# 2: Système d'exploitation et fichiers

Martin Quinson



# Épisode précédent

- Un ordinateur, c'est de la mémoire (ie, des cases mémoires où on range des nombres) et un processeur
  - Notion d'opcode, cycle fetch/decode/execute
- Le langage C est un bon langage pour comprendre la machine
  - Il offre une "abstraction" très représentative de la réalité sous-jacente
  - De toute façon, on est obligé de comprendre la machine pour parvenir à programmer en C
- Le C est un langage préhistorique
  - Faut tout faire soit-même, et ça pique un peu

# I) Langage C (suite)

## I.1) Survivre en C

- Erreurs classiques qu'il faut savoir reconnaître
  - syntax error
  - missing main function
  - redefinition d'une fonction
  - segfault
- Comment réagir: don't panic; lire le message d'erreur; ne pas lire le second message d'erreur (le compilo est aux fraises)
- Aide toi et le ciel t'aidera. En particulier, il faut écrire le plus lisible possible, et tenir compte des warnings

## I.2) Code illisible en C

- Ca sert à rien de chercher à faire le code le plus illisible possible, car on n'a pas le niveau de l'IOCCC (Intl Obfuscated C Code Contest). On regardera ensemble les 3 premières entrées de la liste suivantes (en fonction du temps dispo):
  - 2012 endoh1 (simulation fluide) avec les entrées pour-out, column et clock
  - 2000 dhyang (quine affichant le visage de Saitou Hajime puis aku soku san, ie sin swift slay)
  - 2004 arachnid (simulateur labyrinthe déplacement: adws)
  - 2014 deak (calcul de fractale)
  - 2014 maffido (mario)
  - 2015 mills2: un programme de compression très très court
  - 2015 mills1: un flappy bird ascii

## I.3) printf et scanf

- Relire le premier code de la feuille à ce sujet; le reste est bien expliqué sur la page man
- Formatages:
  - %d: int
  - %f: double (pour scanf, il faut %lf)
  - %c: char
  - %s: string (char*)
  - %%: caractère %
  - %03.2f -> 003.14 (3 caractères à gch, complété par des 0, et 2 à droite)

## I.4) Lire et écrire dans des fichiers

- Deux interfaces disponibles dans Unix
  - Haut niveau (printf/scanf): performances et interface raisonnables, moins de contrôle. C'est ce qu'on va voir ensemble
  - Bas niveau (read/write):contrôle parfait, donc meilleures perfs si bon usage (et bien pire si mal utilisé), et interface inamicale.

- Interface fprintf/fscanf:
  - Comme printf/scanf mais on précise sur quel descripteur de fichier on veut lire ou écrire
  - Tester si le fichier est arrivé à la fin: feof()
  - Lire un caractère est piégeux, faut bien vider le retour chariot que l'humain a mis en plus. Cf code proposé.
  - Lire ligne à ligne est carrément difficile sans `getline`, même si on ne comprend pas tout dans le code proposé.
  - Ouvrir / fermer un fichier (manque feuille de pompe)

```
FILE *fd = fopen("fichier", "r"); // r ou w ou r+
if (fd == NULL) {
  fprintf(stderr, "bobo\n");
  exit(2);
}
// Lecture du fichier, cf feuille de pompe
fclose(fd);
```

### I.5) Paramètres en ligne de commande

- argv est un tableau de chaines de caractères, une cellule pour chaque paramètre de la ligne.
- argv[0] = nom du binaire
- argv[1] = premier paramètre (if any)
- argv[argc-1] = dernier paramètre
- argv[argc] = NULL

