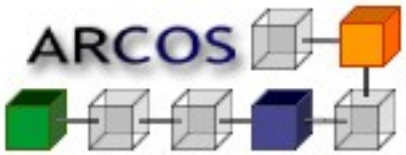


Práctica 1: Intérprete de mandatos

Introducción

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

```
ls, cp, mv, rm, etc.
```

  - Ejecución de secuencias de mandatos (límite 3 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 | grep mio > fichero
```

```
cat | more < fichero
```

```
make install 2> salida_error
```

Proceso de desarrollo

- Se recomienda un **desarrollo incremental**.
  - Soporte para mandatos simples: `ls, cp, mv, etc`.
  2. Soporte para redirecciones sobre mandatos simples.
  3. Soporte para ejecución de mandatos simples en background (&).
  4. Soporte de secuencias de mandatos.
  5. Secuencias de mandatos en background (&).
  6. Secuencias de mandatos con redirecciones.
  7. Mandato interno

     `cd`

Material proporcionado

- Los ficheros proporcionados son:

```
ssoo/
    msh/
        y.c
        Makefile
        parser.y
        scanner.l
        main.c
```

- Para compilar la práctica simplemente ejecutar el comando `make`.
- El alumno sólo debe:
  - Modificar el fichero `main.c` para incluir la funcionalidad pedida.

Obtención de mandatos

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

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

- Devuelve
  - **0** En caso de EOF (CTRL + D)
  - **-1** En caso de error
  - **n** Número de mandatos tecleados más uno.
- Ejemplos:
  - `ls | sort` → Devuelve 3
  - `ls | sort > fich` → Devuelve 3
  - `ls | sort &` → Devuelve 3

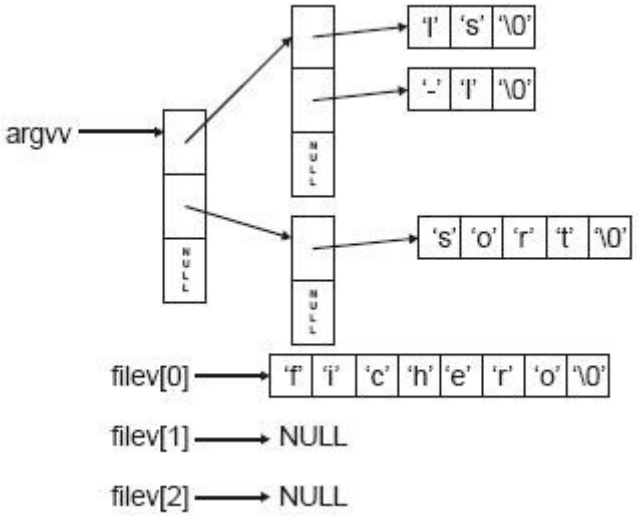
Obtención de mandatos

- En los parámetros retorna:

```
char ***argvv
```

Estructura que contiene los mandatos introducidos por el usuario.

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



- Ejemplo:
  - Imprimir el mandato `i`:

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

  - Imprimir el mandato `i` y sus opciones:

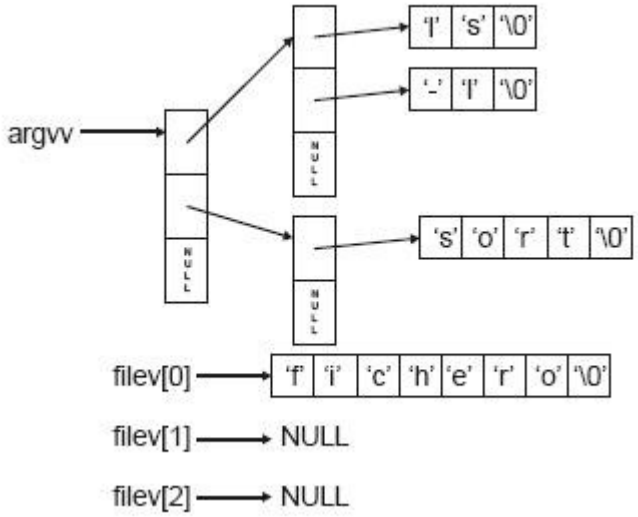
```
printf("Mandato i: %s \n", argvv[i]);
```

Obtención de mandatos

- En los parámetros retorna:

```
char **filev
```

Estructura que contiene los ficheros usados en redirecciones.



- `filev[0]`

  Puntero a cadena que contiene la redirección de entrada (`<`).

- `filev[1]`

  Puntero a cadena que contiene la redirección de salida (`>`).

- `filev[2]`

  Puntero a cadena que contiene la redirección de salida de error (`>&`).

Obtención de mandatos

- En los parámetros retorna:

```
int *bg
```

- Devuelve:

  0 Si no se ejecuta en background

  1 Si se ejecuta en background (&)

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`.

