Área de Arquitectura y Tecnología de Computadores

Universidad Carlos III de Madrid

SISTEMAS OPERATIVOS

Práctica 2. Programación de un intérprete de mandatos (Minishell)

Grado de Ingeniería en Informática
Grado en Matemática Aplicada y Computación
Doble Grado en Ingeniería Informática y Administración de Empresas

Curso 2023/2024

Índice

1 Enunciado de la Práctica

Esta práctica permite al alumno familiarizarse con los servicios para la gestión de procesos que proporciona POSIX. Asimismo, se pretende que conozca cómo es el funcionamiento interno de un intérprete de mandatos (*shell*) en UNIX/Linux. En resumen, una shell permite al usuario comunicarse con el kernel del sistema operativo mediante la ejecución de comandos o mandatos, ya sean simples o encadenados.

Para la gestión de procesos pesados, se utilizarán las llamadas al sistema de POSIX relacionadas como `fork, wait, exit`. Para la comunicación entre procesos, igualmente se utilizarán las llamadas al sistema `pipe, dup, close, signal`.

El alumno debe diseñar y codificar, en lenguaje C y sobre el sistema operativo UNIX/Linux, un programa que actúe como intérprete de mandatos o *shell*. El programa debe seguir estrictamente las especificaciones y requisitos contenidos en este documento.

1.1 Descripción de la Práctica

El intérprete de mandatos a desarrollar o *minishell* utiliza la entrada estándar (*descriptor de fichero = 0*), para leer las líneas de mandatos que interpreta y ejecuta. Utiliza la salida estándar (*descriptor de fichero = 1*) para presentar el resultado de los comandos por pantalla. Y utiliza la salida estándar de error (*descriptor de fichero = 2*) para notificar los errores que se puedan dar. **Si ocurre un error en alguna llamada al sistema, se utiliza la función de biblioteca `perror` para notificarlo.**

1.1.1 Parser proporcionado

Para el desarrollo de esta práctica se proporciona al alumno un **parser** que permite leer los mandatos introducidos por el usuario. El alumno **solo** deberá preocuparse de implementar el intérprete de mandatos. La sintaxis que utiliza este 'parser' es la siguiente:

Blanco Es un carácter tabulador o espacio.

Separador Es un carácter con significado especial (—, <, >, &), el fin de línea o el fin de fichero (por teclado CTRL-C).

Texto Es cualquier secuencia de caracteres delimitada por blanco o separador.

Mandato Es una secuencia de textos separados por blancos.

- El primer texto especifica el nombre del mandato a ejecutar. Las restantes son los argumentos del mandato invocado. Por ejemplo, en el mandato `ls -l`, **ls** es el mandato y **-l** es el argumento.
- El nombre del mandato se pasa como argumento 0 a la llamada **execvp** (`man execvp`).

- **Cada mandato se ejecuta como un proceso hijo directo del *minishell* usando la llamada fork** (`man 2 fork`).
- El valor de un mandato es su estado de terminación (`man 2 wait`), devuelto por la función exit del hijo y recibido por la función wait en el padre.
- Si la ejecución falla se notifica el error por la salida estándar de error.

Secuencia Es una secuencia de dos o más mandatos separados por |.

- La salida estándar de cada mandato se conecta por una tubería (`man 2 pipe`) a la entrada estándar del siguiente.
- La *minishell* normalmente espera la terminación del último mandato de la secuencia antes de solicitar la siguiente línea de entrada.
- El valor de una secuencia es el valor del último mandato de la misma.

Redirección La entrada o la salida de un mandato o secuencia puede ser redirigida añadiendo tras él la siguiente notación:

- $<$ *fichero* $\rightarrow$ Usa *fichero* como entrada estándar abriéndolo para lectura (`man 2 open`).
- $>$ *fichero* $\rightarrow$ Usa *fichero* como salida estándar. Si el fichero no existe se crea, si existe se trunca (`man 2 open / man creat`).
- $!>$ *fichero* $\rightarrow$ Usa *fichero* como salida estándar de error. Si el fichero no existe se crea, si existe se trunca (`man 2 open / man creat`).

En caso de cualquier error durante las redirecciones, se notifica por la salida estándar de error y se suspende la ejecución de la línea.

