Universidad
Rey Juan Carlos

Grado en Ingeniería de la Ciberseguridad

Sistemas Operativos

Práctica 2: Minishell

Redactado por: **Antonio Almeida Romera**
José Manuel Jerónimo Rodríguez

17 de diciembre de 2023

Índice

1. Descripción

1.1. Estructura del proyecto

En cuanto a la estructura del proyecto se refiere, hemos decidido modular el código en distintos archivos para una mayor legibilidad y facilidad a la hora de programar. El proyecto se divide en:

- `job.h-job.c`: Contiene un *struct* llamado `job_t` que agrupa la información de cada *job*. Además, se definen todas las funciones asociadas a este *struct*.
- `stack.h-stack.c`: Es una pila de procesos. Cada uno de los nodos representa un *job*. A esta pila, además de las funciones básicas de esta estructura, se le añaden otras funciones relevantes en la gestión de *jobs*.
- `background.h-background.c`: Estos archivos contienen todas las funciones relativas al manejo de procesos en *background*.
- `exec.h-exec.c`: Contiene funciones para ejecutar una línea de comandos y para el manejo de señales.
- `builtin.h-builtin.c`: Contiene las implementaciones de los mandatos internos `cd`, `jobs` y `fg`
- `redirect.h-redirect.c`: Estos archivos están destinados a contener funciones relativas a las redirecciones de `STDIN`, `STDOUT` y `STDERR`
- `utils.h-utils.c`: Contiene variables para asignar colores y la función para mostrar el *prompt*.
- `minishell.c`: Es el inicio del flujo del programa. Llama al *prompt* y realiza la lectura de la línea de comandos.

1.2. Ejecución de líneas de 1 a n comandos

Para ejecutar los comandos en todas sus longitudes empleamos la función *exec_line* ubicada en la librería *exec.h*. Esta función recibe como parámetros la estructura tline completa que inserta el usuario en consola. Esta función realiza lo siguiente:

1. Reserva espacio con *malloc* para un matriz de pipes y para el array de pids
2. Realiza la gestión de redirecciones y del background lo cuál se comentará con detalle más adelante.

3. Se inicializan los pipes de la matriz comprobando si ocurre algún error en el proceso. La matriz de pipes será de $N \times 2$.

4. A continuación, empieza el bucle que ejecuta cada uno de los comandos de la línea. Lo primero que se hace dentro del bucle es crear un hijo que ejecute ese comando. Lo siguiente es comprobar si es el hijo o el padre el que tiene la CPU mediante un bloque condicional:

   a) Si es el hijo entonces redireccionamos los descriptores de fichero estándar y las entradas y salidas de los pipes como corresponda.

      1) Si el hijo no es el primer comando leo la entrada del extremo de salida pipe $i-1$ mediante `dup2(pipes[i - 1][0], STDIN\_FILENO);`
      2) Si el hijo no es el último comando escribo la salida del extremo de entrada pipe i mediante `dup2(pipes[i][1], STDOUT\_FILENO);`
      3) Se cierran ambos extremos de todos los pipes en cada proceso hijo
      4) Finalmente se ejecuta la línea con *execvp*

   b) Si es el padre lo que se hace es almacenar el pid de cada hijo en cada iteración en un array llamado *pids*. Esto con el propósito de hacer el wait de cada hijo más adelante.

5. Una vez fuera del bucle, se cierran ambos extremos de todos los pipes en el proceso padre

6. Se restauran los STDs (esto está relacionado con las redirecciones y se explicará más adelante)

7. Por último si la línea no está en background se espera por cada pid contenido en el array de pids con un wait bloqueante, de esta manera `waitpid(pids[i], NULL, 0);`. El caso de background se explica más adelante.