```
#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.

Creación de procesos en el minishell

- 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);
```

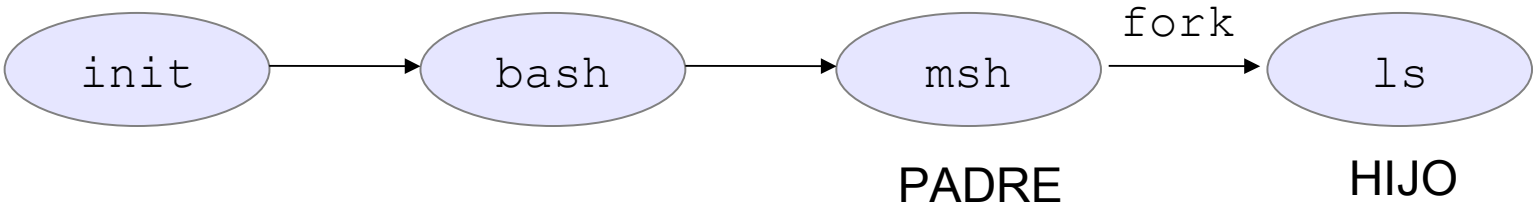
- Un ejemplo:

```
#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;
}
```

Creación de procesos en el minishell

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

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



Llamada al sistema fork()

- Permite generar un nuevo proceso o proceso hijo es una copia exacta del proceso padre. El formato de la llamada *fork()* es :

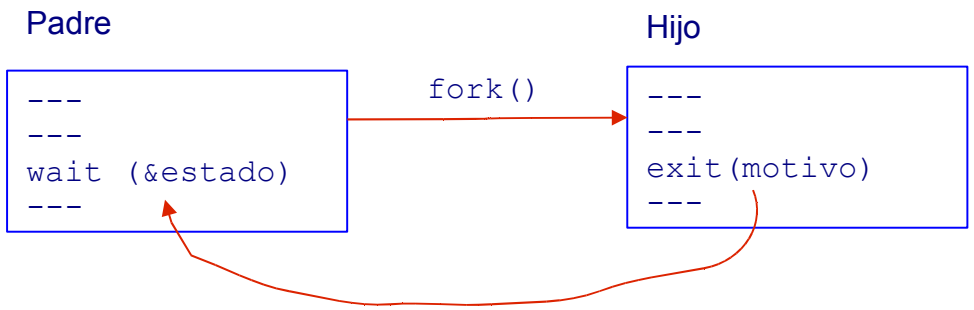
```
#include <sys/types.h>
#include <unistd.h>

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 llamada al sistema `fork()`

```
#define ERROR -1
int main() {
    int pid;
    int estado;

    switch(fork()) {
        case ERROR:
            perror ("fork");
            return (-1);
        case 0 :
            system ("ps -f");
            printf("Fin del proceso HIJO \");
            exit(1);
        default :
            if (wait(&estado)==ERROR)
                errorsistema("wait");
            printf("Fin del proceso PADRE\n");
    }
    return 0;
}
```



Ejecución de procesos (execvp)

- La familia de funciones `exec` reemplaza la imagen del proceso en curso con una nueva.

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

- Argumentos:
  - `file` → Camino del fichero 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 regresa ha ocurrido un error.
  - Devuelve -1 y el código de error está en la variable global `errno`.

Ejemplo llamada al sistema `execvp()`

- Un ejemplo del uso de `execvp`.

```
#include <sys/type.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");
            break;
        default:  /* padre */
            perror ("Soy el proceso padre");
    }
    exit (0);
}
```

Finalización y espera de procesos

- La finalización de un proceso puede hacerse con la sentencia:
  - `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 termina pasa a estado ZOMBIE.

```
ps -axf
kill -9 <pid>
```

