



## Tutorial de Procesos

Por: Henry Arcila – Danny Múnera

## Contenido

Contenido.....	1
1 Llamados al sistema .....	1
2 Procesos .....	3
2.1 Servicios POSIX para la gestión de procesos.....	4
2.2 Ejemplos.....	6
2.2.1 Creación de Procesos.....	6
2.2.1.1 System .....	6
2.2.1.2 Usando Fork .....	7
2.2.2 Terminación de Procesos.....	9
2.2.2.1 Usando exit.....	9
2.2.2.2 Esperar que el hijo termine wait.....	11
2.2.2.3 Usando kill.....	15
2.2.2.4 Procesos Zombies y procesos Huérfanos.....	17
2.2.3 Ejecución de nuevos programas.....	18
2.2.3.1 Familia de funciones exec .....	18
2.2.3.2 Usando fork y exec .....	20
2.2.4 Señales.....	23
2.2.4.1 Usando signal .....	24
2.2.4.2 Usando sigaction .....	26
2.2.4.3 Manejadores de señales .....	28
3 Ejercicios .....	29
4 Referencias .....	34

## 1 Llamados al sistema

Los procesos solo pueden ejecutar algunas operaciones básicas en modo usuario. Muchas 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 invoca 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 se 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 sincrónica) 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 librería apropiados para cada una 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 referencia con los principales llamados al sistemas del sistema operativo Linux [1], esta tabla es de gran utilidad para el desarrollo de todas las prácticas siguientes, así que se recomienda su estudio.

## Ejemplos

Uso de los llamados al sistema a través de 2 métodos, la función syscall y la función de la librería correspondiente.

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

int main(void) {
    long ID1, ID2;
    /*-----*/
    /* direct system call */
    /* SYS_getpid(func no. is 20) */
    /*-----*/
    ID1 = syscall(SYS_getpid);
    printf ("syscall(SYS_getpid)=%ld\n", ID1);
    /*-----*/
    /* "libc" wrapped system call */
    /* SYS_getpid (Func No. is 20) */
    /*-----*/
    ID2 = getpid();
    printf ("getpid()=%ld\n", ID2);
    return(0);
}
```

Código 1. LLamados al sistema código ejemplo

Como habrá notado del ejemplo anterior, la llamada del sistema invocada en el ejemplo fue getpid. En el primer caso se empleó la llamada indirecta con la función syscall<sup>1</sup>, mientras que en el segundo caso se hizo la llamada al sistema con la función propiamente reservada para ello, getpid()<sup>2</sup>.

<sup>1</sup> Cuya sintaxis es: syscall(int number,...), para más información puede consultar:  
<http://www.kernel.org/doc/man-pages/online/pages/man2/syscall.2.html>

<sup>2</sup> Para más información puede consultar: <http://linux.die.net/man/2/getpid>

Preguntas :

1. Escriba en la consola **man syscalls**. Explique que contiene esta llamada al sistema?
2. Explique qué hace el programa anterior (Codigo 1)?

## 2 Procesos

Un proceso se puede definir como una instancia en ejecución de un programa, igualmente, concebimos un proceso como la 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.



Figura 1. Proceso

En los sistemas Linux es la estructura `task_struct` la que contiene esta información. La declaración de la estructura se puede ver 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 POSIX<sup>3</sup> 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 de proceso de tipo pid\_t . La siguiente tabla muestra las principales funciones empleadas para este fin:

Funcion	Descripción
pid_t getpid (void)	Este servicio devuelve el identificador del proceso que realiza la llamada
pid_t getppid (void)	Devuelve el identificado del proceso padre.
uid_t getuid (void)	Devuelve el identificador del usuario real que usa el proceso.
uid_t geteuid (void)	Devuleve el identificador del usuario efectivo. uid_t es un entero con el ID del propietario del proceso llamante.
gid_t getgid (void)	Obtiene el identificador del grupo real.
gid_t getegid (void;	Obtiene el identificador del grupo efectivo.

Tabla 1. Servicios POSIX para identificación de 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:

Funcion	Descripción
pid_t fork (void)	La creación de un proceso se realiza a través de este servicio. Se realiza una clonación del proceso que lo solicita. El proceso que solicita este servicio se convierte en el proceso padre del nuevo proceso que es a su vez el proceso hijo
int execl (char *path, char *arg, ...)	Cambia el programa que está ejecutando un proceso. Es toda una familia de funciones las cuales pueden ser consultadas en este enlace [i].
void exit (int status);	Termina la ejecución de un proceso y retorna status. Similar a return de la función main.
pid_t wait (int *status) pid_t waitpid(pid_t pid, int *status, int options)	Permite que un proceso padre espere hasta que finalice la ejecución de un proceso hijo. (El proceso se queda bloqueado hasta que termina el proceso hijo). Ambas llamadas permiten obtener información sobre el estado de terminación.
int sleep (unsigned int seconds)	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 gestión de procesos

- Entorno de un proceso:** El entorno de un proceso viene definido por una lista de variables que se pasan al mismo en el momento de comenzar su ejecución. Estas son las variables de entorno y son accesibles a un proceso a través de la variable externa environ (extern char \*\*environ). La siguiente tabla muestra algunas funciones relacionadas con lo anterior:

Función	Descripción
char *getenv (cnst char *name);	Permite buscar una variable de entorno dentro de la lista de variables de entorno de un proceso.
char *setenv (char **env)	Permite fijar el entorno de un proceso.

Tabla 3. Servicios POSIX para entorno de procesos

- Señales:** La siguiente tabla muestra los principales servicios para el envío y la captura de señales.

Función	Descripción
int kill (pid_t pid, int sig);	Envía la señal sig al proceso o grupo de proceso especificado por pid.
int sigaction (int sig, struct sigaction* act, struct sigaction * oact);	Permite armar una señal. Recibe tres parámetros: <ul style="list-style-type: none"> <li>el número de la señal para la que se quiere establecer el manejador.</li> <li>Un puntero a una estructura sigaction para establecer el nuevo manejador.</li> <li>Un puntero a la estructura sigaction para obtener información sobre el manejador creado.</li> </ul>
int pause (void);	Espera por la recepción de una señal (bloquea al proceso hasta que llega una señal). Cualquier señal no ignorada saca al proceso del estado de bloqueo.
unsigned int alarm (unsigned int seconds);	Esta función envía al proceso la señal SIGALARM después de pasados el número de segundos especificados en el parámetro seconds. Si seconds es igual a 0 cancelará cualquier petición anterior.

Tabla 4. Servicios POSIX para Señales.

## 2.2 Ejemplos

### 2.2.1 Creación de Procesos

#### 2.2.1.1 System

La función **system** permite la ejecución de comandos del Shell dentro de un programa tal y como si se estuvieran digitando estos comandos en consola. La siguiente tabla resume los aspectos claves de esta función:

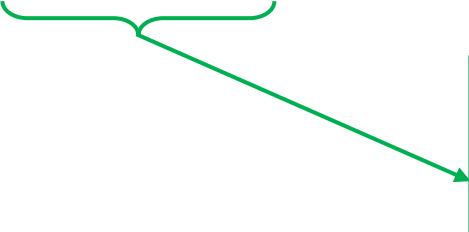
SYSTEM	
<b>Uso</b> <pre>#include &lt;stdlib.h&gt; Int system(const char *command);</pre>	
<b>Descripción</b> La función <b>system()</b> ejecuta el comando especificado en <b>command</b> al llamar el comando <b>/bin/sh</b> , y retorna después de que el comando ha sido completado. El valor retornado será -1 en caso de error y el valor del estado del comando en caso opuesto.	<pre>system("cal");</pre>  <div style="border: 1px solid black; padding: 5px;"><pre>[henry@microe ~]\$ cal septiembre de 2010 do lu ma mi ju vi sá       1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30</pre></div>

Tabla 5. Función system.

#### Ejemplo 1:

Compile y ejecute el siguiente código fuente:

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

int main(int argc, char *argv[]) {
    int valor_retornado;
    printf("El ID del proceso es: %d \n", (int) getpid());
    printf("El ID del padre del proceso es: %d \n", (int) getppid());
    valor_retornado = system("cal");
    valor_retornado = system("ls -l .");
    return valor_retornado;
}
```

