

# Introduction

## Présentation du cours ASR2

- ▶ On se concentre sur le S et R de ASR
  - ▶ Prérequis : C et dans une moindre mesure ASR1.
  - ▶ Organisation : TP/TD le vendredi à 8h en salle Europe (001), cours le mercredi à 15h45 (en amphi ou 001).
  - ▶ Projet(s) avec rendus intermédiaires (50% de la note)
  - ▶ Examen final (50% de la note) : en salle machine (rendu papier + rendu binaire)
  - ▶ Questions, suggestions, bugs : [michael.rao@ens-lyon.fr](mailto:michael.rao@ens-lyon.fr)
- 
- ▶ Vendredi 27 à 8h : cours en salle Europe (001)
  - ▶ Mercredi 1er fevrier : pas de cours

# Références

## Livres :

- ▶ Jean-Marie Rifflet et Jean-Baptiste Yunès, « UNIX : programmation et communication », Dunod, 2003.
- ▶ Andrew S. Tanenbaum et Herbert Bos, « Modern Operating Systems, 4th Ed », Pearson 2015.

## Les pages du manuel (RTFM!)

- ▶ `man <terme>` dans un terminal
- ▶ Attention : il y a plusieurs sections!

# Ce qu'il y aura dans le cours

Le cours est divisé en 3 grandes parties (à peu près équi-réparties)

- ▶ Programmation système
  - ▶ Descripteurs de fichiers, gestion de la mémoire, gestions des processus, signaux et communication inter-processus.
- ▶ Programmation multithreads
  - ▶ programmation en threads POSIX, un peu de théorie sur l'interblocage et l'ordonnancement
- ▶ Réseau
  - ▶ les couches TCP/IP, mécanismes de routage

À côté : un peu de shell, de deguggage, de bonnes pratiques et de sécurité.

## Buts et non-but

- ▶ Comprendre ce qui se passe entre la machine (le matériel, la partie A de ASR) et un processus (que vous allez programmer dans la vraie vie). Dans cette superposition de couches, le système d'exploitation (OS = Operating System) joue un rôle important.

Couches : Matériel / OS / (bibliothèques) / Processus

- ▶ On cherche aussi à comprendre ce qui se passe entre les processus (communication, synchronisation, réseau...)
- ▶ Connaître par coeur toutes les fonctions système n'est pas un but. Savoir se servir de la documentation pour trouver ce que l'on cherche en est un.
- ▶ On ne cherche pas, à notre niveau, à savoir concevoir et programmer un OS, juste à savoir opérer avec lui.
- ▶ On programmera en C, sous GNU/Linux. On essaiera de suivre au maximum la norme POSIX.

# Un OS : pourquoi ?

Que fait un système d'exploitation ?

- ▶ interface avec le matériel
- ▶ gestion de la mémoire
- ▶ gestion des périphériques
- ▶ gestion des interruptions
- ▶ gestion des système de fichiers
- ▶ ...

Ces taches sont très dépendantes du matériel, et souvent fastidieuses. L'OS est là pour nous simplifier la vie.

## Que fait un système d'exploitation « moderne » ?

- ▶ multi-tâches : plusieurs processus peuvent tourner en même temps
  - ▶ multi-utilisateurs : il peut y avoir plusieurs utilisateurs différents qui l'utilisent.
  - ▶ gestion utilisateurs, groupes d'utilisateurs, et droits.
  - ▶ gestion du réseau
- 
- ▶ essayer d'être "compatible" avec les autres OS : respect de normes.

# Systèmes "Unix"

Introduction d'Unix (1969-, K. Thompson & D. Ritchie...)

- ▶ Rompt avec les OS propriétaires, et les programmes monolithiques.
- ▶ Introduit conjointement avec le C (1969-, D. Ritchie & B. Kernighan)
- ▶ Rapidement, plusieurs branches de développement : Unix, BSD, et système propriétaires (Xenix, AIX, System V...)
- ▶ Assez rapidement, une volonté de normalisation : norme POSIX (1988-)
- ▶ Philosophie : être modulaire . Préférer des programmes simples qu'on peut composer via les entrées/sorties standards.