8. El espacio reservado para la matriz de pipes y el array de pids es liberado con *free*

1.3. Redirecciones

Para realizar las redirecciones de `STDIN`, `STDOUT` y `STDERR` usamos las funciones definidas en `redirect.c`. En primer lugar, tenemos la función `redirect_std_fd` que tiene como objetivo redireccionar un STD a un descriptor de fichero pasado como argumento. Para ello realizamos con `dup2` la redirección y se cierra el descriptor de fichero antiguo:

```
int redirect_std_fd(int std, int fd) {
  int newfd;
  newfd = dup2(fd, std);
  if (close(fd) == -1) {
    return -1;
  }
  return newfd;
}
```

Listing 1: Extraído del archivo `redirect.c`

Como necesitabamos hacer redirecciones de los STDs hacia archivos, definimos otra función que hiciese uso de la función anteriormente mencionada. La función `redirect_std_file` redirecciona un STD hacia un fichero. Primero se obtiene el descriptor de fichero, después se guarda el descriptor de fichero original y se llama a `redirect_std_fd`.

```
int redirect_std_file(int std, char* file) {
  int fd, saved_fd;
  if (std ==  STDIN_FILENO) {
    fd = open(file, O_RDONLY);
  } else if (std == STDOUT_FILENO || std == STDERR_FILENO) {
    fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
  }
  if (fd < 0) {
    return -1;
  }
  saved_fd = dup(std);
  if (redirect_std_fd(std, fd) == -1) {
    return -1;
  }
  return saved_fd;
}
```

Listing 2: Extraído del archivo `redirect.c`

Con esto ya teniamos la funcionalidad implementada, pero quisimos añadirle más facilidad a la hora de redireccionar, por lo tanto, hicimos una función para cada tipo de STD que hiciese uso de las funciones anteriormente mencionadas:

```
int redirect_input(char *file) {
  return redirect_std_file(STDIN_FILENO, file);
}

int redirect_output(char *file) {
  return redirect_std_file(STDOUT_FILENO, file);
}

int redirect_error(char *file) {
  return redirect_std_file(STDERR_FILENO, file);
}
```

Listing 3: Extraído del archivo `redirect.c`

Después de tener todo implementado, se debe realizar las redirecciones pedidas[1]:

```
    if (line->redirect_input != NULL) {
            if ((saved_std[STDIN_FILENO] = redirect_input(line->redirect_input)
   ) == -1) {
                fprintf(stderr, RED "[!] Error: %s" RESET, strerror(errno));
```

[1]Se muestra solo la redirección de `STDIN` para evitar repetición de código en la memoria. Las otras dos redirecciones son iguales en cuanto a implmentación.

```
                exit(EXIT_FAILURE);
            }
    }
```

Listing 4: Extraído de la función `exec_line` en el archivo `exec.c`

Cuando el proceso hijo termine, se ha de devolver el flujo de los STD a sus estados anteriores, es por ello que guardábamos los descriptores de ficheros originales:

```
  // Si se ha redireccionado el input, volver al estado original
    if (line->redirect_input != NULL){
      if (redirect_std_fd(STDIN_FILENO, saved_std[STDIN_FILENO]) == -1){
      fprintf(stderr, "[!] Error: %s", strerror(errno));
      exit(EXIT_FAILURE);
    }
  }
```

Listing 5: Extraído de la función `exec_line` en el archivo `exec.c`

1.4. Procesos en background

La gestión de los procesos en *background* abarca los archivos `job.c`, `stack.c` y `background.c`. Los tres ficheros están relacionados entre si y están destinados al mismo objetivo.

1.4.1. Job

En primer lugar, se define en `job.h` la estructura usada como proceso/*job*. Esta estructura guarda la línea de comando ejecutada, una lista de los PIDs de la línea y el número de PIDs:

```
typedef struct {
    pid_t pid[100];
    int index;
    char command[1024];
} job_t;
```

Listing 6: Extraído del archivo `job.h`

Todas las funciones relacionadas con el acceso y modificación a los datos de esta estructura están definidas en el archivo `job.c`.

1.4.2. Stack

En los archivos `stack.c-stack.h` se define la pila de procesos llamada `stackJobs_t`. Es una pila implementada con una lista doblemente enlazada y un puntero al inicio y al final de ella.

