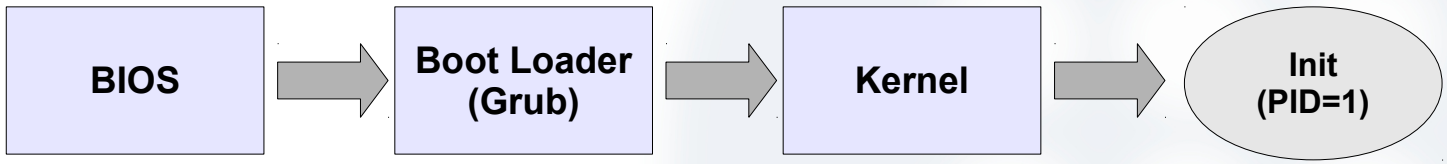
Processes

Jose L. Muñoz, Oscar Esparza, Juanjo Alins, Jorge Mata
Telematics Engineering
Universitat Politècnica de Catalunya (UPC)

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

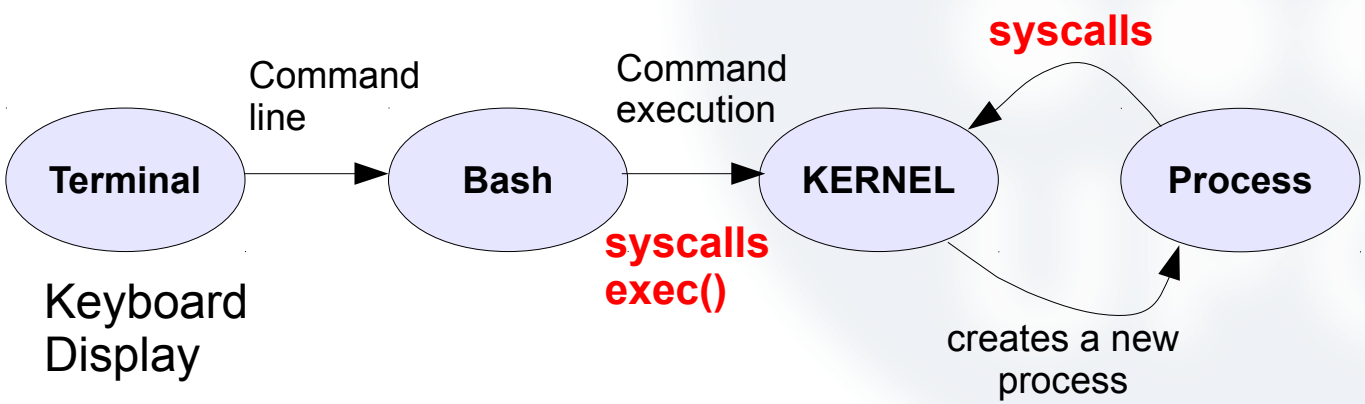
- A process is the abstraction used by the operating system to **represent a running program**.
- Each process in Linux consists of:
  - An address space.
  - A set of data structures within the kernel.
- The address space contains the **code** and **libraries** that the process is executing, the process's **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.

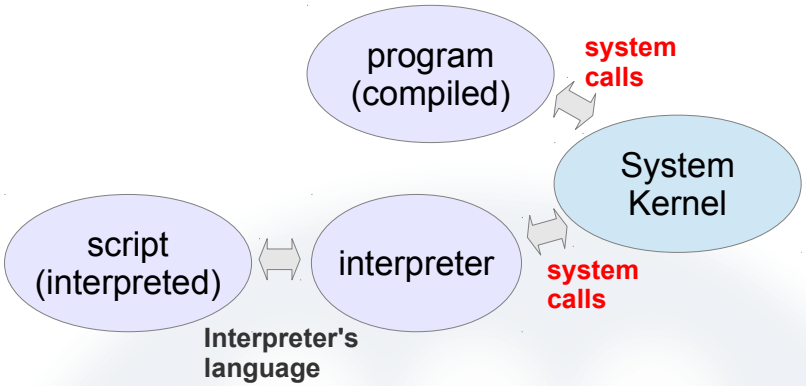
Other commands related to processes

- **`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”:

```
1 #!/bin/bash
2 # Our second script, Hello world!
3 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.

```
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
```

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**.

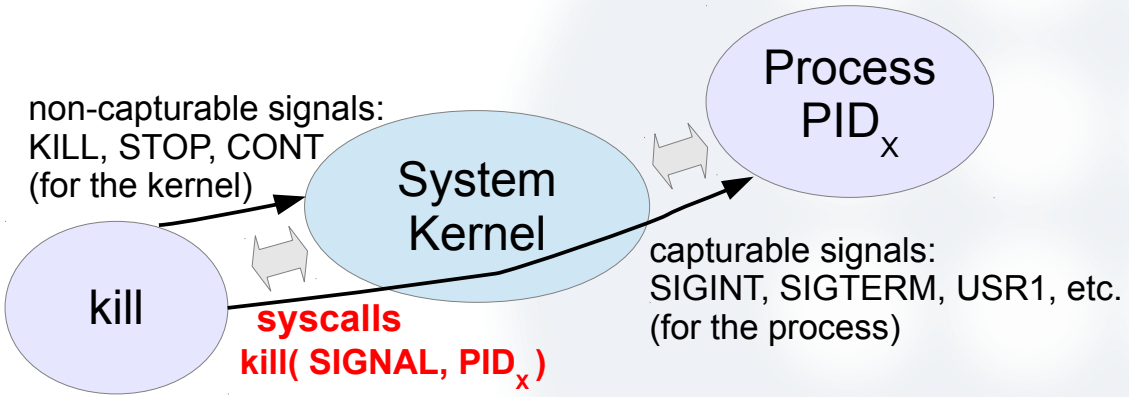
```
$ xeyes &
```

- 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).