# GNU / Linux

- ▶ GNU : "GNU is Not Unix" : projet d'OS libre (1983- R. Stallman)
- ▶ Logiciel libre : code source disponible, que l'on peut modifier et redistribuer (licence GPL "GNU Public Licence", ou similaire)
- ▶ Linux : Un noyau (cœur de l'OS) Unix libre.
- ▶ Autres Unix libres : Minix, \*BSD, Hurd...

# Écosystème...

Un "écosystème" vivant dans un ordinateur :

- ▶ ensemble de processus en exécution
- ▶ interagissant entre eux
- ▶ utilisant les mémoires
- ▶ éventuellement utilisant des périphériques

Les processus n'interagissent pas directement avec le matériel, ils passent par l'OS pour y avoir accès.

# Mémoires

Deux types importantes de mémoire :

- ▶ la mémoire vive, "volatile"
  - ▶ non pérenne
  - ▶ accès très rapide.
- ▶ la mémoire "non volatile"
  - ▶ pérenne
  - ▶ disque dur mécanique, SSD, clef USB...
  - ▶ accès (beaucoup) plus lent

La mémoire non volatile est organisée sous forme arborescente, avec des fichiers et des répertoires : l'arborescence des fichiers.

## Plan approximatif des cours :

- ▶ Arborescence de fichiers / Shell
- ▶ Programmation système : bases, entrées sorties
- ▶ Mémoire
- ▶ Processus
- ▶ Communication inter-processus.
- ▶ Threads
- ▶ Réseau
- ▶ ...

# Arborescence de fichiers et Shell

# Shell

Shell : interface textuelle entre l'humain et l'OS.

Votre premier objectif : maîtriser le shell

Fonctionnement d'un shell

- ▶ prompt : attente d'une commande

commande [argument1] [argument2] ...

- ▶ on entre une commande, éventuellement avec arguments, et on appuie sur "entrée".
- ▶ le shell exécute la commande, puis rend la main quand la commande est terminée.
- ▶ pour quitter "proprement" un shell : `exit` (ou `ctrl+d` sur certains shells).

# Shell / terminal

Un shell se lance au travers d'un terminal.

Il existe plusieurs shells différents. Un des plus courant sous GNU/Linux est `bash`.

Permet de composer facilement des processus via les redirections d'entrée/sorties.

On peut faire des scripts en shell.

# L'arborescence des fichiers

Les fichiers sont organisés sous forme arborescente.

- ▶ La racine : /
- ▶ Deux principaux types de fichiers :
  - ▶ les fichiers standards = fichiers réguliers
  - ▶ les répertoires (ou dossiers).
  - ▶ (Il en existe d'autres : liens, fifo... à suivre)
- ▶ Chaque fichier possède :
  - ▶ un propriétaire et un groupe propriétaire
  - ▶ un ensemble de droits (lecture/écriture/exécution) pour l'utilisateur, le groupe, et le reste du monde.
  - ▶ une date de création, de modification, de lecture.
- ▶ Fichiers commençant par un point : fichiers cachés.

# L'arborescence des fichiers

Répertoires spéciaux :

- ▶ `..` : répertoire parent
- ▶ `.` : répertoire courant

Chemin de la racine à un fichier : chemin absolu

- ▶ `/home/mrao/Documents/cours.pdf`

Chemin du répertoire courant à un fichier : chemin relatif

- ▶ `Documents/cours.pdf`  
= `./Documents/cours.pdf`

# Organisation typique sous Unix/Linux

- ▶ /home : les répertoires des utilisateurs
- ▶ /root : le répertoire "home" du super-utilisateur
- ▶ /bin et /usr/bin les programmes (les "binaires");
- ▶ /sbin et /usr/sbin : les binaires système
- ▶ /lib et /usr/lib : librairies
- ▶ /usr : ressources système
- ▶ /etc : fichiers de configurations
- ▶ /dev : fichiers spéciaux (ressources, périphériques)
- ▶ /tmp : un répertoire pour les fichiers temporaires
- ▶ /var : données variables
- ▶ ...

