# UNIX Shell Project

Sistemas Operativos

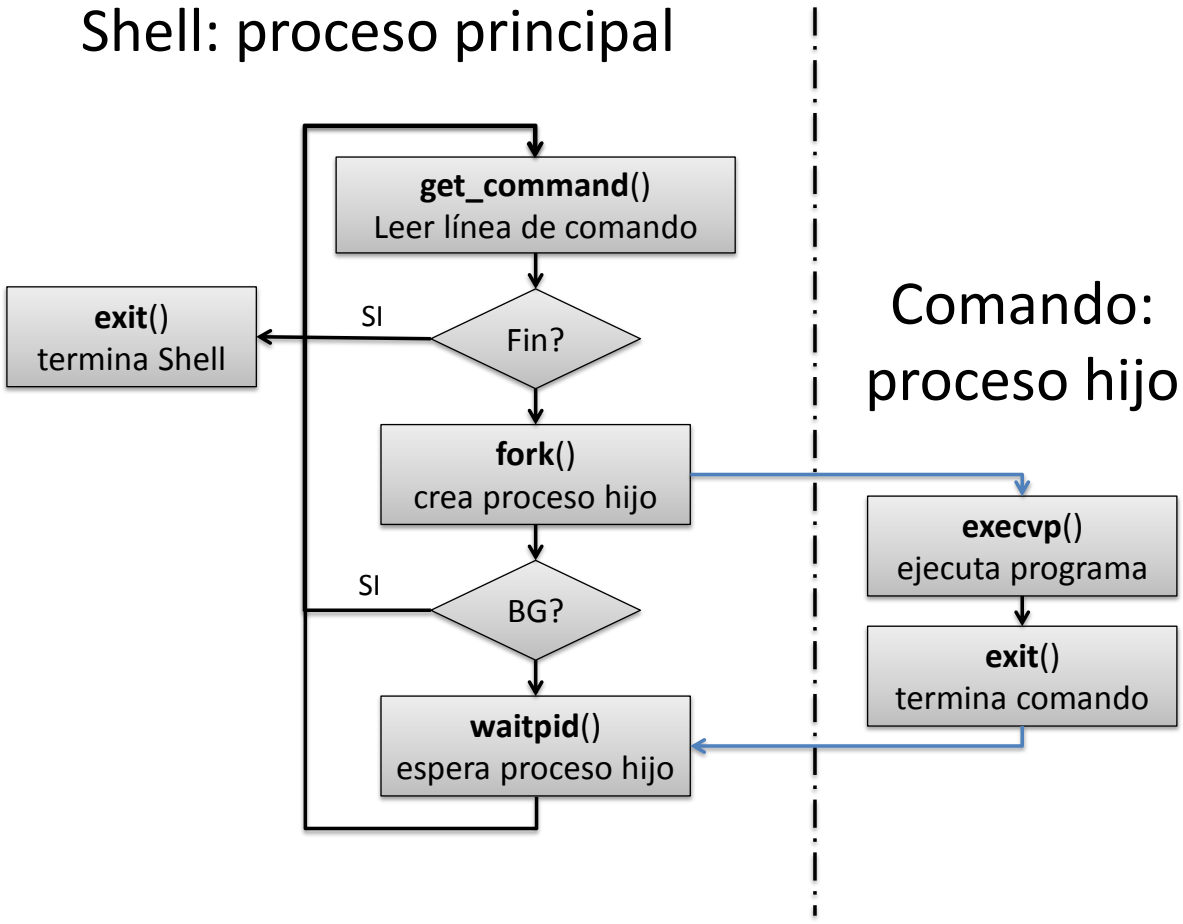
Grados I. Informática, Computadores y Software

Dept. Arquitectura de Computadores - UMA

El objetivo del Shell es servir como interprete de comandos, actúa como interfaz entre el usuario y el sistema. El usuario utilizará el Shell para lanzar sus programas y ejecutar comandos ofrecidos por el sistema operativo.

El Shell debe leer un comando tecleado por el usuario y crear un proceso que ejecute dicho comando. Para implementar este programa se debe hacer uso de las llamadas al sistema: **fork** (crea un proceso hijo), **execvp** (ejecuta un programa), **waitpid** (espera a la terminación de un proceso hijo) y **exit** (termina un proceso). Consultar la ayuda en línea de **linux** (**man**) para detalles sobre el funcionamiento de éstas y otras funciones.

