

# Introduction

LIFSE - Systèmes d'exploitation

L. Gonnord, N. Louvet, M. Moy, G. Pichon, F. Zara

12 janvier 2024

- 1 Introduction
- 2 Rôles et bases d'un système d'exploitation
  - Interface avec le matériel
  - Organisation des ressources
  - Sécurité
  - Interface avec l'utilisateur
  - Résumé
- 3 Programmation et environnement de travail
  - Remarques sur la programmation C/C++
  - Gestion des erreurs
  - Paramètres des programmes
  - Compilation/exécution
  - Les variables d'environnement
- 4 Conclusion

# Plan

- 1 Introduction
- 2 Rôles et bases d'un système d'exploitation
  - Interface avec le matériel
  - Organisation des ressources
  - Sécurité
  - Interface avec l'utilisateur
  - Résumé
- 3 Programmation et environnement de travail
  - Remarques sur la programmation C/C++
  - Gestion des erreurs
  - Paramètres des programmes
  - Compilation/exécution
  - Les variables d'environnement
- 4 Conclusion

# Organisation de l'UE

Page du cours : <https://nlouvet.gitlabpages.inria.fr/lifse/>

## MCC

- 60% contrôle continu
- 40% examen final

## Contenu

- TD : referrez-vous à ADE
- TP : voir votre intervenant et la salle sur TOMUSS

## Point sur les « prérequis » de l'UE

- Avoir des notions de base : architecture des ordinateurs, et réseaux.
- « Se débrouiller » sur un système GNU/Linux
  - ↳ ne pas avoir peur de la ligne de commande !
- « Se débrouiller » pour programmer en C/C++
  - ↳ ne pas hésiter à se documenter, via des sources fiables,
  - ↳ bien comprendre un code que l'on veut réutiliser.
- Compilation d'un code
  - ↳ savoir compiler en ligne de commande (g++ ou clang++),
  - ↳ comprendre les étapes de la compilation,
  - ↳ savoir écrire un Makefile simple.

C'est une UE technique : il faut comprendre des concepts et pratiquer !

# Utilité d'un système d'exploitation

## A quoi sert un système d'exploitation ?

- gérer différents matériels
- gérer de multiples tâches et plusieurs utilisateurs

## Pourquoi est-ce intéressant d'étudier le fonctionnement d'un système d'exploitation ?

- comprendre le fonctionnement permet de mieux l'utiliser, de mieux programmer (notion d'efficacité du code)
- dans la vraie vie, on essaie de rester loin du système la plupart du temps, de nombreuses bibliothèques efficaces existent déjà

## Buts de l'UE :

- **théorie**  $\rightsquigarrow$  présenter les concepts de base des systèmes d'exploitation,
- **programmation**  $\rightsquigarrow$  vous familiariser avec la prog. POSIX en C,
- **administration**  $\rightsquigarrow$  vous rendre plus autonomes sous GNU/Linux.

# Utilité d'un système d'exploitation

## Qu'est-ce qu'un système d'exploitation ?

Littéralement, *ce qui permet d'utiliser la machine*. Quatre grands rôles :

- **Interface entre applications et matériel** (e.g., gestion des périphériques)
- **Organisation des ressources** (e.g., processeurs, mémoire)
- **Sécurité** (e.g., des données, du matériel)
- **Interaction avec le ou les utilisateurs** (e.g., comptes, droits, installation)

Différents types de systèmes d'exploitations, pour différents usages :

- Systèmes « généralistes », multi-utilisateurs et multi-tâches :  
GNU/Linux (Debian, Ubuntu, Redhat...), Windows, Mac OS X...
- Pour l'embarqué (contraintes sur l'utilisation des ressources) :  
Windows Embedded, Android...
- Temps-réel (contraintes d'échéance) : RTLinux, QNX, FreeRTOS.

# Interface avec le matériel

## Que se passe-t-il lorsque l'on branche une clé USB ?

- La clé est traitée comme un disque amovible.
- On peut formater ce disque, lire et écrire des fichiers dessus.
- Pourtant une clé USB n'est pas vraiment un disque dur, ni un périphérique SCSI (*Small Computer System Interface*) !