**Preguntas:**

3. Explique brevemente lo que hace el programa anterior ¿Cuál es su salida?
4. ¿Qué es lo que hacen las funciones `getpid()` y `getppid()`?
5. ¿Cuáles son los comandos del shell invocados en el programa anterior?

### 2.2.1.2 Usando Fork

En la creación de un proceso hay dos elementos claves, el proceso padre y el proceso hijo, más precisamente crear un proceso consiste en realizar una copia en memoria de un proceso (padre) por medio de una llamada de sistema. Dicha copia (hijo) tiene todo lo del padre excepto que posee su propio **PID** (identificador del procesos hijo) y **PPID** (identificador del proceso padre).

La función empleada para crear una copia de un proceso es la función **fork**, la siguiente tabla resume un poco de lo que esta se trata:

FORK	
<b>Uso</b>	<pre>#include &lt;unistd.h&gt; pid_t fork(void);</pre>
<b>Descripción:</b>	<p>Esta función crea un nuevo proceso hijo como duplicado del proceso que la invoca (padre). Cuando la función es exitosa retorna un valor diferente para el padre que para el hijo; para el caso del primero (proceso padre) es un número positivo que contiene el PID del hijo, para el caso del segundo (proceso hijo) el valor retornado es 0. Si esta llamada falla no se puede crear el nuevo proceso hijo y el valor retornado será -1. La siguiente figura resume esto un poco:</p> <pre>graph TD; 0((0)) -- fork() --&gt; 0_1((0)); 0 -- fork() --&gt; 0_2((1))</pre>

Tenga en cuenta de la figura anterior que acá el proceso padre (0) solo creo un hijo (1).

Tabla 6. Función fork

## Ejemplo 2:

Compile y ejecute el siguiente código fuente:

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

int main(int argc, char *argv[]) {
pid_t pid_hijo;

printf("Ejemplo fork. Este proceso va a crear otro proceso\n\n");
printf("El pid del programa principal es: %d\n",(int)getpid());
switch(pid_hijo=fork()) {
    case -1: /* Código ejecutado en caso de error*/
        printf("Error al crear el proceso");
        return -1;
    case 0: /* Código ejecutado por el hijo */
        printf("PROCESO HIJO:\n");
        printf("PID del hijo: %d\n",(int)pid_hijo);
        break;
    default: /* Código ejecutado por el padre */
        printf("PROCESO PADRE: Proceso hijo con PID %d creado\n",(int)pid_hijo);
}
/*Esta linea sera ejecutada por ambos procesos (padre e hijo)*/
printf("Fin del proceso cuyo hijo tiene un PID de %d\n",(int)pid_hijo);
return 0;
}
```

Código 3. Código Ejemplo 2

## Preguntas

6. ¿Cuál fue la salida del código anterior y por qué?

## Ejemplo 3:

Cree la siguiente jerarquía de procesos, cada uno de los procesos hijos deberá ir aumentando el valor de una variable y desplegándolo en pantalla.

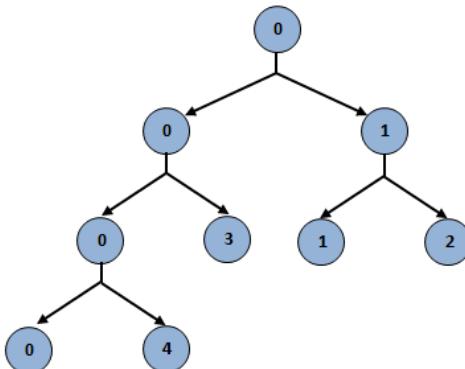


Figura 2. Familia de procesos

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 al problema anterior.