Esquema simplificado de funcionamiento del *shell*:




En el código fuente C "`Shell_project.c`" disponible en el campus virtual está codificado el *shell* que vamos a usar como base para nuestra práctica. Este código de partida servirá para implementar un *shell* con control de tareas siguiendo los conceptos explicados por el profesor y que aparecen en el Anexo al final de este documento (inicialmente sólo será implementada la ejecución de tareas muy simplificada). Para la implementación deben usarse las funciones disponibles en el módulo "`job_control.c`".

A continuación se describen los tipos, funciones y macros más relevantes disponibles en el citado módulo (`job_control`) y que son de uso obligatorio:

Descripción de contenidos del módulo **`job_control`**

### get_command

```
void get_command(char inputBuffer[], int size, char *args[], int *background);
```

Devuelve en `args` el array de argumentos del comando leído, por ejemplo para el comando "`ls -l`", tendremos `args[0]="ls" , args[1]="-l"` y `args[2]=NULL`. La variable `background` se pondrá a 1 (true) si el comando termina con '&'. Necesita como entrada un buffer de caracteres y su tamaño para trabajar sobre él (`inputBuffer` y `size`). Hay un ejemplo de uso en `Shell_project.c`.

### status, job_state, status_strings, state_strings

```
enum status { SUSPENDED, SIGNALED, EXITED};
enum job_state { FOREGROUND, BACKGROUND, STOPPED };

static char* status_strings[] = { "Suspended","Signaled","Exited" };
static char* state_strings[] = { "Foreground","Background","Stopped" };
```

Estos son dos tipos enumerados para identificar la causa de terminación de una tarea y el estado actual de la misma respectivamente. Se han definido dos arrays de cadenas para permitir su impresión por pantalla de forma más directa, por ejemplo, siendo "`estado`" una variable de tipo `status` podríamos informar de su valor así:

```
printf("Motivo de terminación: %s\n", status_strings[estado]);
```

### analyze_status

```
enum status analyze_status(int status, int *info);
```

Esta función analiza el entero `status` devuelto por la llamada `waitpid` y devuelve la causa de terminación de una tarea (como un enumerado `status`) y la información adicional asociada a dicha terminación (`info`).

### job

```
typedef struct job_
{
      pid_t pgid; /* group id = process lider id */
      char * command; /* program name */
      enum job_state state;
      struct job_ *next; /* next job in the list */
} job;
```

El tipo `job` se usa para representar las tareas y enlazarlas en la lista de tareas. Una lista de tareas será un puntero a `job (job *)` y un nodo de la lista se manejará también como un puntero a `job (job *)`

Descripción de contenidos del módulo **`job_control`**

### `new_job`

```
job * new_job(pid_t pid, const char * command, enum job_state state);
```

Para crear una nueva tarea. Se proporciona como entrada a la función el pid, el nombre del comando y el estado (`BACKGROUND o FOREGROUND`).

### `new_list, list_size, empty_list, print_job_list`

```
job * new_list(const char * name);
int list_size(job * list);
int empty_list(job * list);
void print_job_list(job * list);
```

Estas funciones se utilizan para crear una nueva lista (con una cadena como nombre), para averiguar el tamaño de la lista, para comprobar si la lista está vacía (devuelve 1 si está vacía) y para pintar el contenido de una lista de tareas por pantalla. En realidad están implementadas como macros.

### `add_job, delete_job, get_item_bypid, get_item_bypos`

```
void add_job (job * list, job * item);
int delete_job(job * list, job * item);
job * get_item_bypid  (job * list, pid_t pid);
job * get_item_bypos( job * list, int n);
```

Estas funciones trabajan sobre una lista de tareas. Las dos primeras añaden o eliminan una tarea a la lista (`delete_job` devuelve 1 si la eliminó correctamente). Las dos últimas buscan y devuelven un elemento de la lista por `pid` o por posición. Devolverán `NULL` si no lo encuentran. Un ejemplo de uso de `add_job` podría ser:

```
job * my_job_list = new_list("Tareas Shell");
…
add_job( my_job_list, new_job(pid_fork, args[0], background) );
```

### `ignore_terminal_signals(), restore_terminal_signals()`

```
void ignore_terminal_signals()
void restore_terminal_signals()
```

Estas funciones (en realidad macros) se utilizan para desactivar o activar las señales relacionadas con el terminal (`SIGINT SIGQUIT SIGTSTP SIGTTIN SIGTTOU`) . El Shell debe ignorar estas señales, pero el comando creado con `fork`() debe restaurar su comportamiento por defecto.

