Introduction à la programmation système
Notes de cours

Georges-André Silber
École des mines de Paris

Septembre 2024

Table des matières

Préambule

1. Le cœur d'un ordinateur est son processeur, qui agit sur une mémoire principale organisée en cases de tailles fixes, accessibles par des adresses, en utilisant des cases mémoire de travail appelées registres.
2. Fondamentalement, un processeur est un automate exécutant, lorsqu'il est sous tension, une suite d'instructions se trouvant en mémoire, en commençant à une adresse fixe stockée dans un registre spécial appelé *Program Counter* (PC).
3. Les ordinateurs modernes ont une architecture de type Von Neumann.
4. Un processeur ne peut se programmer que grâce aux instructions de son architecture externe (ISA, *instruction set architecture*).
5. Les instructions du processeur peuvent être regroupées dans trois classes principales : affectations, tests et branchements.
6. L'affectation consiste à altérer le contenu d'une case mémoire, en copiant une valeur d'une case mémoire vers une autre, en altérant éventuellement le contenu par une opération arithmétique.
7. Le test consiste à comparer deux valeurs, constantes ou se trouvant en mémoire. Le résultat de ce test peut ensuite être utilisé pour exécuter ou non une instruction.

8. Après l'exécution d'une instruction, le processeur exécute normalement l'instruction se trouvant juste après en mémoire, dont l'adresse est stockée dans le registre `PC` (le *compteur ordinal* ou *program counter*). Le branchement permet de faire exécuter au processeur une instruction se trouvant à une autre adresse, en altérant `PC`.
9. Le langage de programmation constitué par l'ISA d'un processeur est *Turing complete* car il permet, suivant le théorème de Böhm–Jacopini, d'implémenter les trois structures de contrôles que sont la séquence, la sélection via un booléen (le `if`) et l'itération (le `while`).
10. Andrew Tanenbaum, distingue trois niveaux matériels dans un ordinateur : niveau 2, architecture externe ou de jeu d'instructions (ISA) ; niveau 1, microarchitecture ; niveau 0, réalisation en logique numérique. Exemple (fictif) : niveau architecture externe, il existe huit registres d'usage général. L'instruction ADD effectue une addition entre deux quelconques de ces registres et met le résultat dans un quelconque troisième ; niveau microarchitecture (architecture interne) : le traitement des instructions est pipeliné8 et deux unités arithmétiques et logiques sont capables d'effectuer l'addition de l'instruction ADD ; niveau réalisation : le processeur est réalisé dans une technologie CMOS avec trois niveaux de métaux et une finesse de gravure de 130 nanomètres.

Introduction

Un système d'exploitation [1] est un ensemble de logiciels controllant les opérations d'un ordinateur et de ses ressources (fig. 1). La caractéristique principale d'un tel système est d'être capable de charger et d'exécuter des programmes utilisateurs en leur fournissant des interfaces normalisées pour leurs entrées-sorties [2] sur les périphériques de l'ordinateur.

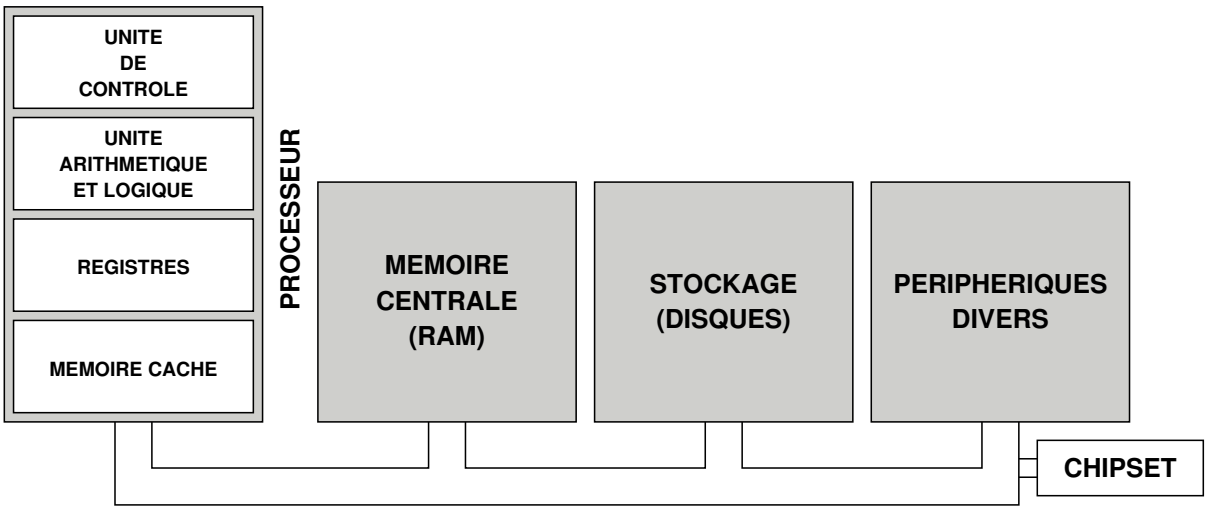

FIGURE 1 – Schéma simplifié d'un ordinateur

Parmi les fonctions principales d'un système, on trouve également la gestion de la mémoire et des ressources matérielles, le mise en place de politiques de sécurité, l'ordonnancement de processus ou de *threads*, la fourniture d'une interface utilisateur pour créer de nouveaux programmes ou les exécuter. Ces fonctions ne sont pas forcément présentes sur tous les systèmes, un système embarqué ne proposant par exemple par forcément d'interface utilisateur, et peuvent différer dans leur mise en œuvre, certains systèmes multi-tâches permettant par exemple l'exécution de plusieurs programmes en même temps alors que d'autres n'autorisent qu'un seul programme à la fois.

1. *operating system* ou OS.
2. *Input-Output* ou IO

La programmation système nécessite une compréhension fine du matériel – architectures des ordinateurs, micro-processeurs – et des couches logicielles dites « *basses* » – langages de bas niveau, systèmes d'exploitation, compilateurs.

Nous allons gratter la surface de ce domaine en balayant l'assembleur de plusieurs types de processeurs, ainsi que le langage C, Rust et certaines notions liées aux systèmes d'exploitation, de la séquence de boot aux processus, en passant par le noyau, la gestion de la mémoire et les systèmes de fichiers.

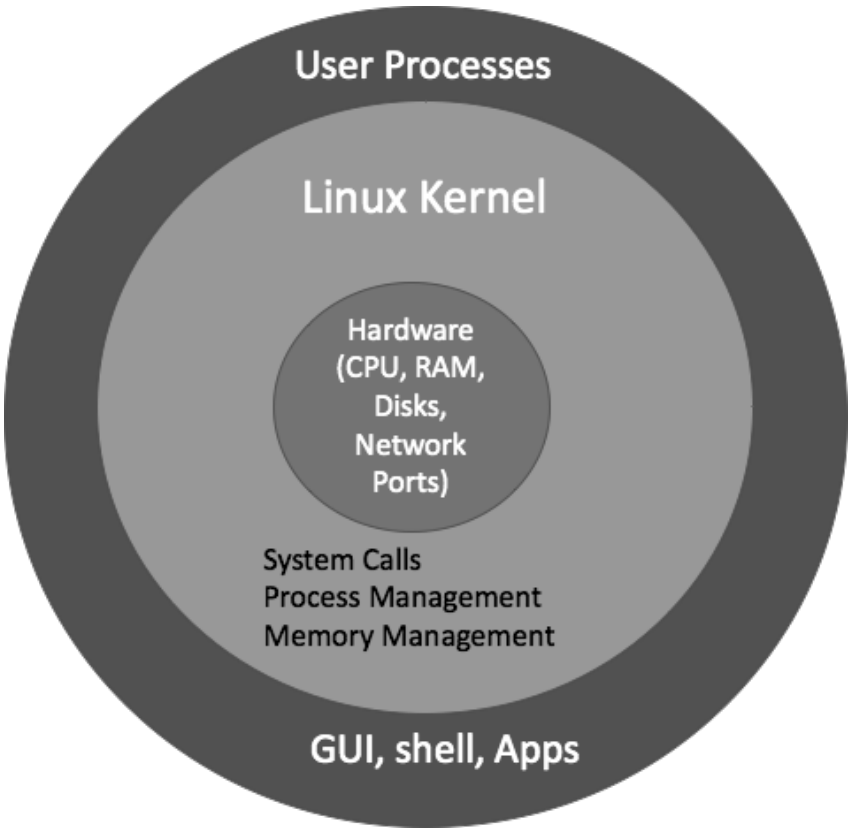

FIGURE 2 – Shell et noyau

Les applications qui tournent dans l'espace utilisateur d'un système d'exploitation ne font pas partie a priori de ce système. Ainsi, un éditeur de texte, un navigateur web ou un jeu ne font pas partie du système d'exploitation, même si ces applications ne tournent que *sur* ce système d'exploitation et sont livrées avec lui, par exemple dans ce que l'on appelle une *distribution* Linux comme Ubuntu ou Red Hat. Ainsi, la plupart des interfaces graphiques ne peuvent pas être considérées comme faisant partie du système d'exploitation, même si elles sont pour la plupart liées à un OS spécifique.

