Linux

Pau Fernández Rafa Genés Jose L. Muñoz Tapia
Universitat Politècnica de Catalunya (UPC)

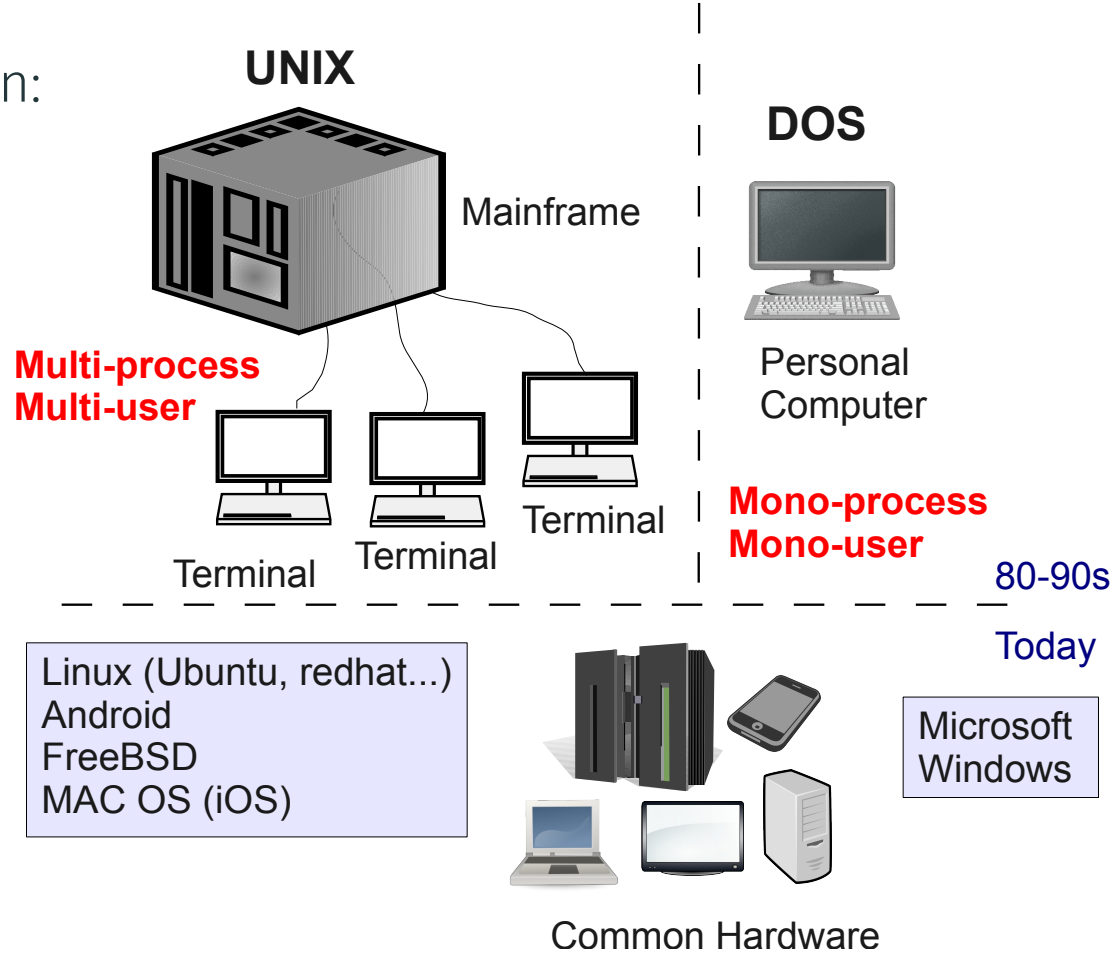
Outline

Outline

Introduction to OS

- An OS has to:
  - Manage the resources of a computer.
  - Provide an interface for human beings: CLI or GUI.

OS Evolution:



Basic Unix Features

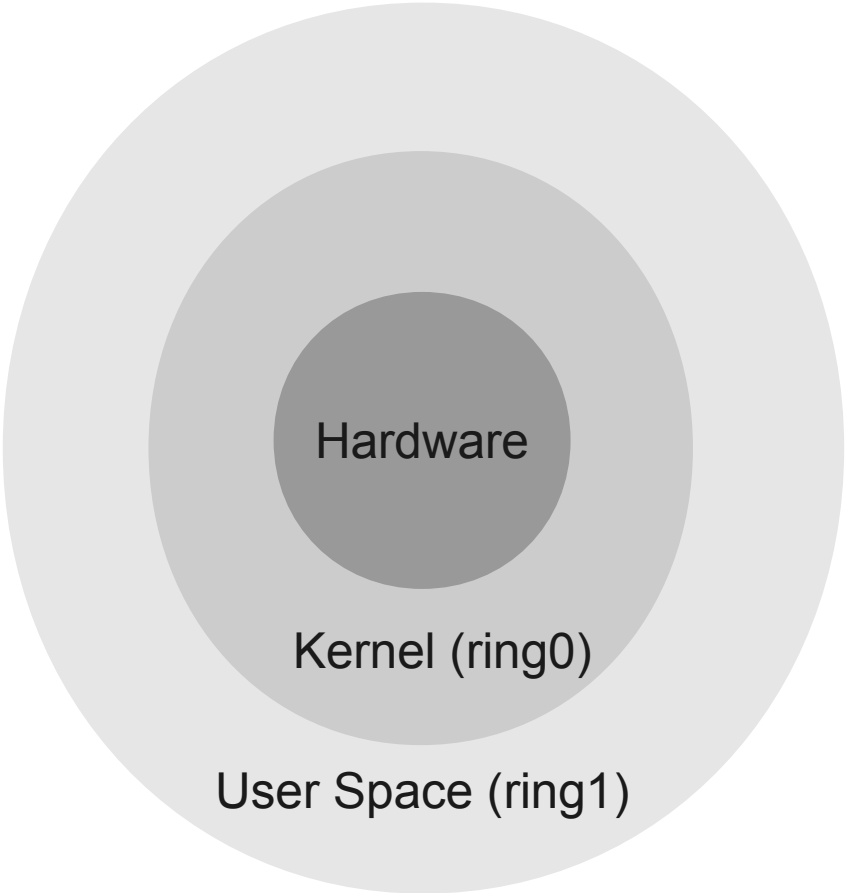
- **Multi-process.** Several processes can run on the same computing environment.
- **Multi-user.** Several users can share the computing environment.
- **Simple Interfaces.** Text as much as possible.

Outline

OS Rings

- **Kernel**. Kernel manages hardware and essential operations with the hardware.
- **User space**. Processes of users.

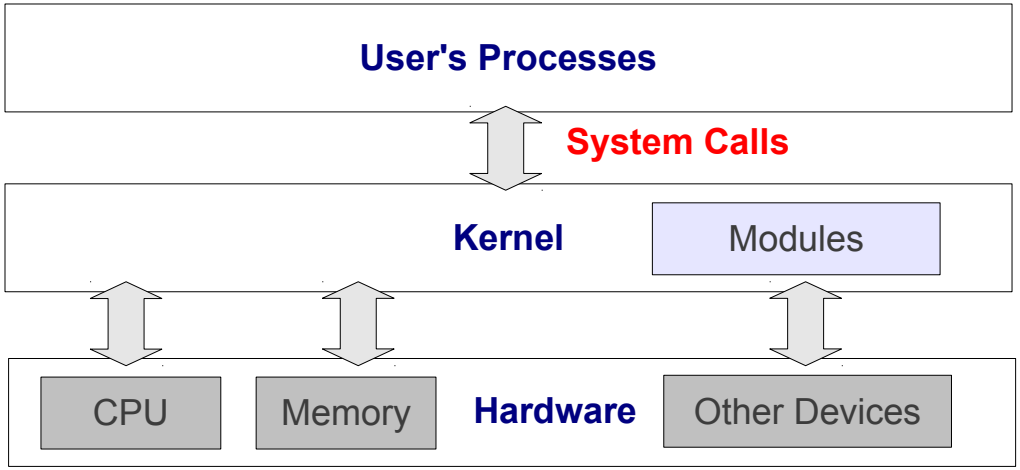
- Interfaces:
  - Hardware to Kernel: **privileged operations** (low level operations). E.g. manage CPU -> CPU scheduler.
  - kernel to user space applications: **system calls**. E.g. ask for creating a new process.



Hardware

- **Central Processing Unit (CPU)**. The CPU is responsible executing programs.
- **Memory.** Memory is used to store both program instructions and data.
- **Input/Output (I/O) Devices.** The kernel manages requests from user applications to perform input and output operations and provides convenient methods for using each device.

System Calls



- The kernel provices an interface to user applications managing low level operations with the hardware.
- The kernel runs in “supervisor mode”.
- Unix systems provide a library or API that sits between user programs and the Kernel.
- C library such as glibc provides wrapper functions for the system calls. For example:

```
int kill(pid_t pid, int sig);
```

Modules

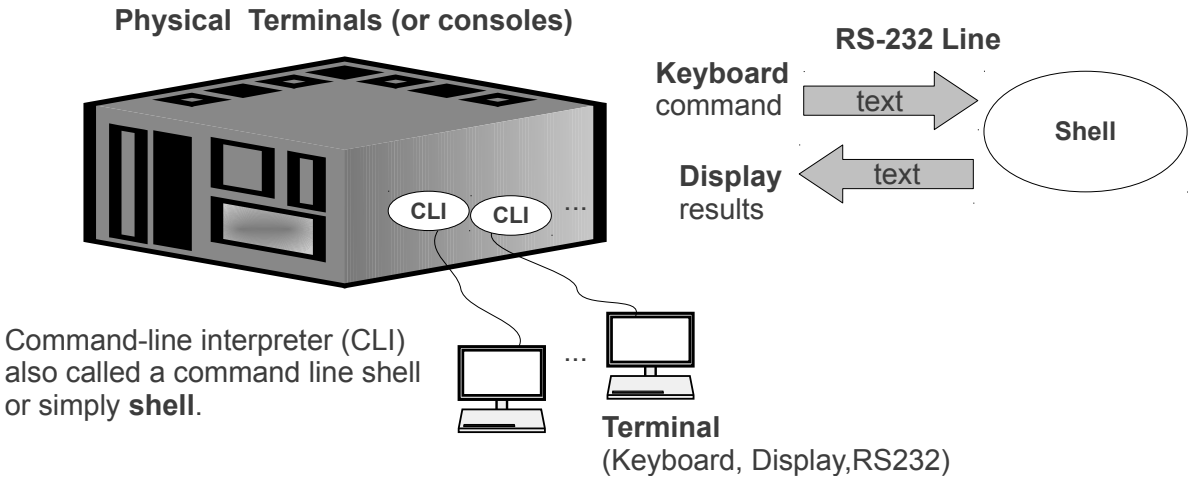
- The Linux Kernel is a **monolithic hybrid kernel**.
- Monolithic means the kernel is alone in supervisor mode.
- Hybrid means that kernel extensions, called modules, can be loaded and unloaded into the kernel upon demand while the kernel is running.
- Device drivers are one type of module.
- When you build a Linux kernel you can decide if you include a certain module inside the kernel (statically compiled module) or if you allow this module to be loaded at run time by your kernel (dynamic module).
- The commands to list, insert and remove modules are: `lsmod`, `modprobe` and `rmmod`.

