# Lab 03 - Building a shell 🐚

**Instructor:** Lorenzo De Carli, University of Calgary ([lorenzo.decarli@ucalgary.ca](mailto:lorenzo.decarli@ucalgary.ca))
***Slides by Lorenzo De Carli & Jay Carriere***

## In this lab

- Building a basic shell
- Adding command-line parameters
- Add path
- Adding pipes
- Adding background execution

## We have seen in class how a shell looks like

### ...at least in its simplest form!

```c
    printf("Enter commands, enter 'quit' to exit\n");
    do {
        printf("$ ");
        scanf("%s", buffer);
        scanf("%c", &newline);
        if ( strcmp(buffer, "quit") == 0 ) {
            printf("Bye!!\n");
            return 0;
        }
        else {
            pid_t forkV = fork();
            if ( forkV == 0 ) {
                args[0] = buffer;
                execve(buffer, args, NULL);
            } else
                wait(NULL);
        }
    } while ( 1 );
```

## Let’s make it more interesting

**In this lab you will add features to make things more realistic**

## Getting code for the Lab

- The code for Lab 03 has been uploaded to D2L in the lab03-template.zip file
- Please unzip the code and rename the folder to *Lab03-GroupXX*
- For this lab we have provided you with:
  - A base *shell.c* file that contains a starting point for you to begin implementing your own shell, notice user input is read by **fgets(…)** rather than **scanf(…)** to prevent buffer overflow
  - The beginnings of your shell parser in *parser.c/h*
  - And an example *test.c* file, which you can use to develop tests for your code as you're developing (hint: you can run **$ make test** to see how this file works)
  - You will likely want to create additional source/header files as you're building out your lab solution to implement the various exercises

## Modifying code for Exercises/Features

- The goal of the labs is to give you the opportunity to think about and explore the concepts covered in the lecture materials in a hands-on manner
- As you all know, Software Engineering is far more than just coding, it’s important that you get the opportunity to critically analyze, design, and work on more freeform projects as you’re developing your skills
- So rather than breaking the lab down and having you just modify a few lines of code here and there, we want to give you the opportunity to work with your group members to break down each of the exercises and modify/structure your code however you think is best

## Modifying code for Exercises/Features

- You can add as many code files/headers as you want (within reason 😉), so feel free to structure your solution in a way that makes sense to your group
- You have the freedom (and will likely want) to change the definitions and purpose of the functions provided in the provided files (e.g. *parser.c/h*), e.g. the **trimstring(…)** function could return tokens rather than a string etc.
- You'll also notice that none of the exercises mention *test.c*, this is to give you and your group members the opportunity to think about what you want to test (or not) as you're coding, you can design your own tests for your solution as you want
- Note: Your solution does not need to include any tests if you don't want, the key thing that is that you meet the requirements in each of the exercises

