UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Advanced Programming for Astrophysics

Lecturers:

Prof. Murante Giuseppe
Prof. Taffoni Giuliano

Author:
Andrea Spinelli

September 27, 2025

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

This document was prepared for the Advanced Programming for Astrophysics course, held by Prof. Murante Giuseppe and Prof. Taffoni Giuliano in the academic year 2025/2026.

Contents

1 Introduction

Operating systems (OS) are the core software that manage a computer's hardware and software resources. Among the most widely used are Microsoft Windows, Apple's macOS, and the open-source Linux. While they differ in many aspects, both macOS and Linux are part of the ***Unix-like*** family of operating systems. These systems are renowned for their stability, security, and a powerful command-line interface known as the ***shell***.

This chapter provides an introduction to the Unix shell, exploring its fundamental role in interacting with the system.

File System

A crucial component of the operating system is the file system, which organizes files and directories on a computer. It is a crucial component of the operating system that allows users to store, retrieve, and manage data efficiently. In unix-like systems, the file system is organized in a hierarchical structure.

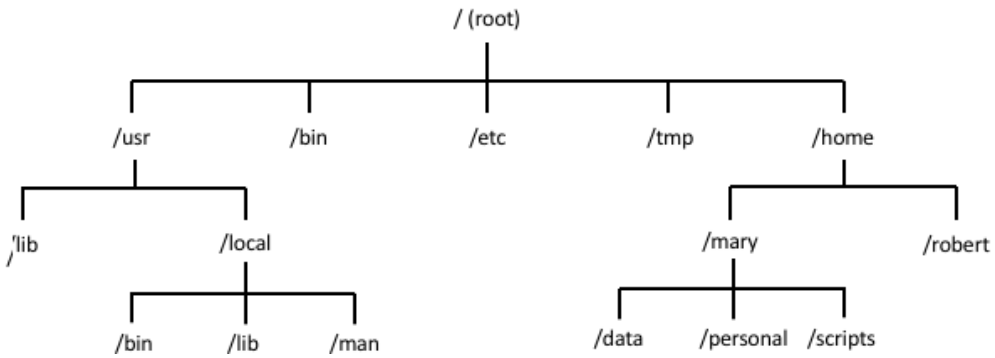

Figure 1.1: The File System

Path

A path specifies the unique location of a file or directory within the file system's hierarchy. Paths can be ***absolute***, starting from the root directory (`/`), or ***relative*** to the current working directory.

`/`	Root Directory	`~`	Home Directory
`.`	Current Directory	`..`	Parent Directory

Example: *Example of a path*

- a file in the user's Desktop: `/home/user/Desktop/file.txt` or `~/Desktop/file.txt`
- a file in the current directory: `./file.txt`
- a file in the parent directory: `../file.txt`
- a file in the home directory: `~/file.txt`

1.1 The Unix Shell

What is a shell?

The shell is the primary interface between users and the computer's core system. When you type commands in a terminal, the shell interprets these instructions and communicates with the operating system to execute them, making complex system operations accessible through simple commands.

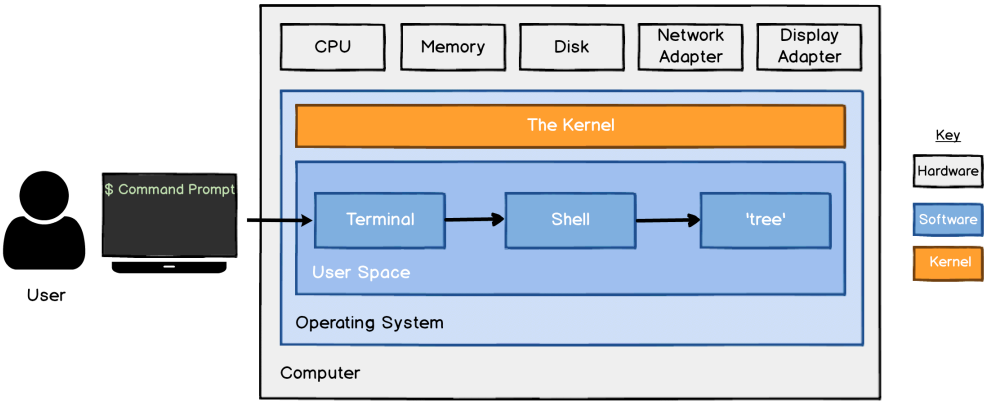

Figure 1.2: The Unix Shell

Definition: *Shell*

A shell is a program that provides the traditional, text-only user interface for Linux and other UNIX-like operating systems. Its primary function is to read commands that are typed into a console […] and then execute (i.e., run) them. The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an interface between the user and the internal parts of the OS (at the very core of which is the kernel). *~Linfo*

1.1.1 Basic Commands

A command is nothing more than a program that is executed by the shell. The usual syntax is:

```
command [options] [arguments]
```

The options can be passed setting the corresponding flag e.g. `-h` for help.