Outline

Interacting with the OS

- There are two types of interfaces:
  - **Graphical User Interface (GUI).**
    - Require a graphical server (e.g. X server).
    - Processes are typically applications that have graphical I/O using mouse, keyboard, touch screens, etc.
  - **Command Line Interface (CLI).**
    - Does not require a graphical server.
    - Processes are commands that have only textual I/O.
    - The I/O of the commands is related or “attached” to a terminal.
    - The terminal is also attached to a command line interpreter or shell.
    - Types of terminals: Physical terminal, Virtual Console and Terminal Emulator.

CLIs



Virtual Consoles

TTY

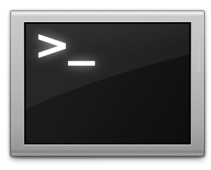
<Ctrl><Alt><F1> **/dev/tty1**

<Ctrl><Alt><F2> **/dev/tty2**

…

<Ctrl><Alt><F6> **/dev/tty6**

No need for a graphic server



Terminal Emulator or pseudo-terminal

xterm (classical from X)
gnome-terminal (GNOME)
konsole (KDE),etc.

<Ctrl><Alt><F7> **/dev/pts/X**

Graphic server running

Outline

su

- The **su** command stands for “switch user”.
- Useful for changing the user without having “to relog“.
- It allows you to become another user or execute commands as another user:

```
$ su telematic
```

- You are prompted to enter **the password associated with the account to which you are switching**.
- To exit: type exit or **Ctrl-d**.
- Per-command **su**:

```
$ su user -c command
```

- However, **su** is potentially dangerous.
- E.g. people knowing the password of the root.

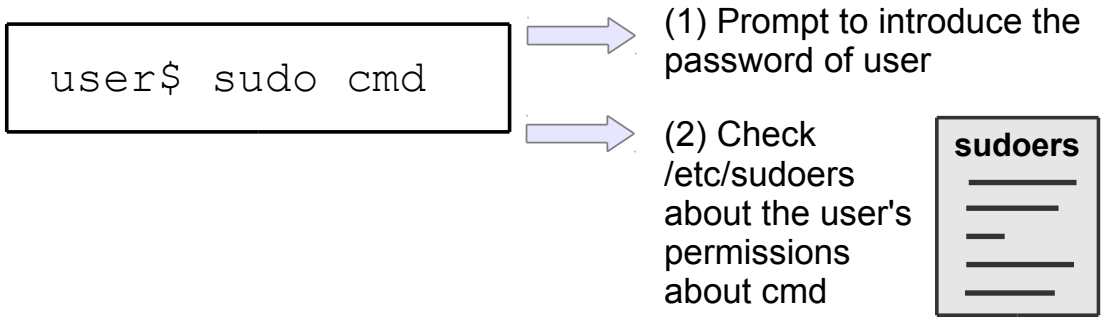
sudo i

- Using the sudoers file (/etc/sudoers), system administrators can define which users or groups will be able to execute certain commands (or even any command) as root.
- The advantage is that none of these users will have to know the password of root.
- The **sudo** command **prompts to introduce the password of the user that is executing sudo**.

```
$ sudo command
```

- Additionally, if your user is configured as system administrator in the sudoers file you can get a shell as root.

sudo ii



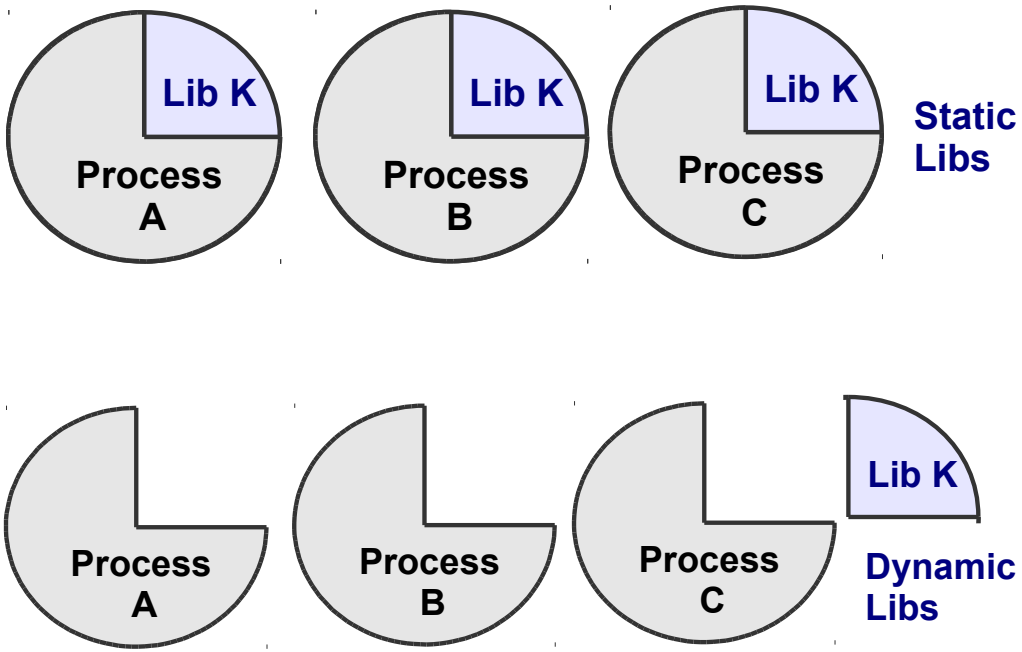
- To become root you can type:

```
user$ sudo -s
root#
```

- Note. We will use $ to mean that the command is being executed as a regular user and # to mean that the command is being executed as root.
- You can use the command `whoami` to know which user you are.

Outline

Types of Libraries i



Types of Libraries ii

- **Static libraries** provide a set of functions that are included at compile-time into the final executable file.
- **Dynamic libraries** allow its set of functions to be referenced in user code and defined in a shared library to be resolved at run time, that is to say, when the program is loaded to become a process in the system.
- Static libraries pros and cons:
  - Anything used from these libraries is available (with the correct version) before the program is executed and this avoids dependency problems.
  - The main **drawback** is that the size of executables is bigger (more disk and memory).

Types of Libraries iii

- Dynamic libraries pros and cons:
  - Shared library code is not present in the executable.
  - Load time may be also reduced (library code might be in memory).
  - Libraries can be updated without updating applications.
  - The main **drawback** is that they usually establish complex relationships between the different packages of software installed in a system (library versions etc.).

Types of Libraries iv

- A typical “hello world” program in C:

```
/* Hello World program */
#include<stdio.h>
void main()
{
  printf("Hello World\n");
}
```

- We can compile and execute the program:

```
$ gcc -o hello hello.c
$ ./hello
Hello World
```

- By default the `gcc` compiler creates dynamic executables.

Types of Libraries v

- You can check dependencies with the `ldd` command.

```
$ ldd hello
linux-vdso.so.1 =>  (0x00007fff5e7ca000)
libc.so.6=>/lib/x86_64-linux-gnu/libc.so.6 (0x0..)
/lib64/ld-linux-x86-64.so.2 (0x00007fca8f089000)
```

- We can view the size of our executable with the `du` (disk usage) command:

```
$ du hello
12  hello
```

Types of Libraries vi

- Compare this when we statically compile the same program:

```
$ gcc -static -o hello.static hello.c
$ ./hello.static
Hello World
$ ldd hello.static
    not a dynamic executable
$ du hello.static
860 hello.static
```

- The advantages of dynamic executables are clear.
- But the management of the different libraries on the system results in a challenge colloquially known as ”dependency hell”.

Sofware Packages

- A package of software tracks where all its files are, allowing the user to easily manage the installed software: view dependencies, uninstall, etc.
- Generally, in Linux, software packages copy their executables in /usr/bin, their libraries in /usr/lib and their documentation in /usr/share/doc/package/.
- There are multiple different package systems, two main are:
  - Red Hat Packages (**.rpm** files).
  - Debian Packages (**.deb** files).
- In Debian (Ubuntu) we use deb packages.
- With the **dpkg** command we can install and remove deb packages, view the files installed by a package, view the packages installed in the system, etc.

Advanced Package Management i

- The tool **dpkg** does not manage dependencies.
- A new generation of package management systems was developed to make this tedious process easier for the user.
- For the deb packages the tool is called APT.
- With APT, we essentially say “install this package” and all dependent packages will be installed/upgraded as appropriate.
- We need to configure the “repositories” that contain our deb packages.
- For APT, repositories are defined in files in /etc/apt.

Advanced Package Management ii

- Typical APT commands:

APT command	Description
`apt-get update`	update the list available packages from online repositories
`apt-get dist-upgrade`	upgrade specified packages (or all installed packages if none specified)
`apt-get install <package list>`	install latest version of package(s)
`apt-get remove <package list>`	remove specified packages from system
`apt-cache list [package list]`	list available packages from repositories

Advanced Package Management iii

- APT requires to execute `apt-get update` to update the available list of packages available in online repositories.
- If the update is not executed, APT works with the local cache, which might be outdated.
- In an Ubuntu system, we can type the name of an application in the console and, if it is not installed, the system will tell us how to install it:

```
$ xcowsay
The program xcowsay is currently not installed.
You can install it by typing:

sudo apt-get install xcowsay
```

Install Terminator

- We install terminator as follows:

```
$ sudo apt install terminator
```

- You can create horizontal splits with `CRL+shift+o`
- You vertical splits with `CRL+shift+e`

Install VSCodium

We can install vscodium[1] which is a community-driven, freely-licensed binary distribution of Microsoft's editor VSCode:

```
$ wget -qO - https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/raw/master/pub.gpg | \
sudo apt-key add -
$ echo 'deb https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/raw/repos/debs/ vscodium main' | \
sudo tee --append /etc/apt/sources.list.d/vscodium.list
$ sudo apt update && sudo apt install codium
```

More info at: `https://vscodium.com/#install`

[1] Microsoft's vscode source code is open source (MIT-licensed), but the the product available for download (Visual Studio Code) is licensed under this not-FLOSS license and contains telemetry/tracking. According to this comment from a Visual Studio Code maintainer.

