Programación de aplicaciones — Parte II

Sistemas Operativos 2022/2023

Vamos a desarrollar una pequeña shell, similar en funcionalidad, en un nivel muy básico, con BASH. En esta segunda parte vamos a implementar los componentes principales de la shell y a incluir el soporte necesario para ejecutar tanto comandos internos como externos. Para la ejecución de comandos externos, tendremos que aprender a crear subprocesos donde ejecutarlos y esperar a que terminen.

Contenidos

1. Bucle principal

Muchas aplicaciones interactivas, sean aplicaciones gráficas, videojuegos o shells, se desarrollando entorno a un bucle principal que se encarga de leer la entrada del usuario, procesarla y mostrar o actualizar la salida. Por tanto, tiene sentido que comencemos esta parte desarrollando las piezas básicas de dicho bucle principal para nuestra shell.

En las shells, el bucle principal tiene la responsabilidad de leer una línea de la entrada del usuario, procesarla y ejecutar los comandos que se hayan introducido. Por eso, para esta parte de la práctica desarrollaremos las siguientes funciones:

- `print_prompt()` imprime el *prompt* de la shell.
- `read_line()` lee una línea de la entrada estándar.
- `parse_line()` procesa la línea leída y devuelve una lista de comandos.
- `execute_commands()` ejecuta los comandos de la lista.
- `execute_program()` ejecuta otro programa en un proceso hijo.

Algunos comandos internos de la shell, como `cd`, `cp` o `exit`, se implementarán directamente en la shell y se ejecutaran desde `execute_commands()` al ser detectados en la lista de comandos introducidos por el usuario. Para facilitar su implementación, usaremos una función por comando:

- `cd_command()` cambia el directorio de trabajo.
- `echo_command()` muestra un mensaje por la salida estándar.
- `cp_command()` copia un fichero.
- `mv_command()` mueve un fichero.

Otros comandos, como `ls` o `ps` serán externos –implementados en un ejecutable instalado en el sistema–. Estos comandos se ejecutarán en un proceso hijo, a través de la función `execute_program()`.

En la Figura 1 se muestra la relación entre todas estas funciones para conformar el bucle principal de nuestra shell.

2. Gestión de errores.

En la primera práctica, todos los errores se propagaban hasta el punto de entrada de `copyfile`, donde se mostraban al usuario y el programa terminaba. Esto tenía sentido, ya que la aplicación no podía completar su tarea si alguna de las peticiones al sistema no podían completarse y, por tanto, no tenía sentido intentar continuar.