### `new_process_group`

```
void new_process_group(pid_t pid)
```

Esta macro permite crear un nuevo grupo de procesos para el proceso recién creado. Debe

| Descripción de contenidos del módulo **`job_control`** |
|---|
| realizarlo el Shell después de hacer `fork`() |
| **`set_terminal`** |
| `void set_terminal(pid_t pid)`<br><br>Esta macro asigna el terminal al grupo de procesos identificados por `pid`. Debe cederse el terminal a las tareas de primer plano y recuperarse para el Shell cuando esta termina o se suspende. |
| **`block_SIGCHLD, unblock_SIGCHLD`** |
| `void block_SIGCHLD()`<br>`void unblock_SIGCHLD()`<br><br>Estas macros nos son de utilidad para bloquear la señal `SIGCHLD` en las secciones de código donde el Shell modifique o acceda a estructuras de datos (la lista de tareas) que pueden ser accedidas desde el manejador de esta señal y así evitar riesgos de acceso a datos en estados no válidos o no coherentes. |

A continuación se relacionan las tareas a realizar en este proyecto de laboratorio. Cada una de estas tareas debe de ser entregada en el campus virtual siguiendo las instrucciones del profesor/a para su evaluación. Para las tareas más complejas (la tarea final) habrá que realizar una entrevista o defensa personal de la misma con el profesor/a de la asignatura.

## *Tarea 1*

El código de partida que se suministra (Shell_project.c listado al final de este enunciado) entra en un bucle donde se leen y analizan líneas de comando desde el teclado. En esta primera etapa simplificaremos el problema y nos centraremos en la tarea esencial de crear un proceso y ejecutar un nuevo programa. Se debe añadir el código necesario al programa inicial para:

1. Ejecutar los comandos en un proceso independiente
2. esperar o no a la finalización del comando dependiendo de si el comando era de primer o de segundo plano (foreground/background)
3. dar un mensaje que informe de la terminación o no del comando y de sus datos identificativos
4. continuar con el bucle para tratar el siguiente comando

En esta tarea será necesario usar las llamadas al sistema fork, execvp, waitpid y exit. Los mensajes con información sobre los comandos ejecutados deben presentar la siguiente información para procesos en primer plano: pid, nombre, tipo de terminación y código de terminación o señal responsable. Por ejemplo:

```
Foreground pid: 5615, command: ls, Exited, info: 0
Foreground pid: 5616, command: vi, Suspended, info: 20
```

Para los comandos de segundo plano habrá que informar de su pid, nombre e indicar que se continúan ejecutando en segundo plano mientras el Shell continua simultáneamente con la ejecución de nuevos comandos. Por ejemplo:

```
Background job running... pid: 5622, command: sleep
```

Para comandos no encontrados, que no se pueden ejecutar, hay que dar el mensaje de error correspondiente:
`Error, command not found: lss`

Para compilar el programa usar:
`gcc Shell_project.c job_control.c –o Shell`
Para ejecutarlo:
`./Shell`
Para salir pulsar CONTROL + D (^D).

## *Tarea 2*

En un Shell hay que diferenciar entre los comandos o aplicaciones externas que se pueden ejecutar y los comandos internos que implementa y ofrece el propio Shell. Nuestro Shell hasta ahora no ofrece comandos internos. Implementar el comando interno "cd" para permitir cambiar el directorio de trabajo actual (usar la función `chdir`).

Cada comando externo que ejecute el Shell debe estar en su propio grupo de procesos independiente, para que el terminal pueda asignarse a una sola tarea en cada momento (la de primer plano). Por tanto se asignarán a los **procesos hijos del shell un id de grupo de procesos (gpid) diferente al del padre** usando la función `new_process_group` (macro que envuelve a `setpgid`).

En el caso de que un **proceso hijo** se ejecute en primer plano, éste debe ser el único **dueño del terminal,** ya que el padre quedará bloqueado en el `waitpid`. El terminal debe ser cedido y recuperado por el Shell usando la función `set_terminal` (macro que envuelve a `tcsetpgrp`). Si la tarea se envía a segundo plano **no se le debe asociar el terminal** ya que es el Shell el que debe seguir leyendo del teclado (el Shell queda en primer plano).

