OPERATING SYSTEMS

Lab 2. Minishell

Introduction

- Development of a minishell in UNIX/Linux in C language.
- It must implement:
  - Execution of simple commands

```
ls, cp, mv, rm, […].
```

  - Execution of command sequences

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

  - Execution of simple commands or sequences in background (&)

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

  - Execution of simple commands or sequences with input, output and error redirections.

```
ls > file
cat | more < file
ls | grep mio > my_entries
make install 2> error_output
```

Development process

- An **incremental development** is recommended.
  1. Support for simple commands: `ls, cp, mv, […].`
  2. Support for simple commands in background (&).
  3. Execution of simple commands with redirections.
  4. Support for command sequences.
  5. Support for command sequences in background (&).
  6. Support for redirections in simple commands and sequences.
  7. Internal command
     `mytime, mypwd`

Material

- For teh development of the lab an initial code will be provided. This code can be downloaded from AulaGlobal.
- The files given are:

```
p2_minishell_2019/
                  y.c
                  Makefile
                  parser.y
                  scanner.l
                  msh.c
                  unzip_script.sh
                  corrector_ssoo_p2-v2.sh
```

- To compile simply execute `make`.
- The student must only:
  - Modify `msh.c` to include the asked functionality

Getting Commands

- For obtaining the orders a sintactical analizer is used. It checks if the order sequence has the correct structure and allows you to get the content through a function.

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

- It returns

0	if EOF (CTRL + D)
-1	when an error happens
n	Number of orders plus one

- Examples:
  - `ls | sort` → Returns 3
  - `ls | sort > fich` → Returns 3
  - `ls | sort &` → Returns 3
  - `cat < input_file` → Returns 2

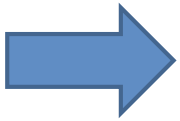
Getting Commands

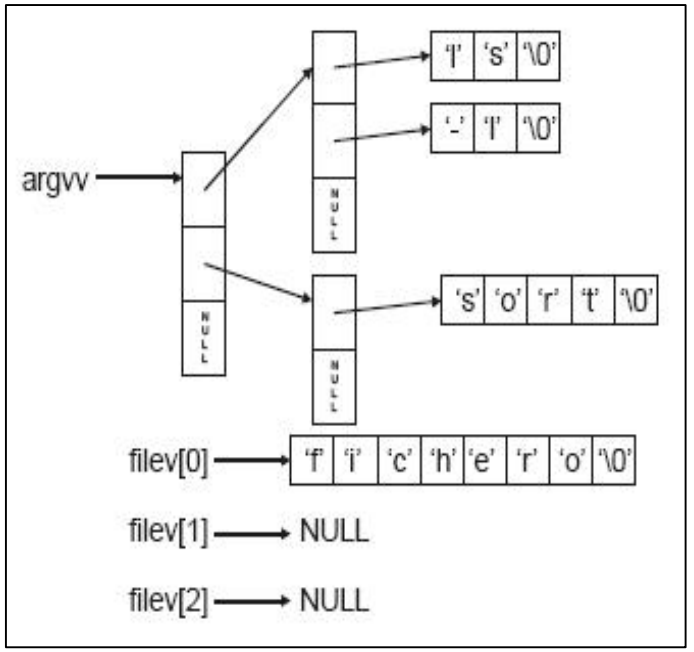
- The function `obtain_order` returns as first parameter:

  `char ***argvv`

  It is a structure that contains the orders introduced by the user.

```
ls -l | sort > file
```





- Example:
  - Print command i:

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

  - Print its first argument:

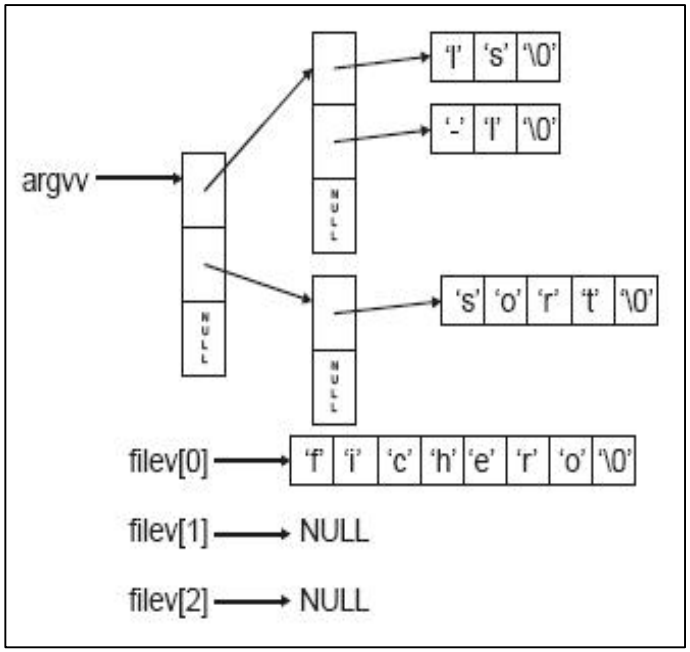
    ```
    printf("Arg 1 de i: %s \n", argvv[i][1]);
    ```

