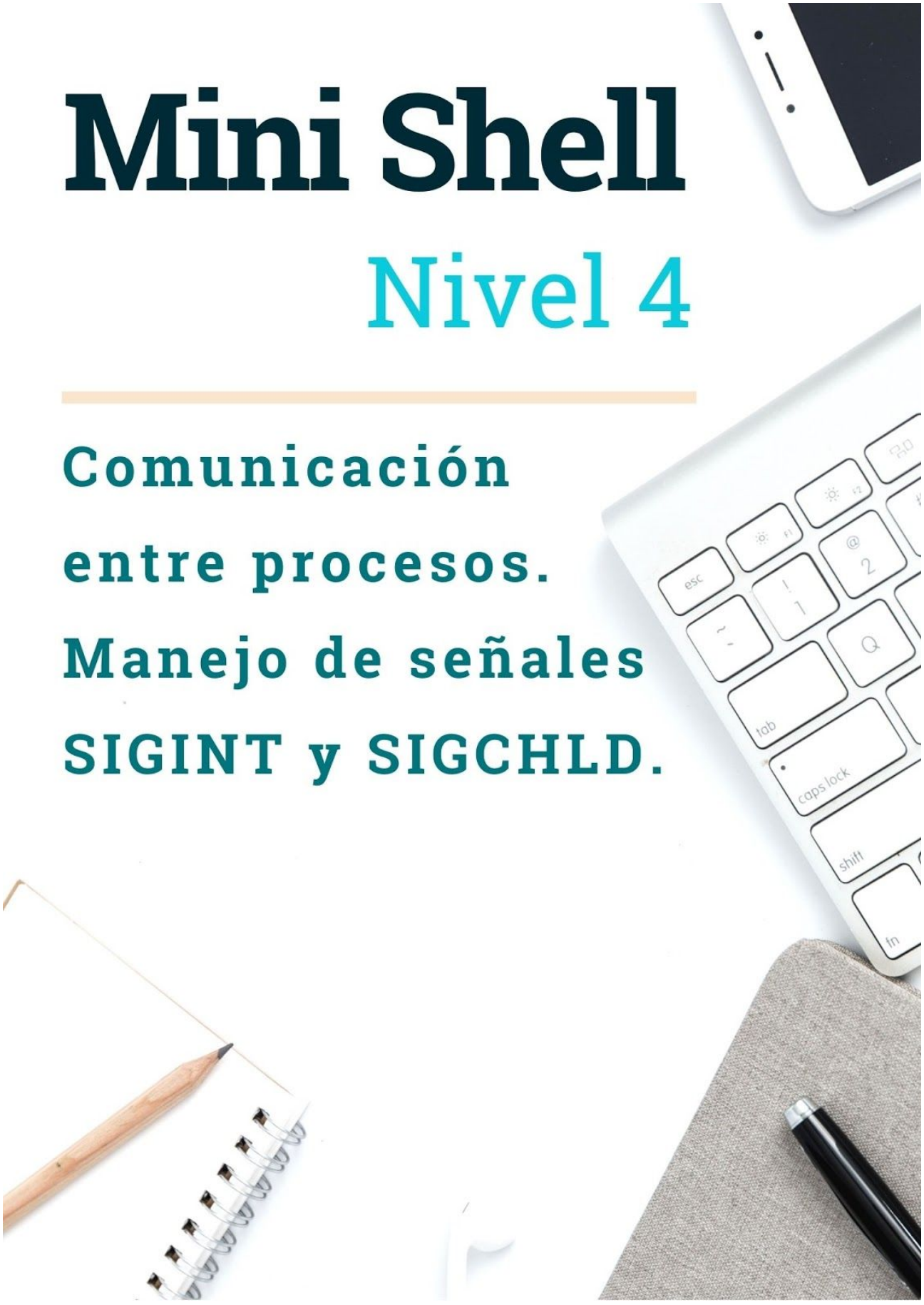
Mini Shell

Nivel 4

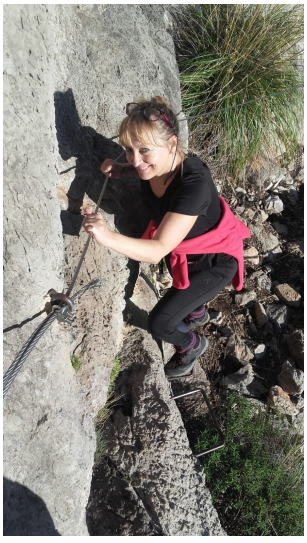
Comunicación entre procesos. Manejo de señales SIGINT y SIGCHLD.