Como estructura de nodo usa `job_t`:

```
typedef struct node {
    job_t job;
    struct node *next;
    struct node *prev;
} node_t;

typedef struct {
    node_t* top;
    node_t* bot;
    int size;

} stackJobs_t;
```

Listing 7: Extraído del archivo `stack.h`

Además de las funciones básicas de una pila como pueden ser `pop`, `push`, `peek`, etc, hemos implementado otras funciones que ayudan a realizar la gestión de los procesos en la pila.

Por un lado, la función `pop_pid` borra un nodo si el PID pasado como argumento y el último PID de la lista de PIDs del job son iguales:

```
void pop_pid(stackJobs_t* s, pid_t pid) {
    node_t *node, *tmp;
    node = s->top;
    while (node != NULL && !equal_pid(&(node->job), pid)) {
        node = node->next;
    }

    if (node != NULL) {
        tmp = node;
        if (node->prev != NULL) {
            node->prev->next = tmp->next;
        }
        if (node->next != NULL) {
            node->next->prev = tmp->prev;
        }
        if (s->top == node) {
            s->top = tmp->next;
        }
        if (s->bot == node) {
            s->bot = tmp->prev;
        }
        free(node);
        s->size--;
    }

}
```

Listing 8: Extraído del archivo `stack.c`

Otra función destinada a la gestión de procesos es `check_jobs_stack`. Su objetivo es comprobar el estado de los procesos en background. En primer lugar comprueba si todos los procesos de un nodo han terminado mediante un `wait` no bloqueante (este punto nos dio problemas al implementarlo y será más detallado en el apartado 2.1.2). Seguidamente, se comprueba la posición que ocupa en la pila y si el proceso ha terminado se muestra por pantalla el comando con un mensaje de hecho. Además, le añadimos a esta función la capacidad de mostrar por pantalla los procesos según los va iterando (útil para el comando `jobs`).

```
1  void check_jobs_stack(stackJobs_t* s, int output) {
2      int i, j, error, finished, deleted;
3      pid_t *pids, pid;
4      node_t *node, *tmp;
5      char c, command[1024];
6      i = 1;
7      tmp = NULL;
8      node = s->bot;
9
10     while (node != NULL) {
11         finished = 0;
12         error = 0;
13         pids = get_pids(&(node->job));
14         for (j = 0; j < node->job.index; j++) {
15             pid = waitpid(pids[j], NULL, WNOHANG);
16             if (pid < 0) { // Error
17                 error++;
18                 continue;
19             } else if  (pid != 0) {
20                 finished++;
21             }
22         }
23         tmp = node->prev;
24         c = ' ';
25         if (node == s->top->next) {
26             c = '-';
27         } else if (node == s->top) {
28             c = '+';
29         }
30         if (finished == node->job.index || error == node->job.index) { // El
       proceso ha terminado
31             get_command(&(node->job), command);
32             printf("[%d]%c %d  Hecho\t\t\t%s\n", i, c, pids[node->job.index-1],
        command);
33
34             pop_pid(s, get_pids(&(node->job))[node->job.index-1]);
35         } else if (output != 0) {
36             get_command(&(node->job), command);
37             printf("[%d]%c %d  Ejecutando\t\t\t%s\n", i, c, pids[node->job.
       index-1], command);
38         }
39         node = tmp;
```

```
40          i++;
41      }
42  }
```

Listing 9: Extraído del archivo `stack.c`

La última función del stack perteneciente a los procesos es `fg_job_stack`, que se comentará en el apartado correspondiente a los comandos `builtin`, más concretamente, en el comando `fg` (veáse 1.5.3).

