

SESIÓN N° 07

PROCESOS Y SEÑALES

OBJETIVOS

- Elaborar programas que hacen uso de múltiples procesos y manejan señales con el fin de crear aplicaciones de funcionalidad robusta que realizan más de una tarea.

TEMAS A TRATAR

- Procesos
- Gestión de Procesos
- Prioridad de los Procesos
- Señales
- Procesos Zombi

MARCO TEORICO

Procesos

Un proceso es una instancia en ejecución de un programa. Si Ud. está ejecutando dos terminales actualmente, entonces está ejecutando dos procesos de un mismo programa (el shell).

IDs de procesos

Cada proceso en Linux es identificado por su ID único numérico, llamado **pid**. Estos **pid** son asignados secuencialmente a medida que se crean nuevos procesos. Cada proceso también tiene un proceso padre (a excepción del proceso especial **init**), resultando en una estructura de árbol, con **init** como raíz. El **ID** único del padre se llama **ppid**. Abra su editor de textos favorito y escriba el código a continuación:

```
//imprimir-pid.c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf ("El id del proceso es %d\n", (int) getpid());
    printf ("El pid del padre (ppid) es %d\n", (int) getppid());
    return 0;
}
```

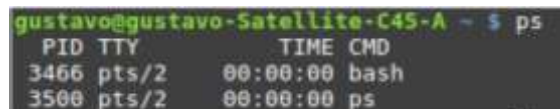
Para compilar y ejecutar el código, diríjase a la carpeta donde esta su código fuente y ejecute:

```
$ cc imprimir-pid.c -o imprimir-pid
```

Viendo los procesos activos

El comando **ps** muestra los procesos en ejecución del sistema. La versión de GNU/Linux tiene muchas opciones por cuestiones de compatibilidad con otras versiones de UNIX. Por defecto, invocar **ps** nos muestra los procesos controlados por la terminal actual.

Pruebe en su terminal.



```
gustavo@gustavo-Satellite-C45-A ~ $ ps
  PID TTY          TIME CMD
 3466 pts/2        00:00:00 bash
 3500 pts/2        00:00:00 ps
```

El primero es **bash**, el shell de la misma terminal. El segundo es la instancia en ejecución de **ps**. Para una versión más detallada, pruebe el siguiente comando:

```
$ ps -e -o pid, ppid, command
```

Revise la documentación de **ps** para averiguar qué hace específicamente cada parámetro. Adicionalmente, busque a la instancia de **ps** que ha ejecutado dentro de la salida y verifique quién es su padre.

Matando a un Proceso

Para matar a un proceso usamos el comando **kill**, seguido del **pid** del proceso. Este comando funciona al enviar una señal de terminación al proceso, conocida como **SIGTERM**. Esto hace que el proceso termine, a menos que este último maneje explícitamente a la señal **SIGTERM** (como veremos más adelante).

Creando Procesos

Usando system

La función **system** de la librería estándar es la manera más fácil de ejecutar un comando dentro de un programa, como si hubiera sido tipeado directamente en el **shell**. De hecho, **system** crea un subproceso que ejecuta un **shell** y le pasa el comando indicado:

```
//system.c
#include <stdlib.h>
int main () {
    int valor_retorno;
    valor_retorno = system ("ls -lh /");
    return valor_retorno;
}
```

Como la función **system** depende del shell para su funcionamiento no es recomendado usarla, ya que la versión y opciones disponibles al shell varían de distribución a distribución de UNIX o Linux. Invocar un programa con privilegios de **root** puede causar resultados diferentes en distintos sistemas. Por eso se recomienda el método a continuación.

Usando fork y exec

Linux nos brinda la función **fork**, la cual crea un proceso hijo que es una copia exacta de su padre. Luego tenemos otro grupo de funciones: la familia **exec**, la cual puede hacer que una instancia de un programa se vuelva la instancia de otro.

Entonces, el flujo es: usar **fork** para crear una copia del proceso actual y luego usar **exec** para transformar dicha copia a una instancia del proceso que queremos crear.

Llamando a fork

Cuando un programa llama a **fork** se crea un proceso hijo o duplicado. El padre continúa la ejecución desde el punto en el que se llamó a **fork**. El hijo comienza su ejecución en el mismo punto.

Como el hijo es un nuevo proceso, tiene su propio **pid**. Una manera de distinguir entre el padre y el hijo es la función **getpid**. Otra manera es usar el resultado de llamar a **fork**, el cual es 0 cuando nos encontramos en el hijo y diferente de cero cuando estamos en el padre. Esto funciona ya que no puede existir un proceso con ID cero.

