Proyecto Shell sobre Linux
Departamento de Arquitectura de Computadores
E.T.S. Ingeniería Informática
Autor: Manuel Ujaldón

En este proyecto desarrollaremos un intérprete de comandos o *Shell* de aspecto similar al ya existente en Linux, que permite al usuario lanzar sus programas y ejecutar comandos de manera personalizada.

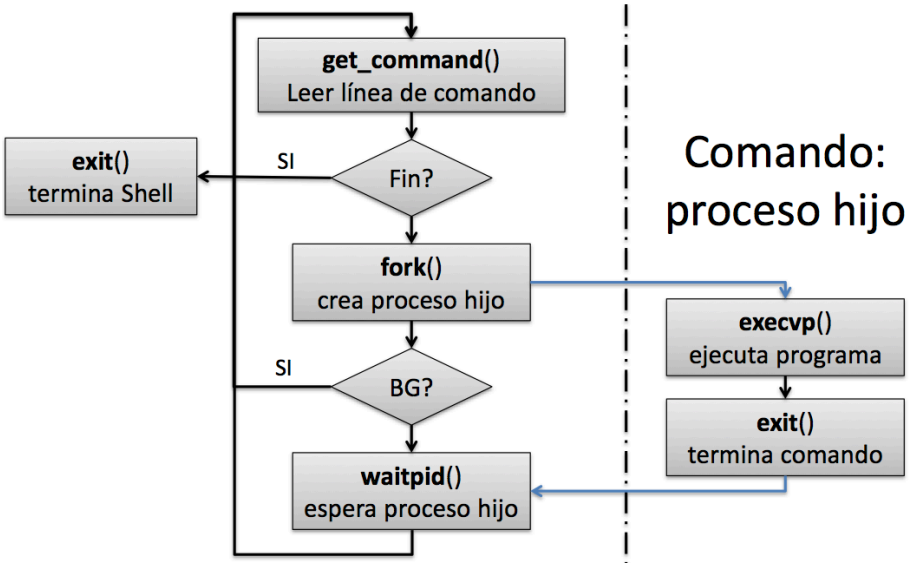
Se proporciona una versión inicial implementada en lenguaje C, `ProyectoShell.c`, que se apoya en funciones de otro fichero auxiliar, `ApoyoTareas.c` (junto a su cabecera, `ApoyoTareas.h`). A lo largo de nuestro trabajo iremos iterando sobre los 4 pasos siguientes:

1. **Compilar:** `gcc ProyectoShell.c ApoyoTareas.c -o MiShell`
2. **Ejecutar:** `./MiShell`
3. **Salir:** Pulsar `Control+D`
4. **Desarrollar:** Editar el fichero `ProyectoShell.c` para depurar errores e incorporar poco a poco las tareas que se solicitan a continuación. El ciclo de desarrollo conlleva realizar los pasos 1, 2 y 3 cada vez que se quiera verificar una funcionalidad nueva incorporada, e ir creciendo así por capas hasta lograr una versión estable y completa del Shell (que a partir de ahora llamaremos `MiShell`). Se recomienda compilar de forma frecuente para tener siempre bien acotadas las posibles fuentes de error, e ir manteniendo un histórico de versiones previas del archivo `ProyectoShell.c` con objeto de disponer de una versión estable en caso de que la incorporación de código nuevo nos lleve a un punto errático o difícilmente depurable.

`MiShell` debe leer el comando tecleado por el usuario y generar un proceso hijo que lo ejecute. Para ello, se apoyará en las siguientes cuatro llamadas al sistema:

- `fork()`: Crea un proceso hijo.
- `waitpid()`: Espera a que termine un proceso hijo.
- `execvp()`: Lanza un programa.
- `exit()`: Termina un proceso.

Consultar la ayuda en línea disponible en Linux a través del comando `man` para ampliar esta información. El siguiente organigrama ubica estas cuatro llamadas dentro de `MiShell`:



Cuando se lanza a ejecutar un comando, todos los procesos que lo integran (y sus sucesivos hijos) forman parte de un **grupo de procesos** *(process group)*. Por ejemplo, el comando `gcc` anterior que compila nuestro Shell lanza procesos *preprocessing*, *compilation*, *assembly*, *linking*, ... todos ellos dentro de un mismo grupo.

