

Introduction au développement logiciel

Programmation système

Georges-André Silber, Centre de recherche en informatique (CRI)

Mines Paris — PSL, octobre 2026

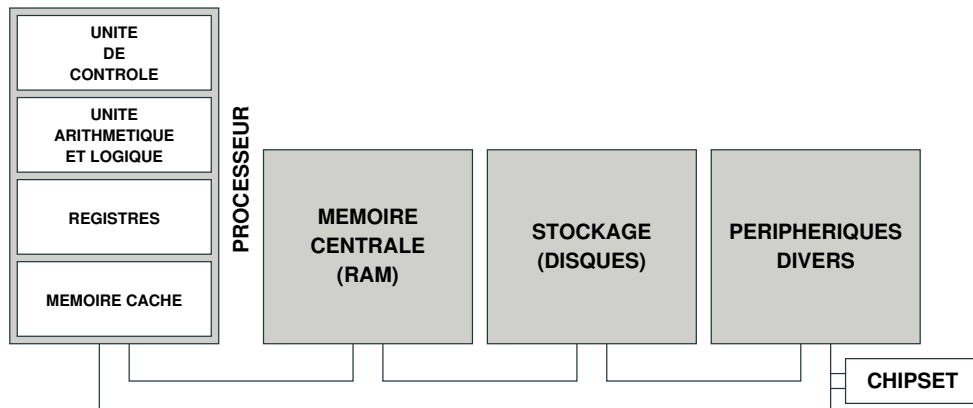
- Programmation à bas niveau, proche du matériel
- Systèmes d'exploitation, pilotes de périphériques
- Émulateurs et virtualiseurs
- Création de distributions (OS complets)
- Utilisation de langages "proches de la machine"
- C, Rust

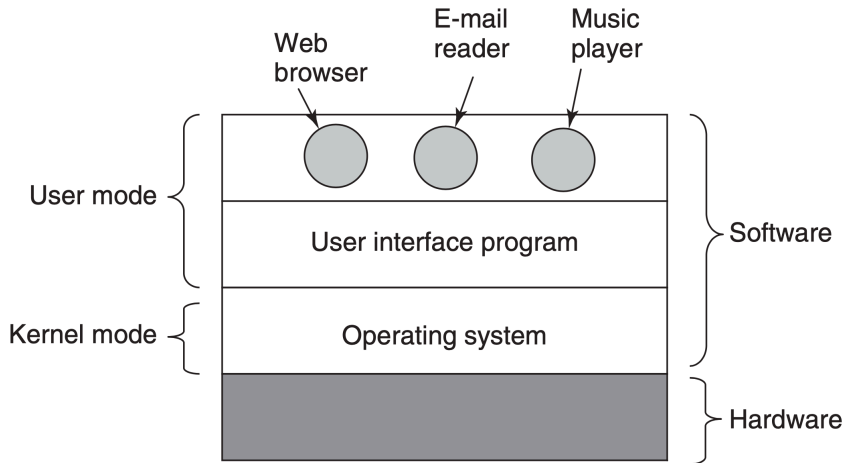
low-level language *A primitive programming language in which each line of code needs, but never gets, 20 lines of comment.*

— *The Devil's Dp Dictionary*

Système d'exploitation

- Un ensemble de logiciels **contrôlant les opérations** d'un ordinateur et de ses **ressources**.
- Caractéristique principale : **charger et exécuter** des programmes utilisateurs en leur fournissant des **interfaces normalisées** pour leurs entrées-sorties sur les périphériques.
- Autres fonctions, pas toujours toutes présentes (système embarqué) :
 - gestion de la mémoire et des ressources matérielles ;
 - politiques de sécurité ;
 - ordonnancement de processus ou de *threads* ;
 - interface utilisateur pour créer ou lancer des programmes.
- Certains systèmes sont **multi-tâches**, d'autres n'exécutent qu'un programme à la fois.