Examine el programa a continuación:

```
//fork.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t pid_hijo;
    printf("El pid del programa principal es %d\n", (int) getpid());
    pid_hijo = fork();
    if(pid_hijo!=0)
    {
        printf("Este es el proceso padre con ID %d\n", (int) getpid());
        printf("El ID del hijo es %d\n", (int) pid_hijo);
    }
    else
        printf("Este es el proceso hijo, con ID %d\n", (int) getpid());
}
```

Usando a la familia exec

Las funciones **exec** reemplazan al programa de un proceso con otro. Por este motivo, **exec** no retorna ningún valor, a menos que haya ocurrido un error. Las funciones de esta familia varían dependiendo de cómo son llamadas:

- Las funciones con la letra **p** en sus nombres (**execvp** y **execlp**) aceptan el nombre de un programa, el cual debe encontrarse en la ruta actual de ejecución; las funciones que no tengan **p** en el nombre deben recibir la ruta completa del programa.

- Las funciones con la letra **v** en sus nombres (**execv**, **execvp** y **execve**) aceptan la lista de argumentos para el nuevo programa en la forma de un arreglo de punteros a cadenas terminado en **NULL**. Las funciones con la letra **l** (**execl**, **execlp** y **execle**) acepta la lista de argumentos usando el mecanismo *varargs* de C.
- Las funciones con la letra **e** en sus nombres (**execve** y **execle**) aceptan un argumento adicional, un arreglo de variables de entorno. Este también debería ser un arreglo de punteros a cadenas terminado en **NULL**. Cada cadena debe ser de la forma *"VARIABLE=valor"*.

La lista de argumentos que recibe el programa es como los argumentos que especificamos cuando ejecutamos un comando en el shell. Esta lista está disponible a través de los parámetros *argc* y *argv* de la función **main**.

Ojo: Cuando ejecutamos el programa desde el shell, el nombre del programa es el elemento *argv[0]*, el primer argumento es *argv[1]* y así sucesivamente. Cuando usemos **exec** también debemos pasar el nombre del programa como primer argumento.

Usando fork y exec juntos

La modalidad común de crear un subproceso consiste en primero realizar el **fork** y luego el **exec** al subprograma. Esto permite que el padre continúe su ejecución mientras el proceso hijo está siendo reemplazado.

El programa a continuación hace lo mismo que el programa **system.c** que vimos arriba, con la diferencia de que ya no depende del shell en la creación del subproceso:

```
//forx-exec.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int crear(char* programa, char** argumentos)
{
    pid_t pid_hijo;
    pid_hijo = fork();
    if(pid_hijo!=0)
    {
        //estamos en el padre
        return pid_hijo;
    }
    else
    {
        //estamos en el hijo
        execvp(programa, argumentos);
        //execvp SOLO retorna si hay ERROR
        fprintf(stderr, "se dio un error");
        abort();
    }
}

int main()
{
    //la lista de argumentos a pasarle a nuestra función
```

```
char* argumentos[] = {  
    "ls", //argv[0], el nombre del programa  
    "-lh",  
    "/",  
    NULL //la lista debe terminar en NULL  
};  
//creamos un proceso hijo que ejecuta el comando "ls"  
//ignoramos el pid retornado  
crear("ls", argumentos);  
return 0;  
}
```

Prioridad de procesos

Linux programa los procesos padre e hijo independientemente; no podemos saber cuánto tiempo se ejecutará un proceso antes que Linux lo interrumpa y lo ponga en segundo plano. De hecho, el comando **ls** de nuestro ejemplo anterior podría correrse antes que el padre termine su ejecución.

Es aquí donde entra el concepto de prioridades: hay procesos que simplemente son más importantes que otros. Esta importancia está definida por su valor **nice** o su **nice**ness.

Por defecto todos los procesos tienen un **nice**ness de cero, un valor alto indica menos prioridad y viceversa. La prioridad más baja entonces está dada por un **nice**ness de 20, mientras que la más alta, por -20.

Para ejecutar un programa con un **nice**ness específico usamos el comando **nice**, especificando el valor con la opción **-n**:

```
$nice -n -20 firefox
```

Puede verificar el **nice**ness de los procesos activos mediante el comando **ps**.

También puede usar el comando **renice** para cambiar el **nice**ness de un proceso en ejecución. Es posible que requiera privilegios de **root** para poder hacer esto. Revise la documentación de este programa.

Señales

Las señales son mecanismos de comunicación entre procesos en Linux. Es un tema amplio así que veremos algunas de las señales más importantes y técnicas usadas para controlar procesos.