Nuestro Shell dará acceso y uso ilimitado del terminal a un único grupo de procesos en cada instante, correspondiente al comando que se ejecuta en primer plano *(foreground - FG)*. El resto de grupos de procesos que haya lanzado el Shell se ejecutan sin acceso al terminal, y se denominan procesos en segundo plano *(background - BG)*. El Shell estará en primer plano mientras acepta los comandos del usuario, y transferirá el control del terminal a cada comando que se vaya a ejecutar en primer plano. Cuando éste acabe o se suspenda, el Shell recuperará el control del terminal y se convertirá de nuevo en la tarea de primer plano para leer el siguiente comando introducido por el usuario.

Primera fase: Estructura básica

Una vez compilado y ejecutado `ProyectoShell.c`, el ejecutable `MiShell` se mete en un bucle infinito del que se sale pulsando `Control+D`. Hasta entonces, cada iteración de ese bucle recoge un comando introducido por el usuario a través de lo que se conoce como el *prompt* del sistema, esto es, el interfaz de diálogo que nosotros denominaremos a partir de ahora **entrada** (y a la que hemos dotado de la siguiente apariencia: `COMANDO->`). Los comandos introducidos aquí desde el teclado, que a partir de ahora recuadraremos para su mejor identificación - `comando` - deben ser atendidos de tres formas distintas:

1. Comando correcto ejecutado en primer plano. Debe aparecer en pantalla el pid del proceso, el nombre del comando y cómo ha terminado. Ejemplos:

   Si introducimos COMANDO->`pwd`, debe mostrar el directorio actual (como corresponde a la ejecución de ese comando), y a continuación deberá verse en pantalla lo siguiente:

   ```
   Comando pwd ejecutado en primer plano con pid 5615. Estado finalizado. Info: 0.
   ```

   De forma similar, si introducimos COMANDO->`ls`, veremos en pantalla un listado de los archivos del directorio, y a continuación deberá mostrarse:

```
Comando ls ejecutado en primer plano con pid 5616. Estado finalizado. Info: 0.
```

2. Comando correcto con sufijo " &" para ser ejecutado en segundo plano. Aparecerá en pantalla únicamente el pid del proceso y el nombre del comando. Ejemplo:
   Si introducimos COMANDO->`sleep 3 &`, deberá verse en pantalla lo siguiente:

   ```
   Comando sleep ejecutado en segundo plano con pid 5622.
   ```

3. Comando incorrecto. Se debe mostrar un mensaje de error. Ejemplo:
   Si introducimos COMANDO->`grita`, deberá verse en pantalla lo siguiente:

   ```
   Error. Comando grita no encontrado.
   ```

Para realizar estas tres funciones básicas, es necesario incorporar a `ProyectoShell.c` el código que se encargue de dos cosas:

1. Lanzar los comandos desde un proceso hijo:
   - En primer plano (FG), en cuyo caso el padre deberá esperar a la finalización del hijo.
   - En segundo plano (BG), continuando entonces el padre con una nueva iteración del bucle para aceptar el siguiente comando desde teclado.
2. Imprimir un mensaje para informar de la terminación (o no) del comando, junto con sus datos identificativos (pid, estado e info) según hemos ejemplificado.

Segunda fase: Tratamiento de comandos internos

Además de los programas ejecutables y los comandos externos, desde `MiShell` se pueden lanzar comandos internos que el propio Shell se encarga de transformar en operaciones al nivel del Sistema Operativo. Aprende a implementarlos aceptando un par de ellos:

1. El comando `cd` debe cambiar el directorio de trabajo actual (utiliza para ello la función *chdir()*).
2. El comando `logout` debe salir de `MiShell` (utiliza para ello la función *exit()*).

Tercera fase: Cesión del terminal

Los comandos en primer plano (FG) deben tomar el terminal desde el proceso hijo, que previamente deberá definir su propio grupo de procesos `gpid` mediante la función *new_process_group()*. El padre deberá retomar el terminal una vez acabe el hijo, puesto que el terminal debe quedar asignado exclusivamente al comando en primer plano. Esta asignación se realiza desde la función *set_terminal()* implementada como macro en el fichero `ApoyoTareas.h`.

La correcta cesión del terminal requiere desactivar las señales asociadas al terminal mediante la función *ignore_terminal_signals()*, y reactivarlas antes de ejecutar el comando mediante la función *restore_terminal_signals()*. La coordinación necesaria entre padre e hijo se resume en el organigrama de la Figura 1.

