II.2407
System programming in C language

Goals of this course

1. Introduction
   1. Vocabulary
   2. operating system architecture
2. The shell
3. Processes and signalling
4. Inter Process Communication (IPC)
   1. pipes and message queues
   2. Shared memory and semaphores
5. Threads
6. Sockets
7. ioctl

System

- A system is a group of components interacting between them according to some rules.
- A system is defined with :
  - the kind of components
  - the interaction between components
  - its border : what is in or out of the system, why a component belongs to the system or to its environment
- The behaviour of a system should always be the same when interacting with its environment

From system to information system

- Real time system:
  - asynchronous
  - synchronous
- Embedded system
- Computer system
  - Operating system
- Information system

Real-time system

- a system is a real-time system if it is able to perform tasks, compute data, without errors, at the same pace as the data flow.
- a real-time system is not faster than another system because the data flow may be slow.
- a real-time system just certify that data will be computed on time (not too later, but not faster)
- with a real-time system, the behaviour is not only predictable but the time to get the result may be estimated or forecast:
  - synchronous system: the time to perform a task may be accurately computed
  - asynchronous system: the maximum time to perform a task may be accurately computed

Embedded system

- an embedded system is a system integrated within another machine:
  - car, train, plane
  - oven, fridge, washing machine, dish washer, TV set
  - mobile phone
  - MP3 player
  - network or communication devices (switches, routers, satellites, PABX, ...)
  - robots, process controllers, ...

Embedded system

- An embedded system is often a dedicated system (versus universal system like a PC)
- An embedded system just manages a limited and determined number of features, this is the reason why I did not include smartphones and tablets in my list of embedded systems.
- An embedded system may be configured at install time or at runtime.

Computer system

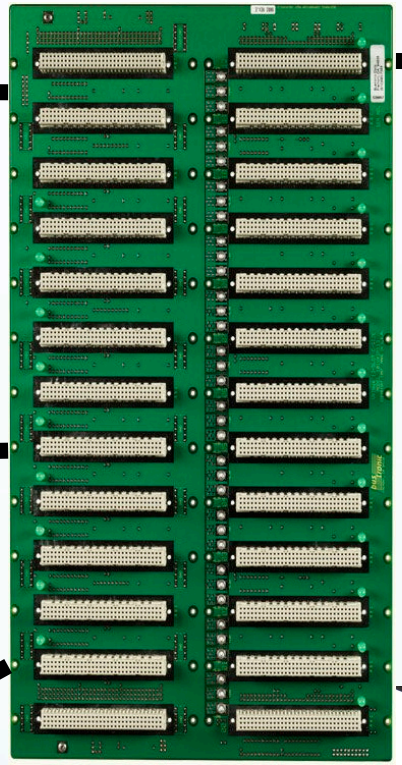
- The purpose of a computer is to compute data.
- A computer is made of hardware and software.
- A computer is universal and may perform lots of different tasks, including tasks not forecast at design time of installation time.
- A computer may be programmed, tuned, configured and evolve.

Hardware

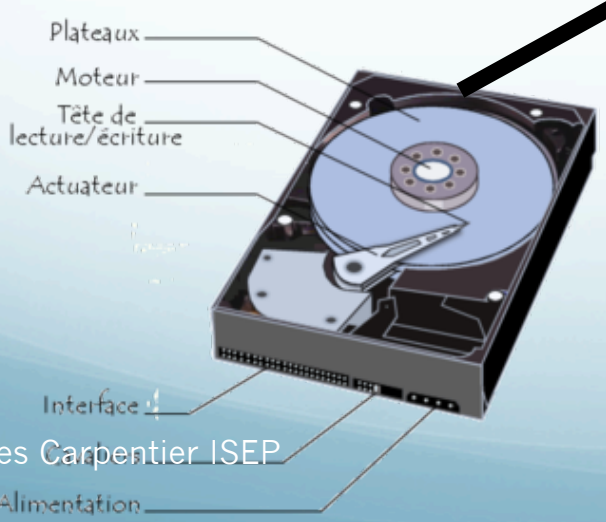
- A computer hardware is made of:
  - a Central Process Unit (CPU) => microprocessors
  - a central RAM memory to store temporary data
  - internal or external devices (screen, keyboard, mouse, hard disks, network interfaces, ...)
  - a communication bus

Microprocessor

bus

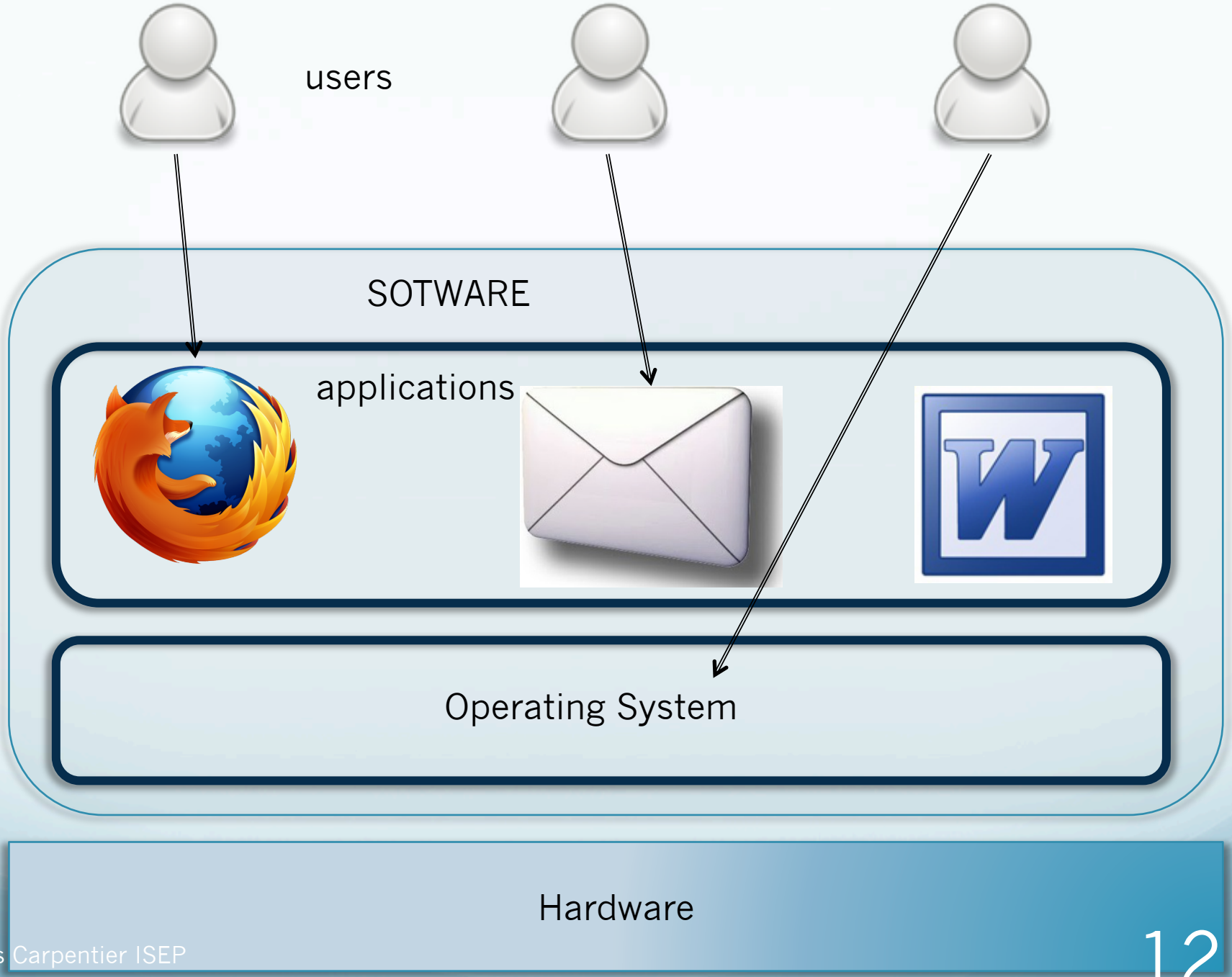


RAM



Software

- The software of a computer is organized in 2 layers:
  - the operating system
  - applications
- Applications are directly operated by users : office, web browser, mail client, ...)
- The operating system provides
  - common services for all applications
  - transparent access to the hardware: the operating system hides the differences between different hardware devices => an application shall be able to run on computers with different devices (different memory size, different kind of mass storage, screen, keyboard, ...) provided they use the same OS. But speed may be different from one computer to the other.


users
SOTWARE
applications
Operating System
Hardware

Information system

- The information system of a company or organization is made of all employees and systems of this organization that manages data production, retrieving, processing and storing inside the organization and also interact with its environment (ecosystem) like vendors, customers, administration ...
- The information system is highly dependant of the organization and activities of the company.
- A company has to adapt to environment change, thus the architecture of the information system too .
- The architecture of the information system should also adapt to technology improvements.

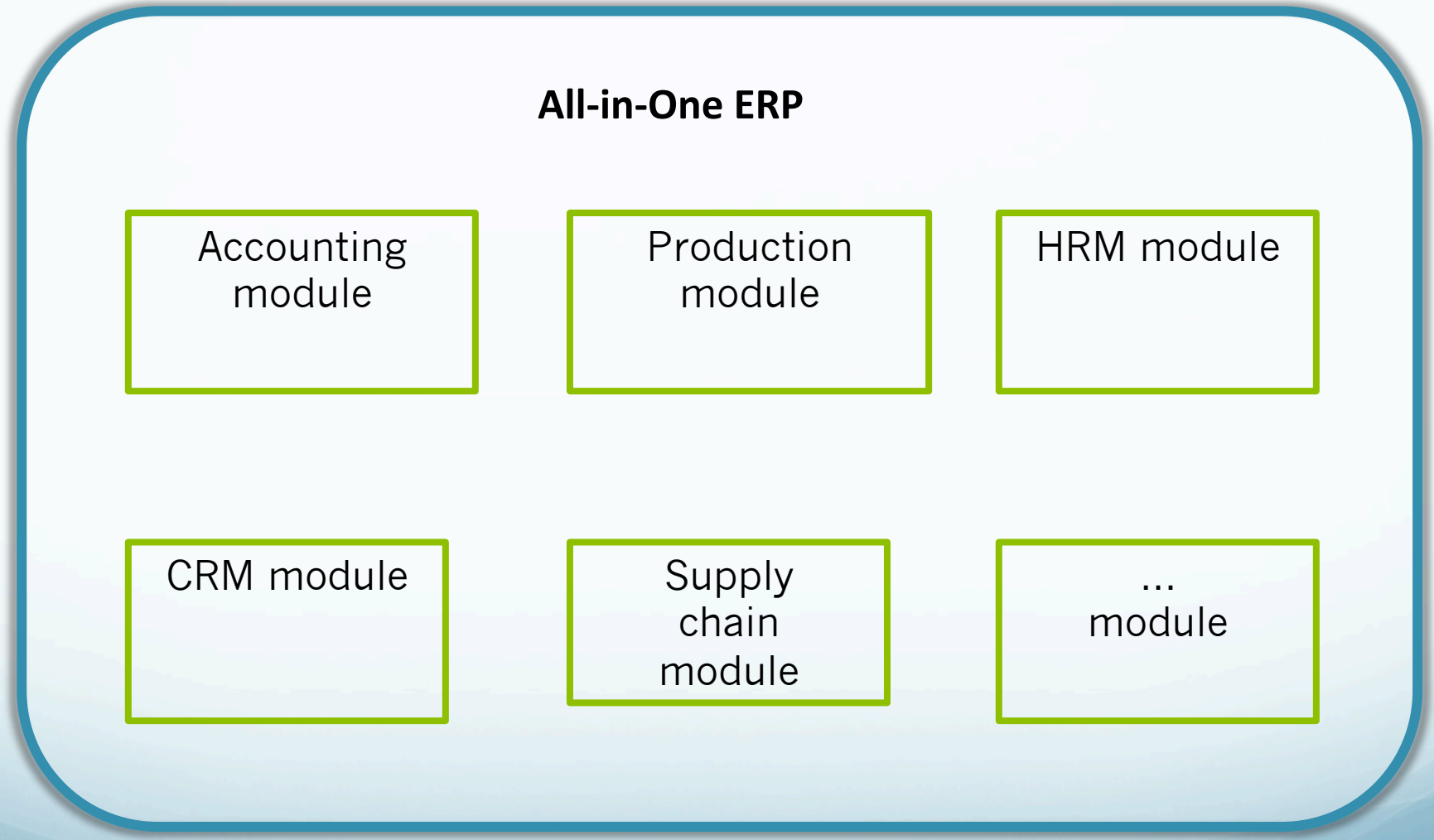
Information system

- any organization has common functions like:
  - accounting
  - human resource management, payroll, ...
  - billing
  - communication
- some functions are specific to their activities:
  - production
  - supply chain, ...
- The information system should provide generic and specific software:
  - generic software is often purchased or open-source software
  - specific software is developed internally or externally (IT company) for the company

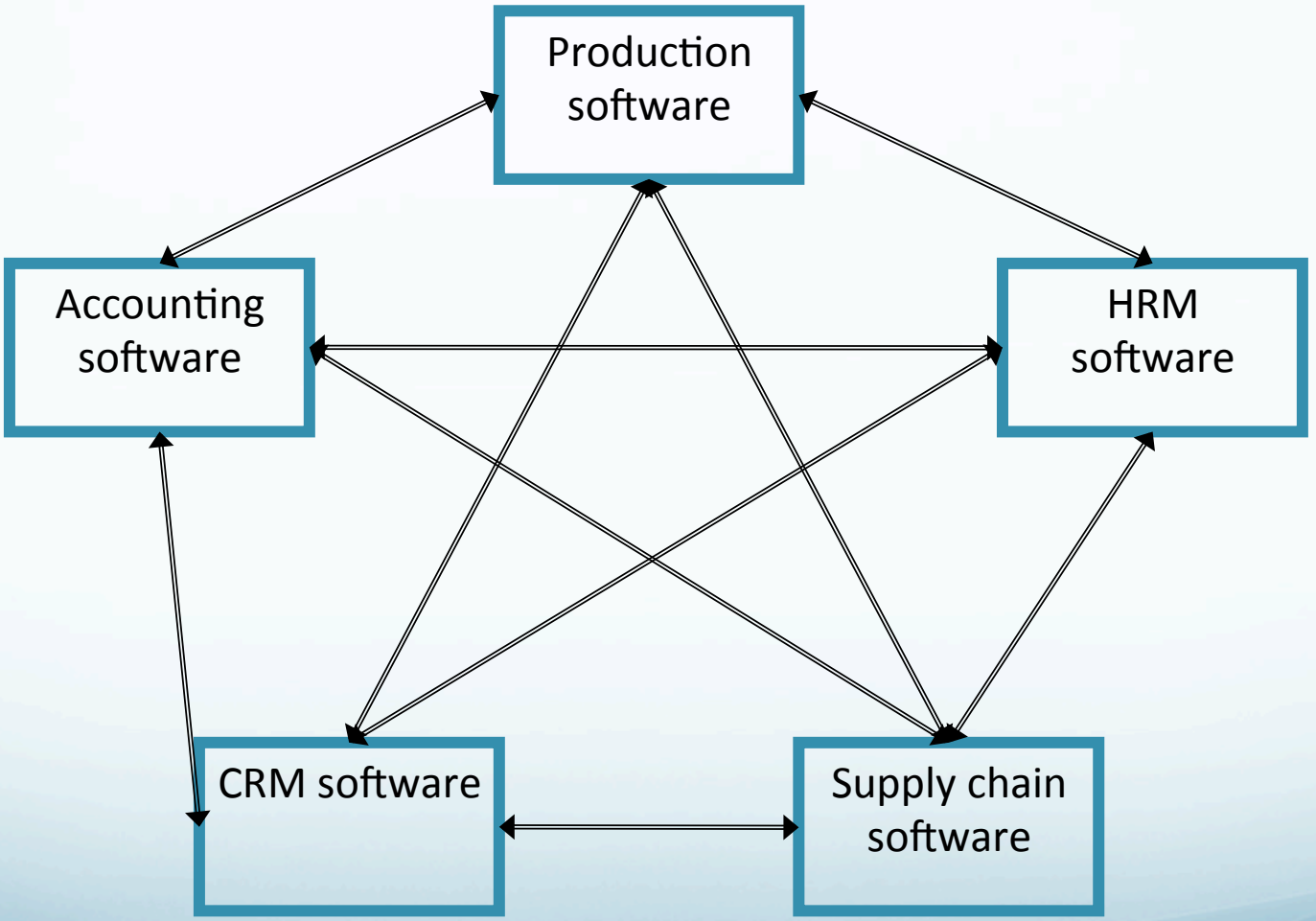
ERP

- ERP (Enterprise Resource Planner) is the core part, the generic part of the Information System.
- It may be:
  - an All-In-One software that provides a maximum of features in one single software
  - or a software for each feature and communication interfaces to exchange data between them (system integration)

All-In-One ERP



System integration



Operating System

- an operating system is the base software of a machine. It provides the common and mandatory functions in order to operate specific software (applications).
- An operating system may be:
  - mono-task: one application at a time
  - multi-task: multiple applications at the same time
    - using time-sharing: each application will use the CPU (particularly when there is one processor) turn by turn for a limited time (time slot)
    - using parallel CPU (multiple processors or multi core processors)

Operating System

- An operation system may be:
  - mono-user: only one single user, no user management
  - multi-user, different users may use the system:
    - alternatively
    - simultaneously (multi-session) through the I/O of the system:
      - serial and USB ports using a terminal
      - network (virtual terminal)

ExoKernels

- when there is only one application to run on top of an operating system, it is possible to embed the operating system inside the application
- **Example:**
  LinuxRT is embedded inside the application that manages all the microcontrollers of Peugeot cars
- Sometimes, only some features of the operating system are directly embedded into the application, like the memory management in Firefox to provide an algorithm that better manages web pages.