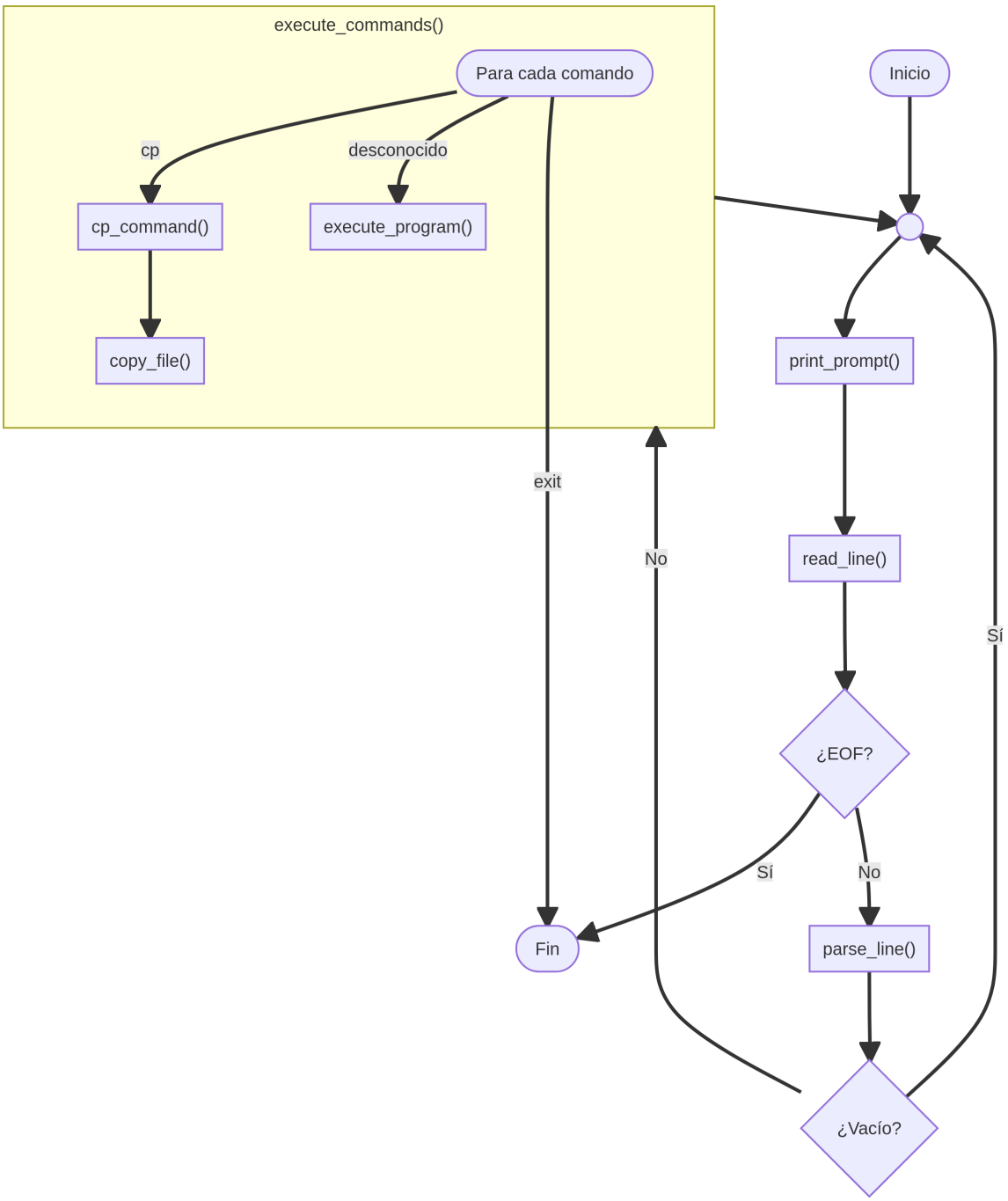

Figura 1: Diagrama de flujo del bucle principal.

Sin embargo, nuestra shell es una aplicación interactiva. Los errores de los comandos deben ser notificados al usuario, pero la shell debe seguir funcionando y permitir al usuario que vuelva a intentarlo con otro comando o con otros argumentos.

Por eso las funciones que implementan comandos internos de la shell –como `cd_command()`, `cp_command()` o `echo_command()`– se harán cargo de mostrar por la salida de error los mensajes de los errores que se produzcan y devolverán a `execute_commands()` un 0, si el comando se ejecutó correctamente, o un valor distinto de 0, si terminó por algún error.

Solo los errores muy graves, que hagan imposible que la shell siga funcionando, deben propagarse hacia `main()` y provocar que la shell termine. Por ejemplo, problemas al reservar memoria o al leer los comandos del usuario desde la entrada estándar.

Un caso que debemos tener en cuenta es que `read_line()` tiene que poder indicarnos cuando hemos alcanzado un *final de archivo* al leer de la entrada estándar. Esto puede ocurrir si el usuario pulsa `CTRL+D`, si se cierra la terminal o si la entrada estándar de la shell ha sido redirigida a aun archivo o a una tubería y ya no hay nada más que leer. Ninguno de estos casos es un error, pero la shell no puede continuar leyendo más comandos desde la entrada estándar, por lo que terminaremos el bucle principal y saldremos con el valor de salida del último comando ejecutado.

3. print_prompt()

La tarea de `print_prompt()` es imprimir el *prompt* de la shell para que el usuario sepa que puede introducir un nuevo comando para su ejecución.

Por ejemplo, podemos mostrar algo como:

```
usuario@hostname:/ruta/actual/de/trabajo $>
```

3.1. Preparación de la cadena de caracteres del prompt

Para preparar un *prompt* como el del ejemplo anterior, necesitamos conocer:

- El nombre del usuario actual, que podemos obtener con la función `getlogin()`.
- El nombre de la máquina, que podemos obtener con la función `gethostname()`.
- La ruta actual de trabajo del proceso de la shell, que podemos obtener con la función `getcwd()`.

