

SISTEMAS OPERATIVOS

Práctica 2. Intérprete de mandatos

Presentación

2

- ▣ Obtención de mandatos
- ▣ Ejecución de un mandato simple
- ▣ Ejecución en background
- ▣ Ejecución de secuencias de mandatos
- ▣ Redirección de E/S
- ▣ Mandatos internos

Introducción

3

- 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 &
```

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

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

```
ls > fichero
```

```
cat | more < fichero
```

```
ls | grep mio > mis_entradas
```

```
make install 2> salida_error
```

Proceso de desarrollo

4

- Se recomienda un **desarrollo incremental**.
 1. Soporte para mandatos simples: `ls`, `cp`, `mv`, `[...]`.
 2. Soporte para ejecución de mandatos simples en background (&).
 3. Soporte de secuencias de mandatos.
 4. Soporte para secuencias de mandatos en background (&).
 5. Soporte para redirecciones sobre mandatos simples y secuencias de mandatos.
 6. Mandatos internos
 - `mycalc`
 - `mycp`

Material proporcionado

5

- Para el desarrollo de la práctica se proporcionará código adicional que puede descargar de aula global

- Los ficheros proporcionados son:

`p2_minishell_19-20/`

`Makefile`

`libparser.so`

`msh.c`

- Para compilar la práctica simplemente ejecutar el comando `make`, y exportar el path para la librería dinámica.
- El alumno sólo debe:
 - ▣ Modificar el fichero `msh.c` para incluir la funcionalidad pedida.

Obtención de mandatos

6

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

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

- Devuelve

0 En caso de EOF (CTRL + C)

n Número de mandatos tecleados

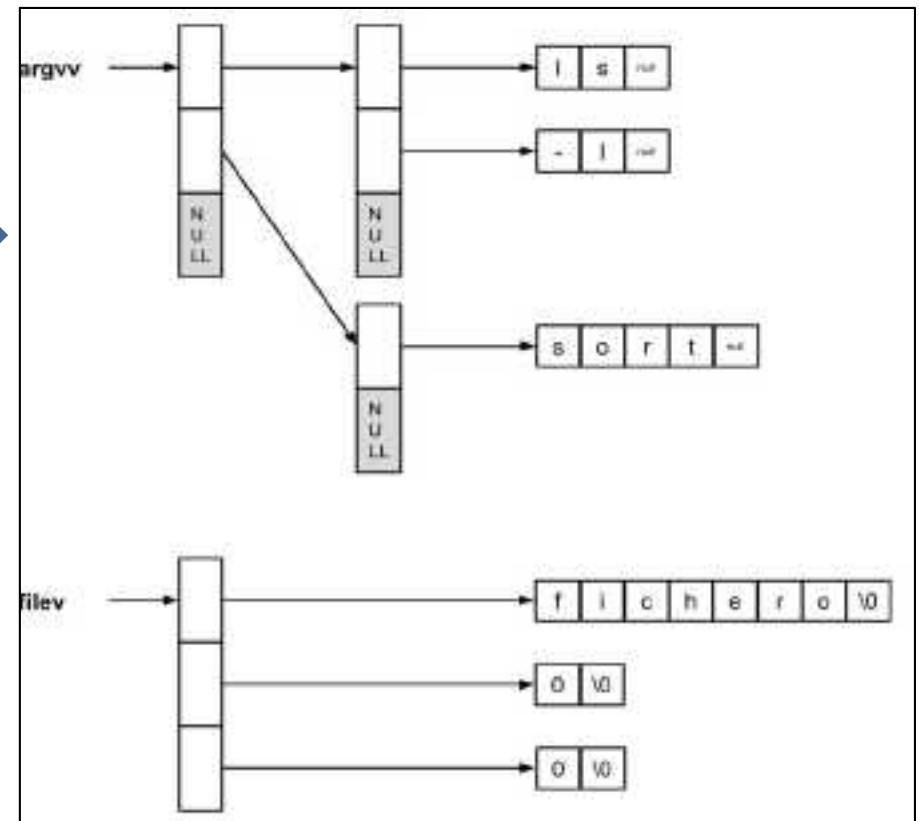
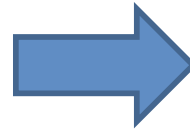
- Ejemplos:

□ ls		sort	Devuelve 2
□ ls		sort > fich	Devuelve 2
□ ls		sort &	Devuelve 2

Obtención de mandatos

7

