



Área de Arquitectura y Tecnología de Computadores
Universidad Carlos III de Madrid

SISTEMAS OPERATIVOS

Práctica 2. 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 2022-2023

- 1 Introducción
- 2 Material
- 3 Descripción de la Práctica
- 4 Mandatos internos
- 5 Probador
- 6 Entrega

- 1** Introducción
- 2 Material
- 3 Descripción de la Práctica
- 4 Mandatos internos
- 5 Probador
- 6 Entrega

Introducción

- Desarrollo de un intérprete de mandatos (minishell) en UNIX/Linux en lenguaje C.
- Debe permitir:

- Ejecución de mandatos simples

```
ls, cp, mv, rm, [...].
```

- Ejecución de secuencias de mandatos

```
ls | wc -l  
ls | sort | wc -l
```

- Ejecución de mandatos simples o secuencias en *background* (&)

```
ls &
```

- Ejecución de mandatos simples o secuencias con redirección de entrada, salida o salida de error:

```
cat | more < fichero  
ls > fichero  
ls | grep mio > mis_entradas  
make install !> salida_error
```

Proceso de desarrollo

1. Soporte para mandatos simples: `ls`, `cp`, `mv`, `[...]`.
2. Soporte para ejecución de mandatos simples en *background* (`&`).
3. Soporte para secuencias de mandatos: `ls | wc -l`, `[...]`.
4. Soporte para secuencias de mandatos en *background* (`&`).
5. Soporte para redirecciones sobre mandatos simples y secuencias de mandatos (`<`, `>`, `!>`).
6. Mandatos internos:
 - `mycalc`
 - `mytime`

- 1 Introducción
- 2 Material**
- 3 Descripción de la Práctica
- 4 Mandatos internos
- 5 Probador
- 6 Entrega

Código Fuente de Apoyo

- Para el desarrollo de la práctica se proporciona código inicial que se puede descargar de Aula Global.
- Los ficheros proporcionados son:

```
1      p2_minishell_21_22/  
2          Makefile  
3          libparser.so  
4          probador_ssoo_p2.sh  
5          msh.c  
6          autores.txt
```

- Para compilar la práctica simplemente ejecutar el comando `make`, y exportar el *path* para la librería dinámica

`export`

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

- El alumno solo debe **modificar el fichero msh.c para incluir la funcionalidad pedida.**

Índice

- 1 Introducción
- 2 Material
- 3 Descripción de la Práctica**
- 4 Mandatos internos
- 5 Probador
- 6 Entrega

Obtención de mandatos

- Para la recuperación de las órdenes se utiliza un analizador sintáctico. Comprueba si la secuencia de órdenes tiene una estructura correcta y permite recuperar el contenido a través de la función:

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

- Devuelve:

- 0 → En caso de EOF (CTRL + C).
- -1 → En caso de error.
- n → Número de mandatos introducidos.

- Ejemplos:

- `ls | sort` → Devuelve 2.
- `ls | sort > fich` → Devuelve 2.
- `ls | sort &` → Devuelve 2.
- `cat < input_file` → Devuelve 1.

Función `read_command`

- La función `read_command` retorna como primer parámetro:

`char *argvv`**

- Es una estructura que contiene los mandatos introducidos por el usuario
- Ejemplo:

- Imprimir el mandato `i`:

`printf("Mandato i: %s \n", argvv[i][0]);`

- Imprimir su primer elemento:

`printf("Arg 1 de i: %s \n", argvv[i][1]);`

Función `read_command`

- La función `read_command` retorna como segundo parámetro:

`char **filev`

- Es una estructura que contiene los nombres de los ficheros usados en las redirecciones o, si no existe, cadena con un cero ("0").
 - **`filev[0]`** Cadena que contiene el nombre del fichero usado para la redirección de entrada (<)
 - **`filev[1]`** Cadena que contiene el nombre del fichero usado para la redirección de salida (>)
 - **`filev[2]`** Cadena que contiene el nombre del fichero usado para la redirección de salida de error (! >)

Función `read_command`

- La función `read_command` retorna como tercer parámetro:

`int *in_background`

- Es una variable que indica si se ejecutan los mandatos en *background*
- Sus valores son:
 - `in_background = 0` → Si no se ejecuta en *background*
 - `in_background = 1` → Si se ejecuta en *background* (&)

Función read_command

■ ls -l | sort < fichero

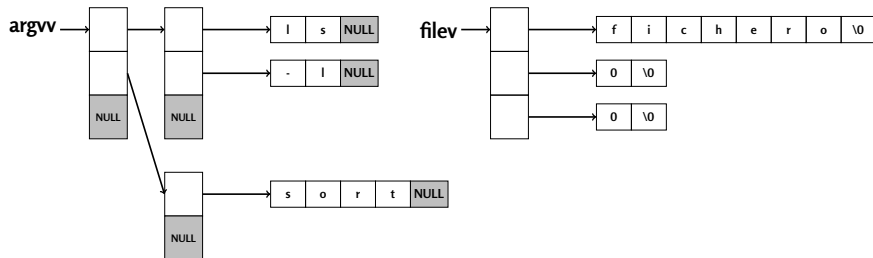


Figura: Estructura de datos generada por el parser.

Descriptores de fichero de un proceso

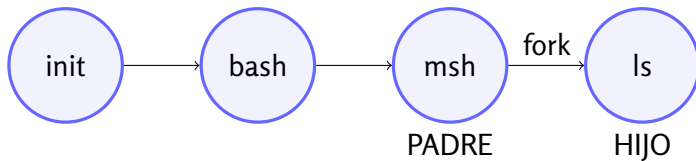
- En UNIX/Linux todo proceso tiene abiertos tres descriptores de fichero por defecto:
 1. Entrada estándar (STDIN_FILENO): Valor = 0
 2. Salida estándar (STDOUT_FILENO): Valor = 1
 3. Salida de error estándar (STDERR_FILENO): Valor = 2
- Los mandatos que se ejecutan en una shell están programados para leer y escribir de la entrada/salida estándar.
- Es posible redireccionar la entrada/salida estándar para leer/escribir de otros ficheros, o para leer/escribir en una tubería.
- Tabla de descriptores de un proceso cuando se crea:

0	STD_IN
1	STD_OUT
2	STD_ERR

Identificadores de procesos

- En el minishell toda la creación de procesos se hace a partir del proceso del propio minishell
- Ejemplo de ejecución de la orden `ls`.

```
1 bash> ./msh      #Ejecucion del minishell
2 msh> ls          #Ejecución de ls dentro del msh
```



- Cada mandato (por ejemplo `ls`) será ejecutado en un proceso hijo del minishell

Creación de procesos con `fork()`

- Permite generar un nuevo proceso o proceso hijo que es una copia exacta del proceso padre `pid_t fork()`, devuelve:
 - **0** si es el hijo
 - **pid** si es el padre
- El proceso hijo hereda:
 - Los valores de manipulación de señales.
 - La clase del proceso.
 - Los segmentos de memoria compartida.
 - La máscara de creación de ficheros, etc.
- El proceso hijo **difiere** en:
 - El hijo tiene un ID de proceso único
 - **Dispone de una copia privada de los descriptores de ficheros abiertos por el padre.**
 - El conjunto de señales pendientes del proceso hijo es vaciado.
 - El hijo no hereda los bloqueos establecidos por el padre.

Ejemplo de creación de procesos con fork()

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main(){
6     int pid, estado;
7     pid = fork();
8     switch(pid){
9         case -1: /* error */
10             perror ("Error en el fork");
11             return -1;
12         case 0: /* hijo */
13             printf("El proceso HIJO se duerme 10 segundos\n");
14             sleep(10);
15             printf("Fin del proceso HIJO\n");
16             break;
17         default: /* padre */
18             if (wait(&estado) == -1) //El padre espera por el hijo
19                 perror ("Error en el wait");
20             printf ("Fin del proceso PADRE\n");
21     }
22     exit(0);
23 }
```

Ejecución de procesos con `execvp()`

- La función `execvp` reemplaza la imagen del proceso que la invoca con una nueva. Esta imagen nueva corresponderá al mandato que desea ejecutar.