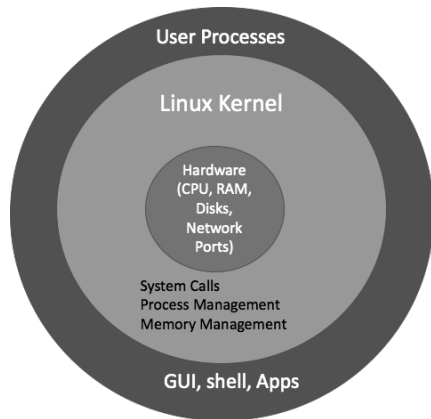


L'épine dorsale de l'infrastructure numérique mondiale est constituée de logiciels libres.

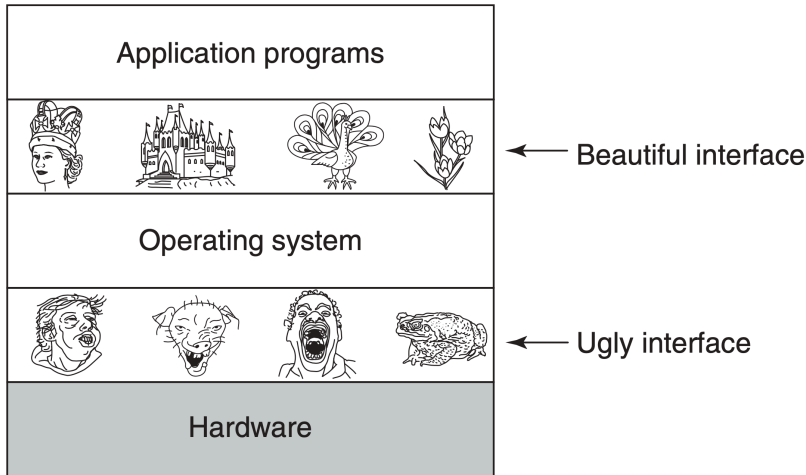
- Firmware et système d'exploitation (**CoreBoot**, **Linux**)
- Virtualisation (**KVM**, **Kubernetes**, **Docker**)
- Compilateurs optimisants (**GCC**, **LLVM**, **rustc**)
- Bibliothèques (**libc**, **libc++**, **libm**, **BLAS/LAPACK**)
- Interpréteurs (**CPython**, **Node.js**, **OpenJDK**)
- Paquets (**numpy**, **pytorch**, **tensorflow**, **scikit-learn**)

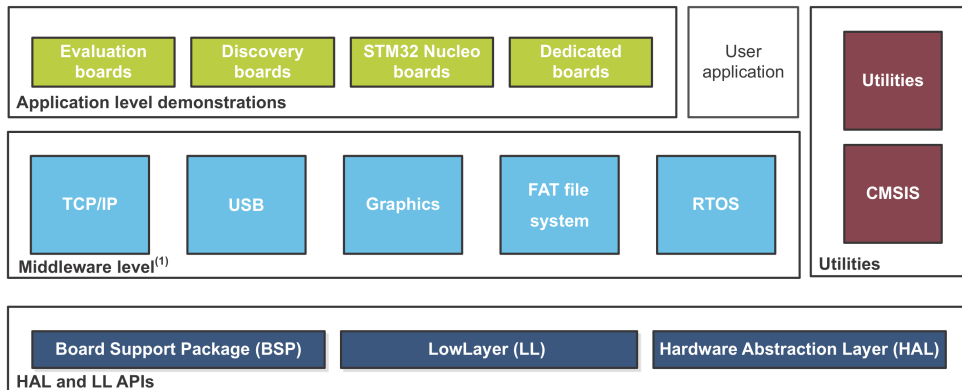
Dans ce cours : Linux, GCC/LLVM, libc, Python, Rust.

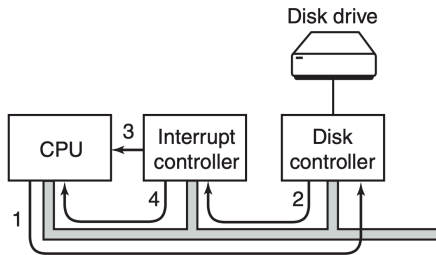
- Les **applications** de l'espace utilisateur ne font pas partie du système : éditeur de texte, navigateur, jeu, même livrés avec lui dans une **distribution** (Ubuntu, Red Hat).
- La plupart des **interfaces graphiques** non plus, même si elles sont liées à un OS spécifique.
- Le cœur du système est le **noyau** (*kernel*) : une **frontière** pour toutes les applications, tout accès au matériel passe par lui.
- Le noyau gère des **événements** :
interruptions générées par le matériel, par exemple l'appui d'une touche ;
appels système toujours initiés par un programme de l'espace utilisateur, par exemple la lecture d'un fichier.