## Feature #1

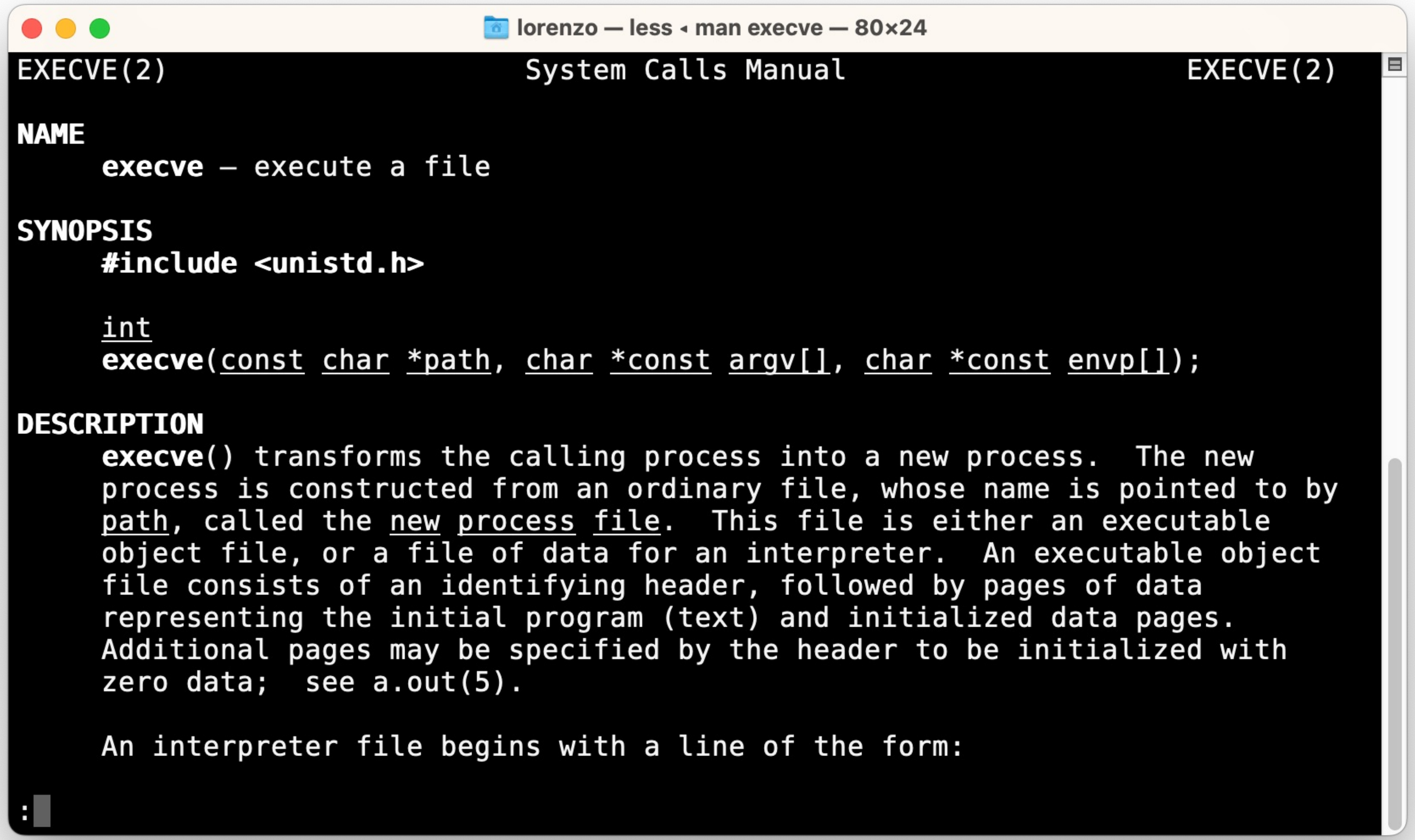
### Basic shell

- This shell should **present a command line** and receive **single-word commands** (a command without any parameters) and execute them by passing them to **execve()**
- The single-word commands should include the full path to the executable of the command to be run (e.g. **$ /usr/bin/ls**)
- “**quit**” should be interpreted as a special command that **terminates the shell**
- This is basically what we seen in class so you should **deliver at least this by end of the lab**

# Feature #2

## Command line parameters

- If the command consist of more than one word, anything past the first term should be passed as command line parameter to the command
- Refer to the **execve()** man page on how to do this, and the example call to **echo** provided in *shell.c*



```
EXECVE(2)                    System Calls Manual                   EXECVE(2)

NAME
     execve – execute a file

SYNOPSIS
     #include <unistd.h>

     int
     execve(const char *path, char *const argv[], char *const envp[]);

DESCRIPTION
     execve() transforms the calling process into a new process.  The new
     process is constructed from an ordinary file, whose name is pointed to by
     path, called the new process file.  This file is either an executable
     object file, or a file of data for an interpreter.  An executable object
     file consists of an identifying header, followed by pages of data
     representing the initial program (text) and initialized data pages.
     Additional pages may be specified by the header to be initialized with
     zero data;  see a.out(5).

     An interpreter file begins with a line of the form:

:
```