```

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

int main(int argc, char *argv[]) {
    pid_t pid_h1, pid_h2, pid_h3;
    pid_t pid_n;
    int i = 0;
    pid_h1 = fork();
    if(pid_h1 == 0) {
        pid_n = fork();
        if(pid_n==0) {
            i++;
            printf("NIETO: i = %d\n",i);
        }else {
            i++;
            printf("HIJO 1: i = %d\n",i);
        }
    }else {
        pid_h2 = fork();
        if(pid_h2 == 0) {
            i++;
            printf("HIJO 2: i = %d\n",i);
        }else {
            pid_h3 = fork();
            if(pid_h3 == 0) {
                i++;
                printf("HIJO 3: i = %d\n",i);
            }else {
                i++;
                printf("PAPA: i = %d\n",i);
            }
        }
    }
    return 0;
}

```

Código 4. Código Ejemplo 3

## 2.2.2 Terminación de Procesos

Así como 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**.

### 2.2.2.1 Usando exit

Esta función causa la terminación normal de un proceso, la siguiente tabla describe esto con más detalle:

EXIT
<p><b>Uso</b></p> <pre>#include &lt;stdlib.h&gt; void exit(int status);</pre> <p><b>Descripción</b></p> <p>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.</p>

Tabla 7. Función exit

**Ejemplo 4:**

Para observar el funcionamiento de **exit** codifique y compile el siguiente programa:

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

int main(int argc, char *argv[]) {
    pid_t pid_hijo, pid_padre;

    printf("El pid del programa principal es: %d\n", (int)getpid());
    switch(pid_hijo=fork()) {
        case -1: /* Código ejecutado en caso de error*/
            printf("Error al crear el proceso");
            return -1;
        case 0: /* Código ejecutado por el hijo */
            printf("PROCESO HIJO:\n");
            printf("PID del proceso: %d\n", (int)pid_hijo);
            printf("PID del padre: %d\n", (int)getppid());
            exit(0);
            printf("Esta instrucción nunca se ejecutara en el proceso hijo\n");
            break;
        default: /* Código ejecutado por el padre */
            printf("PROCESO PADRE: Proceso hijo con PID %d creado\n", (int)pid_hijo);
            printf("PID del proceso hijo: %d\n", (int)pid_hijo);
            printf("PID del padre: %d\n", (int)getppid());
    }
    return 0;
}
```

Código 5. Código Ejemplo 4

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

```
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ ./a.out
El pid del programa principal es: 4280
PROCESO PADRE:
PID del proceso hijo: 4281
PID del proceso padre: 3331
PROCESO HIJO:
PID del proceso: 0
PID del parente: 1
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ []
```

Figura 3. Salida en pantalla ejemplo 4

Observe bien la figura anterior y notara 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.

#### 2.2.2.2 Esperar que el hijo termine wait

Si observa la Figura 3 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:

WAIT
<b>Uso</b>
<pre>#include &lt;sys/types.h&gt; #include &lt;wait.h&gt; pid_t wait(int *status);</pre>
<b>Dónde:</b>
<ul style="list-style-type: none"> <li><b>Valor retornado:</b> 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.</li> <li><b>Status:</b> 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).</li> </ul>
<b>Descripción</b>
Esta función suspende la ejecución del proceso padre hasta que su hijo termine.

Tabla 8. Función Wait

### Ejemplo 5:

Para entender un poco lo anterior codifique y compile el siguiente código:

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

int main(int argc, char *argv[]) {
    pid_t pid_hijo, pid_padre;
    int estado;
    printf("El pid del programa principal es: %d\n", (int) getpid());
    switch(pid_hijo=fork()) {
        case -1: /* Código ejecutado en caso de error*/
            printf("Error al crear el proceso");
            return -1;
        case 0: /* Código ejecutado por el hijo */
            printf("PROCESO HIJO:\n");
            printf("PID del proceso: %d\n", (int) pid_hijo);
            printf("PID del padre: %d\n", (int) getppid());
            exit(0);
            printf("Esta instrucción nunca se ejecutara en el proceso hijo\n");
            break;
        default: /* Código ejecutado por el padre */
            wait(&estado);
            printf("PROCESO PADRE: Proceso hijo con PID %d creado\n", (int) pid_hijo);
            printf("PID del proceso hijo: %d\n", (int) pid_hijo);
            printf("PID del padre: %d\n", (int) getppid());
    }
    return 0;
}
```

Código 6. Código Ejemplo 5

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

```
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 4. Salida en pantalla ejemplo 5

Si se compara la Figura 3 con la Figura 4 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:

#### Ejemplo 6:

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

int main(int argc, char *argv[]) {
    pid_t pid_h1, pid_h2, pid_h3;
    int status_h1, status_h2, status_h3;
    pid_t pid_n;
    int status_n;
    int i = 0;
    pid_h1 = fork();
    if(pid_h1 == 0) {
        pid_n = fork();
        if( pid_n==0 ) {
            i++;
            printf("NIETO: i = %d\n",i);
        }
        else {
            wait(&status_n); // Papa (hijo 1) esperando hijo (nieto)
            i++;
            printf("HIJO 1: i = %d\n",i);
        }
    }
    else {
        pid_h2 = fork();
        if(pid_h2 == 0) {
            i++;
            printf("HIJO 2: i = %d\n",i);
        }
        else {
            pid_h3 = fork();
            if(pid_h3 == 0) {
                i++;
                printf("HIJO 3: i = %d\n",i);
            }
            else {
                // El papa decidió esperar todos los hijos al final
                wait(&status_h1); // Papa esperando hijo 1
                wait(&status_h2); // Papa esperando hijo 2
                wait(&status_h3); // Papa esperando hijo 3
                i++;
                printf("PAPA: i = %d\n",i);
            }
        }
    }
    return 0;
}
```

#### Código 7. Código Ejemplo 6

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

## Preguntas

7. ¿Cuál es el orden de terminación del código anteriormente ejecutado?
8. ¿Cuál es la diferencia entre la salida del código del ejemplo 6 (Código 7. Código Ejemplo 6) y el código del ejemplo 3 (

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

int main(int argc, char *argv[]) {
    pid_t pid_h1, pid_h2, pid_h3;
    pid_t pid_n;
    int i = 0;
    pid_h1 = fork();
    if(pid_h1 == 0) {
        pid_n = fork();
        if(pid_n==0) {
            i++;
            printf("NIETO: i = %d\n",i);
        }else {
            i++;
            printf("HIJO 1: i = %d\n",i);
        }
    }else {
        pid_h2 = fork();
        if(pid_h2 == 0) {
            i++;
            printf("HIJO 2: i = %d\n",i);
        }else {
            pid_h3 = fork();
            if(pid_h3 == 0) {
                i++;
                printf("HIJO 3: i = %d\n",i);
            }else {
                i++;
                printf("PAPA: i = %d\n",i);
            }
        }
    }
    return 0;
}
```

## 9. Código 4)?

10. Usando la llamada **fork**, cree dos procesos hijos, los cuales a su vez crean otro proceso hijo (nieto del inicial). Cada proceso debe devolver un valor diferente al finalizar (por ejemplo **exit(1)**, **exit (2)**, etc.), así mismo cada uno de los procesos (incluyendo el proceso padre) debe imprimir su PID y el PID de su padre. Muestre antes que finalice el proceso padre el valor returned por cada uno de los procesos; así mismo, tabule en una tabla los valores de los PID del proceso en cuestión y su padre.

### 2.2.2.3 Usando 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 encuentre enmascarada. En lo que respecta a la función la siguiente tabla resume sus mayores atributos:

KILL	
<b>Uso</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;signal.h&gt; int kill(pid_t pid,int sig);</pre>
<b>Descripción</b>	<p>Esta función es empleada para enviar una señal a cualquier proceso o grupo de procesos. El argumento <code>pid</code> es el PID del proceso que se desea killear mientras que el argumento <code>sig</code> está asociado a la señal que se desea enviar (KILL, TERM, TRAP, ALRM, SIGINT).</p>

Tabla 9. Función kill.

Para visualizar el uso tanto del comando como de la función sigamos los siguientes pasos:

#### Ejemplo 7:

Codifique el siguiente código y guárdelo<sup>4</sup>:

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

int main(int argc, char *argv[]) {
    printf("El ID del proceso es: %d \n", (int)getpid());
    printf("El ID del padre del proceso es: %d \n", (int)getppid());
    for(;;) {
        pause();
    }
    return 0;
}
```

Código 8. Código Ejemplo 7

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

<sup>4</sup>

En este ejemplo se guardó como ejemplo4.c

<sup>5</sup>

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

```
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ gcc ejemplo4.c -Wall
tigarto@fuck-pc:~/Documents/SO/lab/lab_pr_2$ ./a.out &
[1] 3449
El ID del proceso es: 3449
El ID del padre del proceso es: 3331
```

Figura 5. Salida en pantalla ejemplo 7

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:

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

Figura 7. Ejecución 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 **kill**<sup>6</sup> se invocó pasando el **pid** del proceso a matar (3449) y el número<sup>7</sup> de la señal a enviar (9). Una forma alternativa de invocar este comando seria: **kill -KILL 3449**, donde **KILL** es el mismo número 9.

#### Preguntas:

11. Vuelva a ejecutar el programa **a.out**, tome nota de los PID tanto del padre como del hijo: ¿Qué sucede se en vez de “killear” el proceso en cuestión se “killea” su padre?
12. ¿Qué puede deducir de los resultados anteriores, en realidad cual es el padre de **a.out**?

<sup>6</sup> Para más información sobre el comando **kill**:

<http://lowfatlinux.com/linux-kill-manual.html>  
<http://es.wikipedia.org/wiki/Kill>

<sup>7</sup> Para más información puede consultar las siguientes URLs:

[http://es.wikipedia.org/wiki/S%C3%A9%C3%B1al\\_\(inform%C3%A1tica\)](http://es.wikipedia.org/wiki/S%C3%A9%C3%B1al_(inform%C3%A1tica))  
<http://en.wikipedia.org/wiki/Signal.h>

### Ejemplo 8:

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:

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

int main(int argc, char *argv[]) {
    pid_t pid_hijo;
    pid_hijo = getpid();
    printf("El ID del proceso es: %d \n", (int)pid_hijo);
    printf("El ID del padre del proceso es: %d \n", (int)getppid());
    printf("Hola mundo. \n");
    printf("Hola mundo. \n");
    printf("Hola mundo. \n");
    printf("Hola mundo. \n");
    printf("Hasta la vista baby. \n");
    kill(pid_hijo,9); // Forma alternativa: kill(pid_hijo,SIGKILL);
    printf("Hasta la vista baby. \n");
    printf("Hasta la vista baby. \n");
    printf("Hasta la vista baby. \n");
    return 0;
}
```

Código 9. Código Ejemplo 8

#### 2.2.2.4 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).

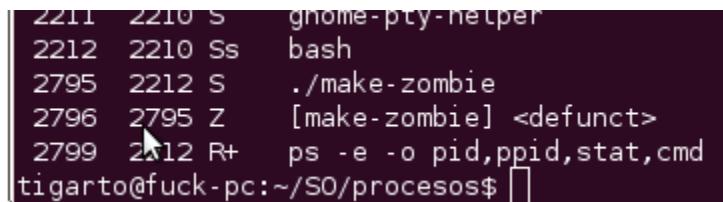
### Ejemplo 9:

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

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main () {
    pid_t child_pid;
    /* Creacion del proceso hijo. */
    child_pid = fork ();
    if (child_pid > 0) {
        /* Este es el proceso padre el cual duerme por 20 segundos. */
        sleep (20);
    }
    else {
        /* Este es el proceso hijo el cual culmina inmediatamente. */
        exit (0);
    }
    return 0;
}
```

Código 10. Código Ejemplo 9

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



```
2211 2210 S  gnome-pty-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 8. Salida en pantalla comando ps Ejemplo 9

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.