Le cœur d'un système d'exploitation est appelé noyau ou *kernel* et est une frontière pour toutes les applications tournant sur la machine : tous les accès aux matériel doivent passer par lui. L'une de ses fonctions principale est de gérer des évènements générés par le matériel, appelés *interruptions*, ou logiciels, appelés *appels systèmes* ou *system calls*. Parmi les interruptions, on trouve par exemple l'appui d'une touche sur le clavier de l'ordinateur. A contrario, un appel système est toujours initié par un programme tournant dans l'espace utilisateur, comme par exemple lors de la lecture d'un fichier.

La plupart des programmes utilisateurs n'utilisent pas directement les appels système du noyau mais passent par des bibliothèques de routines proposant des fonctions de plus haut niveau, la

bibliothèque la plus importante dans l'univers UNIX étant la bibliothèque du langage C, également appelée `libc`, utilisée par quasiment tous les programmes s'exécutant sur le système. Il est toutefois possible d'utiliser directement les appels systèmes du noyau sans employer de bibliothèque tierce, notamment en codant en assembleur.

Ce que l'on appelle le *shell* a une place à part dans cette hiérarchie : son nom provient historiquement du programme élémentaire qui permettait à un utilisateur de lancer des programmes, donc « *d'entourer* » le noyau. Il n'était possible de lancer un programme qu'à travers lui. Avec l'avènement des interfaces graphiques et de mécanisme de lancement intégrés au système, le shell n'est aujourd'hui plus le seul moyen de lancer des programmes sur le système.

La programmation système, objet de ce cours, est un type de programmation qui vise au développement des programmes qui font partie du système d'exploitation d'un ordinateur ou qui en réalisent les fonctions. Elle inclut l'accès aux fichiers, la programmation du clavier, de l'écran, la programmation réseau, et, en général, la programmation de tous les périphériques qui font entrer ou sortir de l'information d'un ordinateur, de la mémoire vive et des processeurs.

La programmation système se fait généralement par le biais de langages considérés comme bas niveau comme le langage assembleur, le langage C ou plus récemment Rust. Un système d'exploitation de type Linux est écrit à 99 % en C, le reste étant de l'assembleur.

Pourquoi s'intéresser aux couches basses ? Il y a plusieurs raisons qui peuvent pousser à s'intéresser aux couches basses, à créer un nouvel OS ou à simplement s'intéresser à la programmation système :

— *Contrôler complètement la machine*. L'écriture d'une application dans l'espace utilisateur fait toujours appel à du code écrit par d'autres et qui s'interpose entre l'application et la machine. En écrivant du code à bas niveau, on contrôle exactement ce que fait la machine et on peut l'adapter exactement à un besoin spécifique.
— *Recherche.* L'étude de nouveaux concepts peut justifier l'écriture d'un nouveau système, notamment pour étudier des algorithmes et protocoles de tolérance aux pannes, pour explorer la création de systèmes à image unique tournant sur plusieurs machines en même temps.
— *Remplacer les OS existants.* Peut-être qu'il n'existe pas de système avec une caractéristique nécessaire à un fonctionnement dans un contexte donné. Un système tournant sur une machine pilotant un métro automatique devrait par exemple être prouvé correct, nécessitant donc un codage spécifique. D'ailleurs, il ne faut pas toujours écouter ceux qui disent « *cela existe déjà* » : Linus Torvalds n'aurait jamais développé Linux à partir de 1991, considérant que plusieurs version d'UNIX existaient déjà.
— *Parce que c'est amusant et instructif.* La programmation bas niveau peut être amusante et gratifiante car elle est difficile mais vous permet de programmer de la manière la plus frugale possible. Elle permet également de comprendre de la manière la plus fine possible comment fonctionne un ordinateur.

Du point de vue d'une entreprise, le passé récent nous apprend que l'innovation est facile, mais que l'exécution est difficile : les compagnies informatiques dominantes, parfois appelées GAFAM, se sont contruites sur la maîtrise d'une informatique à bas niveau. Google a percé dans le domaine des moteurs de recherche en proposant une vitesse de recherche plus importante que les autres : dès le premier jour, les deux créateurs étudiants à Stanford ont construit leur propre *cluster* et une application massivement parallèle. Apple a été dès le départ une compagnie proposant du hardware et a rapidement créé son propre software, maîtrisant aujourd'hui toute la chaîne, du microprocesseur à l'OS. Facebook a été créé avec un socle technologique solide et un investissement massif et constant

sur son infrastructure et ses logiciels, dirigé par un *hacker* aux méthodes parfois discutables. Amazon a compris très tôt que le développement de sa propre infrastructure et de sa propre pile logicielle allait avoir une importance stratégique. Microsoft a comme origine la création d'un OS pour l'IBM PC, MS-DOS.

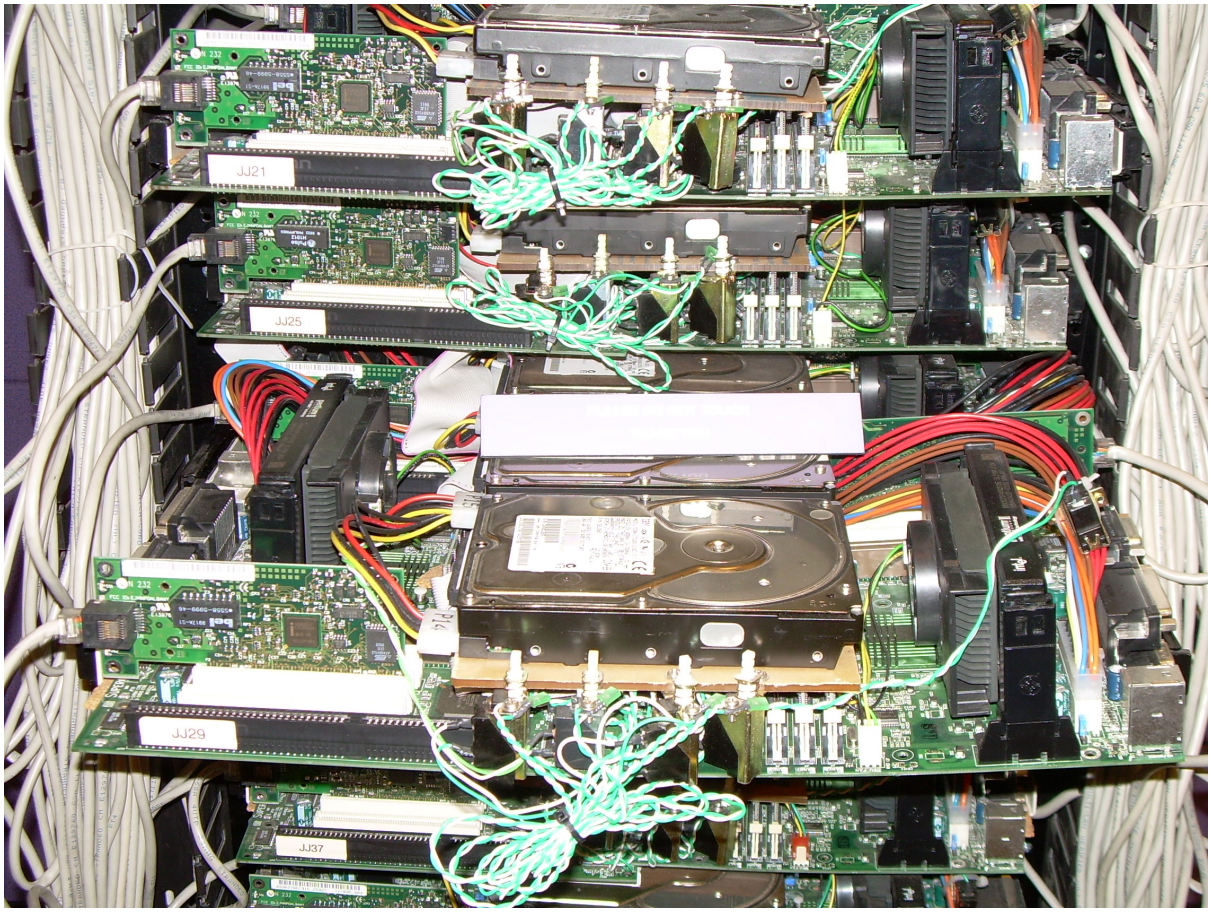

FIGURE 3 – Le premier serveur de production de Google

Architectures externes et internes de processeurs

Une *architecture externe de processeur* ou *architecture de jeu d'instructions* – en anglais ISA, *Instruction Set Architecture* –, ou tout simplement *architecture de processeur*, est la spécification fonctionnelle d'un processeur, du point de vue du programmeur en langage machine. L'architecture comprend notamment le jeu d'instructions, les registres visibles par le programmeur, l'organisation de la mémoire et des entrées sorties, les modalités d'un éventuel support multiprocesseur.

Le terme *externe* permet de se différencier de la *microarchitecture* ou *architecture interne*, qui s'intéresse à l'implémentation pratique du comportement spécifié par une architecture externe. Une architecture externe donnée peut être implémentée sous forme de plusieurs microarchitectures. Ainsi, les sociétés Intel et AMD proposent toutes les deux des microprocesseurs d'architecture externe x86, mais avec des microarchitectures différentes.

Dans ce cours, nous ne nous interesserons qu'aux architectures externes de processeurs et emploierons indifféremment le terme *architecture* ou ISA pour la désigner. L'achitecture matérielle d'un ordinateur complet, incluant la microarchitecture d'un processeur, est un vaste sujet que nous n'abor-

FIGURE 4 – Allen et Gates en 1981

derons dans ce cours hormis sous des aspects comportementaux nécessaires à une programmation système efficace.