Ejemplo llamada de finalización y espera

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

int main(int argc, char **argv) {
    int pid;
    int status;

    pid = fork();
    switch(pid) {
        case -1:  /* error */
            exit(-1);
        case 0:   /* hijo */
            if (execvp(argv[1], &argv[1]) < 0){
                perror("Error en el exec");
                exit(-1);
            }
            break;
        default:  /* padre */
            while (wait(&status) != pid);
            if (status == 0) printf("Ejecución normal \n");
            else printf("Ejecución anormal \n");
    }
    exit (0);
}
```

Redirecciones de ficheros

- En UNIX / Linux todo proceso tiene abiertos tres descriptores de fichero por defecto:
  - Entrada estándar fd=0 (`STDIN_FILENO`)
  - Salida estándar fd=1 (`STDOUT_FILENO`)
  - Salida de error estándar fd=2 (`STDERR_FILENO`)

- Los mandatos del intérprete de comandos están escritos para leer y escribir de la entrada / salida estándar.

- La llamada a `fork()` conserva todas las tablas de descriptores. El proceso hijo hereda los descriptores de fichero.

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

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

Redirecciones de salida y error

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

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

- Las redirecciones de salida error (>&) sólo afectan al último mandato.
- Abre un fichero en modo escritura y lo usa como salida estándar.

```
close (STERR_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.

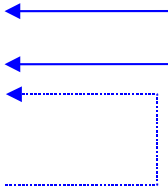
Primitivas dup y dup2

- Las primitivas `dup` y `dup2` permiten duplicar descriptores 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	

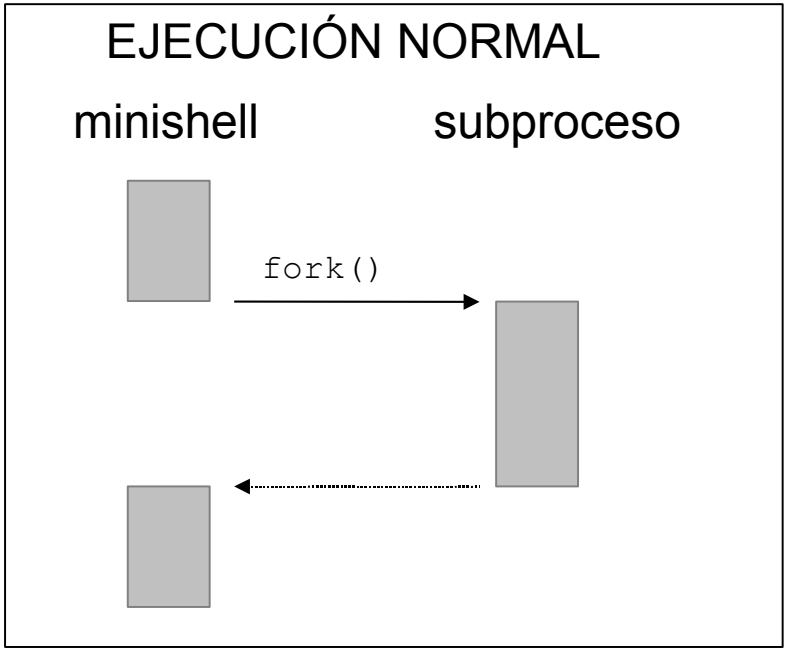


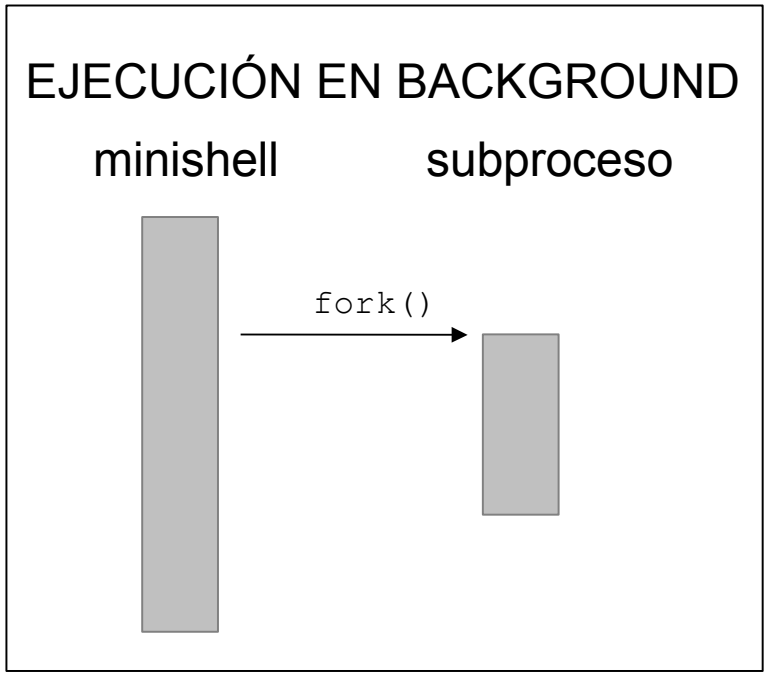
```
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd1, fd2, fd3;
    fd1 = open("./file_a", O_READ");
    fd2 = open("./file_b", O_READ");
    fd3 = dup(fd2);
}
```