Le seul élément courant que l'utilisateur manipule :  
possible installation d'un driver (en tant qu'administrateur)

## Fonctionnalités

En tant qu'interface, un système d'exploitation doit fournir :

- à l'utilisateur/programmeur
  - ▶ une vue unifiée du matériel (mémoire, disque, carte réseau, ...)
  - ▶ des objets abstraits (fichiers, répertoires, processus, threads, ...)
- au matériel
  - ▶ gestion des ressources (conflit d'accès, ordonnancement),
  - ▶ protection contre la mauvaise utilisation,
  - ▶ une gestion des évènements (interruptions, exceptions).

Cela impose des vérifications, des attentes et des accès indirects :

- au moins deux modes de fonctionnement :
  - ▶ *mode utilisateur* (exécution sans accès direct aux ressources),
  - ▶ *mode noyau* (pas de protection accès direct aux ressources),
- un moyen de passer de l'un à l'autre : les *appels systèmes*,
- un *mécanisme d'interruption* et de *mise en attente* (files de priorités).

## Noyau (ou “superviseur”)

- Privilégié *matériellement* par le processeur
- Le noyau vérifie ce que les utilisateurs veulent faire (droits d'accès par ex.) lors des appels systèmes
- Une erreur de programmation peut avoir des effets catastrophiques (accès direct à toute la mémoire noyau, “kernel panic” qui plante la machine)

## Root (utilisateur privilégié en *userland*)

- Pas le noyau, a un accès privilégié au noyau
- Privilégié *logiciellement* par le noyau
- Peut faire des choses qu'un utilisateur classique n'a pas le droit de faire
- Erreur de programmation grave mais souvent rattrapable (cloisonnement mémoire par processus, avec un peu de chance “segmentation fault” ne fait planter qu'un processus)

↪ dès qu'il le peut, le noyau délègue les tâches à des processus utilisateurs (démons).

## Un appel système est :

- une fonction fournie par le système,
- que tout programme peut utiliser,
- qui est exécutée en mode noyau.

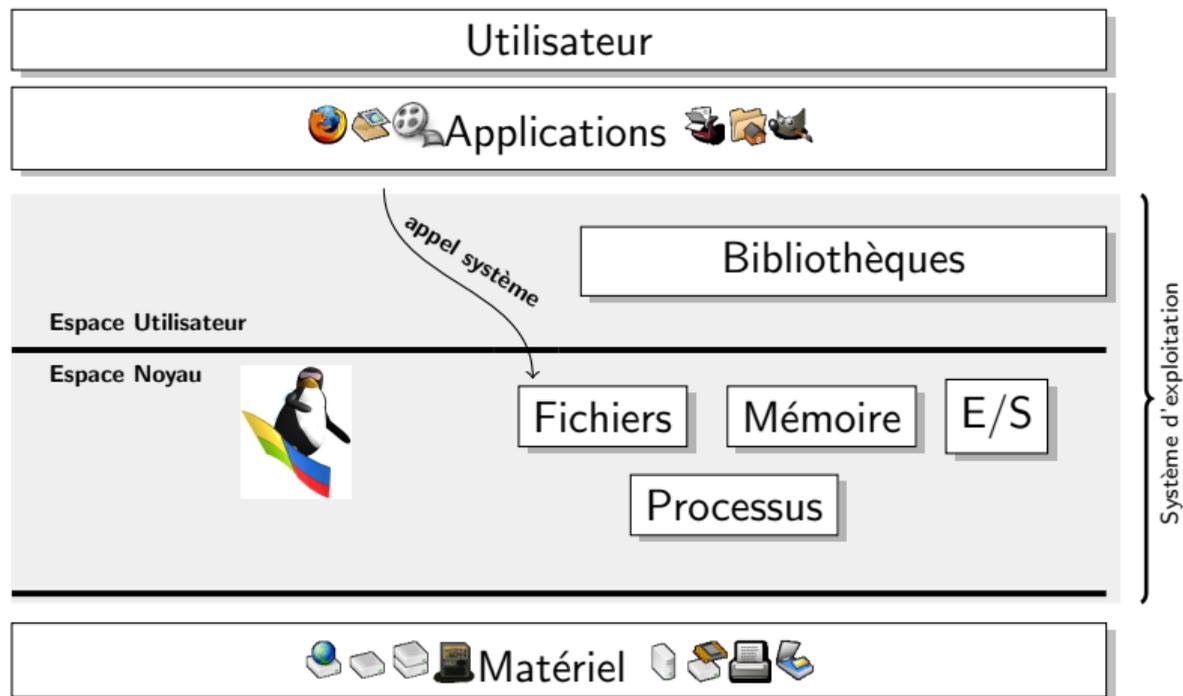
Ce sont des ponts entre le mode utilisateur et le mode noyau.

