

PRACTICA # 3

“PROCESOS” (CONTINUACION)

1. OBJETIVO

Aprender cómo manipular procesos por medio de las funciones *wait()* y *kill()*; además del uso de señales del sistema Linux.

2. INTRODUCCIÓN

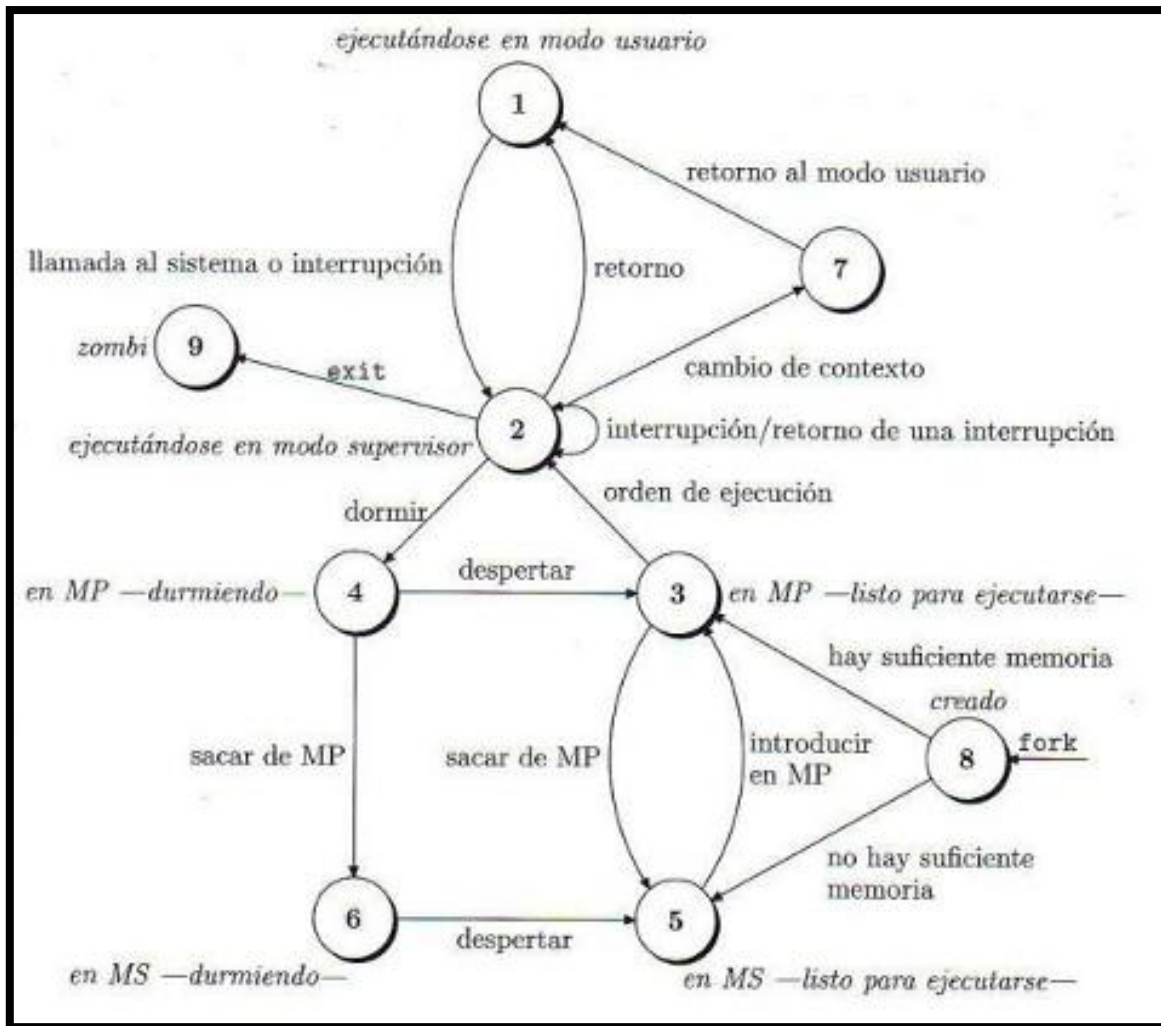
Estados de un proceso

Un proceso puede ser ejecutado en modo usuario o por el núcleo o supervisor. El tiempo de vida de un proceso se puede dividir en un conjunto de estados, cada uno con unas características determinadas.