## Les shell : premiers pas...

Le shell permet de naviguer dans l'arborescence de fichiers, modifier les droits, faire des opérations simples.

- ▶ `ls` : liste les fichiers
  - ▶ option `-a` : affiche également les fichiers cachés
  - ▶ option `-l` : format long (droits, taille, propriétaire...)
- ▶ `cd rep` : entrer dans le répertoire *rep*
  - ▶ `cd ..` : retour au répertoire parent

## Autres commandes de base

- ▶ `cat` : affiche un fichier
- ▶ `rm` : efface un fichier
- ▶ `cp` : copie un fichier
- ▶ `mv` : déplace (renomme) un fichier
- ▶ `mkdir/rmdir` : crée/efface un répertoire
- ▶ ...

# Un peu plus sur les droits des fichiers

```
mrao@meshuggah:~/test$ ls -la
total 32
drwxr-xr-x  4 mrao users 4096 dec.  29 22:40 .
drwx----- 61 mrao users 4096 jan v.  9 17:25 ..
-rwxr-xr-x  1 mrao users 6656 dec.  29 13:37 programme
-rw-r--r--  1 mrao users  173 dec.  29 13:37 programme.c
drwxr-xr-x  2 mrao users 4096 dec.  29 13:39 sousrep
mrao@meshuggah:~/test$
```

premier champ : un sous ensemble de drwxrwxrwx

- ▶ d : répertoire
- ▶ 1er triplet (rwx) : droits pour l'utilisateur (ici, mrao)
- ▶ 2eme triplet : droits pour les utilisateurs du groupe (users)
- ▶ 3eme triplet : droits pour le reste du monde
- ▶ r : droit de lecture
- ▶ w : droit d'écriture
- ▶ x : droit d'exécution (pour les répertoires : droit d'entrer)

Pour changer les droits : chmod

# Liens

Unix supporte des liens. Il y a deux types de liens, fondamentalement différents.

- ▶ Lien symbolique : un "pointeur" vers un autre fichier. Il s'agit d'un type de fichier spécial.  
Commande shell : `ln -s`. Appel système : `symlink`
  
- ▶ Lien physique : fichier correspondant à la même zone sur le disque qu'un autre.  
Commandes shell : `ln`, `link`. Appel système : `link`

## Autres fichiers spéciaux

- ▶ Fichier périphérique (device file).  
Correspond à un périphérique  
Généralement situé dans `/dev/`
  
- ▶ Fichiers tubes (ou fifo).  
Pour créer un fichiers tube : `mkfifo`  
(On reparlera de tubes au moment de la communication inter-processus.)

# Un peu plus sur les système de fichiers

- ▶ L'arborescence des fichiers est un "patchwork" de systèmes de fichiers.
- ▶ Un système de fichier correspond généralement à une partition sur un disque sur.
- ▶ plusieurs types de systèmes de fichiers : FAT, EXT, NTFS...
- ▶ commandes : `mount`, `umount`, `df`

# Utilisateurs et groupes

Chaque utilisateur a :

- ▶ un nom (une chaîne de caractère)
- ▶ un numéro (UID = User IDentifier)
- ▶ un ou plusieurs groupes
- ▶ un répertoire HOME, généralement : /home/<user>
- ▶ un mot de passe, stocké de façons hashée.

root est le “super-utilisateur”. Il a tous les droits. Son UID est 0.

Chaque groupe a un numéro (GID = Group IDentifier).