Par exemple, pour lire, la fonction C `scanf` utilise l'appel système `read`.

En C++, l'opérateur `<<` utilise l'appel système `write` pour écrire.

Les appels système :

- font des vérifications,
- sont toujours susceptibles de générer une erreur,
- prennent du temps !



## Toutes les opérations critiques sont des appels systèmes :

- toutes les lectures / écritures
- la création de processus (voir CM3, CM4) et de sockets (CM5)
- la gestion des threads (voir LIFPCA l'an prochain)

Ex. Que se passe t-il si on veut afficher à l'écran un gros fichier ?

```
strace cat gros_fichier.mp4 > /dev/null
```

- `strace` trace tous les appels systèmes
- une redirection vers `/dev/null` permet de ne pas afficher le fichier sur le terminal : pratique pour ne pas polluer notre sortie standard !
- on voit une suite d'appels systèmes `read()` pour lire le fichier et `write()` pour faire la redirection

# Organisation des ressources

## Les ressources proposées par la machine :

- au moins un processeur,
- de la mémoire vive (RAM),
- de la mémoire de masse (disque dur),
- des périphériques d'entrées/sorties.

## Les besoins des utilisateurs ou des programmes :

- Accès aux ressources : que se passe-t-il lorsque deux *processus* demandent la même chose en même temps ?
- Organisation des données : comment les retrouver rapidement ?
- Gestion des évènements : que faire si l'utilisateur insère une clé usb ?

Il faut arbitrer, et se rappeler quel processus à obtenu quoi, maintenir des listes de demandes, pour assurer un accès équitable aux ressources.

## Ressources les plus importantes : les processeurs et la mémoire

Le noyau doit décider

- quelle tâche devient active : c'est l'*ordonnancement* ;
- quelles données sont en mémoire vive : *swapping* (ou *va-et-vient*).

Pour gérer les ressources, le noyau doit reprendre la main : quand ?

Cela a généralement lieu à chaque passage en mode noyau :

- lors de la gestion des exceptions ; par exemple :
  - ▶ accès mémoire non autorisé (*segmentation fault*),
  - ▶ instruction interdite (*illegal instruction*).
- lors d'une interruption matérielle, par un périphérique ou un timer.
- lors des interruptions logicielles pour les appels système
  - ▶ à chaque fois que le programmeur fait des entrées/sorties par exemple,
  - ▶ cela explique aussi le coût des appels systèmes !

## Quelques ordres de grandeur

Support	Taille	Temps accès
RAM	$\simeq 4 \text{ Gio}$	10 ns
Disque dur	$\simeq 1 \text{ Tio}$	10 ms
Réseau	$\simeq \infty$	10-100 ms

Les Entrées / Sorties sont critiques pour le système :

- On essaie de limiter leur nombre
- La plupart sont réalisées sans utiliser le processeur
- Une gestion fine est fondamentale pour obtenir de bonnes performances

↔ Quand vous écrivez du texte, le système en profite pour travailler !

Ex : Appel à la fonction `sleep` dans un code.

# Sécurité

Que se passe-t-il si :

- on fait tellement de calculs que le processeur dépasse 100 degrés ?
- on interrompt une écriture de disque brutalement ?
- vous essayez de lire le répertoire d'un autre utilisateur ?

Puisque le système est une interface entre les applications et le matériel, il a aussi un rôle de protection :

- du matériel
  - ▶ monitoring (surveillance de la température)
  - ▶ actions automatiques (gestion de l'énergie)
  - ▶ zones critiques (actions qui ne doivent pas être interrompues)
- des données
  - ▶ systèmes de fichiers journalisés (protection contre l'arrêt brutal)
  - ▶ utilisateur, droits, authentification
- des programmes : séparation des tâches, communication, virtualisation

# Interface avec l'utilisateur

Pour les utilisateurs, le système doit :

- permettre d'exécuter des programmes et de gérer des données.
- différencier les utilisateurs avec
  - ▶ une base de données des utilisateurs,
  - ▶ un système d'authentification,
  - ▶ des droits ajustables pour chaque utilisateurs.
- être configurable :
  - ▶ base de données des configurations,
  - ▶ interface de configuration,
  - ▶ fournir des outils pour administrer la machine.
- avoir un système d'installation de programmes :
  - ▶ comment installer ?
  - ▶ notion de packages.
- permettre la programmation (e.g., processus, client/serveur).