Getting Commands

- The function `obtain_order` returns as second parameter:

  `char **filev`

  It is a structure that contains the files used in the redirections.



- `filev[0]`

  String that contains the name of the file used for the input redirection (<).

- `filev[1]`

  String that contains the name of the file used for the output redirection (>).

- `filev[2]`

  String that contains the name of the file used for the error output redirection (>&).

Getting Commands

- The function `obtain_order` returns as third parameter:

```
int *bg
```

  It is a variable that indicates if the commands are executed in background.

- Its values are :

  `bg = 0` → If it is not executed in background

  `bg = 1` → If it is executed in background (&)

Error control

- When a system call fails it returns -1. The error code is inside the global variable `errno`.
- Inside the file `errno.h` you can find all the possible values that it can take.
- To access to the error code there are two possibilities:
  - Use errno as an index to access to the chain `sys_errlist[]`.
  - Use the library function `perror()`. See `man 3 perror`.

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

- `perror` prints the received message as parameter and prints the message associated to the code of the last error ocurred during a system call.

Process identifiers

- A process is a *program in execution*
- All processes have a unique identifier. Two functions let you recover that identifier:

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

- An example:

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

int main() {
   printf("Procces identifier: %s\n", getpid());
   printf("Procces identifier of the parent process%s\n",
   getppid());
   return 0;
}
```

File descriptors of a process

- In UNIX / Linux all processes have three opened file descriptors by default:
  - Standard input — Value = 0 — (STDIN_FILENO)
  - Standard ouput — Value = 1 — (STDOUT_FILENO)
  - Standard error — Value = 2 — (STDERR_FILENO)

- File descriptor table of a process when it is created:

0	STD_IN
1	STD_OUT
2	STD_ERR

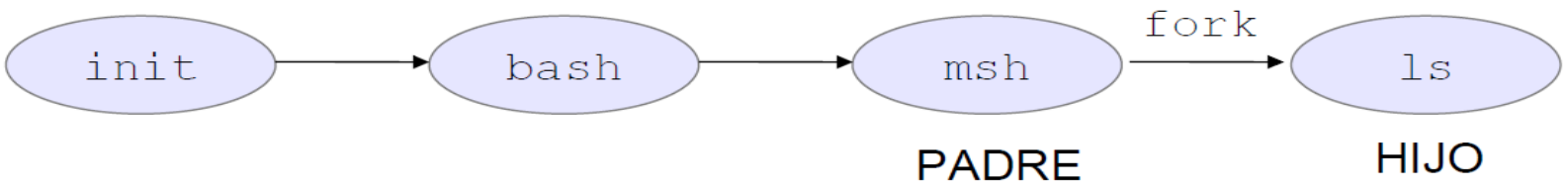
- The commands executed in a shell are writtent to read and write from the standard input /ouput.

- It is possible to redirection the standard input / output to read /write from other files, or to read / write in a pipe.

Needed processes in a shell

- In the minishell all the creation of new processes is done from the procces of the minishell.
- Example of execution of the order `ls`.

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



- Each command (for example: ls) will be executed in a process child of the minishell.

Creation of processes with `fork()`

- It allows to generate a new process or child process that is an exact copy of the parent procress:

```
pid_t fork()
```

It returns:

0 → If it is the children.

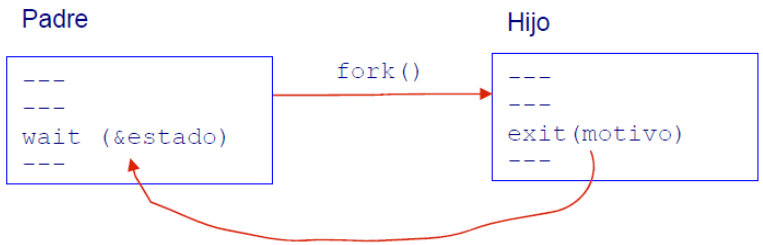
pid → If it is the parent.

- The child process inherites:
  - The values of signal manipulation.
  - The process class.
  - The segments of shared memory.
  - The masks of file creation, etc.
- The child process differs in:
  - The child has an ID of unique process.
  - **It has a private copy of the files descriptors opened by the parent.**
  - The pending signals of the child process is empty.
  - The child does not inherit the stablished locks of the parent.

Example of creation of processes with fork()


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

int main() {
     int pid;
     int estado;
     pid = fork();
     switch(pid) {
             case -1: /* error */
                          perror ("Error in fork");
                          return (-1);
             case 0: /* hijo */
                          printf("The process CHILD sleeps 10 seconds\n");
                          sleep(10);
                          printf("End of process CHILD\n");
                          break;
             default: /* padre */
                          if (wait(&estado) == -1) //the parent waits for the child
                                       perror("Error in the wait");
                          printf("End of process PARENT\n");
     }
     exit(0);
}
```



Process execution with `execvp()`

- The function `execvp` replaces the process image with a new one. This new image corresponds to the order tha you want to execute.

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

- Arguments:
  - `file` → Path of the file that contains the command to be executed . If there is not path, it searches inside the PATH.
  - `argv[]` → List of avalaible arguments for the new program. The first argument must point to the name of the file that is going to be executed.
- Return:
  - If the function returns something is because an error has happened.
  - It returns -1 and the error code is in the global variable `errno`.

Example of process execution with execvp( )


```
#include <sys/type.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 in fork”);
                        exit(-1);
            case 0: /* child */
                        execvp(argumentos[0], argumentos);
                        perror(“Error in exec. If the execution is correct this would never be
    executed.”);
                        break;
            default: /* parent */
                        printf(“I am the parent process\n”);
    }
    exit (0);
}
```

Finalization and waiting for processes

- The finalization of a process can be executed with:

```
return status;
void exit(int status);
void abort (void);
```
→ Abnormal finalization of the process.

- The processes can wait to the finalization of other processes.

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

- Normally, parent processes always wait for its children to finalize.

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

- If a process finalizes and its parent process have not waited for it, the procces goes to ZOMBIE state.

`ps axf` → It allows to visualize the zombie processes.

`kill -9 <pid>` → it allows to kill a process.

Example of finalization and process waiting

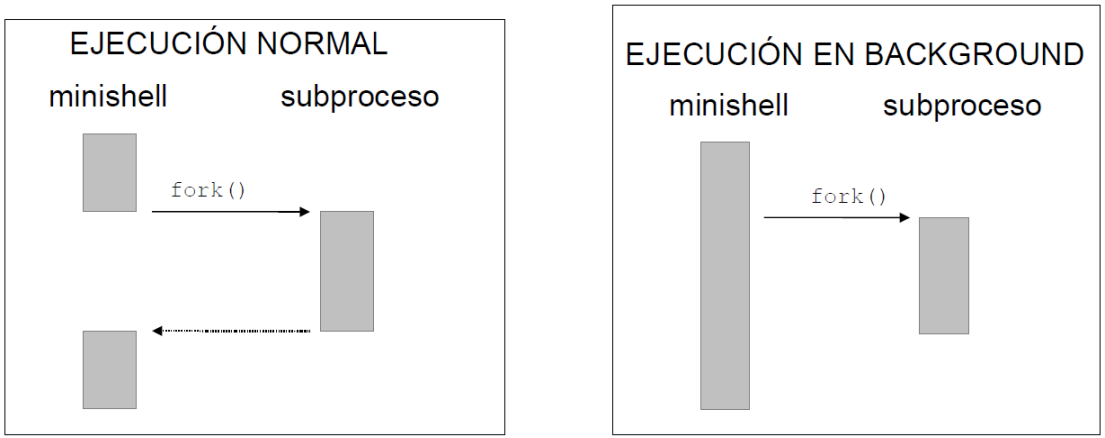

```
#include <sys/type.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main() {
     int pid;
     int status;
     char *argumentos[3] = {“ls”, “-l”, “NULL”};
     pid = fork();
     switch(pid) {
             case -1: /* error */
                          perror(“Error infork”);
                          exit(-1);
             case 0: /* child */
                          execvp(argumentos[0], argumentos);
                          perror(“Error in exec. If all is correct this should never be executed.”);
                          break;
             default: /* parent */
                          while (wait(&status) != pid);
                          if (status == 0) printf(“Normal execution of the child\n”);
                          else printf(“Abnormal execution of the child \n”);
     }
     exit (0);
}
```

Background execution

- A command can be executed in background from the command line using **& at the end**. For example:

```
sleep 10 &
```

- In this case the parent process does not block waiting for the finalization of the child process.