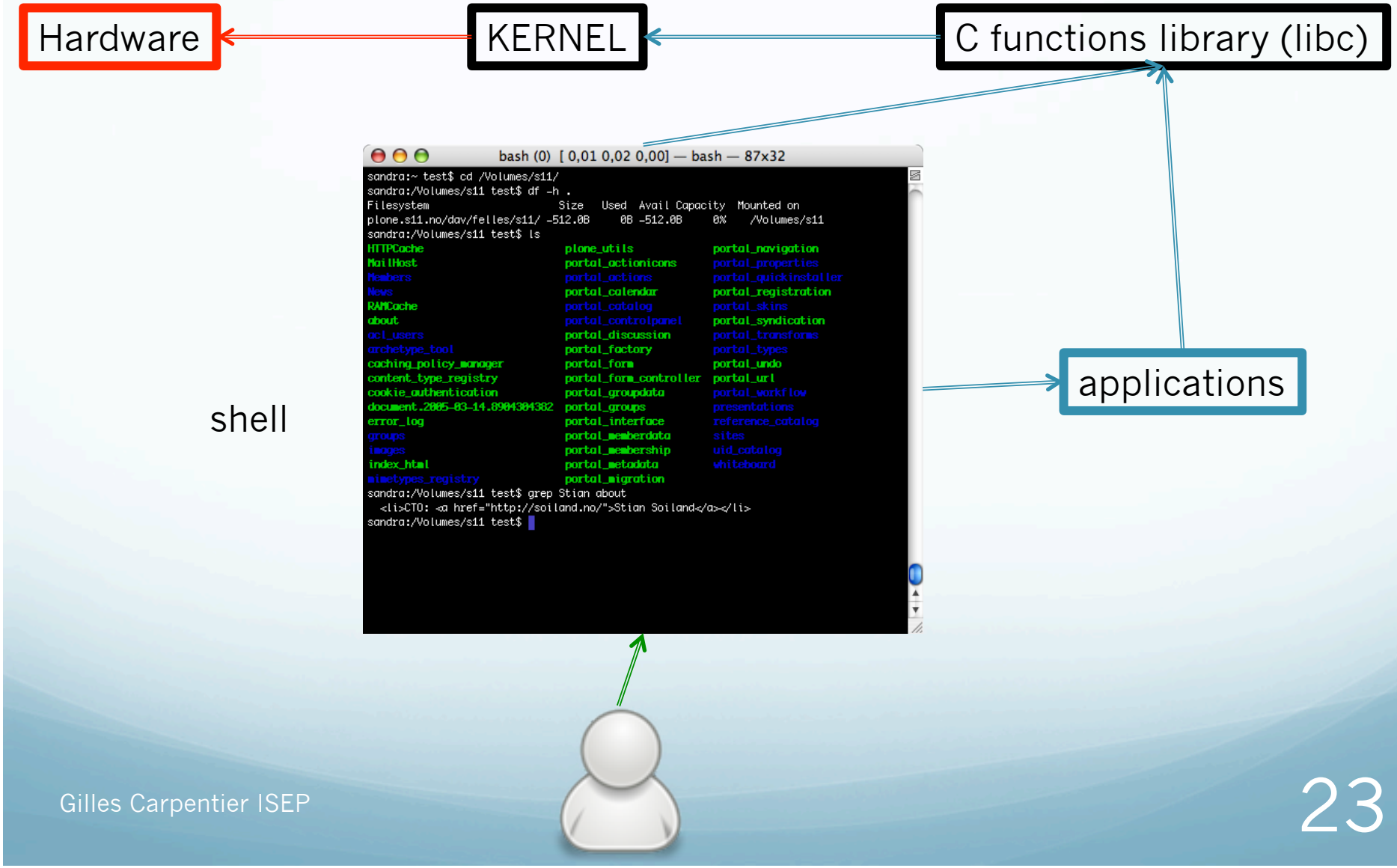
Some operating systems

- Universal OS used on desktop, portable or server computers : Windows, Linux(Unix)
- OS dedicated to servers: Novell Netware, IBM zOS (mini computers) or VM/CMS (mainframes)
- embedded systems for mass market:
  - iOS, Android, Windows, Linux, ChromeOS, FirefoxOS, WebOS
- Real-Time OS: VxWorks, QNX, VRTX, WindowsCE, RTLinux

Components of an operating systems (Unix)

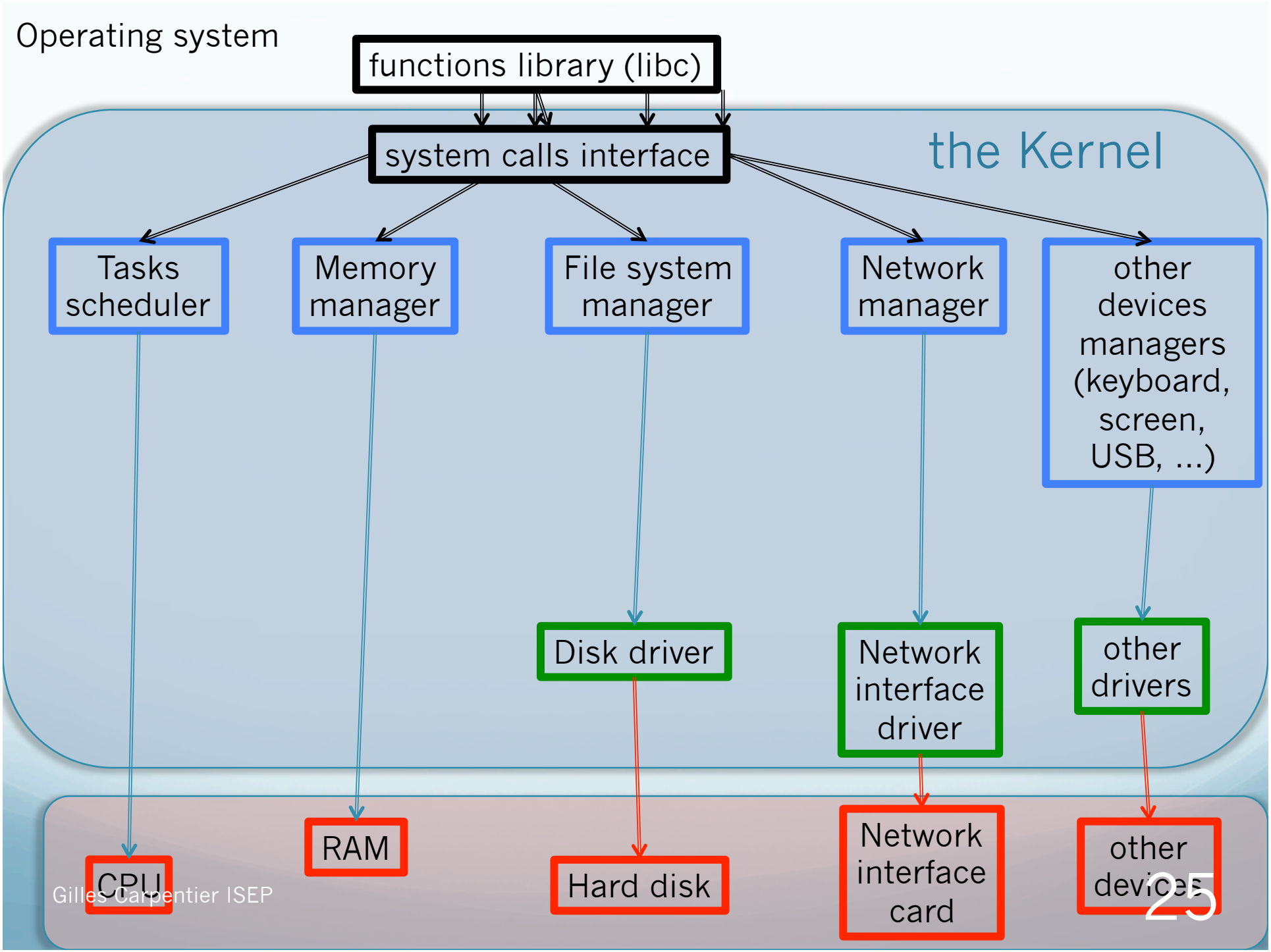
- A simplified view of an operating system is divided into 3 parts:
  1. The kernel
  2. the library of functions (for example libc for Unix)
  3. the command interpreter (the shell)
- The kernel interfaces with the hardware to avoid applications to care about hardware details.
- Applications just have to use the library of functions to access the hardware (memory allocation, networking, ...)
- The shell is the user interface and is used to:
  - configure the OS
  - launch applications
- Some OS like Windows provide a graphic user interface along with a shell.

Components of an OS



the Kernel

- The kernel itself is also made of components:
  - the task scheduler
  - the memory management
  - the storage management
  - the I/O devices management


Operating system
functions library (libc)
system calls interface
the Kernel
Tasks scheduler
Memory manager
File system manager
Network manager
other devices managers (keyboard, screen, USB, ...)
Disk driver
Network interface driver
other drivers
CPU
RAM
Hard disk
Network interface card
other devices

the tasks scheduler

- For multi-tasks OS, access to CPU (whatever the number of processors and cores) is under the control of the tasks scheduler (or process manager).
- The tasks (jobs) are put into one or more queues.
- The number of queues depends of the ability of the OS to manage:
  - priorities
  - multi-core processors or multiple processors

task recycling

- when a task uses the CPU, this task has to release the CPU when:
  - the task has ended and thus does not need the CPU anymore
  - the task is waiting on another resource (disk, network, any I/O) => the scheduler will put this task at the end of the queue (the task is recycled)
  - the task reached is time quota (time slice) the task will also by recycled
  - a new task is launched with an higher priority (pre-emption)
- Time slices and pre-emption are not implemented in every OS.

Memory management

- Applications require RAM to store code and data.
- The memory manager provides blocks of memory (pages) to different tasks that need them.
- The memory manager prohibits tasks to access to the memory allocated to other tasks (protected mode)
- When a task requires more memory than the available RAM, the memory manager uses a part of an hard disk (a dedicated disk volume or just a regular file) to extend the RAM size limit. This space is called the swap or virtual memory.
- When a task ends, the memory manager releases all blocks allocated to this task.

The top command

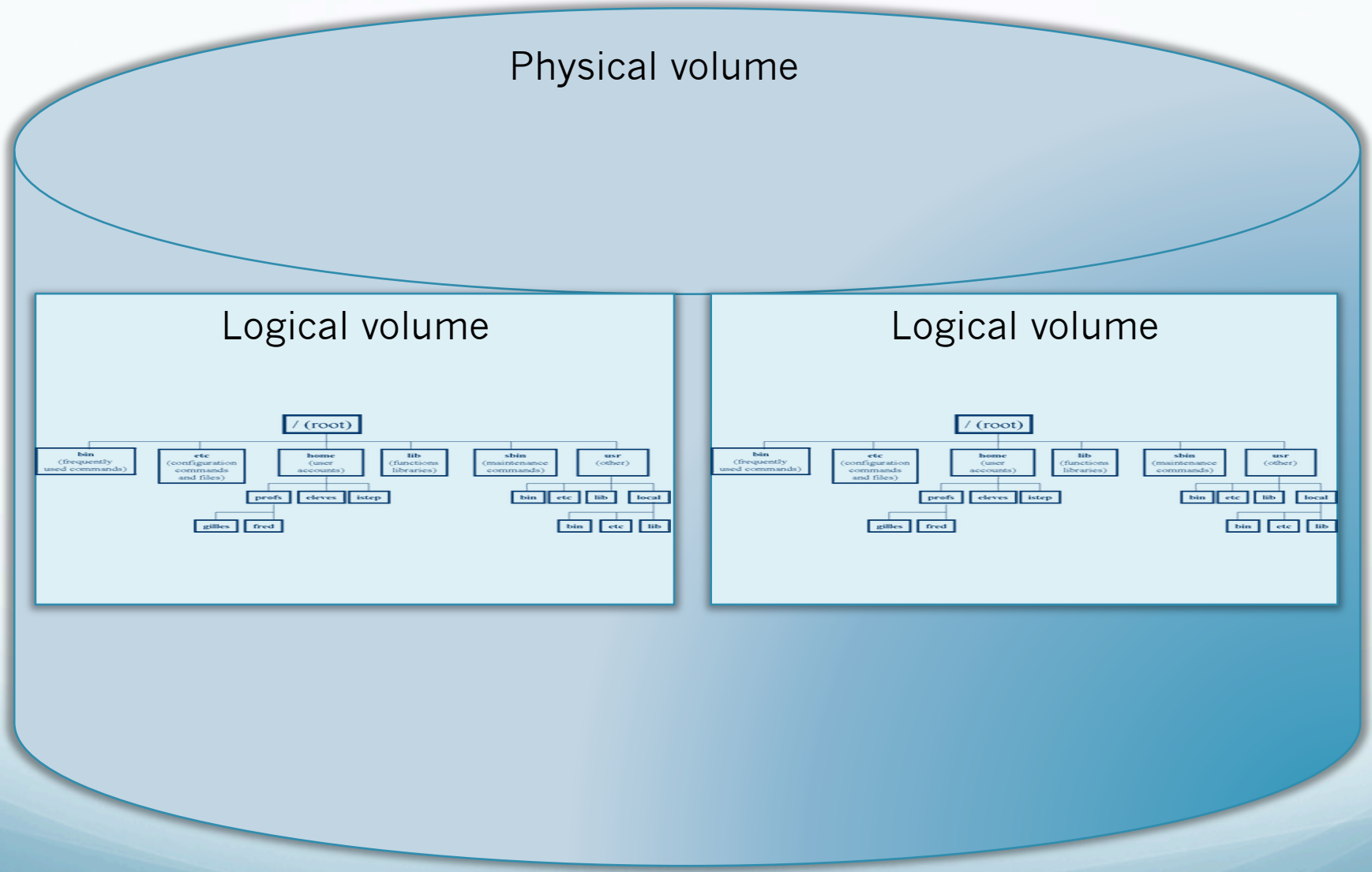
- the top command displays the list of processes and their memory and CPU consumption.
- The header of the list displays the total memory and CPU usage.

```
top - 15:45:08 up 18 min,  2 users,  load average: 0,10, 0,11, 0,13
Tasks: 273 total,   4 running, 269 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,7 us, 27,9 sy,  0,0 ni, 70,8 id,  0,3 wa,  0,3 hi,  0,0 si,  0,0 st
KiB Mem:   2042700 total,   774876 used,  1267824 free,    46328 buffers
KiB Swap:  2094076 total,        0 used,  2094076 free.   413936 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  197 root      20   0       0      0      0 R  0,3  0,0   0:00.87 kworker/0:2
 1149 root      20   0  250556  52648  11192 S  0,3  2,6   0:07.32 Xorg
 1340 root      20   0  165488   4648   3744 R  0,3  0,2   0:01.77 vmtoolsd
 3475 root      20   0   36812    760    492 S  0,3  0,0   0:00.04 tpvmlp
    1 root      20   0   33756   3088   1456 S  0,0  0,2   0:01.60 init
    2 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0,0  0,0   0:00.14 ksoftirqd/0
    4 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kworker/0:0
    5 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S  0,0  0,0   0:00.15 kworker/u128:0
    7 root      20   0       0      0      0 S  0,0  0,0   0:00.21 rcu_sched
    8 root      20   0       0      0      0 R  0,0  0,0   0:00.50 rcuos/0
```

Storage management

- Programs and data are stored in file systems.
- A file system is the method of organization for the files
  - for an OS, you may have different kind of file systems at the same time.
  - for example, the file system of a CD is different of the file system for an hard disk.
- The most frequent organization for a file system is a tree that contains files and folders that also contain files and sub-folders.
- A file system may use the full available space of an hard disk (physical volume).
- You may divide an hard disk into multiple logical volumes. A logical volume may be extended on multiple physical volumes.

Storage management



Input/output management

- For each type and even model of physical interface, you need to include a device driver into the kernel:
  - keyboard, mouse, ...
  - screen
  - audio card
  - serial ports, USB, parallel, Firewire, Bluetooth, Thunderbolt, ...
  - network, including WIFI
  - printer, even for remote printers
  - ...
- For the most popular devices (hard disks, CD players, mouse, keyboard, ...), the driver is already included in the kernel. Otherwise, you have to install it manually from a vendor CD or download site.

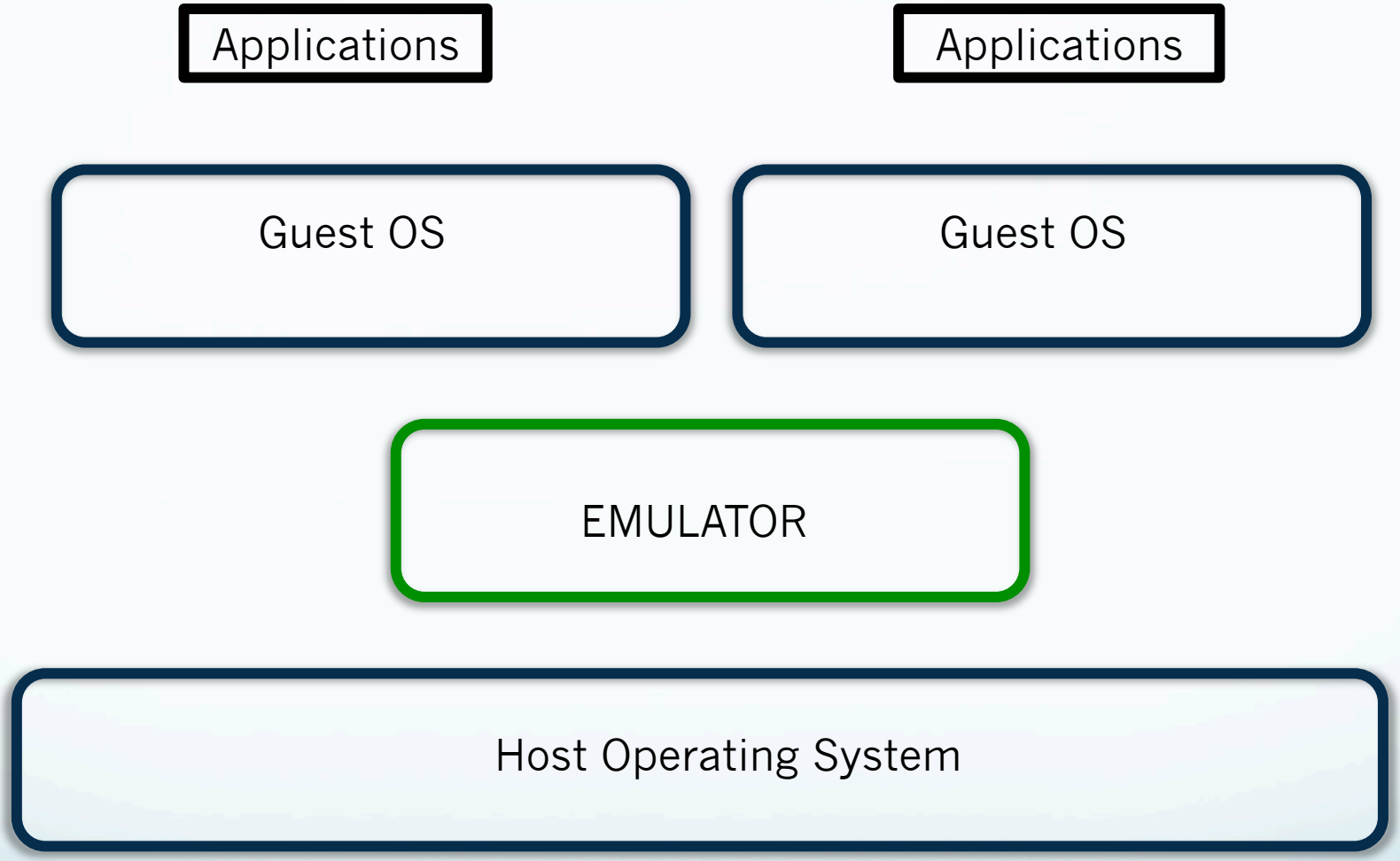
Device virtualization

- We already know that we may virtualize RAM memory using a swap area (virtual memory).
- Virtualization means "using something as if we really had it" but we do not really have at at least locally.
- **Example**: virtual disks are transparent remote disks (Network File System). They are located on a remote server, but we see them and use them as if they were local disks.