Además, nos interesa poder indicar al usuario si el comando que ejecutó anteriormente terminó con éxito o con error. Por eso el prototipo de la función será:

```
void print_prompt(int last_command_status);
```

donde `last_command_status` es el valor de salida del último comando ejecutado, pasado desde el bucle principal al invocar a `print_prompt()`. Si este valor es 0, el *prompt* terminará con `'$> '` para indicar la usuario el éxito del comando anterior. Si el valor es distinto de 0, el *prompt* terminará con `'$< '` para señalar un error en el comando anterior.

> **Nota**
>
> Observa que hemos incluido un espacio al final del *prompt* propuesto, tras los caracteres `'>'` o `'<'`. No se trata de un error. Es así por motivos estéticos, para que el comando que escriba el usuario no quede pegado al texto del *prompt*.

3.2. Detectar si la shell interactiva

El *prompt* es una indicación para el usuario, por lo que no lo mostraremos si la entrada estándar –desde dónde se leerán los comandos– no esta conectada a una terminal. Para saber si la entrada estándar está conectada a una terminal, podemos usar la función `isatty()` con el descriptor de archivo de la entrada estándar. Ese descriptor suele ser 0, pero es preferible usar el valor `STDIN_FILENO` en su lugar.

3.3. Imprimir por la salida estándar

Finalmente, cuando hayamos terminado de construir el `std::string` con el *prompt*, tendremos que imprimirlo por la salida estándar. En lugar de usar las funciones de la librería estándar de C o C++, vamos a usar `write()` de la librería del sistema, que es como lo las primeras lo implementan internamente.

Si tenemos una cadena `str` de tipo `std::string`, podemos usar `write()` así para mostrarla por la salida estándar:

```
std::error_code print(const std::string& str)
{
    int bytes_written = write(STDOUT_FILENO, str.c_str(), str.size())
    if (bytes_written == -1)
    {
        // Manejar error en write()...
    }
}
```

El identificador `STDOUT_FILENO` se define en `<unistd.h>` y se sustituye por el descriptor de fichero de la salida estándar. Por lo general es 1, pero es preferible utilizar `STDOUT_FILENO`.

4. read_line()

La función `read_line()` se llama desde el bucle principal de la shell para leer línea a línea los comandos que el usuario va introduciendo:

```
std::error_code read_line(int fd, std::string& line);
```

La cadena `line` incluye el salto de línea final para diferencia entre cuando el usuario introduce un comando vacío y cuando ya no hay más comandos que leer, por lo que la shell debe terminar:

- Si el usuario introduce un comando vacío, `line` contendrá un solo carácter: `'\n'`.
- Si ya no hay nada más que leer desde la entrada estándar, `line` estará vacía.

La función `read_line()` devuelve 0 en caso de éxito o un valor distinto de 0 en caso de error, siendo ese valor el código de error de `errno`.

La principal dificultad es que usaremos la función `read()` de la librería del sistema para leer de la entrada estándar. Cuando la entrada estándar esta conectada a una terminal, `read()` lee línea a línea porque la terminal no entrega la entrada del usuario, para que sea leída, hasta que el usuario no introduce un salto de línea. Pero si la entrada estándar esta conectada a un fichero o a una tubería, `read()` leerá todo lo que haya en el fichero o en la tubería en ese momento, sin considerar los saltos de línea. Por lo tanto, para asegurarnos de que funciona siempre, tendremos que ir leyendo y buscando los saltos de línea en el buffer leído.

Un forma de hacer podría ser la siguiente:

initialize `pending_input` como un vector vacío de `uint_8`

procedure `read_line(fd, line)`
  loop
    Buscar en `pending_input` el primer salto de línea.
    if encontrado **then**
      Sustituir el contenido de `line` con el contenido de `pending_input`
        desde el principio hasta incluir el salto de línea
      Eliminar del vector `pending_input` el contenido copiado en `line`
      return 0
    end if

    Leer N bytes de `fd` en un buffer de `uint_8`
    if error **then**
      return `errno`
    end if

    if `buffer` vacío **then**
      if `pending_input` *no* vacío **then**

```
                Sustituir el contenido de line con el contenido de pending_input
                Añadir un salto de línea al final de line
                Vaciar pending_input
            end if
            return 0
        else
            Añadir el contenido del buffer al final de pending_input
        end if
    end loop
end procedure
```

Por simplicidad, podemos usar la función `read()` para leer en `std::vector<uint8_t>` que explicamos y usamos en la primera parte de la práctica.