### 2.2.3 Ejecución de nuevos programas

#### 2.2.3.1 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:

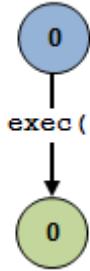


Figura 9. Función exec

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:

EXEC	
<b>Uso</b>	
<pre>#include &lt;unistd.h&gt;  int execl(const char *path, const char *arg,...); int execlp(const char *path, const char *arg,...); int execle(const char *path, const char *arg,            ...,char *const envp[]); int execv(const char *path, char *const argv[]); int execvp(const char *file, char *const argv[]);</pre>	
<b>Donde</b>	
<ul style="list-style-type: none"> <li>• <b>path o file</b>: Cadena de caracteres que contiene el nombre del nuevo programa a ejecutar con su ubicación, <b>/bin/cp</b> por ejemplo.</li> <li>• <b>const char *arg</b>: 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.</li> <li>• <b>char *const argv[]</b>: Array de punteros a cadenas de caracteres que representan la lista de argumentos que recibirá el programa llamado. Por convención, el primer argumento (arg0) deberá tener el nombre del archivo que contiene el programa ejecutado y el último elemento deberá ser un apuntador a NULL.</li> <li>• <b>char *const envp[]</b>: 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.</li> </ul>	
<b>Descripción</b>	
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 retornara nada, si hay una falla el valor retornado será -1.	

Tabla 10. Familia de funciones exec.

### Ejemplo 10:

Codifique y compile el siguiente código.

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

int main(int argc, char *argv[]) {
    printf("Ejecutable: \n");
    char *args[] = {"./bin/ls", "-l", NULL};
    // Forma 1: Con execl
    printf("Forma 1: \n");
    execl("./bin/ls", "./bin/ls", "-l", NULL);
    // Forma 2: Con execv
    printf("Forma 2: \n");
    execv("./bin/ls", args);
    // Forma 3: Con exec
    printf("Forma 3: \n");
    execvp("./bin/ls", args);
    return 0;
}
```

Código 11. Código Ejemplo 10

### Preguntas

13. ¿Qué es lo que hace el programa anterior?

14. ¿Qué tiene de raro la salida?

15. 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 **exec1**, el programa 2 **execv** y el programa 3 **execvp**.

#### 2.2.3.2 Usando fork y exec

Si ejecuto el programa del ejemplo 10 habrá notado que solo se ejecuta el primer llamado al **exec** (**exec1**), 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:

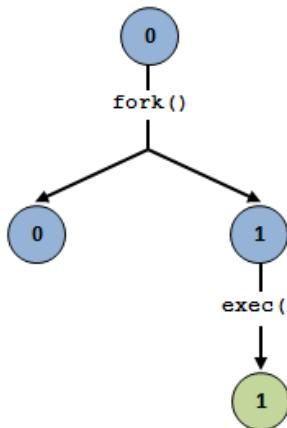


Figura 10. Uso combinado de fork y exec.

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:

```

...
if(fork==0) {
    // Este es el hijo
    execvp(path,args); //Llamado a la función exec para la ejecución del subprograma
}
else {
    // Este es el padre
    wait(&status); // Llamada a wait para esperar a que el hijo termine (opcional
                    // dependiendo la situación).
}
  
```

Código 12. Plantilla de código para el manejo de las funciones fork y exec.

### Ejemplo 11:

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 subprocessos han culminado la frase “**Hasta la vista baby**”.

A continuación se muestra el código asociado al ejemplo anterior:

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

int main(int argc, char *argv[]) {
    pid_t pid_h1, pid_h2;
    int status;
    pid_h1 = fork();
    if(pid_h1 == 0) {
        // Proceso hijo el cual ejecuta el comando ls
        execl("/bin/ls","/bin/ls","",NULL);
    }
    else {
        pid_h2 = fork();
        if(pid_h2 == 0) {
            // Proceso hijo que ejecuta el comando date
            execl("/bin/date","/bin/date",NULL);
        }
        else {
            // Proceso padre
            wait(&status); // wait para esperar un proceso
            wait(&status); // wait para esperar el otro proceso
            printf("Hasta la vista baby\n");
        }
    }
    return 0;
}
```

Código 13. Código ejemplo 11

No solo es posible invocar comandos, también se pueden invocar ejecutables hechos por nosotros. Por ejemplo, supóngase que usted compilo 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:

```
...
if(fork==0) {
    // Este es el hijo
    execl("./myExe.out","./myExe.out",NULL); // ejecución del subprograma
}
else {
    // Este es el padre
    wait(&status); // Llamada a wait (opcional)
}
...
```

Código 14. Código muestra que hace uso de fork y exec para invocar otro programa.