# Les processus

Chaque processus a :

- ▶ un numéro (le PID = Process IDentifier)
- ▶ un père, généralement le processus qui l'a lancé.
- ▶ un utilisateur (généralement, celui qui l'a lancé)
- ▶ une zone mémoire qui lui a été attribué. Il peut en demander plus au système.
- ▶ certains processus peuvent être en attente.
- ▶ une entrée standard, et une sortie standard et une sortie erreur.
- ▶ À sa fin, un processus renvoie un code retour : un entier, généralement 0 s'il n'y a pas d'erreur, et  $\neq 0$  sinon.

## Lancer des commandes/processus dans un shell

- ▶ Certaines commandes sont interprétées directement par le shell, les builtin (comme `cd`). D'autres correspondent à des programmes exécutables (généralement situés dans `/bin/` ou `/usr/bin/`).
- ▶ S'il le trouve, il l'exécute (le processus se lance). Sinon il renvoie un message d'erreur.
- ▶ Pour lancer un programme dans le répertoire courant il faut spécifier le répertoire avant le nom du programme.
- ▶ Les programmes sont cherchés dans les répertoires listés dans la variable d'environnement `PATH`

# Entrées/sorties standard

Quand un processus s'exécute dans un terminal :

- ▶ l'entrée standard est (par défaut) l'entrée du terminal (le clavier)
- ▶ la sortie standard est (par défaut) affichée dans le terminal.
- ▶ la sortie erreur est (par défaut) affichée dans le terminal.

Le shell permet de facilement rediriger ces entrées/sorties.

## Rediriger les E/S standards

- ▶ `commande > fichier` : redirige la sortie standard de la commande vers le fichier (écrase le fichier)
- ▶ `commande >> fichier` : redirige la sortie standard de la commande vers le fichier (rajoute à la fin du fichier)
- ▶ `commande 2> fichier` : redirige la sortie erreur de la commande vers le fichier
- ▶ `commande1 | commande2` : la sortie standard de `commande1` sera redirigée vers l'entrée standard de `commande2`

## Rediriger les E/S standards

- ▶ `tee fichier` : copie entrée standard sur la copie standard et *fichier*
- ▶ `commande < fichier` : l'entrée standard sera lue depuis le *fichier*
- ▶ `commande << EOF` : le shell va lire l'entrée standard, jusqu'à ce qu'il lise EOF. Ce qui est lu sera envoyé dans l'entrée standard de commande.

## Quelques commandes utiles

- ▶ `sleep x` ; attend x seconde (utile pour les scripts)
- ▶ `echo` : affiche les arguments
- ▶ `less` : permet de se déplacer dans le texte
- ▶ `head/tail` : affiche le début/fin de l'entrée
- ▶ `sort` : trie les ligne
- ▶ `grep` : afficher les lignes correspondant à un motif donné
- ▶ `sed` : fait des recherches / remplacements
- ▶ `awk` : un truc qui fait mieux que `grep` / `sed`, mais encore plus compliqué.

## Voir/gérer les processus :

Commandes shell pour voir/gérer les processus :

- ▶ `ps` affiche la liste des processus  
exemple : `ps faux`
- ▶ `top` affiche la liste des processus dynamiquement
- ▶ `kill pid` : tue un processus de PID *pid* (on en reparlera dans la partie "Signaux")

# Variables du shell

- ▶ Le shell manipule des variables.
- ▶ Exemples : HOME, USER, PATH
- ▶ Affecter une variable :
  - ▶ `VARIABLE=affectation`
- ▶ déréférencer une variable : la faire précéder par `$`
  - ▶ Ex : pour afficher une variable : `echo $VARIABLE`
- ▶ `set` : affiche toutes les variables

Certaines variables sont persistantes : les variables d'environnement. Elles seront transmises aux fils

- ▶ `export` : exporte la variable (les rend persistantes)
- ▶ `env` : affiche toutes les variables d'environnement.

