

Mini Shell

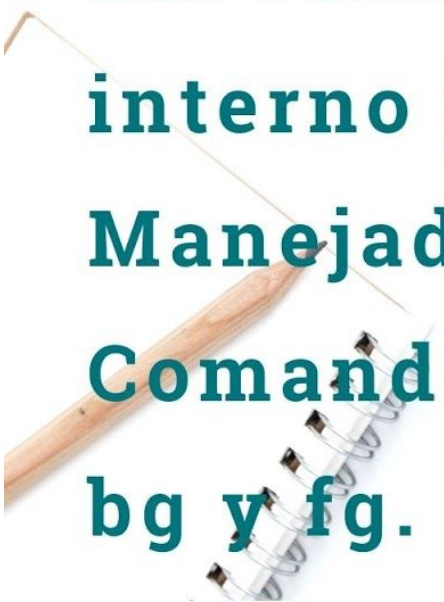
Nivel 5

**Ejecución en
segundo plano
(background).**

**Implementación
del comando
interno jobs.**

Manejador de ^Z.

**Comandos internos
bg y fg.**



Indice

¿Qué conceptos trabajamos con estos retos?	4
¿Cuáles son los detalles de implementación?	4
¿Qué recursos metemos en nuestras mochilas?	7
¿Cómo comprobamos que hemos superado este nivel?	8

En este nivel gestionaremos procesos en background (que añadiremos a nuestro array de `jobs_list`) y añadiremos también la gestión de una nueva señal, la **SIGTSTP** (producida al pulsar **Ctrl + Z**), que al igual que en el nivel anterior, haremos que el hijo la ignore y que el padre la gestione a través de un manejador que enviará una señal similar, la **SIGSTOP**, al proceso en foreground (si éste existe y no es el mini shell ejecutado dentro del mini shell).

Además añadiremos 1 comando interno, **jobs**, que nos permitirá obtener una lista de los trabajos existentes y su estado (necesitaremos funciones auxiliares para gestionar el array `jobs_list`: añadir trabajos, eliminarlos, buscarlos...) y otros 2 comandos internos más, el **fg** y el **bg**, que nos permitirán jugar con la reactivación de procesos detenidos y el paso de primer plano a segundo y viceversa.

¿Cuáles son los retos de este nivel?

En este nivel nuestro mini shell tendrá que:



- Diferenciar entre ejecución en primer plano y en segundo plano (comando finalizado en **&**).
- Gestionar la lista de trabajos y su estado ('E' ejecución, 'D' detenido).¹
- Mostrar una lista de los trabajos existentes mediante el comando interno **jobs**.
- Detener un proceso en foreground cuando se reciba la señal **SIGTSTP** (producida por Ctrl+Z) mediante un manejador propio y pasarlo a la lista de trabajos.

¹ Por señal **SIGTSTP** producida por Ctrl+Z

Y además vuestro mini shell tendrá que ser capaz de:

- Mediante el comando interno **fg**, enviar un trabajo detenido al foreground reactivando su ejecución, o uno del background al foreground
- Mediante el comando interno **bg**, reactivar un proceso detenido para que siga ejecutándose pero en segundo plano



¿Qué conceptos trabajamos con estos retos?

- [Procesos en segundo plano](#)
- [Comandos jobs, fg, bg. Señales SIGTSTP \(^Z\) y SIGCONT](#)

¿Cuáles son los detalles de implementación?

Habrá que cumplir las siguientes especificaciones:

- Si el comando se ejecuta en foreground, el mini shell debe esperar a que finalice para aceptar el siguiente comando (mediante PAUSE()).
- Al pulsar Ctrl+C se debe detener el proceso en ejecución en primer plano **pero no el mini shell ni los procesos en segundo plano**.
- Cuando un proceso es ejecutado con &, el mini shell mostrará por pantalla el identificador de trabajo y los datos del proceso, y presentará nuevamente el *prompt*.
- Cuando termine la ejecución de un proceso en background, el mini shell **mostrará un mensaje indicando qué proceso finalizó** (mostrando su PID y el cmd). No mostrarlo si el que finaliza es el del foreground.