## Feature #2

### Command line parameters

- If the command consist of more than one word, anything past the first term should be passed as command line parameter to the command
- Remember that parameters often have a ‘**-**’ prefix or can include the location of a directory **/home/ensf461** so your shell logic needs to work in these cases
- Your shell must also respect quotation marks with the command input, for example **$ cmd –test “/home/ensf461/some folder”** should be broken down into:
    - One command: **cmd**
    - Two parameters: **–test** and **/home/ensf461/some folder**

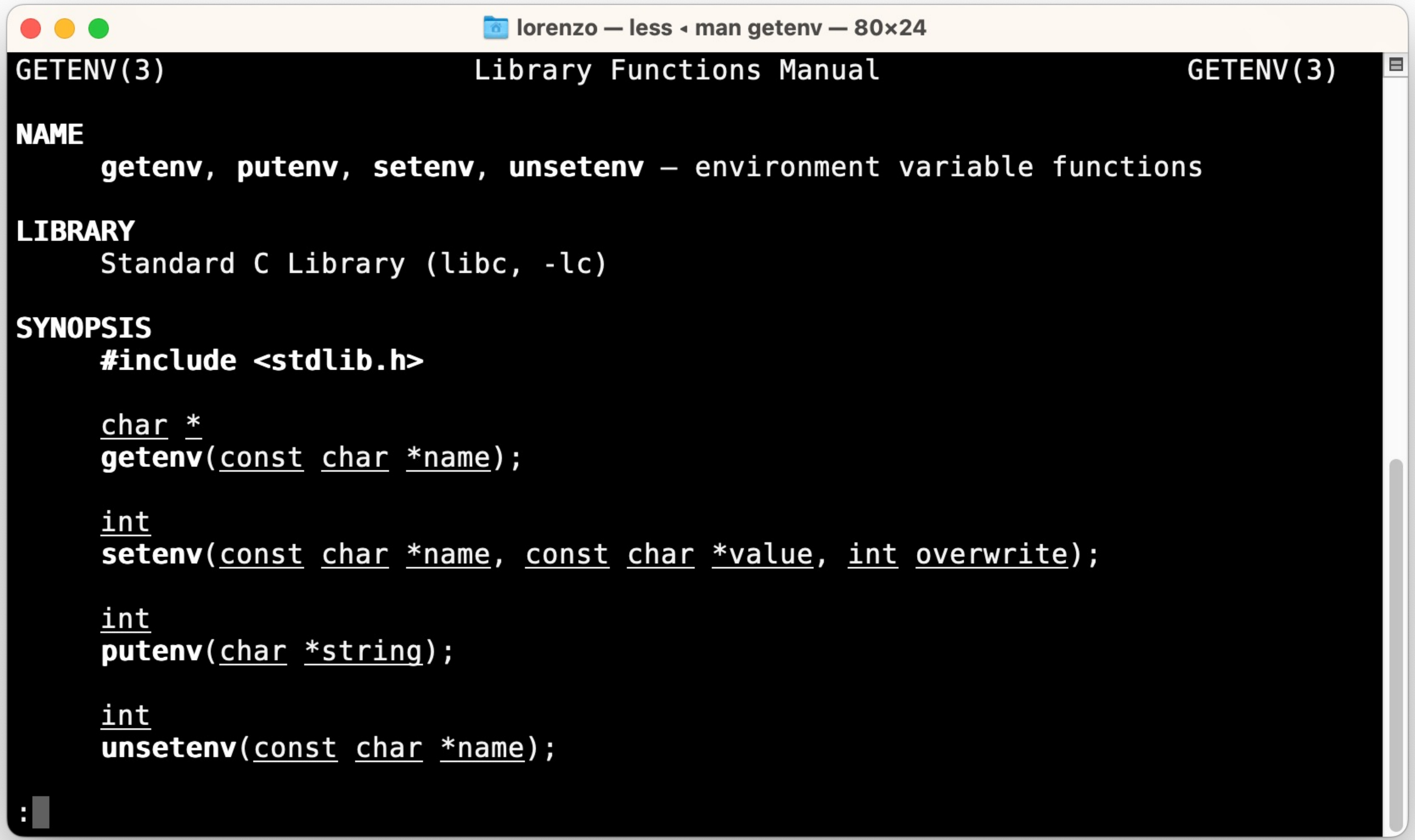
# Feature #3

## Honour the PATH environment variable

- Until now, each command must have been expressed as the **full path** to an executable
- Now, you will need to add support for the **PATH** environment variable
- If the command start with a “**/**“, consider it as an **absolute path** and execute it as such
- If the command does not start with “**/**“ but contains a “**/**“ (e.g., **“app/cmd”**), consider it as a **relative path** and execute it in the **current working directory**
- Otherwise, search for the command in the directories listed in the **PATH** variable; and executed it **from there**

## What is this PATH sorcery?

- If you have no idea what I am talking about, look at the slides for the **lectures about the shell**
- **getenv()** is your friend



```
GETENV(3)                   Library Functions Manual                   GETENV(3)

NAME
     getenv, putenv, setenv, unsetenv – environment variable functions

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdlib.h>

     char *
     getenv(const char *name);

     int
     setenv(const char *name, const char *value, int overwrite);

     int
     putenv(char *string);

     int
     unsetenv(const char *name);

:
```