Ejecución en background

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

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





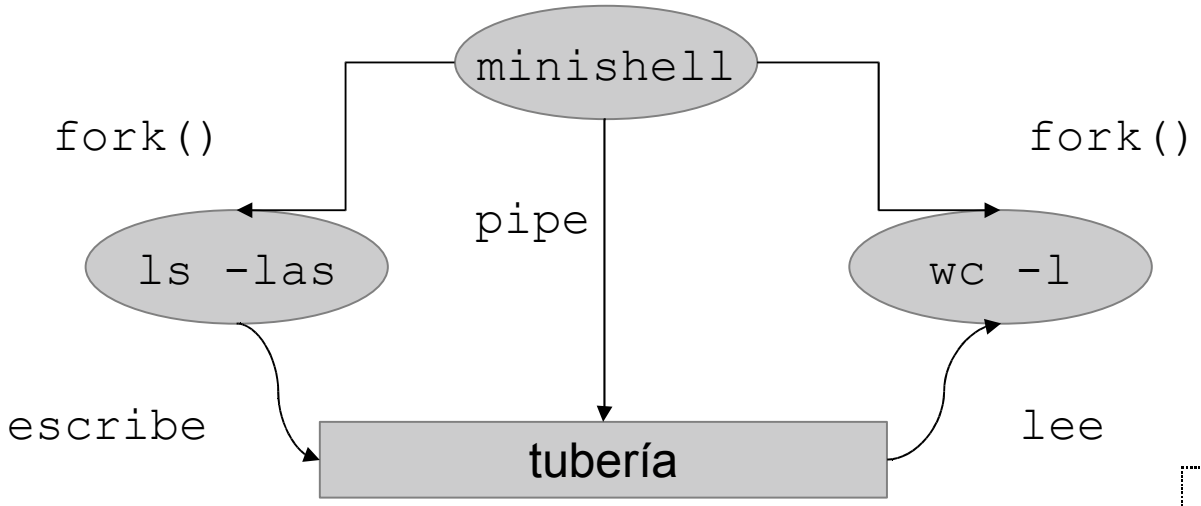
- Las órden `fg <job_id>` permite recuperar un proceso en background. **Recibe un id de trabajo**, no un pid.

Secuencias de mandatos

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

```
ls -l | 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.




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

Creación tuberías. Primitiva `pipe`

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

```
#include <unistd.h>

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

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

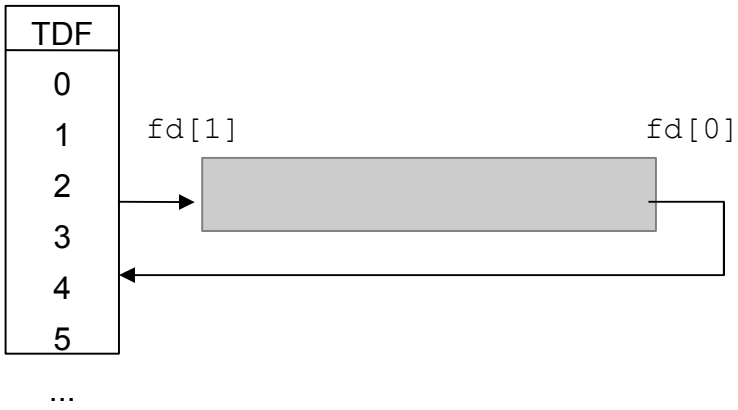
`descf[0]` → Descriptor de entrada (lectura).