## II) Système d'exploitation

### II.1) Qu'est ce qu'un système d'exploitation

- Citez moi des exemples d'OS: Linux, Windows, Mac, Android, FreeBSD, blablabla
- Combien d'OS existent? Autant que d'élèves qui ont fait le projet du MIT ou Stanford
- Quel est le lien entre Linux et UNIX? Et entre MacOSX et UNIX
- Quel est l'OS le plus utilisé sur terre ?
  - Desktop: windows sans aucun doute. Serveurs: linux. Au total: Minix qui est embarqué dans la secure zone des processeurs Intel.
- Pourquoi étudier Linux dans cette diversité?
  - Je connais bien
  - C'est ouvert
  - C'est fait en C, ce qui tombe bien dans ce module, avouez
  - le noyau windows va bientôt s'arrêter quand le WSL3 sera encore mieux intégré. Comme Mac OSX mais avec un Unix moderne dessous.

### II.2) À quoi sert l'OS?

- Historiquement, ct juste un ensemble de fonctions mises en commun, mais maintenant, bcp plus de choses

- C'est le programme entre les programmes et la machine, dont l'objectif est de:
  - Protéger le matériel des applications, et les applications les unes des autres
  - Protéger les applications les unes des autres (sous Dos ou M99, un bug d'un programme = crash généralisé)
  - Adapte (simplifie/uniformise) l'interface matérielle
    * les pilotes traduisent les requetes API de façon spécifique au matériel
    * Sous Dos, les jeux spécifiaient les cartes réseaux utilisables
  - Virtualise les ressources: multiplexage CPU, swap mémoire

## II.3) Fonctionnement d'un OS (SII only)

### II.3.a) Protéger les périphériques des applications

- Motivation
  - Sous Dos, on avait des virus détruisant le lecteur de disque.
  - Le malware Stuxnet a été déployé en 2012 pour détruire les centrifugeuses à uranium iraniennes (cf WP)
- Principe: **interposition et médiation**. Les applications ne peuvent pas accéder directement au matériel, doivent demander à l'OS
  - quand on lit/écrit dans les plages mémoire correspondant aux périphériques, le CPU vérifie qu'il est en mode protégé
  - **mode protégé** = registre sur un bit. Valeur 1 => mode protégé; valeur 0 => mode applicatif
  - Ce n'est pas être super-utilisateur sur linux, c'est encore autre chose (cf. plus loin)
  - Comment on fait pour mettre ce bit à 1 ? On n'a pas le droit par soit-même, il faut déclencher une interruption
- **interruption**: quand le CPU détecte un evt particulier, il passe en mode protégé et exécute une fonction enregistrée au démarrage
  - En gros, chaque interruption du CPU redonne la main à une fonction du système d'exploitation.
    * D'où l'importance pour l'OS d'être celui qui s'exécute en premier, pour enregistrer les gestionnaires d'interruption. Si un virus arrive avant l'OS (en prennant le contrôle de la séquence de boot avant l'OS), tout est fichu.
  - Si une appli tente d'écrire sur un périphérique directement, le CPU réveille l'OS après une interruption, et il est probable que l'OS termine l'application
- **appel système**: pour faire proprement, l'application doit demander à l'OS
  - l'application écrit sa requête et les paramètres dans un endroit spécifique de la mémoire
  - l'application déclenche une interruption signifiant "cher OS, j'ai besoin d'un service"
    * sur x86, y'a une instruction assembleur `INT` qui signifie `interrupt`, et Linux lui passe le paramètre 80
  - le CPU se réveille suite à cette interruption. Il passe en mode protégé, puis passe l'exécution à l'OS
  - l'OS vérifie la validité des paramètres (l'application a le droit de faire cet accès)
  - l'OS fait le travail demandé par l'appli. Au besoin, écrit le résultat dans la mémoire de l'application
  - l'OS repasse en mode non-protégé, et retourne l'exécution à l'application.

### II.3.b) Protéger CPU contre le blocage par app

- Problème: accaparement du CPU par un programme donné qui empêche de rendre le contrôle à l'OS.