Background (&) Un mandato o secuencia terminado en "&" supone la ejecución asíncrona del mismo, es decir, el *minishell* **no queda bloqueado esperando su terminación**. Ejecuta el mandato sin esperar por él, **imprimiendo por pantalla el identificador del proceso hijo (*pid*)**. Para hacer esta impresión se debe seguir el siguiente formato, donde `%d` indica el *pid* del proceso por el que no se espera **(Ver Anexos)**:

`[%d]\n`

Prompt Mensaje de espera por el usuario, antes de leer cada línea. Por defecto será:

`MSH>>`

1.1.2 Obtención de la línea de mandatos

Para obtener la línea de mandatos introducida por el usuario **debe utilizarse la función** `read_command` cuyo prototipo es el siguiente:

```
int read_command(char ***argvv, char **filev, int *bg);
```

La llamada devuelve 0 en caso de teclear CTRL-C (EOF) y -1 en caso de error. **Si se ejecuta con éxito la llamada, devuelve el número de mandatos(*)**. Por ejemplo:

- Para `ls -l` → devuelve 1.
- Para `ls -l | sort` → devuelve 2.

(*) Para un correcto funcionamiento **se debe respetar el formato de entrada** siguiente, separando cada término por un espacio:

```
[<command><args>][|<command><args>]* [<input_file] [>
output_file][!>outer_file][&]
```

El argumento ***argvv*** permite tener acceso a todos los mandatos introducidos por el usuario. Con el argumento ***filev*** se pueden obtener los ficheros utilizados en la redirección:

- **filev[0]** apuntará al nombre del fichero a utilizar en la redirección de entrada en caso de que exista. Si no existe contiene una cadena con un cero ("0").
- **filev[1]** apunta al nombre del fichero a utilizar en la redirección de salida en caso de que exista. Si no existe contiene una cadena con un cero ("0").
- **filev[2]** apunta al nombre del fichero a utilizar en la redirección de la salida de error en caso de que exista. Si no existe contiene una cadena con un cero ("0").

El argumento **in_background** es 1 si el mandato o secuencia de mandatos debe ejecutarse en background y 0 en caso contrario.

EJEMPLO: Si el usuario teclea `ls -l | sort < fichero` los argumentos anteriores tendrán la disposición que se muestra en la siguiente figura:

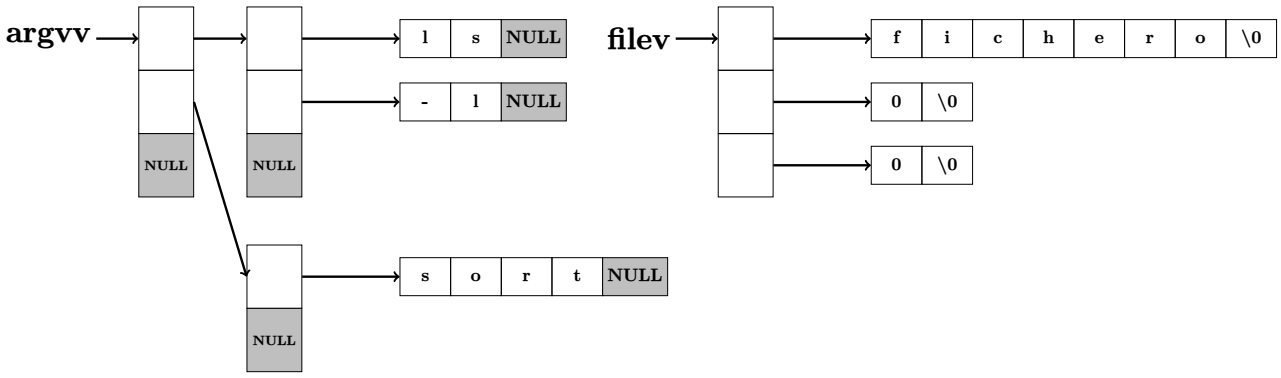

Fig. 1: Estructura de datos generada por el parser.