Install `zshell` and `oh-my-zsh`

Install zsh:

```
$ sudo apt install zsh
```

Install a nice configuration of zsh called oh-my-zsh for your user:

```
$sh -c "$(curl -fsSL https://raw.githubusercontent.com/robbyrussell/oh-my-zsh/master/tools/install.sh)"
```

You can tune your config in `$HOME/.zshrc`:

1. Add plugins:

```
plugins=(
  git
  docker
)
```

2. To allow exiting with jobs in background:

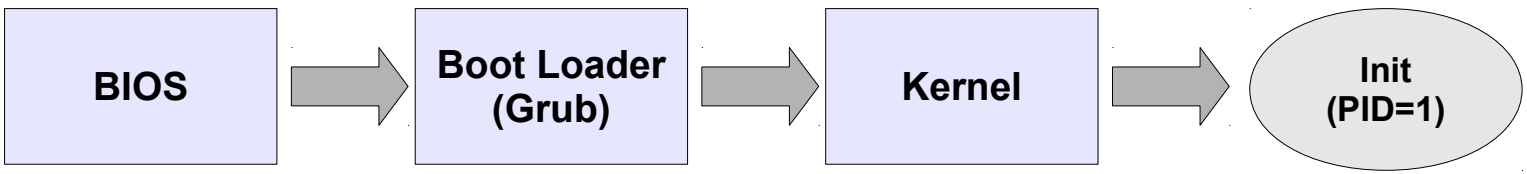
```
setopt NO_HUP
setopt NO_CHECK_JOBS
```

Outline

Outline

Booting

- To boot (start) a Linux system, a sequence is followed in which the control:
  - First goes to the BIOS.
  - Then to a boot loader.
  - Finally, to a Linux kernel (the system core).



- When kernel starts, it executes *init*, the first **process**.

Processes

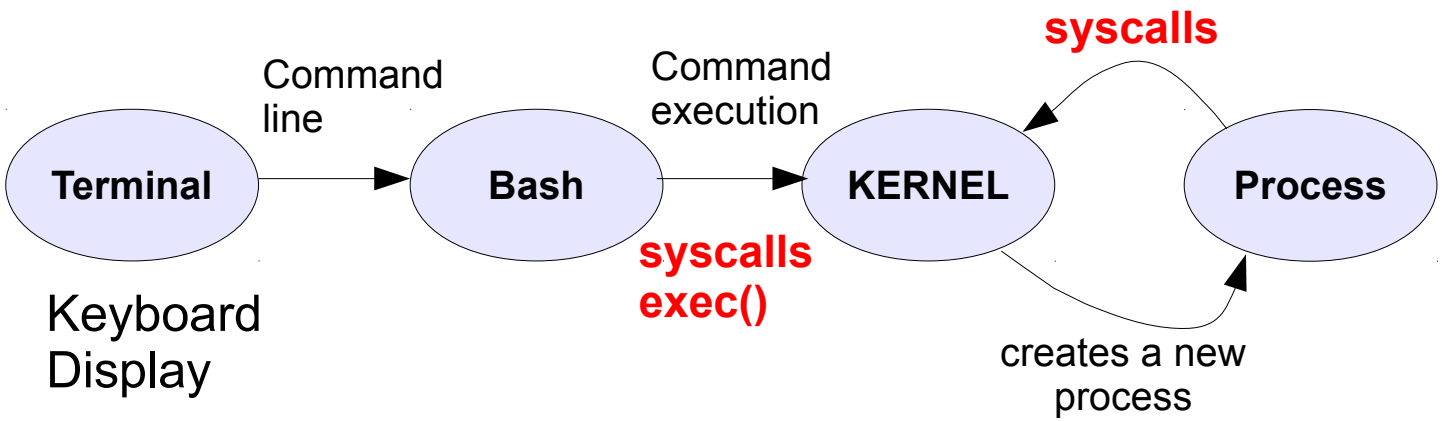
- A process is the abstraction used by the operating system to **represent a running program**.
- Each process in Linux consists of:
  - An address space.
  - A set of data structures within the kernel.
- The address space contains the **code** and **libraries** that the process is executing, the process's **variables**, its **stacks**, and different **additional information** needed by the kernel while the process is running.
- The kernel implements a " CPU scheduler" to share the computing resources.
- Linux processes have "kinship" (parent, child etc.).
- The root of the "tree of processes" is *init*.

Listing processes

- The command `ps` provides information about the processes running on the system.

```
$ ps
  PID TTY          TIME CMD
21380 pts/3    00:00:00 bash
21426 pts/3    00:00:00 ps
```

- We see that two processes `bash` (shell) and `ps` (command).
- The PID is the process identifier.



Command `ps`

- **`ps`** supports many parameters.
- Some of them are (type `man ps`):
  - *-A* shows all the processes from all the users.
  - *-u user* shows proccesses of a particular user.
  - *-f* shows extended information.
  - *-o format* format may be included in a list separated by commas the columns of information you want displayed (use the command `man` for a complete list of possible columns).
  - Examples:

```
$ ps -Ao  pid,ppid,state,tname,%cpu,%mem,time,cmd
$ ps -u user1 -o pid,ppid,cmd
```

Manual

- The `man` command shows the "manual" of other commands.

```
$ man ps
```

- Manual for the command "`ps`".
- Use arrow keys or `AvPag`/`RePaq` to go up and down.
- To search for text *xxx*
  - You can type `/xxx`.
  - To go to the next and previous matches you can press keys `n` and `p` respectively.
- You can use `q` to exit the manual.

Working with the terminal

- History:
  - You can also press the up/down arrow to scroll back and forth through your command history.
  - The history can be seen with the command `history` and you can retype a command with `!number`.
- Completition:
  - When pressing the TAB, bash automatically fills in partially typed commands or parameters.
  - Example: type h+TAB, h+TAB+TAB, hi+TAB+TAB, etc.
- Copy and Paste:
  1. Select text and press the mouse's middle button (or scroll wheel) to paste.
  2. The combinations CRL+SHIFT+c and CRL+SHIFT+v also usually work.

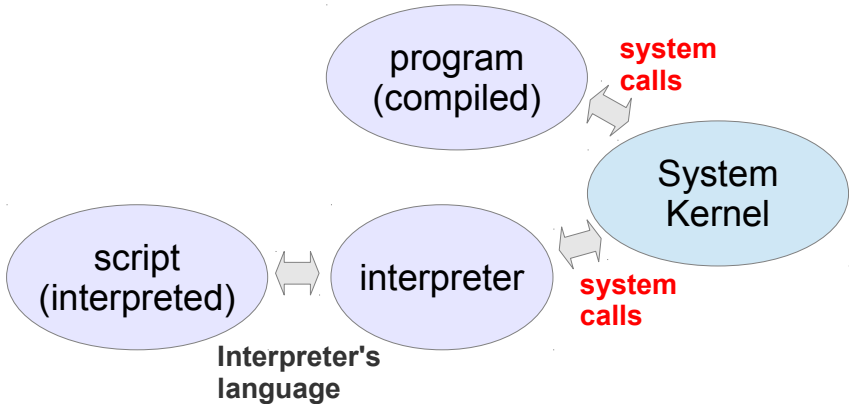
Other commands related to processes

- **`pstree`** displays all system processes tree.
- **`top`** returns a list of processes with information updated periodically.
- **`time`** gives us the duration of execution of a particular command.
  - Real refers to actual elapsed time.
  - User and Sys refer to CPU time used only by the process.

Outline

Script vs. Program

- Programs:
  - The source of a program is first compiled, and the result of that compilation is executed.



  - Examples of languages to build programs: C, C++, etc.
  - Scripts:
    - A script is interpreted. It is written to be understood by an interpreter.
    - Scripting examples: Bash scripts, Python, PHP, Javascript, etc.
    - Typically scripts are written for small applications and they are easier to develop.
    - However, scripts are also usually slower than programs due to the interpretation process.

Shell Scripts i

- A shell script is a text file containing commands and special internal shell commands (if,for, while, etc.).
- The script is interpreted and executed by the shell (bash in most Linux systems).
- The simplest example is:

```
1 pstree
2 sleep 2
3 ps
```

- To run a script you must give it execution permissions:

```
$ chmod u+x myscript.sh
```

- To execute it use:

```
$ ./myscript.sh
```

Shell Scripts ii

- Another example script is the typical “Hello world”:

```
#!/bin/bash
# Our second script, Hello world!
echo Hello world
```

- The script begins with “#!” which contains the path to the shell that will execute the script.
- The lines starting with # are comments.
- To write to the terminal we can use the **echo** command.
- To read you can use the **read** command.

```
#!/bin/bash
# Our third script, using read for fun
echo Please, type a sentence and hit ENTER
read TEXT
echo You typed: $TEXT
```

Outline

Foreground and Background

- By default, the bash executes commands interactively or **foreground**.
- The shell waits until the end of a command before executing another one.

```
$ xeyes
```

- With the ampersand symbol (&), you can execute commands non-interactively or in **background**.

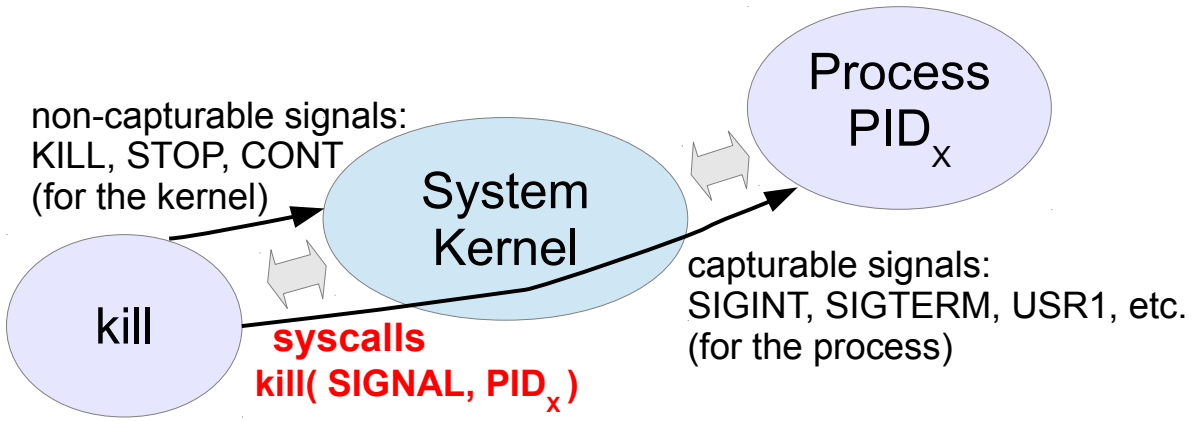
```
$ xeyes &
```

- In foreground or background the output goes to the corresponding terminal.
- You cannot use input from the terminal while in background.

Outline

Signals i

- A signal is a limited form of inter-process communication: signals are INTEGERS.
- Some signals are destined to the kernel (non-capturable) and others to processes running in user space (capturable).



- When the signal is for a process it can be understood as an asynchronous notification.

Signals ii

- When the process receives the signal it interrupts its normal flow of execution and it executes the corresponding signal handler (function).
- In Linux, the most widely used signals and their corresponding integers are:
  - 9 `SIGKILL`. Non-capturable signal sent to the kernel to end a process immediately.
  - 20 `SIGSTOP`. Non-capturable signal sent to the kernel to stop a process. This signal can be generated in a terminal for a process in foreground pressing **`Control-Z`**.
  - 18 `SIGCONT`. Non-capturable signal sent to the kernel that resumes a previously stopped process. This signal can be generated typing **`bg`** in a terminal.