## Résumé

Le système d'exploitation peut être abordé selon 3 points de vues :

- ➊ **Utilisation et programmation** : les outils fournis pour mieux utiliser les possibilités des ordinateurs (prog. multiprocessus par exemple)
- ➋ **Conception et théorie** : les problèmes posés par les systèmes et les moyens de les résoudre.
- ➌ **Administration** : comment configurer et gérer le système ?

On va essayer dans l'UE d'aborder chacun de ces points de vue.

### À retenir dans cette section

- rôles du système d'exploitation,
- notion de mode d'exécution (utilisateur ou noyau),
- les appels systèmes et leur rôle.

# Plan

- 1 Introduction
- 2 Rôles et bases d'un système d'exploitation
  - Interface avec le matériel
  - Organisation des ressources
  - Sécurité
  - Interface avec l'utilisateur
  - Résumé
- 3 Programmation et environnement de travail
  - Remarques sur la programmation C/C++
  - Gestion des erreurs
  - Paramètres des programmes
  - Compilation/exécution
  - Les variables d'environnement
- 4 Conclusion

## Remarques sur la programmation C/C++

- Vous avez déjà appris beaucoup de choses en LIFAPI, LIFAPSD, peut-être en LIFAPCD : il faut continuer !
- En 2024, on peut se permettre d'utiliser certains apports du C++11 : développé à partir de 2003, standard ISO/CEI 14882 publié en 2011.
- Apport intéressant du C++11, la *Standard Template Library* (STL) :
  - ▶ fournit des conteneurs sous la forme de class templates,
  - ▶ sites de référence (avec les classes et méthodes de la STL) :  
<https://en.cppreference.com/>  
<https://www.cplusplus.com/>
- Le but n'est pas de faire un cours sur le C++, juste de donner des points d'entrée... Dans la suite :
  - ▶ un exemple avec `vector<T>` (tableaux dynamiques),
  - ▶ un autre avec `list<T>` (listes doublement chaînées),
  - ▶ un dernier avec `string` (remplacement des tableaux de `char`).

```
#include <iostream>
#include <vector>

int main(void) {
    std::vector<int> tab;

    std::cout << "Entrez des entiers (< 0 pour arrêter) : "
              << std::endl;
    while(true) {
        int n;
        std::cin >> n;
        std::cout << "lu : " << n << std::endl;
        if(n < 0) break;
        tab.push_back(n);
    }

    std::cout << "Vos " << tab.size() << " nombres : "
              << std::endl;
    for(unsigned int i = 0; i < tab.size(); i++)
        std::cout << tab[i] << " ";

    return 0;
}
```

```
#include <iostream>
#include <list>

using namespace std;

int main(void) {
    list<float> l;

    cout << "Entrez des flottants (< 0 pour arrêter) :" << endl;
    while(true) {
        float x;
        cin >> x;
        cout << "lu : " << x << endl;
        if(x < 0) break;
        l.push_front(x);
    }
    cout << "Vos " << l.size() << " nombres :" << endl;
    while(!l.empty()) {
        cout << l.front() << " ";
        l.pop_front();
    }
    cout << endl;
    return 0;
}
```

La classe `string` permet de se passer des tableaux de `char` (`char[]` ou `char*`) pour la gestion des chaînes de caractères :

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
    string ch;
    cout << "Entrez une chaîne de caractères : ";
    getline(cin, ch);
    cout << "Vous avez entré : " << ch << endl;
    return 0;
}
```

On peut avoir besoin d'accéder au tableau de caractères sous-jacent :

```
string ch;
cout << "Entrez une chaîne de caractères : ";
getline(cin, ch);
cout << "Vous avez entré : " << flush;
write(STDOUT_FILENO, ch.data(), ch.length());
```

Problème avec les *templates* : ils compliquent les messages d'erreur...

Par exemple :

```
string chaine = "coucou";  
cout << chaine + 4 << endl;
```

Donne de copieux messages d'injures :