```
int execvp (const char *file, char *const argv[]);
```

- Argumentos:
 - `file`: Ruta del fichero que contiene el comando que va a ser ejecutado. Si no hay ruta busca dentro del `PATH`.
 - `argv[]`: Lista de argumentos disponibles para el nuevo programa. **El primer argumento por convenio debe apuntar al nombre del fichero que se va a ejecutar.**
- Retorno:
 - Si la función retorna algo es porque ha ocurrido un error.
 - Devuelve `-1` y el código de error está en la variable global `errno`.
 - La función no devuelve nada si la ejecución ha sido correcta.

Ejemplo de creación de procesos con `execvp()`

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main(){
6     int pid;
7     char *argumentos[3] = {"ls", "-l", "NULL"};
8     pid = fork();
9     switch(pid){
10         case -1: /* error */
11             perror ("Error en el fork");
12             return -1;
13         case 0: /* hijo */
14             execvp(argumentos[0], argumentos);
15             perror ("Error en el exec. Si todo ha ido bien, NO debería ejecutarse");
16             break;
17         default: /* padre */
18             printf ("Soy el proceso PADRE\n");
19     }
20     exit(0);
21 }
```

Finalización y espera de procesos

- La finalización de un proceso puede hacerse con las sentencias:

```
return status;  
void exit(int status);  
void abort(void); //Finalización anormal del proceso.
```

- Los procesos pueden esperar a la finalización de otros proceso.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Normalmente los procesos padres siempre esperan a que finalicen los hijos

```
pid_t wait(int *status);
```

- Si un proceso finaliza y su proceso padre no ha hecho wait esperando por él, pasa a estado ZOMBIE.

```
ps -axf → Permite visualizar los procesos zombie.
```

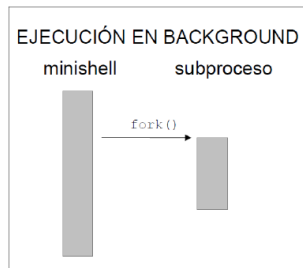
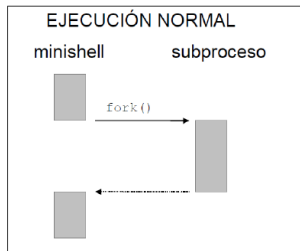
```
kill -9 → Permite matar un proceso.
```

Ejemplo de finalización y espera de procesos

```
1 #include <sys/type.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <sys/wait.h>
5
6 int main(){
7     int pid, status;
8     char *argumentos[3] = {"ls", "-l", "NULL"};
9     pid = fork();
10    switch(pid){
11        case -1: /* error */
12            perror ("Error en el fork");
13            return -1;
14        case 0: /* hijo */
15            execvp(argumentos[0], argumentos);
16            perror ("Error en el exec. Si todo ha ido bien, NO debería ejecutarse");
17            break;
18        default: /* padre */
19            while (wait(&status) != pid);
20            if ( status == 0 ) printf ("Ejecución normal del hijo\n");
21            else printf ("Ejecución anormal del hijo\n");
22    }
23    exit(0);
24 }
```

Ejecución en *background*

1. Un mandato puede ser ejecutado en *background* desde la línea de comandos indicando al final un `&`. Por ejemplo: `sleep 10 &`
2. En este caso el proceso padre no se bloquea esperando la finalización del proceso hijo.
3. La orden `fg <job_id>` permite recuperar un proceso en *background*. Recibe un id de trabajo, no un pid.



Identificadores de procesos

- Un proceso es un *programa en ejecución*
- Todos los procesos tienen un identificador único. Dos primitivas permiten recuperar el identificador de un proceso:

```
pid_t getpid(void);  
pid_t getppid(void);
```

- Un ejemplo:

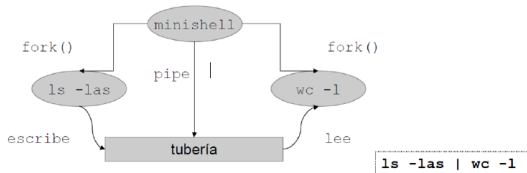
```
1 #include <sys/types.h>  
2 #include <stdio.h>  
3  
4 int main(){  
5     printf("Identificador del proceso: %s\n", getpid());  
6     printf("Identificador del proceso padre: %s\n", getppid());  
7     return 0;  
8 }
```

Secuencias de mandatos con tuberías

- Las secuencias de mandatos se separan por un pipe o tubería |. Por ejemplo:

```
ls -las | wc -l
```

- La salida estándar de cada mandato se conecta a la entrada estándar del siguiente.
- El primer mandato lee de la entrada estándar (teclado) si no existe redirección de entrada.
- El último mandato escribe en la salida estándar (pantalla) si no existe redirección de salida.