Indice

En este nivel nuestro mini shell (proceso padre) gestionará diversas señales mediante unos manejadores propios.
En concreto será capaz de responder a la señal de interrupción **SIGINT**, generada por el usuario desde teclado al pulsar **Ctrl + C**, provocando que aborte el proceso hijo (comando externo) que estaba en ejecución en primer plano (sin abortar el proceso padre y por tanto permaneciendo dentro del mini shell).
También dispondrá de un manejador propio para la señal **SIGCHLD**, (al manejador le llamaremos **reaper()**) que se activará automáticamente en el momento en que se genera tal señal al finalizar un hijo. Será este manejador quien utilice la llamada al sistema **waitpid**(), informando por pantalla cuándo un hijo finaliza y con qué estado.
De esta manera, podremos indicar que el proceso padre (el mini shell) sólo esté en espera condicionado a que haya un proceso en ejecución en primer plano (no lo hará para los de segundo plano que crearemos más adelante) y además lo hará hasta que se produzca cualquier señal que afecte a éste (no solo la de **SIGCHLD** sino también la **SIGINT**, y más adelante también gestionará la **SIGTSTP** producida al pulsar el usuario las teclas **Ctrl + Z**).

¿Cuáles son los retos de este nivel?



En este nivel nuestro mini shell tendrá que disponer de unos manejadores propios para las siguientes señales:

- **SIGINT** (señal de interrupción generada por teclado, Ctrl+C)[1] y
- **SIGCHLD** (señal de parada o salida de un proceso hijo).

Y si hay un proceso en ejecución en primer plano esperará a que se produzca alguna señal que afecte a éste.

¿Qué conceptos trabajamos con estos retos?

- Conceptos sobre señales.
- Funciones **kill**(), **signal**(), **sigaction**(), **pause**(). Macros **SIG_IGN** y **SIG_DFL**

[1] En realidad en este nivel no necesitaríamos un manejador propio (podríamos indicar que el padre ignorase la señal y el hijo ejecutase la acción por defecto) pero lo dejaremos preparado para el siguiente nivel en que tendremos también procesos en ejecución en background y la posibilidad de pasar procesos a foreground o de reactivar los detenidos.

- Ejemplos de uso de las funciones **waitpid**(), **signal**(), **kill**(), **pause**() y **exit**()
- Comandos **ps**, **kill**, **pkill**

¿Cuáles son los detalles de implementación?

Hay que ampliar las siguientes funciones:

int execute_line(char *line);
En este nivel (todavía no podemos lanzar procesos en segundo plano) sigue existiendo un solo hijo, que será el del proceso relativo al comando ejecutándose en primer plano, pero ya dejaremos preparado un enterrador (función **reaper**()) que será quien manejará la señal de finalización (**SIGCHLD**) de cualesquiera de los hijos.
Ahora el padre (nuestro minishell), mientras no acabe el hijo (comando) que se ejecuta en primer plano, estará esperando mediante la función bloqueante **pause**() dentro del bucle condicionado a que haya un hijo ejecutándose en foreground. Pero no tendrá que esperar por los comandos en ejecución en segundo plano, solo por el del foreground.

Podremos controlar cual es el proceso en ejecución en primer plano con una variable global que guarde el PID del proceso en primer plano, inicializada a 0 (0 indicaría que no hay ningún proceso en primer plano en ejecución en ese momento). El padre le dará el valor del PID a esa variable cuando se lance un hijo que ejecute un comando en foreground (en execute_line()). El reaper pondrá de nuevo a 0 esa variable cuando este proceso en foreground finalice su ejecución (así el padre dejará de estar en espera).

Además del PID del proceso, podemos también almacenar su estado ('N': ninguno, 'E': Ejecutándose y 'D': Detenido, 'F': Finalizado) y, también la línea de comandos asociada[2]. Para ello podemos utilizar la siguiente estructura de datos:

```
struct info_process {
  pid_t pid;
  char status; // 'N', 'E', 'D', 'F'
  char cmd[COMMAND_LINE_SIZE]; // línea de comando
};
```

[2] Nos será útil para imprimir el nombre del comando asociado a un PID determinado y así hacer un seguimiento más claro de qué procesos están en ejecución (convendrá eliminarle el \n).

Así tendremos un array de trabajos[3] global:

static struct info_process jobs_list[N_JOBS][4]

para almacenar los datos de los procesos en ejecución y/o detenidos.

La posición 0 del array será para guardar los datos del proceso en **foreground**, el resto de posiciones los utilizaremos más adelante para los trabajos en background o detenidos.