System virtualization

- The complete computer is virtualized, not just some devices.
- System virtualization is used:
  - when I want to run at the same time, on the same computer, 2 applications that require 2 different OS
  - when some applications cannot be hosted on the same computer (very often on servers), I need to have one computer for each application => waste of hardware
- System virtualization is achieved using a software called an emulator like DosBox, Sun/Oracle VirtualBox, Vmware player for workstations (desktops and laptops)
- For companies, most servers are virtualized to reduce costs (investment, monitoring, maintenance, energy, space, ...) => Data Centers

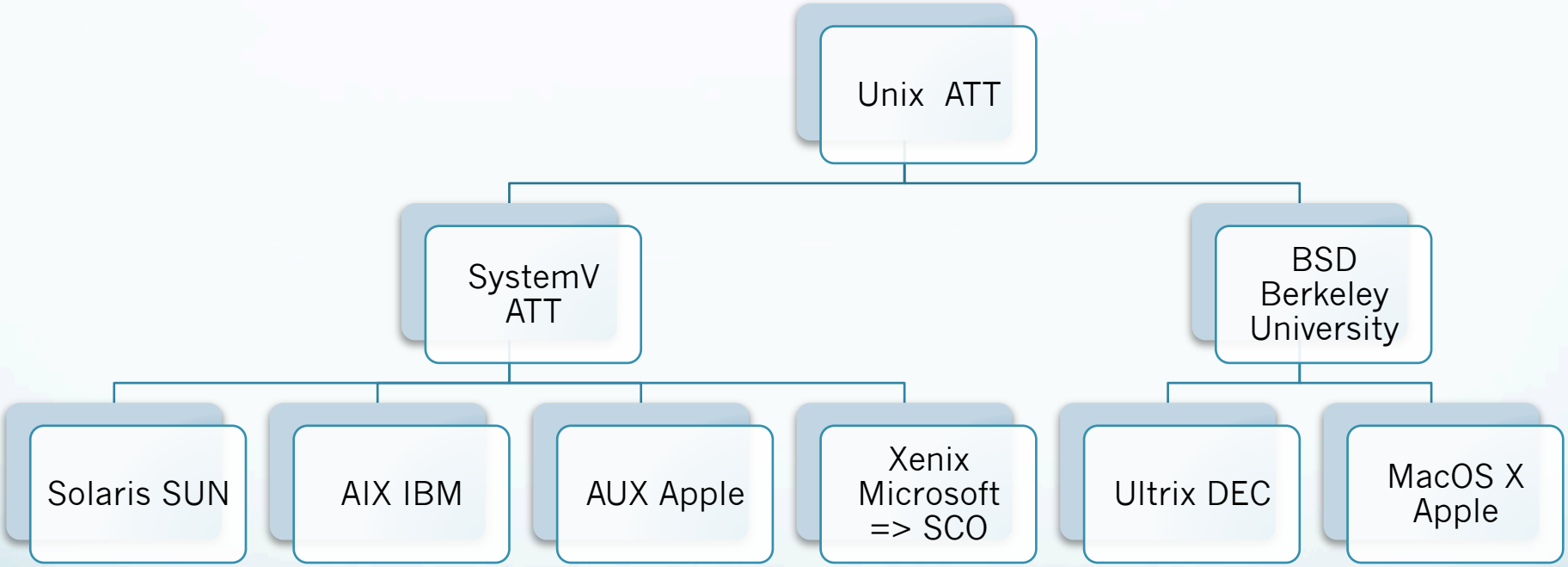
System Virtualization



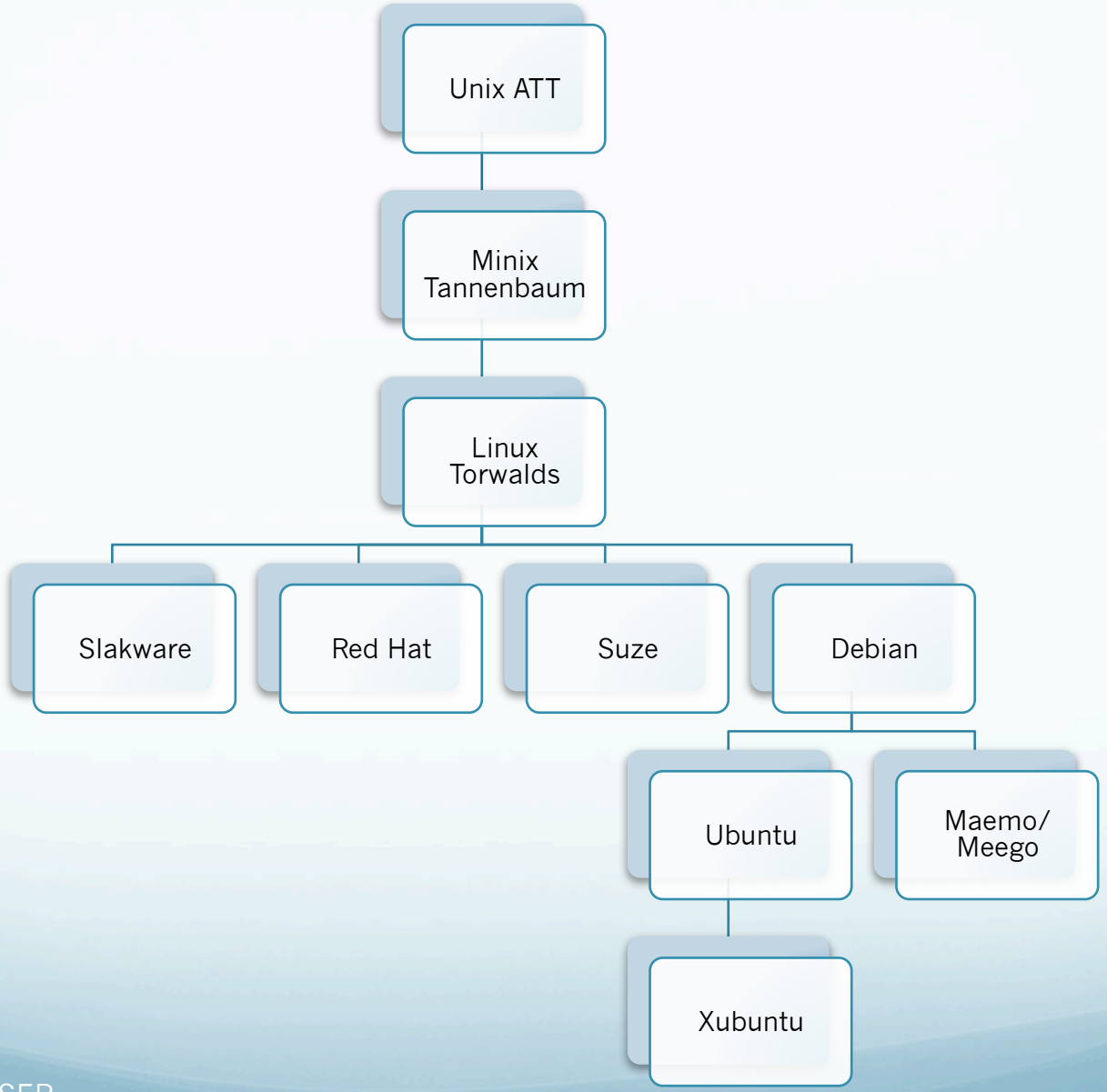
Unix

- Simple and modular, Unix is able to run even on very limited hardware devices.
- Unix is able to work on any hardware architecture :
  - Intel
  - Motorola/IBM PowerPC
  - Sun Sparc
  - ARM
  - NEC, ...
- But Windows too :
  - Intel
  - DEC alpha
  - IBM S/400, ...
- But Windows, unlike Unix, has only be popular on classic Intel processors.
- Unix is open-source enabling manufacturers to adapt it to their particular purpose. Under Unix, you really know how it runs and when you are system administrator, what you learned on one release of Unix, is still true for the next release.

UNIX flavours



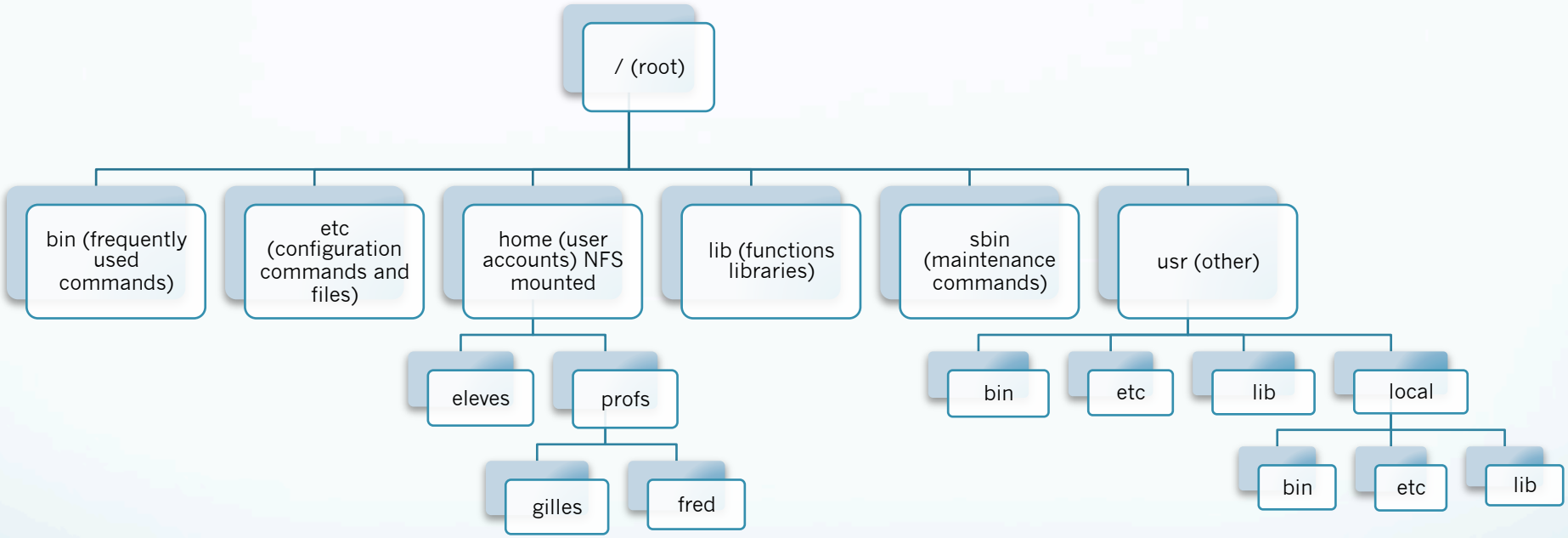
from UNIX to Linux



The shell

1. Using Linux : the file system, the shells, the GUI
2. Users management
3. Useful commands
4. File system commands
5. Files and redirections
6. File edition
7. Archiving
8. Processes
9. Networking

The file system



The path

- The symbol / is used
  - to represent the root of the file system
  - and as separator between the names of folders
- The access path to a file or folder may be written :
  - as an absolute path from the file system's root
    Example : /home/profs/gilles
  - or as a relative path from the current directory
    . (dot) means the current directory
    .. (dot dot) means the parent directory
    Example : suppose that the current directory is /home/profs/gilles
    and I want to access the directory of another teacher (fred), I can write
    :../fred

The shells

- A Unix shell is the program which reads user input from the command line and executes actions based upon that input.
- Unlike Windows, there are many shells under Unix :
  - *sh* the Bourne shell (historically the first shell)
  - *csh* the C shell : C-like syntax
  - *tcsh* is an enhanced, but completely compatible version of the Berkeley UNIX C shell
  - *ksh* the korn shell
  - *bash* the Bourne Again shell

The Graphic User Interface

- The standard GUI for Unix is XWindow/11 (X11) designed and developed by the MIT.
- Unlike Windows, it is not part of the operating system kernel, but is a client-server application and thus manages network terminals.
- The X11 server manages the content of windows, but not the windows:
  - size
  - position
  - foreground/background
  - iconification
- The windows are managed by window managers like :
  - CDE/Motif
  - KDE
  - Gnome
  - Xfence, ...
- You can switch from one window manager to another one, but there is only one window manager managing a server at the same time.
- Graphic applications are X11 clients.
- Under Linux, you may switch from text terminals (Ctrl-Alt-F1 to Ctrl-Alt-F6) to X11 (Ctrl-Alt-F7)

Login

- Unix is a multi-user system, multi-session system.
- Many users may be logged and use a Unix system at the same time.
- Before using Unix, a user should be authenticated.
- Unix prompts the user with a login and a password.
- The login name should be a lowercase string.
- Depending on the version of Unix, the password length may vary.
- Some Unix systems require that the password contains at least one or two special characters (digits, ...).

User identification

- A file, a directory, a process, belong to users.
- Each user is identified by a unique number, the User IDentification (UID).
- For a personal workstation and for each user
  - the UID
  - the login name
  - the home directory
  - the login shell

  are stored locally in the file **/etc/passwd**.
- For companies and here at ISEP, users are managed in a directory located on a server. Workstations authenticate users while sending a LDAP request to the server.
- LDAP servers implementations are :
  - OpenLDAP
  - Microsoft Active Directory, ...
- Users belong to one or more groups.

The login sequence

- Once the user is authenticated, Unix launches a first set of default commands that is stored in /etc and the a second optional set that is stored in the user home directory.
- The names for these files depend on the kind of login shell that is stored in /etc/passwd or the LDAP server for this user.
- Then the user is allowed to access the shell and his current directory is his home directory.

Exit and logout

- The logout command logs the user out of the system if the user runs this command from his login shell.
- The exit command just closes the current shell and if this shell is the login shell, also logs out the user.

List of logged users

- The **who** command displays the list of current logged users and from which terminal or network station.
  Example :

  who
  gilles pts/0 Aug 25 14:53 (nemesis.isep.fr)

- The **w** command displays users activity :

  w
  14:55:50 up 59 min, 1 user, load average: 0.00, 0.00, 0.00
  USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
  gilles pts/0 nemesis.isep.fr 14:53 0.00s 0.03s 0.02s w

Commands history

- Some shells manage an history of commands.
- The **history** command displays the list of previous launched commands.
  Example :

  history

  498 ls
  499 who
  500 w
  501 cls
  502 clr
  503 clear
  504 w
  505 clear
  506 w
  507 clear
  508 history

- To launch a previous command, either :
  - type **!** followed by the number of the command in the history list
  - type **!** followed by the beginning of the name of the command
  - **!!** launches the last command again
- You may also use the up and down arrows to browse the commands.

The manual

- Unix provides an on-line help called the manual.
- The manual contains documentation for:
  - user commands
  - functions and system calls for the C language and other installed languages
  - configuration files
- The manual is organized in sections.
- You also search through the manual using the -k argument for the man command:
  man -k user will display every page that contains the word *user*.

Environment variables

- Environment variables are global variables : you may read the environment variables of the shell from any script or program.
- They generally store information about the configuration of the system and the user session.
- But you may create your own environment variables.
  They are generally named using uppercase letters.

  Examples :
  DISPLAY
  USER
  PATH

Display of environment variables : printenv

- The **printenv** command displays the list of environment variable with their values.

Example :

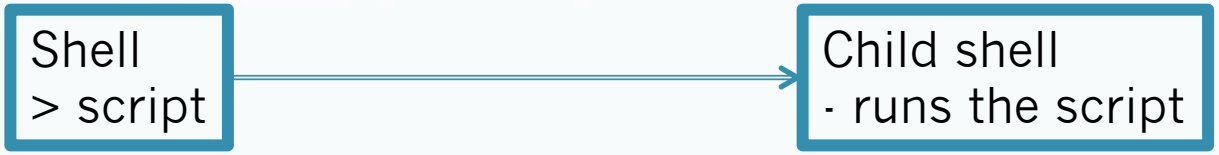
```
printenv
TERM=vt100
HOME=/home/gilles
SHELL=/bin/csh
USER=gilles
PATH=/local_user/unix/ioffice/bin:/usr/local/bin:/etc:/
usr/etc:/usr/ucb:/bin:/usr/bin:/usr/bin/X11:/usr/
sony/bin:/usr/sony/bin/X11:/usr/new:/usr/new/mh:/
usr/pds/bin:/usr/hosts:/usr/games:.
LOGNAME=gilles
MANPATH=/usr/local/man:/usr/man
IOFFICE=/local_user/unix/ioffice
```

Setting environment variables

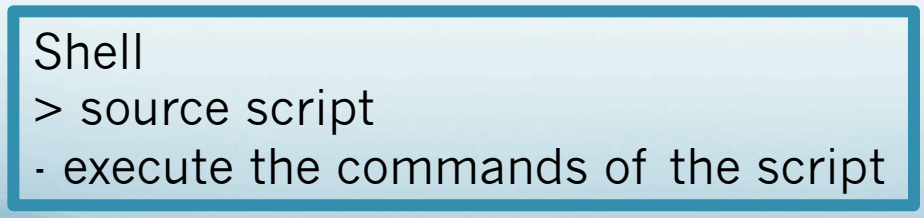
- You may access the content of a variable (including environment variables) using the **$** character.
- **Example:** echo $USER
- setting the value of an environment variable:
  - using csh: **setenv** variable_name value
    - **Example**: setenv TOTO toto
  - using bash: **export** variable_name**=**value
    - **Example**: export TOTO=toto

The source command

- Environment variables changes that are set from within a shell, are visible from any child process of your script, but not from the parent process, except if you run the script using the **source** command.



- The source command runs the script in the same shell than the parent shell.
  source name of the script



The path system variable

- **Predefined variables**
  - path, home, term, user, shell, status
  - They have lowercase names in *csh*, but displayed with uppercases by **printenv**.
- **The PATH variable**
  - The environment variable PATH contains the list of directories in which the shell searches for files matching the command that is launched.
  - When the shell notifies « command not found », it just means that the command you entered has not been found in any directory listed in the PATH variable. But the command may be located in another directory somewhere else in the file system.
  - If a command exists in more than one directory listed in the PATH (for example the **ps** command), the command found in the first directory in the order of the PATH list will be launched.

which

- The **which** command displays the name of the directory where the command you search is located if this directory is listed in the PATH variable.

  Example :
  which ls
  /bin/ls

Browsing the file system

- The **pwd** (Print Working Directory) displays the name of the current directory.
- The **cd** (Change Directory) command enables the user to jump to another directory :
  - **cd** without any parameter jumps to the user's home directory
  - **cd** followed by a absolute or relative path jumps to this directory
- **Examples :**
  cd /home/profs/gilles
  cd ../../eleves
- *tcsh* tries to complete the command if you press the tab key after typing the beginning of the name of the command.

Displaying the content of a directory

- The **ls** command (LiSt) displays the content of a directory (files, sub-directories, special files).
  ls [-arguments] [path]
- If the path is not provided, **ls** displays the content of the current directory.
- **ls** arguments :
  -a : displays every file, included hidden files (files which names begins with a dot)
  -l : displays details about files
  -g : displays the name of the group for which particular rights have been set
- Arguments may be combined : **ls –alg**
- The order of arguments is not important.