```
ls -l | sort > fichero
```



Ejemplo de obtención de mandatos

```
command_counter = read_command(&argvv, filev, &in_background);  
  
for (int i=0; i< command_counter; i++) {  
    getCompleteCommand(argvv, i);  
    printf("%s\n", argv_execvp[0]);  
}
```

argv_execvp se utiliza en execvp así:

```
execvp (argv_execvp[0], argv_execvp);
```


Obtención de mandatos

9

- La función `read_command` retorna como segundo parámetro:

`char **filev`

Es una estructura que contiene los ficheros usados en las redirecciones.

<code>filev[0]</code>	Cadena que contiene el nombre del fichero usado para la redirección de entrada (<).
<code>filev[1]</code>	Cadena que contiene el nombre del fichero usado para la redirección de salida (>)
<code>filev[2]</code>	Cadena que contiene el nombre del fichero usado para la redirección de salida de error (!>).

Obtención de mandatos

10

- 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 (&)

Control de errores

11

- 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`.

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

- `perror` imprime el mensaje recibido como parámetro y a continuación el mensaje asociado al código del último error ocurrido durante una llamada al sistema.

Identificadores de procesos

12

- 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:

```
#include <sys/types.h>  
#include <stdio.h>
```

```
int main() {  
    printf("Identificador del proceso: %s\n", getpid());  
    printf("Identificador del proceso padre %s\n", getppid());  
    return 0;  
}
```

Descriptores de fichero de un proceso

13

- En UNIX / Linux todo proceso tiene abiertos tres descriptores de fichero por defecto:

□ Entrada estándar	Valor = 0	(STDIN_FILENO)
□ Salida estándar	Valor = 1	(STDOUT_FILENO)
□ Salida de error estándar	Valor = 2	(STDERR_FILENO)

- Tabla de descriptores de un proceso cuando se crea:

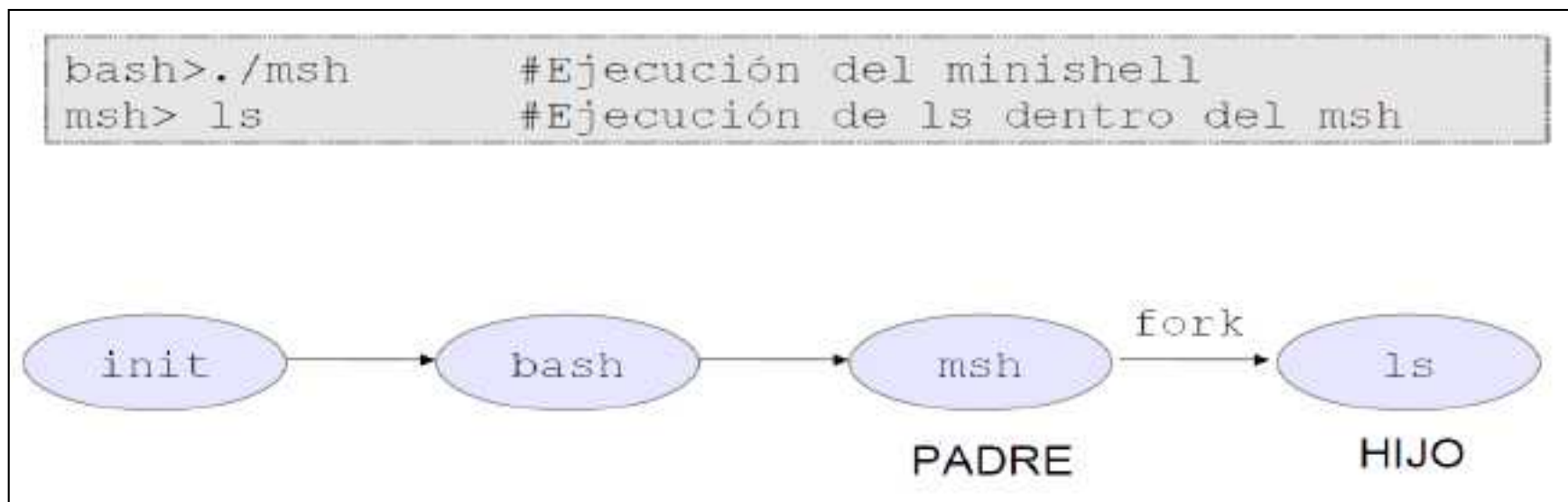
0	STD_IN
1	STD_OUT
2	STD_ERR

- Los mandatos que se ejecutan en una shell están escritos 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.

Procesos necesarios en el minishell

14