1.4.3. Background

Todos los comandos de `job.c` y `stack.c` necesitan de una interfaz que cree una única cola de procesos para el programa y gestione de manera correcta las diversas llamadas a esta, estas funciones están en el fichero `background.c`

Ninguna de las funciones de este archivo son complejas, simplemente manejan el acceso a la pila de procesos. Por un lado en el *header* están las declaración de las diferentes funciones:

```
1  #ifndef BACKGROUND_H
2  #define BACKGROUND_H
3
4  #include "job.h"
5
6  void init_jobs();
7  void save_job(job_t* j);
8  void check_jobs(int output);
9  void fg_job(char * arg);
10 void delete_jobs();
11 #endif
```

Listing 10: Extraído del archivo `background.h`

Por otro lado, en `background.c` se declara la pila de procesos del programa:

```
1  stackJobs_t* jobs_stack;
```

Listing 11: Extraído del archivo `background.c`

Estas funciones son, principalmente, llamadas por `exec_line` para poner los procesos en *background*. Primero se restaura la linea de comandos, seguidamente se muestra por pantalla que esa linea está corriendo en *background*, se guardan todos los pids y el comando en el job y, finalmente, se gaurda el job en la pila de procesos declarada en `background.c`

```
1    if (line->background == 0) {
2      for(i = 0; i < line->ncommands; i++) {
3        waitpid(pids[i], NULL, 0);
```

```
    }
  } else {
    restore_line(line, command);
        printf("[%d]+ Running\t\t\t", pid);
        printf("%s\n", command);
        for(i = 0; i < line->ncommands; i++) {
            set_pid(&job, pids[i]);
        }
    set_command(&job, command);
    save_job(&job);
  }
```

Listing 12: Extraído de la función `exec_line` en el archivo `exec.c`

1.5. Comandos builtin

Los comandos builtin pedidos están definidos en `builtin.c`. Los comandos `cd` y `exit` no dependen de ningún otro archivo, pero `jobs` y `fg` al tratarse de comandos que interaccionan con los procesos en *background* están parcialmente implementados en otros archivos (`stack.c` y `background.c`).

Las llamadas a los comandos `builtin` se realizan en `minishell.c`:

```
        // Comprobar si el comando a ejecutar es jobs
        if (strcmp(line->commands[0].argv[0], "jobs") == 0) {
            jobs();
            prompt();
            continue;
        }

        check_jobs(0);

    // Comprobar si el comando a ejecutar es cd
    if (strcmp(line->commands[0].argv[0], "cd") == 0) {
      cd(line->commands[0].argv[1]);
      prompt();
      continue;
    }

    // Comprobar si el comando a ejecutar es fg
      if (strcmp(line->commands[0].argv[0], "fg") == 0) {
            fg(line->commands[0].argv[1]);
      prompt();
          continue;
      }

        if (strcmp(line->commands[0].argv[0], "exit") == 0) {
            quit();
```

```
26          }
```

Listing 13: Extraído de la función `main` en el archivo `minishell.c`

1.5.1. cd

El comando `cd` se basa principalmente en la llamada al sistema `chdir`. Primero se realiza un control de argumentos:

- Si no se le pasa argumento obtiene la variable de entorno `HOME` con `getenv`.
- Si se le pasa argumento se usa ese para `chdir`.

Posteriormente se llama a `chdir` con el argumento conveniente:

```
1  void cd(char* dir) {
2    // Se comprueba si hay argumento.
3    if (dir == NULL) {
4      // Si no hay argumento se cambia al directorio en la variable de entorno
5      // HOME
6      dir = getenv("HOME");
7      if (dir == NULL) {
8        fprintf(stderr, RED "[!] Error variable HOME not found\n" RESET);
9        exit(EXIT_FAILURE);
10     }
11     if (chdir(dir) == -1) {
12       fprintf(stderr, RED "[!] Error: %s\n" RESET, strerror(errno));
13     }
14     printf("%s\n", dir);
15   }
16   // Se ejecuta con el arguemento pasado
17   if (chdir(dir) == -1) {
18     fprintf(stderr, RED "[!] Error: %s\n" RESET, strerror(errno));
19   }
20 }
```

Listing 14: Extraído del archivo `builtin.c`

1.5.2. jobs

El comando `jobs` hace uso de la función `check_jobs` y `check_jobs_stack` (veáse 1.4.2). Nos dimos cuenta que comprobar si los comandos han acabado y realizar el comando `jobs` tiene la misma implementación, es por ello que le añadimos a la función un parámetro de *output* para indicarle si ha de mostrar los procesos que va recorriendo mientras comprueba si han acabado.