Una señal es un mensaje especial enviado a un proceso. Las señales son asíncronas; cuando un proceso las recibe, debe procesarlas inmediatamente, así no termine la función o línea de código actual. Explicación no técnica de comunicación síncrona y asíncrona:

<http://www.fernandoplaza.com/2009/03/comunicacion-sincrona-vs-comunicacionasincrona.asp>

Cada vez que hacemos **Ctrl+C** o **Ctrl+D** durante la ejecución de un proceso en el Shell estamos invocando a una señal (de terminación).

Existe una docena de señales, cada una con diferente significado. Pueden ser identificadas por su número o por su nombre, ambos detallados en `/usr/include/bits/signum.h`. Échele un vistazo al archivo.

Cuando un proceso recibe una señal, su comportamiento depende de su disposición. Cada señal tiene una disposición por defecto, la cual determina qué pasa si el programa no indica algún comportamiento: La señal puede ser **ignorada** o **enmascarada**. **Enmascarar** o **manejar** una señal consiste en especificar un comportamiento explícito mediante alguna función manejadora (*signal handler*).

Un uso de las señales es para mandar algún comando a un proceso. Para este propósito existen dos señales reservadas para el usuario: **SIGUSR1** y **SIGUSR2**.

Para indicar la disposición hacia una señal hacemos uso de la función **sigaction**. El primer argumento es el número o nombre de la señal. El segundo es un puntero a una estructura **sigaction**, el cual indica la disposición deseada para la señal indicada. El tercer argumento indica la disposición anterior. El campo más importante de la estructura **sigaction** es **sa_handler**. Puede tener uno de estos tres valores:

- **SIG_DFL**, que especifica la disposición por defecto
- **SIG_IGN**, que especifica que debería ser ignorada
- Un puntero a una función manejadora. La función debe tener un solo argumento, el número de señal y retornar **void**.

Como las señales son asíncronas, el programa principal puede estar en un estado frágil cuando la señal es procesada. Por esto, no se recomienda manejar operaciones de E/S desde un manejador. Asimismo, un manejador debería realizar el **trabajo mínimo requerido** para responder, y luego retornar el control al programa. Muchas veces esto consiste en simplemente registrar que se dio la señal. Luego, nuestro programa revisaría periódicamente este registro para actuar apropiadamente.

OJO: Depurar y diagnosticar errores dentro de manejadores de señales es difícil, por eso debemos ser tacaños con la cantidad de código que incluimos en ellos.

El mismo hecho de modificar una variable global es peligroso, ya que durante esta modificación otra señal podría intentar modificarla al mismo tiempo, dejando a la variable en un estado **corrupto**. Una manera de evitar esto es usar el tipo especial **sig_atomic_t** que está diseñado para que Linux SIEMPRE realice cualquier asignación en una sola operación.

El programa *sigusr1.c* usa un manejador para contar el número de veces que se da la señal **SIGUSR1**.

```
//sigusr1.c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
sig_atomic_t sigusr1_contador = 0;

void manejador(int nro_senal)
{
    ++sigusr1_contador;
    printf("SIGUSR1 se dio %d veces\n", sigusr1_contador);
}
```

```
int main()
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &manejador;
    sigaction(SIGUSR1, &sa, NULL);
    //por el bucle infinito el programa debe ser
    //abortado mediante SIGKILL o SIGTERM
    while(1);
    return 0;
}
```

Al ejecutar el programa tendrá una pantalla negra. Para mandarle la señal que espera usaremos el comando **kill**. Abra un shell nuevo y escriba:

```
$kill -l
```

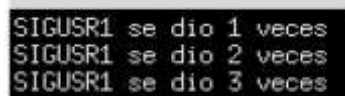
Esto nos mostrará las señales con sus respectivos números. **SIGUSR1** es la señal 9. Sin embargo, para usar **kill** necesitamos el ID del proceso. Una manera de hacer esto es usar el comando **pgrep** que realiza una búsqueda como *grep* pero sobre los nombres de los procesos activos. Como nuestro programa se llama *sigusr1.c*:

```
$pgrep sigusr1
```

Esto nos dará el **pid** de nuestro programa en ejecución. Suponiendo que el *pid* es 1234, para enviar la señal **SIGUSR1** a este proceso entonces:

```
$kill -10 1234
```

Y deberíamos ver una respuesta por parte de nuestro programa:



```
SIGUSR1 se dio 1 veces
SIGUSR1 se dio 2 veces
SIGUSR1 se dio 3 veces
```

Terminación de procesos

Si queremos terminar a nuestro programa podemos usar **kill** sin ningún argumento:

```
$kill 1234
```

Lo cual envía la señal SIGTERM. Otra manera de terminarlo es usar la señal SIGKILL:

```
$kill -SIGKILL 1234
```

¿Cuál es la diferencia? **SIGTERM** le pide al proceso que termine; esta petición puede ser ignorada. **SIGKILL** siempre mata al proceso; es una de las señales no manejables.