  - Soit à cause d'un bug (boucle infinie)
  - Soit programme malicieux, au sens anglais, i.e. qui cherche à nuire au système
- Principe: **préemption** Je ne donne que ce que je peux reprendre.
- Solution: une interruption donnée est déclenchée 1000 fois par seconde. Cela redonne la main à l'OS
- Donc 1000 fois par seconde, l'OS a l'occasion de changer le programme en cours d'exécution si nécessaire
  - Pb de performance maîtrisé si le gestionnaire chargé est suffisament rapide. Qques pourcents de perte de perf.
- multiplexage CPU: une seule chose à la fois, mais suffisament vite pour que plusieurs applis s'exécutant en alternance donnent l'impression de s'exécuter en parallèle
  - Le scheduling est un pb de recherche amusant, même s'il est un peu limité au coeur du CPU car faut aller vite sans bcp contexte
  - les tâches interactives et les tâches de fond sont traitées très différent par linux.
  - On schedule les bouts d'applis distribuées, aussi, et c'est un pb scientifique plus riche
- Changement de contexte: remplacement du context d'exec par un autre.
  - Sauvegarde de tous les registres (= du contexte) qque part en mémoire
  - Lecture de l'autre contexte ailleurs en mémoire
  - Changement de PC

### II.3.c) Protéger les applications les unes des autres

- Pb: protéger la mémoire : on veut empêcher une appli de lire/écrire dans la mémoire des autres
  - vol de mot de passe, ou crash de l'autre appli
  - Cela vient en plus de la protection contre l'accaparement du CPU, bien sûr
- Principe: **interposition** pas d'accès direct à la mémoire. **médiation** généralisée
  - Chaque application ne manipule que des adresses virtuelles (de 0 à MAXINT), et n'a aucune idée des adresses mémoire physiques
  - C'est une protection efficace de la mémoire des autres processus (programme en cours d'exec), puisque l'appli ne peut même pas nommer la mémoire des autres
  - C'est d'ailleurs aussi une protection du matériel, puisqu'on ne peut pas accéder aux adresses physiques servant d'interface aux périphériques
  - Il y a quelque part en mémoire une table d'association (processus; plage mémoire virtuelle) -> (plage mémoire physique)
- Gros problème de perf en vue, si chaque accès mémoire demande encore plus de temps
  - La solution est d'ajouter du matériel dédié. Le MMU (mem managmt unit) est une zone du silicium du CPU en charge de faire ces translations à toute vitesse
- Ceci explique d'ailleurs un message d'erreur classique en C: le SEGFAULT (erreur de segmentation)
  - cela veut dire que le programme a accédé à une zone mémoire virtuelle qui ne correspondait à aucune entrée dans la table du MMU.
  - Pour l'OS, l'accès est invalide puisqu'il sait pas quoi faire. Donc l'appli a fait une faute, donc peine de mort: l'appli est terminée...

### II.3.d) Adapter l'interface

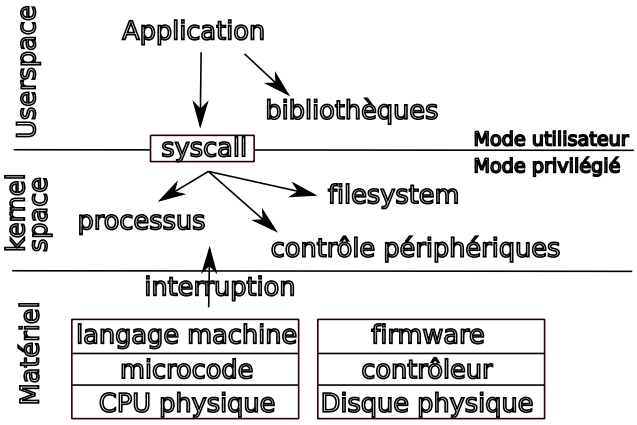
- Puisqu'on intercepte toutes les requêtes de l'appli aux périphériques, on peut facilement offrir une interface commune à chaque catégorie de matériel