Signals iii

- 2 **SIGINT**. Capturable signal sent to a process to tell it that it must terminate its execution. It is sent in an interactive terminal for the process in foreground when the user presses **Control-C**.
- 15 **SIGTERM**. Capturable signal sent to a process to ask for termination. It is sent from the GUI and also this is the default signal sent by the **kill** command.
- **USR1**. Capturable signal that can be used for any desired purpose.

- Syntax of **kill** command: **kill -signal PID**.

```
$ kill -9 30497
$ kill -SIGKILL 30497
```

- As you can observe, you can use both the number and the name of the signal.

Job Control i

- *Job control* refers to the bash feature of managing processes as jobs.
- We use **``jobs''**, **``fg''**, **``bg''** and the hot keys **Control-z** and **Control-c**.
- `jobs` displays a list of processes launched from a specific instance of bash.
- Each job is assigned an identifier called a JID (Job Identifier).
- **Control-z** sends a stop signal (SIGSTOP) to the process that is running on *foreground*.
- To resume the process that we just stopped, type the command **bg**.
- Typping the JID after the command **bg** will send the process identified by it to *background*.

Job Control ii

- The JID can also be used with the command `kill` using %.
- Another very common shortcut is **Control-c** and it is used to send a signal to terminate (SIGINT) the process that is running on *foreground*.
- Whenever a new process is run in *background*, the bash provides us the JID and the PID:

```
$ xeyes &
[1] 25647
```

- Here, the job has JID=1 and PID=25647.

trap

- `trap` allows capturing and processing signals in scripts.
- Example, if we use this script:

```
1 trap "echo I do not want to finish!!!!" SIGINT
2 while true
3 do
4 sleep 1
5 done
```

- Try to press **Control-z**.

Outline

Running multiple commands i

- The commands can be run in some different ways.
- In general, the command returns 0 if successfully executed and positive values (usually 1) if an error occurred.
- To see the exit status type `echo $?`.
- Try:

```
$ ps -k
$ echo $?
$ ps
$ echo $?
```

Running multiple commands ii

- There are also different ways of executing commands:
  - **`$ command`** the command runs in the foreground.
  - **`$ command1 & command2 & ... commandN &`** commands will run in background.
  - **`$ command1; command2 ... ; commandN`** sequential execution.
  - **`$ command1 && command2 && ... && commandN`** `commandX` is executed if the last executed command has exit successfully (return code 0).
  - **`$ command1 || command2 || ... || commandN`** `commandX` is executed if the last executed command has NOT exit successfully (return code >0).

Outline

Outline

File Systems

- File Systems (FS) define how information is stored in data units like hard drives, tapes, dvds, pens, etc.
- The base of a FS is the file.
- Simplest file: data file.
- The names for files in unix are case sensitive.
- Implemented in the kernel (static or dynamic module).
- FS define meta-data, read/write operations, etc.
- Examples of Disk File Systems (DFS): reiserFS, ext2, ext3, ext4, fat16, fat32 and ntfs.

Basic Types of Files

- **Regular files**. These files contain data.
- **Directory files (folders)**. These files are used to group other files in an structured manner.
- **Special Files**. Within this category there are several sorts of files which have some special content used by the OS.

The command `stat` can be used to view the *basic type* of a file.

File Abstraction

- Unix uses the abstraction of “file” for many purposes.
- This is a fundamental concept in Unix systems.
- This type of abstraction allows using the API of files for devices, e.g. printer.
- Also TTY files are special files.
- Another example of special file is the symbolic link.

Outline

Hierarchical File Systems i

- Linux does not depend on the number of hard disks.
- The whole Unix file system has a unique origin: the root (/).

/	File system root.
/dev	Contains system files which represent devices physically connected to the computer.
/etc	This directory is reserved for system configuration files. This directory cannot contain any binary files (such as programs).
/lib	Contains necessary libraries to run programs in /bin and /sbin.
/proc	Contains special files which receive or send information to the kernel. If necessary, it is recommended to modify these files with ”special caution”.
/bin	Contains binaries of common system commands.
/sbin	Contains binaries of administration commands which can only be executed by the superuser *root*.
/usr	This directory contains the common programs that can be used by all the system users. The structure is the following:
	/usr/bin — General purpose programs (including C/C++ compiler).
	/usr/doc — System documentation.
	/usr/etc — Configuration files of user programs.

Hierarchical File Systems ii

	/usr/include	C/C++ heading files (.h).
	/usr/info	GNU information files.
	/usr/lib	Libraries of user programs.
	/usr/man	Manuals to be accessed by command *man*.
	/usr/sbin	System administration programs.
	/usr/src	Source code of those programs.
	Additionally other directories may appear within */usr*, such as directories of installed programs.	
/var	Contains temporal data of programs (this doesn't mean that the contents of this directory can be erased).	
/mnt or /media	Contains mounted systems of pendrives or external disks.	
/home	Contains the working directories of the users of the system except for root.	

Outline

Storage Devices Mapping

- The kernel automatically detects and maps storage devices in the /dev directory.
- If there is an **IDE controller**:
  - **hda** to IDE bus/connector 0 master device
  - **hdb** to IDE bus/connector 0 slave device
  - **hdc** to IDE bus/connector 1 master device
  - **hdd** to IDE bus/connector 1 slave device
  - Each hard drive can have up to 4 primary partitions (limit of PC x86 architecture) and each primary partition can also have secondary partitions. Each particular partition is identified with a number: e.g. hda1, hda2, etc.
- If there is a **SCSI or SATA controller** these devices are listed as devices sda, sdb, sdc, sdd, sde, sdf, and sdg in the /dev directory. Similarly, partitions on these disks can range from 1 to 16 and are also in the /dev directory.

Mounting a Filesystem

- When a Linux/UNIX system boots, the Kernel requires a “mounted root filesystem”.
- The Kernel has to make this device “usable“.
- In UNIX, this is called ”mounting the filesystem”.
- E.g. Let’s consider that the root filesystem is in /dev/sda1 (SATA disk, first partition).
- We say that the device ”/dev/sda1“ mounts ”/“.
- ”/“ is called the mount point.
- The file /etc/fstab contains the list of devices and their corresponding mount points that are going to be used by the system.

Commands df and du

- `df` (abbreviation for disk free) is used to display the amount of available disk space of file systems:

```
$ df -h
Filesystem     Size  Used Avail Use% Mounted on
/dev/sda1      108G   41G   61G  41% /
none           1,5G  728K  1,5G   1% /dev
none           1,5G  6,1M  1,5G   1% /dev/shm
none           1,5G  116K  1,5G   1% /var/run
none           1,5G     0  1,5G   0% /var/lock
```

- `du` (abbreviated from disk usage) is a used to estimate file space used under a particular directory or by certain files on a file system:

```
$ du -sh /etc/apache2/
464K^^I/etc/apache2/
```

Outline

The Path i

- We have three basic commands to move arround the FS and list its contents:
  - The `ls` command (list) lists the files on the current directory.
  - The `cd` command (change directory) allows us to change from one directory to another.
  - The `pwd` command (print current working) prints the current directory.
- The directories contain two special names:
  - `.` (a dot) which represents the current directory.
  - `..` (two dots) which represent the parent directory.
- With commands related to the filesystem you can use absolute and relative names for the files.

The Path ii

- **Absolute path.** An absolute path always takes the root / of the filesystem as starting point. Thus, we need to provide the full path from the root to the file. Example: `/usr/local/bin`.
- **Relative path.** A relative path provides the name of a file taking the current working directory as starting point. For relative paths we use . (the dot) and .. (the two dots). Examples:
  `./Desktop` or for short `Desktop` (the ./ can be omitted). This is names a file called Desktop inside the current directory.
  `./../../etc` or for short `../../etc`. This names the file (directory) etc, which is located two directories up in the FS.

- The special character ~ (ALT GR+4) can used as the name of your "home directory".

PATH variable

- Normally we do not type any relative or absolute path to the command but just the command name.
- You may wonder how the system knows the path to commands.
- The response is that the system utilizes the environment variable PATH.
- You can check the contents of PATH as:

```
$ echo $PATH
```

Outline

Operations with Directories

- Directories can be created, deleted, moved and copied.
- Commands are `mkdir`, `rmdir`, `rm`, `mv` and `cp`.
- Note `rmdir` fails if the directory is not empty (contains some file or directory).
- There are two ways to proceed:
  - Delete the content and then the directory or
  - Force a recursive removal using `rm -rf`:
- To move (or rename) a directory the command `mv` can be used.
- To copy folder contents to other place within the file system the `cp` may be used with ”-r” to make this copy recursive.

Outline

Operations with Files

- Directories can be created, deleted, moved and copied.
- Commands are `touch`, `rm`, `mv` and `cp`.
- Example moving a file:

```
$ mv test.txt ~/Desktop/
```

- Copied file is renamed if a destination name is specified:

```
$ mv test.txt Desktop/test2.txt
```

- Renaming a file is as easy as:

```
$ mv test.txt test2.txt
```

- The copy command works similar to `mv` without origin:

```
$ cp test.txt test2.txt
```

- View hidden files (start with a dot "."):

```
$ ls -a
```

File Content

- Typically some characters appended at the end of the name of files to point out which is the content of a file.
- These characters are known as the file extension.
- Examples: text files .txt, jpeg images .jpg or .jpeg, html documents .htm .html etc.
- In Unix, the file extension is optional.
- GNU/Linux uses a guessing mechanism called *magic numbers*, in which some tests are performed to figure out the type of content of the file.

The command `file` can be used to guess file content of a file.

Outline

File Expansions & Quoting i

- Bash provides us with some special characters that can be used to name groups of files.
- This is called “filename expansion”.
- We have several expansions:

Character	Meaning
*	Expands zero or more characters (any character).
?	Expands one character.
[ ]	Expands one of the characters inside [ ].
!( )	Expands not the file expansion inside ( ).

- Examples:

File Expansions & Quoting ii

```
$ cp ~/* /tmp            # Copies all files from personal
                         # home folder to /tmp
$ cp ~/[Hh]ello.c /tmp   # Copies Hello.c and hello.c from
                         # home (if they exist) to /tmp.
$ rm ~/hello?            # Removes files in the home folder
                         # called "hello0" or "hellou" but
                         # not "hello" or "hellokitty".
$ rm !(*.jpg)            # Delete everything except files
                         # in the form *.jpg
```

- Filename expansion can be disabled with quoting:

Character	Action
’ (simple quote)	All characters between simple quotes are interpreted without any special meaning.
“ (double quotes)	Special characters are ignored except $, ` ' and \
\ (backslash)	The special meaning of the character that follows is ignored.

- Example:

```
$ rm "hello?"  # Removes a single file called hello?
               # but not, for example,
               # a file called hello1.
```

Outline

Text Files

- Text is organized in bytes that can be read using a character encoding table or charset.
- A text file contains human readable characters such as letters, numbers, punctuation, and also some control characters such as tabs, line breaks, carrier returns, etc.
- The simplicity of text files allows a large amount of programs read and modify text.
- The most well known character encoding table is the ASCII table.
- The ASCII table defines control and printable characters.

