente figura ilustra un poco lo anterior:



Dentro de la familia de funciones exec, hay funciones que varían levemente en sus capacidades y como son invocadas, la siguiente tabla trata esto con más detalle:

Función	exec
Uso	<pre>#include <unistd.h> int execl(const char *path, const char *arg,...); int execlp(const char *path, const char *arg,...); int execlx(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[]);</pre> <p>Donde</p> <ul style="list-style-type: none"> ● path o file: Cadena de caracteres que contiene el nombre del nuevo programa a ejecutar con su ubicación, /bin/cp por ejemplo. ● const char *arg: Lista de uno o más apuntadores a cadenas de caracteres que representan la lista de argumentos que recibirá el programa llamado. Por convención, el primer argumento deberá contener el nombre del archivo que contiene el programa ejecutado. El último elemento de la lista debe ser un apuntador a NULL.

	<ul style="list-style-type: none"> ● char *const argv[]: Array de punteros a cadenas de caracteres que representan la lista de argumentos que recibirá el programa llamado. Por convención, el primer argumento (argv[0]) deberá tener el nombre del archivo que contiene el programa ejecutado y el último elemento deberá ser un apuntador a NULL. ● char *const envp[]: Array de apuntadores a cadenas que contienen el entorno de ejecución (variables de entorno) que tendrá accesible el nuevo proceso. El último elemento deberá ser un apuntador a NULL.
Descripción	Esta familia de funciones, reemplaza la imagen actual del proceso con una nueva imagen de proceso. En caso de que la llamada a la función sea correcta esta no retorna nada, si hay una falla el valor retornado será -1.
Tabla 8. Familia de funciones “exec”	

Codifique y compile el siguiente código:

<pre> 1 #include <unistd.h> 2 #include <stdio.h> 3 int main(int argc, char *argv[]) { 4 printf("Ejecutable: \n"); 5 char *args[] = {"/bin/ls", "-l", ".", NULL}; 6 printf("Forma 1: \n"); 7 execl("/bin/ls", "/bin/ls", "-l", ".", NULL); 8 printf("Forma 2: \n"); 9 execv("/bin/ls", args); 10 printf("Forma 3: \n"); 11 execvp("/bin/ls", args); 12 return 0; 13 }</pre>	
Código 14. Uso de “exec”	

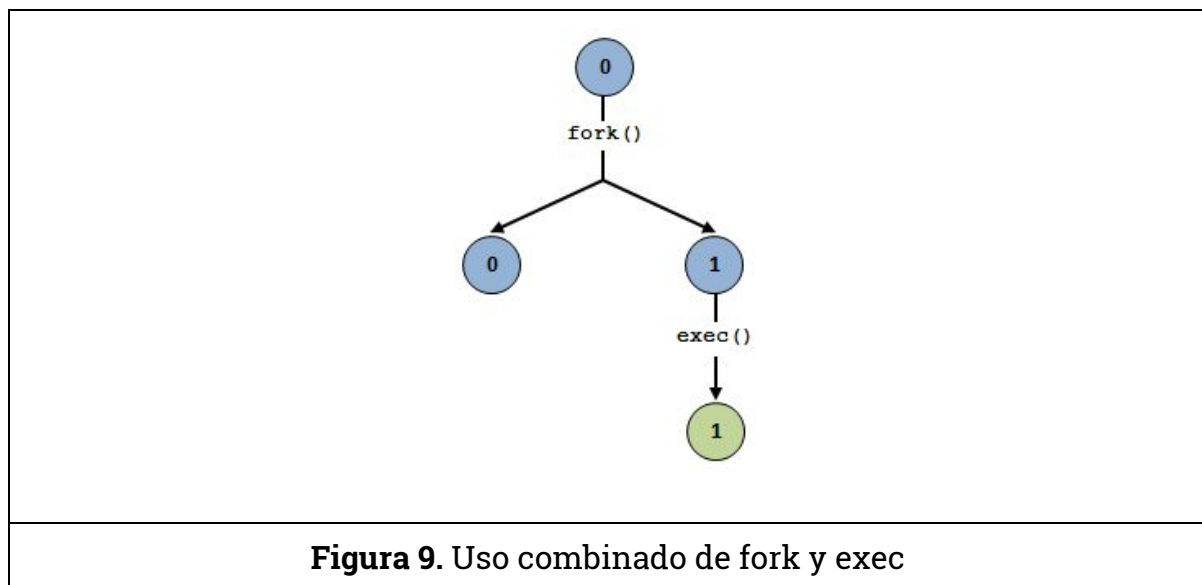
Responda:

- ¿Qué hace el programa anterior?
- ¿Qué tiene de raro la salida?
- Tome el código anterior y pártalo en 3 programas donde cada uno de estos debe colocar cada una de las diferentes invocaciones de la funciones de la familia **exec**; esto es, el código programa 1 debe usar **execl**, el programa 2 **execv** y el programa 3 **execvp**.

b. Usando fork y exec

Si ejecuto el programa del ejemplo 10 habrá notado que solo se ejecuta el primer llamado al **exec** (**execl**), los otros dos llamados (**execv** y **execvp**) una vez culmina la ejecución del **exec** inicialmente invocado, nunca son llamados, esto, porque el proceso invocador es sobre escrito por el nuevo ejecutable invocado. Ahora bien por el lado del **fork** encontramos que se podían crear copias de procesos y que esas copias en realidad siempre ejecutaban el mismo programa.

Así, según lo anterior tenemos una limitante, por un lado podemos crear copias pero estas ejecutan siempre lo mismo, y por otro lado podemos ejecutar un programa nuevo con llamado a una de las funciones de la familia **exec**, pero una vez hecho esto solo se puede ejecutar un solo programa. Pues bien, es posible solucionar limitantes de esta índole, esto mediante el uso combinado de las funciones **fork** y **exec** tal y como se muestra en la siguiente figura:



El efecto de usar estas dos funciones combinadas es que permiten que un programa pueda correr subprogramas. Como se muestra en la figura anterior, para correr un subprograma (nuevo programa invocado) dentro de un programa, lo primero que tiene que hacer el proceso padre es invocar la función **fork** para crear un nuevo proceso hijo (el cual es una copia casi exacta del padre), y luego ese proceso hijo recién creado lo que hace es invocar la función **exec** para empezar el nuevo programa. Lo anterior permite que el programa que realiza la invocación, continúe en ejecución en el lado de ejecución del proceso padre mientras que el programa llamado es reemplazado por el subprograma en el proceso hijo.

El siguiente fragmento de código muestra el esqueleto de como se hace uso de estas llamadas en conjunto:

```
1 // ...
2 if (fork == 0) {
3     // Este es el hijo
4     execvp(path, args); // Llamado a exec para ejecutar subprograma
5 }
6 else
7 {
8     // Este es el padre
9     // Llamado a wait para esperar a que el hijo termine
10    // (opcional: depende de la situacion)
11    wait(&status);
12 }
```

Código 15. Llamada en conjunto

Realizar un programa que invoque los comandos date y ls (ls debe listar el contenido del directorio raíz). El padre debe imprimir una vez que los dos subprocesos han culminado la frase **"Hasta la vista baby"**. A continuación se muestra el código asociado al ejemplo anterior:

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <wait.h>
6
7
8 int main(int argc, char *argv[]) {
9     pid_t pid_h1, pid_h2;
10    int status;
11    pid_h1 = fork();
12    if(pid_h1 == 0) {
13        // Proceso hijo el cual ejecuta el comando ls
14        execl("/bin/ls", "/bin/ls", "/", NULL);
15    }
16    else {
17        pid_h2 = fork();
18        if(pid_h2 == 0) {
19            // Proceso hijo que ejecuta el comando date
20            execl("/bin/date", "/bin/date", NULL);
21        }
22        else {
23            // Proceso padre
24            wait(&status); // wait para esperar un proceso
25            wait(&status); // wait para esperar el otro proceso
26            printf("Hasta la vista baby\n");
27        }
28    }
29 }
```


27	}
28	}
29	return 0;
30	}

Código 16. Hasta la vista baby

No solo es posible invocar comandos, también se pueden invocar ejecutables hechos por nosotros. Por ejemplo, supóngase que usted compiló un programa el cual imprimía la frase hola mundo, cuando realizó esto usted generó el ejecutable con nombre **myExe.out** el cual se encuentra en el directorio de trabajo actual, ahora bien, usted desea invocar este ejecutable desde otro programa con la función `exec`, el siguiente fragmento de código muestra como sería esto:

1	<i>// ...</i>
2	<code>if (fork == 0) {</code>
3	<i>// Este es el hijo</i>
4	<code>execl("./myExe.out", "./myExe.out", NULL);</code> <i>// Ejecutar</i>
5	<i>subprograma</i>
6	<code>}</code>
7	<code>else</code>
8	<code>{</code>
9	<i>// Este es el padre</i>
10	<i>// Llamado a wait para esperar a que el hijo termine</i>
11	<i>// (opcional: depende de la situacion)</i>
12	<code>wait(&status);</code>
13	<code>}</code>
	<i>...</i>

Código 17. Ejemplo de llamada en conjunto

Responda:

- Realizar un programa que ejecute un comando que no existe y luego ejecute el comando `ls -l`.

Taller

1. Escriba en consola “man syscalls” y responda: ¿Qué contiene esta llamada al sistema?

2. Explique qué hace el siguiente programa:

```
1  #include <syscall.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <sys/types.h>
5
6  int main(void) {
7      long ID1, ID2;
8      /*-----*/
9      /* DIRECT SYSTEM CALL */
10     /* SYS_getpid(func no. is 20) */
11     /*-----*/
12     ID1 = syscall(SYS_getpid);
13     printf("syscall(SYS_getpid) = %1d\n", ID1);
14
15     /*-----*/
16     /* "libc" WRAPPED SYSTEM CALL */
17     /* SYS_getpid(func no. is 20) */
18     /*-----*/
19     ID2 = getpid();
20     printf("getpid() = %1d\n", ID2);
21     return 0;
22 }
```

3. Ejecute el comando “man execl”. Liste las funciones e indique qué hace cada una de ellas.

4. Compile y ejecute el siguiente programa:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  main() {
7      int fd;
8      pid_t pid;
9      int num;
10     if ((pid = fork()) < 0) {
11         perror("fork falló");
12         exit(-1);
13     }
```

13	} else if (pid == 0) {
14	for (num=0; num<20; num++) {
15	printf("hijo: %d\n", num);
16	sleep(1);
17	}
18	} else {
19	for (num=0; num<20; num+=3) {
20	printf("padre: %d\n", num);
21	sleep(1);
22	}
23	}
24	}

Responda:

- ¿Qué significa el retorno de la función fork?
- ¿Cuál es la salida esperada en pantalla?
- ¿Cómo es posible que la sentencia printf reporte valores diferentes para la variable num en el hijo y en el padre?

5. Modifique los Códigos del 3 al 8 (añadiendo donde sea necesario un llamado a wait) para que los programas se ejecuten como realmente se esperaría.

6. Dado el siguiente código:

1	#include<stdio.h>
2	#include<unistd.h>
3	main() {
4	printf("Hola ");
5	fork();
6	printf("Mundo");
7	fork();
8	printf("!");
9	}

Resuelva:

- Sin ejecutarlo dibuje la jerarquía de procesos del programa y determine cuál es la posible salida en pantalla.
- Compile y ejecute el programa. ¿Es la salida en consola la que usted esperaba? ¿Cuál puede ser la razón de esto? (ayuda: función fflush: fflush(stdout);)
- Modifique el programa de tal manera que se creen exactamente 3 procesos, el padre imprime "Hola", el hijo imprime "Mundo" y el hijo del hijo imprime "!", exactamente en ese orden.

7. Dado el siguiente código:

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  main() {
7      pid_t pid;
8      int status;
9      printf("PADRE: Mi PID es %d\n", getpid());
10     printf("PADRE: El PID de mi padre es %d\n", getppid());
11     pid = fork();
12     if(pid == 0){
13         sleep(5);
14         printf("HIJO: Mi PID es %d\n", getpid());
15         printf("HIJO: El PID de mi padre es %d\n", getppid());
16         printf("HIJO: Fin!!\n");
17     }
18     else {
19         printf("PADRE: Mi PID es %d\n", getpid());
20         printf("PADRE: El PID de mi hijo es %d\n", pid);
21         // wait(&status);
22         // printf("PADRE: Mi hijo ha finalizado con estado %d\n",
23         status);
24         printf("PADRE: Fin!!\n");
25     }
26     exit(0);
27 }
```

Responda:

- ¿Cuál es la principal función de sleep en el código anterior?
- ¿Quién es el padre del padre? Use este comando:

ps -l

- ¿Por qué el proceso hijo imprime el id del padre como 1? ¿Es el que usted espera de acuerdo la jerarquía de procesos?
- Retire el comentario de las líneas de la función wait y la siguiente función printf. ¿Cuál es el identificador del padre ahora? ¿Para qué sirve la función wait? ¿Qué retorna en status?

8. Proceso Zombie

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
```

```

5  int main()
6  {
7      pid_t pid;
8      char *message;
9      int n;
10     printf("Llamado a fork\n");
11     pid = fork();
12     switch(pid) {
13         case -1:
14             perror("fork falló");
15             exit(1);
16         case 0:
17             message = "Este es el hijo";
18             n = 1;
19             break;
20         default:
21             message = "Este es el padre";
22             n = 30;
23             break;
24     }
25     for(; n > 0; n--) {
26         printf("n=%d ", n);
27         puts(message);
28         sleep(1);
29     }
30     exit(0);
31 }

```

Cuando un proceso hijo termina, su asociación con el padre continúa mientras el padre termina normalmente o realiza el llamado a wait. La entrada del proceso hijo en la tabla de procesos no es liberada inmediatamente. Aunque el proceso no está activo el proceso hijo reside aún en el sistema porque es necesario que su valor de salida exista en caso de que el proceso padre llame wait. Por lo tanto él se convierte en un proceso zombie.

