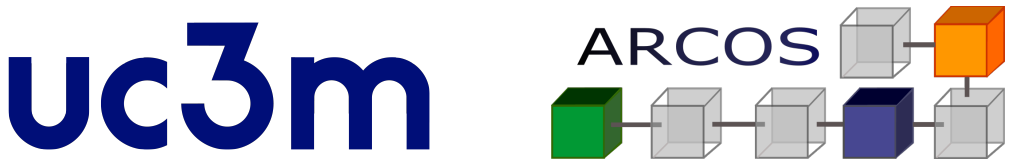



Computer Architecture and Technology Area

Universidad Carlos III de Madrid

# OPERATING SYSTEMS
# Lab 2. Programming a shell (Minishell)

Bachelor's Degree in Computer Science & Engineering
Bachelor's Degree in Applied Mathematics & Computing
Dual Bachelor in Computer Science & Engineering & Business Administration

Year 2023-2024

# Contents

# Contents

# Description

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

    `cat | more < file`

    `ls > file`

    `ls | grep mine > my_inputs`

    `make install !> output_error`

# Development process

1. Support for simple commands: `ls, cp, mv, [...]`.
2. Support for simple commands in background (&).
3. Support for command sequences: `ls | wc -l, [...]`.
4. Support for command sequences in background (&).
5. Support for redirections in simple commands and sequences (<, >, !>).
6. Internal command:
   - `mycalc`
   - `myhistory`

# Contents

# Initial Code

- For the development of the programming assignment, an initial code is provided. This code can be downloaded from Aula Global.

- The files given are:

```
p2_minishell_23_24/
    Makefile
    libparser.so
    msh.c
    checker_os_p2.sh
    authors.txt
```

- To compile, simply execute `make`, and export the path for the dynamic library .

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

- The student must only **modify msh.c to include the asked functionality**.

# Contents

# Command preprocessing

- For obtaining the commands, a syntactic analyzer is used. It checks if the commands sequence has the correct structure and allows you to get the content through a function:

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

- Returns:
  - 0 $\longrightarrow$ If EOF (CTRL + C).
  - -1 $\longrightarrow$ When an error happens.
  - n $\longrightarrow$ Number of commands.
- Examples:
  - `ls | sort` $\longrightarrow$ Returns 2.
  - `ls | sort > fich` $\longrightarrow$ Returns 2.
  - `ls | sort &` $\longrightarrow$ Returns 2.
  - `cat < input_file` $\longrightarrow$ Returns 1.

# read_command function

- The `read_command` function returns as first parameter:

**char ***argvv**

- Which is a structure that contains the commands introduced by the user
- Example:
  - Print command located in position i:

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

  - Print its first argument:

**printf("Arg 1 of command i: %s \n", argvv[i][1]);**

# read_command function

- The function `read_command` returns as second parameter:

**char **filev**

- Which is a structure that contains the files used in the redirections or, if it does not exist, string with a zero ("0").
  - **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 (! >)

# read_command function

- The function `read_command` returns as third parameter:

**int *in_background**

- Which is a variable that indicates if the commands are executed in background
- Its values are:
  - **in_background = 0** $\longrightarrow$ If it is not executed in background
  - **in_background = 1** $\longrightarrow$ If it is executed in background (&)