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

- Devuelve:

-1 → Si error

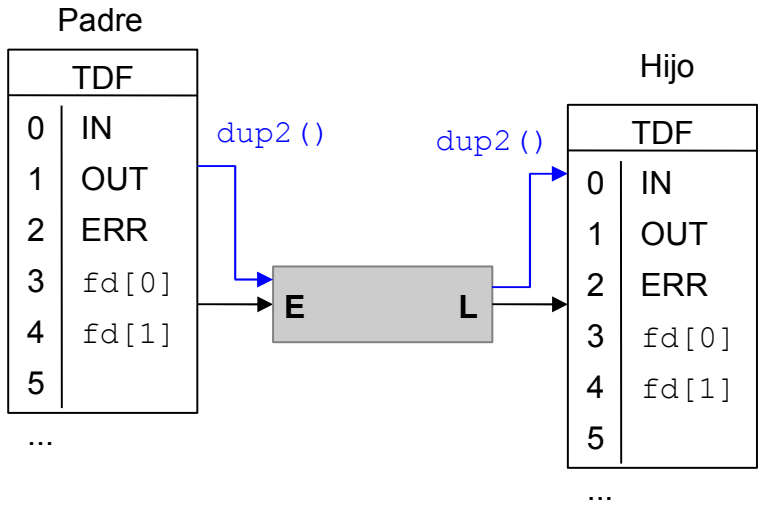
0 → En cualquier otro caso



Ejemplo sencillo de uso de tuberías

- Ejemplo de tubería para el comando: `ls -l | more`

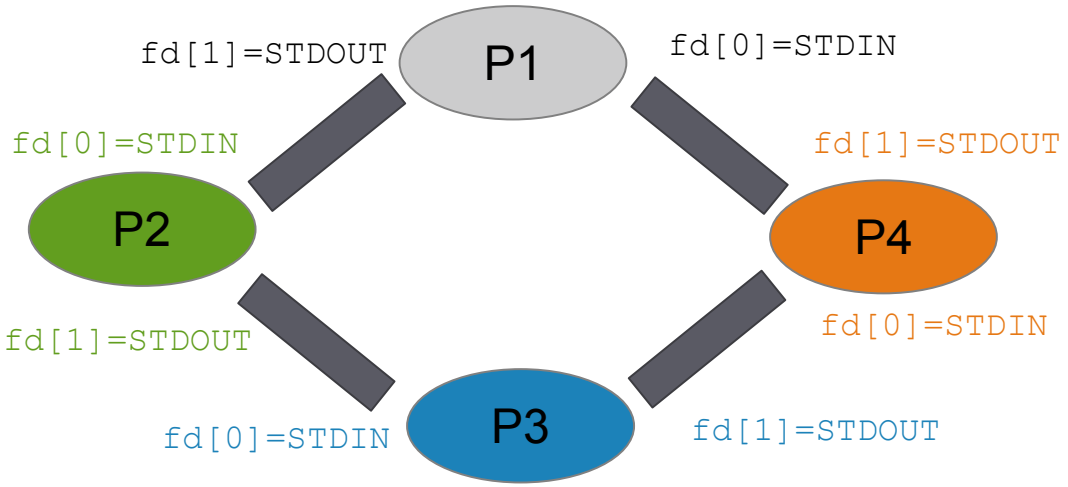
```
int main (int argc, char *argv[]) {
    int fd[2];

        pipe(fd);

        if (fork() == 0) {
            close(0);
        dup(fd[0]);
            close (fd[1]);
            execlp ("more","more", NULL);
      }
    else {
        close(1);
            dup(fd[1]);
            close (fd[0]);
            execlp ("ls", "ls", "-l", argv[1], NULL);
      }
    printf("ERROR: %d\n",errno);
}
```



Anillo de procesos

- Creación de un anillo de procesos con redirección de entrada / salida estándar.

```
for(i=1;i<nprocs;i++) {
    pipe(fd)
    pidhijo=fork();

    if (pidhijo>0)   /* padre */
        dup2(fd[1],STDOUT_FILENO);
    else   /* hijo */
        error = dup2(fd[0],STDIN_FILENO);
    if (pidhijo)
        break; /* el padre muere */
}
```



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 ejecutará:
  - En un proceso hijo (`fork`)
    - Si el mandato interno es invocado en background (&)
    - Si aparece en una secuencia de mandatos y no es el último
  - En el proceso padre
    - En el resto de los casos

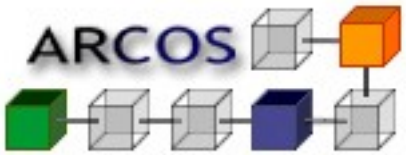
Mandatos internos del minishell: `cd`

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

```
cd

cd <directorio>
```

- Para ello debe:
  - Comprobar que la sintaxis es correcta.
  - Ejecutar el cambio de directorio.
  - Mostrar por pantalla la ruta actual.

  Ej.
```
msh> cd
/home/my_user
```



Práctica 1: Intérprete de mandatos