- A cada proceso en background se le asignará un identificador numérico de trabajo a partir del 1. Cuando un proceso finalice, este identificador será reutilizado por el último de la lista.
- El comando **jobs** mostrará la lista de trabajos junto con su identificador de trabajo, indicando el PID, estado y cmd.
- Se implementará un manejador propio para la señal SIGTSTP, ctrlz(), correspondiente a Ctrl+Z, que incorpore el proceso suspendido a la lista de trabajos (pasando el estado de "E" a "D", detenido) y resetee los datos del proceso en foreground.
- Se programarán las funciones para añadir **fg**² y **bg** como comandos internos (en ambos casos el estado pasará de 'D' (Detenido) a 'E' (Ejecutándose))
- Mediante el comando externo `$ kill -9 <PID>` se podrán finalizar procesos en segundo plano (se mostrará por pantalla el PID de los procesos finalizados). No contemplaremos el caso de detención de procesos desde consola con el comando Kill: `$ kill -20 <PID>` ni de su reanudación: `$ kill -18 <PID>` ya que esas acciones no actualizarían nuestra lista de tabajos.

Funciones que hay que crear/ampliar/modificar en este nivel:

```
int execute_line(char *line);
```

Desde aquí llamaremos a la función **is_background()**³ para analizar si en la línea de comandos hay un **&** al final (podemos utilizar una variable "booleana" local para recoger el resultado).

Después del fork():

El proceso hijo:

- ignorará la señal **SIGINT** producida al pulsar Ctrl+C (le asociamos la acción SIG_IGN en vez del SIG_DFL antes de llamar a execvp()) tanto si el proceso se ejecuta en background como en foreground. Mantenemos el manejador propio para el padre, **ctrlc**, que enviará la señal SIGTERM sólo al proceso en foreground para finalizarlo.
- ignorará la señal **SIGTSTP** producida al pulsar Ctrl+Z (le asociamos la acción SIG_IGN en vez del SIG_DFL antes de llamar a execvp()) tanto si el proceso es en background como foreground. Crearemos un manejador propio para el padre, **ctrlz**, que enviará la señal SIGSTOP sólo al proceso en foreground para detenerlo.
- ejecutará la acción por defecto **SIG_DFL** para la señal **SIGCHLD** tanto si es en background o en foreground.

² La señal 18 SIGCONT reanuda un proceso que ha sido detenido con la señal 20 SIGTSTP o con la 19 SIGSTOP. La diferencia entre SIGTSTP y SIGSTOP es que la segunda no puede ser manejada ni ignorada ni bloqueada.

³ Esta función sustituirá el token con el & por NULL.

El proceso padre:

- si se ha creado un hijo en background, añadirá un nuevo trabajo a la lista mediante la función **jobs_list_add()**.
- si no, al igual que en el nivel anterior, dará valores a los campos del proceso en foreground (**jobs_list[0]**) y ejecutará un **pause()** en vez del **wait()**, mientras haya un proceso ejecutándose en foreground⁴.

int jobs_list_add(pid_t pid, char status, char *cmd);

Si no hemos llegado al nº máximo de trabajos permitidos, añadimos el nuevo elemento al array en la posición indicada por la variable global **n_pids** que nos lleva la cuenta de la cantidad de trabajos no finalizados, e incrementamos el valor de dicha variable.

int jobs_list_find(pid_t pid);

Busca en el array de trabajos el PID que recibe como argumento y retorna la posición en él.

int jobs_list_remove(int pos);

Recibe como parámetro la posición del array del trabajo que hay que eliminar y mueve el registro del último proceso de la lista a la posición del que eliminamos. Decrementamos la variable global **n_pids**.

void reaper(int signum);