## Preguntas:

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

### 2.2.4 Señales

Las señales son un mecanismo para comunicar y manipular procesos en Linux. Básicamente, una señal es un mensaje enviado a un proceso cuando ocurre determinado evento. Las señales son asíncronas, de tal modo que cuando un proceso recibe una señal, este la procesa inmediatamente. El estándar POSIX definió un total de 64 señales, cada una de las cuales tiene un nombre que empieza por SIG y a las cuales se asocia un número entero positivo único (por ejemplo **SIGINT** es 2, **SIGQUIT** es 3 y **SIGKILL** es 9 entre otras). Los números de las señales están definidos en los archivos `signum.h`<sup>8</sup> y `signal.h`. También es posible ver las señales existentes y los números asociados a estas ejecutando el comando `kill -l`.

Cuando un proceso recibe una señal pueden pasar 3 cosas dependiendo de la disposición<sup>9</sup> de la señal:

- **Ignorar la señal:** Lo cual sucede cuando no pasa nada ante la ocurrencia de la señal.
- **Ejecutar una acción por defecto:** Una acción por defecto que determina lo que le sucede al proceso cuando una señal es enviada. Esta acción normalmente se ejecuta cuando el programa no especifica que es lo que se debe hacer cuando ocurre dicha señal. Por ejemplo la acción por defecto para la señal 15 (SIGTERM) es terminar el proceso.
- **Ejecutar una acción especial (signal handler):** Este caso se da cuando el programa especifica lo que se va a hacer cuando una señal es enviada. La función que contiene las instrucciones que se llevan a cabo cuando la señal es lanzada se conoce como **signal handler**<sup>10</sup>. Si el programa usa un signal handler, cuando ocurre un evento que envía una señal, la ejecución actual del programa es detenida, y el signal handler se ejecuta. Cuando el signal handler retorna, el programa continúa su ejecución normal donde se detuvo.

El manejador asociado será una función que no retorna ningún valor y que recibe un argumento entero que contendrá el valor numérico de la señal recibida. Lo anterior implica que la declaración de la función debe ser del tipo:

```
void handler(int signum);
```

Existen 2 formas mediante las cuales un proceso puede cambiar la disposición de una señal, estas son:

- Usando la función `sigaction`.
- Usando la función `signal`.

Veamos con más detalle lo anterior.

<sup>8</sup> Esta cabecera no se debería incluir en el archivo cuando éste está usando el archivo signal.h

<sup>9</sup> Por disposición se entiende a lo que hace el programa cuando una señal es lanzada.

<sup>10</sup> En español manejador de la señal.

## 2.2.4.1 Usando signal

La llamada a sistema **signal** es empleada para capturar, ignorar o asignar una acción por default a una señal especificada. Debido a que el comportamiento de **signal** varía a través de las versiones de Linux, esta no es tan portable por lo que no se recomienda su uso. La siguiente tabla detalla un poco esta función:

SIGNAL	
<b>Uso</b>	<pre>#include &lt;signal.h&gt; typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);</pre>
<b>Donde</b>	<ul style="list-style-type: none"><li>• <b>signum</b>: Entero que representa la señal (una de las posibles 64 tratadas con anterioridad) a trabajar.</li><li>• <b>handler</b>: Especifica la dirección del manejador de la señal. El manejador de la señal puede ser una función definida por el usuario o uno de los dos manejadores predefinidos en /usr/include/signal, para un total de 3 posibilidades. Los manejadores son:<ul style="list-style-type: none"><li>✓ <b>SIG_DFL</b>: Manejador por defecto.</li><li>✓ <b>SIG_INT</b>: Manejador que ignora la señal recibida.</li></ul></li></ul>
<b>Descripción</b>	<p>Esta función es usada para identificar el número de la señal y la forma como esta es tratada. Esta función retorna la dirección del manejador asignado antes de la ejecución al sistema o <b>SIG_ERROR</b> en caso de error.</p>

Tabla 11. Función signal

### Ejemplo 12:

El siguiente código corresponde a un ejemplo tomado de la revista **Linux Journal**<sup>11</sup> el cual ejemplifica el uso de la función **signal**:

```
#include <signal.h>

void my_handler (int sig); /* function prototype */

int main ( void ) {

/* Part I: Catch SIGINT */
    signal (SIGINT, my_handler);
    printf ("Catching SIGINT\n");
    sleep(3);
    printf (" No SIGINT within 3 seconds\n");

/* Part II: Ignore SIGINT */
    signal (SIGINT, SIG_IGN);
    printf ("Ignoring SIGINT\n");
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");

/* Part III: Default action for SIGINT */
```

<sup>11</sup>

<http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/064/6483/6483l1.html>

```

    signal (SIGINT, SIG_DFL);
    printf ("Default action for SIGINT\n");
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");
    return 0;
}

/* User-defined signal handler function */
void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);
    exit(0);
}

```

Código 15. Código ejemplo 12

El anterior programa muestra las diferentes disposiciones (ignorar, acción por default, manejador) que se pueden tomar ante la ocurrencia de la señal **SIGINT** (la cual se llama al usar la combinación de teclas: **CTRL + C**). Las siguientes capturas de pantalla muestran cada uno de los posibles escenarios de ejecución del programa ante el manejo de dicha señal.