# Variables spéciales du shell

- ▶ `$?` : code retour de la précédente commande
- ▶ `$$` : PID du shell
- ▶ `$!` : PID du dernier processus lancé en arrière plan
  
- ▶ `~` : interprété par le shell comme le répertoire HOME
- ▶ `~user` : interprété par le shell comme le répertoire HOME de l'utilisateur user

# Jokers et échappements

- ▶ \* dans un nom de fichier : n'importe quelle chaîne de caractère
- ▶ ? : exactement un caractère

Pour qu'un caractère spécial ne soit pas interprété par le shell

- ▶ \ : exemple `echo \*`
- ▶ ' : exemple `echo '*'`
- ▶ " : exemple `echo "*"`
  - ▶ le shell interprète les \$ et certains \ dans des chaînes entre "

Le caractère "espace" peut être aussi échappé, pour ne pas séparer les arguments

## Processus en arrière plan

- ▶ commande `&` lance un processus, mais le shell n'attend pas la fin du processus pour rendre la main.
- ▶ `ctrl + z` : stoppe un processus
- ▶ `jobs` : liste les taches (jobs) en cours d'exécution dans le shell
- ▶ `bg` : passe une tache en arrière plan (similaire à `&`)
- ▶ `fg` : passe une tache en premier plan (le shell rend la main au job)
- ▶ `%i` : identifie le job numéro *i* du shell.

## nohup

- ▶ Si on termine un shell, tous ses jobs seront arrêtés.
- ▶ Pour qu'un processus survive aux déconnexions, à la mort de son père, on peut le lancer précédé de la commande `nohup`

## Scripts : enchaînement et composition des commandes

- ▶ `commande1 ; commande2`  
execute `commande1`, puis `commande2`
- ▶ `( listecommandes )`  
crée un groupement de commande
- ▶ `commande1 'commande2'`  
la sortie de `commande2` est donnée en argument à `commande1`

Sur certains shells, on peut également faire ceci :

```
commande1 $(commande2)
```

## Scripts : enchaînement et composition des commandes

- ▶ `commande1 && commande2`  
execute `commande1`, puis `commande2` si `commande1` réussit (i.e. renvoie 0)
- ▶ `commande1 || commande2`  
execute `commande1`, puis `commande2` si `commande1` échoue
- ▶  
`if condition ; then commande2 ; else commande3 ;fi`

## Scripts : tests

`test expression` permet de tester une expression conditionnelle.

Note : sur certains shells, c'est équivalent à `[ expression ]`

`expression` construite avec `( )` `&&` `||` `!` et des expression élémentaires.

Expression élémentaire (exemples) :

- ▶ tester si un fichier existe : `-e fichier`
- ▶ tester si un fichier est un répertoire : `-d fichier`
- ▶ tester si un fichier est un fichier régulier : `-f fichier`
- ▶ tester si un fichier est lisible : `-r fichier`
- ▶ tester si deux chaînes de caractères sont égales :  
`chaine1 = chaine2`
- ▶ tester si expression numériques sont égales :  
`chaine1 -eq chaine2`

## Scripts : évaluer une expression

`expr expression` permet d'évaluer une expression

*expression* construite avec ( ) + - \* / % = >= ... et des expressions élémentaires (entiers...)

Attention aux échappements : `expr \"( 2 + 3 \) \" * 5`

Sous bash on peut utiliser directement `$((expression))`

## Scripts : boucles

```
while condition ; do commandes ; done
```

- ▶ Ex : `while true ; do date ; sleep 1; done`

```
for v in liste; do commandes ; done
```

- ▶ entre `do` et `done`, `v` est une variable

- ▶ Ex :

```
for f in *wav; do lame $f 'basename $f wav'mp3 ; done
```

- ▶ `seq a b` : tous les entiers entre `a` et `b`.

Voir également : `break`, `continue`, `until`, `case`

## Scripts : fichiers scripts

```
#!/bin/bash  
for i in "$*" ; do  
    echo $i  
done
```

- ▶ `#!` : shebang : dit au système quel interpréteur utiliser
- ▶ `$1` : 1er argument, `$2` : 2eme argument ...
- ▶ `$*` : tous les arguments
- ▶ `$#` : nombre d'arguments