Al igual que en el nivel anterior, el enterrador controlará si el hijo que acaba es el que se ejecuta en primer plano (**waitpid()** devuelve el pid del hijo que ha terminado), y en tal caso reseteará los datos de **jobs_list[0].pid**, pero **en caso de ser background llamará a la función **jobs_list_find()** para buscar el PID del proceso que ha acabado en la lista de trabajos, imprimirá por la salida estándar de errores que ese proceso ha terminado (indicando los datos del mismo) y llamará a la función **jobs_list_remove()** para eliminar el proceso de la lista.**

int internal_jobs();

Imprimirá por pantalla el identificador de trabajo entre corchetes (a partir del 1), su PID, la línea de comandos y el status (D de Detenido, E de Ejecutando).

⁴ El reaper pondrá **jobs_list[0]** a 0 cuando finalice el proceso en foreground


```
void ctrlz(int signum);5
```

Volvemos a asociar este manejador a la señal SIGTSTP.

Si hay un proceso en foreground entonces

Si el proceso en foreground NO es el mini shell entonces⁶

Enviarle la señal **SIGSTOP** y provisionalmente notificarlo por pantalla.

Cambiar el status del proceso a 'D' (detenido).

Añadir los datos del proceso detenido a job_list[n_pids] utilizando jobs_list_add().

Resetear los datos de job_list[0] ya que el proceso ha dejado de ejecutarse en foreground.

si_no error ("Señal SIGSTOP no enviada debido a que el proceso en foreground es el shell")

fsi

si_no

error ("Señal SIGSTOP no enviada debido a que no hay proceso en foreground")

fsi

En el **main()** asociar los manejadores propios indicados a las señales correspondientes.
Añadimos signal (SIGTSTP, ctrlz)

¿Qué recursos metemos en nuestras mochilas?

Documentos de apoyo:

- [Señales. Llamadas al sistemas con ejemplos prácticos](#)
- [Sincronización de procesos en C](#)
- [Comunicación entre procesos](#)
- [Procesos](#)
- [Señales](#)
- [Lección 4: señales](#)

⁵ **NO** contemplaremos el caso de detención de procesos desde consola con el comando Kill: `$ kill -20 <PID>` (sólo desde teclado con Ctrl+Z), ni de su reanudación desde consola: `$ kill -18 <PID>` (sólo se reanudarán con los comandos internos fg y bg):

- Si tuviéramos un trabajo detenido con Ctrl+Z, guardado como (PID 'D' comando), y ejecutásemos `$ kill -18 <PID>` el proceso continuaría su ejecución (esta vez en segundo plano !!!) y el reaper lo enterraría al finalizar, pero habría entonces que crear un manejador para la señal SIGCONT que gestionara la lista de procesos para indicar la nueva situación: (PID 'E' comando &).
- Por otro lado, si tuviéramos un trabajo guardado como (PID 'E' comando &) y ejecutásemos `$ kill -20 <PID>` el proceso debería detener su ejecución, y debería pasar a (PID 'D' comando), pero nuestro manejador sólo se activa con Ctrl+Z

⁶ Lo sabemos comparando el cmd con el comando de ejecución de nuestro programa que podemos tener guardado en una constante, ej. `"/nivel5"` o mejor aún [obtenerlo desde consola](#) (parámetro argv[0] del main)

¿Cómo comprobamos que hemos superado este nivel?

- Podéis ejecutar vuestro programa **nivel5.c** e ir escribiendo en la línea de comandos los ejemplos que se exponen a continuación, comparando vuestros resultados con los mostrados aquí

```
uib:$ ./nivel5
$ ^C
[ctrlc()→ Soy el proceso con PID 22579 (./nivel5), el proceso en foreground es 0 ()]
[ctrlc()→ Señal 15 (SIGTERM) no enviada por 22579 (./nivel5) debido a que no hay
proceso en foreground]]
$ sleep 40 &
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 22586 (sleep 40 &)]

[1] 22586    E    sleep 40 &
$ sleep 20 &
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 22591 (sleep 20 &)]

[2] 22591    E    sleep 20 &
$ sleep 60 &
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 22595 (sleep 60 &)]

[3] 22595    E    sleep 60 &
$ jobs
[1] 22586    E    sleep 40 &
[2] 22591    E    sleep 20 &
[3] 22595    E    sleep 60 &
$ #esperamos que acabe algún proceso
[reaper()→ Proceso hijo 22591 en background (sleep 20 &) finalizado con exit code 0]

Terminado PID 22591 (sleep 20 &) en jobs_list[2] con status 0
$ jobs
[1] 22586    E    sleep 40 &
[2] 22595    E    sleep 60 &
$ kill -9 225957
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 22637 (kill -9 22595)]

[reaper()→ Proceso hijo 22637 en foreground (kill -9 22595) finalizado con exit code 0]
```