Un proceso no permanece siempre en un mismo estado, sino que está continuamente cambiando de acuerdo con unas reglas bien definidas. Estos cambios de estado vienen impuestos por la competencia que existe entre los procesos para compartir un recurso escaso como es la CPU.

Los posibles estados de un proceso son:

1. Ejecución en modo de usuario.
2. Ejecución en modo del núcleo.
3. Listo para ejecutar y en memoria, listo para ejecutar tan pronto como el núcleo lo planifique.
4. Dormido y en memoria no dispuesto para ejecutar hasta que se produzca un suceso; el proceso está en memoria principal.
5. Listo para ejecutar y descargado, el proceso está listo para ejecutar, pero se debe cargar el proceso en memoria principal antes de que el núcleo pueda planificarlo para su ejecución.
6. Dormido y descargado, el proceso está esperando un suceso y ha sido expulsado al almacenamiento secundario.
7. Expulsado, el proceso retorna del modo núcleo al modo usuario pero el núcleo lo expulsa y realiza un cambio de contexto para planificar otro proceso.
8. Creado, el proceso está recién creado y aún no está listo para ejecutarse.
9. Zombie, el proceso ya no existe, pero deja un registro para que lo recoja el proceso padre.



El contexto de un proceso son: su código, los valores de sus variables de usuario globales y sus estructuras de datos, el valor de los registros de la CPU, los valores almacenados en su entrada de la tabla de procesos y en su área de usuario, y el valor de sus pilas de usuario y núcleo.