Para que nuestro Shell funcione correctamente cediendo el terminal a las tareas de primer plano es necesario que al comienzo del programa se ignoren todas las señales relacionadas con el terminal mediante la función `ignore_terminal_signals()`. Cuando se crea un nuevo proceso y antes de ejecutar el comando se deben restablecer esas señales a su funcionamiento por defecto (dentro del proceso hijo) con la función `restore_terminal_signals()`.

Examinando el valor de la variable `status` devuelta por el `waitpid` se puede **averiguar si el proceso hijo ha terminado o está parado**. Para tener en cuenta también la suspensión de un hijo es necesario añadir la opción WUNTRACED a la función `waitpid`. Asegúrese de que los procesos de primer plano suspendidos son reportados por el Shell.

## *Tarea 3*

Si nos fijamos los comandos enviados a segundo plano que son desatendidos por el Shell quedan en estado Zombie al terminar (aparecen marcados con `<defunct>` al usar el comando ps).

Vamos a instalar un manejador a la señal SIGCHLD para tratar la terminación/suspensión de las tareas en segundo plano de forma asíncrona (usando la función `signal`). Para ello crearemos una lista de tareas (usando `job_control`) y añadiremos a esta lista cada comando lanzado en segundo plano.

Al activarse el manejador de la señal SIGCHLD habrá que revisar cada entrada de la lista para comprobar si algún proceso en segundo plano ha terminado o se ha suspendido. Para comprobar si un proceso ha terminado sin bloquear al Shell hay que añadir la opción WNOHANG en la función `waitpid`.

En este ejercicio debemos conseguir que las tareas de segundo plano no queden `zombies` y que el Shell notifique su terminación/suspensión. Cuando una tarea se termina o se suspende hay que actualizar la lista de tareas de forma adecuada.

## *Tarea 4*

Esta tarea, además de entregarse en el campus virtual, deberá defenderse en entrevista personal con el profesor/a de la asignatura.

Se debe implementar de forma segura **la gestión de tareas en segundo plano o *background*** (las tareas lanzadas con '&'). Si la tarea se envía a segundo plano **no se le debe asociar el terminal** ya que es el Shell el que debe seguir leyendo del teclado (el shell queda en primer plano). Añadir la lista para el control de tareas (usar los tipos y funciones disponibles en el módulo `job_control`)**. Se deben controlar en la lista varias tareas ejecutándose en segundo plano, así como varias tareas suspendidas. Las tareas suspendidas pueden proceder de tareas en primer plano suspendidas (ctrl.+Z) o tareas en segundo plano que fueron suspendidas tras intentar leer del terminal (la señal SIGSTOP también suspenderá a cualquier tarea de primer o segundo plano)**. El código que estamos diseñando es reentrante (mientras se está ejecutando el código del Shell puede llegar la señal SIGCHLD y lanzar el manejador). Para evitar problemas de coherencia en el acceso a la lista de procesos, se debe bloquear esta señal cuando se vaya a acceder a las regiones de código que manejan o actualizan la lista de procesos. En cualquier parte del código del Shell donde se acceda a la lista de procesos de sebe bloquear la señal usando las funciones disponibles en `job_control` (`block_SIGCHLD, unblock_SIGCHLD`)

El shell debe **incluir los siguientes comandos internos**:

**Implementar el comando interno `jobs`,** que imprima una lista de tareas en segundo plano y suspendidas, o un mensaje si no hay ninguna. La lista de jobs debe incluir tanto la lista de tareas enviadas de forma explícita a segundo plano (con '&'), como las tareas que se encuentren suspendidas (paradas). Ya existe una función en `job_control` para imprimir una lista de tareas.

**Implementar el comando interno fg,** para, trabajando con la lista de tareas**, poner una tareas de las que está en segundo plano o suspendida en primer plano**. El argumento de dicho comando debe ser el identificador del lugar que ocupa esa tarea en la lista (1, 2, ...), si no se indica argumento deberá aplicarse a la primera tarea de la lista (la última que ingreso a la lista). Es importante que las señales se envíen al grupo de procesos completo usando `killpg`() (la función `kill` con el `pid` en negativo sería equivalente, ver la ayuda en línea).

> Nota: las operaciones que se deben realizar sobre el proceso en segundo plano, una vez localizado en la lista deben ser: cederle el terminal (que hasta ese momento lo tenía el padre), cambiar lo necesario de su

estructura job y enviarle una señal para que continúe (por si estuviera suspendido, por ejemplo, esperando a tener el terminal).