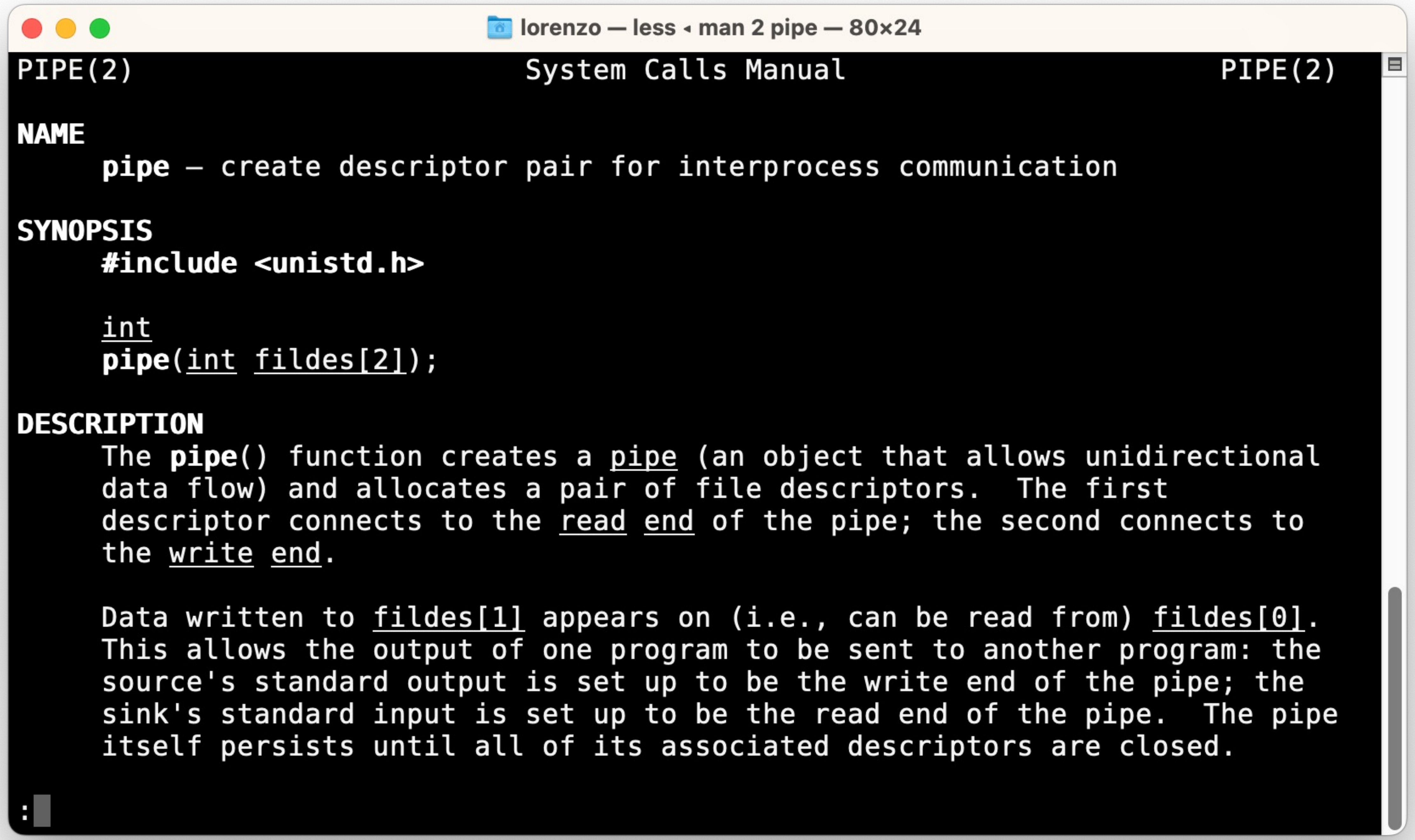
# More on PATH

- **getenv()** will allow you to get the current value of the **PATH** variable (hint: you can use `char* path = getenv("PATH");` for this)
- This is call to **getenv()** is going to return a list of paths separated by colon (e.g., "/**usr/sbin:/usr/bin:…**")
- You shall **tokenize the string**, and check whether the user-provided command is in that path. You can use the **strtok()** command as a starting point to tokenize the input, but think about what else your tokenizer needs to do
- For example, if the user enters "**ls**":
  - Check if "**/usr/sbin/ls**" exists
  - If not, check if "**/usr/bin/ls**" exists and can be executed
- How to check that? **man access** is your friend

## Feature #4

### Adding pipes

- Your shell should now support the “**|**” (**pipe**) command
- If multiple commands are listed and separated by “**|**” , the **standard output** of one command should be redirected to standard input of the next one
- How to do this? “**pipe()**” is your friend



```
PIPE(2)                       System Calls Manual                      PIPE(2)

NAME
     pipe – create descriptor pair for interprocess communication

SYNOPSIS
     #include <unistd.h>

     int
     pipe(int fildes[2]);

DESCRIPTION
     The pipe() function creates a pipe (an object that allows unidirectional
     data flow) and allocates a pair of file descriptors.  The first
     descriptor connects to the read end of the pipe; the second connects to
     the write end.

     Data written to fildes[1] appears on (i.e., can be read from) fildes[0].
     This allows the output of one program to be sent to another program: the
     source's standard output is set up to be the write end of the pipe; the
     sink's standard input is set up to be the read end of the pipe.  The pipe
     itself persists until all of its associated descriptors are closed.

:
```

# More on pipes

- First, you only need to support two-command pipe
  - Example: **ls -1 | grep .h** (no need to pipe more than two!)
- The process is as follows:
  - Use **pipe()** to create a pair of connected file descriptors
  - **pipe()** returns a **pair of file descriptors**. The **first** is the **write end** of the pipe, the **second** the **read end**
  - You must turn the **first descriptor** into the **standard output of the first command**, and the **second descriptor** into the **standard input of the second command**