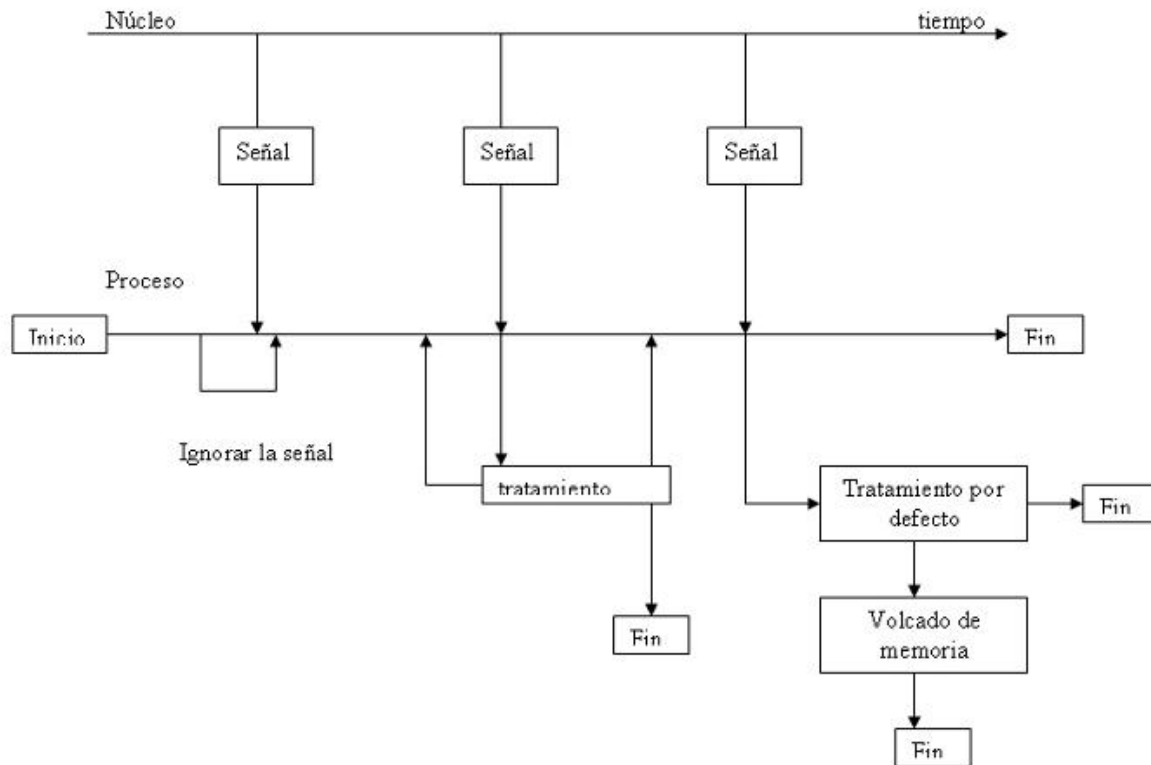
Señales

Las señales son un mecanismo para comunicación y manipulación de procesos en Linux. Una señal es un mensaje especial que se manda a un proceso. Las señales son asíncronas, cuando un proceso recibe una señal, éste procesa la señal inmediatamente, sin terminar la función o línea de código que se está ejecutando. Cada tipo de señal es especificada por un número, pero en los programas se hace referencia a través de un nombre. Para hacer uso de las señales se tiene que agregar la cabecera *<signal.h>*.

En Linux, el sistema manda señales a los procesos en respuesta a condiciones específicas. Por ejemplo, *SIGBUS* (error en el bus), *SIGSEGV* (violación de segmento) y *SIGFPE* (excepción de punto flotante) pueden ser enviadas a los procesos cuando se trata de realizar una operación ilegal. Lo que hacen estas señales es terminar el proceso y producir un archivo *core*.

Un proceso también puede mandar una señal a otro proceso. Por ejemplo, las señales *SIGTERM* y *SIGKILL* es un mecanismo para terminar un proceso desde otro proceso. La señal *SIGHUP* es usada para despertar un proceso.

Un manejador de señales debe ejecutar el mínimo trabajo necesario para responder a una señal y luego regresar el control al programa principal o terminar el programa. Es posible que un manejador de señales sea interrumpido por otra señal.



Las señales se clasifican en los siguientes grupos:

- ✓ Señales relacionadas con la terminación de procesos.
- ✓ Señales relacionadas con las excepciones inducidas por los procesos. Ejemplos: el intento de acceder fuera del espacio de direcciones virtuales, los errores producidos al manejar números de punto flotante, etc.
- ✓ Señales relacionadas con los errores irre recuperables originados en el transcurso de una llamada al sistema.
- ✓ Señales originadas desde un proceso que se está ejecutando en modo usuario. Ejemplos: cuando un proceso envía una señal a otro vía kill, cuando un proceso activa un temporizador y se queda en espera de la señal de alarma, etc.
- ✓ Señales relacionadas con la interacción con el terminal. Ejemplo: pulsar las teclas Ctrl+C.
- ✓ Señales para ejecutar un proceso paso a paso. Son usadas por los depuradores. En el fichero de cabecera <signal.h> están definidas las señales que puede manejar el sistema.

A continuación se describen las 19 señales de UNIX System V.

SIGHUP (1) Hangup. Es enviada cuando un terminal desconecta de todo proceso del que es terminal control. También se envía todos los procesos de un grupo cuando

el líder del grupo termina su ejecución. La acción por defecto de este proceso es terminar la ejecución del proceso que la recibe.

- SIGINT (2) Interrupción. Se envía a todo proceso asociado con un terminal de control cuando se pulsa la tecla interrupción. Su acción por defecto es terminar la ejecución del proceso que la recibe.
- SIGQUIT (3) Salir. Similar a SIGINT, pero es generada al pulsar la tecla de salida (Control-\\). Su acción por defecto es generar un fichero core y terminar el proceso.
- SIGILL (4) Instrucción ilegal. Es enviada cuando el hardware detecta una instrucción ilegal. En los programas escritos en C suele producirse este tipo de error cuando manejamos punteros a funciones que no han sido correctamente inicializadas. Su acción por defecto es generar un fichero core y terminar el proceso.
- SIGTRAP (5) Trace trap. Es enviada después de ejecutar cada instrucción cuando el proceso se está ejecutando paso a paso. Su acción por defecto es generar un fichero core y terminar el proceso.
- SIGIOT (6) I/O trap instruction. Se envía cuando se da un fallo de hardware. La naturaleza de este fallo depende de la maquina. Es enviada cuando llamamos a la función abort, que provoca el suicidio del proceso generando un fichero core.
- SIGEMT (7) Emulator trap instruction. Tambien indica un fallo de hardware. Raras veces se utiliza. Su acción por defecto es generar un fichero core y terminar el proceso.
- SIGFPE (8) Error en coma flotante. Es enviada cuando el hardware detecta un error de coma flotante, como el uso de un número en coma flotante con un formato desconocido, errores de overflow o underflow, etc. Su acción por defecto es generar un fichero core y terminar el proceso.
- SIGKILL (9) Kill. Esta señal provoca irremediabilmente la terminación del proceso. No puede ser ignorada y siempre ejecuta su acción por defecto, que consiste en generar un fichero core y terminar el proceso.
- SIGBUS (10) Bus error. Se produce cuando se da un error de acceso a la memoria. Las dos situaciones típicas que la provocan suelen ser intentar acceder a una dirección que físicamente no existe o intentar acceder a una dirección impar, violando así las reglas de alineación que impone el hardware. Su acción por defecto es generar un fichero core y terminar el proceso.