5. parse_line()

La función `parse_line()` se llama desde el bucle principal de la shell. Se encarga de dividir la línea introducida por el usuario en comandos y, para cada comando, separar los argumentos. Por ejemplo, la siguiente línea de entrada:

```
$ cd /etc; ls | wc -l
```

tiene 3 comandos: `cd`, `ls` y `wc`. El primer comando tiene 2 palabras: `cd` y `/etc`. El segundo comando tiene una palabra: `ls`. Y el tercer comando tiene 2 palabras: `wc` y `-l`. La primera palabra de cada comando es el nombre del comando y el resto son sus argumentos.

Por tanto, la función `parse_line()` debe devolver un vector de comandos:

```
std::vector<shell::command> parse_line(const std::string& line);
```

cada uno de los cuales es un vector de palabras `std::vector<std::string>`. Pero usamos `shell::command` como un alias de `std::vector<std::string>` para mejorar la legibilidad del código:

```
namespace shell
{
    using command = std::vector<std::string>;
}
```

Obviamente, no hace falta llamar a `parse_line()` si la línea leída por `read_line()` está vacía.

5.1. Dividir la línea en comandos

Una de la formas más sencillas de dividir una cadena de caracteres en palabras es usar un `std::istringstream`. Un `std::istringstream` es un flujo de entrada que lee de una cadena de caracteres, en lugar de hacerlo desde un archivo o desde la entrada estándar. Como en el caso de `ifstream`, cuando usamos el operador `>>` para leer un `std::string` del flujo de entrada, lo que leemos es una palabra y los espacios entre palabras se pierden:

```
std::istringstream iss(s);
while(! iss.eof()) {
    std::string word;
    iss >> word;

    // Usar o almacenar la palabra word...
}
```

Como se observa en la figura anterior, cuando el flujo de entrada llega al final de la cadena de caracteres `line`, el método `eof()` devuelve `true` y el bucle termina.

Al procesar cada palabra leída debemos considerar una serie de casos especiales antes de incluirla en el vector de argumentos del comando actual:

- Si la palabra es o termina en: `';'`, `'&'` o `'|'`; significa que hemos terminado de leer el comando actual. Por tanto:
    1. Sí se trata del último carácter de la palabra, lo eliminamos de la palabra antes de insertarla en el vector de argumentos del comando actual.
    2. Introducimos el carácter en solitario como último argumento del comando actual. El objetivo es que funciones posteriores sepan con qué carácter terminó el comando, ya que eso puede implicar algún tipo de tratamiento especial.
    3. Iniciamos un nuevo vector dónde comenzar a introducir los argumentos del siguiente comando.
- Si la palabra empieza por `'#'`, significa que el resto de la línea es un comentario. La función `parse_line()` debe ignorar el resto de la línea y salir, retornando el vector de comandos leídos hasta el momento.

6. execute_commands()

La función `execute_commands()` se llama desde el bucle principal de la shell.

```
shell::command_result execute_commands(
    const std::vector<shell::command>& commands);
```

Recibe por medio de `commands` la lista de comandos obtenidos en la llamada a `parse_line()`. Por tanto, no hace falta llamarla si `parse_line()` devuelve un vector vacío.

La tarea de `execute_commands()` es ejecutar los comandos, uno tras otro, devolviendo el resultado del último comando ejecutado al bucle principal.

6.1. Ejecución de comandos

Para cada comando en `commands`, la función comprueba mediante la primera palabra en la lista de palabras del comando –es decir, el nombre del comando– si es un comando interno o no. Si se trata de un comando interno, ejecutará la función `foo_command()` correspondiente. En caso contrario, intentara ejecutará el programa con los argumentos indicados usando la función `execute_program()`.

Tanto las funciones que implementan los comandos internos como `execute_program()` devolverán un valor de retorno: 0 si el comando se ejecutó correctamente o distinto de 0 si se produjo algún error.