Example of directory listing

ls -alg

```
total 3
drwxrwxr-x 3 purna wheel 512 Mar 14 1998 .
drwxrwxrwx 23 gilles wheel 1024 Mar 13 20:50 ..
-rwxr-xr-x 1 purna wheel 421 Mar 13 20:57 .cshrc
-rwxr-xr-x 1 purna wheel 263 Mar 13 20:58 .login
drwxrwxr-x 2 purna wheel 512 Mar 14 1998 ada
```

. (the single dot) means the current directory
.. (double dot) means the parent directory
2 hidden files => .cshrc and .login
1 sub-directory => ada

Type of files

- The first character of each line displayed by the **ls** command, represents the type of file :
  - => (dash) a real file
  d => directory
  l => symbolic link (symlink)
  c => character device (a serial line for example)
  b => block device (an hard disk for example)
  p => named pipe
  ....

The file command

- The **file** command provides more details about the content of files :

Example :

```
base.sql: ISO-8859 text
bootimg: Netboot image, mode 2
carre: ELF 32-bit MSB executable, SPARC, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
dead.letter: ASCII mail text
Desktop: directory
GEF.log: empty
Mail: directory
MscSI: ISO-8859 English text
odp_j2ee.tgz: gzip compressed data, deflated, last modified: Mon
Jan 28 03:09:42 2002, os: Unix
recup_passwd_cisco.html: ASCII HTML document text, with CRLF, LF
line terminators
ssh: ELF 32-bit MSB executable, SPARC, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
system: ASCII C program text
```

Permissions

- Only 3 permissions are managed by Unix :
  - r for READ
  - w for WRITE : create, update, delete
  - x pour eXecute :
    - for a file, x means: this file is program and could be launched
    - for a directory, x means that this directory may be parsed
- These permissions are assigned to :
  - the file owner
  - a group
  - other users

Owner and group

- The third column displayed by the **ls** command, is the UID or login name of the user that owns the file or directory.
- The fourth column is the GID (Group ID) or the name of the group that has permissions on this file or directory.
- **Example :**

  drwxrwxr-x 2 purna wheel 512 Mar 14 1998 ada

  The directory « ada » belongs to the user « purna »
  The owner (purna) has « rwx » permissions (read, write, parse)
  The permissions for the group « wheel » are « rwx »
  The other users have only « r-x » permissions, they are not allowed to write into the directory "ada".

How to modify permissions ? (symbolic coding)

- The **chmod** command (CHange MODe) enables the owner of the file to modify the permissions.

- The permissions may be presented with symbols (letters r, w ,x) or digits.
  - **[ugoa] [[+-=] [rwx]]** files
    The **ugoa** arguments represents the category of users for which permissions are set or modified.
    - **u** : owner (User)
    - **g** : group
    - **o** : others
    - **a** : all (default value)
  - The operator
    - **+** adds a permission
    - **-** removes a permission
    - **=** assigns permission
  - rwx correspond to the read, write and execute permissions.

- **Examples**
  - **chmod o+x toto**
    adds (+) the execute permission (x) to the other users (o) than the owner of the file and the members of the group
  - **chmod o-w toto**
    removes (-) the write permission (w) to the other users (o)

How to modify permissions ? (octal coding)

- This coding uses 3 digits in octal.
  - The first digit represents the permissions for the owner.
  - The second digit, the permission for the group,
  - The third digit, the permissions for the other users.
- For each user category (owner, group and others), we add the permissions according to the following rule :
  - 4 read
  - 2 write
  - 1 execute
- **Examples**

  chmod 750 toto
  - gives all permission to the owner (4+2+1)
  - gives read and execute permissions to the member of the group (4+1)
  - gives no permissions to other users (0)

Directories and files

- To create a directory => **mkdir** (Make DIRectory)
  Example mkdir profs eleves istep
- To remove an empty directory => **rmdir** (ReMove DIRectory)
- To copy files => **cp**
- To remove files => **rm**
- To move (or rename files) => **mv**

Wildcards characters

- You may use wildcards characters in the name of a file to replace one or a sequence of characters.
- The character « * » replaces any sequence of characters :
  cp *.c prog => copy any file which name is ending by **« .c »** in the **prog** directory
  rm messages* => removes any file which name begins with « **messages** »
- The character « **?** » replaces one character :
  rm t?t? => removes every file for which the name's length is 4 et which first and third letter of the name is «t»
- The character "~" corresponds to the user home directory.

Recursion

- Most commands accept the -**r** or -**R** argument that enables to apply the command recursively (parsing the directories and sub-directories).

  Examples :
  rm –r prog => removes the directory **prog** and every sub-directory
  chmod –R 750 * => sets the permissions for each file in the current directory and all its sub-directories

Links

- Instead of copying a file into another directory, it is possible to create a link to this file.
- The **ln** command (LiNk) creates links.
- There are 2 kinds of links :
  - hard links
  - symbolic links

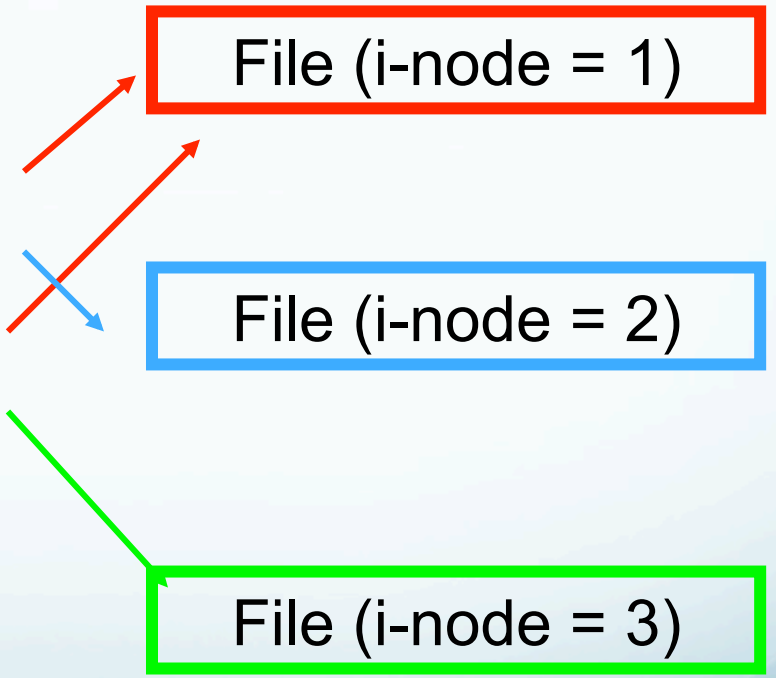
Hard links

- In a file system, files are identified by a unique number called i-node. In fact, a file is an hard link to this i-node.
- You may create as many links to this i-node as you want.
- Hard links can only point to files (not to directories).
- The second column displayed by the **ls** command, shows the number of hard links for this file.
- Deleting a file actually means to remove an hard link to this file. If the number of hard links becomes 0, the file (the data) is finally deleted.

Hard links

- **ln** *target link_name*
- Example : **ln** *toto tata*

toto i-node =1, count = 2
titi i-node = 2, count =1
tata i-node =1, count = 2
tutu i-node = 3, count =1

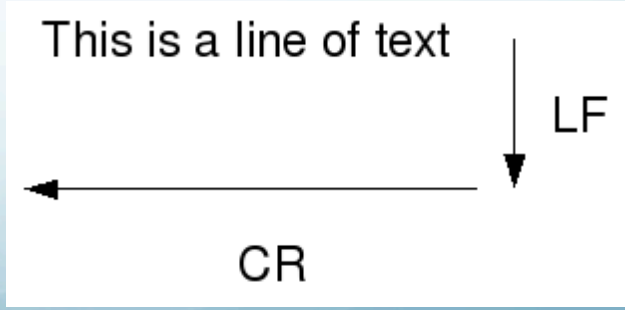


Symlinks

- Symbolic links point to the name (the path) of a file (or directory) instead of directly pointing to the i-node.
- They also are created by the **ln** command but with a **-s** argument.
- A symbolic link may point to a file or directory, located on the same or on another file system (even through the network).
- **ln –s** *target link_name*
- Example : **ln –s** *../purna/ada Ada*

Text files

- Text file contains text without any formatting control characters. (Plain ASCII).
- Each line ends with a Line Feed (LF) character (0x0A ASCII code).
- When you display the content of a text file on the screen, the Unix terminal automatically adds a Carriage Return (CR, 0x0D ASCII code) character before the line feed in order to reset the cursor position at the beginning of the line.



File conversion

- But under MS-DOS or Windows, each line of text is ended by the pair of characters CRLF. (0x0D0A).
- Under previous versions of MacOS, the end of a text line was a CR without LF ...
- Consequences (bad news) :
  - If you transfer a Unix text file to Windows without conversion, the CR is missing. If you display it under Windows, the text will looks like stairs.
  - If you transfer a Windows file to Unix without conversion, the CR character will be displayed as Ctrl-m (^M) and causes some trouble typically for shells or compilers.
- Converting files
  - **fromdos** and **todos** (under Linux)
  - The **ascii** command embedded in the ftp program (file transfer program) converts file when transferring them over the network.

Printing files

- The **lpr** (Line PRinter) command sends printable files to printer devices. Printable files are :
  - plain ASCII files
  - Postscript files
  - HPGL files
- The printer is selected with the **-P** argument followed by the name of the printer.
  Example : lpr –P orsay cours.ps
- The **lpq (**Line Printer Queue) displays the list of print jobs.
- **lprm** [id] [user] removes a print job identified by id or all the print jobs sent by a user.
- The commands **lpq** and **lprm** are only useful when the printer is directly connected to the Unix system, on the serial or parallel port for example.
- For network printers, you have to use :
  - the queue management of the printer
  - or a print server

Output redirection

- The result of any command may be redirected to a file instead of being displayed on the screen.
- The operator **>** creates a new file with the result.
- The operator **>>** appends the result to an existing file.
- **Examples** :
  ls –l > listing
  unix2dos program.c > program_dos.c

Error redirection

- for each process, you have not only a standard input stream (the keyboard), a standard output stream (the screen) but also a standard error stream.
- This error stream may also be redirected to a file:
  myprogram 2> errors
- You may redirect the standard error stream into the standard output stream:
  myprogram 2>&1
- Or the standard output stream into the error stream:
  myprogram 1>&2
- Thus you redirect any output of a program into 1 file:
  myprogram &> mylog

Input redirection

- In order to automate a program, the data that would have been entered by a user from the keyboard, are stored in a file (one data per line) and the input of the program my be redirected from this file if you use the **<** operator.

  Example :
  The file answers stores the input data
  prog < answers

Program redirection

- The «| » (pipe) operator redirects the output of one program to the input of another program.
- This feature enables you to create a workflow of computations over a stream of data.
- Each program runs in parallel of the other programs.

Example : ls –l | unix2dos | lpr –P orsay
prints on a windows printer (« orsay ») the result of the "ls –l" command after converting it from unix to dos.

Text files concatenation

- The **cat** command (CATenate) merges text files.
  Example :
  cat toto tutu
  will display on the screen the content of the file "toto" and then the content of the file "tutu.

- Of course, it is possible to redirect the output to a file :
  Example :
  cat toto tutu > titi
  the file "titi" will contain the content of the file "toto" and the content of the file "tutu".

Paging with the more command

- The **more** command displays a text file page per page.
- At the end of each page, **more** waits until the user press
  - the space bar to display the next page,
  - or the enter key to display the next line of the text,
  - or **Ctrl-B** to display the previous page again,
  - or **q** to quit more.

Dumping a binary file

- Sometimes, you need to display the content of a file in a numeric format :
  - when the file is not a text file
  - the file contains text and formating codes or non printable characters
- The **od** (Octal Dump) command displays the content of a file in octal notation by default or
  - -h for hexadecimal
  - -d for decimal
  - -c or –a for ASCII

Editing text files

- Linux offers a lot of text editors in graphic mode : **kedit**, **gedit**, ...
- The IDE (eclipse, netbeans, BlueJ, ...) also.
- But if for any reason (maintenance mode, remote access, ...) you could not have access to the GUI, Unix and Linux offers editors in text mode : **vi** or **emacs**.
- **vi (basic concepts)**
- First, you cannot and use internal commands (even cursor positioning) at the same time.
  So, there are 2 modes :
  - insertion mode
  - command mode
- To enter the insertion mode,
  - type i to insert text at current cursor position
  - type A to append text at the end of the line
- To come back to command mode, press the escape key.

Some vi commands

- Control-F => next page
- Control-B => previous page
- x deletes the character under the cursor
- dd deletes the current line
- :w [file name] saves the file
- :q quits vi
- :q! quits without saving

Finding text inside files : grep

- **grep** (General Regular Expression Parser) parses files to find text that matches a pattern.
- By default, the **grep** command displays the lines matching the expression.
- The **-n** argument displays the line number.

Example :
grep –n hosts /etc/init.d/*
Retrieves any line with the « hosts » text in every file within the "/etc/init.d" directory.

Finding files : find

- **find** retrieves files locations matching some condition.

  find path conditions
  - « path » => in which directories to look to find files
  - « conditions » => search criteria and eventually actions to perform on match

- Search criteria :
  - -name file_name : search according to the file name
  - -user user : search according to the ownership of the file
  - -type type of file : f (file), d (directory), l (symbolic link) ...
  - -mtime n : search according to the last update time
  - -size n : search according to the size of the file

- The action to perform when matching
  - -print : displays the absolute path for the file
  - -exec : the command to perform

Disk usage

- The **du** command (Disk Usage) displays the space used by each directory (including sub-directories).
- The default directory is the current directory.
- The –s argument (summary) displays the total by directory and not the details.

Example :

```
du
6943 ./MATT/webdocs/api
48 ./MATT/webdocs/tools
7025 ./MATT/webdocs
8129 ./MATT
```

archiving files with tar

- The **tar** command (Tape Archive) enables the user to create archives or to extract files from archives. The archives files have the .tar extension in their names.

  tar [options] [file .tar] [files]

- Options :
  - c (create) => creates an archive
  - t (table) => displays the Table Of Content of an archive
  - x (eXtract) => extract files
  - f (file) just before the name of the archive file. If this option is not provided by the user, the archive is not stored into a file but displayed on the screen.

- **Example** :
  tar cvf programs.tar *.c

  Creates an archive named programs.tar containing every file which name ends with ".c".

Compression

- **compress** and **uncompress** are the standard Unix commands to compress and uncompress files.
- Files compressed by **compress** have the .Z extension at the end of their names.
- But today and particularly under Linux, the most popular command is **gzip**/**gunzip**.
- **gzip** files have the .gz extension.
- **gzip** files may be uncompressed under standard Windows tools too.
- The **zip** and **unzip** commands provides the same features than the corresponding one under DOS/ Windows (Winzip).

Process

- A program is a file stored in the file system, that contains code runnable by the system.
- A process is an instance (a copy in RAM) of a running program. Shortly, when you launch a program, it becomes a process.
- You may have many processes for the same program, for example many open shell terminals.
- A process may launch processes. (this is the beginning of system programming).

Process Identifier

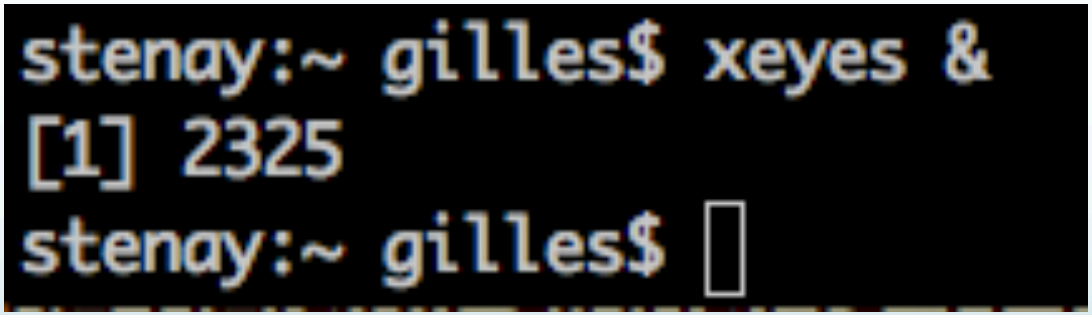
- Each process is identified by a unique PID (Process Identifier), this PID is assigned by the system.
- Each process is linked to the process that launched it (the parent process).
- The PID of the parent process is called the PPID (Parent Process Identifier).
- The processes are linked together in a tree which root is the first process launched by the system (init).

List of processes

- The **ps** command displays information about current processes.
- There is a lot of options for this command to select the processes and details to display. Read the manual page (**man**) for this command.
- If you type **ps**, it just display the children processes of this shell.
- I mostly use **ps –edf** under Linux and **ps –ef** under MacOS X.

Foreground/background process

- A process launched in foreground mode blocks the shell until its end of use.
- To launch a process in background, just add the character « & » at the end of the command line.
- The background process starts, displays its PID, and the user is still able to use the shell.



Foreground/background process

- Control-C stops a foreground process
- Control-Z suspends a foreground process
- the **fg** command (ForeGround) restarts in foreground a suspended process
- the **bg** command (BackGround) restarts in background a suspended process

jobs

- The command **jobs** display the status of processes launched from the current shell.

- **Example** :

```
jobs
[1] - Running xterm
[2] + Suspended xclock
```

- The job number (the number between brackets) may be used to signal the process : **bg** %2

Signalling processes

- The **kill** command signals the process which PID is provided as a parameter.
- The **kill** command also accepts the kind of signal as a parameter.
- The default signal is TERMINATE.
- **Examples** :
  kill 1238 (terminates the process which PID is 1238),
  kill %2 (terminates the process which is the second in the list displayed by the command **jobs**).