# read_command function

- ls -l | sort < file

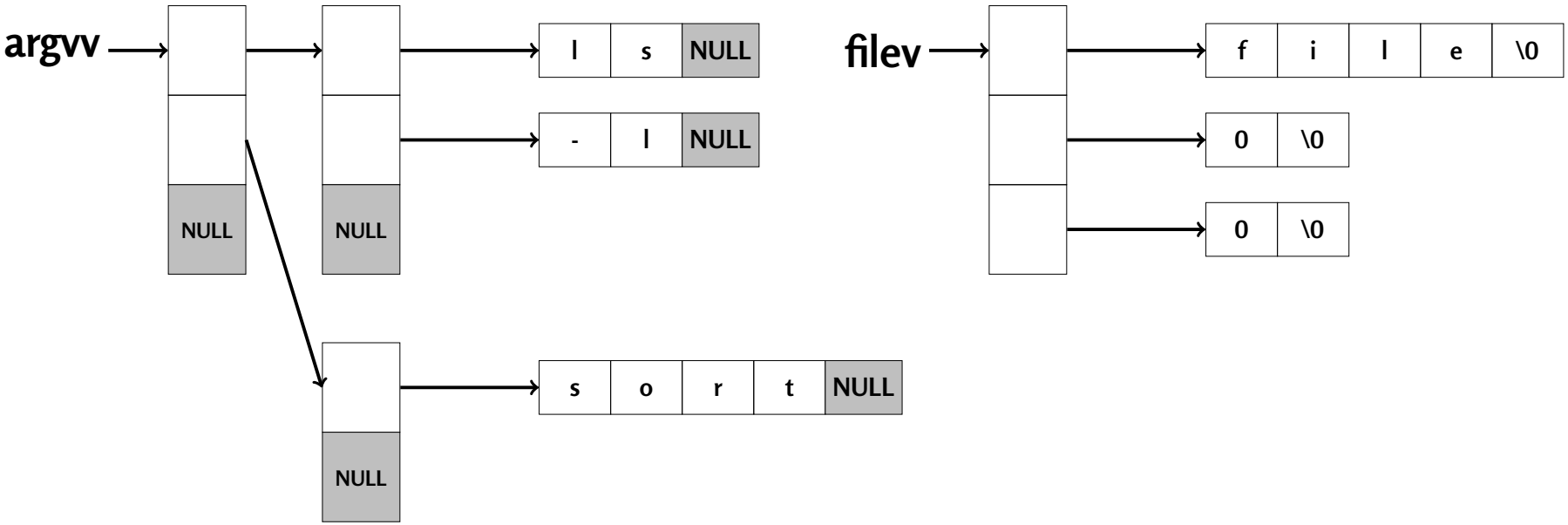



**Figure:** *Data structure used by the parser.*

# File descriptors of a process

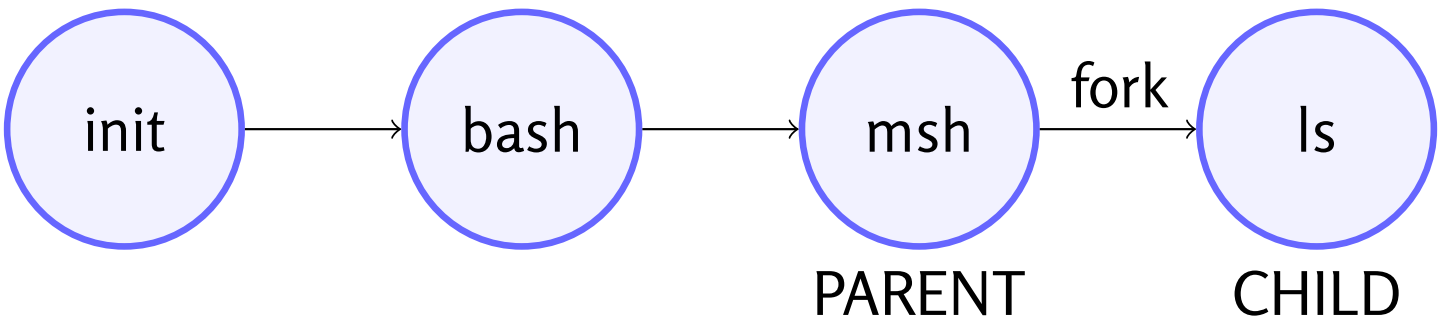
- In UNIX / Linux all processes have three opened file descriptors by default:
  1. Standard input (`STDIN_FILENO`): Value = 0
  2. Standard output (`STDOUT_FILENO`): Value = 1
  3. Standard error (`STDERR_FILENO`): Value = 2
- The commands executed in a shell are implemented to read and write from the standard input /output.
- It is possible to redirect the standard input/output to read/write using other files, or to read/write in a pipe.
- They are in the file descriptor table of a process when it is created:

| | |
|---|---|
| **0** | **STD_IN** |
| **1** | **STD_OUT** |
| **2** | **STD_ERR** |

# Needed processes in the minishell

- The minishell is the parent process of all the new processes generated from it.
- Example of execution of the command `ls`.

```
1 bash> ./msh      #Minishell (msh) execution
2 msh> ls          #ls command executed through minishell
```




- Every new process created from the minishell (for example: `ls`) will be executed as its child process

# Creation of processes with fork()

- It allows to generate a new process or child process that is an exact copy of the parent process: `pid_t fork()`. It returns:
  - **0** If it is the children.
  - **pid** If it is the parent.
- The child process inherits:
  - 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 a new ID (each process ID is unique).
  - **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 established locks of the parent.