Parmi toutes les architectures de processeurs existantes, nous ferons dans ce cours et dans les exercices un survol des architectures x86, ARM, qui couvrent la majorité des processeurs dans l'on trouve dans les machines actuelles.

Voici un exemple de fonction codée en langage C, ajoutant à un emplacement mémoire contenant un entier signé sur 64 bits (`long`), un entier signé sur 32 bits (`int`) :

```
void f(long *a, int c)
{
    *a = *a + c;
}
```

Voici le code équivalent en Rust :

```
fn f(a: &mut s64, c: s32) {
  let cst = c as s64;
  *a = *a + cst;
}
```

En assembleur x86 en 64 bits, le corps de la fonction peut être écrit ainsi, avec `a` contenu dans `rdi` et `b` contenu dans `esi`

```
movsxd rsi, esi            # rsi <- SignExtend(esi)
add qword ptr [rdi], rsi   # memory[rdi] <- memory[rdi] + rsi
```

En assembleur ARM en 64 bits (arm), on aurait :

```
ldr     x2, [x0]           # x2 <- memory[x0]
add     x1, x2, w1, sxtw   # x1 <- x2 + SignExtend(w1)  (sxtw = Sign eXTend Word)
str     x1, [x0]           # memory[x0] <- x1
```

Architecture x86

L'architecture x86 est une famille d'architectures dont le premier réprésentant matériel est le processeur 8086 d'Intel introduit en 1978. Architecture 16 bits à l'origine, elle a évolué en architecture 64 bits sous l'impulsion d'AMD en 2005 [3].

Le processeurs implémentant cette architecture fonctionnent principalement dans deux modes :

— le mode en *adresses réelles*, dit aussi *real mode*, implémente l'environnement de programmation d'un processeur 8086 des origines, avec aujourd'hui des extensions, comme la possibilité d'utiliser les registres 32 bits ou la possibilité de passer en mode *protégé*. Le processeur démarre toujours dans ce mode lorsqu'il est alimenté en courant ;
— le mode *protégé*, qui est le mode natif du processeur – celui dans lequel il est censé fonctionner. C'est ce mode qui permet à un système d'exploitation d'être multi-tâches et d'avoir une représentation de la mémoire « *à plat* ».

L'architecture x86, adoptant une organisation des octets de type *little-endian* (voir fig. 5) possède notamment les registres suivants (voir figs. 6, 7, 8, 9, 10, 11, 12) :

3. Intel avait à l'époque tablé sur une nouvelle architecture aujourd'hui quasiment disparue, l'Itanium.

— `eax` — Accumulator for operands and results data
— `ebx` — Pointer to data in the `ds` segment
— `ecx` — Counter for string and loop operations
— `edx` — I/O pointer
— `esi` — Pointer to data in the segment pointed to by the DS register ; source pointer for string operations
— `edi` — Pointer to data (or destination) in the segment pointed to by the `es` register ; destination pointer for string operations
— `esp` — Stack pointer (in the `ss` segment)
— `ebp` — Pointer to data on the stack (in the `ss` segment)

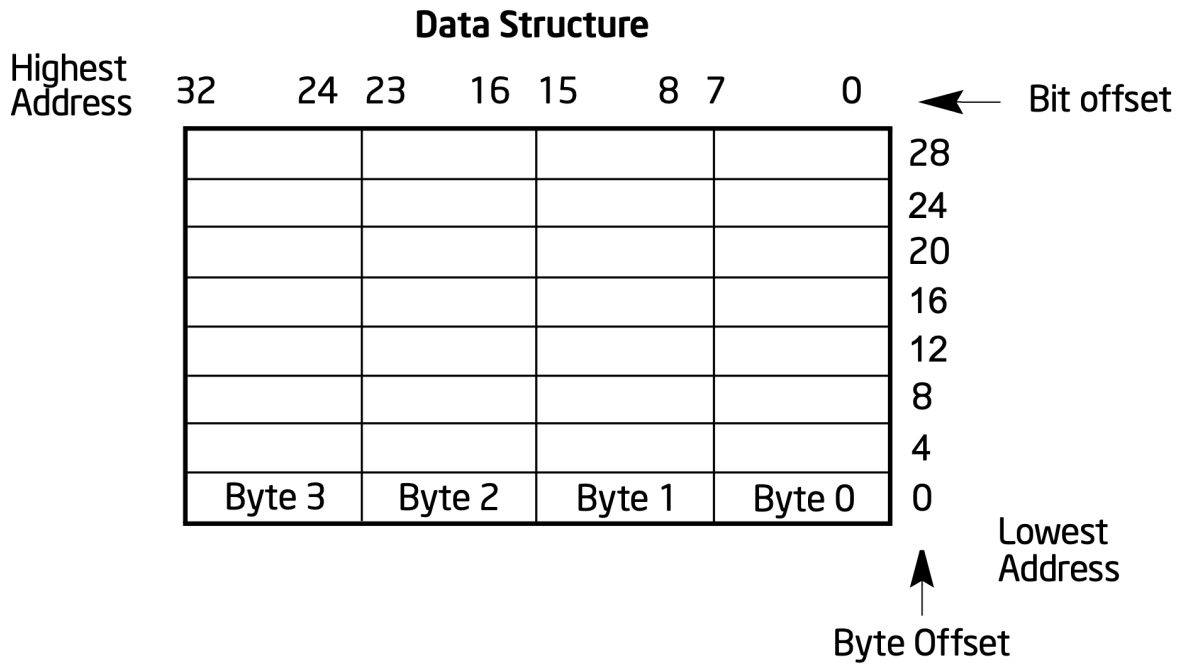

FIGURE 5 – Bits et octets, architecture Intel *little-endian*.

Les principales instructions sont présentées dans la documentation de référence Intel ou sur des sites comme `sandpile.org`. L'accès à la mémoire se fait selon des calculs d'adresse présentés à la fig.13. Les types de données fondamentaux que l'architecture x86 peut manipuler sont les entiers et les flottants, voir figs.14, 15, 16, 17, 18, 19, 20, 21. Le processeur a plusieurs modes de protection, voir fig. 22.

Architecture ARM

ARM [4] est une famille d'architectures RISC de microprocesseurs, avec des variantes pour plusieurs environnements, développées par *ARM Ltd* depuis 1983.

La société *Arm Ltd.*, à l'origine une coentreprise fondée par Acorn Computers, Apple Computers et VLSI Technologies, développe les architectures et concède des licences à d'autre compagnies, qui conçoivent leurs propres produits implémentant ces architectures, comme des SoC intégrant plusieurs

4. stylisé en bas de casse en arm, auparavant un acronyme de *Advanced RISC Machines*, et à l'origine de *Acorn RISC Machine*