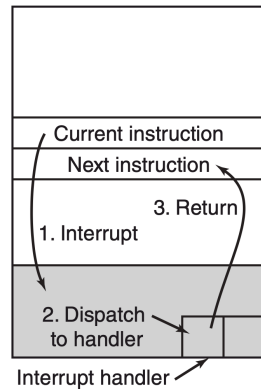
- La plupart des programmes n'appellent pas le noyau directement mais passent par des **bibliothèques** : sous UNIX, la **libc**, utilisée par quasiment tous les programmes.
- On peut appeler le noyau sans bibliothèque, notamment en **assembleur**.
- Le **shell** « entoure » le noyau : historiquement le seul programme permettant d'en lancer d'autres. Ce n'est plus le cas avec les interfaces graphiques.



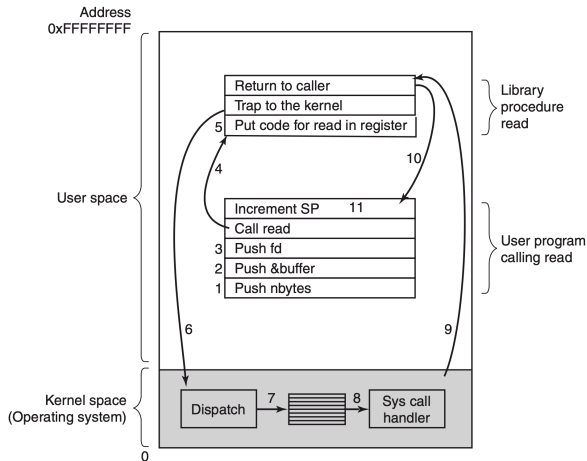




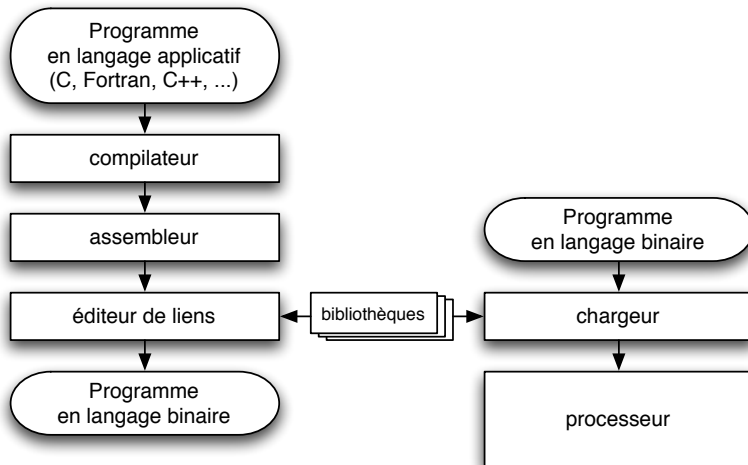
(a)



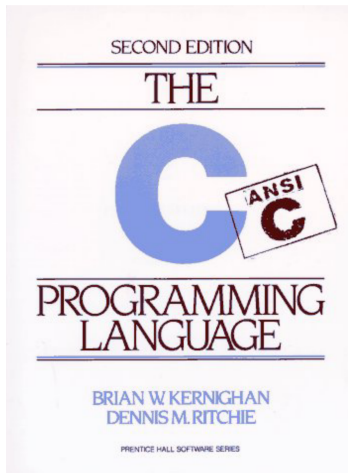
(b)



- Appels système Linux
- <https://filippo.io/linux-syscall-table/>
- `man syscall`
- `man syscalls`
- `man 2 intro`



- Développer les programmes qui **font partie** du système d'exploitation ou qui en **réalisent les fonctions**.
- Accès aux fichiers, programmation du clavier, de l'écran, du réseau : de tous les **périphériques** qui font entrer ou sortir de l'information, de la mémoire vive et des processeurs.
- Nécessite une compréhension fine du **matériel** et des couches logicielles « basses » : langages de bas niveau, systèmes d'exploitation, compilateurs.
- Langages : **assembleur**, **C**, plus récemment **Rust**. Linux est écrit à 99 % en C, le reste en assembleur.