Para averiguar si un proceso hijo ha terminado o no, hay que examinar el valor del segundo parámetro, `status`, devuelto por la llamada a la función *waitpid()*. La consulta de este valor es más sencilla y elegante si nos apoyamos en el tipo de dato enumerado `status` y la función *analyze_status()* que hemos incorporado en el fichero `ApoyoTareas.h`. Por ejemplo:

```
pid_t pid = waitpid(pgid, &status, ...);
enum status estado = analyze_status(status, &info);
if (estado == FINALIZADO) ...
```

Además, `MiShell` debe informar acerca de los procesos en primer plano que han sido suspendidos. Este evento se controla desde la opción `WUNTRACED` de la función *waitpid()*. Otras opciones que usaremos a partir de la siguiente fase son `WNOHANG` y `WCONTINUED`. Un ejemplo de uso conjunto de estas 3 opciones es el siguiente:

```
pid = waitpid(pgid, &status, WUNTRACED | WNOHANG | WCONTINUED);
```

Cuarta fase: Manejador de señales

Hasta este punto, nuestro Shell desatiende los comandos que son lanzados en segundo plano, lo que provoca que queden en estado *zombie* al terminar. Esto lo puedes comprobar usando el comando `ps`, que proporciona la lista de procesos que habitan en tu sistema (el estado *zombie* aparece como <`defunct`>).

Para evitar esta situación, vamos a instalar un manejador de la señal `SIGCHLD` que trate la terminación o suspensión de estos comandos de forma síncrona y controlada. La función *signal()* activa el manejador de la señal `SIGCHLD` de la siguiente forma:

```
signal(SIGCHLD, manejador);
```

Este manejador deberá además notificar dicha terminación o suspensión imprimiendo por pantalla mensajes como:

```
Comando xclock ejecutado en segundo plano con PID 2345 ha concluido.
Comando sleep ejecutado en segundo plano con PID 6789 ha suspendido su ejecución.
```