Si quisiéramos enviar una señal dentro de nuestros programas podemos utilizar la función **kill**, cuya sintaxis es:

```
kill (pid_hijo, SEÑAL)
```

Esperando a que un proceso termine

En el programa *fork-exec.c* podrá haber visto que la salida de **ls** suele aparecer después de que el programa principal ha finalizado. Esto es porque Linux es un SO multitarea y como tal, los procesos son programados independientemente de sus padres.

Sin embargo, existen ocasiones en las cuales nos gustaría que el padre espera hasta que uno o más de sus hijos terminen su ejecución para continuar.

Las llamadas de sistema wait

La función **wait** simplemente bloquea el proceso invocante hasta que sus procesos hijos terminen (u ocurra un error). A continuación, tenemos el ejemplo **fork-exec** modificado para que ahora el padre espere a que **ls** finalice antes de continuar:

```
//forx-exec-wait.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int crear(char* programa, char** argumentos)
{
    pid_t pid_hijo;
    pid_hijo = fork();
    if(pid_hijo!=0)
    {
        //estamos en el padre
        return pid_hijo;
    }
    else
    {
        //estamos en el hijo
        execvp(programa, argumentos);
        //execvp SOLO retorna si hay ERROR
        fprintf(stderr, "se dio un error\n");
        abort();
    }
}
int main()
{
    int estado_hijo;
    //la lista de argumentos a pasarle a nuestra función
    char* argumentos[] = { "ls", "-lh", "/", NULL };
    //creamos un proceso hijo que ejecuta el comando "ls"

    //ignoramos el pid retornado
    crear("ls", argumentos);
    //esperamos a que el hijo termine
    wait(&estado_hijo);
    //WIFEXITED indica si finalizó correctamente el hijo
```



```
if (WIFEXITED(estado_hijo))
    printf("El proceso hijo finalizó correctamente, con código %d\n",
           WEXITSTATUS(estado_hijo));
else
    printf("El proceso hijo finalizó anormalmente\n");
return 0;
}
```

Procesos Zombi

Cuando un proceso hijo termina mientras su padre usa **wait**, el hijo desaparece y le pasa su estado de finalización al padre. Pero ¿qué pasa si es que el padre no usa **wait**? No desaparece, porque si no se perdería toda su información. En vez de eso se vuelve zombi.

Un proceso zombi es aquel proceso que ha terminado, pero no ha sido “**limpiado**”. Esa es la responsabilidad de su padre. La función **wait** ya se encarga de esto, así el hijo haya finalizado antes de la llamada a esta función.

¿Qué ocurre si el padre no limpia a sus hijos? Lo podemos observar con el siguiente ejemplo:

```
//zombi.c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t pid_hijo;
    pid_hijo = fork();
    if(pid_hijo > 0){
        //proceso padre. dormimos por un minuto
        sleep(60);
    }
    else{
        //proceso hijo. finalizamos inmediatamente
        exit(0);
    }
    return 0;
}
```

Después de ejecutar el ejemplo de arriba, inmediatamente abra un shell y escriba:

```
$ps -e -o pid,ppid,stat,cmd | grep zombi
```

Veremos que aparte del proceso zombi padre existe un proceso hijo (revisar el *ppid*) el cual está marcado como **<defunct>**, ósea zombi. Una vez que el padre termine de dormir limpiará al hijo. Verifique esto.

01. Modifique el ejemplo **fork.c** para que el hijo a su vez cree otro proceso hijo (nieto). Su programa debería mostrar los mensajes apropiados con los **pid** de cada proceso.
02. Escriba un programa que maneje dos señales: **SIGTERM** y **SIGUSR2**. Cada vez que el programa reciba la señal **SIGTERM** deberá mostrar un mensaje indicando el número de veces que se dio. Adicionalmente, cada vez que su programa reciba la señal **SIGUSR2** deberá invocar al programa **cowsay** como proceso hijo. Pruebe el siguiente comando:

```
$cowsay hola
```

El comando tiene varias opciones y otros animales, elija la combinación de argumentos que desee. No le está permitido usar la función **system**, deberá usar **fork** y **exec**. El programa debe ejecutarse en un bucle infinito, similar a **sigusr1.c**

V

REFERENCIAS

- Mitchell, M., Oldham, J., & Samuel, A. (2001). Processes. Advanced linux programming.(pp. 45-60) New Riders Publishing.
- Ellingwood, J. (2013). How To Use ps, kill, and nice to Manage Processes in Linux. Recuperado a partir de <https://www.digitalocean.com/community/tutorials/how-to-use-pskill-and-nice-to-manage-processes-in-linux>