La posición en la pila también es importante, pues se añade + si es la cima de la pila o - si es el nodo siguiente al de la cima.

```
void jobs() {
  check_jobs(1);
}
```

Listing 15: Extraído del archivo `builtin.c`

1.5.3. fg

El comando `fg` depende de las funciones `fg_job` (`background.c`) y `fg_job_stack` (`stack.c`).

```
void fg(char* arg) {
    fg_job(arg);
}
```

Listing 16: Extraído del archivo `builtin.c`

La función `fg_job_stack` primero realiza una comprobación de argumentos:

- Si no se le pasa argumento, escoge el proceso que este en la cima de la pila.
- Si se le pasa argumento, busca la posición indicada en la pila.

Seguidamente, elimina el proceso de la pila de procesos, muestra el proceso que se pasa a *foreground*, se cambia el *handler* de `SIGINT`, se realiza la gestión perteneciente a control de señales y se espera por cada uno de los PIDs.

```
void fg_job_stack(char * arg, stackJobs_t * s) {
    int size, i;
    job_t j;
    pid_t * pids;
    size = size_stack(s);
    if (arg == NULL){
        j = peek(s);
    }else{
        char * endptr;
        long parsed = strtol(arg, &endptr, 10);
        if (*endptr != '\0' || parsed < 1 || parsed > size || errno == ERANGE)
    {
            printf("[!] N mero de trabajo en segundo plano inv lido\n");
            return;
        }
        j = get(s ,((int)parsed) - 1);
    }
    pids = get_pids(&(j));
```

```
18      pop_pid(s, pids[j.index-1]);
19      printf("%s\n", j.command);
20      signal(SIGINT, sig_handler);
21      set_pgid_fg(pids[0]);
22      for (i = 0; i < j.index; i++){
23          waitpid(pids[i], NULL, 0);
24      }
25  }
```

Listing 17: Extraído del archivo `stack.c`

1.5.4. exit

Para el comando `exit`, se elimina la pila de procesos creada anteriormente y se realiza un exit.

```
1  void quit(){
2      delete_jobs();
3      exit(0);
4  }
```

Listing 18: Extraído del archivo `builtin.c`

1.6. Control de señales

En referencia al control de señales, más concretamente en el control de la señal `SIGINT`, se pueden discernir varios escenarios:

- Ignorar la señal en el proceso padre para que este no muera al hacer `CTRL + C`.
- Matar a los hijos que se estén ejecutando en *foreground* cuando se presione `CTRL + C`.
- Ignorar la señal en los procesos hijos cuando el proceso se esté ejecutando en *background*.
- Matar a los hijos que se estén ejecutando en *background*, se pasen a *foreground* con `fg` y se presione `CTRL + C`

Para el caso de ignorar la señal en el proceso padre, se puede establecer `SIG_IGN` como *handler* de `SIGINT`:

```
1      while (fgets(buf, 1024, stdin)) {
2          signal(SIGINT, SIG_IGN);
3          line = tokenize(buf);
```

Listing 19: Extraído de la función `main` en el archivo `minishell.c`

Para el caso de matar a los hijos en *foreground* debe restablecerse el *handler* de la señal `SIGINT` a `SIG_DFL` (*handler* por defecto) en cada hijo (después de realizar el `fork`):

```
        } else if (pid == 0) {
            signal(SIGINT, SIG_DFL);
```

Listing 20: Extraído de la función `exec_line` en el archivo `exec.c`

Sin embargo, si el proceso hijo ha de ejecutarse en *background* se debe ignorar la señal `SIGINT`:

```
            if (line->background != 0) {
                signal(SIGINT, SIG_IGN);
            }
```