⁷ Mandamos SIGTERM al último trabajo de la lista

[reaper()→ Proceso hijo 22595 en background (sleep 60 &) finalizado por señal 9]

Terminado PID 22595 (sleep 60 &) en jobs_list[1] con status 9

\$ jobs

[1] 22586 E sleep 40 &

\$

[reaper()→ Proceso hijo 22586 en background (sleep 40 &) finalizado con exit code 0]

Terminado PID 22586 (sleep 40 &) en jobs_list[1] con status 0

\$

\$ sleep 15 &

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 22889 (sleep 15 &)]

[1] 22889 E sleep 15

\$ sleep 6

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 22891 (sleep 6)]

^C

[ctrlc()→ Soy el proceso con PID 22579 (./nivel5), el proceso en foreground es 22891 (sleep 6)]

[ctrlc()→ Señal 15 (SIGTERM) enviada a 22891 (sleep 6) por 22579 (./nivel5)]

[reaper()→ Proceso hijo 22891 en foreground (sleep 6) finalizado por señal 15]

\$

[reaper()→ Proceso hijo 22889 en background (sleep 15 &) finalizado con exit code 0]

Terminado PID 22889 (sleep 15 &) en jobs_list[1] con status 0

\$

\$ sleep 30

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 22981 (sleep 30)]

^Z

[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 22981 (sleep 30)]

[ctrlz()→ Señal 19 (SIGSTOP) enviada a 22981 (sleep 30) por 22579 (./nivel5)]

[1] 22981 D sleep 30

\$ sleep 10 &

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 22987 (sleep 10 &)]

[2] 22987 E sleep 10 &

\$ sleep 20

```
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 22992 (sleep 20)]
^Z
[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 22992 (sleep 20)]
[ctrlz()→ Señal 19 (SIGSTOP) enviada a 22992 (sleep 20) por 22579 (./nivel5)]
[3] 22992    D    sleep 20
$
[reaper()→ Proceso hijo 22987 en background (sleep 10 &) finalizado con exit code 0]

Terminado PID 22987 (sleep 10 &) en jobs_list[2] con status 0
$ jobs
[1] 22981    D    sleep 30
[2] 22992    D    sleep 20
$ kill -9 22981
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 23011 (kill -9 22981)]

[reaper()→ Proceso hijo 22981 en background (sleep 30) finalizado por señal 9]

Terminado PID 22981 (sleep 30) en jobs_list[1] con status 9

[reaper()→ Proceso hijo 23011 en foreground (kill -9 22981) finalizado con exit code 0]
$ kill -9 22992
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 23021 (kill -9 22992)]

[reaper()→ Proceso hijo 23021 en foreground (kill -9 22992) finalizado con exit code 0]

[reaper()→ Proceso hijo 22992 en background (sleep 20) finalizado por señal 9]

Terminado PID 22992 (sleep 20) en jobs_list[1] con status 9
$ jobs
$ exit
uib:$
```



Observaciones (recordatorio):

- Por defecto, el comando kill envía la señal **SIGTERM** al proceso especificado. Sin embargo hay ocasiones que un proceso puede no responder a esta señal. Si se desea matar ese proceso se puede enviar la señal 9 (**SIGKILL**). Podemos hacerlo utilizando el número precedido del signo "-" o utilizando el nombre de la señal precedido de "-s":

```
$ kill -9 PID
```

```
$ kill -s SIGKILL PID
```