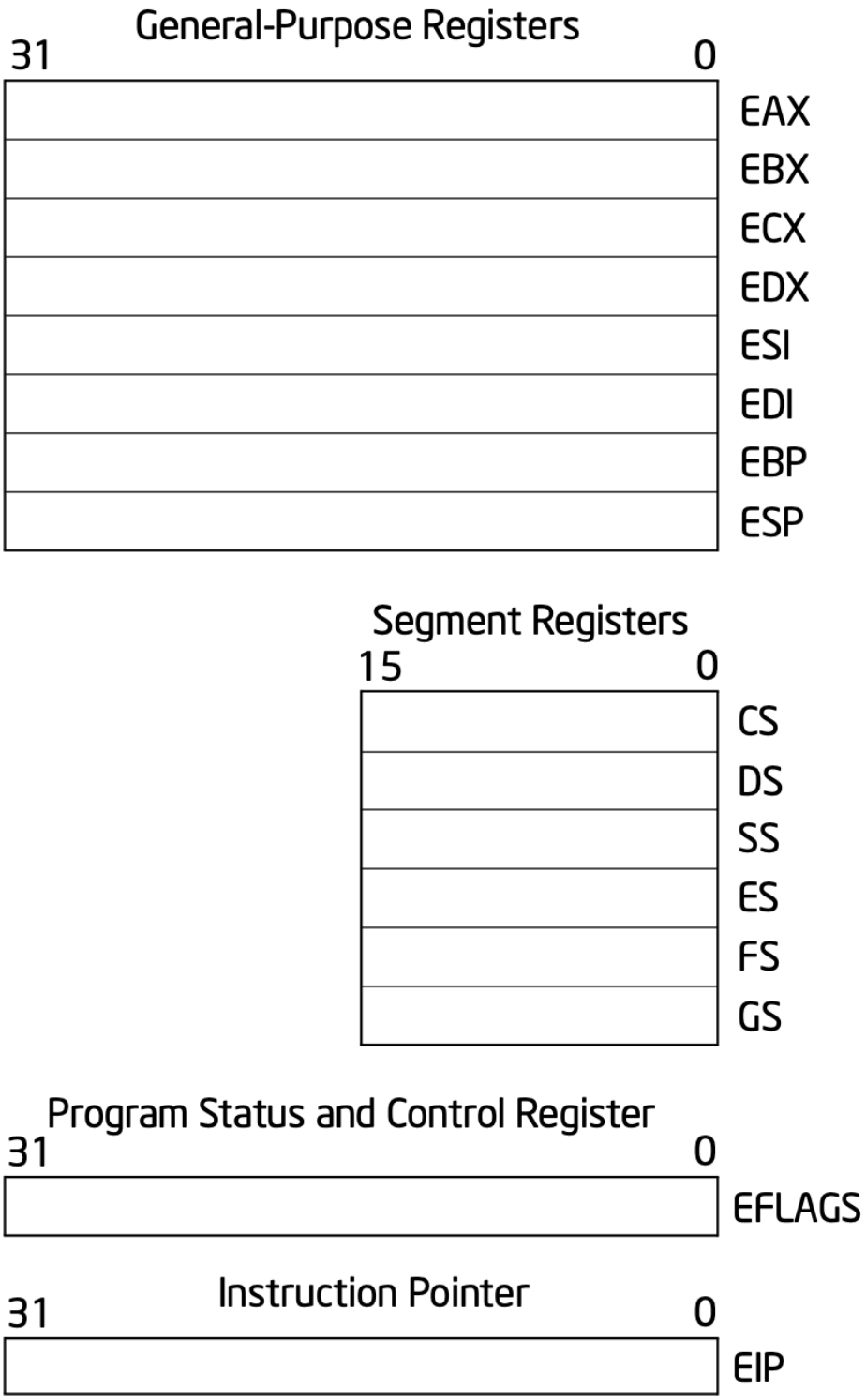

FIGURE 6 – Registres généraux et de segment, x86 en 32 bits

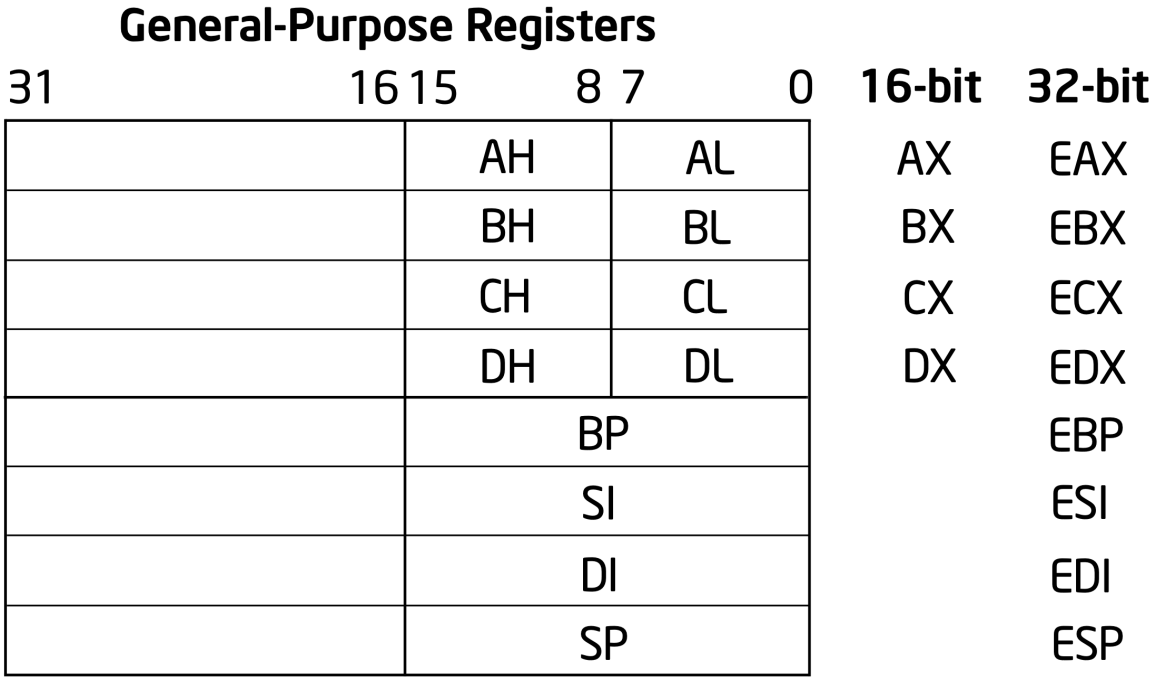

FIGURE 7 – Sous-registres, x86 en 32 bits

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8B - R15B
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