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



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

Creación de procesos con `fork()`

15

- 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 creación de procesos con fork()

16

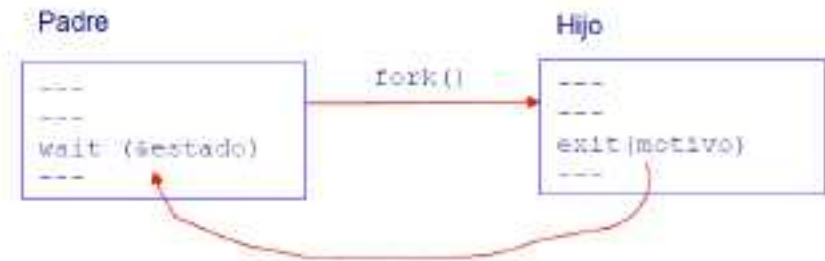
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int pid;
    int estado;
    pid = fork();
    switch(pid) {
        case -1: /* error */
            perror ("Error en el fork");
            return (-1);

        case 0: /* hijo */
            printf("El proceso HIJO se duerme 10 segundos\n");
            sleep(10);
            printf("Fin del proceso HIJO\n");
            break;

        default: /* padre */
            if (wait(&estado) == -1) //el padre espera por el hijo
                perror("Error en el wait");
            printf("Fin del proceso PADRE\n");

    }
    exit(0);
}
```



Ejecución de procesos con `execvp ()`

17

- La función `execvp` reemplaza la imagen del proceso que la invoca con una nueva. Esta imagen nueva corresponderá al mandato que se 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 (argumentos del programa `main`)
- 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`.

Ejemplo de ejecución de procesos con `execvp ()`

18

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int pid;
    char *argumentos[3] = {"ls", "-l", "NULL"};

    pid = fork();
    switch(pid) {
        case -1: /* error */
            perror("Error en el fork");
            exit(-1);
        case 0: /* hijo */
            execvp(argumentos[0], argumentos);
            perror("Error en el exec. Si todo ha ido bien esto nunca debería ejecutarse");
            break;
        default: /* padre */
            printf("Soy el proceso padre\n");
    }
    exit (0);
}
```

Finalización y espera de procesos

19

- 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 procesos.

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

- Normalmente los procesos padres siempre esperan a que finalizen 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 <pid>
```

→ Permite matar un proceso.