El valor del retorno del último comando ejecutado será el valor de retorno de `execute_commands()`, devuelto dentro de un objeto `shell::command_result` al bucle principal.

Una forma posible de definir `shell::command_result` es la siguiente:

```
namespace shell
{
    struct command_result
    {
        int return_value;
        bool is_quit_requested;

        command_result(int return_value, bool request_quit=false)
            : return_value{return_value},
              is_quit_requested{request_quit}
        {}

        static command_result quit(int return_value=0)
        {
            return command_result{return_value, true};
        }
    };
}
```

donde `return_value` es el valor de retorno del último comando ejecutado por `execute_commands()`.

6.2. Terminadores

La función `parse_line()` puede introducir como último argumento de cada comando un carácter terminador: `';'`, `'&'` o `'|'`. Para que las funciones que implementan los comandos internos y `execute_program()` no los consideren como parte de los argumentos del comando a ejecutar, este último argumento debe eliminarse del vector antes de pasarse como parámetro a dichas funciones.

6.3. exit

El comando `exit` es especial ya que tiene por cometido terminar la shell. Si `execute_commands()` encuentra el comando `exit` en su lista, debe retornar inmediatamente con el atributo `request_quit` de `command_result` a `true`.

Observa que construir el objeto `command_result` a retornar con `command_result::quit()` indica mejor las intenciones del código:

```
shell::command_result execute_commands(
    const std::vector<shell::command>& comands)
{
    // ...

    return shell::command_result::quit(return_value);
}
```

que usar directamente el constructor:

```
shell::command_result execute_commands(
    const std::vector<shell::command>& comands)
{
    // ...

    // return shell::command_result{return_value, true};
    return {return_value, true};
}
```

Respecto al valor de `return_value` devuelto en este caso, se usa el valor indicado en el primer argumento del comando `exit` o el valor de retorno del último comando ejecutado por la shell, si el usuario no especificó ningún argumento para `exit`.

> **Nota**
>
> Recuerda que si te hiciera falta, una variable local `static` mantiene su valor entre llamadas a la función, como si fuera una variable global.

En el bucle principal, al obtener el resultado de `execute_commands()`, si `is_quit_requested` es `true`, tendremos que terminar el bucle principal y salir de la shell con el `return_value` como valor de salir del proceso:

```
auto [return_value, is_quit_requested] = execute_commands(commands);
if (is_quit_requested)
{
    // Terminar el bucle principal y salir de la shell con `return_value`.
}
```

7. Comandos internos

Todas funciones que implementan comandos internos deben tener el mismo prototipo, para que sean fácilmente intercambiables en `execute_commands()`:

```
int foo_command(const std::vector<std::string>& args);
```

donde es `args` es la lista de argumentos que se le pasan al comando, incluido el nombre del comando como primer argumento –como en `argv` en `main()`–. El valor de retorno debe ser 0, si el comando se ejecutó correctamente, o un valor distinto de 0, si se produjo algún error.

7.1. echo_command()

El comando `echo`:

1. Concatena todas las cadenas en `args`, intercalando un carácter de espacio entre cada una.
2. Añade un salto de línea `\n'` al final de la cadena.
3. Imprime el resultado en la salida estándar.

7.2. cd_command()

El comando `cd` debe cambiar el **directorio actual de trabajo** de la shell al indicado en `args[1]`. Indicar más argumentos es un error, lo que se puede notificar al usuario mediante el mensaje `demasiados argumentos`.

El **directorio actual de trabajo** es una propiedad de los procesos que indica el directorio en el que se resuelven las rutas, cuando usamos rutas relativas en las funciones de la librería del sistema. Por ejemplo, si el **directorio actual de trabajo** es `/home/usuario`, entonces `open("fichero.txt", O_RDONLY)` abrirá el archivo `/home/usuario/fichero.txt`.

El **directorio actual de trabajo** se hereda de proceso padre a proceso hijo, por lo que cuando cambiamos el **directorio actual de trabajo** en la shell, sabemos que este cambio

no solo afectará a las funciones que esta use internamente, sino también a los programas externos que se ejecuten desde la shell.

Nuestra shell, al igual que cualquier otro proceso, pueden cambiar el **directorio actual de trabajo** mediante la función `chdir()`.