- **Implementar un comando interno bg, para**, trabajando con la lista de tareas, **poner una tarea que está suspendida ejecutando en segundo plano.** El argumento de dicho comando debe ser el identificador del lugar que ocupa ese proceso en la lista (1, 2, 3, ...).

## PROGRAMA FUENTE: Shell_project.c

```
/**
UNIX Shell Project

Sistemas Operativos
Grados I. Informatica, Computadores & Software
Dept. Arquitectura de Computadores - UMA

Some code adapted from "Fundamentos de Sistemas Operativos", Silberschatz et al.

To compile and run the program:
   $ gcc Shell_project.c job_control.c -o Shell
   $ ./Shell
       (then type ^D to exit program)

**/

#include "job_control.h"   // remember to compile with module job_control.c

#define MAX_LINE 256 /* 256 chars per line, per command, should be enough. */

// -----------------------------------------------------------------------
//                            MAIN
// -----------------------------------------------------------------------

int main(void)
{
       char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
       int background;             /* equals 1 if a command is followed by '&' */
       char *args[MAX_LINE/2];     /* command line (of 256) has max of 128 arguments */
       // probably useful variables:
       int pid_fork, pid_wait; /* pid for created and waited process */
       int status;             /* status returned by wait */
       enum status status_res; /* status processed by analyze_status() */
       int info;                       /* info processed by analyze_status() */

       while (1)   /* Program terminates normally inside get_command() after ^D is typed*/
       {
              printf("COMMAND->");
              fflush(stdout);
              get_command(inputBuffer, MAX_LINE, args, &background);  /* get next command */

              if(args[0]==NULL) continue;   // if empty command

              /* the steps are:
                     (1) fork a child process using fork()
                     (2) the child process will invoke execvp()
                     (3) if background == 0, the parent will wait, otherwise continue
                     (4) Shell shows a status message for processed command
                     (5) loop returns to get_commnad() function
              */

       } // end while
}
```

Anexo: **Conceptos fundamentales del control de tareas (*job control*) en UNIX**

El propósito fundamental de un shell interactivo es leer comandos desde el terminal (del usuario) y crear procesos para ejecutar los programas especificados por esos comandos. Como ya hemos visto, el shell puede hacer esta función mediante las llamadas al sistema `fork` y `exec`.

Cuando se lanza a ejecutar un comando, todos los procesos que realizan esta tarea (y los hijos de estos procesos) forman parte de lo que se llama un **grupo de procesos** (*process group*) o **tarea** (*job*). Por ejemplo, un simple comando de compilación como `'cc -c shFSO.c'` puede lanzar varios procesos para su ejecución (*preprocessing, compilation, assembly, linking*) que completan la tarea (*job*) de compilación.

Todos los procesos pertenecen a un grupo de procesos. Cuando se crea un proceso con fork, éste será miembro del mismo grupo de procesos que su proceso padre. Se puede poner un nuevo proceso en otro grupo de procesos usando la llamada al sistema `setpgid`.

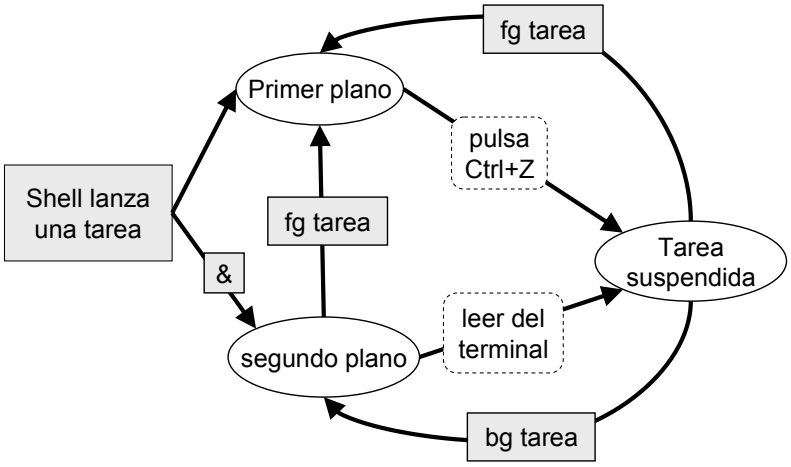
En UNIX existe una agrupación de procesos aún más grande que la tarea: **la sesión**. Una nueva sesión se crea cuando hacemos login en el sistema y el shell se convierte en el primer proceso (leader) de la sesión. Todas las tareas (grupos de procesos) que cree el shell formarán parte de su sesión. Asociado a cada sesión existe un **terminal**, un dispositivo de entrada/salida que permitirá a las tareas de la sesión comunicarse con el usuario a través de la entrada y salida estándar (teclado y pantalla)