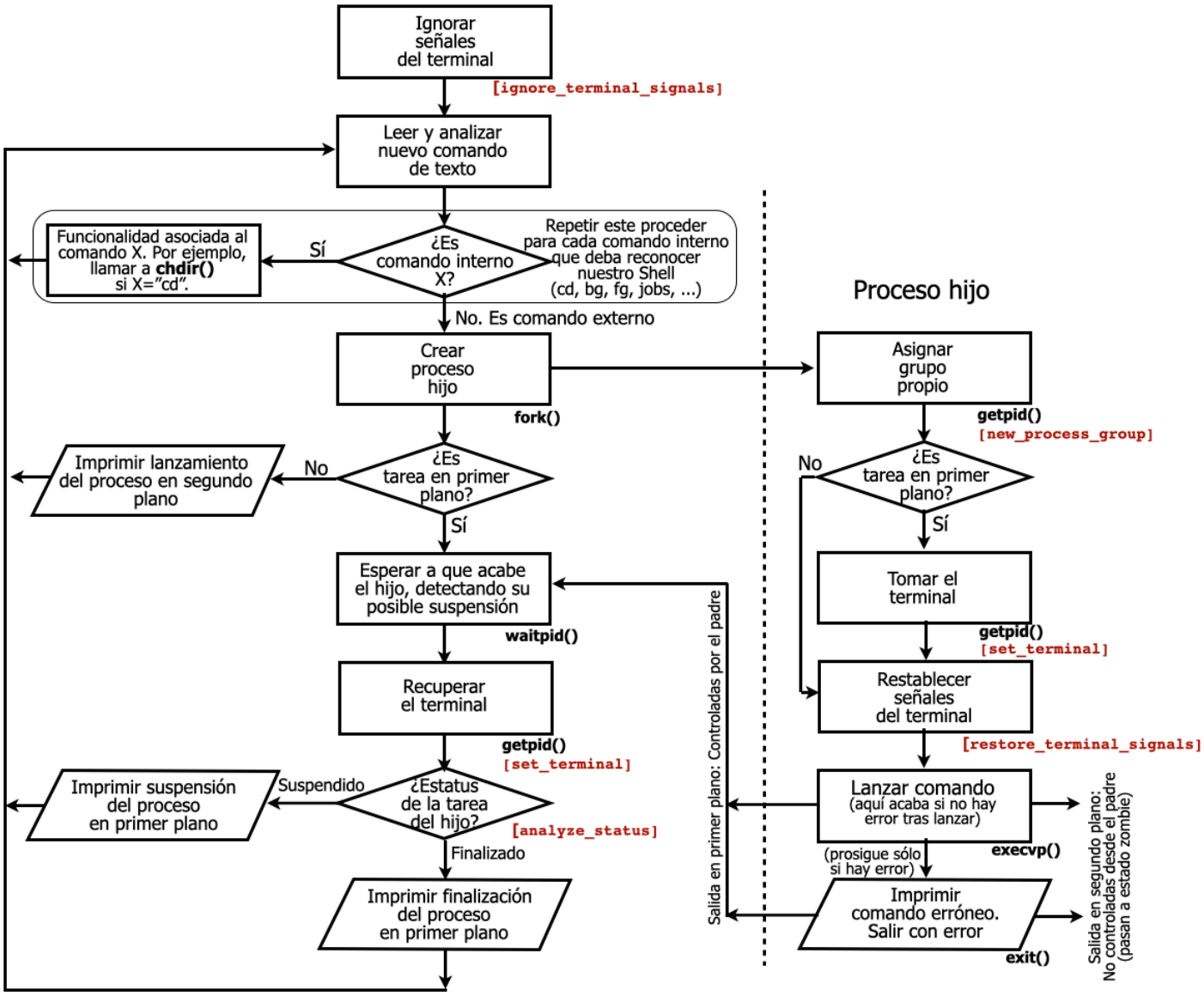


Figura 1: Organigrama que resume el proceder de nuestro Shell para las tres primeras fases de implementación. Hemos representado las llamadas al sistema en negrita y con sufijo '()', y las funciones de apoyo contenidas en el fichero `ApoyoTareas.c` en rojo y entre corchetes.

A su vez, crearemos una lista de tareas, haciendo uso de las funciones descritas en el fichero `ApoyoTareas.c`. En ella iremos incorporando cada comando lanzado en segundo plano (así como el comando en primer plano en caso de que sea suspendido). Si consultamos en el apéndice final de este documento la estructura de datos `job` que contiene la información necesaria para registrar cada comando en la lista de tareas, vemos que se incluye una variable `ground` (para denotar si es **fore**ground o **back**ground, es decir, si está en primer o segundo plano). `ground` contempla además un tercer estado, `Detenido`, para reflejar el caso de que el comando sea suspendido.

Para hacer un correcto seguimiento de cada proceso, el Shell deberá encargarse de actualizar el valor de `ground` en cada caso. No ocurre lo mismo con otra variable similar, `status`, que devuelve como segundo parámetro la función *waitpid()*, y que va actualizando automáticamente el Sistema Operativo según evoluciona el comando (esto es, finaliza o cambia de estado al recibir una señal). El uso conjunto de `ground` y `status` permitirá a nuestro Shell, además, implementar la funcionalidad de la próxima fase, tal y como hemos reflejado en la Figura 3.

Al igual que ya ocurrió con `status`, para manipular los tres valores que puede tomar `ground` se ha habilitado un tipo de datos enumerado con su mismo nombre dentro del fichero `ApoyoTareas.h`, tal y como se describe de forma conjunta en el apéndice de este documento:

```
enum status { SUSPENDIDO, REANUDADO, FINALIZADO };
enum ground { PRIMERPLANO, SEGUNDOPLANO, DETENIDO };
```

Una vez creada la lista de tareas, la misión de nuestro manejador será revisar cada una de sus entradas para comprobar si algún comando lanzado en segundo plano (1) ha sido suspendido, (2) ha terminado o (3) ha reanudado su ejecución tras ser suspendido. Ya vimos que (1) se controla desde la opción `WUNTRACED` en el tercer argumento de la función *waitpid()*. (2) y (3) se controlan de forma similar desde otras dos opciones de esa misma función *waitpid()*:

- `WNOHANG`, que permite comprobar si un proceso ha terminado.
- `WCONTINUED`, que permite saber si un hijo suspendido ha reanudado su ejecución.

Estas tres opciones, usadas conjuntamente, permiten hacer todas esas consultas sin bloquear a MiShell. Es decir, si el hijo cuyo pid se utiliza como primer argumento en la función *waitpid()* no ha cambiado su `status`, la función no se queda esperando a que lo haga, sino que termina devolviendo un valor 0.

La Figura 2 refleja en verde la funcionalidad añadida para esta cuarta fase.

Quinta fase: Gestión de comandos en primer y segundo plano

En esta última fase implementaremos la gestión de comandos en segundo plano o *background* (esto es, aquellos que se lanzan terminados en " &").

Cuando un comando se lanza en segundo plano, no se le debe asociar el terminal, ya que el Shell queda en primer plano y debe seguir leyendo del teclado.

En la lista de tareas que hemos creado en la fase anterior debemos controlar la ejecución de múltiples comandos, tanto en segundo plano como suspendidos. Un comando queda suspendido por dos razones:

- Se pulsó `Ctrl+Z` mientras se estaba ejecutando en primer plano.