En el fichero msh.c (**fichero que debe modificar el alumno con el código del *minishell***) se invoca a la función `read_command` y se ejecuta el siguiente código:

```
if (command_counter > 0) {
    if (command_counter > MAX_COMMANDS){
        printf("Error: Numero maximo de comandos es %d\n'', MAX_COMMANDS);
    }
    else {
        // Print command
        print_command(argvv, filev, in_background);
    }
}
```

En dicho código aparece la función **`print_command()`**, encargada de mostrar la información almacenada a partir de la línea de comandos. Dicha función ejecuta el siguiente código:

```
for (int i = 0; i < num_commands; i++){
    for (int j = 0; argvv[i][j] != NULL; j++){
        printf("%s\n'', argvv[i][j]);
    }
}

printf("Redir IN:  %s\n'', filev[0]);
printf("Redir OUT: %s\n'', filev[1]);
printf("Redir ERR: %s\n'', filev[2]);

if (in_background == 0)
    printf("No Bg\n'');
else
    printf("Bg\n'');
```

Se recomienda al alumno que antes de comenzar a implementar la práctica, compile y ejecute el *minishell*, introduciendo diferentes mandatos y secuencias de mandatos para comprender claramente cómo acceder a cada uno de los mandatos de una secuencia.

1.1.3 Desarrollo de la *minishell*

Para desarrollar el *minishell* se recomienda al alumno seguir una serie de pasos, de tal forma que se construya el *minishell* de forma incremental. En cada paso se añadirá nueva funcionalidad sobre el anterior.

1. Ejecución de mandatos simples del tipo `ls -l`, `who`, etc.
2. Ejecución de mandatos simples en background (véase el Anexo para conocer los comandos en *foreground* y en *background*).
3. Ejecución de secuencias de mandatos conectados por pipes. El número de mandatos, en una secuencia, se limitará a 3 (*), es decir: `ls -l | sort | wc`.

   (*) Si los alumnos deciden implementar una secuencia indeterminada de pipes (no limitada a 3, ni a ninguna cantidad), se considerará para nota complementaria.

4. Ejecución de mandatos simples y secuencias de mandatos con redirecciones (entrada, salida y de error) y en background.

5. Ejecución de mandatos internos.

   Un mandato interno es aquel que bien se corresponde directamente con una llamada al sistema o bien es un complemento que ofrece el propio *minishell*. Para que su efecto sea permanente, **ha de ser implementado y ejecutado dentro del propio *minishell* (en el proceso padre) sin redirecciones y sin *background***. Todo mandato interno comprueba el número de argumentos con que se le invoca y si encuentra este o cualquier otro error, lo notifica y termina con valor distinto de cero. Los mandatos internos que se piden en el *minishell* son:

Mandato interno: mycalc

Funciona como una calculadora muy sencilla en el terminal. Toma una ecuación simple, con la forma **operando_1 operador operando_2**, donde *operando* es un número entero y *operador* puede ser la suma (add), la multiplicación (mul) o la división (div), en la que se deberá calcular el cociente entero y el resto de la división.

Para la suma, se almacenarán los valores en una **variable de entorno** llamada "Acc". Dicha variable comienza valiendo 0, y posteriormente se van sumando los resultados de las sumas, pero **no de las multiplicaciones y divisiones**.

Si la operación tiene éxito, se muestra **por la salida estándar de error** el resultado de resolver el cálculo precedido de la etiqueta [OK]. En el caso de la suma, además se deberá mostrar el valor acumulado, y en el caso de la división el resto.

Para la suma, se mostrará el resultado con el siguiente mensaje:

```
[OK] <Operando_1> + <Operando_2> = <Resultado>; Acc <Valor_Acc>
```

Para la multiplicación, se mostrará el resultado con el siguiente mensaje:

```
[OK] <Operando_1> * <Operando_2> = <Resultado>
```

Para la división, se mostrará el resultado con el siguiente mensaje:

```
[OK] <Operando_1> / <Operando_2> = <Cociente>; Resto <Resto>
```

Si el operador no se correspondiese con los dos establecidos, o no se introdujesen todos los términos de la ecuación, se mostrará por la **salida estándar** el mensaje:

```
[ERROR] La estructura del comando es mycalc
<operando_1><add/mul/div><operando_2>
```

A continuación, se muestran algunos ejemplos de uso:

```
msh>> mycalc 3 add −8
[OK] 3 + −8 = −5; Acc −5
msh>> mycalc 5 add 13
[OK] 5 + 13 = 18; Acc 13
msh>> mycalc 4 mul 8
[OK] 4 * 8 = 32
msh>> mycalc 10 div 7
[OK] 10 / 7 = 1; Resto 3
msh>> mycalc 10 / 7
[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>
msh>> mycalc 8 mas
[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>
```