# Wait, what?

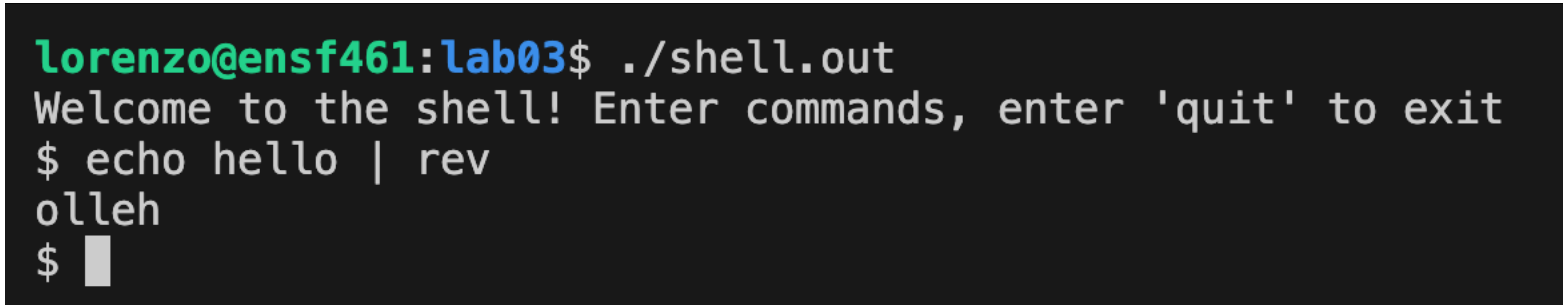
- When you call **pipe()**, the kernel internally creates a **queue**
  - It also makes a note of the **mapping** between the two file descriptors returned by **pipe()** and that queue
- When a process **writes to one file descriptor**, the OS is smart enough to intercept that data and **push into the queue**
- When a process **reads from the other file descriptor**, the OS **pulls data from the queue**
- This business of **using file descriptors to read/write from things that are not files** is pretty common in UNIX systems, so you should get used to it

# Even more about pipes

pipe_fd was initialized by pipe() in the parent process

- You must call **pipe()** prior to calling **fork()**
- Then, in the first child process:
  - Use **dup2()** to turn the **write end of the pipe** into the **standard output**
  - How? **dup2(pipe_fd[1], STDOUT_FILENO);**
- In the second child process:
  - Use **dup2()** to turn the **read end** into the **standard input**
  - How? **dup2(pipe_fd[0], STDIN_FILENO);**

## Pipe example



```
lorenzo@ensf461:lab03$ ./shell.out
Welcome to the shell! Enter commands, enter 'quit' to exit
$ echo hello | rev
olleh
$ 
```

## Feature #5

### Adding background execution

- If a command ends with “**&**”, the shell should execute it and return to the prompt **without waiting for the command to complete**
- Before executing a command, the shell should **check whether any background command has completed** and print a  message: “Background command <X> terminated”, where <X> is the PID  of the command which just completed
- We are going to keep thing simple. A command may contain a pipe, or a background command, or neither of them, **but not both**