Navigation Commands

It is possible to navigate and manage files or directories using the following commands:

Command	Description
`pwd`	Print Working Directory
`cd`	Change Directory
`ls`	List Directory
`mkdir`	Make Directory
`rm`	Remove File or Directory
`cp`	Copy File or Directory
`mv`	Move File or Directory
`touch`	Create File

Recursive Commands

`rm`, `cp` and many other commands can be used recursively into a folder using the flag `-r` (recursive).

Help Flag

Many commands support the `--help` flag to display information about the command and its usage.

Tip: *Hidden Files*

Hidden files are files that are not shown by default when listing the contents of a directory. They are usually marked with a dot (.) at the beginning of their name. e.g. `.config`

Searching Tools

Unix-like systems provide powerful tools for searching files and content. These tools often support regular expressions (regex), which are patterns used to match character combinations in text.

File Search Commands

Command	Description	Example
`find`	Search for files and directories	`find /home -name "*.txt"`
`locate`	Fast file search using database	`locate filename`
`grep`	Search text within files	`grep "pattern" file.txt`
`which`	Find location of executable	`which gcc`
`whereis`	Find binary, source, and manual pages	`whereis python`

Wildcards and Pattern Matching

Wildcards are special characters that represent one or more characters in a filename or text string:

Character	Name	Description
`*`	asterisk	Matches zero or more characters
`?`	question mark	Matches exactly one character
`[]`	brackets	Matches any single character within brackets
`[!]`	negated brackets	Matches any character not in brackets

Example: *Wildcard Examples*

- `*.txt` : matches all files ending with .txt
- `file?.c` : matches file1.c, file2.c, but not file10.c
- `[abc]*` : matches files starting with a, b, or c
- `[!0-9]*` : matches files not starting with digits

Operators

Commands can be combined using operators to manipulate input and output streams:

Operator	Name	Description
`>`	output redirection	Redirects standard output to a file, overwriting its contents
`>>`	append redirection	Redirects standard output, appending it to the end of a file
`|`	pipe	Passes the output of a command as input to another
`&&`	logical AND	Executes the next command only if the previous one succeeds
`||`	logical OR	Executes the next command only if the previous one fails
`;`	separator	Executes commands sequentially, regardless of their outcome

1.1.2 Environment Variables

Environment variables are dynamic values that affect the behavior of processes and programs running on the system. They provide a way to pass configuration information to applications without hardcoding values into the program. These variables are stored in the system's environment and can be accessed by any process.

Variable	Description	Example
`PATH`	Colon-separated list of dirs to search for executables	`/usr/bin:/bin:/usr/sbin`
`HOME`	Path to the user's home directory	`/home/username`
`USER`	Current username	`username`
`SHELL`	Path to the current shell	`/bin/bash`
`PWD`	Current working directory	`/home/username/Desktop`

To view environment variables, use the `env` command or `echo $VARIABLE_NAME`. To set a variable temporarily, use `export VARIABLE_NAME=value`. For permanent changes, modify configuration files like `.bashrc` or `.bash_profile`.

1.1.3 File Permissions

Unix-like systems use a permission system to control access to files and directories. Each file has three types of permissions: read (r), write (w), and execute (x). These permissions are assigned to three categories of users: owner (user), group, and others.

Permission Structure

File permissions are displayed using a 10-character string where the first character indicates the file type, and the remaining nine characters represent permissions for owner, group, and others:

Character	Position	Meaning
`-`	1st	Regular file
`d`	1st	Directory
`r`	2nd, 5th, 8th	Read permission
`w`	3rd, 6th, 9th	Write permission
`x`	4th, 7th, 10th	Execute permission

For instance, `-rwxr-xr--` means:

- Regular file (-)
- Owner: read, write, execute (rwx)
- Group: read, execute (r-x)
- Others: read only (r–)

Octal Notation

Permissions can be represented numerically using octal notation:

(`r`) Read	**4**	`100`
(`w`) Write	**2**	`010`
(`x`) Execute	**1**	`001`

By summing these values for each user category (owner, group, and others), we get a three-digit code. For example, `rwx` permissions translate to $4+2+1=7$, and `r-x` to $4+0+1=5$. Thus, a full permission string like `rwxr-xr-x` can be concisely represented as 755.
Common combinations include: **755** (`rwxr-xr-x`), **644** (`rw-r--r--`), and **600** (`rw-------`).

Permission Commands

Command	Description	Example
`chmod`	Change file permissions	`chmod 755 script.sh`
`chown`	Change file ownership	`chown user:group file.txt`
`chgrp`	Change group ownership	`chgrp staff file.txt`
`umask`	Set default permissions	`umask 022`

TODO: notes about file viewing and text processing, few words about multiprocessing and context switching

1.2 Intro to C

C is a general-purpose programming language created in the 1970s by Dennis Ritchie. Its design gives programmers direct access to the underlying hardware, making it highly efficient and powerful. It is instrumental in implementing operating systems, device drivers, and protocol stacks.
The easiest code we can write in C is:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
}
```

- `#include <stdio.h>`: is a preprocessor directive.
- `int main()`: is the main function.
  - `int`: is the return type of the function, if omitted, the function returns an integer.
  - `main()`: is the name of the function.
- `printf(" ... ")`: is the function that prints the string `"Hello, World!"` to the console.
- `{}`: is the block of code that contains the statements of the function.

1.2.1 Compilation

To compile a C program, use the following command:

```
gcc main.c -o main.x
```

The compiler generates the object code, which is a binary file that contains the machine language translation of the program. The command above actually executes 3 steps:

1. **Preprocessing**:
   This step expands the macros and includes the header files. e.g. `#include <stdio.h>`

   ```
   gcc -E main.c -o main_preprocessed.c
   ```

2. **Compilation**:
   The compiler translates the preprocessed code into object code.

   ```
   gcc -c main_preprocessed.c -o main.o
   ```

   During this step, it is possible to set the `-I` flag to include other directories (useful if the headers are not in the default directory). e.g. `-I /path/to/include`
3. **Linking**:
   The linker combines the object code with the standard library to create an executable.

   ```
   gcc main.o -o main.x
   ```

   The `-o` flag is used to set the output file name.

   During this step, it is possible to set the `-L` flag to include other directories (useful if the libraries are not in the default directory). e.g. `-L /path/to/lib` and to link against external libraries using the `-l` flag. e.g. `-lmylib`

1.2.2 Libraries

A library is a collection of pre-compiled code, such as functions and data structures, that can be reused across multiple programs. They promote code modularity and prevent developers from having to reimplement common functionalities, such as input/output operations or mathematical calculations. There are two types of libraries:

- `lib___.a`: is a static library.
- `lib___.so`: is a shared library.

The difference between the two is that the static library is linked directly into the executable at compile time, while the shared library is loaded at runtime.

1.2.3 Data Types

C provides several fundamental data types to represent different kinds of data. Understanding these types is crucial for writing efficient and correct programs.

TODO: notes about data types

1.2.4 Loops

Loops are used to execute a block of code a specified number of times. There are two main types of loops: the `while` loop and the `for` loop.

While and Do While Loops

The `while` loop is used to execute a block of code as long as a specified condition is true.

```
1 while (condition) {
2     // code
3 }
```

If the condition is never met, the block of code is not executed. If we need to execute the block of code at least once, we can use the `do while` loop.

```
1 do {
2     // code
3 } while (condition);
```

For Loop

The `for` loop is used to execute a block of code a specified number of times.

```
1 for (int i = 0; i < 5; i++) {
2     // code
3 }
```

Break and Continue

The `break` statement is used to exit a loop prematurely. The `continue` statement is used to skip the rest of the code in a loop and move to the next iteration.

```c
while (1) { // infinite loop (should be avoided)
    // code
    if (condition_1) {
        break; // exit the loop
    }
    else if (condition_2) {
        continue; // skip the code below and move to the next iteration
    }
}
```

Increment and Decrement

The `++` and `--` operators are used to increment and decrement the value of a variable.
There are two types of increment and decrement operators:

- `++i` (`--i`): is the prefix increment (decrement) operator. The value of the variable is incremented (decremented) before it is used.
- `i++` (`i--`): is the postfix increment (decrement) operator. The value of the variable is incremented (decremented) after it is used.

1.2.5 Directives

Directives are used to control the compilation process. They are used to define constants, macros, and to conditionally compile code.

Macros

Macros are used to define values that are not expected to change during the execution of the program. They are defined using the `#define` directive.

```c
#define CYCLE 0
```

Conditional Compilation

Conditional compilation is used to compile code only if a certain condition is met. They are defined using the `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` directives.

```c
#ifdef CYCLE
    // code to compile if CYCLE is defined
#endif
```

TODO: notes about pointers

1.2.6 Functions

Functions are used to group code into ***routines***: reusable units of code.

```
return-type function-name(parameters declarations, if any) {
    declarations
    statements
    return return-value;
}
```

Parameters are passed to the function by value, meaning that a copy of the argument is passed to the function. If the argument is a *pointer* (we will see), the function will modify the original argument.

It is possible to return only a single value. We will see later that it is possible to return a *pointer* or a *struct*, which can be useful to return multiple values from a function.

If the return value is not the same specified in the function declaration, the compiler probably won't notice, so be careful.

Variable scopes

- **Local scope**:
  Variables declared inside a function are said to have ***local scope***. This means they only exist and are accessible from within that specific function. Once the function finishes its execution, these variables are destroyed. This allows different functions to use the same variable names without causing conflicts.
- **Global scope**:
  Variables declared outside any function are said to have ***global scope***. This means they are accessible from any function in the program.

Observation: *Variable shadowing*

If a local variable is declared with the same name as a global variable, the local variable takes precedence within its scope. This is known as *variable shadowing*. The global variable is temporarily hidden, and any reference to that variable name inside the function will refer to the local one. The global variable remains unaffected outside of this scope.

```c
#include <stdio.h>

int a = 10; // Global variable

void myFunction() {
    int a = 20; // Local variable shadows the global one
    printf("Inside function, a = %d\n", a); // Prints 20
}

int main() {
    printf("Before function call, a = %d\n", a); // Prints 10
    myFunction();
    printf("After function call, a = %d\n", a);  // Prints 10
    return 0;
}
```