SIGSEGV (11) Violación de segmento. Es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos. Su acción por defecto es generar un fichero core y terminar el proceso.

SIGSYS (12) Argumento erróneo en una llamada al sistema. No se usa.

SIGPIPE (13) Intento de escritura en una tubería en la que no hay nadie leyendo. Esto suele ocurrir cuando el proceso de lectura termina de forma anormal. Su acción por defecto es terminar el proceso.

SIGALRM (14) Alarm clock. Es enviada a un proceso cuando alguno de sus temporizadores descendientes llega a cero. Su acción por defecto es terminar el proceso.

SIGTERM (15) Finalización software. Es la señal utilizada para indicarle a un proceso que debe terminar su ejecución. Esta señal no es tajante como SIGKILL y puede ser ignorada. Lo correcto es que la rutina de tratamiento de esta señal se encargue de tomar las acciones necesarias antes de terminar un proceso (como, por ejemplo, borrar los archivos temporales) y llame a la rutina exit. Esta señal es enviada a todos los procesos durante el shutdown o para del sistema. Su acción por defecto es terminar el proceso.

SIGUSR1 (16) Señal número 1 de usuario. Esta señal está reservada para el uso del programador. Ninguna aplicación estándar va a utilizarla y su aplicación es la que le quiera dar el programador en su aplicación. Su acción por defecto es terminar el proceso.

SIGUSR2 (17) Señal número 2 de usuario. Su significado es idéntico al de SIGUSR1.

SIGCLD (18) Muerte del proceso hijo. Es enviada al proceso padre cuando alguno de sus procesos hijo termina. Esta señal es ignorada por defecto.

SIGPWR (19) Fallo de alimentación. Esta señal tiene diferentes interpretaciones. En algunos sistemas es enviada cuando se detecta un fallo de alimentación y le indica al proceso que dispone solo de unos instantes antes de que se produzca una caída del sistema. En otros sistemas, esta señal es enviada después de recuperarse de un fallo de alimentación, a todos aquellos procesos que estaban en ejecución y que se han podido rearrancar. En estos casos, los procesos deben disponer de mecanismos para restaurar las posibles pérdidas producidas durante la caída de la alimentación.

3. DESARROLLO

>> Ejemplo 1 - Espera de procesos

En algunas ocasiones, es deseable para un proceso padre esperar a que todos sus procesos hijos hayan sido completados. Esta acción puede ser ejecutada por la función *wait()*. Esta función permite esperar por un proceso a que termine de ejecutarse y permite que el proceso padre obtenga información de su proceso hijo.

La función *wait()* se encarga de bloquear el proceso padre hasta que el hijo termine su ejecución. Esta función regresa un código de estado por medio de un entero, de donde se puede obtener información de cómo terminó su ejecución el proceso hijo. Para esto existen las siguientes macros:

WIFEXITED devuelve verdadero —cualquier valor distinto de 0— cuando el proceso termina con una llamada a *exit()* o *wait()*.

WEXITSTATUS si **WIFEXITED** es verdadero, devuelve los 8 bits menos significativos que *exit()* le pasa al proceso padre.

WIFSIGNALED devuelve verdadero cuando el proceso termina debido a alguna señal.

WTERMSIG si **WIFSIGNALED** es verdadero, devuelve el número de la señal que ha causado la terminación del proceso.

WCOREDUMP devuelve verdadero si se ha generado un archivo con un volcado de la memoria del proceso.

WIFSTOPPED devuelve verdadero si el proceso está parado.