- l'OS a besoin d'un bout de code qui sait traduire les requêtes haut niveau en accès direct à la mémoire, bas niveau. C'est le pilote (driver en anglais)
  - Distribué avec le matériel y'a 10 ans (sur un beau ptit CD inutile)
  - Inclu dans le noyau (Linux, c'est la majorité des lignes de code de Linux)
  - Téléchargé sur internet (mais attention à la sécurité, ce code s'exécute DANS le noyau, en mode protégé. D'ailleurs, micro-noyau pour éviter tout pb du genre)

### II.3.e) Virtualiser les ressources

- C'est l'art de faire croire aux applis qu'on a plus de ressources qu'en réalité.
  - Laisse à penser qu'il y a plus de ressources
- Le multiplexage CPU est une forme de virtualisation puisque les applis ont l'impression de s'exécuter en //
- L'interposition mémoire permet de virtualiser la mémoire, aussi. Chaque appli peut nommer $2^{64}$ octets en mémoire virtuelle, mais l'ordi bien plus limité.
  - Quand la mémoire physique commence à manquer, certaines pages mémoires d'application sont déplacées sur disque et retirées de la mémoire vive.
  - Cela fait de la place pour plus de mémoire
  - Quand on a besoin des pages passées sur disque, c'est très très lent. Il faut donc choisir avec précaution.
  - **swap**: L'espace disque qui stocke des pages qui débordent de la RAM. **swapping** action de déplacer des choses sur disque.
- Machine virtuelle: un programme qui fait croire aux autres programmes qu'ils sont sur une vraie machine alors que pas du tout.
  - Bcp de technos. La plus simple est la réécriture de l'assembleur des programmes (QEMU fait ça).
    - légère : INT en assembleur (appel systeme) est remplacé par un appel de fonction
    - lourde : tout est réécrit, ce qui permet d'exécuter du code ARM sur un CPU x86
  - Demande d'aide du processeur: les processeurs modernes offrent des services de virtualisation, aussi. Mais je sais plus comment

### II.3.f) Vue d'ensemble de l'OS

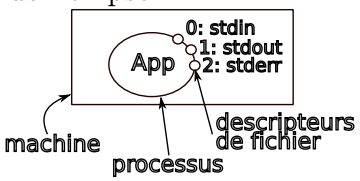



## II.4) Design d'UNIX

- UNIX est un OS dominant depuis 1970, grâce à qques bonnes idées de design

### II.4.a) Tout est un fichier (ou y ressemble, au moins)

- Utilisateur : stdin /stdout /stderr
  - Dans tout le cours, on utilisera la même représentation simple ci-dessous. Les machines (ordinateurs) sont des boites, les programmes qui s'exécutent dedans sont des ellipses, et les descripteurs de fichier sont des petits ronds à la surface de l'ellipse.




- Autres programmes: socket réseau, tubes
  - Notez que le réseau n'est pas parfaitement intégré vu que c'est arrivé en 1983 seulement.
  - Plan 9 est un UNIX distribué, mais Unix est good enough. Plan 9 (et Inferno) sont abandonnés maintenant.
- Le noyau: `/proc` et `/sys`
- Le matériel: `/dev`
- Au passage, une chose agréable sous Unix est que la place des choses sur disque est standardisé (FHS – Filesystem Hierarchy Standard), et je ne comprend pas qu'Apple mette autant d'efforts à ne plus le respecter.