FIGURE 8 – Registres x86 en 64 bits

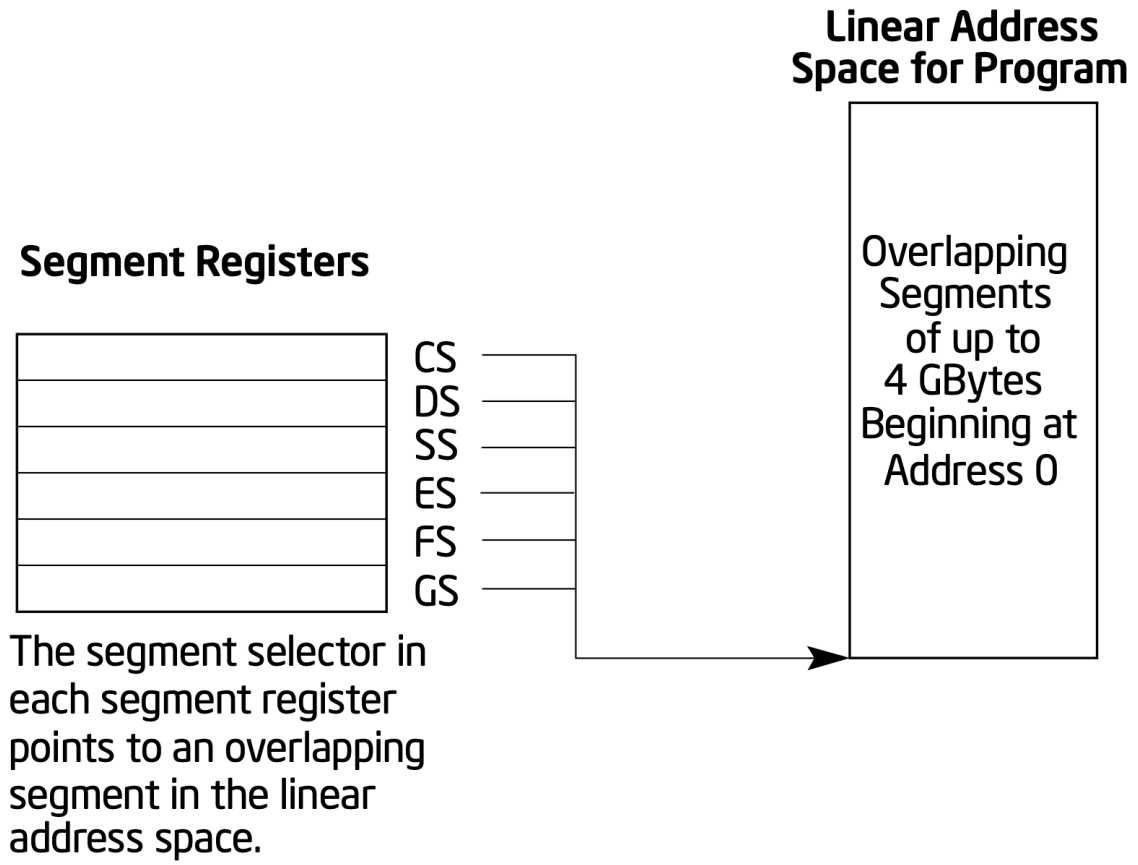

FIGURE 9 – Registres de segment, mémoire segmentée

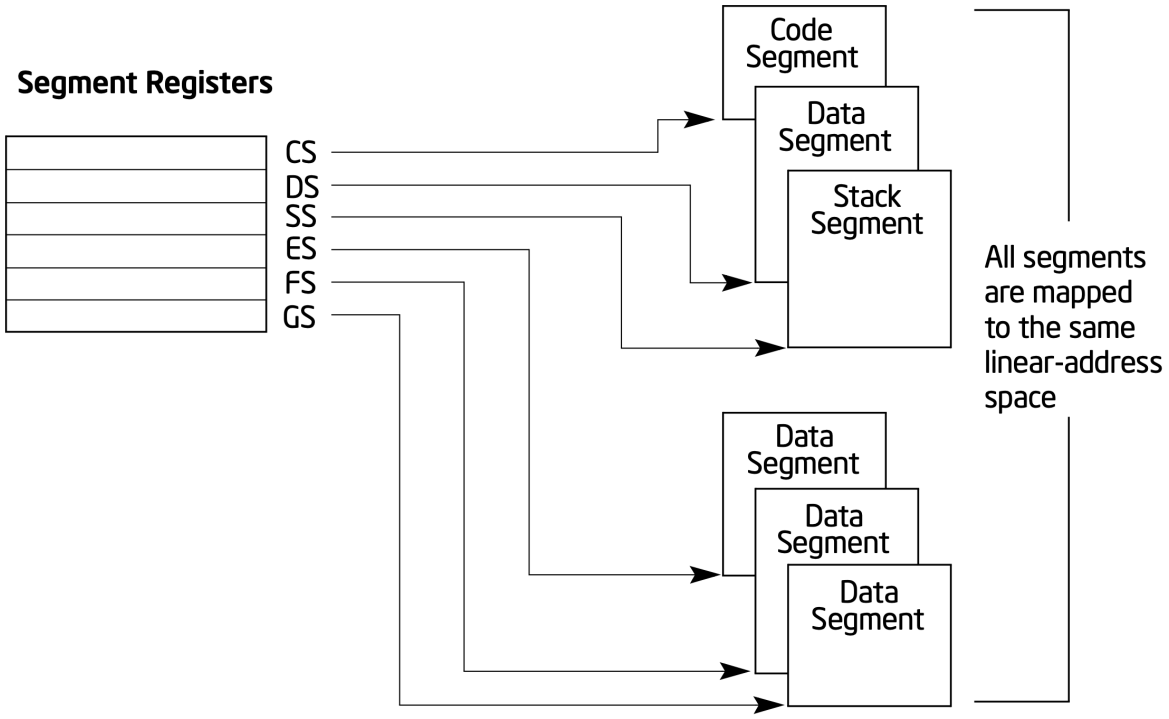

FIGURE 10 – Registres de segment, mémoire à plat

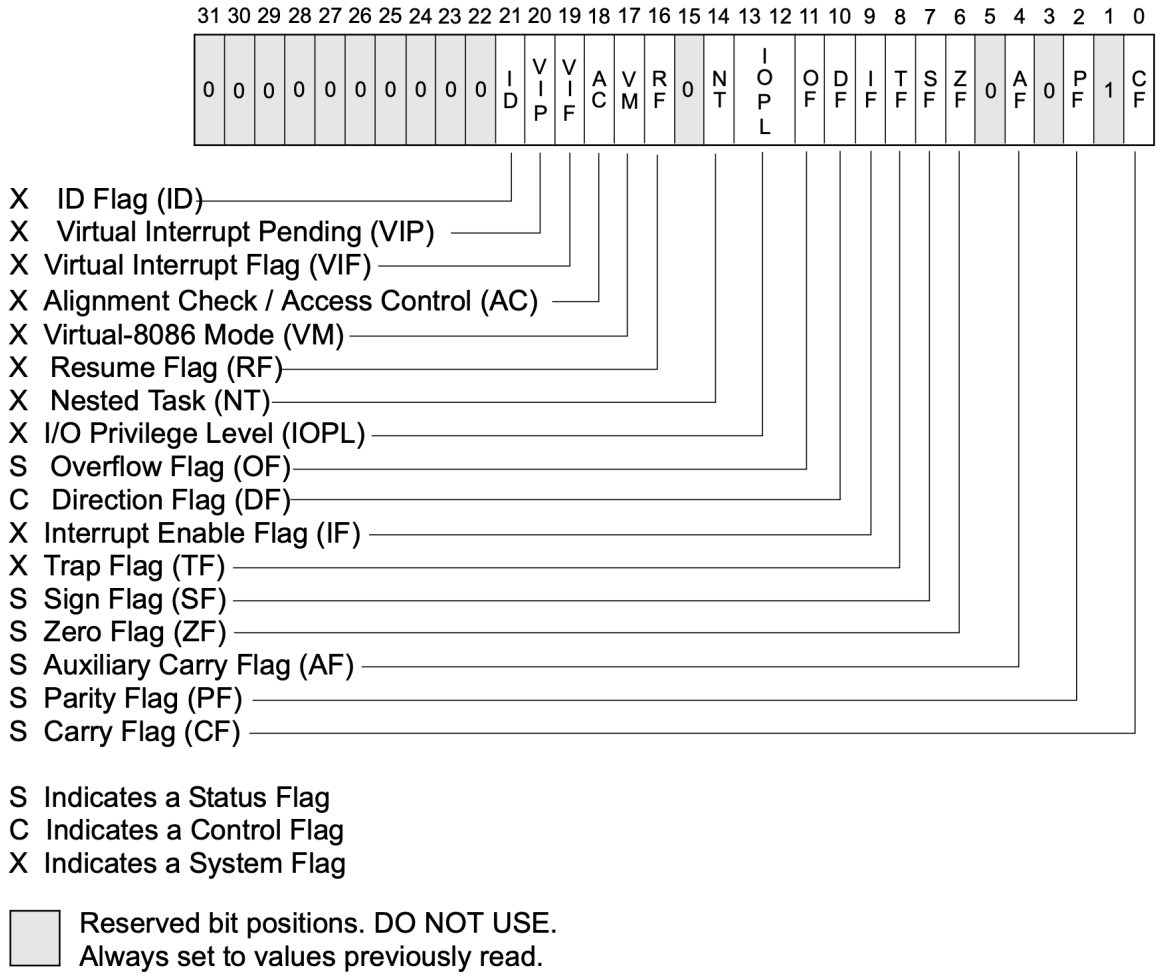

FIGURE 11 – Registre d'état

FIGURE 12 – Tables des registres de l'architecture x86

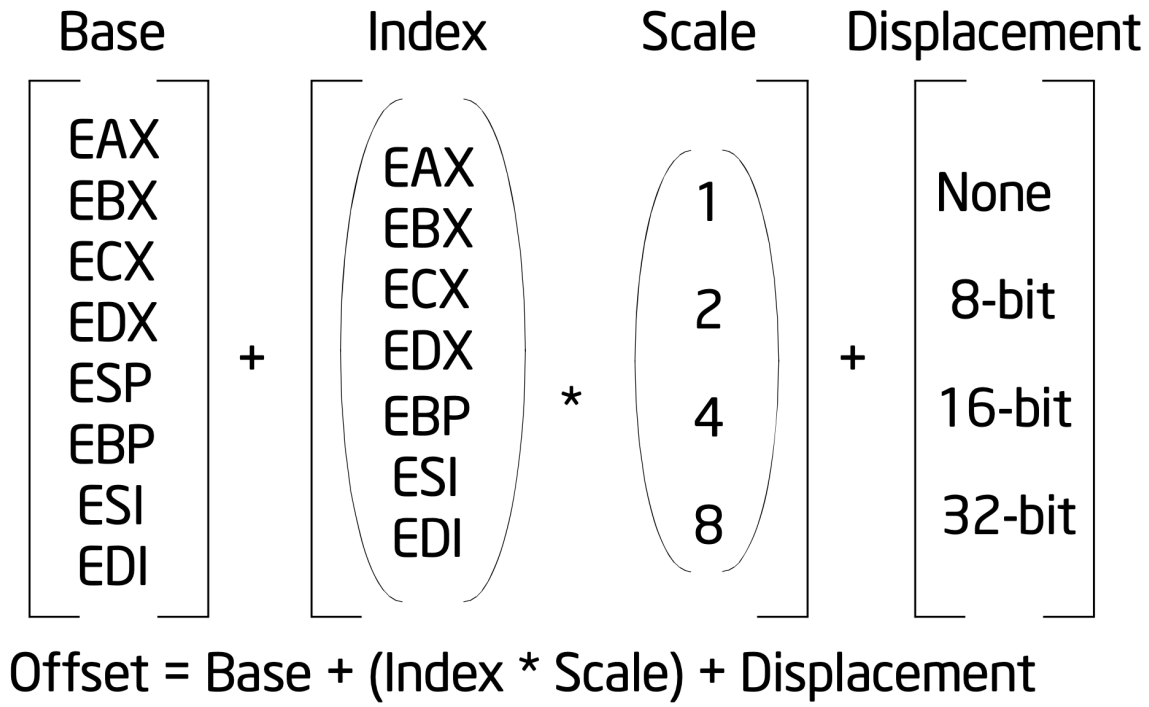

FIGURE 13 – Calcul d'adresse effective

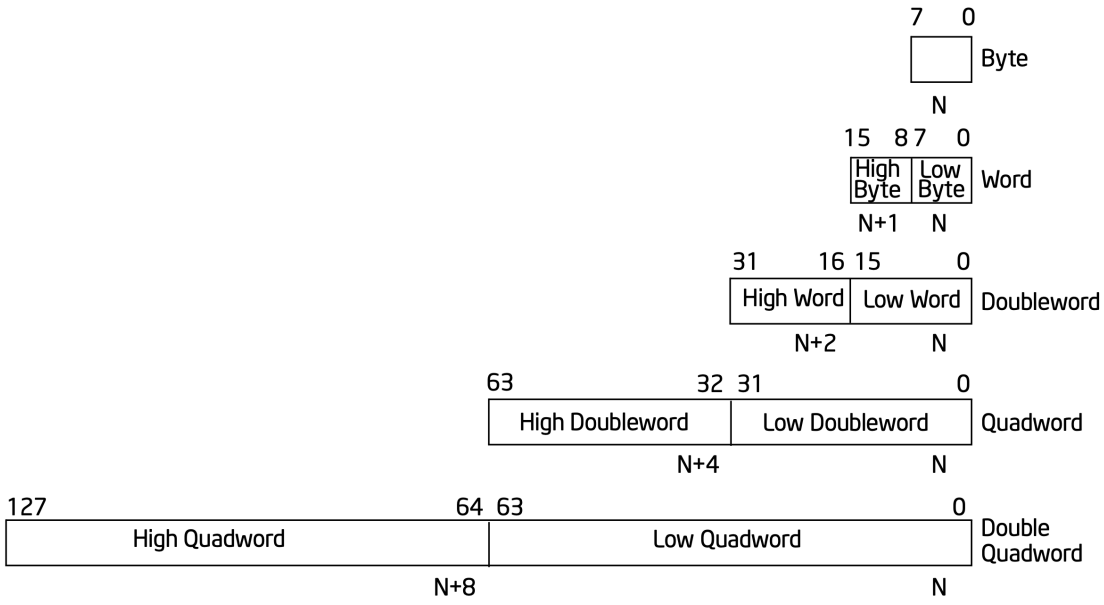

FIGURE 14 – Fundamental datatypes

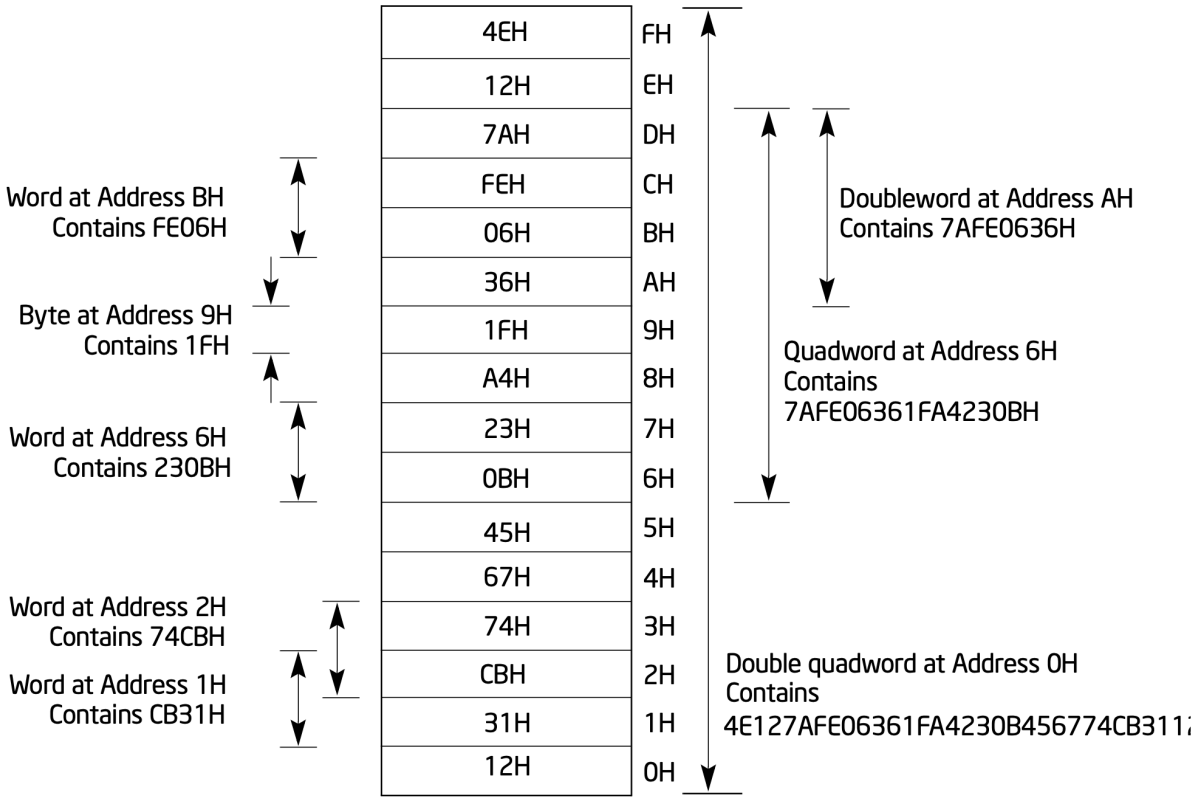

FIGURE 15 – Word in memory

composants : processeurs, mémoire, GPU. Elle conçoit également des cœurs qui implémentent ces architectures et concède des licences à des compagnies qui intègrent ces conceptions à leurs propres produits. Ainsi, *Arm Ltd.* conçoit des architectures externes et internes sans fabriquer elle-même de composants électroniques, elle vend uniquement de la propriété intellectuelle (*Intellectual Properties*).

Il y a eu au cours du temps plusieurs générations d'architectures ARM, l'ARM1 utilisait une structure interne de 32 bits avec un espace d'adressage sur 26 bits, limitant la mémoire principale à 64 Mo, architecture évoluant en plusieurs étapes pour avoir un espace d'adressage sur 32 