De momento en la ejecución del padre, jobs_list[0].pid cogerá el valor del PID del único proceso hijo que se ejecute. Más adelante cuando analicemos una línea de comandos para determinar si se ha de ejecutar o no en segundo plano (si lleva o no el símbolo & al final de la línea), **jobs_list[0].pid sólo cogerá el valor del PID del proceso hijo si éste se ha de ejecutar en primer plano, y valdrá 0 en caso contrario**. Antes de llamar a parse_args() guardaremos en jobs_list[0].cmd el valor de la línea de comandos sin trocear.

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 pondrá la variable **jobs_list[0].pid** de nuevo a 0.

El **pause**() del padre sólo se ejecutará mientras haya un proceso ejecutándose en foreground (**jobs_list[0].pid** > 0).

En cuanto al proceso hijo:

- reseteará la señal **SIGCHLD** asociándole la acción por defecto **SIG_DFL** (ya que el comando a ejecutar podría ser la llamada a otro programa que también manejase sus propias señales).

[3] La diferencia entre un "job" y un "process" es que los "jobs" son obligatoriamente iniciados desde una terminal y están asociados a ella (son procesos "hijos" de la terminal y tienen el mismo identificador de grupo, pgid, que puede ser obtenido mediante la función getpgid() y modificado con la función setpgid()). Se puede ver también el identificador de grupo desde la terminal con $ps o pid,ppid,pgid,comm. Ejemplo:

```
$ps -t 0 fo pid,ppid,pgid,comm
  PID  PPID  PGID COMMAND
 3254  3212  3254 bash
12151  3254 12151  \_ nivel4
12253 12151 12151      \_ nivel4
12547 12253 12151          \_ sleep
```

[4] Podéis darle el valor 64 a NJOBS

- ignorará la señal **SIGINT** (haremos que el padre le envíe la señal **SIGTERM**, que tiene también el efecto de abortar el proceso, mediante la función kill() en el manejador ctrlc())[5]

void reaper(int signum);
Manejador propio para la señal SIGCHLD (señal enviada a un proceso cuando uno de sus procesos hijos termina), también denominado enterrador de hijos.

Conviene volver a poner la instrucción **signal**() asociándola a esta misma función porque en algunos sistemas una vez capturada por primera vez, se restaura al uso de la acción por defecto.

Como se ha indicado anteriormente, el enterrador controlará si el hijo que acaba es el que se ejecuta en primer plano, y en tal caso pondrá la variable **jobs_list[0].pid** de nuevo a 0. El status pasará a ser 'F' y borramos el cmd asociado a ese proceso.
Podemos analizar el valor que devuelve **waitpid**()[6] para saber qué hijo termina:

```
while ((ended=waitpid(-1, &status, WNOHANG)) > 0 {
...
}
```

En este nivel mostraremos un mensaje indicando el PID del hijo que ha finalizado y si ha sido por una señal.
Podéis utilizar las macros WIFEXITED(status), WEXITSTATUS(status), WIFSIGNALED(status) y WTERMSIG(status) para mostrar el mensaje apropiado, o si no al menos imprimir el status obtenido por waitpid().

void ctrlc(int signum); //Manejador propio para la señal SIGINT (Ctrl+C).

Conviene volver a poner la instrucción signal asociándola a esta misma función porque en algunos sistemas una vez capturada por primera vez, se restaura a la acción por defecto.

[5] Podéis probar de asociarle la acción por defecto **SIG_DFL** y comprobar que en tal caso el hijo finaliza por la señal 2 (SIGINT) en vez de por la 15 que le envía el padre (SIGTERM)
[6] El parámetro convalor -1 sirve para cualquier hijo, no especificamos un PID concreto.
Si deseamos controlar qué ha hecho finalizar el proceso (por ejemplo si ha sido la recepción de la señal 2 SIGINT, o la señal 15 SIGTERM), en vez de NULL podemos poner &estado (siendo estado una variable local de tipo int).
WNOHANG hace que el proceso padre no quede bloqueado mientras los hijos van acabando (nuestro shell no tendrá que quedar inactivo esperando por los procesos que se ejecuten en background, sólo por el de foreground, y eso lo controlaremos desde execute_line() llamando a **pause**(), condicionado a que se trate del proceso en primer plano)

Dado que al pulsar Ctrl+C, TODOS los procesos en ejecución en el terminal (incluídos los que ejecutemos más adelante en backgroung) reciben la señal 2 (SIGINT), y hemos indicado a los hijos que la ignoren (en el execute_line()) para que no afecte también a los de segundo plano, enviaremos a través del padre una señal diferente, SIGTERM, para controlar la terminación sólo del proceso en primer plano, siempre y cuando éste no sea nuestro propio mini shell[7].