### II.4.b) Assembler les outils simples interchangeables

- Ca aide à l'établissement d'un écosystème aux briques évolutives
- Mécanismes dans le shell:
  - Redirection des E/S avec des tubes: `<`, `>`, `»`, `2>`, `|`
  - grep, sed, cat
  - voir les pages de manuel
- C'est dommage que la même chose n'ai jamais vu le jour pour l'UI, malgré les nombreuses tentatives
  - DOM-X windows a tourné au cauchemard de sécurité
  - AppleScript avait l'air sympa mais ca n'a pas marché
  - Gnome pousse l'usage du javascript (et c'est donc une mauvaise idée)

### II.4.c) Préférer les textes en clair (vs. binaire)

- Ca participe à simplifier l'assemblage de briques hétérogènes
  - Interopérabilité et debug.
  - Plus généralement, UNIX applique souvent le KISS
- C'est pour la config (vs. système de registres de windows) et pour les logs sur disque. Mais c'est aussi pour les protocoles du web (http and co)
- Dans l'histoire récente, cet aspect tend à disparaitre avec l'avènement de System D qui fait le contraire
  - Grosse crise dans Debian, à deux doigts de la scission. J'adore pas ce qui est advenu

## II.5) Interfaces de l'OS

- Interface graphique vs. shell: même usage. Le mieux est celui que l'on connait le mieux.

  - Pour moi c'est le shell car interactions plus riches (paramètres), et c'est utilisable à distance (accès ssh)
- Interface de programmation: En C la plupart du temps, même s'il faudrait arrêter ça et passer au Rust maintenant.
  - Importance de la notion de standardisation: avant POSIX, nombreuses variantes d'Unix incompatibles, ce qui rendait l'écriture de prog impossible.

## II.6) Repères historiques pour UNIX et C

- Ils ont été inventés ensemble, par les mêmes personnes
- UNIX
  - 1965: ambitieux projet MULTICS aux Bell Labs
  - 1969: Abandon de MULTICS, le projet UNICS commence
  - 1970: début officiel du projet UNIX
  - 1973: réécriture en C
  - 1982: AT&T perd son procès anti-monopole. UNIX est donné aux universités, dont Berkeley, et à diverses entreprises qui vont commercialiser leurs versions.
  - 1983: TCP/IP arrive, bien après le design d'UNIX. Le résultat ressemble à une verrue.
  - 80-90: UNIX War. c'est BSD contre System V (V=5 pas la lettre). Développer des applis portable entre toutes ces différences subtiles devient difficile
  - 90-00: normalisation. Les différentes normes POSIX posent un socle commun
  - 00-10: fin des UNIX propriétaires, abandonnés les uns après les autres par leurs entreprises. AIX (IBM), IRIX, HP-UX. Microsoft Windows règne sans partage. La légende dit que Microsoft s'assure de la survie d'Apple (en les finançant secrètement) Pour éviter le procès antitrust.
  - 10-20: internet et les webapps font passer windows au second plan. On parle peut-être moins des OS installés.
- Langage C:
  - 1967: BCPL aux Bell Labs
  - 1968: B (simplifié mais tjs très illisible)
  - 1971: C version K&R. Brian Kernighan et Dennis Ritchies (+ Thompson pour UNIX)
  - 1983: C++ Bjarne Stroustrup
  - 1989: ANSI C
  - 1990 puis 1999 puis 2011: versions de ISO C
  - Le C++ a bcp plus de variantes, l'institut de normalisation en ajoute tjs plus. C++11, C++14, C++17, C++20. Et c'est pas fini.

### II.6.a) Le shell en pratique

- Syntaxe fondamentale: `cmd options paramètres`
- commandes de base: ls, cd, gcc, emacs
- CtrlC CtrlZ & bg fg: lancer emacs.
- Historique des commandes, édition de commande, Complétion,
- Tubage de commandes
- Interaction avec les processus: `ps aux`, `kill 42`, `killall -STOP firefox`
- micro scripts de conversion d'images ou de concaténation de fichiers mp3: `convert -rotate 90`