bits, puis passer à une architecture en 64 bits à partir de l'ARMv8 en 2011. Plusieurs jeux d'instructions additionnels ont été introduits, comme l'extension Thumb avec des instructions en 32 et 16 bits permettant d'améliorer la densité du code ou encore Jazelle pour prendre en charge au niveau matériel le bytecode Java et JavaScript.

Dotés d'une architecture relativement plus simple que d'autres familles de processeurs et faibles consommateurs d'électricité, les processeurs ARM sont aujourd'hui dominants dans le domaine de l'informatique embarquée, en particulier la téléphonie mobile et les tablettes. De plus, les processeurs ARM sont aujourd'hui de plus en plus utilisés sur les postes de travail et les serveurs, comme par exemple les ordinateurs Apple se fondant sur des puces M1 ou les serveurs à base de Neoverse. Le supercalculateur actuellement le plus puissant du monde utilise une architecture de processeur ARMv8.2-A, fabriquée par Fujitsu avec 48 cœurs : le supercalculateur utilise 158 976 processeurs, soit 7 630 848 cœurs. [5]

Le Raspberry Pi 4 B comporte un processeur à l'architecture ARMv!. [6]

5. https ://www.r-ccs.riken.jp/en/fugaku/about/
6. SoC Broadcom BCM2711, Quad core Cortex-A72 1.5 GHz, voir `https://developer.arm.com/ip-products/proce`