Explicación de las funciones internal_fg, internal_bg⁸ (las ejecuta el padre):

int internal_fg(char **args); // Implementación del comando interno fg⁹

- Chequear la sintaxis (ha de tener 1 argumento correspondiente al nº de trabajo, o sea el índice pos para jobs_list[]).
- Si pos >= n_pids¹⁰ o pos=0 entonces error("no existe ese trabajo") y retornar.
- Enviar a jobs_list[pos].pid la señal SIGCONT (**si su status es 'D'**), y provisionalmente notificarlo por pantalla. (Si el status es 'E' no hay porqué que enviársela).
- Copiar los datos de jobs_list[pos] a jobs_list[0], habiendo eliminado previamente del cmd el ' &' (en caso de que lo tuviera), y cambiando el estado a 'E'.
- Eliminar jobs_list[pos] utilizando la función jobs_list_remove().
- Mostrar por pantalla el cmd (habiendo eliminado el '&' si lo tenía).
- Mientras haya un proceso en ejecución en foreground ejecutar un pause()

int internal_bg(char **args) // Implementación del comando interno bg¹¹

- Chequear la sintaxis (ha de tener 1 argumento correspondiente al nº de trabajo, o sea el índice pos para jobs_list[]).
- Si pos >= n_pids o pos=0 entonces error("no existe ese trabajo") y salir.
- Si el status de jobs_list[pos] es 'E' entonces error("el trabajo ya está en 2º plano")

⁸ L@s que estéis interesados en analizar otra implementación en la que se usa el identificador de grupo de proceso para controlar que las señales SIGINT y SIGTSTP, producidas desde el teclado, no afecten a todos los procesos que inicialmente pertenecen al mismo grupo del terminal, podéis mirar el código de:

<https://github.com/hungys/mysh/find/master>

⁹ En la lista de trabajos podemos tener las siguientes situaciones:

- (PID 'D' comando)
- (PID 'E' comando &)

en ambos casos el proceso se eliminará de la lista de trabajos (en realidad pasará a la posición 0 que indica que está en foreground)

¹⁰ Suponemos n_pids inicializado a 1

¹¹ En la lista de trabajos podemos tener las siguientes situaciones:

- (PID 'D' comando) → pasará a (PID 'E' comando &)
- (PID 'E' comando &) → dará error porque ya está ejecutándose en 2º plano

y retornar.

- Cambiar el status de ese trabajo a 'E' y **añadir ' & ' a su cmd.**
- Enviar a jobs_list[pos].pid la señal SIGCONT y provisionalmente notificarlo por pantalla.
- Mostrar por pantalla el nº de trabajo, el PID, el estado y el cmd.

Ejemplos de tests para el manejo combinado de las señales¹² SIGTSTP, SIGINT, y comandos internos fg y bg¹³

```
$ sleep 60
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 23320 (sleep 60)]
^Z
[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 23320 (sleep 60)]
[ctrlz()→ Señal 19 (SIGSTOP) enviada a 23320 (sleep 60) por 22579 (./nivel5)]
[1] 23320    D    sleep 60
$ fg 1
[internal_fg()→ Señal 18 (SIGCONT) enviada a 23320 (sleep 60)]
sleep 60
^Z
[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 23320 (sleep 60)]
[ctrlz()→ Señal 19 (SIGSTOP) enviada a 23320 (sleep 60) por 22579 (./nivel5)]
[1] 23320    D    sleep 60
$ fg 1
[internal_fg()→ Señal 18 (SIGCONT) enviada a 23320 (sleep 60)]
sleep 60
^C
[ctrlc()→ Soy el proceso con PID 22579 (./nivel5), el proceso en foreground es 23320 (sleep 60)]
[ctrlc()→ Señal 15 (SIGTERM) enviada a 23320 (sleep 60) por 22579 (./nivel5)]

[reaper()→ Proceso hijo 23320 en foreground (sleep 60) finalizado por señal 15]
$
```