7.3. cp_command() y mv_command()

El comando `cp` debe copiar el archivo indicado en `args[1]` en el archivo indicado en `args[2]` y aceptar la opción `-a` para copiar también los atributos del archivo original. Mientras que el comando `mv` debe mover el archivo indicado en `args[1]` al archivo indicado en `args[2]`.

Ambos comandos deben ser implementados usando las funciones `copy_file()` y `move_file()` desarrolladas en la práctica anterior.

8. Comandos Externos

La función `execute_program()` tiene la responsabilidad de ejecutar los comandos externos usando las funciones de la librería del sistema `fork()` y `exec()`.

Por simplicidad, su prototipo es similar al de las funciones que implementan comandos internos:

```
int execute_program(const std::vector<std::string>& args,
                    bool has_wait=true);
```

donde `args` es la lista de argumentos que se le pasan al comando, incluido el nombre del comando como primer argumento. Por tanto, el programa a ejecutar viene indicado en `args[0]`.

En el tema 9 de los apuntes de la asignatura se explica en detalle cómo ejecutar un programa externo usando las funciones de la librería del sistema `fork()` y `exec()` y se enlazan algunos ejemplos de código.

8.1. Elección de la versión de exec()

Como comentamos en los apuntes, la `exec()` es una familiar de funciones, con varias versiones que se diferencian en el tipo de argumentos que reciben.

Como los usuarios esperan poder ejecutar comandos así:

```
$ firefox
```

sín especificar la ruta para encontrar el ejecutable de `firefox` –esperando que la shell lo localice y lo ejecute– necesitamos que la función `execute_program()` use la versión de

`exec()` que sabe buscar una ejecutable en los directorios de la variable de entorno `PATH`, cuando no se le indica la ruta del programa.

Además, los usuarios pueden ejecutar cualquier programa con cualquier número de argumentos, por lo que no sirven las versiones de `exec()` donde el número de argumentos queda preestablecido en el código, sino aquellas que reciben un array de argumentos, similar a lo que recibe `main()`.

8.2. Ejecución de programas

Una ver elegida la función `exec()`adecuada, el reto es convertir `args` –nuestra lista de argumentos `std::vector<std::string>`– en los argumentos que espera `exec()` –un array de cadenas de caracteres `char*`–.

Como comentamos en la primer parte, es conveniente envolver las llamadas a las funciones de la librería del sistema –como `exec()`– en una función auxiliar en C++, que se encargue de convertir los argumentos con tipos de C++ a los tipos de C que espera cada función.

```
int execute(const std::vector<std::string>& args);
```

Esta función podría hacer lo siguiente:

> Crear `argv` como `std::vector` de `const char*`
> **for** `arg` **in*** `args`do**
> Añadir `arg.c_str()` a `argv`
> **end for**
>
> Añadir `nullptr` a `argv`
> Ejecutar el programa con `exec(argv[0], argv.data())`

Es muy importante el paso de añadir `nullptr` al final de `argv`, ya que `exec()` espera que el array de argumentos ques se le pase termine con un puntero nulo. De esta forma `exec()` sabe, al recorrer el array, que ha llegado al final y cuántos argumentos contiene.

Por otro lado, es posible que el tipo devuelto por `argv.data()` no coincida por con el esperado por `exec()`. El tipo esperado por `exec()` es `char* const[]` –o el equivalente `char* const*`– mientras que el tipo devuelto por `argv.data()` es `const char**`. Esto se puede resolver haciendo un typecast explícito: `const_cast<char* const*>(argv.data())`.

8.3. Ejecución de programas en segundo plano

En los ejemplos de los apuntes el procesos padre siempre espera a que el proceso hijo termine, usando funciones como `wait()` o `waitpid()`, antes de continuar su ejecución.

Ese es el comportamiento esperable para `execute_programa()` cuando `has_wait` es `true`. Además, al terminar el proceso hijo, debe retornar el valor de salida del programa ejecutado.

Pero cuando `has_wait` es `false`, el comando externo debe ejecutarse en segundo plano, sin que `execute_programa()` espere a que termine su ejecución. En ese caso, `execute_programa()` debe retornar el PID del proceso hijo para que `execute_commands()` pueda monitorizarlo periódicamente.