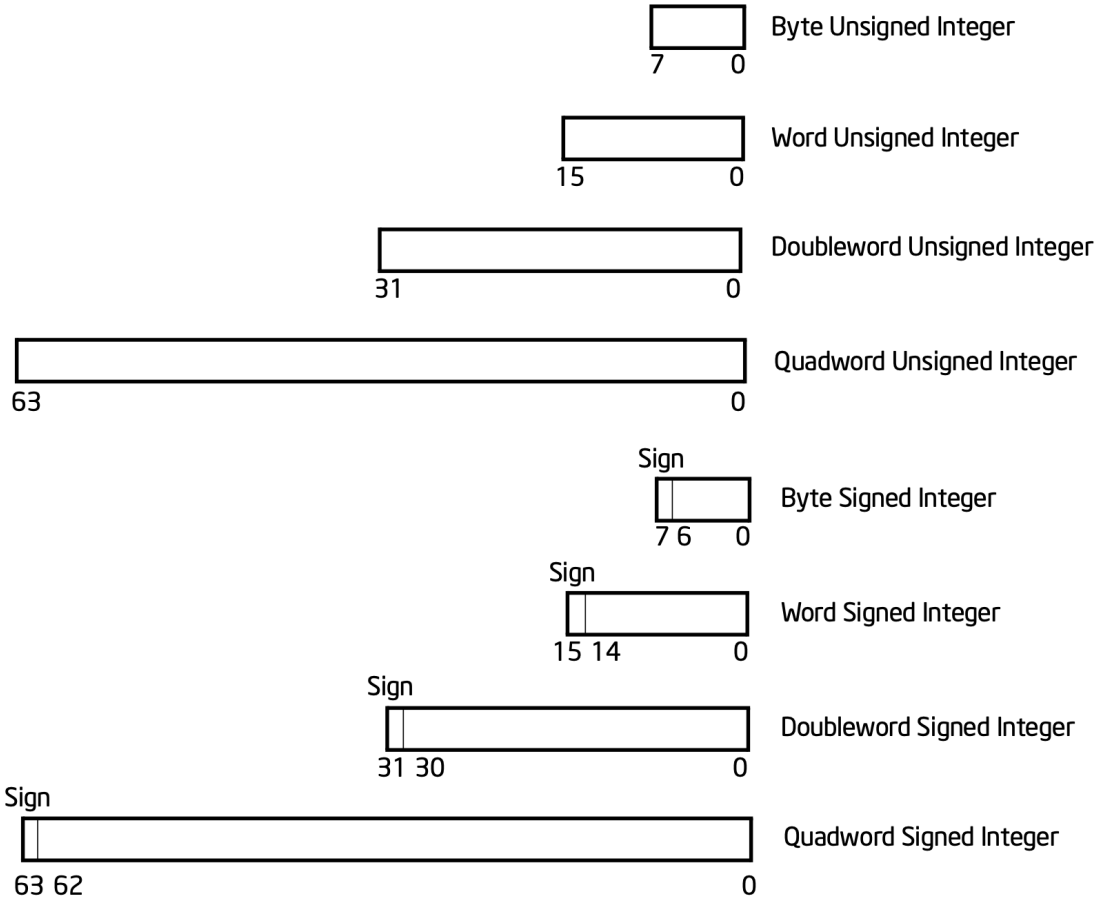

FIGURE 16 – Integers

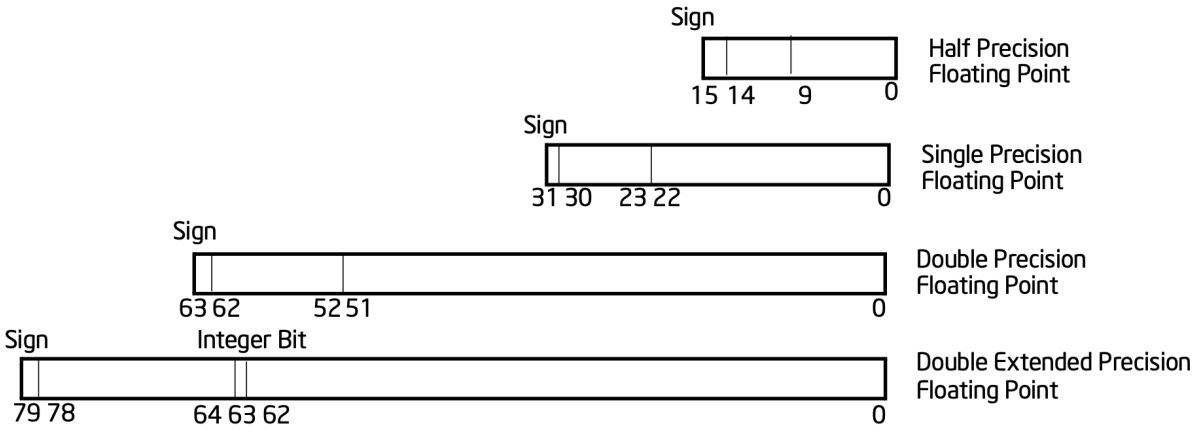

FIGURE 17 – Floats

Class		Two's Complement Encoding	
		Sign	
Positive	Largest	0	11..11
		.	.
		.	.
	Smallest	0	00..01
Zero		0	00..00
Negative	Smallest	1	11..11
		.	.
		.	.
	Largest	1	00..00
Integer indefinite		1	00..00
		Signed Byte Integer: Signed Word Integer: Signed Doubleword Integer: Signed Quadword Integer:	← 7 bits → ← 15 bits → ← 31 bits → ← 63 bits →

FIGURE 18 – Représentation des entiers signés : le complément à deux

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{15}	3.1×10^{-5} to 6.50×10^{4}
Single Precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}
Double Extended Precision	80	64	2^{-16382} to 2^{16383}	3.37×10^{-4932} to 1.18×10^{4932}

FIGURE 19 – Représentation des réels : intervalles des représentations à virgule flottante

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	–Zero	1	00..00	0	00..00
	–Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	–Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	-∞	1	11..11	1	00..00

FIGURE 20 – Représentation des réels : codage à virgule flottante

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	QNaN Floating-Point Indefinite	1	11..11	1	10..00
	Half-Precision Single-Precision: Double-Precision: Double Extended-Precision:		← 5Bits → ← 8 Bits → ← 11 Bits → ← 15 Bits →		← 10 Bits → ← 23 Bits → ← 52 Bits → ← 63 Bits →

FIGURE 21 – Représentation des réels : codage à virgule flottante (suite)

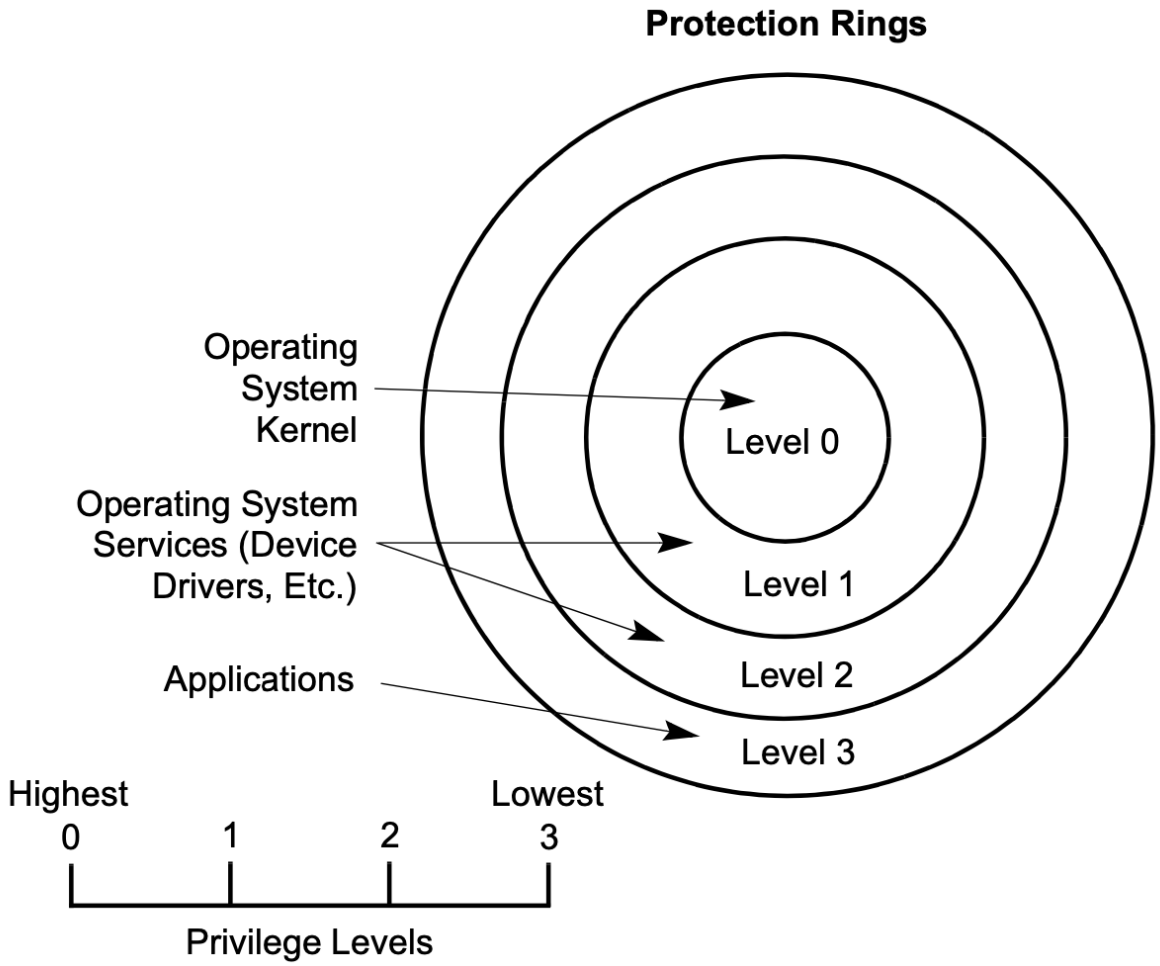

FIGURE 22 – Protection rings

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

Lors de la mise sous tension d'un ordinateur, le processeur principal, ou CPU, lance un logiciel interne, appelé en anglais *firmware*, déclenchant une séquence d'amorçage, appelée en anglais *boot* ou *bootstrap sequence*, qui doit se trouver dans une mémoire non volatile atteignable par le CPU.

Le terme anglais imagé *firmware* n'a pas d'équivalent en français car il représente quelque chose de *firm* – ferme – entre le *hardware* – dur – et le *software* – mou – : un logiciel stocké sur le matériel lui permettant de démarrer et d'effectuer des opérations de base sur les périphériques. En français ou trouve les termes micrologiciel, microcode ou encore logiciel interne, mais ils sont imprécis et ne comportent pas le jeu de mot sur *firm* : un logiciel fourni par la firme fabriquant le matériel.

Sur un PC, le *firmware* est stocké dans une mémoire de type EEPROM ou Flash et est appelé BIOS [7] ou UEFI [8]. Des logiciels internes similaires existents pour tous les types d'ordinateurs, télévisions, enceintes connectées, téléphones ou tablettes. Certaines séquences d'amorçages sont assez exotiques, comme celle du Raspberry Pi utilisant le GPU [9].

Lorqu'il est exécuté, le *firmware* teste et initialise les composants matériels présents dans l'ordinateur, identifie les périphériques et sélectionne parmi ces périphériques ceux utilisables pour poursuivre la séquence de démarrage, classés selon un certain ordre modifiable : clef USB, disque dur interne ou encore carte réseau en utilisant un protocole comme PXE [10].

Depuis un périphérique de stockage ou une carte réseau, le programme d'amorçage cherche à obtenir un programme exécutable appelé *bootloader*. Dans le cas d'un PC sous architecture x86, le BIOS sait lire les 512 premiers octets d'un disque, appelés le *Master Boot Record* (MBR). Ce MBR contient à la fois le *bootloader*, qui ne doit pas excéder 446 octets et la table des partitions de ce disque, limitée à 4 partitions de 2,2 To maximum. En pratique, les premiers 446 octets vont contenir un *bootloader* très simple qui va savoir lire dans une partition et charger un *bootloader* plus compliqué, comme GRUB, qui va lui-même être capable de charger d'autres fichiers binaires, capables eux de charger le noyau et donc de progressivement passer la main à l'OS. Dans le cas d'UEFI, qui est une technologie plus récente, outre la gestion du MBR pour de raisons de compatibilité, le *firmware* utilise le format de table de partition GPT, supportant jusqu'à 128 partitions de 9,4 Zo ($9,4 \times 10^{21}$ octets). UEFI peut directement charger un bootloader comme GRUB sur une partition au format FAT.

Dansle cas d'un chargement réseau de type PXE, supporté par les BIOS récents et par UEFI, le *bootloader* est obtenu en interrogeant un serveur DHCP qui fournit l'adresse IP d'un serveut TFTP fournissant le fichier contenant le *bootloader*.

Une fois que le *firmware* a obtenu le *bootloader*, il lui passe la main et ce dernier devient maître de la machine, tout en continuant à pouvoir utiliser des routines fournies par le BIOS, l'UEFI ou équivalent, notamment pour afficher des messages sur la console ou lire des données sur le disque.