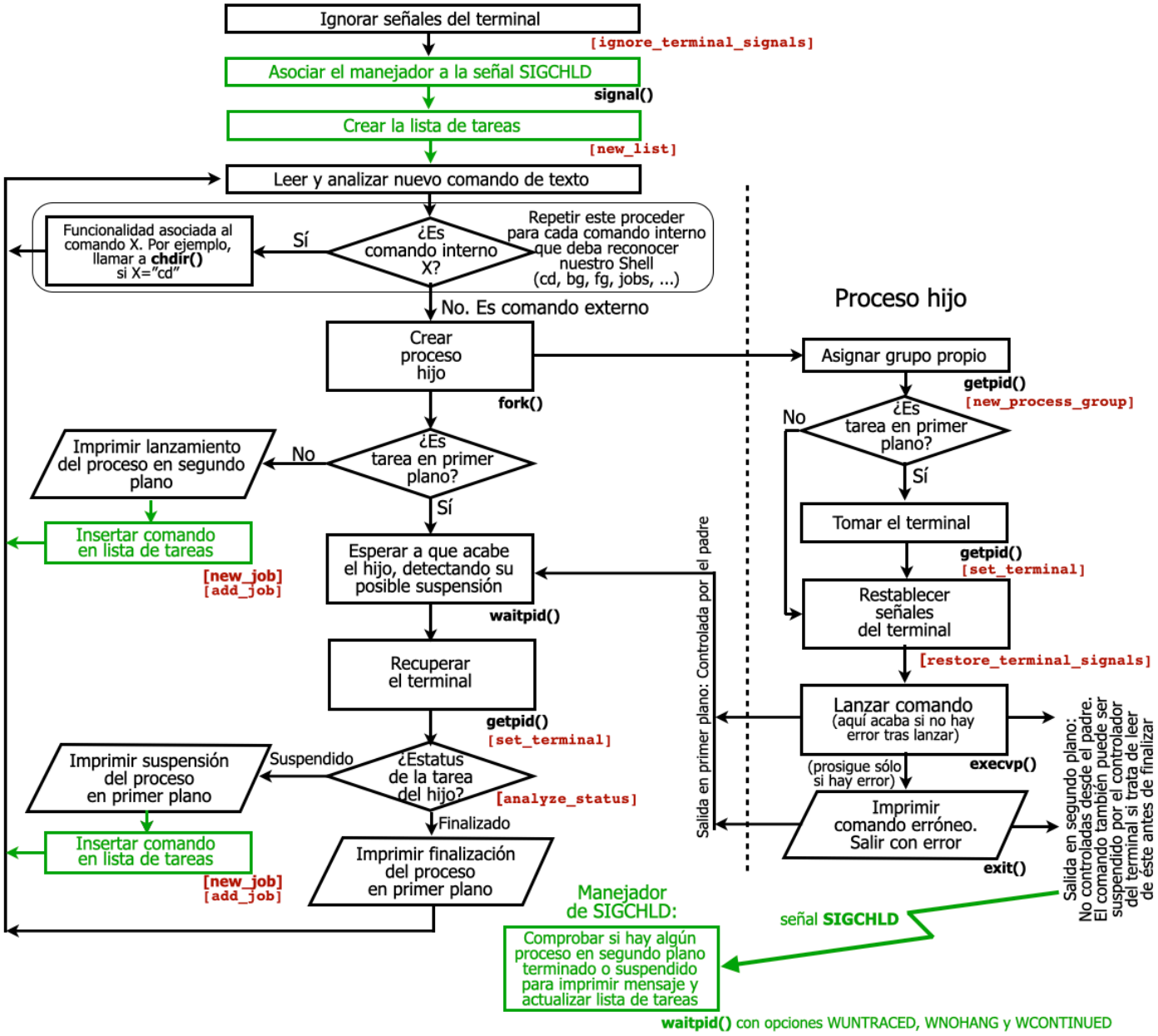


Figura 2: Organigrama que resume el proceder de nuestro Shell para la cuarta fase de implementación. Hemos representado las llamadas al sistema en negrita y con sufijo '()', y las funciones de apoyo contenidas en el fichero `ApoyoTareas.c` en rojo y entre corchetes. La funcionalidad incorporada respecto a la Figura 1 se ha coloreado de verde.

- Intentó leer del terminal mientras se estaba ejecutando en segundo plano.

En ambos casos, el Sistema Operativo manda la señal `SIGSTOP` al proceso asociado a ese comando para provocar su suspensión. Como ya hemos visto en fases anteriores, dicha suspensión se detecta analizando el valor `status` devuelto por la llamada al sistema *waitpid()*.