Mandato interno: myhistory

Utilidad para recuperar los 20 últimos comandos o secuencias introducidas en la sesión actual de ejecución del *minishell*. Este mandato interno realizará las siguientes acciones:

- Si se ejecuta sin argumentos, muestra una lista con los 20 últimos comandos introducidos por el usuario en la sesión de *minishell* actual por la **salida estándar de error** usando el formato:

  `<N> <comando>`

- Si se pasa como argumento un número entre 0 y 19 (inclusive) se ejecutará el comando de la lista del histórico correspondiente en lugar de pedir un nuevo comando al parser y mostrará por la **salida estándar de error** el mensaje:

  `Ejecutando el comando <N>`

  Si el número de comando no existe o está fuera de rango, se mostrará el siguiente mensaje por la **salida estandar**:

  `ERROR: Comando no encontrado`

Para la implementación de este mandato interno deberán almacenarse los **20 últimos comandos** introducidos durante la ejecución actual de la *minishell*. Para facilitar la implementación de este mandato interno se proporciona la variable `run_history`, la estructura `command` y dos funciones auxiliares:

- `run_history` indica si el siguiente mandato a ejecutar proviene del mandato interno `myhistory`.
- `store_command` realiza la copia de un comando en el formato usado por el parser a la estructura proporcionada.
- `free_command` libera los recursos utilizados por dicha estructura.

A continuación se muestra un ejemplo de uso para almacenar un comando y liberar la memoria posteriormente.

```
// Declaración de una estructura comando
struct  command cmd;

// Almacenar el comando recibido por el parser en argvv, filev y bg en la estructura
store_command(argvv, filev, bg, &cmd);

// Liberar los recursos usados por la estructura comando
free_command(&cmd);
```

A continuación, se muestran un ejemplo de uso:

```
msh>> myhistory
0 ls
1 ls | grep a
2 ls | grep b &
3 ls | grep c > out.txt &
msh>> myhistory 0
Ejecutando el comando 0
file.txt file2.txt
msh>> myhistory 27
ERROR: Comando no encontrado
```

Importante

Los mandatos internos (**mycalc** y **myhistory**) se ejecutan en el proceso *minishell*, y por tanto:

- **No forman parte de secuencias de mandatos**
- **No tienen redirecciones de ficheros**
- **No se ejecutan en *background***

1.2 Código Fuente de Apoyo

Para facilitar la realización de esta práctica se dispone del fichero `p2_minishell_2024.zip` que contiene código fuente de apoyo. Para extraer su contenido ejecutar lo siguiente:

```
unzip p2_minishell_2024.zip
```

Al extraer su contenido, se crea el directorio p2_minishell_2024/, donde se debe desarrollar la práctica. Dentro de este directorio se habrán incluido los siguientes ficheros:

- **Makefile**
  NO debe ser modificado. Fichero fuente para la herramienta `make`. Con él se consigue la recompilación automática sólo de los ficheros fuente que se modifiquen.
- **libparser.so**
  NO debe ser modificado. Biblioteca dinámica con las funciones del parser de entrada. Permite reconocer sentencias correctas de la gramática de entrada del *minishell*.

- **probador_ssoo_p2.sh**
  NO debe ser modificado. Shell-script que realiza una auto-corrección guiada de la práctica. En ella se muestra, una por una, las instrucciones de prueba que se quieren realizar, así como el resultado esperado y el resultado obtenido por el programa del alumno. Al final se da una nota tentativa del código de la práctica (sin contar revisión manual y la memoria). Para ejecutarlo se debe dar permisos de ejecución al archivo mediante:

  ```
  chmod +x probador_ssoo_p2.sh
  ```

  Y ejecutar con:

  ```
  ./probador_ssoo_p2.sh <fichero_zip_código>
  ```

- **msh.c**
  Este fichero es el que se DEBE MODIFICAR PARA HACER LA PRÁCTICA. Fichero fuente de C que muestra cómo usar el parser. Se recomienda estudiar detalladamente para la correcta comprensión de la práctica, la función `read_command`. La versión actual que se ofrece hace "eco" (*eco = impresión por pantalla*) de las líneas tecleadas que sean sintácticamente correctas. **Esta funcionalidad debe ser eliminada y sustituida por las líneas de código que implementan la práctica.**

- **autores.txt**
  Debe modificarse. Fichero `txt` donde incluir los autores de la práctica.