ASCII i

- The original specification of the ASCII table defined only 7bits.
- Examples of 7-bit ASCII codification are:

```
a: 110 0001 (97d) (0x61)
A: 100 0001 (65d) (0x41)
```

- Later, the ASCII table was expanded to 8 bits (a byte).
- Examples of 8-bit ASCII codification are:

```
a: 0110 0001
A: 0100 0001
```

- For those bytes whose codification started with 1, several specific encodings per language appeared.

ASCII ii

- The ISO/IEC 8859 standard defines several 8-bit character encodings.
- For instance, ISO/IEC 8859-1 is for Latin languages and includes Spanish or ISO/IEC 8859-7 is for Latin/Greek alphabet.
- An example of 8859-1 codification is the following:

  `ç (ASCII): 1110 0111 (231d) (Oxe7)`

UTF8

- Nowadays, we have other types of encodings.
- The most remarkable is UTF-8 (UCS Transformation Format 8-bits), which is the default text encoding used in Linux.
- UFT-8 defines a variable length universal character encoding.
- In UTF-8 characters range from one byte to four bytes.
- UTF-8 matches up for the first 7 bits of the ASCII table, and then is able to encode up to 2^{31} characters unambiguously (universally).
- Example:

  `ç (UTF8): 0xc3a7`

Newlines i

- A new line, line break or end-of-line (EOL) is a special character or sequence of characters signifying the end of a line of text.
- Systems based on ASCII or a compatible character set use either:
  - LF (Line feed, "`\n`", 0x0A, 10 in decimal).
  - CR (Carriage return, "`\r`", 0x0D, 13 in decimal).
  - CR followed by LF (CR+LF, "`\r\n`", 0x0D0A).

Newlines ii

- The actual codes representing a newline vary across operating systems:
  - **CR+LF**: Microsoft Windows...
  - **CR**: Commodore 8-bit machines, Mac OS up to version 9 and OS-9...
  - **LF**: Unix and Unix-like systems...
- The different codifications for the newline can be a problem when exchanging data between systems with different representations.

Applications for Text i

- On CLI there are text editors: most well-known is `vi`:
  - The next command opens myfile.txt with `vi` in *command mode*:

```
$ vi myfile.txt
```

  - In this mode:
    - We can navigate through myfile.txt.
    - Delete a line: *dd*
    - Delete from cursor to line end: *d$*
    - Delete from cursor to line beginning: $\hat{d}$
    - Go to line: *Gn* (*n* is the line number)
  - If we want to edit the file, we have to press "`i`", which puts `vi` in *insertion mode*.
  - After modifying the document, we can hit ESC to go back to *command mode* (default one).
  - To save the file we must type `:wq`.

Applications for Text ii

  - To quit without saving, we must type `:q!`.
- There are also GUI text editors: example ”gedit”.
- There are also other commands to view text files like `cat`, `more` and `less`.
- The `less` command works in the same way as `man`.
- Another couple of useful commands are `head` and `tail`, which respectively, show us the text lines at the top of the file or at the bottom of the file.

```
$ head /etc/passwd
$ tail -3 /etc/passwd
```

- A very interesting option of tail is *-f*, which outputs appended data as the file grows:

```
$ tail -f /var/log/syslog
```

Applications for Text iii

- The `grep` command allows us to search for a pattern within a file.

```
$ grep bash /etc/passwd
$ grep -v bash /etc/passwd
```

- The command `cut` can be used to split text content using a specified delimiter:

```
$ cat /etc/passwd /etc/group
$ cut -c 1-4 /etc/passwd
$ cut -d ":" -f 1,4 /etc/passwd
$ cut -d ":" -f 1-4 /etc/passwd
```

Applications for Text iv

- If we have a binary file, we can use `hexdump` or `od` to see its contents in hexadecimal and also in other formats.
- Another useful command is `strings`, which will find and show characters or groups of characters (strings) contained in a binary file. Try:

```
$ hexdump /bin/ls
$ strings /bin/ls
$ cat /bin/ls
```

Outline

Links i

- A *Hard Link* is just another name for the same file.
  - The associated name is a simple label stored somewhere within the file system.
  - Hard links can just refer to existent data in the same file system.
  - In most of FS, all files are hard links.
  - Even named differently, the hard link and the original file offer the same functionality.
  - Any of the hard links can be used to modify the data of the file.
  - A file will not exist anymore if all its hard links are removed.

Links ii

- A *Symbolic Link* (also Soft Link) is considered a new file whose contents are a pointer to another file or directory.
  - If the original file is deleted, the link becomes unusable.
  - The link is usable again if original file is restored.
  - Soft links allow to link files and directories between different FS, which is not allowed by hard links.
- The `ln` command is used to create links.
- If the `-s` option is passed as argument, the link will be symbolic.

Outline

Unix Permission System i

- Unix operating systems are organized in users and groups.
- Upon entering the system, the user must enter a login name and a password.
- A user can belong to several groups, but at least the user must belong to one group.
- Linux FS provides us with the ability of having a strict control of files and directories.
- The basic mechanism (despite there are more mechanisms available) is the Unix Filesystem permission system.
- There are three specific permissions on Unix-like systems: read, write and execute.
- A user is in one of the three following categories or classes:
    1. User Class. The user is the owner of the file or directory.

Unix Permission System ii

2. Group Class. The user belongs to the group of the file or directory.
3. Other Class. Neither of the two previous situations.

- The most common form of showing permissions is symbolic notation.
  - `-rwxr-xr-x` for a regular file whose user class has full permissions and whose group and others classes have only the read and execute permissions.
  - `dr-x------` for a directory whose user class has read and execute permissions and whose group and others classes have no permissions.
- The command to list the permissions of a file is `ls -l`.

Change permissions (chmod) i

- The command **chmod** is used to change the permissions of a file or directory.
- Syntax: `chmod user_type operation permissions file`

User Type	u	user
	g	group
	o	other
Operation	+	Add permission
	-	Remove permission
	=	Assign permission
Permissions	r	reading
	w	writing
	x	execution

Change permissions (chmod) ii

- Another way of managing permissions is to use octal notation. With three-digit octal notation:

```
0 --- no permission
1 --x execute
2 -w- write
3 -wx write and execute
4 r-- read
5 r-x read and execute
6 rw- read and write
7 rwx read, write and execute
```

- Examples:

Change permissions (chmod) iii

```
$ chmod g+rx file1.c
$ chmod u=r file1.c
$ chmod u=r,g=rx file1.c
$ chmod 654 file1.c
```

Default permissions

- Users can also establish the default file permissions for their new created files.
- The `umask` command allows to define these default permissions. When used without parameters, returns the current mask value:

```
$ umask
0022
```

- You can also set a mask. Example:

```
$ umask 0044
```

- The two permissions that can be used by default are read and write but not execute.
- The mask tells us in fact which permission is subtracted (i.e. it is not granted).

Outline

Outline

What is a File Descriptor?

- An fd is an integer that is used as index of a kernel-resident data structure containing the details of all **open files**.
- Each process has its own “file descriptor table“.
- The same file can have different file descriptors:
  - We can open a file for reading and get one fd.
  - We can open the same file for writing and get another fd.
  - One of the main elements contained in the file descriptor table is the file pointer, which indicates the current position (for r/w) on a file.

Standard fds i

- There are 3 standard file descriptors which presumably every process should have opened.
- When a process is created using a shell (like Bash), it inherits the 3 standard file descriptors from this shell.
- The `lsof` command shows us the “list of open files”:

```
$ ps
 PID TTY          TIME CMD
7988 pts/3    00:00:00 bash
8839 pts/3    00:00:00 ps
```

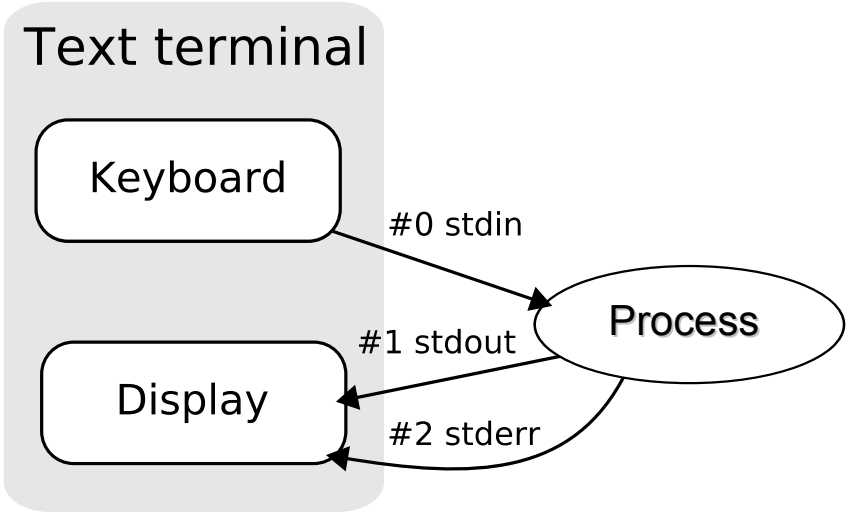
```
$ lsof -a -p 7988 -d0-10
COMMAND  PID     USER    FD TYPE DEVICE NAME
bash    7988 telematics 0r CHR  136,3  /dev/pts/3
bash    7988 telematics 1w CHR  136,3  /dev/pts/3
bash    7988 telematics 2w CHR  136,3  /dev/pts/3
```

Standard fds ii

- The previous command shows the file descriptors active (up to fd=10) for the process with PID 7988.
- fd=0 is opened for reading (0r) and fd=1,2 are opened for writing (1w and 2w).

fd (integer value)	Name
0	Standard Input (STDIN)
1	Standard Output (STDOUT)
2	Standard Error (STDERR)

Standard fds iii



Outline

Redirecting Output i

- The “output redirection” allows to send STDOUT/STDERR to files different from default ones.
- Examples:

```
$ echo Hello, how are you?
Hello, how are you?
```

```
$ echo Hello, how are you? > file.txt
$ cat file.txt
Hello, how are you?
```

```
$ echo Are you ok? >> file.txt
$ cat file.txt
Hello, how are you?
Are you ok?
```

- You can also redirect the standard error (*fd*=2).

Redirecting Output ii

- Example:

```
$ ls -qw
$ ls -qw 2> error.txt
```

- In general:
  - **N>file**. It is used to redirect fd=N to a file. If the file exists, is deleted and overwritten. In case file does not exist, it is created.
  - **N>>file**. Similar to the first case but opens file in mode append.
  - & **>file**. Redirects STDOUT and STDERR to file.