Listing 21: Extraído de la función `exec_line` en el archivo `exec.c`

Para contemplar el caso de cuando se ejecute el comando `fg` y se quiera matar a ese proceso, hemos decido hacer un *handler* que mande una señal `SIGKILL` a todos los procesos con un determinado PGID (el PGID de todos los comandos de una linea es cambiado por el PID del primer comando). Para ello en el comando `fg` se cambia el *handler*:

```
    signal(SIGINT, sig_handler);
    set_pgid_fg(pids[0]);
    for (i = 0; i < j.index; i++){
        waitpid(pids[i], NULL, 0);
    }
```

Listing 22: Extraído de la función `fg_job_stack` en el archivo `stack.c`

El *handler* se define como[2]:

```
void sig_handler(int sig){
    if(kill(-pgid_fg, SIGKILL) == -1) {
        fprintf(stderr, "[!] Error killing %d: %s", pgid_fg, strerror(errno));
    }
}
```

Listing 23: Extraído del archivo `exec.c`

[2]La variable `pgid_fg` es usada para determinar que PGID se quiere eliminar y la cambia el comando `fg` mediante la función `set_pgid_fg`

2. Comentarios

2.1. Problemas

A lo largo del desarrollo del proyecto nos hemos encontrado con varios obstáculos y problemas. El los siguientes subapartados, se comentarán aquellos más importantes y de los cuales más hemos aprendido.

2.1.1. Cierre incorrecto de los pipes

Este fue uno de los primeros problemas a los que nos enfrentamos. Lo que nos ocurría era que no se ejecutaban bien las líneas con más de un mandato y esto era debido a que en la función *exec_line* (véase 1.2) se cerraban bien los pipes en el padre pero no así en los hijos.

La solución a este problema, aunque sencilla, no la supimos ver. Había que cerrar los pipes tanto dentro del bucle donde se realiza un fork() por cada comando como fuera de este. De esta manera se cierra la copia de los pipes que los hijos tienen del padre y los pipes que creó el padre antes del bucle. En resumen, basta con añadir un bucle como el de la figura dentro del bucle cuando $pid == 0$ (ejecuta el hijo) y fuera del bucle.

```
1    for (i = 0; i < line->ncommands - 1; i++) {
2        close(pipes[i][0]);
3        close(pipes[i][1]);
4    }
```

Listing 24: Extraído de la función `exec_line` en el archivo `exec.c`

2.1.2. Comprobación del estado de los procesos en background

Cuando emepezamos a desarrollar la ejecución de comandos en background nos encontramos con la incógnita de cómo saber cuándo un proceso en segundo plano había finalizado su ejecución para quitarlo del stack de procesos. De otra forma el proceso nunca saldría del stack y las ejecuciones del mandato interno *jobs* mostrarían un resultado incorrecto.

Esto lo solucionamos con la implementación de la función *check_jobs_stack* ubicada en el *stack.c* (véase 1.4.2). El fragmento que interesa de esta función y que nos ayudó a solventar este fallo es el siguiente:

```
1    while (node != NULL) {
2        finished = 0;
3        error = 0;
```

```
4         pids = get_pids(&(node->job));
5         for (j = 0; j < node->job.index; j++) {
6             pid = waitpid(pids[j], NULL, WNOHANG);
7             if (pid < 0) { // Error
8                 error++;
9                 continue;
10            } else if  (pid != 0) {
11                finished++;
12            }
13        }
```

Listing 25: Extraído de la función `check_jobs_stack` en el archivo `stack.c`

Lo que realiza este fragmento es el recorrido por cada uno de los procesos en segundo plano almacenados en el stack. Luego se hace un recorrido entero del array de pids que contiene cada *job_t* y se hace una espera no bloqueante del pid. Si el resultado devuelto por el *waitpid* es negativo significa que hubo un error y se incrementa el contador de errores y si es distinto de 0 significa que el proceso ya terminó por lo que se incrementa el contador *finished*. Entonces si el contador de errores o el contador *finished* es igual al total de pids de ese job entonces si elimina del stack con un `pop_pid(s, get_pids(&(node->job))[node->job.index-1]);`

2.1.3. Desfase de la impresión del *prompt* con el resultado del comando

Una vez finalizada la implementación de la ejecución de comandos en background nos dimos cuenta de que cuando lanzábamos un (p.e. `sleep 20 &`) y seguíamos ejecutando comandos durante esos 20 segundos todo iba bien. El problema aparecía cuando pasaban esos 20 segundos y seguíamos ejecutando comandos. Lo que ocurría entonces es que se imprimía antes el *prompt* que el output del comando ejecutado. Por ejemplo, si ejecutábamos un `ls` el resultado se veía como `msh > fich1.txt fich2.txt dir1`.