- Dennis Ritchie
- Brian Kernighan
- 1972 → C23 (2024?)
- <https://godbolt.org>

Contrôler complètement la machine. Une application de l'espace utilisateur fait toujours appel à du code écrit par d'autres, qui s'interpose entre elle et la machine. À bas niveau, on contrôle exactement ce que fait la machine.

Recherche. Étudier de nouveaux concepts : tolérance aux pannes, systèmes à image unique sur plusieurs machines.

Remplacer les OS existants. Un système pilotant un métro automatique devrait être prouvé correct. Ne pas toujours écouter « cela existe déjà » : Linus Torvalds n'aurait jamais écrit Linux en 1991, plusieurs UNIX existaient.

Parce que c'est amusant et instructif. Difficile, mais permet de programmer de la manière la plus **frugale** possible et de comprendre finement comment fonctionne un ordinateur.

Démarrage d'un ordinateur : du *firmware* au *bootloader*



- Firmware en mémoire Flash
- Accès bas niveau à un périphérique
- Lecture du MBR (512 octets)
- Bootloader + partition disque
- Chargement bootloader niveau 2
- Chargement binaire plus compliqué
- etc...

- À la mise sous tension, le CPU lance un logiciel interne, le ***firmware***, stocké dans une mémoire **non volatile** atteignable par le CPU (EEPROM, Flash).
- Il déclenche la **séquence d'amorçage** (*boot, bootstrap sequence*).
- *Firm*, ferme : entre le *hardware*, dur, et le *software*, mou. Un logiciel fourni par la *firme* qui fabrique le matériel. Micrologiciel, microcode ou logiciel interne en français, termes imprécis.
- Sur un PC : **BIOS** (*Basic Input Output System*) ou **UEFI** (*Unified Extensible Firmware Interface*).
- Équivalents sur tous les appareils : télévisions, enceintes, téléphones. Certaines séquences sont exotiques : le Raspberry Pi démarre par son **GPU**.

1. Le *firmware* **teste et initialise** les composants matériels.
2. Il **identifie les périphériques** et sélectionne ceux utilisables pour poursuivre, dans un **ordre modifiable** : clef USB, disque interne, carte réseau (protocole **PXE**).
3. Depuis ce périphérique, il cherche un programme exécutable : le ***bootloader***.
4. Il lui **passe la main** : le *bootloader* devient maître de la machine, tout en pouvant encore utiliser les routines du BIOS ou de l'UEFI, par exemple pour afficher sur la console ou lire le disque.

BIOS, MBR le BIOS sait lire les **512 premiers octets** d'un disque, le *Master Boot Record* : un *bootloader* de **446 octets** au plus, et la **table des partitions**, 4 partitions de 2,2 To au maximum. En pratique le *bootloader* du MBR sait lire une partition et charger un *bootloader* plus complexe, comme **GRUB**, qui charge à son tour le **noyau**.

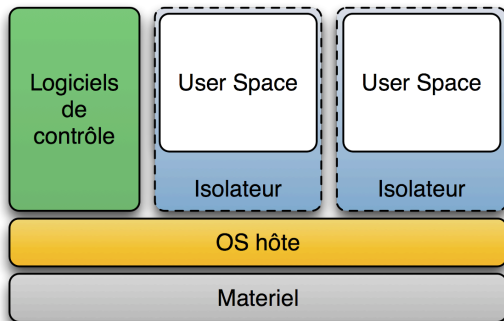
UEFI, GPT gère le MBR par compatibilité, mais utilise la table de partitions **GPT** : jusqu'à 128 partitions de 9,4 Zo ($9,4 \times 10^{21}$ octets). Peut charger directement GRUB depuis une partition **FAT**.

PXE chargement par le réseau : un serveur **DHCP** fournit l'adresse d'un serveur **TFTP** qui fournit le *bootloader*.