¹² Se muestra un mensaje por pantalla indicando el código de la señal que se envía y a qué proceso. En el caso de Ctrl+C, la función ctrlc() recibe la señal 2 (SIGINT), pero envía la 15 (SIGTERM) al proceso en foreground. Ello es debido a que en esta implementación de execute_line(), para el proceso hijo, se ha tenido que indicar que ignore la señal SIGINT: signal(SIGINT, SIG_IGN), puesto que si se le asociaba la macro SIG_DFL a un proceso cuando se creaba, pero después éste era detenido con Ctrl+Z y vuelto a pasar a foreground con fg, si resultaba que anteriormente se había pulsado Ctrl+C para algún otro proceso, éste también capturaba la señal y finalizaba.

¹³ Se ha omitido el símbolo % en la sintaxis por simplicidad

\$ sleep 20 &

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 23417 (sleep 20 &)]

[1] 23417 E sleep 20 &

\$ bg 1

bg: el trabajo 1 ya está en segundo plano

\$ bg 0

bg: 0: no existe ese trabajo

\$ bg 2

bg: 2: no existe ese trabajo

\$

[reaper()→ Proceso hijo 23417 en background (sleep 20 &) finalizado con exit code 0]

Terminado PID 23417 (sleep 20 &) en jobs_list[1] con status 0

\$ sleep 80

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 23476 (sleep 80)]

^Z

[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 23476 (sleep 80)]

[ctrlz()→ Señal 19 (SIGSTOP) enviada a 23476 (sleep 80) por 22579 (./nivel5)]

[1] 23476 D sleep 80

\$ sleep 90

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 23481 (sleep 90)]

^Z

[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 23481 (sleep 90)]

[ctrlz()→ Señal 19 (SIGSTOP) enviada a 23481 (sleep 90) por 22579 (./nivel5)]

[2] 23481 D sleep 90

\$ sleep 70 &

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 23487 (sleep 70 &)]

[3] 23487 E sleep 70 &

\$ jobs

[1] 23476 D sleep 80

[2] 23481 D sleep 90

[3] 23487 E sleep 70 &

\$ bg 2

[internal_bg()→ señal 18 (SIGCONT) enviada a 23481 (sleep 90 &)]

[2] 23481 E sleep 90 & ¹⁴

\$ jobs

¹⁴ Hay que añadir & al cmd

```
[1] 23476    D    sleep 80
[2] 23481    E    sleep 90 &
[3] 23487    E    sleep 70 &
$ fg 2
sleep 90
^C
[ctrlc()→ Soy el proceso con PID 22579 (./nivel5), el proceso en foreground es 23481 (sleep
90)]
[ctrlc()→ Señal 15 (SIGTERM) enviada a 23481 (sleep 90) por 22579 (./nivel5)]

[reaper()→ Proceso hijo 23481 en foreground (sleep 90) finalizado por señal 15]
$ jobs
[1] 23476    D    sleep 80
[2] 23487    E    sleep 70 &
$ fg 1
[internal_fg()→ Señal 18 (SIGCONT) enviada a 23476 (sleep 80)]
sleep 80
^Z
[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 23476 (sleep 80)]
[ctrlz()→ Señal 19 (SIGSTOP) enviada a 23476 (sleep 80) por 22579 (./nivel5)]
[2] 23476    D    sleep 80
$ jobs
[1] 23487    E    sleep 70 &
[2] 23476    D    sleep 80
$ fg 2
[internal_fg()→ Señal 18 (SIGCONT) enviada a 23476 (sleep 80)]
sleep 80
^C
[ctrlc()→ Soy el proceso con PID 22579 (./nivel5), el proceso en foreground es 23476 (sleep
80)]
[ctrlc()→ Señal 15 (SIGTERM) enviada a 23476 (sleep 80) por 22579 (./nivel5)]

[reaper()→ Proceso hijo 23476 en foreground (sleep 80) finalizado por señal 15]
$ jobs
[1] 23487    E    sleep 70 &
$ fg 1 #esperamos hasta que aparezca el prompt
sleep 70

[reaper()→ Proceso hijo 23487 en foreground (sleep 70) finalizado con exit code 0]
$

$ ps f
[execute_line()→ PID padre: 22579 (./nivel5)]
[execute_line()→ PID hijo: 24052 (ps f)]
PID TTY    STAT  TIME COMMAND
```



```
22515 pts/2  Ss   0:00 /bin/bash
22579 pts/2  S+   0:00 \_ ./nivel5
24052 pts/2  R+   0:00 \_ ps f
```