Some networks commands

- hostname, hosts, DNS, nslookup
- ifconfig, arp and netstat
- telnet, ssh
- ftp
- X11
- mail architecture

hosts

- The **hostname** command displays the name of the system to which the user is connected.
- The host name should be unique (inside a domain).
- The host name should always begins with a lowercase letter.
- The /etc/hosts file manages locally the conversion between hostnames and IP addresses.
- At company and internet level, a distributed directory manages the conversion (Domain Name Service).
- nslookup allows you to manually retrieve the IP address corresponding to an hostname (or the reverse).

ifconfig

- **ifconfig** displays information about your network interfaces :
  MAC addresses, IP addresses, speed, ...
- If you are the system administrator, you may use **ifconfig** to change the network configuration.

```
en0:
flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX
,MULTICAST> mtu 1500
options=2b<RXCSUM,TXCSUM,VLAN_HWTAGGING,TSO
4> ether a8:20:66:26:8b:d7 inet6
fe80::aa20:66ff:fe26:8bd7%en0 prefixlen 64 scopeid
0x4 inet 172.18.123.101 netmask 0xffff0000 broadcast
172.18.255.255 media: autoselect (100baseTX <full-
duplex,flow-control>) status: active
```

arp

- arp displays the mapping between IP addresses and physical addresses (Ethernet at ISEP).

```
pf-18.isep.fr (172.18.0.1) at 0:0:5e:0:1:5 on en0 ifscope [ethernet]
miller-18.isep.fr (172.18.0.3) at 0:18:8b:f9:2d:c1 on en0 ifscope [ethernet]
marcus-18.isep.fr (172.18.0.4) at 0:13:72:5b:c:e on en0 ifscope [ethernet]
goeland.isep.fr (172.18.31.2) at d4:be:d9:a0:79:2d on en0 ifscope [ethernet]
midnight.isep.fr (172.18.53.2) at 0:25:4b:9d:33:8e on en0 ifscope [ethernet]
newhart.isep.fr (172.18.54.107) at f8:b1:56:b1:53:8e on en0 ifscope [ethernet]
imp-n54-4.isep.fr (172.18.54.109) at a0:2b:b8:5f:3:6d on en0 ifscope [ethernet]
gentiane.isep.fr (172.18.57.2) at 0:25:64:9c:b9:31 on en0 ifscope [ethernet]
mangue.isep.fr (172.18.59.5) at 0:19:b9:12:4a:69 on en0 ifscope [ethernet]
roitelet.isep.fr (172.18.60.9) at b8:ca:3a:7e:f5:d on en0 ifscope [ethernet]
erable.isep.fr (172.18.61.5) at bc:30:5b:b1:a0:1a on en0 ifscope [ethernet]
hyperv.isep.fr (172.18.66.4) at 0:50:56:a8:51:40 on en0 ifscope [ethernet]
```

netstat

- netstat displays information about:
  - -i : interfaces
  - -r : the routing table
  - -a : the sockets

```
stenay:~ gilles$ netstat -r
Routing tables

Internet:
Destination        Gateway            Flags        Refs      Use   Netif Expire
default            pf-18.isep.fr      UGSc           13        0     en0
127                localhost          UCS             0        0     lo0
localhost          localhost          UH              4    11129     lo0
169.254            link#4             UCS            15        0     en0
```

netstat -a

```
stenay:~ gilles$ netstat -a|more
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  stenay.isep.fr.60072   par03s15-in-f5.1.https ESTABLISHED
tcp4       0      0  stenay.isep.fr.60071   imap4.orange.fr.imaps  ESTABLISHED
tcp4       0      0  stenay.isep.fr.60070   imap4.orange.fr.imaps  ESTABLISHED
tcp4       0      0  stenay.isep.fr.60069   imap4.orange.fr.imaps  ESTABLISHED
tcp4       0      0  stenay.isep.fr.51539   eric.isep.fr.imaps     CLOSE_WAIT
tcp4       0      0  stenay.isep.fr.49234   imap-y.bbox.fr.imaps   ESTABLISHED
tcp4       0      0  stenay.isep.fr.49233   imap-y.bbox.fr.imaps   ESTABLISHED
tcp4       0      0  stenay.isep.fr.49232   imap-y.bbox.fr.imaps   ESTABLISHED
tcp4       0      0  stenay.isep.fr.49231   imap-y.bbox.fr.imaps   ESTABLISHED
tcp4       0      0  stenay.isep.fr.49217   eric.isep.fr.imaps     ESTABLISHED
tcp4       0      0  stenay.isep.fr.49195   eric.isep.fr.imaps     ESTABLISHED
tcp4       0      0  stenay.isep.fr.49193   eric.isep.fr.imaps     ESTABLISHED
tcp6       0      0  localhost.25035        *.*                    LISTEN
tcp4       0      0  localhost.25035        *.*                    LISTEN
tcp4       0      0  stenay.isep.fr.49169   17.172.232.218.https   ESTABLISHED
tcp4       0      0  localhost.49154        localhost.1023         ESTABLISHED
tcp4       0      0  localhost.1023         localhost.49154        ESTABLISHED
```

remote shell

- The **telnet** command enables the user to connect to another system (usually a Unix system) and to access a shell on this system.

  telnet hostname or @IP of the remote system

- **Control-]** enables the user to access the menu of **telnet**. This feature is useful to exit telnet when the remote shell is blocked.

- But telnet is not safe : the communication is not encrypted.
- That is the reason why telnet is generally replaced by ssh (Secure SHell).

  ssh -l user hostname

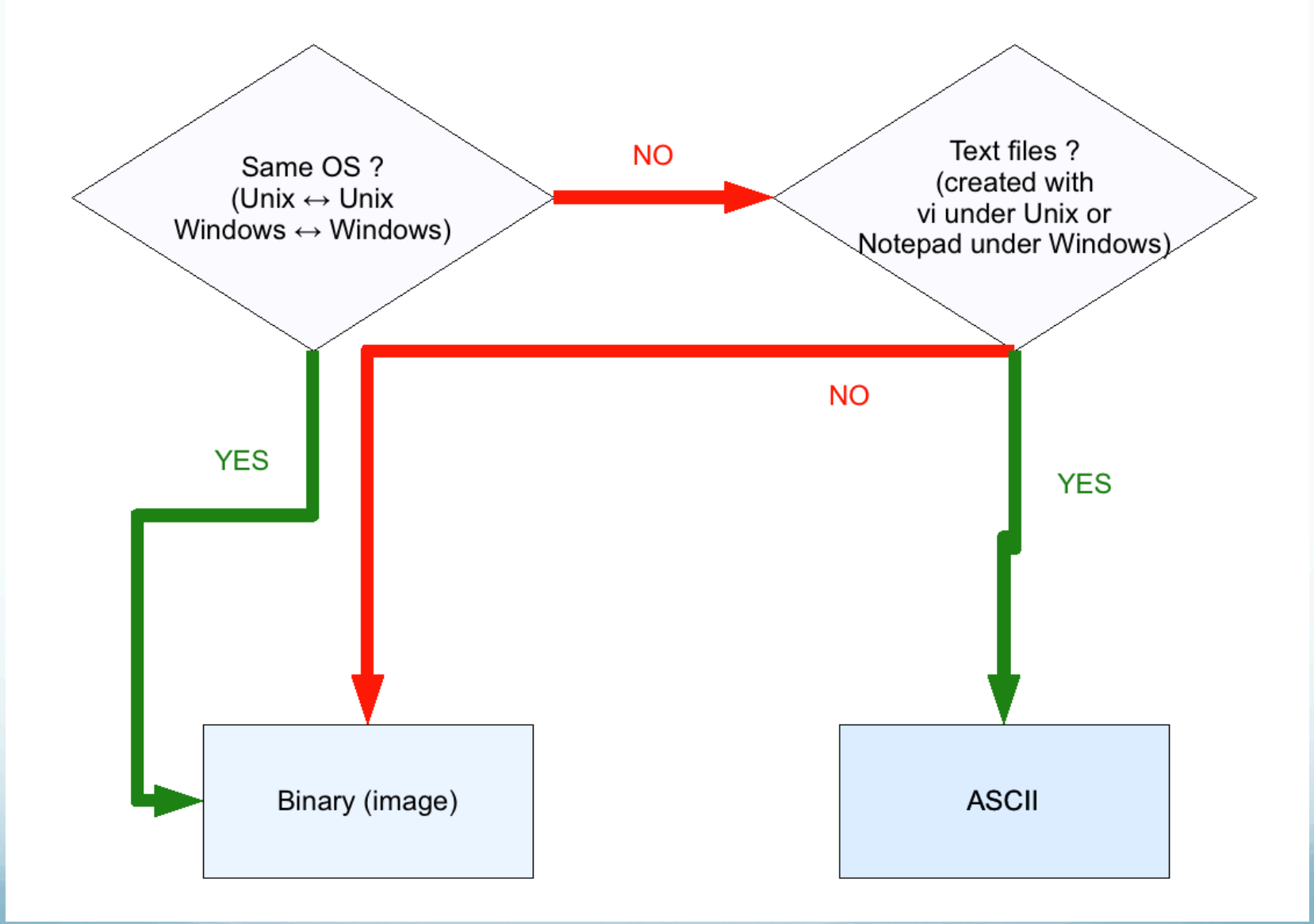
file transfer with ftp

- feature : transfer files between systems
- user interface : commands, on line help
- daemon (server side) : ftpd, may be launched by inetd
- used files :
  - /etc/passwd : the name and the password of the user should exist in the passwd file on the remote system
  - /etc/shells : the user should use a login shell that exists in the /etc/shells file on the remote system
  - /etc/ftpusers : list of users who have not the right to use ftp (generalyy anonymous users like root, oracle, ...)
  - $HOME/.netrc : allows ftp to automatically authenticate the user !!! DANGER !!!

ftp parameters

- -v : verbose, displays messages sent by the remote server (ftpd) and the number of transfered blocks
- -n : nologin, remove the automated authentication for this connexion
- -i : no inquire, diables the prompt for multiple file transfer
- -g : no global, disables the processing of wildcard characters(*, ?)
- -d : debug
- hostname : name of the remote server

ftp text format conversion



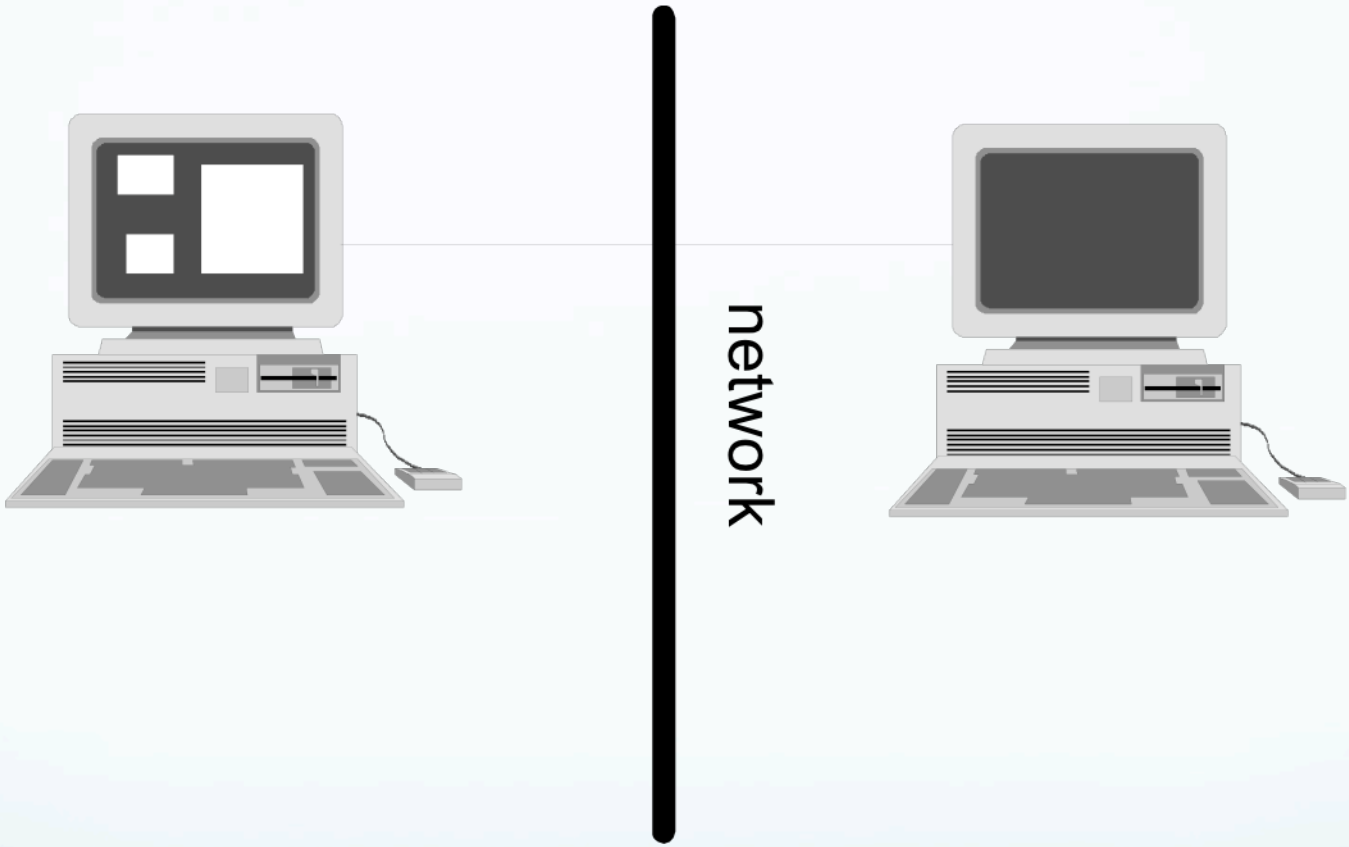
ftp internal commands

- ! [command] : runs the command (or a shell if no parameter) locally instead of remotely
- append fl [fd] : appends the local file (fl) to the remote file (fd)
- ascii : converts text files from unix to windows (or the reverse)
- bell : bips each time a file transfer is finished
- binary : doest not convert files
- bye, quit : exit ftp
- cd directory : jumps to this remote directory
- close : closes the connexion with the remote server but does not quit ftp
- delete fd : deletes the remote file (fd)
- ls [-l] [directory] [fl] : displays the content of a remote directory and if provided, stores the result into this local file (fl) (dir = ls –l)
- get fd [fl] : receives a remote file (fd); recv=get
- glob : like the -g option (does not take wildcards into account)
- hash : does not display # for each block transferred
- help [command] : displays the list of available commands or provides help for one command (? = help)
- lcd [répertoire] : changes the current local directory, ($HOME) by default.
- mdelete fd1 .. fdn : deletes multiple remote files
- mls : like ls but you my use wildcards within the name of the directory
- mget fd1 .. fdn : receives mulltiple files
- mkdir répertoire : creates a remote directory
- mput fl1 .. fln : sends multiple files
- open hostname : connects to this remote server
- prompt : requires confirm for each file for commands with m prefix (mget, mput, mdelete).
- put fl [fd] : sends this local file (fl); send=put
- pwd : displays the name of the current directory on the remote server

X/Window 11

- X11 is the standard GUI for Unix.
- X11 is designed as a client/server application.
- A X11 is a program that uses the X11 protocol to send its output to a X11 server that may be located on a remote system over the network.
- Thus, you have to provide the location of the server to the client either :
  - with the -display parameter when you launch the client
  - or with the DISPLAY environment variable

X/Window 11



- The X11 server (on the left) manages input/output (screen, keyboard, mouse). The system on the right hosts the differents X11 clients.

The display parameter or variable

[unix]:0[.0]

- hostname or IP address of the X11 server
- number for the X11 service on the server = offset of the socket number related to the base socket number for X11(6000)
- terminal or screen number

How to launch a X11 client ?

- **Example :** xclock –display zouzoute:0 &
- The clock (xclock) will be displayed on the X11 server of the system called zouzoute.
- The program may beterminated either in terminating the client or in closing the window on the server side.

Windows managers

- The X11 manages the **content** of the different windows, but not the windows (size, location, iconification, ...).
- The windows and the menus are managed by one of the different windows managers :
  - KDE
  - GNOME
  - Xfence
  - uwm
  - twm ...

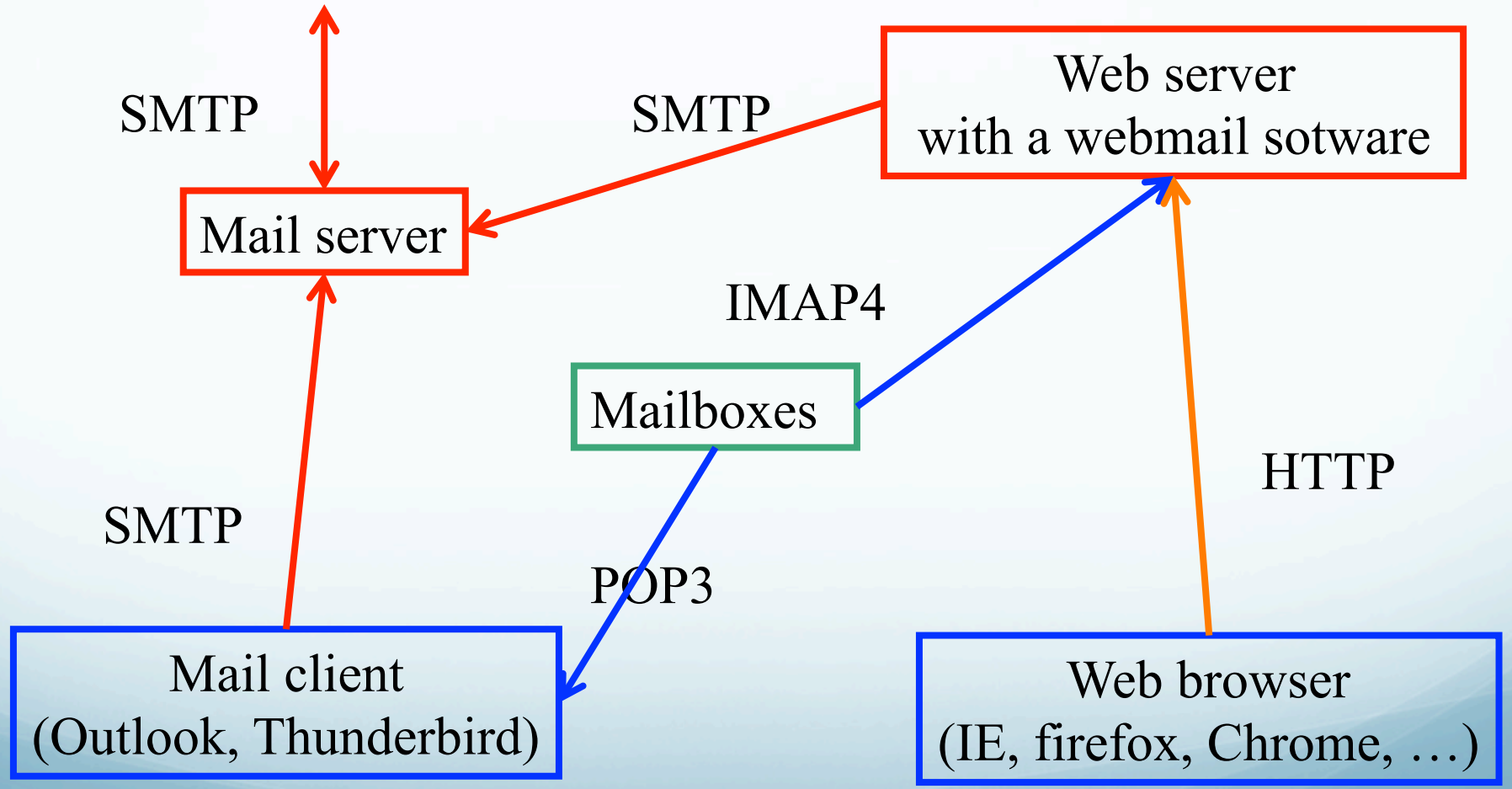
Access management to X11

- Each X11 server manages a list of hosts that are allowed to communicate with this server (i.e. to send it windows to display).
- At server launch time, this list is restricted to the local host (the server itself).
- The xhost command :
  - without argument => displays the list of allowed client systems
  - + hostname => adds this host to the list of allowed systems
  - + => disable security, all systems are allowed to communicate with this server
  - - hostname => removes this host from the list of allowed systems
  - - => restricts access only to the systems that are explicitely named in the list

The mail

- SMTP protocol (Simple Mail Transfer Protocol) between mail servers to forward the mails
- POP protocol (Post Office Protocol) or IMAP to retrieve your mail out of the mailbox located on the mail server
- You may also use a web browser to connect to a webmail software like SquirrelMail that provides an acces to your mailbox.
- You also may use mail clients to manage your mails :
  - Thunderbird, evolution
  - Microsoft outlook ...