La Figura 11 muestra el programa en ejecución, para este caso nunca se presionó la combinación **CTRL + C** por lo que el programa ejecutó todas las instrucciones del **main** de manera normal.

```
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$ ./a.out
Catching SIGINT
No SIGINT within 3 seconds
Ignoring SIGINT
No SIGINT within 3 seconds
Default action for SIGINT
No SIGINT within 3 seconds
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$ █
```

Figura 11. Ejecución programa ejemplo 11 sin lanzar la señal SIGINT

La Figura 12 muestra el programa nuevamente en ejecución; pero en este caso, se lanzó la señal SIGINT (presionando la combinación **CTRL + C**) justo después de que se asignó un manejador para esta usando la función **signal**, note que una vez sucede esto la ejecución del normal programa se detiene y pasa a ejecutarse el manejador de la señal, luego como este tiene una llamada a **exit** el programa culmina.

```
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$ ./a.out
Catching SIGINT
^CI got SIGINT, number 2
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$
```

Figura 12. Ejecución programa ejemplo 11 lanzando la señal SIGINT para ser gestionada por un handler

En la Figura 13 nuevamente se inicia la ejecución del programa pero en este caso la señal es lanzada justo después de que se registra para esta una disposición en la cual va a ser ignorada (**signal(SIGINT, SIG\_IGN)**), el resultado de esto es que pese a que la señal SIGINT fue lanzada el programa continua su curso normal porque la disposición asignada fue para ignorarla.

```
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$ ./a.out
Catching SIGINT
  No SIGINT within 3 seconds
Ignoring SIGINT
^CNo SIGINT within 3 seconds
Default action for SIGINT
No SIGINT within 3 seconds
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$
```

Figura 13. Ejecución programa ejemplo 11 lanzando la señal SIGINT para ser ignorada

Finalmente, la Figura 14 en la cual el programa nuevamente empieza su ejecución se muestra el caso en el cual la señal es lanzada cuando la disposición registrada para manejar la señal es que se ejecute la función por defecto. En este caso, para la señal SIGINT la disposición por defecto es terminar la ejecución del programa por lo que el programa culmina tal y como se muestra en la figura.

```
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$ ./a.out
Catching SIGINT
  No SIGINT within 3 seconds
Ignoring SIGINT
No SIGINT within 3 seconds
Default action for SIGINT
^C
tigarto@fuck-pc:~/Documents/SO/lab/lab_03/ensayos/linux_journal$
```

Figura 14. Ejecución programa ejemplo 11 lanzando la señal SIGINT para ser gestionada por el handler por defecto.

#### 2.2.4.2 Usando sigaction

Esta función hace básicamente lo mismo que la anterior pero es más flexible y portable. Antes de entrar a tratar con más detalle esta función conviene hablar un poco de la estructura **sigaction**, la cual se muestra a continuación:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

Código 16. Struct sigaction

Dónde:

- **sa\_handler**: Es un puntero al manejador de la señal predefinido (SIG\_IGN o SIG\_DFL) o definido por el usuario.
- **sa\_mask**: Específica una máscara de señales cuando la señal es manejada. Para evitar el bloqueo de señales las banderas SA\_NODEFER o SA\_NOMASK pueden ser usadas.

- **sa\_flags:** Especifica la acción de la señal. Conjuntos de banderas están disponibles para controlar el funcionamiento de la señal de manera diferente. Más de una bandera puede ser usada usando or.
- **sa\_action:** Si la bandera SA\_SIGINFO es usada en sa\_flags, en vez de especificar los manejados en sa\_handler, sa\_action puede ser usado. Sa\_action es un puntero a una función que tiene tres argumentos, no como los sa\_handlers (los cuales contienen un solo argumento).

Ahora bien, la siguiente tabla muestra con más detalle la función de interés (**sigaction**):

SIGACTION
<b>Uso</b>
<pre>#include &lt;signal.h&gt; int sigaction(int signum, const struct sigaction *act,               struct sigaction *oldact);</pre>
<b>Donde</b>
<ul style="list-style-type: none"> <li>• <b>signum:</b> Especifica la señal y puede ser cualquier señal válida excepto <b>SIGKILL</b> y <b>SIGSTOP</b>.</li> <li>• <b>act:</b> Si este apuntador tipo <b>struct sigaction</b> (tratado anteriormente) es no nulo, instala una nueva acción para la señal especificada en <b>signum</b>. Especifica la dirección del manejador de la señal.</li> <li>• <b>oldact:</b> Si este parámetro es no nulo, la acción previa es almacenada en <b>oldact</b>.</li> </ul>
<b>Descripción</b>
Esta función es usada para cambiar la acción tomada por un proceso en respuesta a una señal específica. Esta función retorna 0 en caso de éxito o -1 cuando hay error.

Tabla 12. Función Sigaction

### Ejemplo 13:

El siguiente programa hace lo mismo que el programa mostrado en el ejemplo 12, pero en esta ocasión se hace uso de la función `sigaction` (la cual es más portable), el código se tomó de la revista **Linux Journal**<sup>12</sup>.

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

void my_handler (int sig); /* function prototype */

int main ( void ) {

    struct sigaction my_action;

/* Part I: Catch SIGINT */

    my_action.sa_handler = my_handler;
    my_action.sa_flags = SA_RESTART;
    sigaction (SIGINT, &my_action, NULL);
    printf ("Catching SIGINT\n");
    sleep(3);
    printf (" No SIGINT within 3 seconds\n");

/* Part II: Ignore SIGINT */

    my_action.sa_handler = SIG_IGN;
    my_action.sa_flags = SA_RESTART;
    sigaction (SIGINT, &my_action, NULL);
    printf ("Ignoring SIGINT\n");
    sleep(3);
    printf (" Sleep is over\n");

/* Part III: Default action for SIGINT */

    my_action.sa_handler = SIG_DFL;
    my_action.sa_flags = SA_RESTART;
    sigaction (SIGINT, &my_action, NULL);
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");
}

void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);
    exit(0);
}
```

Código 17. Código ejemplo 13

#### 2.2.4.3 Manejadores de señales

Un manejador de una señal debería realizar el menor trabajo posible para responder a la señal, y entonces retornar el control al programa principal. En muchos casos, esto consiste simplemente en el hecho de indicar que una señal ha ocurrido para lo cual se puede hacer uso de una bandera la cual es chequeada en el programa principal periódicamente.

Por lo tanto es recomendable evitar realizar cualquier operación IO o llamar muchas funciones de librería y sistema en los manejadores de la señal.

<sup>12</sup>

<http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/064/6483/648312.html>

### 3 Ejercicios

Compile y ejecute el siguiente programa:

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

main() {
    int fd;
    pid_t pid;
    int num;
    if ((pid = fork()) < 0) {
        perror("fork falló");
        exit(-1);
    } else if (pid == 0) {
        for (num=0; num<20; num++) {
            printf("hijo: %d\n", num);
            sleep(1);
        }
    } else {
        for (num=0; num<20; num+=3) {
            printf("padre: %d\n", num);
            sleep(1);
        }
    }
}
```

Responda las siguientes preguntas:

17. ¿Qué significa el retorno de la función fork?