Un shell de UNIX con control de tareas debe gestionar y ordenar qué tarea puede usar el terminal en cada momento para realizar su entrada/salida. Si no fuese así, podrían coincidir varias tareas tratando de leer del terminal a la vez y se produciría una indeterminación acerca de qué tarea recibiría finalmente la entrada tecleada por el usuario en el terminal. Para prevenir estas confusiones, el shell debe asignar el uso del terminal de forma apropiada mediante la función `tcsetpgrp`.

El shell le dará acceso y uso ilimitado del terminal a un sólo grupo de procesos (tarea) en cada instante. A este grupo de procesos que está en uso del terminal se le conoce como **tarea de primer plano** (*foreground job*). Los otros grupos de procesos que ha lanzando el shell y que se ejecutan sin acceso al terminal se conocen como **tareas en segundo plano** (*background jobs*). Por lo tanto, en una sesión sólo puede haber una tarea activa en primer plano. El shell será la tarea de primer plano mientras acepta los comandos del usuario y después el propio shell es el encargado de transferir el control del terminal a cada tarea que vaya a ejecutar en primer plano. Cuando una tarea de primer plano termina o se suspende, el shell se encarga de recuperar el control del terminal y se convierte de nuevo en la tarea de primer plano para leer el siguiente comando del usuario.

El shell además de controlar qué tareas tiene corriendo en primer y segundo plano, debe controlar cuándo una tarea se encuentra **suspendida**. Si una tarea en segundo plano intenta leer del terminal (al que no tiene acceso por ser de segundo plano), el driver del terminal suspenderá la tarea (si el modo de funcionamiento del terminal `TOSTOP` está activo, también se suspenderá al intentar escribir en el terminal desde segundo plano). Además el usuario puede suspender la tarea de primer plano pulsando el carácter SUSP (CTRL+Z) en el teclado, y un programa puede suspender cualquier tarea enviándole la señal `SIGSTOP`. Es responsabilidad del shell notificar el momento en que las tareas se suspenden y facilitar mecanismos para permitir de forma interactiva al usuario continuar tareas suspendidas y decidir si una tarea debe continuar en primer o segundo plano (mediante los comandos: `fg, bg y jobs`).

Veamos un esquema de cómo funcionará el control de tareas de nuestro shell:




Será responsabilidad de nuestro shell colocar una tarea en primer o segundo plano al lanzarla, asignándole o no el control del terminal. Cuando una tarea es suspendida por el terminal (al pulsar ctrl.+Z o al intentar leer del terminal en segundo plano) el shell tiene que detectar este evento (analizando el valor `status` que devuelve `wait`) y notificar la suspensión de la tarea mediante un mensaje. El shell deberá implementar los comandos internos:

- `jobs`, para listar las tareas en segundo plano, indicando si están ejecutándose o suspendidas (una tarea suspendida también pude provenir de primer plano y debe ser listada).
- `fg`, para continuar ejecutádo una tarea en primer plano (la tarea estaba suspendida o ejecutando en segundo plano)
- `bg`, para continuar ejecutando una tarea en segundo plano (la tarea estaba suspendida)

Por último, recalcar que todos los procesos agrupados en una tarea (grupo de procesos), pueden ser controlados en grupo mediante señales. Por ejemplo cuando pulsamos CTRL+C (carácter INTR) el driver del terminal manda a todos los procesos que forman parte de la tarea de primer plano la señal SIGINT para finalizarlos. De igual forma el resto de señales que tienen que ver con control de tareas son enviadas al conjunto de procesos que forman una tarea (grupo de procesos). Algunas de estas señales son:

| | |
|---|---|
| `SIGINT` | carácter INTR (CTRL+C ) *interrupt* desde el terminal |
| `SIGQUIT` | carácter QUIT (CTRL+\) *quit* desde el terminal |
| `SIGTSTP` | carácter SUSP (CTRL+Z) *stop* desde el terminal |
| `SIGTTIN` | tarea en background intentando leer del terminal |
| `SIGTTOU` | tarea en background intentando escribir en el terminal |