Creación de tuberías con pipe()

- Para la creación de tuberías sin nombre se utiliza la primitiva pipe.

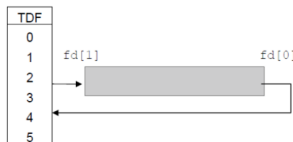
```
#include <unistd.h>
int pipe(int descf[2])
```

- Devuelve:

- -1 → Si error.
- 0 → En cualquier otro caso.

- Recibe un array con los descriptores de fichero para entrada y salida.

- descf[0] → Descriptor de entrada del proceso (read).
- descf[1] → Descriptor de salida del proceso (write).



Primitivas dup y dup2

- Las primitivas `dup` y `dup2` permiten duplicar descriptors de ficheros.

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

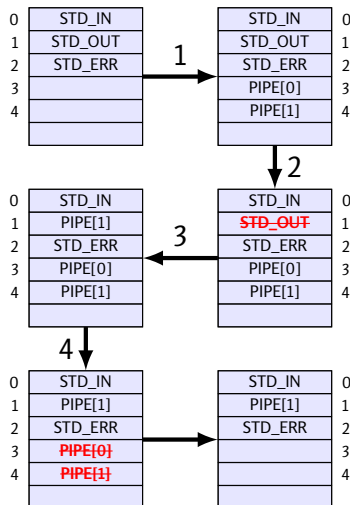
- La primitiva `dup` utiliza el primer descriptor disponible de la tabla un fichero.

Tabla Descriptores	
0	STDIN
1	STDOUT
2	STDERR
3	./file_a
4	./file_b
5	
6	

```
#include <unistd.h>
#include <stdio.h>
int main() {
    int fd1, fd2, fd3;
    fd1 = open("./file_a", O_READ);
    fd2 = open("./file_b", O_READ);
    fd3 = dup(fd2);
}
```

Uso de pipe + dup

1. `pipe` \rightarrow `pipe(pipe)`
2. `close` \rightarrow `close(STDOUT_FILENO)`
3. `dup` \rightarrow `dup(pipe[1])`
4. `close` \rightarrow `close(pipe[0])`
`close(pipe[1])`

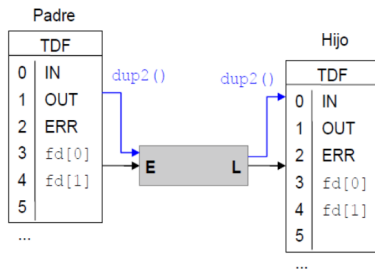


Ejemplo de uso de tuberías

```

1  int main(int argc, char *argv[])
2  {
3      int fd[2];
4      char *args1[2] = {"more", "NULL"};
5      char *args2[3] = {"ls", "NULL"};
6      pipe(fd);
7      if(fork() == 0)
8      {
9          close(STDIN_FILENO);
10         dup(fd[0]);
11         close(fd[1]);
12         execvp(args1[0], args1);
13     } else
14     {
15         close(STDOUT_FILENO);
16         dup(fd[1]);
17         close(fd[0]);
18         execvp(args2[0], args2);
19     }
20     printf("ERROR: %d\n", errno);
21 }

```



Redirecciones de entrada, salida y error

- La primitiva `open` utiliza el primer descriptor disponible de la tabla al abrir un fichero.
- Es posible redireccionar la entrada/salida estándar para escribir/leer de otros ficheros.
- La redirección de entrada (`<`) sólo afecta al primer mandato.
- Ejemplo: Abre un fichero en modo lectura y lo usa como entrada estándar.

```
close (STDIN_FILENO);  
df = open ("./fichero_entrada", O_RDONLY);
```