Una peculiaridad de nuestro código es la concurrencia, es decir, mientras se está ejecutando el código del Shell puede llegar la señal `SIGCHLD` y lanzar su manejador. Dado que la lista de procesos que hemos creado es una estructura de datos compartida por Shell y manejador, se debe impedir que ambos puedan actualizarla de forma simultánea. Para ello, si uno lo hace, el otro debe esperar a que acabe de hacerlo. Este mecanismo de sincronización puede implementarse bloqueando la señal antes de efectuar cualquier modificación en la lista de procesos, y desbloqueándola después. Para ello están disponibles en el fichero `ApoyoTareas.c` las funciones de apoyo *block_SIGCHLD()* y *unblock_SIGCHLD()*.

En esta quinta fase nuestro Shell debe incorporar tres comandos internos:

1. `jobs`. Imprimirá una lista de comandos que estén siendo ejecutados en segundo plano o hayan sido suspendidos. Por tanto, incluirá tanto los comandos que se lanzaron con sufijo " &" como los que hayan detenido su ejecución (tanto en primer plano al pulsar `Ctrl+Z` como en segundo plano por intentar leer del terminal). El fichero `ApoyoTareas.c` tiene una función para imprimir una lista de tareas. Si la lista estuviera vacía, deberá imprimirse un mensaje indicándolo.

2. `fg`. Permite pasar a primer plano un comando que estuviera en segundo plano o suspendido, por lo que deberá retomar el terminal. `fg` puede usar como argumento el lugar que ocupa en la lista (1, 2, ...), y si no se usa dicho argumento, tomará el primero de la lista (esto es, el último que ingresó).

3. `bg`. Permite pasar a segundo plano un comando que se encuentre suspendido. El argumento será igualmente el lugar que ocupa en la lista, y el primero por omisión.

Las operaciones a realizar sobre el proceso que representa cada comando en nuestra lista de tareas son las siguientes:

1. Cederle el terminal (que hasta ese momento tenía el padre) si pasa a primer plano.

2. Actualizar su estado `ground` y la lista de tareas para reflejar la nueva situación.

3. Enviarle la señal `SIGCONT` para que continúe si estuviera detenido (por ejemplo, esperando a disponer de la terminal). Esta señal debe enviarse al grupo de procesos completo, usando para ello la función *killpg()*, de igual forma que actúa el controlador del terminal del Sistema Operativo al enviar la señal `SIGSTOP` cuando pulsamos `Ctrl+Z` para forzar su suspensión.

La Figura 3 resume en un diagrama de estados todas las consideraciones de esta fase final de nuestro Shell, y la Figura 4 sintetiza todas las fases en un pseudo-código preliminar.