Ejemplo de finalización y espera de procesos

20

```
#include <sys/type.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main() {
    int pid;
    int status;
    char *argumentos[3] = {"ls", "-l", "NULL"};
    pid = fork();
    switch(pid) {
        case -1: /* error */
            perror("Error en el fork");
            exit(-1);
        case 0: /* hijo */
            execvp(argumentos[0], argumentos);
            perror("Error en el exec. Si todo ha ido bien esto nunca debería ejecutarse");
            break;
        default: /* padre */
            while (wait(&status) != pid);
            if (status == 0) printf("Ejecución normal del hijo\n");
            else printf("Ejecución anormal del hijo \n");
    }
    exit (0);
}
```

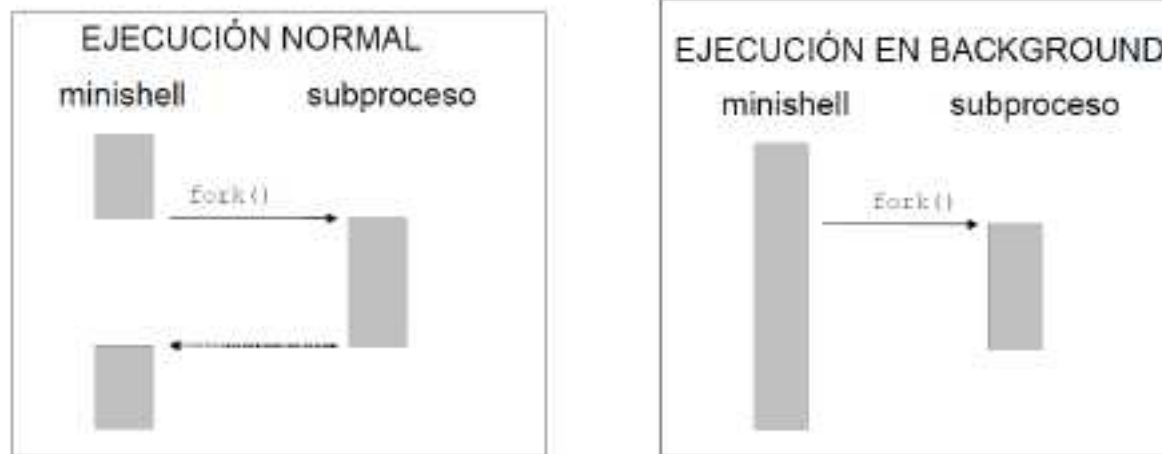
Ejecución en background

21

- Un mandato puede ser ejecutado en background desde la línea de comandos indicando al final un **&**. Por ejemplo:

```
sleep 10 &
```

- En este caso el proceso padre no se bloquea esperando la finalización del proceso hijo.



- La orden `fg <job_id>` permite recuperar un proceso en background. Recibe un id de trabajo, no un pid.
 - ▣ `fg` → No entra en esta práctica