Al principio pensamos que tenía que ver con que el background estaba mal implementado. Sin embargo, no tenía que ver nada con eso. El problema real se encontraba en la función *exec_line* (véase 1.2) que hacía un `wait(NULL);` tantas veces como comandos haya en la línea cuando la ejecución de la línea era en foreground.

Para solucionarlo declaramos una variable llamada *pids*, un array en el bloque del padre dentro del bucle en el que se creaba cada hijo donde se almacenan el pid de cada comando de la línea. De esta forma tras salir del bucle se podía hacer un *waitpid* bloqueante de cada pid como se muestra en la figura.

```
1   if (line->background == 0) {
2     for(i = 0; i < line->ncommands; i++) {
3       waitpid(pids[i], NULL, 0);
```

```
    }
```

Listing 26: Extraído de la función `exec_line` en el archivo `exec.c`

2.1.4. Problemas en la implementación del control de señales

En una primera implementación del control de la señal `SIGINT` creíamos que todo nos iba bien porque todo ocurría como se esperaba, pero nos dimos cuenta de un fallo y era que si un proceso estaba en *background* y se presionaba `CTRL + C`, el proceso hijo que no estaba en *foreground* moría. En este punto, no sabíamos como implementar el control de la señal, más específicamente el *handler* de la señal. Teníamos que llamar al *handler* que estaba en otro fichero desde el proceso padre con todos los PIDs de los hijos.

Para solucionar este caso optamos por asignarle al PGID de todos los procesos hijos el PID del primer comando de la línea. Así teníamos todos los comandos de una línea agrupados por un PGID para poder mandar una señal a todos sin conocer sus PIDs. La implementación y explicación del *handler* puede verse en el apartado 1.6.

2.2. Mejoras

En este apartado mencionaremos la mejora en el *prompt* que hemos añadido y comentaremos otras que podrían ser añadidas para una mayor completitud de nuestra *minishell*.

La mejora que hemos añadido respecto los requisitos de la práctica es un mejor diseño del *prompt* dándole a nuestra *minishell* un mejor aspecto.

```
void prompt() {
  char buf[1024];
  getcwd(buf, 1024);
  printf(BLUE "msh@%s" RESET ":" CYAN "%s" GREEN " > " RESET, getlogin(), buf);
}
```

Listing 27: Extraído de la función `prompt` en el archivo `utils.c`

Como se puede ver en la figura el código es muy sencillo. Se imprime primero el nombre de la shell (msh) seguido del @usuario que se consigue mediante la función *getlogin* y por último el directorio actual o *current work directory* obtenido con ayuda de la función *getcwd*. La definición de los colores se encuentra en el fichero *utils.h*.

Algunas de las mejoras que podríamos añadir a nuestro proyecto de minishell son:

- Añadir la posibilidad de ejecutar varios comandos en una línea con ; y &&

- Incorporar más mandatos internos como *echo*, *type*, *bg*, *pwd*, *kill*, etc.
- Colorear la salida de algunos mandatos con una paleta de colores.

2.3. Evaluación del tiempo dedicado

Ambos pensamos que hemos dedicado bastante tiempo a este proyecto y creemos que el resultado lo refleja. A ambos miembros de este equipo nos gusta realizar un buen trabajo y aprovechar este tipo de prácticas para aprender y profundizar en los conocimientos.

En cuanto a la práctica nos ha parecido que es una buena manera de afianzar lo aprendido en clase referente a la programación a nivel de sistema operativo en C. Enfrentándonos a los problemas con los que nos hemos ido topando hemos entendido mejor lo impartido en clase.