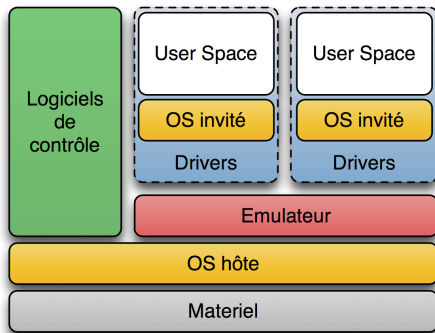
→ TP : écrire un MBR de 512 octets et le lancer dans QEMU.

Conteneurs, émulation et virtualisation

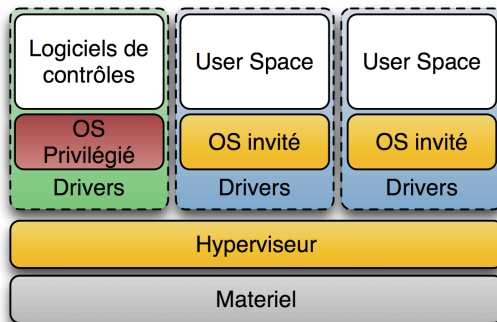
- chroot, Jails de FreeBSD
- *cgroups* et *namespaces* de Linux : LXC, LXD, Docker



- VirtualBox, QEMU



- Xen, KVM



Les fondements de la philosophie d'UNIX

Doug McIlroy, *Unix Time-Sharing System : Forward*, 1978 :

- *Make each program do **one thing well**. To do a new job, build afresh rather than complicate old programs by adding new « features ».*
- *Expect the **output** of every program to become the **input** to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.*
- *Design and build software, even operating systems, to be **tried early**, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.*
- *Use **tools** in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.*

« This is the Unix philosophy : write programs that do one thing and do it well, write programs that work together, write programs to handle text streams, because that is a universal interface. »

— Doug McIlroy

Universel si l'on fait abstraction de l'encodage des caractères, donc si l'on reste en ASCII.

Rob Pike, *Notes on Programming in C*, 1989, reprises dans *The Art of UNIX Programming* d'Eric Raymond.

Rule 1. *You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.*

Rule 2. *Measure. Don't tune for speed until you've measured, and even then don't, unless some part of the code overwhelms the rest.*

On peut quand même parfois savoir où le programme va passer du temps (Hoare, Knuth : « *premature optimization is the root of all evil* »). À discuter : énergie et code performant sont indispensables aujourd'hui.

- Rule 3.** *Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy.*
- Rule 4.** *Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures : array, linked list, hash table, binary tree. A symbol table might be implemented as a hash table containing linked lists of arrays of characters.*
- Rule 5.** *Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.*

Ken Thompson : « *when in doubt, use brute force* ».

« *Those who do not understand Unix are condemned to reinvent it, poorly.* »

— Henry Spencer, signature Usenet, novembre 1987

Références

- R. et A. Arpaci-Dusseau, *Operating Systems : Three Easy Pieces*, <https://pages.cs.wisc.edu/~remzi/OSTEP/>
- OSDev.org, *Introduction*, <https://wiki.osdev.org/Introduction>
- P. Oppermaun, *Writing an OS in Rust*, <https://os.phil-opp.com>
- D. Both, *An introduction to the Linux boot and startup processes*, 2017, <https://opensource.com/article/17/2/linux-boot-and-startup>
- L. Torvalds, emails commentés retraçant la genèse de Linux, 1992, <https://www.cs.cmu.edu/~awb/linux.history.html>
- L. A. Barroso, J. Dean, U. Hölzle, *Web Search for a Planet : The Google Cluster Architecture*, IEEE Micro, 2003

- Microsoft, sources de MS-DOS 1.25 et 2.0, <https://github.com/microsoft/MS-DOS>
- F. Cloutier, *x86 and amd64 instruction reference*,
<https://www.felixcloutier.com/x86/>; Intel SDM
- Table des appels système Linux, <https://filippo.io/linux-syscall-table/>
- QEMU <https://www.qemu.org>, Buildroot <https://buildroot.org>, Busybox
<https://busybox.net>, Compiler Explorer <https://godbolt.org>
- A. Finch, *Moon Machines, 3 : l'ordinateur de navigation*, 2008,
<https://www.youtube.com/watch?v=DWcITjqZtpU>