1.3 Compilación y enlazado de biblioteca compartida

Para poder compilar y ejecutar la práctica es **necesario enlazar la biblioteca libparser.so** que se encarga de aceptar datos de entrada, tanto por teclado como en formato de fichero para la realización de pruebas. Para enlazar y compilar se debe seguir el siguiente proceso:

1. Crear un directorio en la ruta deseada (desde ahora **path**) y descomprimir el código de apoyo.
2. Compilar el código inicial con la biblioteca proporcionada:
   - Ejecutar `make` y evaluar el comportamiento de la compilación.
3. Ejecutar el programa `msh`:
   (a) Si se produce un error porque no se encuentra la biblioteca, se debe indicar al compilador dónde buscar. Por ejemplo:

   ```
   export LD_LIBRARY_PATH=/home/username/path:$LD_LIBRARY_PATH
   ```

   (b) Ejecutar de nuevo.

2 Entrega

2.1 Plazo de Entrega

La fecha límite de entrega de la práctica en AULA GLOBAL será el **12 de abril de 2024 (hasta las 23:55h)**

2.2 Procedimiento de entrega de las prácticas

La entrega de las prácticas ha de realizarse de forma electrónica y por **un único integrante del grupo**. En AULA GLOBAL se habilitarán unos enlaces a través de los cuales se podrá realizar la entrega de las prácticas. En concreto, **se habilitará un entregador para el código de la práctica, y otro de tipo TURNITIN** para la memoria de la práctica.

2.3 Documentación a Entregar

Se debe entregar un archivo comprimido en formato zip con el nombre

`ssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip`

Donde A...A, B...B y C...C son los NIAs de los integrantes del grupo. Los grupos estarán formados por un máximo de tres estudiantes, en caso de realizar la práctica en solitario, el formato será
ssoo_p2_AAAAAAAAA.zip. **El archivo zip se entregará en el entregador correspondiente al código de la práctica**. El archivo debe contener:

- **msh.c**
- **autores.txt: Fichero de texto en formato csv con un autor por línea. El formato es: NIA, Apellidos, Nombre**

NOTA

Para comprimir dichos ficheros y ser procesados de forma correcta por el probador proporcionado, se recomienda utilizar el siguiente comando:

```
zip ssoo_p2_AAA_BBB_CCC.zip msh.c autores.txt
```

La memoria se entregará en formato PDF. Solo se corregirán y calificarán memorias en formato pdf. El fichero debe llamarse:

`ssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCC.pdf`

Tendrá que contener al menos los siguientes apartados:

- **Portada** con los nombres completos de los autores, NIAs y direcciones de correo electrónico.
- **Índice**

- **Descripción del código** detallando las principales funciones implementadas. NO incluir código fuente de la práctica en este apartado. Cualquier código será automáticamente ignorado.
- **Batería de pruebas** utilizadas y resultados obtenidos. Se dará mayor puntuación a pruebas avanzadas, casos extremos, y en general a aquellas pruebas que garanticen el correcto funcionamiento de la práctica en todos los casos. Hay que tener en cuenta:
  1. Que un programa compile correctamente y sin advertencias (**warnings**) no es garantía de que funcione correctamente.
  2. Evite pruebas duplicadas que evalúan los mismos flujos de programa. La puntuación de este apartado no se mide en función del número de pruebas, sino del grado de cobertura de las mismas. Es mejor pocas pruebas que evalúan diferentes casos a muchas pruebas que evalúan siempre el mismo caso.
- **Conclusiones**, problemas encontrados, cómo se han solucionado, y opiniones personales.

Se puntuará también los siguientes aspectos relativos a la **presentación** de la práctica:

- Debe contener portada, con los autores de la práctica y sus NIAs.
- Debe contener índice de contenidos.
- La memoria debe tener números de página en todas las páginas (menos la portada).
- El texto de la memoria debe estar justificado.

El archivo pdf se entregará en el entregador correspondiente a la memoria de la práctica (entregador TURNITIN). La longitud de la memoria no deberá superar las 15 páginas (portada e índice incluidos). Es imprescindible aprobar la memoria para aprobar la práctica, por lo que no debe descuidar la calidad de la misma.