WSTOPSIG si **WIFSTOPPED** es verdadero, devuelve el número de la señal que ha causado la parada del proceso.

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

int spawn (char* program, char** arg_list)
{
    pid_t child_pid;
    /* Duplicate this process. */
    child_pid = fork ();

    if (child_pid != 0) /* This is the parent process. */
        return child_pid;
    else {
        /* Now execute PROGRAM, searching for it in the path. */
        execvp (program, arg_list);
        /* The execvp function returns only if an error occurs. */
        fprintf (stderr, "an error occurred in execvp\n");
        abort ();
    }
}

int main ()
{
    int child_status;

    /* The argument list to pass to the "ls" command. */
    char* arg_list[] = {
        "ls", /* argv[0], the name of the program. */
        "-l",
        "/",
        NULL /* The argument list must end with a NULL. */
    };

    /* Spawn a child process running the "ls" command. Ignore the
    returned child process ID. */
    spawn ("ls", arg_list);

    /* Wait for the child process to complete. */
    wait (&child_status);
    if (WIFEXITED (child_status))
        printf ("the child process exited normally, with exit code %d\n",
            WEXITSTATUS (child_status));
    else
        printf ("the child process exited abnormally\n");
    return 0;
}
```

Código 1 Uso de la función wait (fork-exec.c)


```

[root@vesta CLinux]# ./exe
total 152
drwxr-xr-x  2 root root  4096 Apr 26  2011 bin
drwxr-xr-x  3 root root  4096 Jan 24  2008 boot
drwxr-xr-x 10 root root 3720 Feb  2 13:56 dev
drwxr-xr-x 88 root root 12288 Feb 10 04:24 etc
drwxr-xr-x 16 root root  4096 Sep 15  2010 home
drwxr-xr-x 11 root root  4096 Aug  9  2006 lib
drwx----- 2 root root 16384 Jul  6  2006 lost+found
drwxr-xr-x  3 root root  4096 Feb 26  2008 media
drwxr-xr-x  2 root root  4096 Feb 10  2006 misc
drwxr-xr-x  5 root root  4096 Sep  8  2010 mnt
drwxr-xr-x  2 root root    0 Feb  2 13:55 net
drwxr-xr-x  6 root root  4096 Aug 21  2008 opt
dr-xr-xr-x 106 root root    0 Feb  2 07:53 proc
drwxr-xr-x 27 root root  4096 Feb  8 16:15 root
drwxr-xr-x  2 root root  4096 Aug 12  2006 sbin
drwxr-xr-x  2 root root  4096 Jul  6  2006 selinux
drwxr-xr-x  2 root root  4096 Feb 11  2006 srv
drwxr-xr-x 11 root root    0 Feb  2 07:53 sys
drwxrwxrwt  4 root root  4096 Feb 17 15:28 tmp
drwxr-xr-x 16 root root  4096 Mar 29  2007 usr
drwxr-xr-x 24 root root  4096 Aug 16  2006 var
the child process exited normally, with exit code 0

```

>>Ejemplo 2 – Procesos zombies

Un proceso zombi es un proceso que ha terminado su ejecución, pero que no ha sido eliminado del sistema. La responsabilidad de realizar esta acción es el proceso padre. La función `wait()` se encarga de realizar la limpieza de los procesos, cuando se hace una llamada de la función, esta revisa el estado de terminación del proceso hijo, si éste ya terminó su ejecución se elimina el proceso y el control regresa al proceso padre. Si el padre no termina el proceso, éste estará en el sistema como un proceso zombi.

```

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

int main ()
{
    pid_t child_pid;
    /* Create a child process. */
    child_pid = fork ();
    if (child_pid > 0) {
        /* This is the parent process. Sleep for a minute. */
        sleep (60);
    } else {
        /* This is the child process. Exit immediately. */
        exit (0);
    }
    return 0;
}

```

Código 2 Procesos zombies (zombie.c)

Al ejecutar el código anterior, el proceso padre se duerme durante 60 segundos, mientras que el proceso hijo termina su ejecución. Si ejecutamos el siguiente comando `ps -e -o pid,ppid,stat,cmd` —desde otra terminal para revisar los estados de los procesos—, tenemos que el proceso hijo con identificador 4427 ha terminado su ejecución y ahora se encuentra en estado zombi —denotado por Z+ y <defunct>—.