The mail

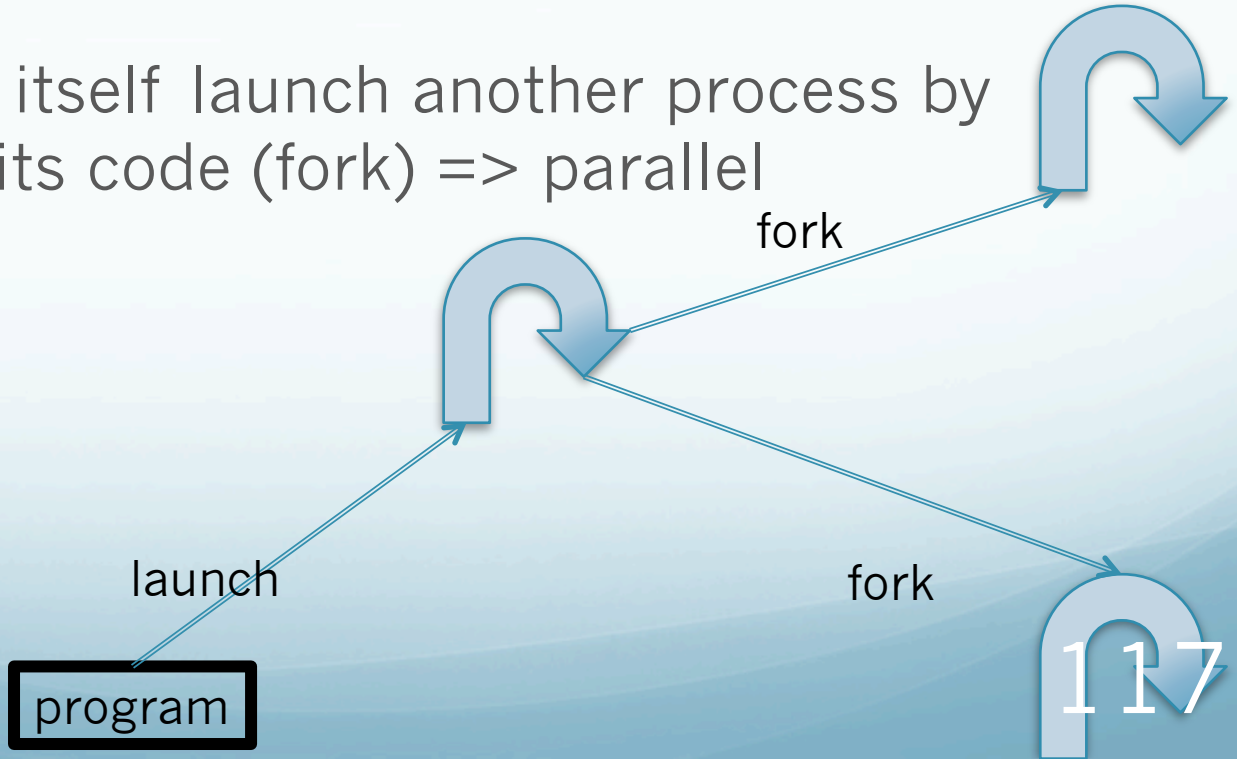

SMTP
SMTP
Web server
with a webmail sotware
Mail server
IMAP4
Mailboxes
HTTP
SMTP
POP3
Mail client
(Outlook, Thunderbird)
Web browser
(IE, firefox, Chrome, …)

System programming

- I call "system programming", writing programs that:
  - will be part of the OS, like device drivers => Kernel mode programming
  - interact with the OS => User mode programming
- We will for this course, just develop in User mode.
- We just need to use the system calls functions provided by the libc library.

Process

- When we launch a program (i.e. a file that contains code), a process is launched and managed by the task scheduler of the OS kernel.
- If the program is launched many times, there will be as many processes in the memory running the same code.
- A process may itself launch another process by duplication of its code (fork) => parallel programming



PID

- You often have to know the PID (Process Identifier) to:
  - execute code according to a process
  - communicate between processes (signalling)
- The **getpid** function returns the PID of the process.
- The **getppid** function returns the PID of the parent process, i.e., the PID of the process that launched this one.

```
#include <sys/types.h>
#include <unistd.h>

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

- Sous Unix, **pid_t** est un entier (**int**).

Exercise 1

- write a program that displays its PID and its PPID.
- which program matches the PPID ?

```
stenay:processus gilles$ ./affichePID
mon PID = 3324
le PID du processus père = 2299
stenay:processus gilles$
```

Process duplication : fork

- In many application, typically in image processing, it is convenient and efficient to split the data and process the slices in parallel.
- Thus, this is why we should be able to run the code of a program multiple times in parallel.
- The **fork**() function duplicates the code of a process and launch it in parallel.

Exercise 2

- Write a program that duplicates and displays the integers from 0 to 9.

- The result should display 0 twice, the 1 twice and so on.
- But, because, the loop is too short, one process has enough time to finish the loop before the other begins.
- We will slow down the loop to be able to see the parallel computing.

sleep

- This is not always a good practice, but sometimes, we need a process to wait some time in order to let other processes to finish their computation before it will get the results.
- This is typically the case when different processes access data storage with different speed (RAM // disk)
- The **sleep** function blocks a process for the time passed as a parameter.

Exercise 3

- Reuse the code of the previous exercise (2) and add a sleep call in the loop.
- 1 second should be enough.

Running a different code in the child process

- The **fork**() function returns the PID of the child process in the parent process.
- In the child process, the returned value is 0.
- This is the way to differentiate the parent and the child process.
- The return code is -1 in case of error:
  - not enough memory left to create the child
  - max number of processes reached for this user

Exercise 4

- Write a program that duplicates.
- Each process should display if it is the parent or the child process.
- The parent will display its PID and the PID of the child process.
- The child process will display its PID and the parent PID.

```
stenay:processus gilles$ ./perefils
je suis dans le père, mon PID est 3267, mon fils est 3268
je suis dans le fils, mon PID est 3268, mon père est 1
stenay:processus gilles$
```

Why is the Parent PID 1 when displayed from the child process ?

Zombie process

- So far, we did not care about the order of termination between the parent and the child process.
- If a child process terminates while its parent was not waiting or terminates, it becomes a zombie.
- It stays in the list of processes of the scheduler, but its parent is now the root process (PID 1).
- This issue should be avoided to spare memory.
- The **wait(int * status)** function blocks the parent process until the state of one or more of its children processes change. For example, when it terminates.

Signals

- Sending a signal is the lowest level way of communication between processes.
- A signal does not transport a message.
- The only information that differentiates signals is the signal number.
- A signal is often compared to an interruption or called a software interruption.

Different signals

- When a process receives a signal, its behaviour will be different according to the signal number:
  - terminates the process
  - ignore the signal
  - resume processing for a process blocked by a **sleep** call for example, ...
- It is difficult to remember the behaviour of each signal just with the signal number. But, in C language, for each signal will have macro definition with a name :
  - 3 => SIGQUIT (Ctrl-C from the keyboard)
  - 15 => SIGTERM ...

User signals

- Some signals may be used by the developers for they own signalling requirements:
  - SIGUSR1 (numbers 30, 10, 16) TermSignal user1
  - SIGUSR2 (numbers, 31, 12, 17) TermSignal User2

- The default behaviour of the receiver process is to terminate (Term).
- But the developer may override this behaviour using the **signal()** function.

Sending a signal

- The C language function to send a signal is kill() like the shell command.

```
#include <sys/types.h>
#include <signal.h>

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

  - The 1st parameter (pid) is the PID of the target process.
  - The 2nd parameter (sig) is the number of the signal to send. You may use the name instead of the number.

- The error case may be:
  - wrong signal number
  - no process matching this PID
  - not allowed to signal this process

Sending a signal

- Examples:

kill (15678, 15);

sends the signal number 15 (SIGTERM) to the process with PID 15678 that should terminate when it receives it.

kill(123454, SIGUSR1);

sends the signal 10 to the process with PID 123454 that should terminate if the developer of this process does not provide an handler to manage this signal.

Exercise 5

- write 2 programs, one that sleeps and the other that wakes it.
- The sleeper should first display its PID and then sleeps 20 seconds.
- When the sleeper wakes up, it should display that it wakes up.
- The alarm asks the user to enter the PID of the sleeper and sends a signal SIGUSR1 to the sleeper.
- Launch the sleeper in background then launch the alarm.

Exercise 5

```
Stenay:processus gilles$ ./dormeur &
[1] 963
Stenay:processus gilles$ mon pid est 963
je vais m'endormir 20 secondes

Stenay:processus gilles$ ./reveil
PID du processus à réveiller : 963
[1]+ User defined signal 1: 30 ./dormeur
```

managing signals (handlers)

- when a process receives a signal, the developer may provide an handler : a code that will be performed according to the signal number
- except for SIGINT and SIGKILL
- The **signal** function maps the signal number and the function to call
  signal(signal number, function)
- **signal** is very low-level. For more sophisticated applications, you'd better use **sigaction**.

Exercise 6

- Reuse the sleeper program of previous exercise (5) and add a function called alarm.
- The function alarm will display "I have been waked up too early".
- In the main function, use the **signal**() function to map the alarm function with signal SIGUSR1.
- The **sleep**() function returns the number of seconds that left when the signal was received. Display the remaining time.

Exercise 6

```
Stenay:processus gilles$ ./dormeur &
[1] 1023
Stenay:processus gilles$ mon pid est 1023
je vais m'endormir 20 secondes

Stenay:processus gilles$ ./reveil
PID du processus à réveiller : 1023
j'ai été reveillé
il restait 13 secondes de sommeil
[1]+ Done ./dormeur
```

- You also may use **wait**() instead of **sleep**() to let the sleeper wait infinitely until it receives a signal

Communication between processes

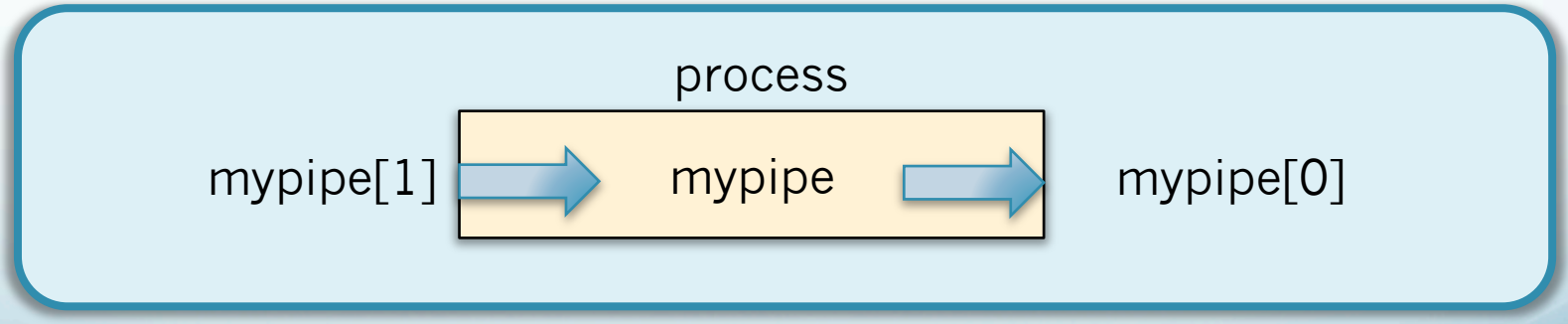
- between parent and children => pipes (volatile pipes)
- between unrelated processes =>
  - named pipes (FIFO)
  - IPC (Inter Process Communication)
    - message queues
    - shared memory
    - => semaphores

Pipes

- a pipe is a one-way communication between a parent process and one of its children
  - character-oriented => a stream of bytes
  - or message oriented => you first have to send the length of the message
- one-way => you need another pipe if you want to communicate from the child to the parent
- Children may communicate between them using the same pipe(s) because they inherit it after the **fork()**

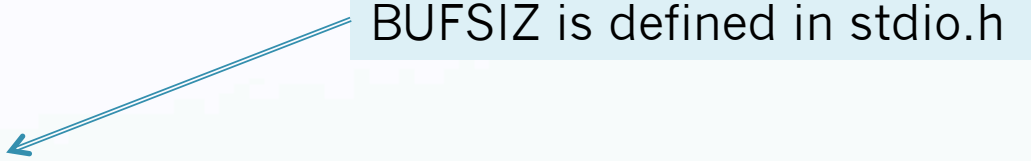
Pipes

- a pipe is an pair of 2 descriptors (an array of 2 integers):
  - **int mypipe[2];**
  - mypipe[0] => for reading data out from the pipe
  - mypipe[1] => for writing into the tube
- you create a pipe with the pipe(int[]) function
  - **pipe(mypipe)**;



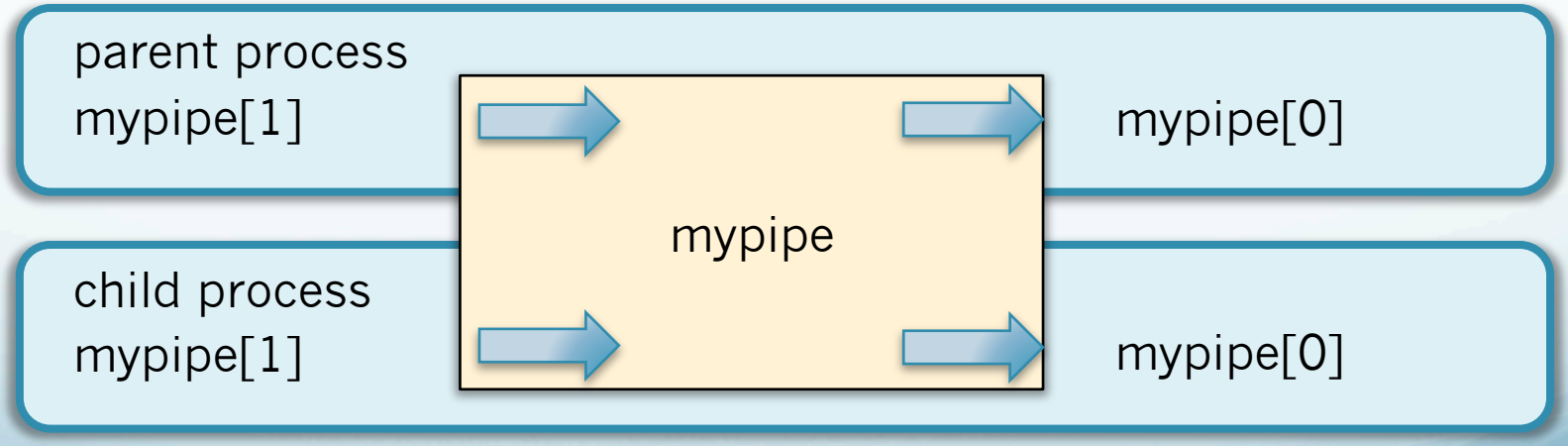
creation and use of a pipe

```c
#include <stdio.h>
#include <memory.h>
#include <unistd.h>

  int main()
  {
   char buffer[BUFSIZ+1];

   /* create the pipe */
   int mypipe[2];
   pipe(mypipe);

   /* write into the pipe */
   write(mypipe[1], "Hello World\n", strlen("Hello World\n"));

   /* read the pipe and print the read value */
   read(mypipe[0], buffer, BUFSIZ);
   printf("%s", buffer);
   return 0;
  }
```



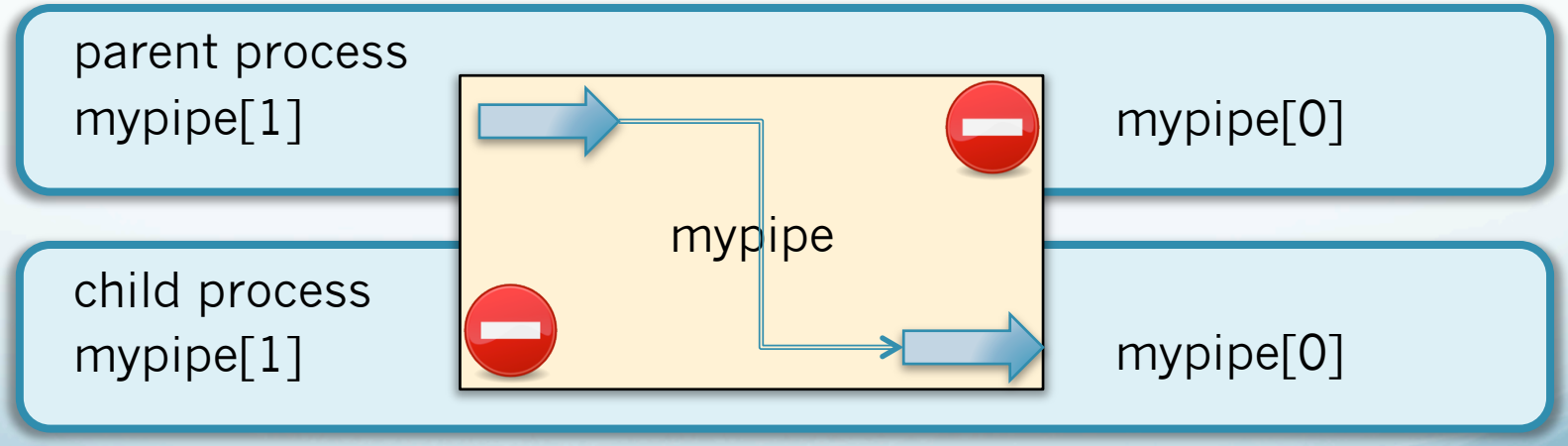
Sharing the pipe between the parent and the child

- The goal is to communicate between a parent and a child process.
- To share the pipe, you have to create it before the fork.
- But after the fork, if you write into the pipe, we do not know which process will receive the data, the parent or the child ?



closing unused ends of the pipe

- to ensure which process will receive the data, we have to close the unused pipe descriptors
- If the parent is the writer and the child the reader, we have to close:
  - mypipe[0] in the parent
  - mypipe[1] in the child



writing and reading with pipes

- a pipe is a stream => you use the same kind of read and write functions than with files
- `write(mypipe[1], buffer, strlen(buffer)+1);`
- `read(mypipe[0], buffer, BUFSIZ);`

Exercise 7

- write a program that creates:
  - creates a pipe
  - creates a child process
  - the parent writes a message into the pipe
  - the child reads and displays the message

Named pipes (FIFO)

- The drawback of the previous pipe are:
  - this is just between related processes
  - it is automatically destroyed when the program terminates
- a **named pipe** (also known as a **FIFO** for its behaviour) is an extension to the traditional pipe concept on Unix, and is one of the methods of IPC.
- A named pipe works much like a regular pipe, but does have some noticeable differences:
  - Named pipes exist as a device special file in the file system.
  - Processes of different ancestry can share data through a named pipe.
  - When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

How to create a named pipe

- You may create the named pipe from the shell or in a C program.
- From the shell, you have may:
  - either use the **mknod** command and then assign the permissions:

    mknod myFIFO p

    chmod 666 myFIFO
  - or use the **mkfifo** command that combines the 2 previous commands:

    mkfifo -m a=rw myFIFO
- In C language, you may use the **mknod**() function
- The result is entry like this in the folder:

prw-rw-rw- 1 root root 0 Apr 23 23:01 myFIFO

Using a named pipe

- Using a named pipe is just like using a file:
  - you have to open it using the **fopen**() function
  - you may use the **fgets**, **fputs**, **fprintf**, ... function to read and write
- But
  - reading data out of the pipe is a blocking function => it waits until some data is available
  - writing into a named pipe with no readers will generate a signal.