El pseudocódigo para esta función sería:

```
Si (hay un proceso en foreground) entonces //jobs_list[0].pid > 0
  Si (el proceso en foreground NO es el mini shell) entonces
    //ya que puedo haber ejecutado el minishell  dentro del minishell
    enviarle la señal SIGTERM y provisionalmente notificarlo por pantalla.
  si_no error ("Señal SIGTERM no enviada debido a que el proceso en foreground es el shell")
  fsi
si_no error ("Señal SIGTERM no enviada debido a que no hay proceso en foreground")
fsi
```

Del mini shell sólo se ha de salir con **exit** o **Ctrl+D** !!!
Ctrl+C no ha de abortar el mini shell ni tampoco cuando se ejecuta como hijo de sí mismo.
Podéis realizar un salto de línea seguido de un fflush(stdout) para forzar el vaciado del buffer de salida.

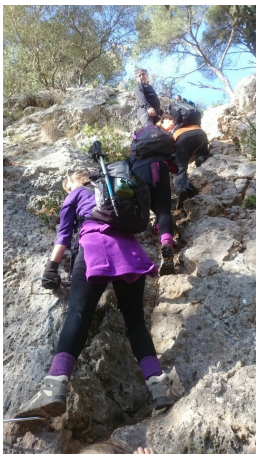
Al **main** tendremos que añadirle que escuche las señales SIGCHLD y SIGINT:

signal(SIGCHLD, reaper);
//llamada al enterrador de zombies cuando un hijo acaba (señal SIGCHLD)

signal(SIGINT, ctrlc);
//SIGINT es la señal de interrupción que produce Ctrl+C

También en el main inicializaremos los campos de jobs_list[0].

[7] Lo sabemos comparando el cmd con el comando de ejecución de nuestro programa que podemos tener guardado en una constante, ej. "./nivel4" o mejor aún obtenerlo desde consola (parámetro argv[0] del main)



¿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

¿Cómo comprobamos que hemos superado este nivel?

- Podéis ejecutar vuestro programa **nivel4.c** e ir escribiendo en la línea de comandos los ejemplos que se exponen a continuación, comparando vuestros resultados con los mostrados aquí[8]

```
uib:$ ./nivel4
$ ^C
[ctrlc()→ Soy el proceso con PID 8334, el proceso en foreground es 0 ()]
[ctrlc()→ Señal 15 no enviada por 8334 debido a que no hay proceso en foreground]
$ sleep 2[9]
[execute_line()→ PID padre: 8334 (./nivel4)]
[execute_line()→ PID hijo: 8409 (sleep 2)]
[reaper()→ Proceso hijo 8409 (sleep 2) finalizado con exit code 0]
$ sleep 60[10]
[execute_line()→ PID padre: 8334 (./nivel4)]
[execute_line()→ PID hijo: 8423 (sleep 60)]
^C
```

[8] Es aconsejable no utilizar printf() o fprintf() dentro de un manejador. En su lugar podemos formar la cadena a visualizar con sprintf() e imprimirla por pantalla con la llamada a sistema write().
Ejemplo:

```
char mensaje[1200];
sprintf(mensaje, "[reaper()→ Proceso hijo %d (%s) finalizado con exit code %d]\n", ended, jobs_list[0].cmd,
WEXITSTATUS(estado));
write(2, mensaje, strlen(mensaje)); //2 es el flujo stderr
```