# Example of creation of processes with fork()

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

int main(){
    int pid, status;
    pid = fork();
    switch(pid){
        case -1: /* error */
            perror ("Error in fork");
            return -1;
        case 0: /* child */
            printf("The process CHILD sleeps 10 seconds\n");
            sleep(10);
            printf("End of process CHILD\n");
            break;
        default: /* parent */
            if (wait(&status) == -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 command that 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 available 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 it is because an error has happened.
  - It returns **-1** and the error code is in the global variable `errno`.
  - The function never returns in a successful execution

# Example of process execution with execvp()

```
1  #include <sys/type.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main(){
6      int pid;
7      char *arguments[3] = {"ls", "-l", "NULL"};
8      pid = fork();
9      switch(pid){
10         case -1: /* error */
11             perror ("Error in fork");
12             return -1;
13         case 0: /* child */
14             execvp(arguments[0], arguments);
15             perror ("Error in exec. If the execution is correct this would never be
                   executed");
16             break;
17         default: /* parent */
18             printf ("I am the parent process\n");
19     }
20     exit(0);
21 }
```

# 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 process goes to ZOMBIE state until the parent dies.

`ps -axf` $\longrightarrow$ Visualize all processes, including zombie processes.

`kill -9 <pid>` $\longrightarrow$ Kill (terminate) a process.

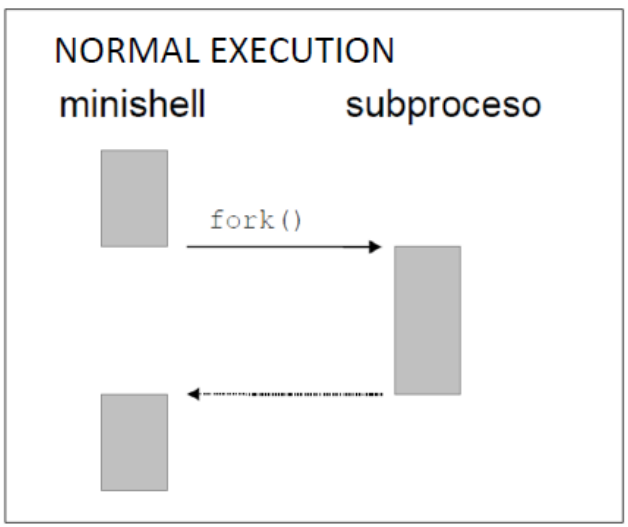
# Example of finalization and process waiting

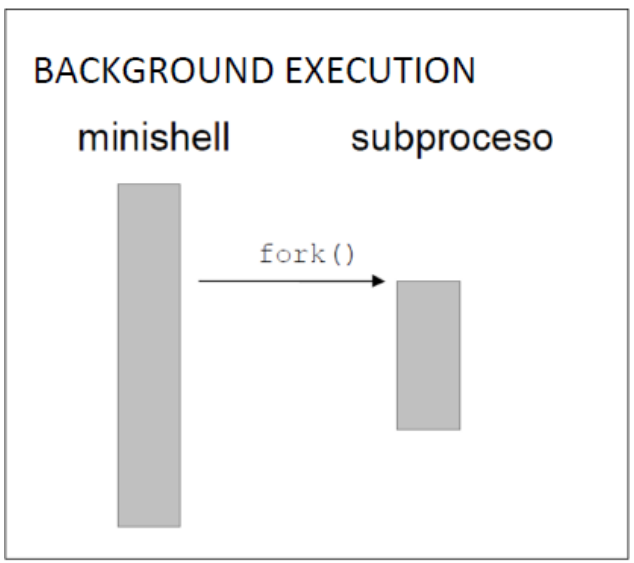
```c
#include <sys/type.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main(){
    int pid, status;
    char *arguments[3] = {"ls", "-l", "NULL"};
    pid = fork();
    switch(pid){
        case -1: /* error */
            perror ("Error in fork");
            return -1;
        case 0: /* child */
            execvp(arguments[0], arguments);
            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 ("Normal execution of the child\n");
    }
    exit(0);
}
```

# Background execution

1. A command can be executed in background from the command line using & at the end. For example: `sleep 10 &`
2. In this case the parent process does not block, waiting for the finalization of the child process.
3. The command `fg <job_id>` let you get a process from background to foreground. It receives a job id, not a pid.