Redirecciones de entrada, salida y error

- La redirección de salida ($>$) sólo afecta al último mandato.
- Ejemplo: Abre un fichero en modo escritura y lo usa como salida estándar

```
close (STDOUT_FILENO);  
df = open ("./fichero_salida", O_CREAT | O_WRONLY, 0666);
```

- La redirección de salida error ($!>$) afectan a cualquier mandato.
- Ejemplo: Abre un fichero en modo escritura y lo usa como salida de error.

```
close (STDERR_FILENO);  
df = open ("./fichero_error", O_CREAT | O_WRONLY, 0666);
```

Control de errores

- Cuando una llamada al sistema falla devuelve -1. El código de error asociado se encuentra en la variable global `errno`.
- En el fichero `errno.h` se encuentran los posibles valores que puede tomar.
- Para acceder al código de error existen dos posibilidades:
 - Usar `errno` como índice para acceder a la cadena de `sys_errlist[]`.
 - Usar la función de librería `perror()`. Ver `man 3 perror`.
- `perror` imprime el mensaje recibido como parámetro y a continuación el mensaje asociado al código del último error ocurrido durante una llamada al sistema.

```
1  #include <stdio.h>
2  void perror (const char *s);
```

- 1 Introducción
- 2 Material
- 3 Descripción de la Práctica
- 4 Mandatos internos**
- 5 Probador
- 6 Entrega

Mandatos internos

- Un mandato interno es aquel que o bien se corresponde con una llamada al sistema o bien es un complemento que ofrece el propio minishell.
- Su función ha de ser implementada dentro del propio minishell.
- Deberá analizarse la entrada de los mandatos. **El parser no lo hace.**
- Se ejecutarán en el proceso minishell.
- No forman parte de secuencias de mandatos.
- No tienen redirecciones de fichero.
- No se ejecutan en *background*.

mycalc

- El minishell debe proporcionar el comando interno `mycalc` cuya sintaxis es:

`mycalc <operando_1> <add/mul/div> <operando_2>`

- Para ello debe:
 1. Comprobar que la sintaxis es correcta.
 2. Ejecutar la operación elegida: suma (add), multiplicación (mul) o división (div).
 3. Mostrar por pantalla los resultados.
 - Caso correcto por la salida estándar de error.
 - Error por la salida estándar.
- Operación “add” tiene una variable de entorno “Acc” que acumula los resultados de las sumas realizadas, pero no de las multiplicaciones y las divisiones.

mycalc

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

mytime

- El minishell debe proporcionar el comando interno `mytime` cuya sintaxis es:

`mytime`

- Para ello debe:
 1. Comprobar que la sintaxis es correcta.
 2. Obtener el tiempo que lleva ejecutando la minishell milisegundos (almacenado en la variable `mytime`).
 3. Mostrar por la salida estandar de error el tiempo en formato HH:MM:SS.

mytime

```
1 msh>> mytime  
2 00:14:33
```

Índice

- 1 Introducción
- 2 Material
- 3 Descripción de la Práctica
- 4 Mandatos internos
- 5 Probador**
- 6 Entrega

Probador

- Se proporciona a los alumnos el shell-script **probador_ssoo_p2.sh**
- El probador deberá ejecutarse en las máquinas virtuales con Ubuntu Linux o en la Aulas Virtuales del Laboratorio de Informática
- 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>
```

- Ejemplo:

```
$ ./probador_ssoo_p2.sh  
ssoo_p2_100254896_100047014.zip
```

- 1 Introducción
- 2 Material
- 3 Descripción de la Práctica
- 4 Mandatos internos
- 5 Probador
- 6 Entrega**

Plazo de entrega y grupos

- Grupos de 3 personas máximo.
- Entrega de:
 - Código Fuente en un archivo comprimido
 - Memoria de práctica en PDF a través de entregador TURNITIN
 - Solamente podrá entregar un integrante del grupo
- Fecha de entrega:

16 de Abril de 2023 (hasta las 23:55h)



Área de Arquitectura y Tecnología de Computadores
Universidad Carlos III de Madrid

SISTEMAS OPERATIVOS

Práctica 2. 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 2022-2023