18. ¿Cuál es la salida esperada en pantalla?

19. ¿Cómo es posible que la sentencia printf reporte valores diferentes para la variable num en el hijo y en el padre?

Dado el siguiente código:

```
#include<stdio.h>
#include<unistd.h>
main() {
    printf("Hola ");
    fork();
    printf("Mundo");
    fork();
    printf("!");
}
```

20. Sin ejecutarlo dibuje la jerarquía de procesos del programa y determine cuál es la posible salida en pantalla.

21. 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);)

22. 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.

## Señalización, sleep, getppid

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
main() {
    pid_t pid;
    int status;
    printf("PADRE: Mi PID es %d\n", getpid());
    printf("PADRE: El PID de mi padre es %d\n", getppid());
    pid = fork();
    if(pid == 0){
        sleep(5);
        printf("HIJO: Mi PID es %d\n", getpid());
        printf("HIJO: El PID de mi padre es %d\n", getppid());
        printf("HIJO: Fin!!\n");
    }
    else {
        printf("PADRE: Mi PID es %d\n", getpid());
        printf("PADRE: El PID de mi hijo es %d\n", pid);
        // wait(&status);
        // printf("PADRE: Mi hijo ha finalizado con estado %d\n", status);
        printf("PADRE: Fin!!\n");
    }
    exit(0);
}
```

23. ¿Cuál es la principal función de sleep en el código anterior?

24. ¿Quién es el padre del parent? Use este comando:

```
ps alf
```

25. ¿Por qué el proceso hijo imprime el id del parent como 1? ¿Es el que usted espera de acuerdo a la jerarquía de procesos?

26. Retire el comentario de las líneas de la función wait y la siguiente función printf. ¿Cuál es el identificador del parent ahora? ¿Para qué sirve la función wait? ¿Qué retorna en status?

## Proceso Zombie

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    pid_t pid;
    char *message;
    int n;
    printf("Llamado a fork\n");
    pid = fork();
    switch(pid) {
        case -1:
            perror("fork falló");
            exit(1);
        case 0:
            message = "Este es el hijo";
            n = 1;
            break;
        default:
```

```

        message = "Este es el padre";
        n = 30;
        break;
    }
    for(; n > 0; n--) {
        printf("n=%d ",n);
        puts(message);
        sleep(1);
    }
    exit(0);
}

```

Cuando un proceso hijo termina, su asociación con el padre continua 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.

**27. 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?**

**28. ¿Qué sucede si el proceso padre termina de manera anormal?**

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

Programa 1:

```

#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
    int pid;
    if ((pid = fork()) == 0) {
        execl("/bin/ls", "ls", "/", 0);
    }
    else {
        wait(&pid);
        printf("exec finalizado\n");
    }
}

```

Programa 2:

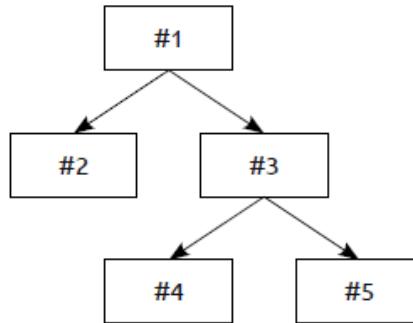
```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Corriendo ps con execl\n");
    execl("ps", "ps", "-ax", 0);
    printf("Echo.\n");
    exit(0);
}

```

**29. ¿Qué es lo que hace cada uno de los programas anteriormente mostrados?**

30. 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 parent.
- Imprimir el id del proceso parent del proceso 1.
- A través de la función system imprimir el árbol del proceso y verificar la jerarquía (pstree).

31. 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):

```
HIJO1: fact(1) = 1
HIJO2: fact(2) = 1
HIJO2: fact(2) = 2
HIJO1: fact(2) = 2
```

El proceso parent tiene que esperar a que los hijos terminen.

32. 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.

Dado el siguiente fragmento de código:

```
#include<stdio.h>
#include<error.h>
#include<stdlib.h>
#include<fcntl.h>
int main(int argc, char *argv[]) {
    int fd;
    int pid;
    char ch1, ch2;
    fd = open("data.txt", O_RDWR);
    read(fd, &ch1, 1);
    printf("En el padre: ch1 = %c\n", ch1);
    if ((pid = fork()) < 0) {
        perror("fork falló");
        exit(-1); //Sale con código de error
    } else if (pid == 0) {
        read(fd, &ch2, 1);
        printf("En el hijo: ch2 = %c\n", ch2);
    } else {
        read(fd, &ch1, 1);
        printf("En el padre: ch1 = %c\n", ch1);
    }
    return 0;
}
```

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

hola

33. 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?

## 4 Referencias

---

- [1] <http://pubs.opengroup.org/onlinepubs/00095399/functions/exec.html>
- [2] <http://siber.cankaya.edu.tr/ozdogan/OperatingSystems/spring2010/index.html>
- [3] <http://www.eg.bucknell.edu/~cs315/Spring11/lab/>
- [4] [http://www.makelinux.net/kernel\\_map/](http://www.makelinux.net/kernel_map/)
- [5] <http://www.eg.bucknell.edu/~cs315/Spring11/lab/>