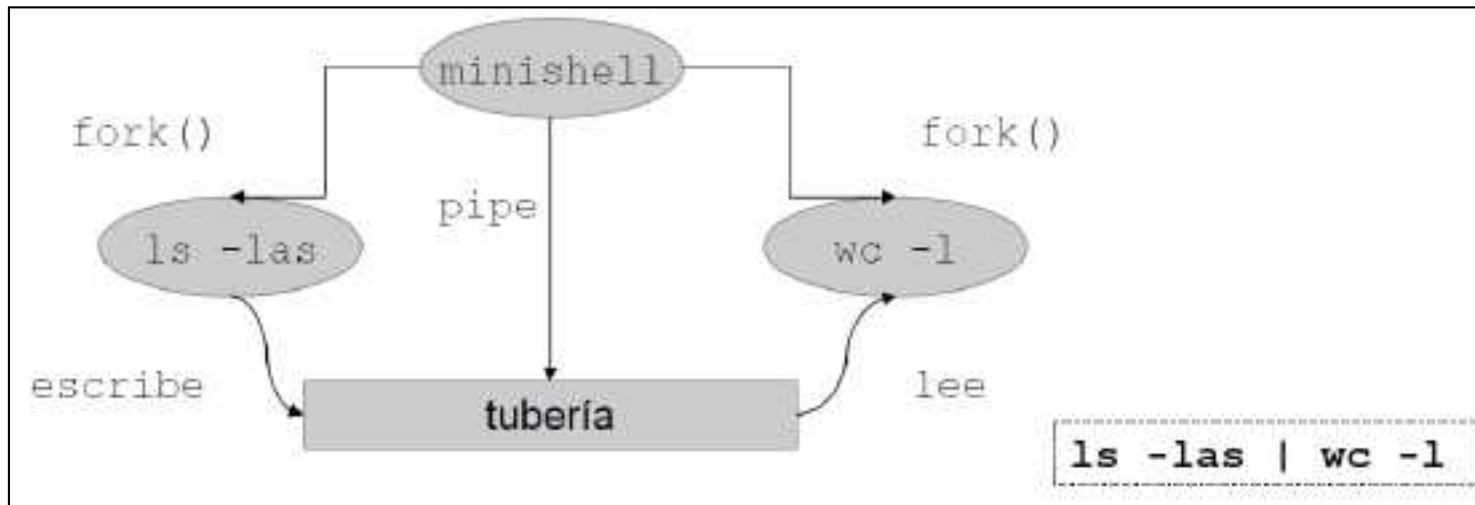
Secuencias de mandatos con tuberías

22

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

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

- ¿Cómo funciona una tubería?
 - 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()`

23

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

```
#include <unistd.h>
```

```
int pipe(int descf[2])
```

Devuelve:

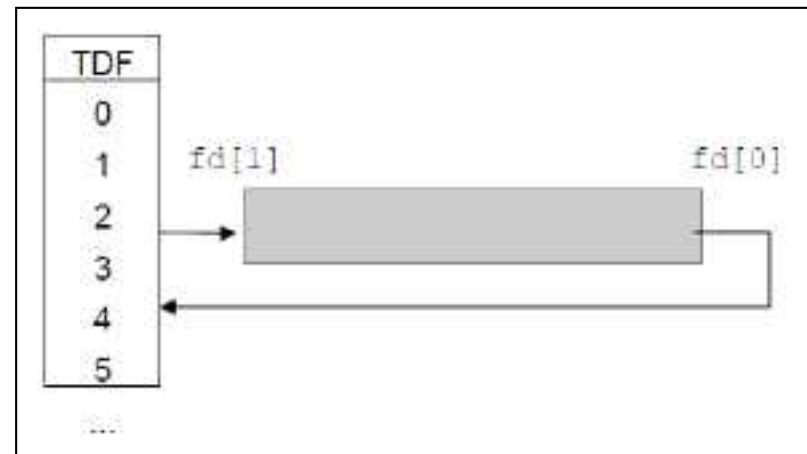
-1 → Si error.

0 → En cualquier otro caso.

- Recibe un array con los descriptors de ficheros para entrada y salida.

`descf[0]` → Descriptor de entrada (lectura).

`descf[1]` → Descriptor de salida (escritura).



Primitivas dup y dup2

24

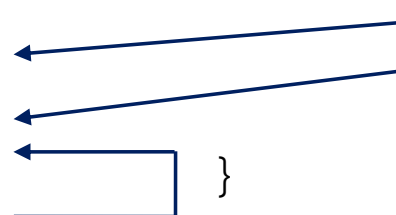
- Las primitivas dup y dup2 permiten duplicar descriptors de ficheros.

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

La primitiva dup utiliza el primer descriptor disponible de la tabla al abrir 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