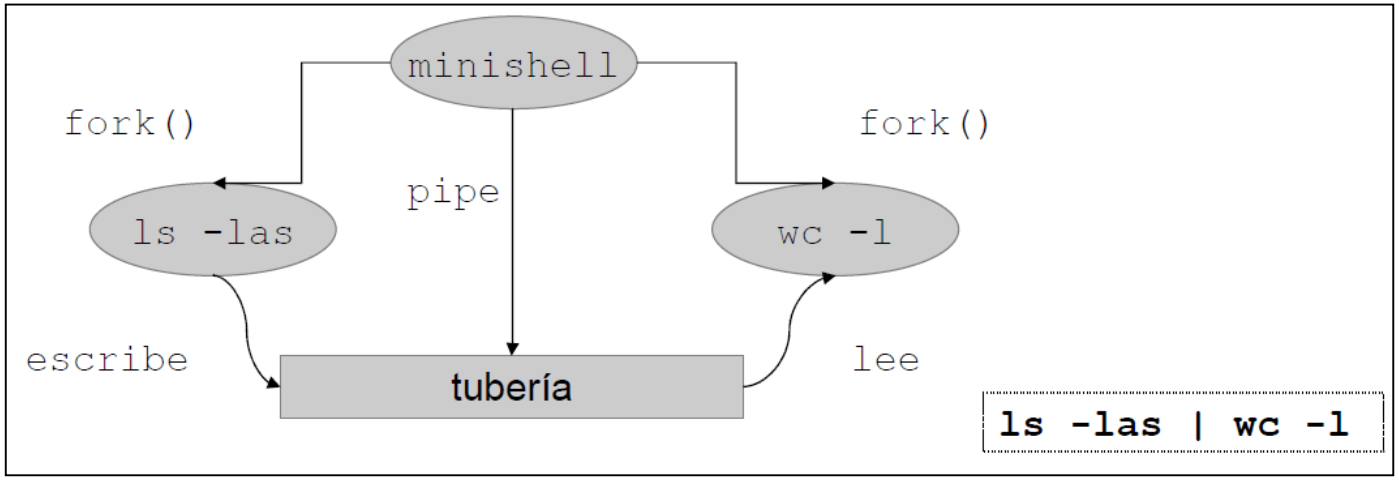
- The command `fg <job_id>` let you recover a process in background. It receives a job id, not a pid.
  - `fg` → It does not enter in this lab

Command sequences with pipes

- Command sequences are divided by a pipe | . For example:

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

- ¿How does a pipe work?
  - The standard ouput of each command is connected to the standard input of the next one.
  - The first command reads from the standard input (keyboard) if there is not an input redirection.redirección de entrada.
  - The last command writes in the standard output (screen) if there is no output redirection.



Creation of pipes with `pipe( )`

- For the creation of pipes without name you have to use the function `pipe`.

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

It returns:

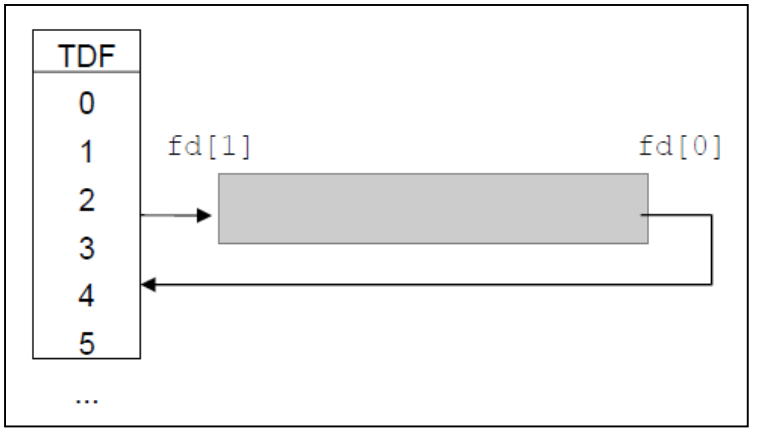
-1 → If there is an error.

0 → In any other case.

- It receives an array with the file descriptor for input and ouput.

`descf[0]` ⟶ Input Descriptor (read).

`descf[1]` ⟶ Output descriptor (write).