Exercise 8

1. create a named pipe
2. write a program (the server) that is reading messages from the named pipe and displays them on the screen
3. launch your program in a terminal window
4. write a program (the client) that prompts the user to enter a message and send this message into the pipe
5. launch this program in another window
6. What happens if we launch 2 servers ?

IPC (Inter-Process Communication)

- Unfortunately, we have 2 implementations of IPC:
  - Unix System V
  - POSIX
- I will present only System V. The main differences are for the semaphores.
- Linux provides System V and POSIX IPC
- Mac OS, POSIX only.

Inter-Process Communication (IPC)

- IPCs provide 2 more sophisticated ways of communicating between processes:
  - Message queues
  - Shared memory
- And semaphores to synchronize the access to queues or memory
- each IPC is identified by an integer ID called a key.
- the key should be unique for each kind of IPC.

IPC key

- You may provide your own key but for applications that will be published, your key may conflict with the key of an existing IPC used by some other application.
- To avoid this issue, you may use the ftok() function that will compute an integer out from 2 parameters:
  1. a path to an entry of the file system (i-node number and minor number)
  2. a character

```
key_t   mykey;
mykey = ftok(".", 'a');
```

ipcs command

- The ipcs command can be used to obtain the status of all System V IPC objects.

ipcs -q: Show only message queues
ipcs -s: Show only semaphores
ipcs -m: Show only shared memory

- All is the default

ipcs command

- Here is an example of ipcs on a Beaglebone:

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          debian     600        196608     2          dest
0x00000000 32769      debian     600        393216     2          dest
0x00000000 65538      debian     600        196608     2          dest
0x00000000 98307      debian     600        393216     2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 0          www-data   600        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

ipcrm command

- The ipcrm command can be used to remove an IPC object from the kernel. While IPC objects can be removed via system calls in user code (we'll see how in a moment), the need often arises, especially under development environments, to remove IPC objects manually.

ipcrm <msg | sem | shm> <IPC ID>

- Simply specify whether the object to be deleted is a message queue (*msg*), a semaphore set (*sem*), or a shared memory segment (*shm*). The IPC ID can be obtained by the ipcs command. You have to specify the type of object, since identifiers are unique among the same type.

Message queues

- Message queues can be best described as an internal linked list within the kernel's addressing space.
- Messages can be sent to the queue in order and retrieved from the queue in several different ways => this is not a FIFO, readers may filter the messages.

Creating or retrieving a queue

- In order to create a new message queue, or access an existing queue, the **msgget**() system call is used.
- **int msgget (key_t key, int msgflg);**
- RETURNS:
  - message queue identifier on success
  - -1 on error: errno =
    - EACCESS (permission denied)
    - EEXIST (Queue exists, cannot create)
    - EIDRM (Queue is marked for deletion)
    - ENOENT (Queue does not exist)
    - ENOMEM (Not enough memory to create queue)
    - ENOSPC (Maximum queue limit exceeded)

msgget() parameters

1. The first argument to **msgget**() is the key value (in our case returned by a call to **ftok**()).
   - This key value is then compared to existing key values that exist within the kernel for other message queues.
2. **msgflag** parameter:
   - **IPC_CREAT** Create the queue if it doesn't already exist in the kernel.
   - **IPC_EXCL** When used with IPC_CREAT, fail if queue already exists.
   - An optional octal mode may be OR'd into the mask, since each IPC object has permissions that are similar in functionality to file permissions on a UNIX file system.

msgget() example

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

int main()
{
    int     qid;
    key_t   mykey;
    mykey = ftok(".", 'a');

    qid = msgget( mykey, IPC_CREAT | 0660 );
    if (qid == -1)
    {
        perror("cannot get or create the queue");
        return -1;
    }
    printf("the quid is %d\n", qid);
    return 0;
}
```

run it an then use **ipcs** to check the creation of the queue

Structure of the messages

- The messages that you send into the queue should be declare like this:

```
struct msgbuf {
    long mtype;         /* type of message */
    your own data and type ;      /* message content */
};
```

- the message should be a struct called **msgbuf**
- the 1st field should be a long, it may be used by the reader to filter the messages
- the other fields are at your convenience
- the size should not exceed the limit defined by the system (4 Kbytes)

sending a message

int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

- RETURNS: 0 on success
- -1 on error: errno =
  - EAGAIN (queue is full, and IPC_NOWAIT was asserted)
  - EACCES (permission denied, no write permission)
  - EFAULT (msgp address isn't accessable – invalid)
  - EIDRM (The message queue has been removed)
  - EINTR (Received a signal while waiting to write)
  - EINVAL (Invalid message queue identifier, nonpositive message type, or invalid message size)
  - ENOMEM (Not enough memory to copy message buffer)

msgsend example

```
struct msgbuf {
        long mtype;          /* type of message */
        char message[10];       /* message content */
        };
struct msgbuf buffer;
buffer.mtype = 88;
strcpy(buffer.message, "Bonjour");

int result = msgsnd(qid, &buffer, 10, 0);
if (result == -1)
{
    perror("error when sending a message");
    return -2;
}
```

retrieving messages

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);

1. msgqid is the id of the queue
2. *msgp is a pointer o a message structure variable
3. msgsz is the size of the content of the message
4. msgtyp is a filter of message types
5. if msgflag is equal to 0, msgrcv shall block until a message is received

filtering messages

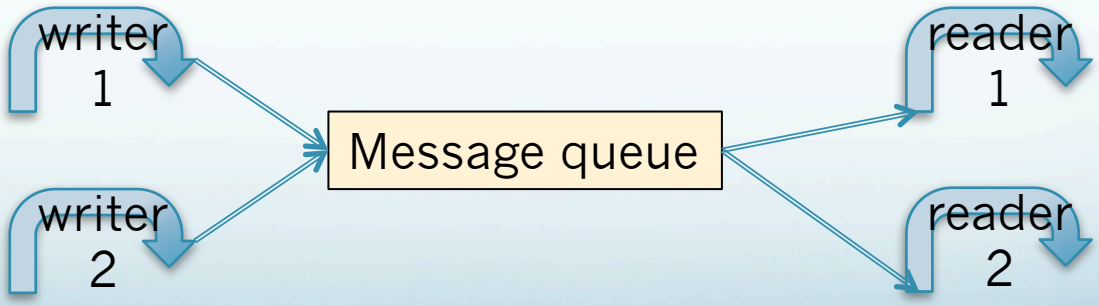
- the *msgtyp* parameter enables us to filter the messages that we want to receive.
- If *msgtyp* is 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgrcv example

```
struct msgbuf reception;

msgrcv(qid, &reception, 10, 88, 0);

printf("Type: %ld Text: %s\n", reception.mtype,
reception.message);
```

Exercise 9

- share a message between 2 writer processes and 2 reader processes.
  - writer 1 will send messages using msgtype 1
  - writer 2 will send messages using msgtype 2
  - reader 1 will read and display messages of msgtype 1
  - reader 2 will read and display messages of msgtype 2



Semaphores

- If we launch multiple readers for the same kind of msgtyp, the messages will be retrieved by the readers at random.
- We may use signalling between the readers to synchronize reading. But each reader should know the PID of the other readers.
- a better way is to share a common semaphore between all readers => each reader will be able to receive a message at its turn.

Semaphores

- Semaphores can best be described as counters used to control access to shared resources by multiple processes.
- They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it.
- The name *semaphore* is actually an old railroad term, referring to the crossroad ``arms'' that prevent cars from crossing the tracks at intersections. They have been replaced by red or green lights.
- If the semaphore is *on* (the arms are up, the light is green), then a resource is available (cars may cross the tracks). However, if the semaphore is *off* (the arms are down, the light is red), then resources are not available (the cars must wait).

Semaphores

- when you create semaphores, you do not just create one semaphore but a set of semaphores.
- For example, if we want to synchronize access not to just one message queue but more, we just need to create one set of semaphores.
- a semaphore is not just a boolean value (0 or 1). It's a counter that we may increment and decrement at our convenience.
- For example, using our previous message queue, we may allow each reader to read at most 2 messages.

Creating or retrieving a semaphore

- **int semget (key_t key, int nsems, int semflg);**
- **nsems** is the number of semaphores in the set
- the **semflg** is the same than with a message queue
- RETURNS:
  - semaphore set IPC identifier on success
  - -1 on error: errno =
    - EACCESS (permission denied)
    - EEXIST (set exists, cannot create (IPC_EXCL))
    - EIDRM (set is marked for deletion)
    - ENOENT (set does not exist, no IPC_CREAT was used)
    - ENOMEM (Not enough memory to create new set)
    - ENOSPC (Maximum set limit exceeded)

Creating or retrieving a semaphore

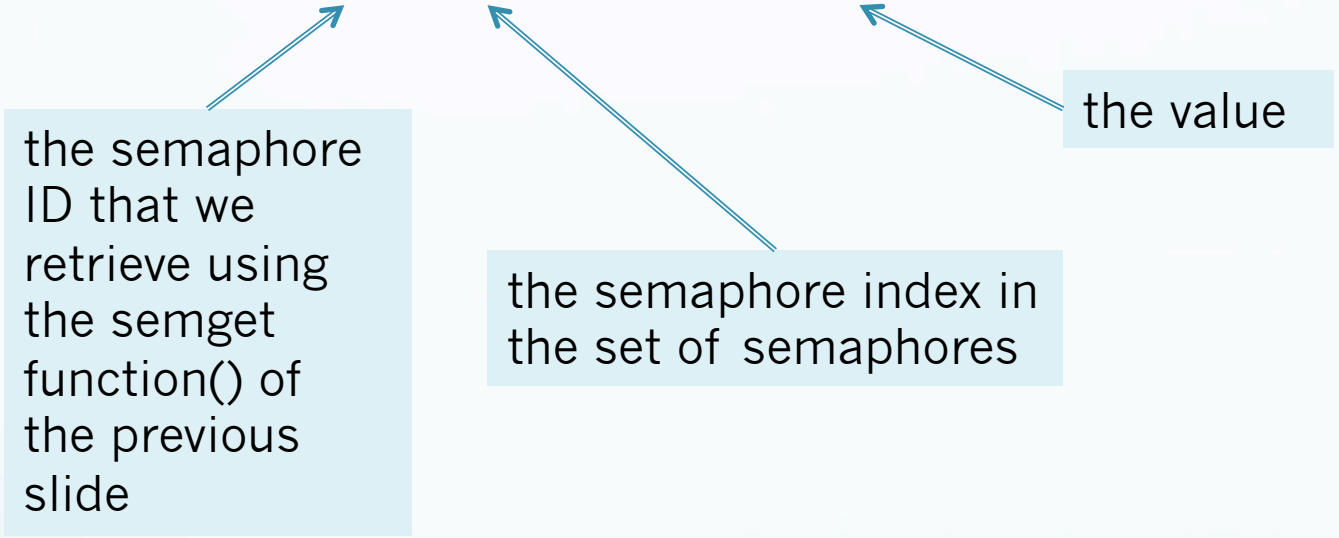
```c
#include <sys/sem.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int     sid;
    key_t   mykey;
    mykey = ftok(".", 'a');

    sid = semget(mykey, 1, IPC_CREAT | 0660);

    if (sid == -1)
    {
        perror("cannot get or create the semaphore");
        return -1;
    }
    printf("the sid is %d\n", sid);
    return 0;
}
```

run it an then use **ipcs** to check the creation of the semaphore

here we use the same key for the semaphore as the message queue

initialising or resetting the value of the semaphore

- **semctl(sid, 0, SETVAL, 1);**



add this instruction at the end of your semaphore creation program

operations on a semaphore

- we may perform on operation at a time on a semaphore or a list of operations => operations are stored in an array.
- each operation is a structure that contains:
  - the semaphore index in the set of semaphores
  - the operation:
    - 0 => wait until the value of the semaphore is 0
    - + value => adds this value to the semaphore
    - - value => subtract this value to the semaphore
  - flags: 0 is in blocking mode

example with just one semaphore and one operation

```
struct sembuf ops[1];
ops[0].sem_num = 0;
ops[0].sem_flg = 0;

printf("try to lock the semaphore\n");
ops[0].sem_op = -1;
int result = semop(sid, ops, 1);
if (result == 0) printf("semaphore locked\n");
else perror("cannot lock semaphore");


printf("try to release the semaphore\n");
ops[0].sem_op = 1;
result = semop(sid, ops, 1);
if (result == 0) printf("semaphore released\n");
else perror ("cannot release semaphore");
```

Exercise 10

- improve the code of the message queue reader in order to balance the reading of messages between the readers
- each reader should read a message at its turn => mutual exclusion (mutex)
- if you have only 2 readers, one reader will read the odd messages and the other the even messages

POSIX semaphores

- System V implementation is not portable, does not run under MacOS X for example.
- POSIX is a standard not just for semaphores (threads, ...)
- A POSIX semaphore is also an integer whose value is never allowed to fall below zero.
- Two operations can be performed on semaphores:
  - increment the semaphore value by one (*sem_post()*);
  - and decrement the semaphore value by one (*sem_wait()*).
- If the value of a semaphore is currently zero, then a *sem_wait()* operation will block until the value becomes greater than zero.
- POSIX semaphores come in two forms:
  - named semaphores => file system
  - unnamed semaphores (memory base => shared memory)

POSIX named semaphores

- A named semaphore is identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX**-*4* (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes.
- On Linux, named semaphores are created in a virtual file system, normally mounted under */dev/shm*, with names of the form *sem.somename*. (This is the reason that semaphore names are limited to **NAME_MAX**-*4* rather than **NAME_MAX** characters.)
- Two processes can operate on the same named semaphore by passing the same name to *sem_open()*.

opening a named semaphore

```
#include <fcntl.h> /* For O_* constants */
#include <sys/stat.h> /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
mode_t mode, unsigned int value);
```

- **Link with** *–pthread*.
- **sem_open**() creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by *name*.

named semaphore example

```
sem_t * sem = sem_open("/semaphore", O_CREAT,
0644, 1);

if (sem == SEM_FAILED)
{
    perror("cannot open or create the semaphore");
    return -2;
}
```

named semaphore operations

- decrementing the semaphore (block if 0):

```
printf("try to lock the semaphore\n");
result = sem_wait(sem);
if (result == 0) printf("semaphore locked\n");
else perror("cannot lock semaphore");
```

- incrementing the semaphore

```
result = sem_post(sem);
if (result == 0) printf("semaphore released\n");
else perror ("cannot release semaphore");
```

Exercise 11

- replace the System V semaphore of exercise 10 with a POSIX named semaphore
- **Link with** *–pthread*
- **Example:**

gcc -Wall –pthread –o queueReaderNS queueReaderNS.c

Shared memory

- Shared memory allows two or more processes to access the same memory as if they all called **malloc** and were returned pointers to the same actual memory.
- When one process changes the memory, all the other processes see the modification.
- Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory.
- Access to this shared memory is as fast as accessing a process's heap memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily.
- But you have to use semaphores to synchronize write and/or read operations.

Shared memory

1. to be able to use shared memory, one process must first allocate a memory segment in the shared memory
2. then, the processes that wish to write into or read from the shared memory, have to attach the segment
3. a process that does not need to access the shared memory has to detach the segment
4. when no more processes are attached to the shared memory, one process may de-allocate the shared memory

creating or retrieving a shared memory segment

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int segment_id = shmget (shm_key, nb *
getpagesize (), IPC_CREAT | S_IRUSR | S_IWUSR);
```

- shm_key is the IPC key
- getpagesize() is a function that returns the size of the memory pages of this OS
- S_IRUSR | S_IWUSR grants read and write permissions to the owner

attachment of a shared memory segment

- To make the shared memory segment available, a process must use shmat, "SHared Memory ATtach."

char* shared_memory = (char*) shmat (segment_id, 0, 0);

1. Pass it the shared memory segment identifier SHMID returned by shmget.
2. The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address.
3. The third argument is a flag, which can include the following:
   1. SHM_RND indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don't specify this flag, you must page-align the second argument to shmat yourself.
   2. SHM_RDONLY indicates that the segment will be only read, not written.

- If the call succeeds, it returns the address of the attached shared segment. Children created by calls to fork inherit attached shared segments; they can detach the shared memory segments, if desired.

detachment of a shared memory segment

- When you're finished with a shared memory segment, the segment should be detached using **shmdt** ("SHared Memory DeTach").
- Pass it the address returned by **shmat**.
- If the segment has been deallocated and this was the last process using it, it is removed.
- Calls to exit and any of the exec family automatically detach segments.
- Removing a share memory segement is achieved like this:
  shmctl (segment_id, IPC_RMID, 0);

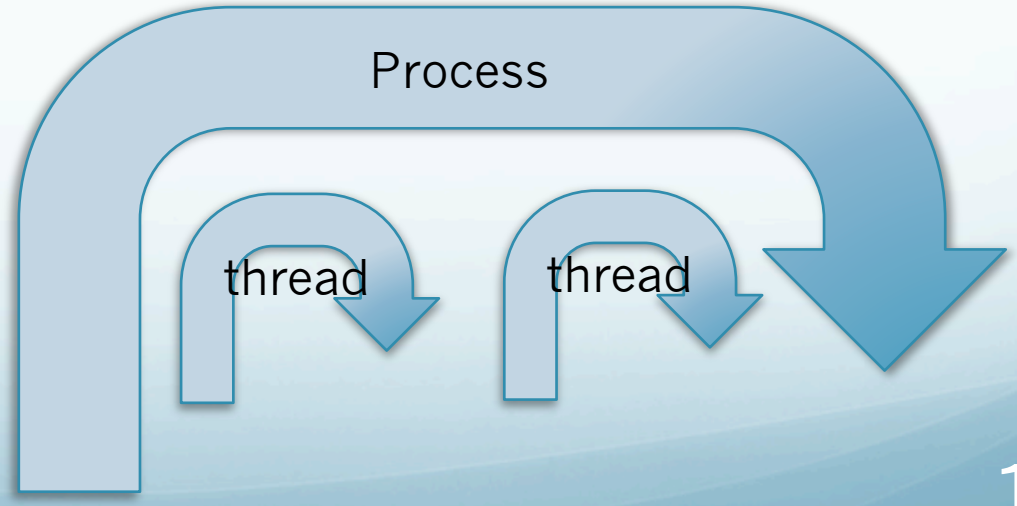
Exercise 12

- Write 2 programs:

1. a writer that:
   1. creates the shared memory segment
   2. attach it
   3. write "Hello world" in the shared memory segment
   4. detach the shared memory segment
2. a reader that:
   1. attach the shared memory segment
   2. read and display the content of the segment
   3. detach it
   4. de-allocate it

POSIX threads

- According to *Blaise Barney, Lawrence Livermore National Laboratory*
  - Threads exist within a process and uses the process resources.
  - A thread has its own independent flow of control as long as its parent process exists and the OS supports it.
- Of course, if the OS is not multi-tasked, you will not have threads.