[9] Esperamos los 2”
[10] Pulsamos Ctrl+C al cabo de unos segundos y reaparece nuestro prompt

```
[ctrlc()→ Soy el proceso con PID 8334 (./nivel4), el proceso en foreground es 8423 (sleep 60)]
[ctrlc()→ Señal 15 enviada a 8423 (sleep 60) por 8334 (./nivel4)]
[reaper()→ Proceso hijo 8423 (sleep 60) finalizado por señal 15]
$ #sigo dentro del mini shell[11]
$ exit
```

```
uib:$ ./nivel4
$ ps f
[execute_line()→ PID padre: 8334 (./nivel4)]
[execute_line()→ PID hijo: 8678 (ps f)]
  PID TTY      STAT   TIME COMMAND
 3254 pts/0    Ss     0:00 /bin/bash
 8334 pts/0    S+     0:00  \_ ./nivel4
 8678 pts/0    R+     0:00      \_ ps f
[reaper()→ Proceso hijo 8678 (ps f) finalizado con exit code 0]
$ ./nivel4[12]
[execute_line()→ PID padre: 8334 (./nivel4)]
[execute_line()→ PID hijo: 8778 (./nivel4)]
$ ps f
[execute_line()→ PID padre: 8778 (./nivel4)]
[execute_line()→ PID hijo: 8784 (ps f)]
  PID TTY      STAT   TIME COMMAND
 3254 pts/0    Ss     0:00 /bin/bash
 8334 pts/0    S+     0:00  \_ ./nivel4
 8778 pts/0    S+     0:00      \_ ./nivel4
 8784 pts/0    R+     0:00          \_ ps f
[reaper()→ Proceso hijo 8784 (ps f) finalizado con exit code 0]
$ sleep 60
[execute_line()→ PID padre: 8778 (./nivel4)]
[execute_line()→ PID hijo: 8820 (sleep 60)]
^C
[ctrlc()→ Soy el proceso con PID 8778 (./nivel4), el proceso en foreground es 8820 (sleep 60)]
[ctrlc()→ Soy el proceso con PID 8334 (./nivel4), el proceso en foreground es 8778 (./nivel4)]
[ctrlc()→ Señal 15 enviada a 8820 (sleep 60) por 8778 (./nivel4)]
[ctrlc()→ Señal 15 no enviada por 8334 (./nivel4) debido a que su proceso en foreground es el shell]
[reaper()→ Proceso hijo 8820 (sleep 60) finalizado por señal 15]
$ ps f
```

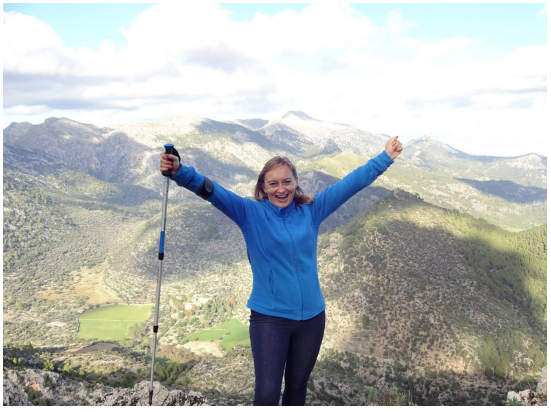
[11] (Al pulsar Ctrl+C se debe detener el comando en ejecución pero debe de aparecer el prompt del mini shell porque el padre sigue ejecutándose. En el siguiente nivel podremos ejecutar procesos en background, manejando así las señales de espera de varios hijos y la interrupción del proceso en foreground).
[12] Lanzamos de nuevo el ejecutable nivel4 pero como hijo del propio nivel4

```
[execute_line()→ PID padre: 8778 (./nivel4)]
[execute_line()→ PID hijo: 8879 (ps f)]
  PID TTY      STAT   TIME COMMAND
 3254 pts/0    Ss     0:00 /bin/bash
 8334 pts/0    S+     0:00  \_ ./nivel4
 8778 pts/0    S+     0:00      \_ ./nivel4
 8879 pts/0    R+     0:00          \_ ps f
[reaper()→ Proceso hijo 8879 (ps f) finalizado con exit code 0]
$ exit[13]
[reaper()→ Proceso hijo 8778 (./nivel4) finalizado con exit code 0]
$ ps f
[execute_line()→ PID padre: 8334 (./nivel4)]
[execute_line()→ PID hijo: 8924 (ps f)]
  PID TTY      STAT   TIME COMMAND
 3254 pts/0    Ss     0:00 /bin/bash
 8334 pts/0    S+     0:00  \_ ./nivel4
 8924 pts/0    R+     0:00      \_ ps f
[reaper()→ Proceso hijo 8924 (ps f) finalizado con exit code 0]
$ exit[14]
uib:~/practicasSO/SOI/miP2/niveles$
```

```
uib:$ ./nivel4
$ ./contar 10[15]
[execute_line()→ PID padre: 9831 (./nivel4)]
[execute_line()→ PID hijo: 9839 (./contar 10)]
0
1
2
3
^C
[ctrlc()→ Soy el proceso con PID 9831 (./nivel4), el proceso en foreground es 9839
(./contar 10)]
[ctrlc()→ Señal 15 enviada a 9839 (./contar 10) por 9831 (./nivel4)]
[reaper()→ Proceso hijo 9839 (./contar 10) finalizado por señal 15]
$ ./contarSIGINTDFL 10[16]
[execute_line()→ PID padre: 9831 (./nivel4)]
[execute_line()→ PID hijo: 9873 (./contarSIGINTDFL 10)]
```