> **Nota**
>
> Esto funciona porque `pid_t` es un `int` en Linux y otros sistemas POSIX. En sistemas donde el identificador de procesos utiliza un tipo completamente diferente –como Windows, donde se usa `HANDLE`– lo correcto sería retornar un `std::variant` con el `int` del valor de retorno de `exec()` y el tipo de los PID.

La función `execute_commands()` debe llamar a `execute_program()` con `has_wait` a `false` cuando el último argumento del comando sea `'&'` y debe guardar el PID retornado y el nombre del comando en un vector.

Tras la ejecución de cada comando, `execute_commands()` debe llamar a `waitpid()` para cada uno de los PID en el vector para comprobar si alguno de los procesos hijos ha terminado. Para cada uno de los que haya terminado, debe mostrar un mensaje indicado qué comando ha terminado, cuál era su PID y con qué valor terminó. Por ejemplo:

```
$ firefox &
$ cp /etc/passwd /tmp
[83] firefox terminado (0)
```

La función de la librería del sistema `waitpid()` bloquea al proceso hasta que el proceso hijo con el PID indicado termine. Como lo que interesante es saber si el proceso ha terminado, no esperar a que termine, se debe llamar a `waitpid()` con el parámetro `WNOHANG` para que no bloquee al proceso padre. Por el valor de retorno de la función sabremos si el proceso consultado ya ha terminado o no.

9. Comandos internos en segundo plano

Actualmente, los comandos internos se ejecutan cuando se les invoca y no se puede introducir otro comando mientras se ejecutan. Para poder ejecutan comandos internos en segundo plano, necesitaríamos utilizar alguna técnica que permita ejecutar las funciones que implementan los comandos internos, al mismo tiempo que el bucle principal continua su ejecución. Esto lo podríamos hacer tanto con hilos como utilizando subprocesos.

Las shell suelen optar por los subprocesos, ya que así es muy sencillo implementar la conexión de comandos mediante tuberías y las redirecciones de la entrada salida. Además, al implementar `execute_program()` ya hemos dejado preparada buena parte del trabajo.

Si has implementado `execute_program()` como hemos indicado, la función crea un proceso hijo y dentro de invoca a `execute()`, que recibe los argumentos como `std::vector<std::string>` Observa que la función `execute()` tiene exactamente el

mismo prototipo que las funciones `foo_command()` que implementan los comandos internos. Por tanto, podríamos sustituir en `execute_program()` la llamada a `execute()` por una llamada a cualquiera de los comandos internos y eso nos permitiría ejecutarlo en segundo plano, si llamamos a `execute_program()` con `has_wait` a `false`.

Para no tener una versión de `execute_program()` para cada comando interno, vamos a crear una nueva versión donde la función invocada en el proceso hijo creado es configurable. La llamaremos `spawn_proccess()` y tendrá el siguiente prototipo:

```
int spawn_proccess(
    std::function<int (const std::vector<std::string>&)>& command,
    const std::vector<std::string>& args,
    bool has_wait=true
);
```

La función `spawn_proccess()` es idéntica a `execute_program()`:

- Crea un proceso hijo usando `fork()`.
- Si `has_wait` es `true`, el proceso padre espera a que el proceso hijo termine y devuelve el valor de salida del hijo.
- Si `has_wait` es `false`, la función retorna inmediatamente con el PID del proceso hijo.

pero en lugar de invocar a `execute()` en el proceso hijo, invoca a la función `command` pasándole los argumentos `args`>

```
if (/* Proceso hijo */) {
    command(args);
}
```

Como en el caso de `execute_program()`, la función `execute_commands()` debe examinar el último argumento de cada comando para ver si es `'&'`. En ese caso, debe llamar a `spawn_proccess()` con la función del comando interno y `has_wait` a `false`. Además, debe guardar el PID retornado y el nombre del comando en un vector para poder monitorizar cuándo termina el subprocesos y mostrar el mensaje correspondiente.

Si el comando no termina en `'&'`, `execute_commands()` debe llamar directamente a la función del comando interno, pasándole los argumentos del comando. En este caso, usar `spawn_proccess()` con `has_wait` a `true` no aportaría nada y es menos eficiente que invocar directamente la función.