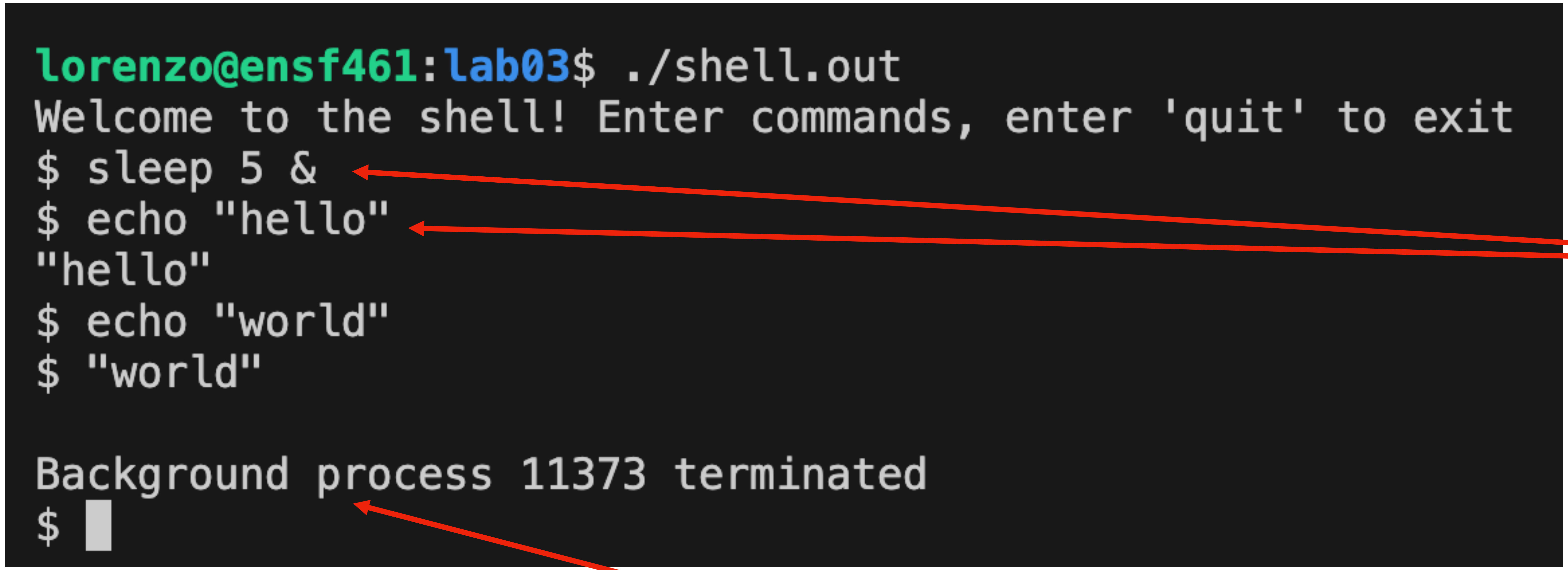
# How do I executed things in background?

- Easy, you just don’t call **wait()** in the parent process
- That way, the parent **does not hang** waiting for the child process to end
- But you still are asked to **print a message** when a child **terminates**
- **How to do that?**
  - You need to call **waitpid()** prior to displaying the prompt to the user
  - Do that at every iteration of the main loop

# Checking the status of a child process

- You can check whether a child process terminated using **waitpid()**
- **wait()** always pause until a child terminates (unless there are no children)
- **waitpid()** can be called in such a way that will report whether a child terminated, but if not it will just continue execution
- Example: **waitpid(-1, &status, WNOHANG))**
  - **-1** means “check if ANY children terminated” (not a specific one)
  - **&status** is an *int with the reason why it terminated
  - **WNOHANG** means “do not pause waiting for children to terminate”
  - The return value is either a **PID** (if anything terminated), or **0**

## Background execution example



```
lorenzo@ensf461:lab03$ ./shell.out
Welcome to the shell! Enter commands, enter 'quit' to exit
$ sleep 5 &
$ echo "hello"
"hello"
$ echo "world"
$ "world"

Background process 11373 terminated
$ 
```

Those two commands can be run back to back
(even if “sleep 5” pauses execution for 5 seconds)

Note that if I wait long enough, eventually “sleep 5” will terminate and the shell will notify me

## Grading rubric

- **3 pts** for uploading initial progress in D2L by end of the lab
- **1 pt** for **command line parameters** support
- **2 pts** for supporting the **PATH** environment variable
- **2 pts** for **pipe** support
- **2 pts** for **background execution** support
- **Please provide a Makefile to build your code.** It must be executed with "make" and generate an executable called "shell.out"

## Submission instructions

- You need to submit **a single zip file to the D2L “Lab 3” dropbox twice**
  - One “in-lab” submission is due t**oday before 4 PM**, including all of the work you were able to complete in the lab, in a single zip called **Lab03-GroupXX.zip**, this submission is worth **3 Pts**
  - You will have until **11:59 PM on the day before the next lab** to submit your completed lab solution, in a single zip file **with the same Lab03-GroupXX.zip name**, this submission will be graded for the remaining **7 Pts**.
- You may submit your solution to your dropbox **as many times as you’d like**, but **only the LAST submitted zip file will be graded** for the three exercises (please double-check that the zip file opens properly!)

## Some ground rules

- You **cannot** use **system()** from **stdlib.h** at any point in the program
- You can assume **malloc()** and other system calls do not fail