25

1. **pipe**

```
pipe(pipe)
```

1. **close**

```
close(STDOUT_FILENO)
```

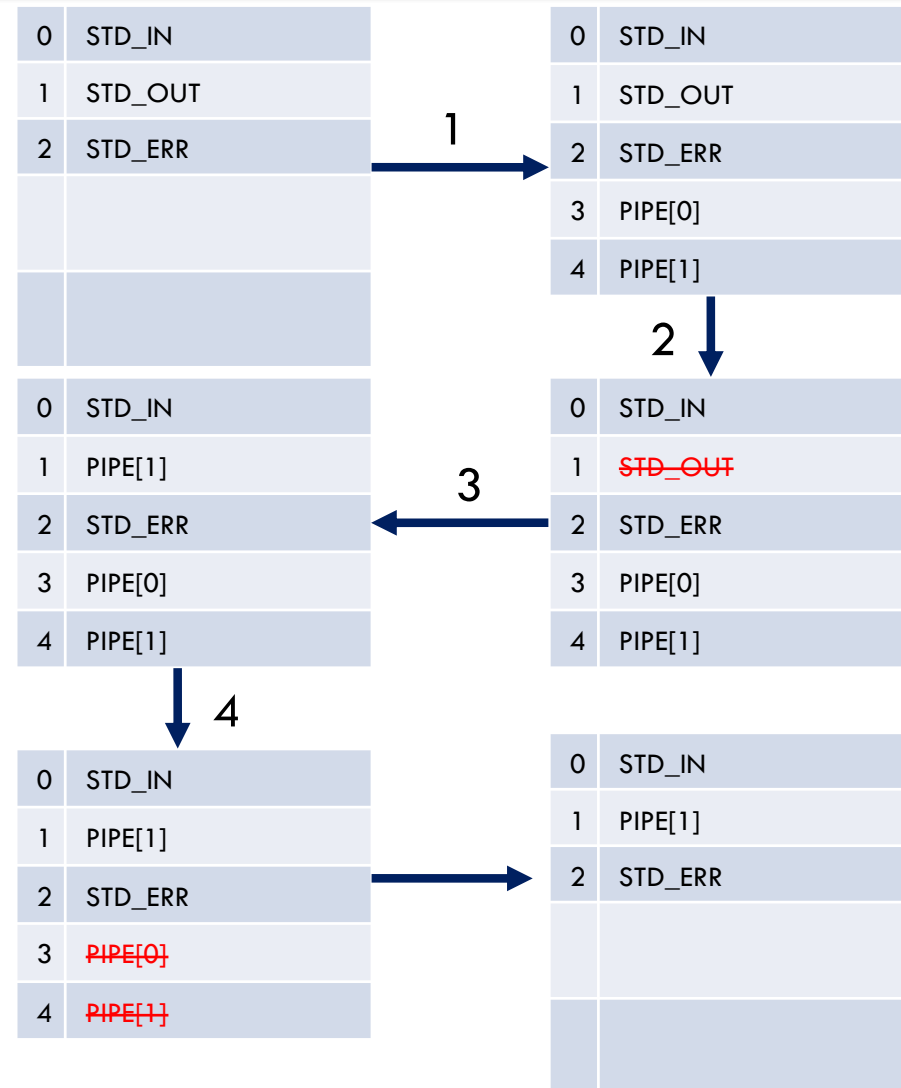
1. **dup**

```
dup(pipe[1])
```

1. **close**

```
close(pipe[0])
```

```
close(pipe[1])
```

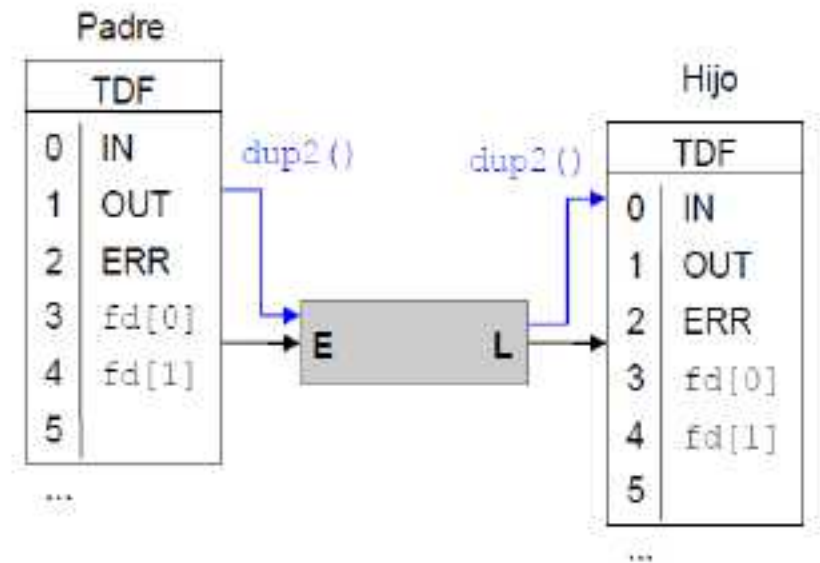


Ejemplo de uso de tuberías

26

- Ejemplo de tubería para el comando: `ls | more`. El hijo ejecuta el comando `more` el padre ejecuta el comando `ls`.

```
int main (int argc, char *argv[]) {
    int fd[2];
    char *argumentos1[2]={ "more", "NULL" };
    char *argumentos2[3]={ "ls", "NULL" };
    pipe(fd);
    if (fork() == 0) {
        close(STDIN_FILENO);
        dup(fd[0]);
        close(fd[1]);
        execvp(argumentos1[0], argumentos1);
    }
    else {
        close(STDOUT_FILENO);
        dup(fd[1]);
        close(fd[0]);
        execvp(argumentos2[0], argumentos2);
    }
    printf("ERROR: %d\n", errno);
}
```



Redirecciones de entrada, salida y error

27

- Es posible redireccionar la entrada / salida estándar para escribir / leer de otros ficheros.

- La redirección de entrada (<) sólo afecta al primer mandato.

- Abre un fichero en modo lectura y lo usa como entrada estándar.

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

- La redirección de salida (>) sólo afecta al último mandato.

- Abre un fichero en modo escritura y lo usa como salida estándar.

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

- Las redirecciones de salida error (!>) afectan a cualquier mandato.

- 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);
```

La primitiva open utiliza el primer descriptor disponible de la tabla al abrir un fichero.

Mandatos internos

28

- 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.