# Process identifiers

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

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

- An example:

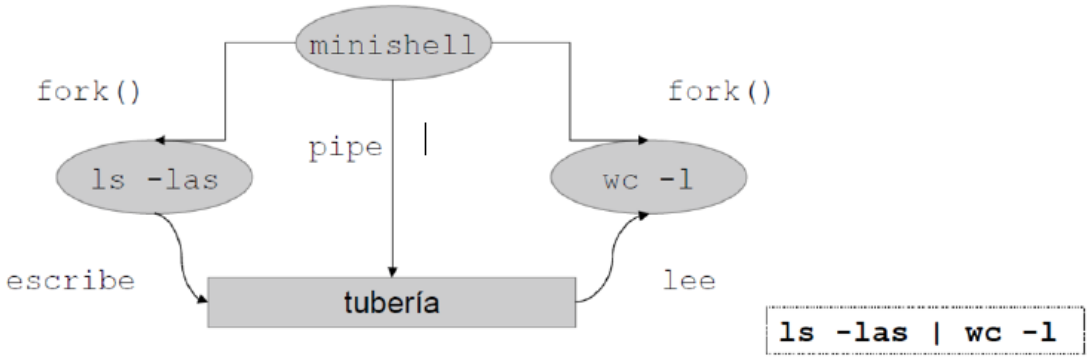
```
1 #include <sys/type.h>
2 #include <stdio.h>
3
4 int main(){
5     printf("Process identifier: %s\n", getpid());
6     printf(Process identifier of the parent process: %s\n", getppid());
7     return 0;
8 }
```

# Command sequences with pipes

- Command sequences are built using pipes: |. For example:

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

- The standard output 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.
- The last command writes in the standard output (screen) if there is no output redirection.

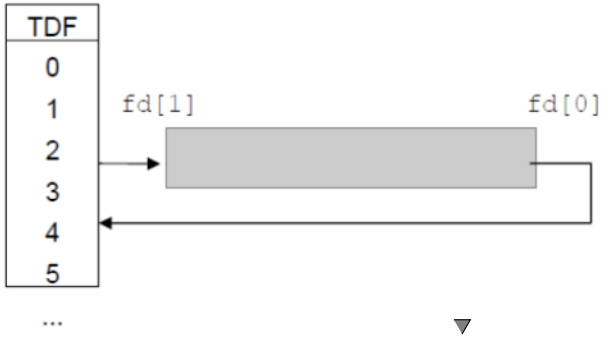


# Creation of pipes with pipe()

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

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

- Returns:
  - -1 $\longrightarrow$ If there is an error.
  - 0 $\longrightarrow$ In any other case.
- It receives an array with the file descriptor for input and output.
  - `descf[0]` $\longrightarrow$ Process input Descriptor (read).
  - `descf[1]` $\longrightarrow$ Process output descriptor (write).