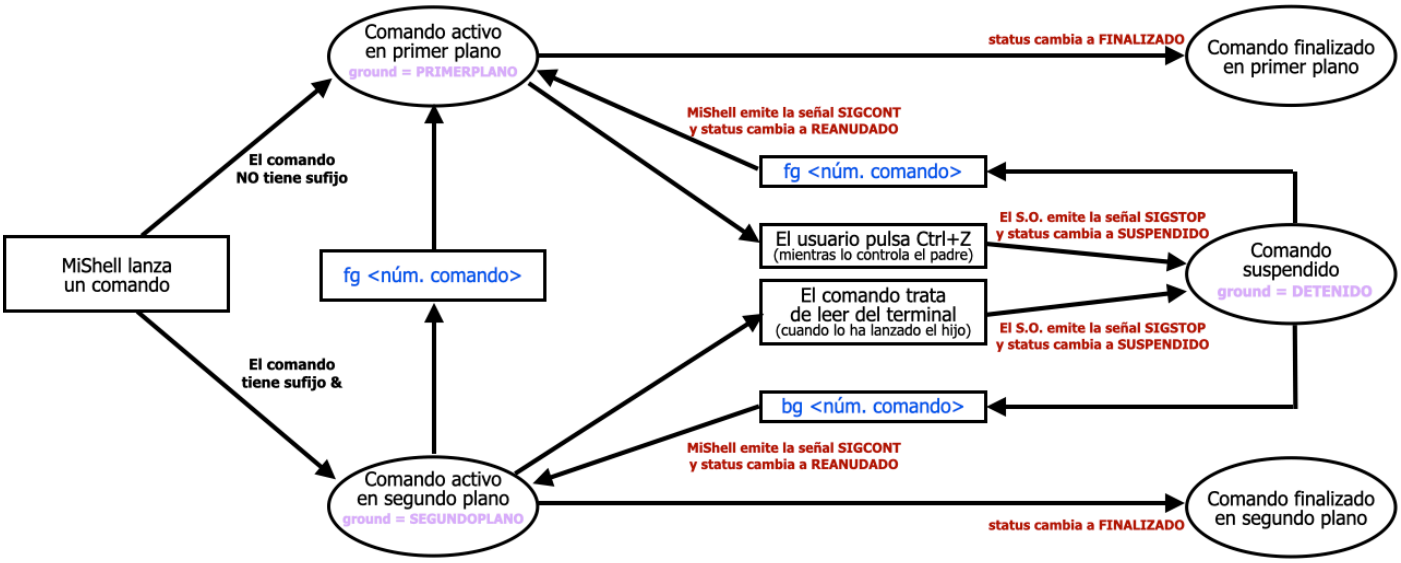


Figura 3: Diagrama de estados de la fase 5, reflejando en rojo los cambios que el Sistema Operativo va realizando sobre `status` a medida que evoluciona el proceso vinculado a cada comando introducido desde nuestro Shell. Estos cambios suelen producirse cuando el proceso recibe señales, unas veces emitidas por el propio Sistema Operativo (como en el caso de la señal `SIGSTOP`), y otras veces generadas por nuestro Shell a través de la función *killpg()* (como ocurre con la señal `SIGCONT`).

```
void manejador(int senal)
{
//  Recorrer la lista de comandos, y para cada uno de ellos, si no es un comando
//  en primer plano, mirar si ha cambiado su estado, y en ese caso, informar
//  por pantalla y reflejarlo en la representación del proceso en la lista
}

int main(void)
{
//  Declarar variables
//  Ignorar señales
//  Asociar el manejador a la señal
//  Crear la lista de comandos
  while (1)    // El programa acaba cuando se pulsa Ctrl+D
  {
//  Aceptar comando desde el teclado
//  Gestionar comando "cd" [sencillo]
//  Gestionar comando "logout" [sencillo]
//  Gestionar comando "jobs" [sencillo]
//  Gestionar comando "fg" [complejo]
//  Gestionar comando "bg" [complejo]
//  fork();    // FG = comando en primer plano. BG = comando en segundo plano.
//    El hijo crea su propio grupo de procesos (y coge el terminal si FG)
//    El hijo lanza el comando controlando posibles errores
//    El padre debe distinguir entre FG y BG. Entre otras cosas:
//      - Si es FG, debe coger el terminal
//      - Según sea FG o BG, incorporará el proceso a la lista de forma distinta
  }
}

FASE 1    FASE 2    FASE 3    FASE 4    FASE 5
```

Figura 4: Pseudo-código preliminar que sintetiza todas las fases de nuestro Shell.

Apéndice: Descripción de las funciones de apoyo suministradas en el fichero auxiliar ApoyoTareas.c

---

`get_command` (función)

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

Utilidad: Controla la entrada desde teclado.

Entradas:

- `inputBuffer` es el buffer de caracteres introducidos desde el teclado.
- `size` es el tamaño del buffer.

Salidas:

- `args` devuelve 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.
- `background` devuelve 1 (true) si el comando termina con & (que en nuestro Shell es el símbolo que utilizamos para indicar la ejecución de un comando en segundo plano.

`status, ground, status_strings, ground_strings` (tipos de datos enumerados)

```
enum status { SUSPENDIDO, REANUDADO, FINALIZADO };
enum ground { PRIMERPLANO, SEGUNDOPLANO, DETENIDO };

static char* status_strings[] = { "Suspendido", "Reanudado", "Finalizado" };
static char* ground_strings[] = { "PrimerPlano", "SegundoPlano", "Detenido" };
```

Utilidad: `status` identifica la evolución de un comando, mientras que `ground` (para denotar si es **fore**ground o **back**ground), indica el estado actual del mismo, que también puede ser detenido. Manejando tipos de datos enumerados y asociándoles cadenas de caracteres podemos imprimir directamente por pantalla su valor. Por ejemplo, si declaramos `estado` como variable de tipo `status`, podemos usar:

```
printf("Motivo de la terminacion del comando: %s\n", status_strings[estado]);
```

`analyze_status` (función)

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

Utilidad: Permite monitorizar la evolución de un hijo desde su padre.

Entrada:
- `status` es el mismo valor entero que devuelve la llamada al sistema *waitpid()*.

Salidas:
- La función devuelve en un tipo enumerado `status` la causa de terminación de ese comando.
- `info` devuelve la información asociada a dicha terminación.