Outline

Redirecting Input i

- Input redirection allows you to specify a file for reading standard input (*fd=0*).
- The format for input redirection is $<$ *file*.
- Example:

```
1 #!/bin/bash
2 # Our third script, using read for fun
3 echo Please, type a sentence and hit ENTER
4 read TEXT
5 echo You typed: $TEXT
```

- To redirect the input:

Redirecting Input ii

```
$ echo hello world > file.txt
$ ./thirdscript.sh < file.txt
Please, type a sentence and hit ENTER
You typed: hello world
$
```

Redirecting Input iii

- Let's observe redirections with `lsof`:

```
1 #!/bin/bash
2 lsof -a -p $$ -d0-10
3 echo Now, please type a sentence and hit ENTER
4 read TEXT
5 echo You typed: $TEXT
```

```
$ ./thirdscript.sh < file
COMMAND PID USER  FD  TYPE DEVICE NAME
script  7728 user  0r  REG   8,6  /home/user/file
script  7728 user  1u  CHR 136,3  /dev/pts/3
script  7728 user  2u  CHR 136,3  /dev/pts/3
Now, please type a sentence and hit ENTER
You typed: hola
```

- The variable $ contains the PID of the script in execution.

Outline

Unnamed pipes i

- Unix based operating systems like Linux offer a unique approach to join two commands on the terminal.
- You can take the output of the first command and use it as input of the second command, this is the concept of pipe or "|".

```
$ ls | grep x
```

- This type of pipe is called "unnamed pipe" because the pipe exists only inside the kernel and cannot be accessed by processes that created it, in this case, the bash shell.
- All Unix-like systems include a variety of commands to manipulate text outputs.
- We have already seen some of these commands: `head`, `tail`, `grep` and `cut`.

Unnamed pipes ii

- We also have other commands for text pipelines like:
  - `uniq` which displays or removes repeating lines.
  - `sort` which lists the contents of the file ordered alphabetically or numerically.
  - `wc` which counts lines, words and characters.
  - `find` which searches for files.
- The following example shows a compound command with several pipes:

```
$ cat *.txt | sort | uniq > result.txt
```

- `tee` reads STDIN and writes to a file and also to STDOUT.

```
$ ls | tee output.txt | sort -r
```

Outline

Outline

What is a Shell Script?

- Some of the files on a UNIX system are deemed *executable*.
- These can be thought of as *programs*
- Some of these programs are compiled from source code (such as C).
- These are sometimes known as *binary executables*.
- Other executables contain only text to be interpreted.
- These are called *scripts*.
- We have scripts for shells.
- Also, there are other different interpreters for scripts, such as `python` and `perl`.

Which Shell?

- There are a variety of UNIX shells to choose from.
- The original and most widely supported shell is called the *Bourne shell* (after S.R. Bourne).
- The binary is usually in /bin/sh.
- 95% of the world's shell scripts are created for use with the Bourne Shell.
- There are a number of Bourne shell derivatives, each offering a variety of extra features, including:
  - The Korn shell (after David Korn) (**ksh**) (not open/free).
  - The Bourne-again shell (**bash**) (open/free).
  - zsh.
- Such shells are supposed to be Bourne-compatible.

Shebang

- The first 2 chars of a script should be “#!”.
- This is called the **shebang**.
- The kernel executes the name that follows, with the file name of the script as a parameter.
- Example: a file called `find.sh` has this as the first line:

```
#! /bin/sh
```

- Then, kernel executes:

```
/bin/sh find.sh
```

- For bash scripts:

```
#! /bin/bash
```

- If you make any typing mistake in the name of the interpreter, you will get an error message such as
  `bad interpreter: No such file or directory.`

Outline

Quoting i

- Quoting is used when defining variables.

```
$ MYVAR=Hello
```

- If we want to define a variable with a value that contains spaces or tabs we need to use quotes.

```
$ MYVAR='Hello World'
```

- Single quotes mean literal:

```
#!/bin/bash
MYVAR='Hello world'
MYVAR2='$MYVAR, How are you?'
echo $MYVAR2
```

Quoting ii

	Quotes	Meaning
'	simple	The text between single quotes is treated as a literal (without modifications). In bash, we say that it is not *expanded*.
"	double	The text between double quotes is treated as a literal except for what follows to characters \, `and $.
`	reversed	The text between reverse quotes is interpreted as a command, which is executed and whose output is used as value. In bash, this is known as *command expansion*.

- Example double quotes:

```
#!/bin/bash
MYVAR='Hello world'
echo "$MYVAR, how are you?"
```

Quoting iii

- Reverse quotes cause the quoted text to be interpreted as a shell command:

```
$ echo "Today is `date`"
Today is Tue Aug 24 18:48:08 CEST 2008
```

- Braces {} are used to separate variables:

```
$ MYVAR='Hello world'
$ echo "foo$MYVARbar"
foo
```

- We have to use braces to resolve the ambiguity:

```
$ echo foo${MYBAR}bar
fooHello worldbar
```

Outline

Positional & Special Params i

- Often, you will need that your script can process arguments when invoked.
- These arguments are called positional parameters: $1 to $N.
- $0 is the basename of the script as it was called.

```
#!/bin/bash
echo "$1"
```

- Execute:

```
$ ./my_script.sh Hello word
Hello
$ ./my_script.sh 'Hello word'
Hello word
```

Positional & Special Params ii

- Another example:

```
#!/bin/bash
echo Script name is "$0"
echo First positional parameter $1
echo Second positional parameter $2
echo Third positional parameter  $3
echo The number of positional parameters is $#
echo The PID of the script is $$
```

- $# expands to the number of positional parameters.
- $$ expands to the PID of the bash that is executing the script,
- $@ expands to all the positional parameters (also $* does this).
  - $@ and $* both expand to all the positional parameters.
  - There are no differences between $* and $@,
  - But there is a difference between "$@" and "$*".
  - "$*" means always one single argument,

Positional & Special Params iii

- ”$@” contains as many arguments.

Positional & Special Params iv

- Example:

```
$ cat script1.sh
mkdir "$*"
$ cat script2.sh
mkdir "$@"
$ ./script1 dir1 dir2 ; ./script2 dir3 dir4
$ ls | grep dir
dir1 dir2
dir2
dir3
```

- ”$@” is a special token which means ”wrap each individual argument in quotes”.

Outline

Expansions

- Before executing your commands, bash checks whether there are any syntax elements in the command line that should be interpreted rather than taken literally.
- These are called expansions and we have several types.
- In processing order they are:
  - **Brace Expansion**: create multiple text combinations.
  - **Tilde Expansion**: expand useful pathnames home dir, working dir and previous working dir.
  - **Parameter Expansion**: how bash expands variables to their values.
  - **Command Substitution**: using the output of a command as an argument.
  - **Arithmetic Expansion**: how to use arithmetics.
  - **Process Substitution**: a way to write and read to and from a command.
  - **Filename Expansion**: a shorthand for specifying filenames matching patterns.

Brace Expansion

- Brace expansions are used to generate all possible combinations with the optional surrounding preambles and postscripts.
- The general syntax is:
  `[preamble]{X,Y[,...]}[postscript]`
- Examples:

```
$ echo a{b,c,d}e
abe ace ade ^^I^^I^^I
$ echo "a"{b,c,d}"e"^^I^^I
abe ace ade
$ echo "a{b,c,d}e"
a{b,c,d}e
$ mkdir $HOME/{bin,lib,doc}
$ echo {x,y}{1,2}
```

- The are also a couple of alternative syntaxes for brace expansions using two dots:

```
$ echo {5..12}
5 6 7 8 9 10 11 12
```

Tilde Expansion

- The tilde expansion is used to expand three specific pathnames:
  - Home directory: `~`
  - Current working directory: `~+`
  - Previous working directory: `~-`
- Examples:

```
$ cd /
/$ cd /usr
/usr$ cd ~-
/$ cd ~
~$ cd /etc
/etc$ echo ~+
/etc
```

Parameter Expansion i

- Parameter expansion allows us to get the value of a parameter (variable).
- On expansion time, you can do extra processing with the parameter or its value.
- **${VAR:-string}** If the parameter VAR is not assigned, the expansion results in 'string'. Otherwise, the expansion returns the value of VAR. For example:

```
$ VAR1='Variable VAR1 defined'
$ echo ${VAR1:-Variable not defined}
Variable VAR1 defined
$ echo ${VAR2:-Variable not defined}
Variable not defined
```

Parameter Expansion ii

- You can also use positional parameters. Example:

```
USER=${1:-joe}
```

- **${VAR:=string}** If the parameter VAR is not assigned, VAR is assigned to 'string' and the expansion returns the assigned value. Note. Positional and special parameters cannot be assigned this way.
- There are more parameter expansions described in our book.

Command Substitution

- Command substitution yields as result the standard output of the command executed.
- Syntaxes:
  `$(COMMAND)` or `` `COMMAND` ``
- We also can use command substitution together with parameter expansion.
- Example:

```
$ echo VAR=${VAR1:-Parameter not defined in `date`}
Parameter not defined in Thu Mar 10 15:17:10 CET 2011
```

Arithmetic Expansion

- **bash** is primary designed to manipulate text strings but it can also perform arithmetic calculations.
- Syntax is: ((...)) or $[...]
- Examples:

```
VAR=55              # Asign the value 55 to the variable VAR.
((VAR = VAR + 1))   # Adds one to the variable VAR (notice that we
((++VAR))           # C style to add one to VAR (preincrease).
((VAR++))           # C style to add one to VAR (postincrease).
echo $[VAR * 22]    # Multiply VAR by 22
echo $((VAR * 22)) # Multiply VAR by 22
```

- We also can use the extern command **expr** to do arithmetic operations:

```
$ X=`expr 3 \* 2 + 7`
$ echo $X
13
```

- However, **expr** is a new process (less efficient).
- **bash** cannot handle floating point calculations.

Process Substitution

- When you enclose several commands in parenthesis, the commands are actually run in a “subshell”.
- Since the outer shell is running only a “single command”, the output of a complete set of cmds can be redirected as a unit.
- Process substitution occurs when you use $<$ or $>$ with parenthesis:
  - cmdX $<$(cmds) redirects the output of cmds to the input of cmdX. For example:

```
$ cat <(ls -l)
```

  - cmdX $>$(cmds) redirects the output of cmdX to the input of cmds. For example:

```
$ (ps ; ls) >commands.out
```

- In detail, the previous command line uses a temporary named pipe, which bash creates, names and later deletes.

Filename Expansion

- A pattern is replaced by a list names of files sorted alphabetically. Possible patterns:

Format	Meaning
*	Any string of characters, including a null (empty) string.
?	Any unique character.
[*List*]	A unique character from the list. We can include range of characters separated by hyphens (-). If the first character of the list is ^ or !, this means any single character that it is not in the list.

- Examples:

```
$ ls *.txt
file.txt  doc.txt  tutorial.txt
$ ls [ft]*
file.txt  tutorial.txt
$ ls [a-h]*.txt
file.txt  doc.txt
$ ls *.??
script.sh file.id
```

- Filename expansions use glob patterns, which are a type of regular expression.