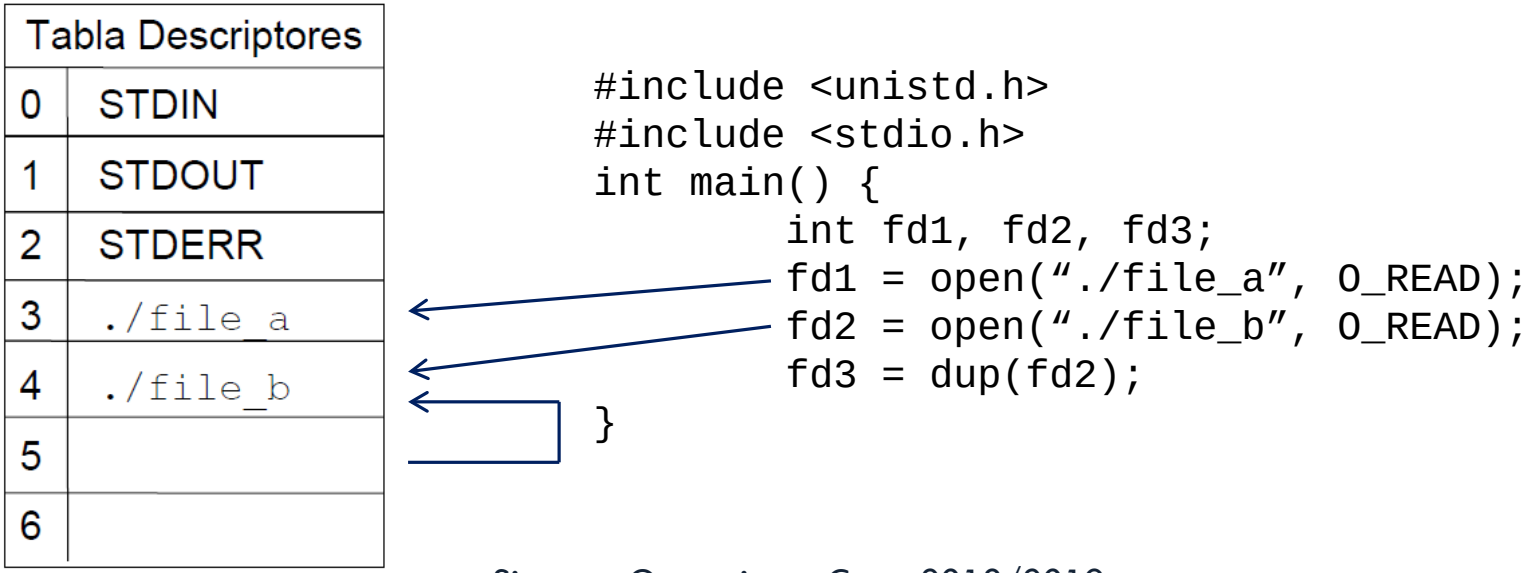


Functions dup and dup2

- The functions dup y dup2 let you duplicate the file descriptors.