```
427 24465 S+   ./exe
428  4427 Z+   [exe] <defunct>
429 17667 R+   ps -e -o pid,ppid,stat,cmd
```

>> Ejemplo 3 - Terminación de procesos

Los procesos pueden terminar de dos maneras. La primera es cuando estos completan su tarea y regresan el control a su proceso padre. La segunda parte es cuando los procesos terminan anormalmente en respuesta a una señal. Por ejemplo, la señal SIGINT es enviada al proceso cuando un usuario oprime la combinación de teclas `Ctrl+C` en la terminal. Otra señal para terminar un proceso es SIGTERM, que es mandada por el comando `kill`. Cuando se hace la llamada a la función `abort()`, el proceso se manda a sí mismo la señal SIGABRT que termina el proceso y genera un archivo `core`. La señal más poderosa para terminar un proceso es SIGKILL, termina el proceso inmediatamente y no puede ser bloqueada o manejada por un programa.

Cualquiera de las señales antes mencionadas pueden ser usadas por el comando `kill` especificando una bandera de la siguiente manera: `kill -SEÑAL PID`, donde SEÑAL es cualquiera de las antes mencionadas y PID es el identificador del proceso a terminar. También se puede usar la función `kill()`, agregando las cabeceras `<sys/types.h>` y `<signal.h>` y usando la siguiente sintaxis:

`kill (child_pid, SIGTERM)`

Por convención, cuando el código de salida es cero, se considera que no hubo problemas en la ejecución; si regresa algún otro número indica que hubo un error. En Linux es posible obtener el código de salida del más reciente programa ejecutado por medio del comando `$?`. Como códigos de salida, se usa un número entre 0 y 127, cuando el código de salida es 128 o mayor significa que el proceso fue interrumpido por una señal.

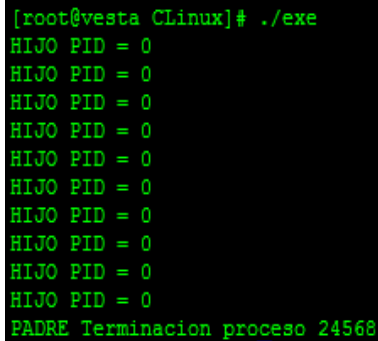
```
[root@vesta CLinux]# ls /
bin  dev  home  lost+found  misc  net  proc  sbin  srv  tmp  var
boot  etc  lib  media  mnt  opt  root  selinux  sys  usr
[root@vesta CLinux]# echo $?
0
[root@vesta CLinux]# ls otro
ls: otro: No such file or directory
[root@vesta CLinux]# echo $?
2
```

El siguiente código muestra como el hijo imprime PID hasta que el padre se despierta después de 10 segundos y termina el proceso hijo y así mismo por medio de la señal SIGTERM.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
    pid_t pid;
    if ((pid = fork()) == 0) {
        while(1) {
            printf("HIJO PID = %d\n", pid);
            sleep(1);
        }
    }

    sleep(10);
    printf("PADRE Terminacion proceso %d\n", pid);
    kill (pid,SIGTERM);
    exit(0);
}
```

Código 3 Uso de la funcion kill (kill.c)



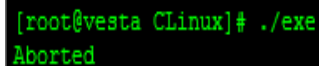
```
[root@vesta CLinux]# ./exe
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
HIJO PID = 0
PADRE Terminacion proceso 24568
```

El siguiente código muestra el uso de la función *abort()*.

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

int main(void)
{
    abort();
    exit(EXIT_SUCCESS);
}
```

Código 4 Invocando la función abort() (abort.c)



```
[root@vesta CLinux]# ./exe
Aborted
```

4. CUESTIONARIO

[1] Usando el código de *kill.c*, crear otro proceso hijo e indicar qué es lo que pasa con el programa.

[2] Modificar el proceso creado anterior para que después de 20 segundos termine su ejecución.

5. BIBLIOGRAFÍA

[1] Francisco Manuel Márquez García. “Unix: programación avanzada”. Editorial RA-MA

[2] Richard Stevens. “Advanced Unix programming”. Editorial Addison Wesley

[3] Kurt Wall. “Linux Programming by Example”. Editorial QUE

[4] <http://mermaja.act.uji.es/docencia/ii22/teoria/TraspasTema2.pdf>