Outline

Introduction

- A regular expression is a special text string for describing a search pattern mainly for use in find and replace like operations.
- In Linux systems, we have to types of regular expressions: **glob patterns** and **regular expressions (regex)**.
- glob patterns are the oldest and simplest type of regular expressions used in Unix-like systems.
- They are used by commands related with the file system like `ls`, `rm`, `cp` etc.
- regex are more versatile regular expressions.
- regex are used by commands related to string manipulation and filtering like `grep` or `sed`.

Glob Patterns

- We have two sets of globs, basic and extended.
- The **basic globs** and their meaning are listed below:

Character	Meaning
?	Expands one character.
*	Expands zero or more characters (any character).
[ ]	Expands one of the characters inside [ ].

Extended globs

- To use the following expansions you have to have activated extended globbing.
- You can check this with:

```
$ shopt -s extglob
```

- The **extended globs** and their meaning are listed below:
  - ?(...|...) Expands **zero or one** occurrence of the items listed between the pipes
  - *(...|...) Expands **zero or more** occurrence of the items listed between the pipes
  - +(...|...) Expands **one or more** occurrence of the items listed between the pipes
  - @(...|...) Expands **any** occurrence of the items listed between the pipes
  - !(...|...) Expands anything **except** the items listed between the pipes

Examples of globs

- Examples:

```
$ touch 03.txt ; touch {aa,b,c}{00,01,02}.txt; ls
03.txt a00.txt  a01.txt  a02.txt  aa00.txt  aa01.txt  aa02.txt
b00.txt  b01.txt  b02.txt  c00.txt  c01.txt  c02.txt

$ ls *(a|b)??.txt
03.txt a00.txt  a01.txt  a02.txt  aa00.txt  aa01.txt  aa02.txt
b00.txt  b01.txt  b02.txt

$ ls ?(a)0*
03.txt  a00.txt  a01.txt  a02.txt

$ ls +(a|?0)*
a00.txt  a02.txt   aa01.txt  b00.txt  b02.txt  c01.txt
a01.txt  aa00.txt  aa02.txt  b01.txt  c00.txt  c02.txt

$ ls @(aa|*2)*
a02.txt  aa00.txt  aa01.txt  aa02.txt  b02.txt  c02.txt

$ ls !(a0*|aa0*)
03.txt b00.txt  b01.txt  b02.txt  c00.txt  c01.txt  c02.txt
```

Regular Expressions (regex)

- Regular Expresions (regex) provide three main features:
  - **Concatenation**. RE ab matches an input string of ab.
  - **Union**. RE $a|b$ matches an input string of a or an input string of b. It does not match ab.
  - **Closure**. RE a* matches the empty string, or an input string of a, or an input string of aa, etc.
- We have two sets of regex, basic and extended.

Basic Regex i

- The **basic regex** and their meaning are listed below:
  - \ is used to escape special characters.
  - . matches any single character of the input line.
  - ^ This character does not match any character but represents the beginning of the input line. For example, ^A is a regular expression matching the letter A at the beginning of a line.
  - $ This represents the end of the input line.
  - [ ] A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches a, b, or c. [a-z] specifies a range which matches any lowercase letter from a to z. These forms can be mixed: [abcx-z] matches a, b, c, x, y, or z, as does [a-cx-z].

Basic Regex ii

- `[^ ]` Matches a single character that is not contained within the brackets.
- RE* A regular expression followed by * matches a string of zero or more strings that would match the RE. For example, A* matches A, AA, AAA, and so on. It also matches the null string (zero occurrences of A).
- `\( \)` are used for grouping. The RE inside the escaped parenthesis creates a group that can be referenced with `\1` through `\9` (up to nine groups can be created).
- `\{ \}` means the same as {m,n} (without backslash) does in Extended regex (see next Section).

- Example:

```
$ ps -u $USER | grep '^ [0-9][0-9][0-9]9'
```

Extended regex

- The **extended regex** and their meaning are listed below:
  - `RE+` A regular expression followed by + matches a string of one or more strings that would match the RE.
  - `RE?` A regular expression followed by ? matches a string of zero or one occurrences of strings that would match the RE.
  - `( )` which are parenthesis without backslash, are used for grouping in extended regex.
  - `{}` is used to create unions. Examples:
    - `a{3}` is equivalent to regular expression aaa (exactly `a` three times).
    - `a{3,}` is equivalent to `aaaa*` (3 or more `a`).
    - `a{,3}` is equivalent to `|a|aa|aaa` (matches the empty string or `a` or `aa` or `aaa`).
    - `a{3,5}` is equivalent to regular expression `aaa|aaaa|aaaaa`.

Examples of regex

- `abc` matches any line of text containing abc.
- `.at` matches any three-character string ending with at, including hat, cat, and bat.
- `[hc]at` matches hat and cat.
- `[^b]at` matches all strings matched by .at except bat.
- `^[hc]at` matches hat and cat at the beginning of string.
- `[hc]at$` matches hat and cat at the end of string.
- `\[.\]` matches any single character surrounded by [], for example: [a] and [b].
- `^.$` matches any line containing exactly one character (the newline is not counted).
- `.* [a-z]+ .*` matches any line containing a word, consisting of lowercase alphabetic characters, delimited by at least one space on each side.

Outline

Outline

If i

- The clause `if` is the most basic form of conditional.
- The syntax is:
  `if` **expression** `then` **statement1** `else` **statement2** `fi`.
- Where **statement1** is only executed if **expression** evaluates to true and **statement2** is only executed if **expression** evaluates to false.
- Examples:

```
if [ -e /etc/file.txt ]
  then
  echo "/etc/file.txt exists"
  else
  echo "/etc/file.txt does not exist"
fi
```

If ii

- In this case, the expression uses the option `-e file`, which evaluates to true only if “file” exists.
- Be careful, you must leave spaces between “[” and “]” and the expression inside.

If iii

- With the symbol “`!`” you can do inverse logic.
- We can also create expressions that always evaluate to true or false:

```
if true
  then
  echo "this will always be printed"
  else
  echo "this will never be printed"
fi
if false
  then
  echo "this will never be printed"
  else
  echo "this will always be printed"
fi
```

- Other operators for expressions are the following.

If iv

- File Operators:

```
[ -e filename ] true if filename exists
[ -d filename ] true if filename is a directory
[ -f filename ] true if filename is a regular file
[ -L filename ] true if filename is a symbolic link
[ -r filename ] true if filename is readable
[ -w filename ] true if filename is writable
[ -x filename ] true if filename is executable
[ filename1 -nt filename2 ] true if filename1 is more recent than
[ filename1 -ot filename2 ] true if filename1 is older than filena
```

- String comparison:

```
[ -z string ] true if string has zero length
[ -n string ] true if string has nonzero length
[ string1 = string2 ] true if string1 equals string2
[ string1 != string2 ] true if string1 does not equal string2
```

- Arithmetic operators:

```
[ X -eq Y ] true if X equals Y
[ X -ne Y ] true if X is equal to Y
[ X -lt Y ] true if X is less than Y
[ X -le Y ] true if X is less or equal than Y
[ X -gt Y ] true if X is greater than Y
[ X -ge Y ] true if X is greater or equal than Y
```

- The syntax ((...)) for arithmetic expansion can also be used in conditional expressions.
- The syntax ((...)) supports the following relational operators: $==, !=, >, <, >=, y <=$.
- Example:

```
if ((VAR == Y * 3 + X * 2))
then
   echo "The variable VAR  is equal to Y * 3 + X * 2"
fi
```

- We can also use conditional expressions with the OR or with the AND of two conditions:

```
[ -e filename -o -d filename ]
[ -e filename -a -d filename ]
```

- **In general it is a good practice to quote variables** inside your conditional expressions:

If vii

```
if [ $VAR = 'foo bar oni' ]
then
echo "match"
else
echo "not match"
fi
```

- The previous expression might not behave as you expect.
- If the value of VAR is “foo”, we will see the output “not match”, but if the value of VAR is ”foo bar oni”, bash will report an error saying ”*too many arguments*”.
- The problem is that spaces present in the value of VAR confused bash.
- In this case, the correct comparison is:

If viii

```
if [ "$VAR" = 'foo bar oni' ]
```

- Recall that you have to use double quotes to use the value of a variable (i.e. to allow parameter expansion).

If with Regular Expressions

- We can use **glob patterns** and **regex** in conditional statements.
- Syntax `[[ ... ]]`
- When comparing with glob patters we have to use `==`
- When comparing with regex we have to use `=~`.
- Examples:

```
[[ $a == z* ]]   # True if $a starts with an "z" (pattern matching
[[ $a == "z*" ]] # True if $a is equal to z* (literal matching).
```

- In the previous example, note that quotes deactivate the meaning of the glob pattern (also happens with regex).
- Another example with globbing:

```
if [[ $VAR == ??grid* ]] ; then echo $VAR; fi
```

- The same example using a regex:

```
if [[ $VAR =~ ^..grid.*$ ]] ; then echo $VAR; fi
```

Conditions on Execution i

- Each command has a return value or exit status.
- Exit status is used to check the result of the execution of the command.
- If the exit status is zero, this means that the command was successfully executed.
- The special variable $? is a shell built-in variable that contains the exit status of the last executed command.
- For a script, $? returns the exit status of the last executed command in the script or the number after the keyword `exit`:

Conditions on Execution ii

```
command
if [ "$?" -ne 0]
 then
 echo "the previous command failed"
 exit 1
fi
```

- We can also replace the previous code by:

```
$ command || echo "the previous command failed"; exit 1
```

- We can also use the return code of conditions.
- Conditions have a return code of 0 is the condition is true or 1 if the condition is false.
- Using this, we can get rid of the keyword `if` in some conditionals:

Conditions on Execution iii

```
$ [ -e filename ] && echo "filename exists" || echo "filename does
```

- Based on this return code, the echo is executed.

For i

- A `for` loop is classified as an iteration statement.
- The syntax is:

```
for VARIABLE in item1 item2 item3 item4 item5 ... itemK
do
^^Icommand1
^^Icommand2
^^I...
        commandN
done
```

- The previous `for` loop executes K times a set of N commands.
- You can also use the values (item1, item2, etc.) that takes your control VARIABLE in the execution of the block of commands:

For ii

```
#!/bin/bash
for X in one two three four
do
echo number $X
done
```

- for accepts any list after the key word "*in*":

```
#!/bin/bash
for FILE in /etc/r*
do
if [ -d "$FILE" ]
then
echo "$FILE (dir)"
else
echo "$FILE"
fi
done
```

For iii

- Furthermore, we can use *filename expansion* to create the list:

```
for FILE in /etc/r??? /var/lo* /home/student/* /tmp/${MYPATH}/*
do
cp $FILE /mnt/mydir
done
```

- We can use relatives paths too:

```
for FILE in ../* documents/*
do
echo $FILE is a silly file
done
```

- In the previous example the expansion is relative to the script location.