- Realice el comando `ps -ux` en otra terminal mientras el proceso hijo haya finalizado pero antes de que el padre lo haga. ¿Qué observa en las líneas de los procesos involucrados?
- ¿Qué sucede si el proceso padre termina de manera anormal?

9. Familia de funciones execl: execl, execlp, execle, exect, execv y execvp y todas las que realizan una función similar empezando otro programa. El nuevo programa empezado, sobrescribirá el programa existente, de manera que nunca se podrá retornar al código original a menos que la llamada execl falle.

Programa 1:

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  main()
6  {
7      int pid;
8      if ((pid = fork()) == 0) {
9          execl("/bin/ls", "ls", "/", 0);
10     }
11     else {
12         wait(&pid);
13         printf("exec finalizado\n");
14     }
15 }
```

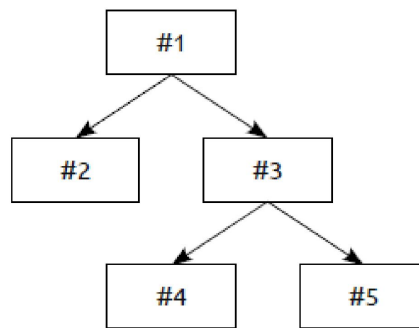
Programa 2:

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  int main()
5  {
6      printf("Corriendo ps con execlp\n");
7      execlp("ps", "ps", "-ax", 0);
8      printf("Echo.\n");
9      exit(0);
10 }
```

Resuelve:

- ¿Qué es lo que hace cada uno de los programas anteriormente mostrados?

10. Haga un programa que cree 5 procesos donde el primer proceso es el padre del segundo y el tercero, y el tercer proceso a su vez es padre del cuarto y el quinto:



El programa debe tener la capacidad de:

- Verificar que la creación de proceso con fork haya sido satisfactoria.
- Imprimir para cada proceso su id y el id del padre.
- Imprimir el id del proceso padre del proceso 1.
- A través de la función system imprimir el árbol del proceso y verificar la jerarquía (pstree).

11. Codifique un programa que haga lo siguiente:

- Cree 3 procesos diferentes.
- Cada uno de los procesos hijos, calculará por recursión el factorial de los enteros entre 1 y 10, imprimirá los resultados en pantalla y terminará.
- El mensaje impreso por cada proceso debe ser lo suficientemente claro de modo que sea posible entender cuál es el proceso hijo que está ejecutando la operación factorial.
- Una salida tentativa se muestra a continuación (esto no quiere decir que el orden en que se despliegue sea el mismo):

HIJ01: fact(1) = 1

HIJ02: fact(2) = 1

HIJ02: fact(2) = 2

HIJ01: fact(2) = 2

- El proceso padre tiene que esperar a que los hijos terminen.

12. Realice un programa llamado *ejecutador* que lea de la entrada estándar el nombre de un programa y cree un proceso hijo para ejecutar dicho programa.

13. Dado el siguiente fragmento de código:

```
1  #include<stdio.h>
2  #include<error.h>
3  #include<stdlib.h>
4  #include<fcntl.h>
5  int main(int argc, char *argv[]) {
6      int fd;
7      int pid;
8      char ch1, ch2;
9      fd = open("data.txt", O_RDWR);
10     read(fd, &ch1, 1);
11     printf("En el padre: ch1 = %c\n", ch1);
12     if ((pid = fork()) < 0) {
13         perror("fork falló");
14         exit(-1); //Sale con código de error
15     } else if (pid == 0) {
16         read(fd, &ch2, 1);
17         printf("En el hijo: ch2 = %c\n", ch2);
18     } else {
19         read(fd, &ch1, 1);
20         printf("En el padre: ch1 = %c\n", ch1);
21     }
22     return 0;
23 }
```

- Cree manualmente el archivo data .txt con el siguiente contenido:

hola

- Ejecute el programa, capture en pantalla la salida producida. ¿Por qué el programa produce la salida vista? ¿Qué sucede con un padre que abre un archivo, lo hereda?