- Lien vers les exos: `https://www.katacoda.com/mquinson/`

# III) Sécurité des systèmes d'exploitation

## III.1) Les systèmes réels ont des failles

- Parfois exprès pour ne pas brider l'implémentation
  - `while (1) fork()`
  - `while (1) {int*a = malloc(512); *a = '1'; }`
  - `while true ; do mkdir toto ; cd toto ; done`
  - Y'a pleins de *undefined behavior* en langage C aussi. Après un tel comportement, tout est possible (overflow sur les entiers => parfois $2000 < -2000$)
- Souvent pas exprès. Très nombreux problèmes de sécurité
  - dans les applis, motivation pour le Rust, donc. Faites du Rust je vous dis
  - mais aussi dans le matériel: meltdown
    - C'est une fuite de données (canal caché) car mise en commun des caches entre les processus.
    - Attaque RowHammer (!= Meltdown): Lecture de la mémoire des autres depuis un script javascript s'exécutant dans le sandbox du navigateur
    - Pour s'en protéger, Linux vide les caches à chaque changement de contexte. Perte de presque 30% de performances.

## III.2) Solutions classiques en sécurité

- Réponses techniques:
  - les mécanismes de protection mis en place par l'OS
  - Utilisateurs UNIX, droit d'accès (`rwx r-x r-x` : 755) et délégation de droit (sudo et setuid)
    - répertoires: r=ls; w=création éléments; x=cd
    - `chmod` et `ls -lh`
  - Windows offre des Acces Control List plus fins
- Réponses sociales:
  - quota disque $\rightsquigarrow$ publication des stats d'usage de chacun
  - Parfois, c'est la seule solution. Il est interdit juridiquement de reverse-engineerer certains matériels.
    - Portail Skylander sur Wii = simple lecteur RFID, mais service juridique interdit de regarder comment ça marche. Grosses menaces.
    - Wii turing complete, donc le constructeur ne peut pas les brider pour n'exécuter que du code maison. Mais l'installation de certains jeux efface tout code non certifié Nitendo, y compris les créations personnelles.

## III.3) Quelques propriétés de sécurité

- confidentialité: accessible aux seuls autorisés.
  - basé sur le secret et les maths. chiffrement asymétrique
  - confidentialité persistante (forward secrecy): si qqun sauvegarde le canal chiffré puis fini par trouver la clé dans le futur, il peut tjs pas lire. Génération de nouvelles clés à intervales réguliers, et échange des nouvelles clés sous la protection des précédentes. Donc on peut pas remonter le flux.

- intégrité: pas de modif indésirée
- authentification: garantie utilisateur est qui il prétend.
  - Attention à la biométrie car changer de mdp quand qqun sait le reproduire est plus facile que changer d'oeil ou d'empreinte digitale.
  - Un ministre allemand a perdu l'exclusivité de ses empreintes suite à une photo HD en conférence de presse.
- Anonymat: /!\ pseudonymat pas suffisant; TOR: on noie son flot dans la masse
- non-répudiation: celui qui a envoyé ça ne peut pas nier l'avoir fait. Banque, contrat.
  - Centralisé (notaire), basé sur la possession d'un secret (carte bleu ou clé PGP), ou décentralisée (contrat bitcoin)

## III.4) Règles à suivre

- Protéger ses données (c'est la loi)
  - Il faut offrir une protection raisonnable au regard des données protégées. Certains systèmes ne sont pas connectés à internet (mais stuxnet a fonctionné quand même)
  - Hadopi: charge de preuve inversée sur le défaut de sécurisation
- Ne pas contourner les protection (c'est la loi)
- Avoir la possibilité technique ne donne pas le droit
  - c'est comme avec le sac des petites vieilles
- La loi actuelle ne plaisante pas
  - Loi programmation militaire: internet = circonstance agravante, passage en antiterrorisme
  - jurisprudence blue touf: notion d'effraction n'existe pas. Interdit d'entrer quand tu sais que t'es pas sensé être là, même si c'est ouvert
- Vous êtes informaticiens, vous ne pourrez plus plaider l'incompétence technique, alors faites attention. Surtout vues la législation et la jurisprudence françaises