Benefits of threads

- A thread duplicates only the essential resources it needs to be independently schedulable.
- A thread may share the process resources with other threads that act equally independently (and dependently)
- A thread is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer (semaphores).

Creating a thread

- You have to include **pthread.h**
- **Link with** *–pthread*
- **pthread_create** accepts 4 parameters:
  1. a pointer to **pthread_t**
  2. a pointer to a structure of particular attributes. You may pass NULL to use the default attributes for a thread.
  3. the pointer to the function to call when the thread is launched
  4. a pointer to a parameter (or a structure of parameters) that will be passed to the called function

Example of a thread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! thread id = %ld\n", tid);
   pthread_exit(NULL);
}
int main (int argc, char *argv[])
{
            pthread_t thread;
            long t = 1;
            int rc;
            printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&thread, NULL, PrintHello, (void *)t);
    if (rc){
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Closing a thread

- You have to explicitly exit a thread to terminate it. **pthread_exit**(NULL);
- the **main** function of your process runs in the main thread. You do not have to create it, but you have to exit it when this is not the only thread.

Simple thread synchronization with join

- The **pthread_join**(thread) subroutine blocks the calling thread until the specified thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to **pthread_exit**().
- A joining thread can match one **pthread_join**() call. It is a logical error to attempt multiple joins on the same thread.
- **Example**:

```
void * status;
pthread_join(thread, &status);
printf("Main: finished join with thread %ld having a status of
%ld\n",t,(long)status);
```

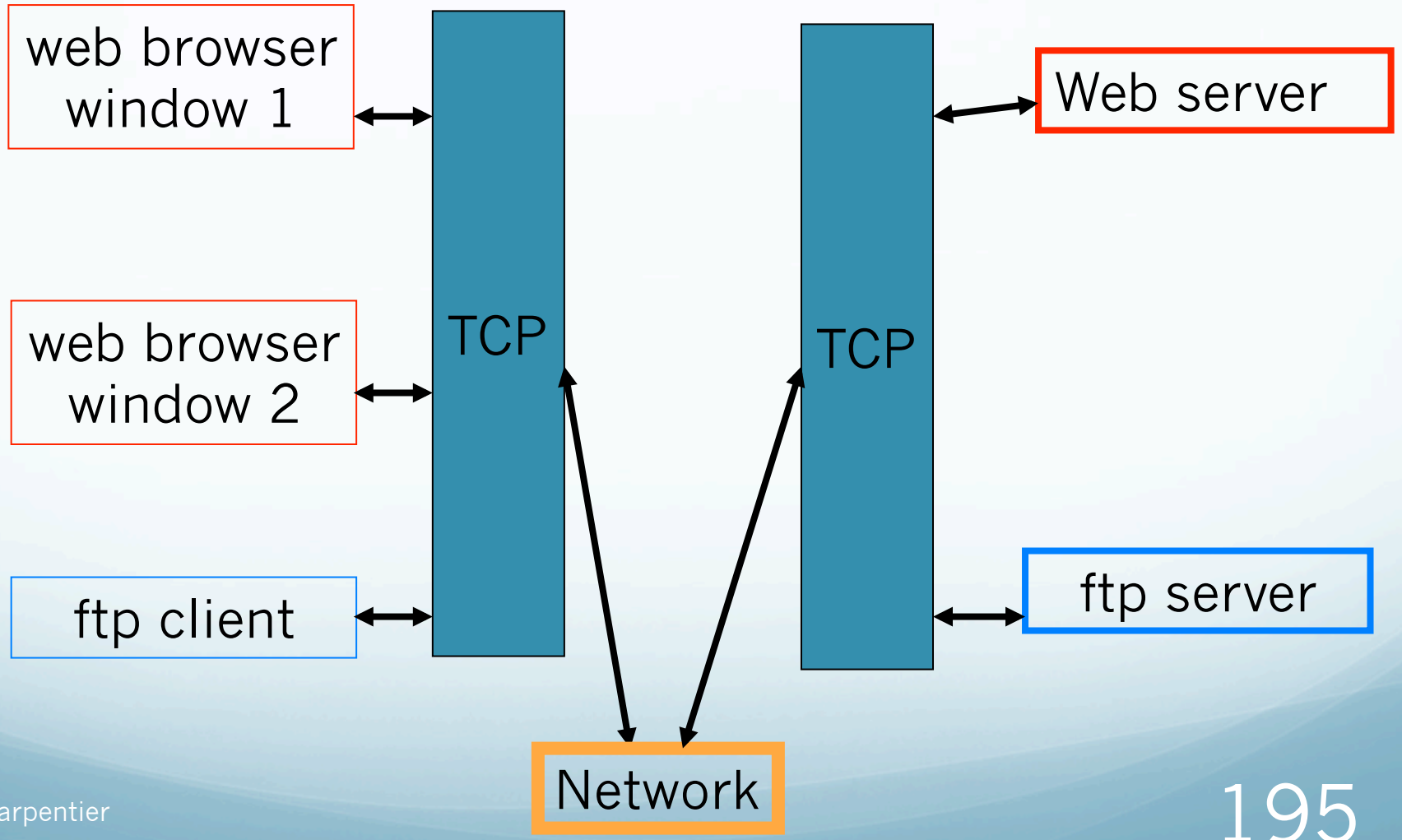
Network programming

- Processes may communicate over the network using IP protocols like:
  - TCP sockets
  - UDP sockets
  - RPC (Remote Procedure Call)
  - ...
- These protocols may be used on such media:
  - Ethernet
  - WiFi
  - Bluetooth
  - USB
  - NFC, ...

TCP // UDP

- TCP communication is like a phone call:
  - the client should be connected to the server before sending data
  - data loss (corrupted or missing packet) is notified and the packets are transmitted again
  - flow control, the server notifies the client if it is ready or not to receive
  - end of transmission is negotiated
- UDP is like the post mail:
  - no connection => allow multicast and broadcast
  - the client sends data to the server without checking if it is available or ready
  - no safety =>
    - the network should be reliable (LAN)
    - or the application should check the data itself
  - 15 times faster than TCP

Rôle of sockets: application multiplexing



sockets, example of http request

sockets, example of http response

Apache Web server

Firefox

socket destination = 32496
socket source = 80

@IP destination = 194.2.232.105
@IP source = 194.2.232.252

Client-server

- On the server side, the program should create a socket that listen to incoming packets on a particular local IP address (or any) and well-known port number.
- The port number should be unique on this server.
- On the client side, the program creates a socket which number is provided by the OS in order to avoid conflict with other socket numbers.
- The client connect (TCP) or send data (UDP) to the server IP address and port.

TCP socket creation

#include <sys/socket.h>
int id =socket(AF_INET, SOCK_STREAM, 0);

1. The 1st parameter (the domain) AF_INET specifies that we create an IPv4 socket
2. The 2nd parameter (the type) SOCK_STREAM specifies that it is a TCP socket
3. The last parameter 0 (the protocol) is used in case of multiple protocols for the same transport layer. For TCP, there is only one protocol.
4. The return code is an integer that identifies the socket.

TCP server

1. create the **socket** as in the previous slide. Call the id *listenerId*.
2. initialize the data structure that contains the IP address an port number
3. **bind** the socket with this IP address and port number
4. set the number of connections at the same time (**listen**)
5. in an endless loop, wait and accept incoming connections. An incoming connection releases the **accept**() function that returns the id of the connection.
6. in a child process or a thread,
   1. read the incoming message,
   2. process it and return the result to the client
   3. close the connection

address and port structure

```
struct sockaddr_in servaddr, clientaddr;
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
servaddr.sin_port=htons(5588);
```

- **bzero** initializes with 0 each field of the structure
- **INADDR_ANY** means that the server will listen requests coming from any network interface (Ethernet, WiFi, ...). If you just want to accept one interface, provides the local IP address of this interface.
- **5588** is the port number on which our socket will filter incoming messages. I just took this number because it was not already used by other applications.
- **htonl** and **htons** are type conversion functions for hostnames, IP addresses, services and port numbers.

bind and listen

bind(listenerId, (struct sockaddr *) &servaddr, sizeof(servaddr));

listen(listenerId, 1024);

- **bind()** has 3 parameters:
  1. the id of the socket that you just created
  2. a pointer to the IP and port structure
  3. the size of the structure
- **listen**() sets the number of max incoming request at the same time (1024)

accepting a connection

```
struct sockaddr_in clientaddr;
socklen_t clientlen;

int connectionId = accept(listenerId, (struct sockaddr *)
&clientaddr, &clientlen);
```

- the structure clientaddr will store the IP address and the socket number of the client that performed this request.
- We may immediately process the request, but meanwhile, others requests will be blocked in the incoming queue.
- We'd better either fork a child process or creates a thread to process the request and immediately come back to the accept instruction.
- The connection id will be duplicated in the child process, but you have to pass it as a parameter when you create a thread.

reading and writing over the network

- you may use the standard streams functions read and write to exchange data between the client and the server.

```
char mesg[1000];
read(connectionId, mesg, 1000);

write(connectionId, mesg, strlen(mesg) + 1);
```

- do not forget to close the connection when finished:

```
close(connectionId);
```

TCP client

1. create a socket the same way as on the server side
2. initialize the structure for the IP address and port number of the client
3. bind the socket
4. initialize the structure for the IP address and port number of the server
5. connect to the server
6. exchange data with the server
7. when finished, close the connection with the server

Creating and binding the socket

```
int fd = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in myaddr;      /* our address */

memset((char *)&myaddr, 0, sizeof(myaddr));
myaddr.sin_family = AF_INET;


// bind to any address

myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0); // any available port

bind(fd, (struct sockaddr *)&myaddr, sizeof(myaddr))
```

Connection to the server

```
struct sockaddr_in servaddr;
struct hostent *hp;

memset((char*)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(5588);

hp = gethostbyname(host);
memcpy((void *)&servaddr.sin_addr, hp->h_addr_list[0],
hp->h_length);

connect(fd,(struct sockaddr *)&servaddr,sizeof(servaddr));
```

Exchanging data with the server

```
/* sending our value to be added to the server counter */
write(fd, argv[2], sizeof(argv[2]));
/* waiting for the new value of counter*/
read(fd, recvBuff, sizeof(recvBuff)-1);
printf("new value of counter = %f\n", atof(recvBuff));
printf("disconnected\n");
shutdown(fd, 2);    /* 2 means future sends & receives are disallowed */
return 0;
```

Exercise 13

1. Write a TCP client that takes 2 parameters on the command line
   1. the hostname or IP address of the server
   2. a numeric value
2. The client connects to the server and send the numeric value translated into a string of characters.
3. The server will add it to an accumulator and return the new value to the client that should display it and terminates.

Exercise 14

- Write the TCP server that implements the counter.
- Reuse your client to test the server.

UDP socket

- Creating a UDP socket is like a TCP socket except the type parameter: SOCK_DGRAM
- DGRAM stands for DataGRAM (the IP packet)

int fd = socket(AF_INET, SOCK_DGRAM, 0);

UDP server

- Initializing the address and port structures:

```
struct sockaddr_in myaddr;

memset((char *)&myaddr, 0, sizeof(myaddr));
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(5588);
```

- binding the socket:

```
bind(fd, (struct sockaddr *)&myaddr, sizeof(myaddr));
```

UDP client

- Initializing the address and port structures, the only difference is to port number 0:

```
struct sockaddr_in myaddr;

memset((char *)&myaddr, 0, sizeof(myaddr));
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0);
```

- binding the socket is the same as for the server:

```
bind(fd, (struct sockaddr *)&myaddr, sizeof(myaddr));
```

Exchanging data

- Reading and writing into the socket is different from TCP:
  - the client does not connect to the server
  - the server does not accept connections
- you should provide the destination address and port each time you send a message
- you should retrieve the source address and port each time you receive a message to be able to send a response
- => you have to use dedicated functions to exchange data:
  - **sendto**()
  - **recvfrom**()

recvfrom

```
unsigned char buf[2048];
struct sockaddr_in remaddr;

socklen_t addrlen = sizeof(remaddr);

int recvlen = recvfrom(fd, buf, BUFSIZE, 0, (struct
sockaddr *) &remaddr, &addrlen);
```

sendto

```
char buf[2048];
char *server = "172.16.233.209";

struct sockaddr_in remaddr;
int slen=sizeof(remaddr);

memset((char *) &remaddr, 0, sizeof(remaddr));
remaddr.sin_family = AF_INET;
remaddr.sin_port = htons(5588);
inet_aton(server, &remaddr.sin_addr);

sendto(fd, buf, strlen(buf), 0, (struct sockaddr *)
&remaddr, slen);
```

Exercise 15

1. write a UDP client that sends a message (a string of characters) to my UDP server.
2. write a UDP server that display the message sent by you client (just change the IP address of the server).

Multicast

- With the UDP protocol, you are able to use multicast or broadcast destination addresses.
  - broadcast: any system will receive the message
  - multicast: only systems belonging to a group will receive the message
- But broadcast packets are not routed and may be blocked by our switches at ISEP.
- The multicast systems must join a multicast group: share the same multicast IP address.
- The senders do not have to belong to the multicast group. They just have to send their message to the multicast address.

Joining a multicast group

```
#define EXAMPLE_GROUP "239.0.0.1"

struct sockaddr_in addr;
int sock = socket(AF_INET, SOCK_DGRAM, 0);
bzero((char *)&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(EXAMPLE_PORT);
int addrlen = sizeof(addr);

bind(sock, (struct sockaddr *) &addr, sizeof(addr));

struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = inet_addr(EXAMPLE_GROUP);
mreq.imr_interface.s_addr = htonl(INADDR_ANY);

setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
&mreq, sizeof(mreq));
```

Direct hardware access

- The Unix kernel abstracts the handling of devices.
- Most devices can be accessed through the file system just like files.
- They are located in the /dev folder
- 2 kinds of device files:
  - character mode (c) like tty (terminals)
  - block mode (b) like disks, CD, ...
- The devices are managed in the kernel by device drivers.

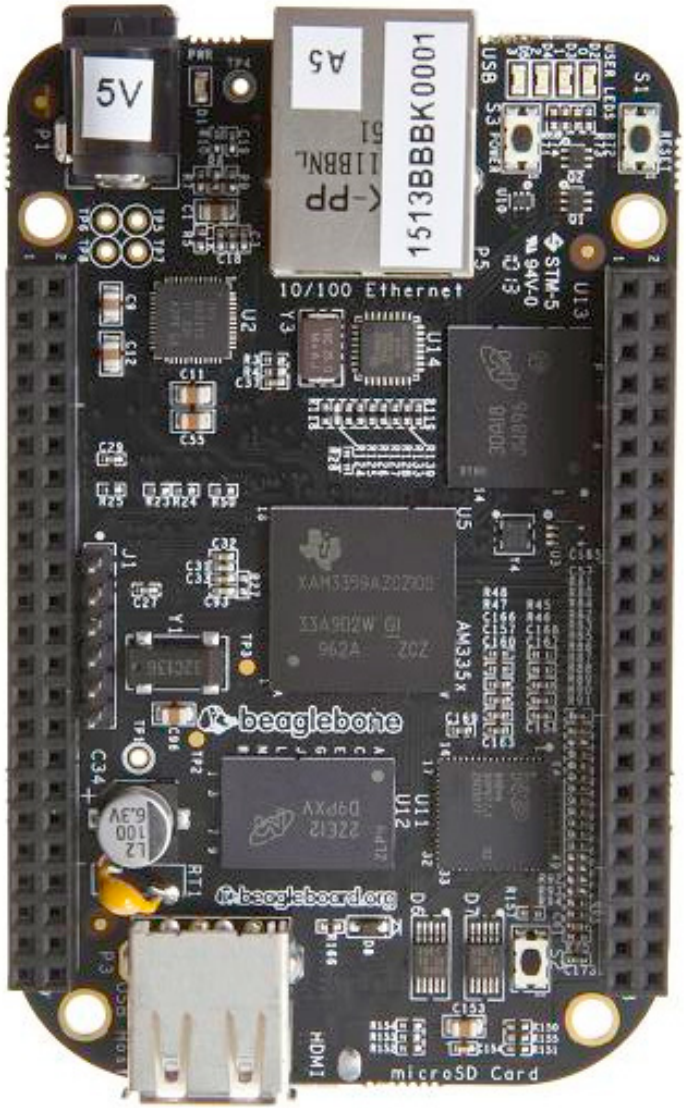
Major/minor numbers

- A device driver is identified by 2 numbers:
  - the major number identifies the type of device, USB for example
  - the minor number identifies a instance of the device USB port 1 for example
- Example on a beaglebone:

```
crw-------  1 root root      4,   0 Jan  1  2000 tty0
crw-rw----  1 root tty       4,   1 Jan  1  2000 tty1
brw-rw---T  1 root floppy  179,   0 Jan  1  2000 mmcblk0
brw-rw---T  1 root floppy  179,   8 Jan  1  2000
mmcblk0boot0
brw-rw---T  1 root floppy  179,  16 Jan  1  2000
mmcblk0boot1
brw-rw---T  1 root floppy  179,   1 Apr 23 20:23 mmcblk0p1
brw-rw---T  1 root floppy  179,   2 Jan  1  2000 mmcblk0p2
```

Beaglebone

- PC on card using TI ARM processor
- USB, Ethernet, HDMI, SD card
- multiple extension cards (caps)
- OS Linux or Android
- C, python, javascript
- IP address on USB: 192.168.7.2



access to LEDs

- We will just try to show how to access the hardware through te file system for LEDs.
- The device files for leds are located in /sys/class/leds
- Here we have fo directories, one for each LED:

beaglebone:green:usr0 -> blinking (heartbeat)
beaglebone:green:usr1 -> microSD card access
beaglebone:green:usr2 -> CPU processing
beaglebone:green:usr3 -> eMMC access

- we will use the number 1 or 3

LED control

- each LED provides access to different controls:

```
-rw-r--r-- 1 root root 4096 Apr 23 21:16 brightness
lrwxrwxrwx 1 root root    0 Apr 23 21:16 device -> ../../../
gpio-leds.8
-r--r--r-- 1 root root 4096 Apr 23 21:16 max_brightness
drwxr-xr-x 2 root root    0 Apr 23 21:16 power
lrwxrwxrwx 1 root root    0 Jan  1  2000 subsystem -
> ../../../../../class/leds
-rw-r--r-- 1 root root 4096 Apr 23 21:16 trigger
-rw-r--r-- 1 root root 4096 Jan  1  2000 uevent
```

switching the LED on/off

- we just have to put "0" or "1" into the brightness special file.

```
char ledPath[1024];

sprintf(ledPath, "/sys/class/leds/beaglebone:green:usr
%d/brightness", ledNum);

int fid = open(ledPath, O_WRONLY);

write(fid, "0", 1);
```

Control/Command/Monitor

- Industrial and home applications are made of 3 parts:
  1. the control that provides a user interface (today web and mobile) to activate equipment
  2. the command that performs the requests received from the control
  3. the monitor that retrieves, displays and analyse data received from sensors (web or mobile GUI)

Light monitoring using BeagleBone