For iv

- We can make loops with the positional parameters of the script as follows:

```
#!/bin/bash
for THING in "$@"
do
echo You have typed: ${THING}.
done
```

- With the command `seq` or with the syntax (()) we can generate C-styled loops:

```
      #!/bin/bash
      for i in `seq 1 10`;
      do
      echo $i
      done
```

For v

```
#!/bin/bash
for (( i=1; i < 10; i++));
do
echo $i
done
```

While

- A `while` executes a code block while a expression is true:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
^^Iecho $X
^^IX=$((X+1))
done
```

- The loop is executed while the variable X is less or equal (-le) than 20.
- We can also create infinite loops, example:

```
#!/bin/bash
while true
 do
  sleep 5
  echo "Hello I waked up"
 done
```

Case i

- A `case` construction is a bash programming language statement which is used to test a variable against set of patterns.
- Often `case` statement let's you express a series of if-then-else statements that check single variable for various conditions or ranges in a more concise way.
- The generic syntax of `case` is the following:

```
case VARIABLE in
    pattern1)
        1st block of code ;;
    pattern2)
        2nd block of code ;;
    ...
esac
```

Case ii

- The pattern is a glob pattern and it can actually be formed of several subpatterns separated by pipe character "|".
- If the VARIABLE matches one of the patterns (or subpatterns), its corresponding code block is executed.

Case iii

- The patterns are checked in order until a match is found; if none is found, nothing happens.
- For example:

```
#!/bin/bash
for FILE in $*; do
  case $FILE in
   *.jpg | *.jpeg | *.JPG | *.JPEG)
   echo The file $FILE seems a JPG file.
   ;;
  *.avi | *.AVI)
   echo "The filename $FILE has an AVI extension"
   ;;
  -h)
   echo "Use as: $0 [list of filenames]"
   echo "Type $0 -h for help" ;;
  *)
   echo "Using the extension, I don't now which type of file is $F
```

Case iv

```
    echo "Use as: $0 [list of filenames]"
    echo "Type $0 -h for help" ;;
  esac
done
```

- The final pattern is *, which is a catchall for whatever didn't match the other cases.

Outline

printf i

- `printf` is a (more complete) command for generating outputs:

```
$ printf "hello printf"
hello printf$
```

- No new line had been printed out as it it in case of when using default setting of `echo` command.
- To print a new line:

```
$ printf "Hello, $USER.\n\n"
```

- or

```
$ printf "%s\n" "hello printf"
hello printf
```

printf ii

- The format string is applied to each argument:

```
$ printf "%s\n" "hello printf" "in" "bash script"
hello printf
in
bash script
```

- %s is used as a format specifier to to print all argument in literal form.
- The specifiers are replaced by their corresponding arguments:

```
$ printf "%s\t%s\n" "1" "2 3" "4" "5"
1       2 3
4       5
```

- The %b specifier is essentially the same as %s but it allows us to interpret escape sequences with an argument:

printf iii

```
$ printf "%s\n" "1" "2" "\n3"
1
2
\n3
$ printf "%b\n" "1" "2" "\n3"
1
2

3
$
```

printf iv

- To print integers, we can use the %d specifier:

```
$ printf "%d\n" 255 0xff 0377 3.5
255
255
255
bash: printf: 3.5: invalid number
3
```

- %d specifiers refuses to print anything than integers.
- To `printf` floating point numbers use %f:

```
$ printf "%f\n" 255 0xff 0377 3.5
255.000000
255.000000
377.000000
3.500000
```

printf v

- The default behavior of %f `printf` specifier is to print floating point numbers with 6 decimal places.

printf vi

- To limit a decimal places to 1 we can specify a precision in a following manner:

```
$ printf "%.1f\n" 255 0xff 0377 3.5
255.0
255.0
377.0
3.5
```

- Formatting to three places with preceding with 0:

```
$ for i in $( seq 1 10 ); do printf "%03d\t" "$i"; done
001     002     003     004     005     006     007     008
009     010
```

- You can also print ASCII characters using their hex or octal notation:

printf vii

```
$ printf "\x41\n"
A
$ printf "\101\n"
A
```

Outline

Functions i

- As with most programming languages, you can define functions in bash.
- The syntax is:

```
function function_name {
commands...
}
```

- or

```
function_name () {
commands...
}
```

- Functions can accept arguments in a similar way as the script receives arguments from the command-line:

Functions ii

```
#!/bin/bash
# file: myscript.sh
zip_contents() {
 echo "Contents of $1: "
  unzip -l $1
}
zip_contents $1
```

- Note that we use the first positional parameter received in the command-line as the first argument for the function.
- The other special variables have also a meaning related to the function.
- E.g. $? returns the exit status of the last command executed in the function or the value after `return` or `exit`.

Functions iii

- The difference between `return` and `exit` is that the former does not finish the execution of the script, while the latter exits the script.
- A function may be compacted into a single line:

```
$ zip_contents () { echo "Contents of $1: "; unzip -l $1; }
```

- A semicolon must follow each command (also the final command).

Introduction to Variables

- Unlike many other programming languages, by default bash does not segregate its variables by type.
- Essentially, bash variables are character strings.
- Depending on context, bash permits arithmetic operations and comparisons on variables.
- The determining factor is whether the value of a variable contains only digits or not.
- You can explicitly define a variable as numerical using the bash built-in `declare`:

```
declare -i VAR
```

- We will use the convention of defining the name of the variables in **capital letters** to distinguish them easily from commands and functions.
- Variables have one of the following scopes: **shell** variables, **local** variables, **environment** variables, **position** variables and **special** variables.

Shell Variables

- A shell variable, as its name states, is a variable defined for a shell (e.g. a bash).
- In general, when you define a variable in a script, the variable is defined for the shell that is executing the script.
- To execute scripts, **the bash in which the script is launched clones itself** and the variables defined in the script are valid **only in this cloned shell**.
- You can also define a shell variable in the shell that is attached to a terminal:

```
$ VAR=hello
$ echo $VAR
```

- However, the previous variable **will not available to child programs** like a cloned bash that executes a script.
- If you need that a certain variable **is available for a child program**, you have to make it an **environment variable** as described later.

Local Variables i

- Local variables are those variables used **inside a function**.
- In a bash script, when you create a variable inside a function, it is added to the script's namespace.
- This means that if we set a variable inside a function with the same name as a variable defined outside the function, we will override the value of the variable.
- Furthermore, a variable defined inside a function will exist after the execution of the function.
- Let's illustrate this issue with an example:

Local Variables ii

```
#!/bin/bash
VAR="Outside the function"
my_function(){
VAR="Inside the function" }

my_function
echo $VAR
```

- When you run this script the output is ”Inside the function“.
- If you really want to declare VAR as a local variable, whose scope is only its related function, you have to use the keyword `local`:

Local Variables iii

```
#!/bin/bash
VAR="Outside the function"

my_function(){
local VAR="Inside the function"
}

my_function
echo $VAR
```

- In this case, the output is ”Outside the function“.

Environment Variables i

- shell variables are not available for child processes like the bash that executes a script.
- Child processes only inherit the **"exported context"**. Variables (and functions) are not exported by default.
- To export variables and functions and view the exported and current environment you can use the following commands:
  - **Export a variable**. Syntax: `export VAR` or `declare -x VAR`.
  - **Export a function**. Syntax: `export -f function_name`.
  - **View current context**. Syntax: `declare`.
  - **View exported context**. Syntax: `export` or `declare -x`.
- For example, let's use a script to test environment variables:

```
#!/bin/bash
echo $VAR
```

Environment Variables ii

- Then, execute the following:

```
$ VAR=hello ; echo $VAR
hello
$ declare | grep VAR
VAR=hello
$ ./script.sh
$ export VAR ; export | grep VAR
MY_VAR=hello
$ ./script.sh
hello
```

- Notice that `script.sh` can use the variable 'VAR' only after it is exported.
- As mentioned, exported variables and functions get passed on to child processes, but not-exported variables do not.

Environment Variables iii

- Anyway, if VAR is defined inside the script, the value of the shell variable is the one used in that script.
- You can delete (unset) an environment variable (exported or not) using the command `unset`:

```
$ unset MY_VAR
$ ./env-script.sh
```

Initial Environment

- When a bash is executed, **a set of environment variables is loaded**.
- The two main files in which the system administrator or the user can set the initial environment variables for bash are:
  - ~/.bashrc (per user configuration).
  - /etc/profile (system-wide configuration).
- Typical environment variables widely used are:

Variable	Meaning
SHELL=/bin/bash	The shell that you are using.
HOME=/home/student	Your home directory.
LOGNAME=student	Your login name.
OSTYPE=linux-gnu	Your operating system.
PATH=/usr/bin:/sbin:/bin	Directories where bash will try to locate executables.
PWD=/home/student/documents	Your current directory.
USER=student	Your current username.
PPID=45678	Your parent PID.

`source` Command i

- As mentioned, when you execute a script, bash creates a child bash to execute the commands of the script.
- This is the default behavior because executing the scripts in this way, the parent bash is not affected by erroneously programmed scripts or by any other problem that might affect the execution of the script.
- In addition, the PID of the child bash can be used as the "PID of the script" to kill it, stop it, etc. without affecting the parent bash (which the user might have used to execute other scripts).
- While this default behavior is convenient most of the times, there are some situations in which is not appropriate.
- For this purpose, there exists a shell built-in called `source`.

`source` Command ii

- The `source` built-in allows executing a script without using any intermediate child bash.
- `source` indicates that the current bash must directly execute the commands of the script:

```
#!/bin/bash
echo PID of our parent process $PPID
echo PID of our process $$
echo Showing the process tree:
pstree $PPID
```

- If you execute the script without `source`:

source Command iii

```
$ echo $$
2119
$ ./script.sh
PID of our parent process 2119
PID of our process 2225
Showing the process tree:
bash---bash---pstree
```

- Now, if you execute the script with `source`:

```
$ source ./script.sh
PID of our parent process 2114
PID of our process 2119
Showing the process tree from PID=2114:
gnome-terminal---bash---pstree
```

`source` Command iv

- As you observe, when script.sh is executed with `source`, the bash does not create a child bash but executes the commands itself.
- An alternative syntax for `source` is a single dot.
- So the two following command-lines are equivalent:

```
$ source ./pids.sh
$ . ./pids.sh
```

- “Sourcing“ is a way of including variables and functions in a script from the file of another script.
- This is a way of creating an environment but without having to ”export“ every variable or function.
- We can create scripts that serve as a “library” of functions and variables.

Position Variables

- Position parameters are special variables that contain the arguments with which a script or a function is invoked.
- These parameters have already been introduced.
- A useful command to manipulate position parameters is `shift`:
  - It is a built-in and takes one argument, a number.
  - The positional parameters are shifted to the left by this number N.
  - The positional parameters from N+1 are renamed.

Special Variables

- Special variables indicate the status of a process.
- They are treated and modified directly by the shell so they are read-only variables.
- We have already used most of them.
- For example, the variable $$ contains the PID of the running process.

Variable	Meaning
$$	Process PID.
$*	String with all the parameters received.
$@	Same as above but treats each parameter as a different word.
$#	Number of parameters.
$?	Exit status (or return code) of last command (0=normal, >0=error).
$!	PID of the last process executed in background.
$_	Value of last argument of the command previously executed.