[13] Salimos de nivel4 hijo
[14] Habíamos salido de nivel4 como hijo de nuestro mini shell, ahora saldremos de nivel4 padre y volveremos al prompt del bash
[15] Programa externo, sin especificación de manejadores de señal, que va imprimiendo desde 0 hasta el nº especificado como parámetro. En tal caso nuestro mini shell le envía la señal 15 (SIGTERM) al ser pulsado Ctrl+C, y finaliza debido a esa señal
[16] Programa externo, con signal(SIGINT, SIG_DFL), que va imprimiendo desde 0 hasta el nº especificado como parámetro. En tal caso nuestro mini shell le envía la señal 15 (SIGTERM) al ser pulsado Ctrl+C, pero finaliza por la señal 2 gracias a la acción por defecto SIG_DFL

```
0
1
2
^C
[ctrlc()→ Soy el proceso con PID 9831 (./nivel4), el proceso en foreground es 9873
(./contarSIGINTDFL 10)]
[ctrlc()→ Señal 15 enviada a 9873 (./contarSIGINTDFL 10) por 9831 (./nivel4)]
[reaper()→ Proceso hijo 9873 (./contarSIGINTDFL 10) finalizado por señal 2]
```



Observaciones:

- Comprobad que antes de la inclusión de las librerías tenéis incorporada la siguiente directiva para el preprocesador, por cuestiones de portabilidad:
  #define _POSIX_C_SOURCE 200112L[17]

Anexo 1: Conceptos sobre señales. Funciones kill(), signal(), sigaction(), pause(). Macros SIG_IGN y SIG_DFL

Las señales son sucesos asíncronos (que no se sabe cuándo se van a producir) y que afectan al comportamiento de un proceso. Por su naturaleza, los procesos son interrumpidos y forzados a manejarlas inmediatamente. Cada señal se identifica con un número entero al igual que un nombre simbólico, que empieza por SIG. Con el comando `$ kill -l` podemos ver las señales que se utilizan en nuestro sistema:

```
uib$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL      10) SIGUSR1
11) SIGSEGV      12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
```

[17] POSIX, feature_test_macros

```
21) SIGTTIN      22) SIGTTOU      23) SIGURG       24) SIGXCPU      25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF      28) SIGWINCH     29) SIGIO        30) SIGPWR
31) SIGSYS       34) SIGRTMIN     35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3
38) SIGRTMIN+4   39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12  47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14  51) SIGRTMAX-13  52) SIGRTMAX-12
53) SIGRTMAX-11  54) SIGRTMAX-10  55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7
58) SIGRTMAX-6   59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

En un programa en C, la llamada al sistema **kill**() sirve para enviar señales (cualquier señal) a procesos.
Las señales también se pueden ocasionar mediante interrupciones provocadas desde teclado[18]. Ejemplos:

- **Ctrl+C** origina la señal **SIGINT** que aborta un proceso,
- **Ctrl+** origina la señal **SIGQUIT** que finaliza un proceso y genera un volcado de memoria del proceso *(core dump)*, útil para tareas de depuración,
- **Ctrl+Z** origina la señal **SIGTSTP** que suspende un proceso y lo manda a segundo plano *(background)*.

También hay eventos que producen señales. Por ejemplo, cuando un hijo acaba con un exit, el núcleo envía la señal **SIGCHLD**.
Y también errores, como por ejemplo, una violación de segmento (se genera la señal **SIGSEGV**) o una división por cero.

Las funciones **signal**() y **sigaction**() (versión POSIX de signal()) capturan una señal y le asocian la acción a realizar. La acción puede ser una función propia, o la macro **SIG_IGN** que ignora la señal o la macro **SIG_DFL** que realiza la acción por defecto asociada a la señal.
Hay dos señales que no pueden ser interceptadas ni manipuladas: **SIGKILL** (mata un proceso) y **SIGSTOP**.
Una limitación importante de las señales es que no tienen prioridades relativas, es decir, si dos señales llegan al mismo tiempo a un proceso puede que sean tratadas en cualquier orden, no podemos asegurar la prioridad de una en concreto. Otra limitación es la imposibilidad de tratar múltiples señales iguales, el proceso funcionará como si hubiera recibido sólo una.

Cuando queremos que un proceso espere a que le llegue una señal, podemos usar la función **pause**(). Esta función provoca que el proceso (o thread) en cuestión "duerma" hasta que le llegue una señal. Para capturar esa señal, el proceso deberá haber establecido un tratamiento de la misma con la función **signal**() o **sigaction**(). Ej.:

[18] En este caso la reciben TODOS los procesos que cuelgan del terminal