NOTA: Es posible entregar el código de la práctica tantas veces como se quiera dentro del plazo de entrega, siendo la última entrega realizada la versión definitiva. **LA MEMORIA DE LA PRÁCTICA ÚNICAMENTE SE PODRÁ ENTREGAR UNA ÚNICA VEZ A TRAVÉS DE TURNITIN.**

3 Normas

1. Las prácticas que no compilen o que no se ajusten a la funcionalidad y requisitos planteados, obtendrán una calificación de 0.
2. **Las prácticas donde se implementen las redirecciones en el hilo máster recibirán una sanción grave.**
3. **Los programas que ejecuten mandatos o secuencias de mandatos con procesos que no sean hijos de la minishell (jerarquía hijo, nieto, bisnieto) recibirán una calificación baja. Tampoco se permite utilizar sentencias o funciones como `goto`.**
4. Se prestará especial atención a detectar funcionalidades copiadas entre dos prácticas. En caso de encontrar implementaciones comunes en dos prácticas, los alumnos involucrados (copiados y copiadores) perderán las calificaciones obtenidas por evaluación continua.
5. Los programas deben compilar sin **warnings**.
6. Los programas deberán funcionar bajo un sistema Linux, no se permite la realización de la práctica para sistemas Windows. Además, para asegurarse del correcto funcionamiento de la práctica, deberá chequearse su compilación y ejecución en máquina virtual con Ubuntu Linux o en las Aulas Virtuales proporcionadas por el Laboratorio de Informática de la universidad. Si el código presentado no compila o no funciona sobre estas plataformas la implementación no se considerará correcta.
7. Un programa no comentado, obtendrá una calificación **muy baja**.
8. La entrega de la práctica se realizará a través de Aula Global, tal y como se detalla en el apartado Entrega de este documento. No se permite la entrega a través de correo electrónico sin autorización previa.
9. Se debe respetar en todo momento el formato de la entrada y salida que se indica en cada programa a implementar.
10. Se debe realizar un control de errores en cada uno de los programas, más allá de lo solicitado explícitamente en cada apartado.

Los programas entregados que no sigan estas normas no se considerarán aprobados.

4 Anexo

4.1 Manual (comando man)

man es el paginador del manual del sistema, es decir permite buscar información sobre un programa, una utilidad o una función. Véase el siguiente ejemplo:

```
man 2 fork
```

Las páginas usadas como argumentos al ejecutar man suelen ser normalmente nombres de programas, utilidades o funciones. Normalmente, la búsqueda se lleva a cabo en todas las secciones de manual disponibles según un orden predeterminado, y sólo se presenta la primera página encontrada, incluso si esa página se encuentra en varias secciones.

Una página de manual tiene varias partes. Éstas están etiquetadas como NOMBRE, SINOPSIS, DESCRIPCIÓN, OPCIONES, FICHEROS, VÉASE TAMBIÉN, BUGS, y AUTOR. En la etiqueta de SINOPSIS se recogen las librerías (identificadas por la directiva #include) que se deben incluir en el programa en C del usuario para poder hacer uso de las funciones correspondientes. **Para salir de la página mostrada, basta con pulsar la tecla 'q'.**

Las formas más comunes de usar man son las siguientes:

- **man sección elemento**: Presenta la página de elemento disponible en la sección del manual.
- **man -a elemento**: Presenta, secuencialmente, todas las páginas de elemento disponibles en el manual. Entre página y página se puede decidir saltar a la siguiente o salir del paginador completamente.
- **man -k palabra-clave**: Busca la palabra-clave entre las descripciones breves y las páginas de manual y presenta todas las que casen.

4.2 Modo background y foreground

Cuando una secuencia de mandatos se ejecuta en background, **el pid que se imprime es el del proceso que ejecuta el último mandato de la secuencia.**

Cuando un mandato simple se ejecuta en background, **el pid que se imprime es el del proceso que ejecuta ese mandato**.

Con la operación de background, es posible que el proceso minishell muestre el prompt entremezclado con la salida del proceso hijo. **Esto es correcto**.

Después de ejecutar un mandato en foreground, la minishell no puede tener procesos zombies de mandatos anteriores ejecutados en background.

5 Bibliografía

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (`man function`)