Introduction

LIFASR5 - Systèmes d'exploitation

L. Gonnord, N. Louvet

24 janvier 2022

Plan

Introduction

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 periphériques)
- 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 type de systèmes d'exploitations, pour différents usages :

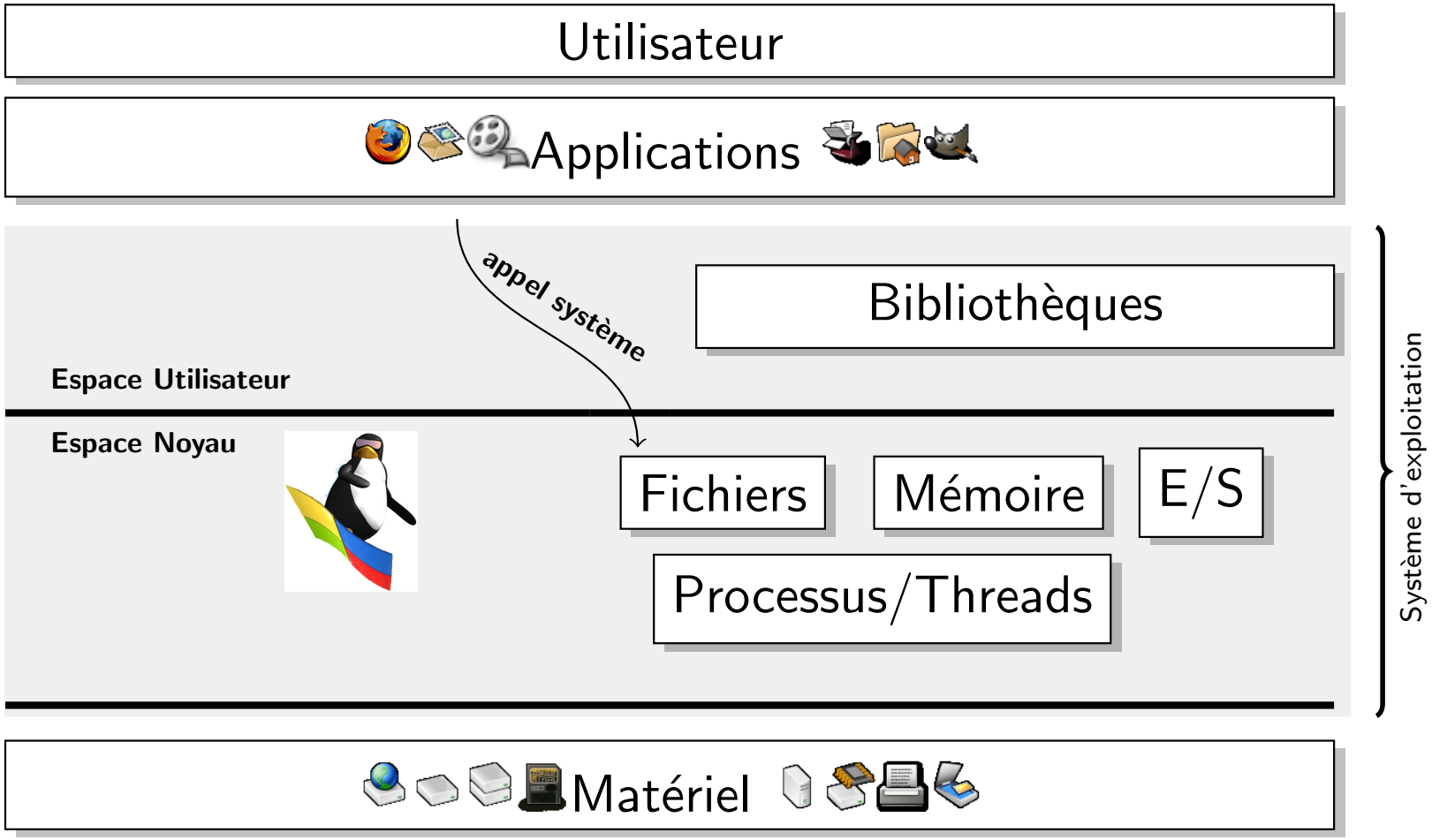
- Systèmes « généralistes », multi-utilisateurs et multi-tâches : GNU/Linux (Debian, Ubuntu, Redhat...), Windows, Mac OS X...
- Pour l'embarqué (contraintes sur l'utilisation des ressources) : Windows Embedded, Android...
- Temps-réel (contraintes d'échéance) : RTLinux, RTAI, FreeRTOS.

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 le formater, lire et écrire des fichiers ;
- 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)


Utilisateur
Applications
appel système
Bibliothèques
Espace Utilisateur
Espace Noyau
Fichiers
Mémoire
E/S
Processus/Threads
Système d'exploitation
Matériel

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).

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 !

Organisation des ressources

Les ressources proposées par la machine :

- au moins un processeur,
- de la mémoire vive,
- de la mémoire de masse,
- 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 insére 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 !

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

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 :

1. **Utilisation et programmation :** les outils fournis pour mieux utiliser les possibilités des ordinateurs (programmation multiprocessus, multithread, réseau par exemple)
2. **Conception et théorie :** les problèmes posés par les systèmes et les moyens de les résoudre.
3. **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

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 !

Remarques sur la programmation C/C++

- Vous avez des déjà appris beaucoup de choses, en LIFAP1, LIFAP3, peut-être en LIFAP4 : il faut continuer !
- En 2021, 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 chainées),
  - 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 !

Compilation

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

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` $\rightsquigarrow$ `prog.o`

Edition des liens. L'éditeur de lien, *ld* en l'occurence, 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` râle maintenant *lors de l'édition des liens* :

```
/tmp/ccL8xKgc.o : Dans la fonction "main"
tst.cpp:(.text+0x5) : référence indéfinie 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 bilioprof.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 `bilioprof.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)* :

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

De toute façon, il faut pratiquer !

Plan

Nous avons :

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

Buts de l'UE :

- théorie $\rightsquigarrow$ présenter les concepts de base des sytèmes 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 !