```
signal(SIGCHLD, reaper);
/* Cuando un hijo acaba se genera señal SIGCHLD.
 Asociamos esa señal con la función propia reaper() que es un enterrador de zombies
 que contendrá el wait() */

signal(SIGINT, ctrlc);
/* Cuando se pulsa Ctrl+C se genera la señal SIGINT..
 Asociamos esa señal con la función propia ctrlc() que enviará la señal al proceso
 en primer plano (foreground) y mostrará de nuevo el prompt */
```

En algunos entornos, cuando se recibe una señal y se ejecuta nuestro manejador, se restaura automáticamente la acción por defecto SIG_DFL. Por ello, puede ser necesario que nuestra función vuelva a ponerse ella misma como manejador de esa señal.. Ej:

```
void reaper(int signum) {
 ...
 signal(SIGCHLD, reaper);
}
```

Anexo 2: Ejemplos de uso de las funciones waitpid(), signal(), kill(), pause(), exit().

```
/* uso_signal1.c - ejemplo manejador SIGINT - Adelaida Delgado*/
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

// Esta función es la que vamos a usar como manejador de la señal SIGINT
void despedida(int signum) {
  pid_t pid;

  signal(SIGINT, SIG_DFL);
 // Provocará la acción por defecto cuando enviemos la señal con kill()
  puts("Me has interrumpido al pulsar Ctrl+C\n");
  pid=getpid();
  printf("Así que yo, proceso %d, me despido\n", pid);

 if (kill(pid, SIGINT)==0) { // Enviamos la señal SIGINT al proceso
    fprintf(stderr, "\nSeñal %d enviada a %d", signum,  getpid());
 } else {
    perror("kill");
```

```
    exit(-1);
  }
}

int main() {
  // Asociamos la señal SIGINT (Ctrl+C) con la funcion despedida()
  signal(SIGINT, despedida);
  // Comenzamos un bucle que hará que el programa muestre sin
  // parar el mensaje "Me aburro esperando una señal"
  while(1) {
    printf("Me aburro esperando una señal\n");
  }
}
```

Modificar el programa anterior substituyendo el bucle del main por pause() y observar qué sucede.

Vamos a ver un ejemplo con dos manejadores de señal, uno para la señal SIGINT (generada al pulsar Ctrl+C) y otro para la señal SIGQUIT (generada al pulsar Ctrl+4 o Ctrl+\)

```
// uso_signal.c - Ejemplo de manejadores de señales SIGINT y SIGQUIT - Adelaida Delgado

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void ctrlc() {
  signal(SIGINT, ctrlc);
  puts("\nCtrl+C pulsado");
}

void ctrl4() {
  puts("\nCtrl+\ pulsado: FIN");
  exit(EXIT_SUCCESS);  //provoca la salida del programa
}

int main() {
  signal(SIGINT, ctrlc); //Ctrl+C
  signal(SIGQUIT, ctrl4); //Ctrl+\
  //sleep(1);
  puts("\nPrueba a pulsar Ctlr+C y comprobarás que no funciona.");
  puts("\nPara salir hay que pulsar Ctrl+\");
```

```
  while(1);
}
```

En el programa anterior no iría bien sustituir el bucle infinito por un pause() ya que el proceso acabaría en cuanto se produjera una de las dos señales (probadlo!!!). Probad también de asociar los manejadores por defecto o de ignorar las señales.

Vamos ahora a ver cómo programar un enterrador de hijos zombies y cómo forzar a que el padre espere a que acabe una cantidad determinada de hijos

```
/* ejenterrador.c - ejemplo de función enterrador - Adelaida Delgado */

#define NUM_HIJOS 10

#include <signal.h>  // signal()
#include <stdio.h>   // print(), fprintf()
#include <stdlib.h>  // exit()
#include <unistd.h>  // fork(), getpid(), sleep(), pause()
#include <sys/wait.h>  // waitpid()

static int acabados=0; //variable global para contabilizar procesos acabados

// Creamos la función enterrador que tratará los procesos zombies:
void reaper(int signum) {
  pid_t ended;

  printf("reaper()→ Recibida señal %d.\n", signum); // la señal 17 es SIGCHILD

  while ((ended=waitpid(-1, NULL, WNOHANG))>0) {
     acabados++;
     printf("reaper()→ Enterrado proceso con PID %d. Total acabados: %d\n", ended,
acabados);
  }
}

int main() {
  int hijo;
  pid_t pid;
  //Asociamos la función reaper a la señal de finalización de proceso:
  signal(SIGCHLD, reaper);

  for (hijo=1; hijo<=NUM_HIJOS; hijo++) {
     pid=fork();
```

```
    if (pid==0) { //hijo (!pid)
        printf("Soy el hijo nº %d, mi PID es %d\n", hijo, getpid());
        sleep(3);
        exit (EXIT_SUCCESS);
    } else if (pid<0) {
        fprintf(stderr, "Error en iteración %d\n", hijo);
        exit(-1);
    }
    sleep(1);
 }