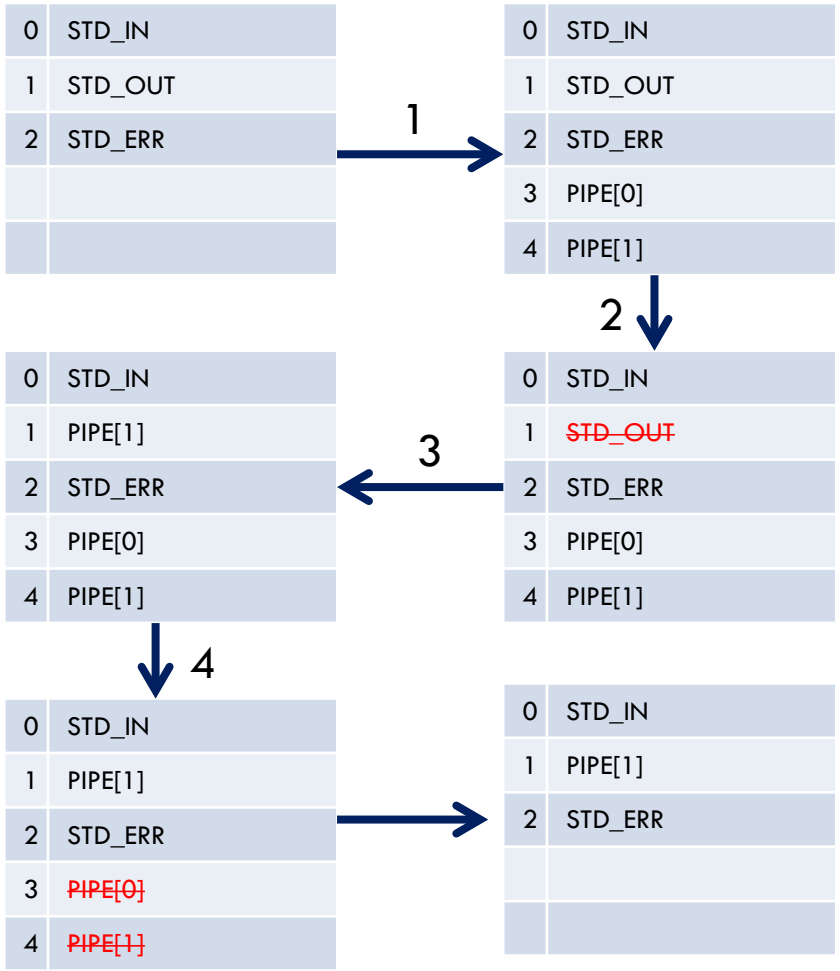
```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

The function dup uses the first descriptor avalaible in the table when opening a file.



Use of pipe + dup

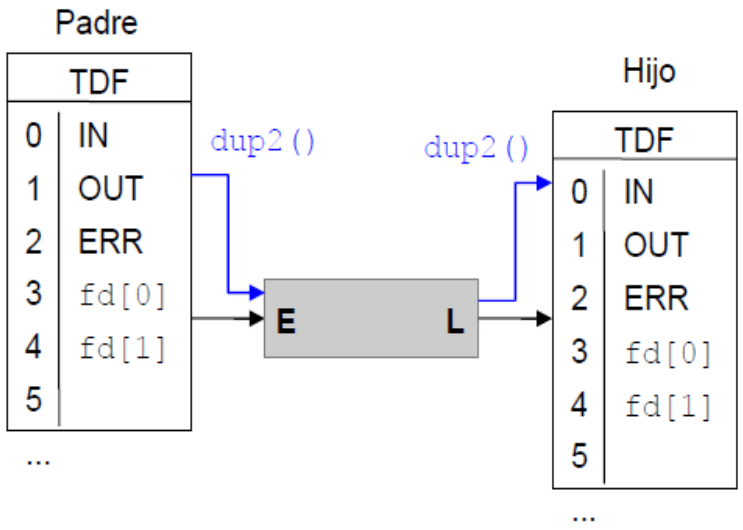
1. pipe
   ```
   pipe(pipe)
   ```
2. close
   ```
   close(STDOUT_FILENO)
   ```
3. dup
   ```
   dup(pipe[1])
   ```
4. close
   ```
   close(pipe[0])
   close(pipe[1])
   ```



Example of pipe use

- Example of pipe for the command : `ls | more`. The children executes the command `more` the parent executes the command `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);
}
```



Input, output and error redirections

- It is possible to redirect the input / ouput to write/ read from other files

  - The input redirection (<) only affects the first command.
    - It opens in read mode and uses it as standard input.

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

  - The output redirection (>) only redirects the last command.
    - It opens a file in write mode and uses it as standard output.

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

  - The redicrection of the standard error affects all commands (>&).
    - It opens a file in write mode and use it as standard error.

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

System call open uses the first file descriptor available in the files table.

Tip: Consider that previous to open the file (./input_file) the STDIN_FILENO was closed

Internal commands

- An internal command is a command that correspond to a system call or is a complement offered by the minishell.
- Its function must be implemented inside the minishell.
- The command input must be analyzed. ¡¡The parser does not do it!!
- It must be executed inside the process minishell.

Internal commands of the minishell: `mytime`

- The minishell must provide the internal command `mytime`.
  - It receives one argument: a command with its own arguments.
  - It shown on the screen the execution time of a process.
  - Output:
    - Correct execution: "`Time spent: %f secs.\n`".
    - Error: "`Usage: mytime <command <args>>\n`".

- This internal command must:
  - Start measuring time.
  - Launch a process which executes the command given by user.
  - When the child process ends, stop measuring time and display the information.

Internal commands of the minishell: `mypwd`

- The minishell must provide the internal command `mypwd`.
  - It does not receives arguments
  - It shows on the screen the current working directory.
  - Output:
    - Correct execution: "`Current dir: %s\n`".
    - Error: "`Mypwd error\n`".

- It is recommended using `getcwd()`.