```
template.cpp:9:18: erreur : no match for <<operator+ >>  
  (operand types are <<std::__cxx11::string  
  {aka std::__cxx11::basic_string<char>} >>and <<int >>  
  ...  
  /usr/include/c++/6.3.1/bits/stl_iterator.h:341:5:  
  note : candidate: template ...  
  ...
```

Le compilateur tente de nous aider : à la ligne 9 du fichier, il ne trouve pas d'opération + entre un string et un int.

Tout le même, vous avez beaucoup moins de chance d'introduire des bugs qu'en reprogrammant les structures de données de base !

## errno

Le fichier d'en-tête `<errno.h>` définit la variable entière `errno`, qui est renseignée par les appels système.

### Utilisation

- Toujours vérifier la valeur de retour d'un appel système
- Si la valeur est égale à `-1` il y a eu une erreur
- La variable `errno` nous donne des informations sur cette erreur

Ex. Ouverture d'un fichier qui n'existe pas en lecture

```
int fd = open("FICHER", ...); // spoil CM 2
if (fd == -1){
    cerr << "Erreur : " << strerror(errno) << endl;
}
```

Le programme écrit sur `stderr` : Erreur : Aucun fichier ou dossier de ce type

## argc et argv

En C/C++ on peut récupérer les paramètres passés au programme en ligne de commande via les paramètres de la fonction `int main(int argc, char *argv[])`.

### Signification

- `argc` donne le nombre d'arguments passés **+1**
- `argv[0]` contient le nom de l'exécutable (d'où le **+1**)
- `argv[1]`, ..., `argv[argc-1]` contiennent chacun des paramètres passés

# Compilation

La compilation est le fait de traduire votre code pour en faire un programme exécutable.

Schématiquement, elle comporte 4 phases :

- *précompilation* : utilisation de directives pour modifier le source.
- *compilation* : traduction du code en langage d'assemblage.
- *assemblage* : traduction du code en langage machine.
- *édition de liens* : résolution des liens avec les fonctions utilisées.

À chaque phase il y a des options et des erreurs différentes. . .

Un compilateur comme g++ combine ces quatre phases, donc il n'est pas toujours facile de savoir lors de quelle phase se produit une erreur. . .

*Mais il faut le comprendre pour pouvoir corriger !*

Supposons que l'on veuille compiler `prog.cpp` en un exécutable `prog`.

**Précompilation.** Le compilateur modifie le code source d'après les ordres donnés par les directives :

- `#include <fichier>` = recopie le fichier ici ;
- `#define NOM VAL` dans la suite remplace `NOM` par `VAL`
- `#ifdef NOM ... #endif` = ne conserve le code que si `NOM` est défini.

Pour arrêter après le préprocessing : `g++ -E prog.cpp -o prog_tmp.cpp`

**Compilation.** Le compilateur traduit le source en langage d'assemblage, et lorsqu'il détecte un problème, il génère un message d'erreur :

- les messages sont là *pour vous aider* ;
- corriger une erreur suppose *un choix éclairé du programmeur* ;
- si un algorithme était capable de corriger, il le ferait !

Pour arrêter après la compilation : `g++ -S prog_tmp.cpp -o prog_tmp.s`

**Assemblage.** L'assembleur transforme le code en langage d'assemblage en un code en langage machine, appelé *code objet, dans lequel les adresses des fonctions ne sont pas résolues.*

Pour arrêter après l'assemblage : `g++ -c prog.cpp ~> prog.o`

**Edition des liens.** L'éditeur de lien, *ld* en l'occurrence, se débrouille pour retrouver chaque fonction appelée, et place son adresse dans le code objet.

Quand vous voyez apparaître un message d'erreur en provenance de *ld*, c'est souvent que l'éditeur de lien n'arrive pas à trouver une fonction : il faut indiquer dans quel code objet ou bibliothèque se trouve la fonction.

Par défaut, le compilateur s'arrête après l'édition des liens : `g++ prog.cpp -o prog`

**Exemple.** Dans `tst.cpp`, on veut utiliser une fonction `void msg(void)` :

```
int main(void) {  
    msg();  
    ...  
}
```

`g++ -o tst tst.cpp` donne une erreur *de compilation* :

```
tst.cpp: In function 'int main()':  
tst.cpp:4:3: error: 'msg' was not declared in this scope
```

Du coup, on modifie le source on ajoutant une déclaration de la fonction :