[reaper()→ Proceso hijo 24052 en foreground (ps f) finalizado con exit code 0]

\$./nivel5

[execute_line()→ PID padre: 22579 (./nivel5)]

[execute_line()→ PID hijo: 24060 (./nivel5)]

\$ ps f

[execute_line()→ PID padre: 24060 (./nivel5)]

[execute_line()→ PID hijo: 24063 (ps f)]

```
PID TTY  STAT  TIME COMMAND
22515 pts/2  Ss   0:00 /bin/bash
22579 pts/2  S+   0:00 \_ ./nivel5
24060 pts/2  S+   0:00 \_ ./nivel5
24063 pts/2  R+   0:00 \_ ps f
```

[reaper()→ Proceso hijo 24063 en foreground (ps f) finalizado con exit code 0]

\$ ^Z

[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 24060 (./nivel5)]

[ctrlz()→ Soy el proceso con PID 24060, el proceso en foreground es 0 ()]

[ctrlz()→ Señal 19 (SIGSTOP) no enviada por 22579 (./nivel5) debido a que su proceso en foreground es el shell]

[ctrlz()→ Señal 19 (SIGSTOP) no enviada por 24060 (./nivel5) debido a que no hay proceso en foreground]

\$ sleep 60

[execute_line()→ PID padre: 24060 (./nivel5)]

[execute_line()→ PID hijo: 24079 (sleep 60)]

^Z

[ctrlz()→ Soy el proceso con PID 22579, el proceso en foreground es 24060 (./nivel5)]

[ctrlz()→ Señal 19 (SIGSTOP) no enviada por 22579 (./nivel5) debido a que su proceso en foreground es el shell]

[ctrlz()→ Soy el proceso con PID 24060, el proceso en foreground es 24079 (sleep 60)]

[ctrlz()→ Señal 19 (SIGSTOP) enviada a 24079 (sleep 60) por 24060 (./nivel5)]

```
[1] 24079  D    sleep 60
```

\$ fg 1

sleep 60

^C

[ctrlc()→ Soy el proceso con PID 24060 (./nivel5), el proceso en foreground es 24079 (sleep 60)]

[ctrlc()→ Soy el proceso con PID 22579 (./nivel5), el proceso en foreground es 24060 (./nivel5)]

[ctrlc()→ Señal 15 (SIGTERM) no enviada por 22579 (./nivel5) debido a que su proceso en foreground es el shell]

[ctrlc()→ Señal 15 (SIGTERM) enviada a 24079 (sleep 60) por 24060 (./nivel5)]

[reaper()→ Proceso hijo 24079 en foreground (sleep 60) finalizado por señal 15]

\$ ^D

\$ ^D

\$ sleep 10 &

[1] 14933 E sleep 10 &

\$ fg 1

sleep 10

^Z

[ctrlz()→ Soy el proceso con PID 14839, el proceso en foreground es 14933 (sleep 10)]

[ctrlz()→ Señal 19 (SIGSTOP) enviada a 14933 (sleep 10) por 14839 (./nivel5)]

[1] 14933 D sleep 10

\$ fg 1

[internal_fg()→ Señal 18 (SIGCONT) enviada a 14933 (sleep 10)]

sleep 10

[reaper()→ Proceso hijo 14933 en foreground (sleep 10) finalizado con exit code 0]