`job` (tipo de dato estructurado)

```
typedef struct job
{
  pid_t pgid;        // id para este grupo de procesos
  char *command;     // nombre del comando
  enum ground ground;  // registra si el comando está en primer o segundo plano o detenido
  struct job_ *next; // siguiente trabajo de la lista
} job;
```

Utilidad: Permite representar los comandos y enlazarlos en la lista de tareas. Esta lista de tareas será un puntero a `job` (esto es, `job *`), y cada nodo de la lista irá enlazado a partir de ahí, también como un puntero a `job`.

`new_job` (función)

```
job *new_job(pid_t pid, const char *command, enum ground commandground);
```

Utilidad: Creación de la tarea asociada a un nuevo comando.

Entradas:
- `pid`: Es el pid del proceso asociado al comando.
- `command`: Es la cadena de caracteres con el nombre del comando.
- `commandground`: Indica si el comando se lanza en primer o segundo plano (*foreground* o *background*).

Salida:
- `job` devuelve un puntero a la tarea que ubica el comando en memoria.

`new_list, list_size, empty_list, print_job_list` (cuatro macros)

```
job *new_list(const char *name);
```

Utilidad: Devuelve el puntero a una nueva lista de comandos que crea con el nombre proporcionado como argumento.

```
int list_size(job *list);
```

Utilidad: Devuelve el número de elementos de la lista cuyo puntero se pasa como argumento.

```
int empty_list(job *list);
```

Utilidad: Devuelve 1 (true) si la lista de comandos que se pasa como argumento está vacía.

```
void print_job_list(job *list);
```

Utilidad: Imprime en pantalla el contenido de la lista de comandos que se pasa como argumento.

`add_job, delete_job, get_item_bypid, get_item_bypos` (cuatro funciones)

```
(1) void add_job (job *list, job *item);
(2) int delete_job(job *list, job *item);
```

Utilidad: (1) Añade o (2) suprime de la lista `list` un comando representado por `item`.

```
(3) job *get_item_bypid(job *list, pid_t pid);
(4) job *get_item_bypos(job *list, int n);
```

Utilidad: Devuelve un puntero al comando de la lista `list` que (3) tiene el `pid` que se pasa como argumento, o que (4) está ubicado en la n-ésima posición de dicha lista.

Ejemplo de uso de una de estas 4 funciones junto con una de las 4 macros del lote anterior:

```
job *mi_lista_de_comandos = new_list("Comandos de mi Shell");
...
add_job(my_job_list, new_job(pid_fork, args[0], background) );
```

`ignore_terminal_signals(), restore_terminal_signals()` (dos macros)

```
void ignore_terminal_signals();
```

Utilidad: Permite desactivar las señales que llegan a la terminal que gobierna nuestro Shell (`SIGINT`, `SIGQUIT`, `SIGTSTP`, `SIGTTIN` y `SIGTTOU`). No es necesario hacer un tratamiento individualizado de cada una de ellas, sino de forma conjunta, ya que se trata únicamente de evitar que cualquiera de ellas nos haga perder el control del terminal. De esta manera, desactivar estas señales será lo primero que hagamos (antes incluso de aceptar por teclado el comando de entrada), y una vez hayamos creado con *fork()* el proceso asociado al comando, procederemos a restaurar el comportamiento por defecto utilizando su contrapartida:

```
void restore_terminal_signals();
```

`set_terminal` (macro)

```
void set_terminal(pid_t pid);
```

Utilidad: Asigna el terminal al proceso o grupo de procesos identificado(s) por el argumento `pid`. Nuestro Shell deberá ceder el terminal a cada comando ejecutado en primer plano, y recuperarlo cuando éste acabe o sea suspendido.

`new_process_group` (macro)

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

Utilidad: Crea un nuevo grupo de procesos para asociarlo a un proceso recién creado, cosa que deberá hacer nuestro Shell después de llamar a `fork()` (cuando crea el proceso asociado al nuevo comando introducido desde teclado).

`block_SIGCHLD, unblock_SIGCHLD` (dos macros)

```
void block_SIGCHLD();
void unblock_SIGCHLD();
```

Utilidad: Bloquean la señal `SIGCHLD` en las secciones de código en las que nuestro Shell tenga que modificar o acceder a estructuras de datos como la lista de comandos. Estas estructuras pueden ser accedidas concurrentemente desde el manejador de esta señal, y por lo tanto, pueden producirse condiciones de carrera (inconsistencias por el acceso simultáneo a datos compartidos) que deben evitarse. Este recurso es sólo necesario para la parte avanzada de las prácticas, a partir de nuestra tercera sesión de trabajo.