```
void msg(void); // déclaration  
int main(void) { ...  
}
```

`g++ -o tst tst.cpp` r le maintenant *lors de l' dition des liens* :

```
/tmp/ccL8xKgc.o : Dans la fonction "main"  
tst.cpp:(.text+0x5) : r f rence ind finie vers "msg()"  
collect2: error: ld returned 1 exit status
```

En fait, le code objet de `msg()` se retrouve dans `utils.o` :

on peut compiler avec `g++ -o tst tst.cpp utils.o`

**Exemple.** On peut indiquer à l'éditeur de liens d'aller chercher du code dans des fichiers objets et dans des bibliothèques. Par exemple :

```
g++ -o prog main.o biblioprof.o -L/opt/lib/ -lpthread -llibopt
```

Cela signifie : « fabrique moi s'il te plaît l'exécutable prog :

- en prenant le code objet des fonctions dans `main.o` et `biblioprof.o`,
- mais aussi dans les bibliothèques `libpthread.so` et `libopt.so`,
- qui sont dans le répertoire `/opt/lib` ou dans un répertoire habituel. »

Les « endroits habituels » dépendent de la configuration de votre système (`ldconfig`, `/etc/ld.so.conf...`).

Certaines bibliothèques sont parcourues d'office par l'éditeur de liens ; c'est le cas de la bibliothèque C++ standard utilisée par `g++` ; sur mon ordi :

```
/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
```

## En résumé.

Une option utile du *préprocesseur* :

- `-D NOM` pour définir le nom dans le source ;

Options utiles du *compilateur* :

- `-g` rendre possible le *débogage* ;
- `-std=c++11` prévenir que le code est du C++11 ;
- `-Wall` demander le plus possible de messages d'aide ;
- `-c` s'arrêter à la compilation pour faire un fichier objet.

Options utiles pour l'*édition des liens (ld)* :

- `-LNOM_REPERTOIRE` : ajoute un répertoire où chercher des fichiers ;
- `-lNOM_LIBRAIRIE` : nom d'une bibliothèque à utiliser (`libNOM.so` typiquement)
- `NOM.o` : fichier objet à utiliser.

De toute façon, il faut pratiquer !

## Les variables d'environnement

Les processus sont séparés, mais fonctionnent dans un environnement :

- ils sont exécutés depuis un **working directory**,
- ils **appartiennent à un utilisateur** et héritent de ses droits, ...

Ces informations sont transmises par des variables héritées de leur processus père : les *variables d'environnement*.

Exemples :

- **PATH** : liste des répertoires où sont cherchés les exécutables.
- **USER** et **HOME** : nom et répertoire personnel de l'utilisateur.

Commandes utiles (Bash) :

- **echo \$VAR** pour afficher la variable VAR !
- **env** pour lister toutes les variables d'environnement.
- **export VAR=valeur** pour définir une variable d'environnement.

# Plan

- 1 Introduction
- 2 Rôles et bases d'un système d'exploitation
  - Interface avec le matériel
  - Organisation des ressources
  - Sécurité
  - Interface avec l'utilisateur
  - Résumé
- 3 Programmation et environnement de travail
  - Remarques sur la programmation C/C++
  - Gestion des erreurs
  - Paramètres des programmes
  - Compilation/exécution
  - Les variables d'environnement
- 4 Conclusion

Nous avons :

- passé en revue les quatre grands rôles d'un système d'exploitation : **interface, organisation, sécurité, interaction avec les utilisateurs.**
- introduit des techniques de programmation C++ (STL),
- fait des rappels sur la compilation d'un programme C/C++.

Buts de l'UE :

- **théorie**  $\rightsquigarrow$  présenter les concepts de base des systèmes d'exploitation,
- **programmation**  $\rightsquigarrow$  vous familiariser avec la prog. POSIX en C,
- **administration**  $\rightsquigarrow$  vous rendre plus autonomes sous GNU/Linux.

Que faire pour réussir ?

- **pratiquer**  $\rightsquigarrow$  programmer les exemples des CM et TD, terminer les TP,
- **se documenter**  $\rightsquigarrow$  sur le web par exemple, pages de manuel. . .
- **progresser en « débrouillardise »**  $\rightsquigarrow$  c'est important aussi !