\$

\$ sleep 15

^Z

[ctrlz()→ Soy el proceso con PID 14839, el proceso en foreground es 15159 (sleep 15)]

[ctrlz()→ Señal 19 (SIGSTOP) enviada a 15159 (sleep 15) por 14839 (./nivel5)]

[1] 15159 D sleep 15

\$ bg 1

[internal_bg()→ señal 18 (SIGCONT) enviada a 15159 (sleep 15 &)]

[1] 15159 E sleep 15 &

\$ ^Z

[ctrlz()→ Soy el proceso con PID 14839, el proceso en foreground es 0 ()]

[ctrlz()→ Señal 19 no enviada por 14839 (./nivel5) debido a que no hay proceso en foreground]

[reaper()→ Proceso hijo 15159 en background (sleep 15 &) finalizado con exit code 0]

Terminado PID 15159 (sleep 15 &) en jobs_list[1] con status 0

\$

Procesos en segundo plano (background)

En Linux podemos iniciar procesos en primer plano (foreground) o en segundo plano (background). Un proceso iniciado en foreground monopoliza la terminal e impide iniciar más procesos desde la misma. Un proceso en background, una vez iniciado deja de monopolizar la terminal, y devuelve el control al usuario (mostrando el prompt).

Es posible iniciar procesos en background utilizando el caracter ampersand **&** al final de la línea de comandos. Cuando un proceso es ejecutado con **&**, el shell devuelve el identificador de proceso (PID) y presenta nuevamente el *prompt*. A cada proceso en background se le asigna un identificador numérico del trabajo, que se muestra entre [].

```
uib ~$ gedit &
[1] 12692
uib ~$ sleep 40 &
[2] 12703
uib ~$
```

Comandos jobs, fg, bg. Señales SIGTSTP (^Z) y SIGCONT

Mediante el comando **jobs** podemos obtener el listado de trabajos, con sus identificadores, y así ver los procesos en segundo plano.

```
uib ~$ jobs
[1]- Ejecutando      gedit &
[2]+ Ejecutando      sleep 40
```

Podemos pasar un proceso de background a foreground mediante el comando **fg** y el identificador de trabajo (precedido opcionalmente del símbolo %):

```
uib ~$ fg %2
sleep 40
```

Un comando que se ejecuta en foreground es posible suspenderlo utilizando la combinación de teclas **Ctrl+Z** (que produce la señal **SIGTSTP**). El proceso detenido pasa a la lista de trabajos.

```
uib ~$ sleep 20
```

```
^Z
[3]+ Detenido      sleep 20
uib ~$
```

Utilizamos el comando **fg** y el identificador de trabajo (precedido opcionalmente del símbolo %) si deseamos volver a ejecutarlo en primer plano (internamente se le enviará la señal **SIGCONT**):

```
uib ~$ fg %3
sleep 20
...
```

Si lo que queremos es seguir ejecutándolo pero en segundo plano (background), en lugar del comando fg utilizamos **bg**:

```
uib ~$ sleep 20
^Z
[3]+ Detenido      sleep 20
uib ~$ bg %3
[3]+ sleep 20 &
uib ~$
```

Esto es realmente útil en situaciones como la ejecución de un script que se demora más de lo esperado y necesitamos ejecutarlo sin necesidad de una terminal, lo dejamos en segundo plano y seguirá corriendo sin necesidad de ningún entorno visual. Útil si solo disponemos de una única terminal y necesitamos realizar varias tareas simultáneamente.

Podemos forzar la finalización de un proceso ejecutándose en segundo plano mediante el comando **kill** y su identificador precedido de %:

```
uib ~$ sleep 30 &
[2] 3317
uib ~$ kill -9 %2
uib ~$ jobs
[1]- Ejecutando    gedit &
[2]+ Terminado    sleep 30
uib ~$
```

Recordad que mediante **Ctrl+C** se fuerza la finalización solo del proceso en primer plano.