Mandatos internos del minishell: `mycalc`

29

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

`mycalc <operando 1> <add/mod> <operando 2>`

- Para ello debe:

- ▣ Comprobar que la sintaxis es correcta.
- ▣ Ejecutar la operación elegida: suma (add) o módulo (mod) .
- ▣ Mostrar por pantalla los resultados.

```
msh> mycalc 3 add -8
```

```
[OK] 3 + -8 = -5; Acc -5
```

```
msh> mycalc 5 add 13
```

```
[OK] 5 + 13 = 18; Acc 13
```

```
msh> mycalc 10 mod 7
```

```
[OK] 10 % 7 = 7 * 1 + 3
```

```
msh> mycalc 10 % 7
```

```
[ERROR] La estructura del comando es <operando 1> <add/mod>  
<operando 2>
```

```
msh> mycalc 8 mas
```

```
[ERROR] La estructura del comando es <operando 1> <add/mod>  
<operando 2>
```

Mandatos internos del minishell: mycp

30

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

`mycp <fichero origen> <fichero destino>`

- Para ello debe:

- ▣ Comprobar que la sintaxis es correcta.
- ▣ Comprobar que el fichero existe y leerlo.
- ▣ Crear un nuevo fichero con el nombre definido y escribir en él.
- ▣ Cerrar los ficheros abiertos

```
msh> mycp msh.c msh.c_bak
```

```
[OK] Copiado con exito el fichero msh.c a msh.c_bak
```

```
msh> mycp inexistente.c fichero
```

```
[ERROR] Error al abrir el fichero origen
```

```
: No such file or directory
```

```
msh> mycp origen.txt
```

```
[ERROR] La estructura del comando es mycp <fichero  
origen>      <fichero destino>
```

Entrega

31

- **Martes 28 de marzo 2020 (hasta las 23:55h).**
- Fichero zip con el nombre **ssoo_p2_AAA_BBB_CCC.zip** que contiene:
 - Msh.c
 - Autores.txt
- Memoria: se entrega mediante turnitin.
- Se deben seguir las normas incluidas en el enunciado. Leedlo cuidadosamente.