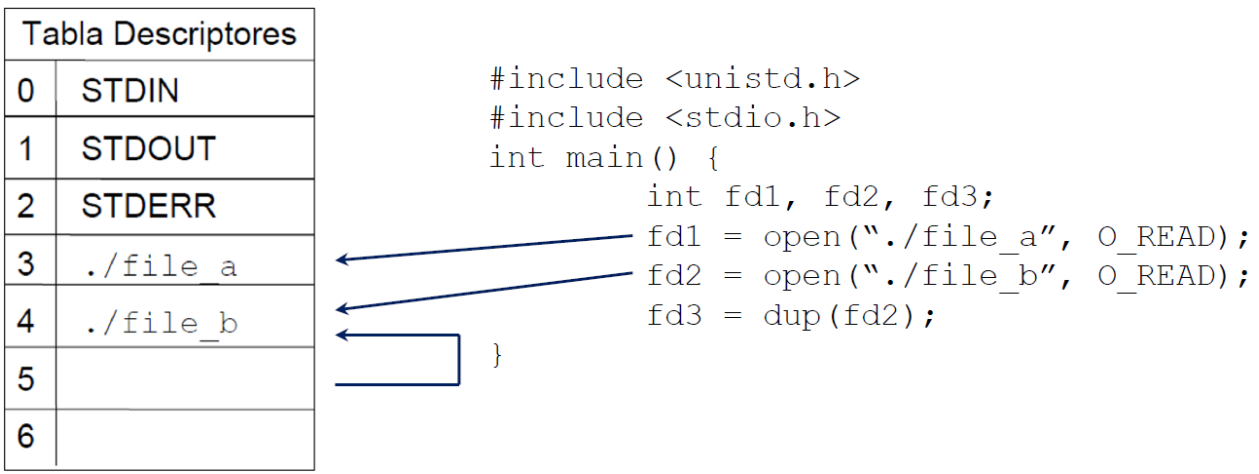


# Functions dup y dup2

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

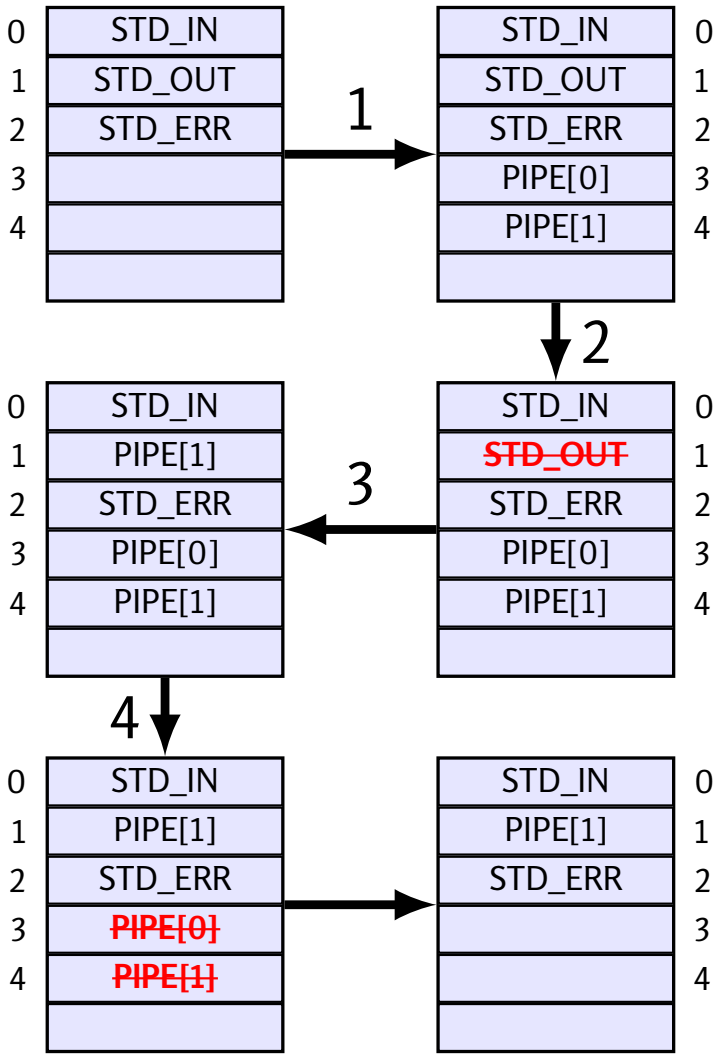
```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

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



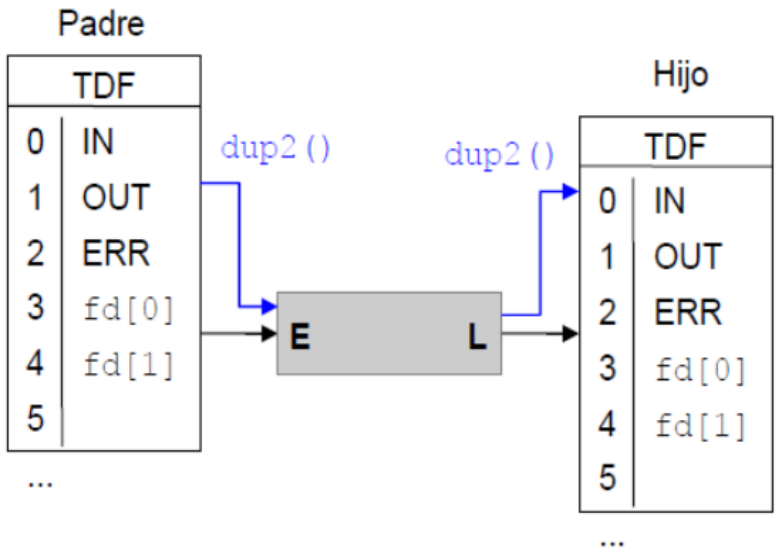
# Use of pipe + dup

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