 //Tendremos que permitir que el padre espere por todos los hijos:
 while (acabados < NUM_HIJOS) {
    pause();
 }

 printf("Total de procesos terminados: %d\n", acabados);
return EXIT_SUCCESS;
}
```

Ejemplo de ejecución del programa anterior:

```
uib ~$ ./ejenterrador
Soy el hijo nº 1, mi PID es 5542
Soy el hijo nº 2, mi PID es 5543
Soy el hijo nº 3, mi PID es 5544
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5542. Total acabados: 1
Soy el hijo nº 4, mi PID es 5545
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5543. Total acabados: 2
Soy el hijo nº 6, mi PID es 5547
Soy el hijo nº 5, mi PID es 5546
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5544. Total acabados: 3
Soy el hijo nº 7, mi PID es 5548
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5545. Total acabados: 4
Soy el hijo nº 8, mi PID es 5549
Soy el hijo nº 9, mi PID es 5550
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5547. Total acabados: 5
reaper()→ Recibida señal 17.
```

```
reaper()→ Enterrado proceso con PID 5546. Total acabados: 6
Soy el hijo nº 10, mi PID es 5552
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5548. Total acabados: 7
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5549. Total acabados: 8
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5550. Total acabados: 9
reaper()→ Recibida señal 17.
reaper()→ Enterrado proceso con PID 5552. Total acabados: 10
Total de procesos acabados: 10
```

Anexo 3: Comandos ps, kill y pkill

Con `$ ps -s` podemos ver para cada proceso las señales que están pendientes, ignoradas, bloqueadas, y capturadas.
Desde la línea de comandos también podemos enviar señales a procesos que se están ejecutando en primer plano en otro terminal mediante el comando `$ kill <-señal> <PID>`. Si se omite <señal> entonces se envía la señal **SIGTERM**. La señal puede indicarse tanto por su nombre como por su número.

Ejemplo:
Ejecutamos el siguiente programa (un contador infinito)[19] y lo ejecutamos. Desde otra consola vamos enviándole señales. El programa no contiene manejadores de señales, pero podemos probar de ejecutar Ctrl+Z (o enviarle directamente la señal SIGTSTP desde otro terminal para detenerlo, y luego enviarle un SIGCONT para reanudarlo), también ejecutar Ctrl+C y si no responde enviarle un SIGKILL desde otro terminal.

```c
/* buclecontador.c - contador infinito - Adelaida Delgado */

#include <stdio.h>
#include <unistd.h>

void main ()
{
  int i=0;
  while (1) {
     printf("%d\n",i);
     i++;
     sleep(1);
  }
```

[19] En los procesos asociados al comando sleep aunque los detengamos con Ctrl+Z sigue en marcha el contador (también en el bash…). Si queremos comprobar la reanudación de un proceso, en el punto en el que se había detenido, es mejor lanzar un programa que por ejemplo imprima un bucle segundo a segundo.

```
}
```

Ejecución en terminal 1:	Ejecución en terminal 2:
uib:~/practicasSO/SOI/ejemplos$ ./buclecontador 0 1 2 3 4 5 6 7 8 9 10 11 12	**uib:~$ ps -a** PID TTY TIME CMD 23175 pts/1 00:00:00 buclecontador 23189 pts/2 00:00:00 ps **uib:~$ ps -t 1 f** PID TTY STAT TIME COMMAND 18863 pts/1 Ss 0:00 bash 23175 pts/1 S+ 0:00 _ ./buclecontador **uib:~$ kill -SIGTSTP 23175**
[1]+ Detenido ./buclecontador	**uib:~$ kill -SIGCONT 23175**
uib:~/practicasSO/SOI/ejemplos$ 13 14 15 16 17 18 19 20	**uib:~$ kill -9 23175 #SIGKILL**
[1]+ Terminado (killed) ./buclecontador **uib:~/practicasSO/SOI/ejemplos$**	

Si en vez de referenciar los procesos por su PID queremos hacerlo por su nombre entonces usaríamos el comando $ pkill *<-señal> <nombre_proceso>*. Ejemplo: $ pkill -9 buclecontador