```
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 }
```



# Input, output and error redirections

- System call `open` uses the first file descriptor available in the files table.
- It is possible to redirect the input/ouput to write/read from other files.
- The input redirection (<) only affects the first command.
- Example: Opens a file in read mode and uses it as standard input.

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

# Input, output and error redirections

- The output redirection (>) only redirects the last command.
- Example: 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 redirection of the standard error (!>) affects all commands.
- Example: Opens a file in write mode and use it as standard error.

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

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

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

# Contents

# 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.
- **They are not part of the command sequences.**
- **They do not have file redirections.**
- **They are not executed in background.**

# mycalc

- The minishell must offer the following internal command `mycalc` whose syntax is:

  `mycalc <operand_1> <add/mul/div> <operand_2>`

- For this purpose:
  1. Check that the syntax is correct.
  2. Execute the selected operations: sum (add), multiplication (mul) or division (div).
  3. Show the results on the screen.
     - Correct case by **standard error output**.
     - Error by **standard output**.

- The "add" operation has an "Acc" environment variable that accumulates the results of the sums performed, but not of the multiplications and divisions.

# mycalc

```
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; Remainder 3
msh>> mycalc 10 / 7
[ERROR] The structure of the command is mycalc <operand_1> <
    add/mul/div> <operand_2>
msh>> mycalc 8 mas
[ERROR] The structure of the command is mycalc <operand_1> <
    add/mul/div> <operand_2>
```

# myhistory

- The minishell must offer the following internal command `myhistory` whose syntax is:

```
myhistory
myhistory <N>
```

# myhistory

- For this purpose:
  1. Check that the syntax is correct.
  2. If it is executed without arguments, it will show by the **standard error output** a list with the last 20 commands introduced with the following format:

     `<N> <command>`

  3. If it is executed with an argument, the associated command in the list will be executed and the following message will be shown in the **standard error output**:

     `Running command <N>`

  4. If the command number does not exist, the following message will be shown on the **standard output**:

     `ERROR: Command not found`

# myhistory

- Provided:
  1. The variable `run_history` indicates whether the next command to be executed comes from the internal command `myhistory`
  2. The structure `command`
  3. The `store_command` function: copy the introduced command
  4. The `free_command` function: frees the resources used by the structure

# myhistory

```
msh>> myhistory
0 ls
1 ls | grep a
2 ls | grep b &
3 ls | grep c > out.txt &
msh>> myhistory 0
Running command 0
file.txt file2.txt
msh>> myhistory 27
ERROR: Command not found
```

# Contents

# Tester

- Students are provided with the shell-script **checker_os_p2.sh**
- The tester must be executed in the Linux computers of the Virtual Aulas of the university.
- To run it, you must give execution permissions to the file using:

```
chmod +x checker_os_p2.sh
```

- And run with:

```
./checker_os_p2.sh <zip_file>
```

- Example:

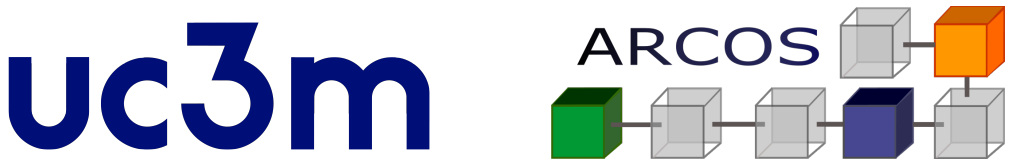
```
$ ./checker_os_p2.sh ssoo_p2_100254896_100047014.zip
```

# Contents

# Assignment Submission & Rules

- Groups: 3 students maximum
- Delivery:
  - Source code in a compressed file
  - Lab report in PDF through TURNITIN
  - Only one member of the group may deliver
- Delivery date:

**April 12$^{th}$ 2024 (until 23:55h)**




Computer Architecture and Technology Area

Universidad Carlos III de Madrid

# OPERATING SYSTEMS
# Lab 2. Programming a shell (Minishell)

Bachelor's Degree in Computer Science & Engineering
Bachelor's Degree in Applied Mathematics & Computing
Dual Bachelor in Computer Science & Engineering & Business Administration

Year 2023-2024