FIGURE 23 – Le baron de Munchausen, gravure de Gustave Doré

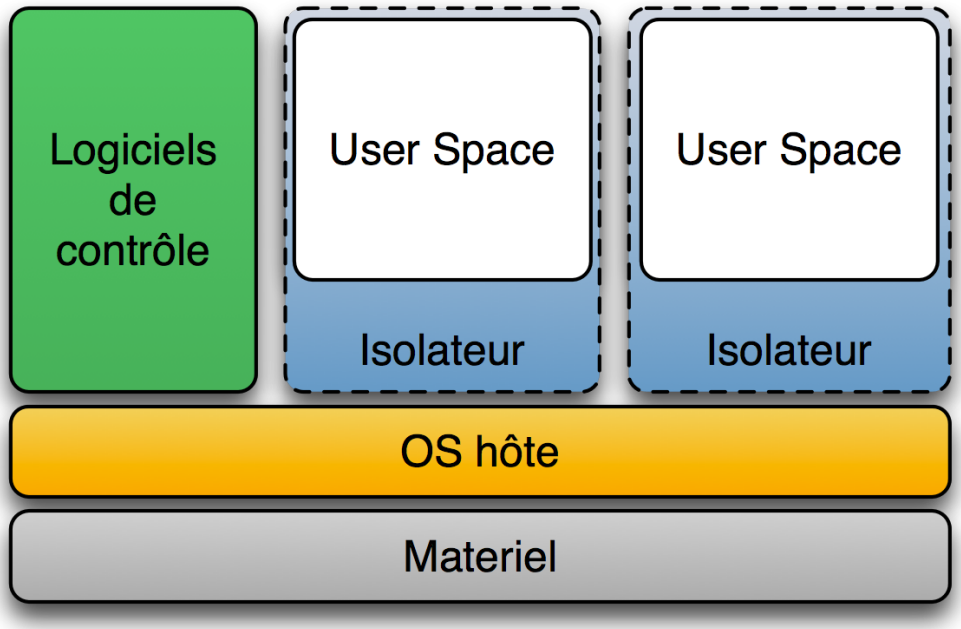

FIGURE 24 – Isolateur

Conteneurs, émulation et virtualisation

Isolateur

Exemples : l'ancêtre `chroot`, Jails de FreeBSD [11], Linux Containers [12], Docker [13], tous deux fondés sur les *cgroups* de Linux. [14]

Hyperviseur de type 2

Exemples : Oracle VM VirtualBox [15], QEMU.

Hyperviseur de type 1

Exemples : Xen [16], KVM [17].

---

ssors/cortex-a/cortex-a72 et https://datasheets.raspberrypi.org/rpi4/raspberry-pi-4-datasheet.pdf

7. Basic Input Output System
8. Unified Extensible Firmware Interface
9. https://www.raspberrypi.org/forums/viewtopic.php?t=6685
10. https://fr.wikipedia.org/wiki/Preboot_Execution_Environment
11. https://docs.freebsd.org/fr/books/handbook/jails/
12. https://linuxcontainers.org
13. https://www.docker.com
14. https://www.kernel.org/doc/Documentation/cgroup-v2.txt
15. https://www.virtualbox.org
16. https://xenproject.org
17. https://www.linux-kvm.org

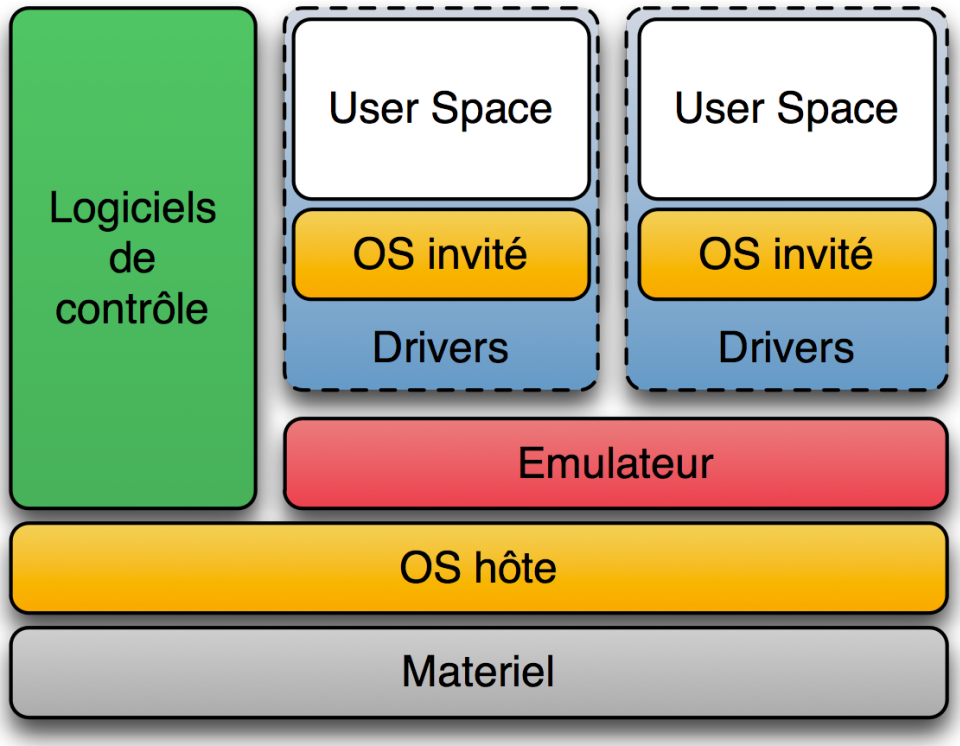

FIGURE 25 – Hyperviseur de type 2

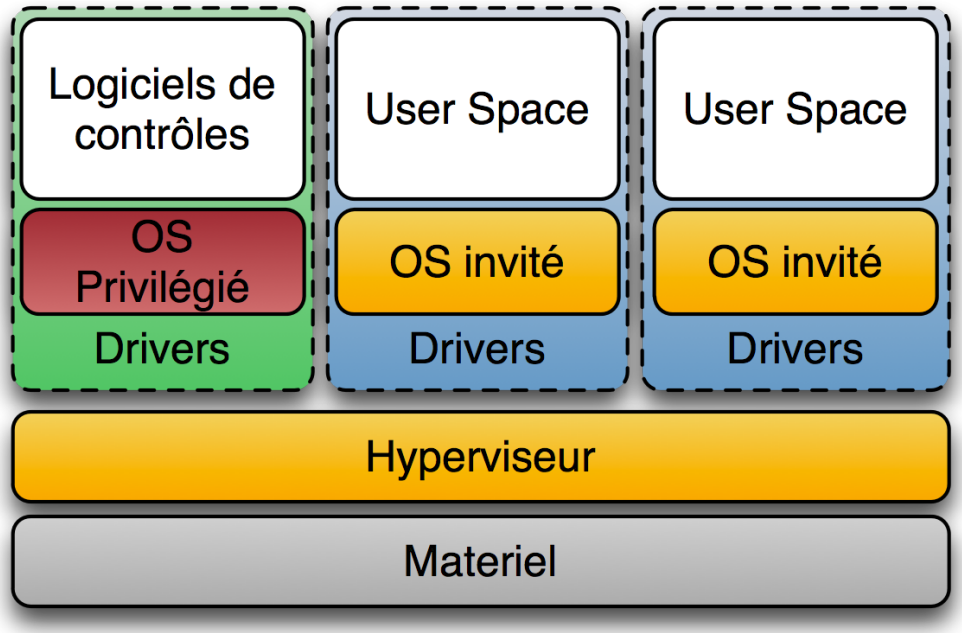

FIGURE 26 – Hyperviseur de type 1

Les fondements de la philosophie d'UNIX

Unix : 1969. *Everything is a byte stream.*

McIlroy : Unix Time-Sharing System Forward

— 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.

Ce qui peut se résumer ainsi :

> 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.`SionfaitabstractiondelencodagedescaractÃÍres,doncunivers elsionestdansuncodeASCII.`

Règles de Pike [18] reproduites 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. [19]
— **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. [20]
— **Rule 3.** 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. (Even if n does get big, use Rule 2 first.) For example, binary trees are always faster than splay trees for workaday problems.
— **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. The following data structures are a complete list for almost all practical programs :

  — array
  — linked list
  — hash table
  — binary tree

18. `https://www.lysator.liu.se/c/pikestyle.html`, `http://www.literateprogramming.com/pikestyle.pdf`
19. On peut quand même parfois *savoir* où le programme va passer du temps. En lien avec la phrase de Hoare/Knuth : « *premature optimization is the root of all evil* ».
20. À discuter, énergie et code performant sont indispensables aujourd'hui.

Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters. [21]

— **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. (See Brooks p. 102.)

> Those who do not understand Unix are condemned to reinvent it, poorly.
>
> *Usenet signature, November 1987*
> — Henry Spencer

Références

[1] *Compiler explorer*, `https://godbolt.org`

[2] Philipp Opperman, *Writing an OS in Rust*. `https://os.phil-opp.com`

[3] *Rust by example*. `https://doc.rust-lang.org/rust-by-example/index.html`

[4] *Buildroot, making embedded Linux easy*. `https://buildroot.org`

[5] *OSDev.org, Introduction*, 2020. `https://wiki.osdev.org/Introduction`

[6] *Gabian, ingénierie de la récupération*, 2021. `https://gabian.systems`

[7] Linus Benedict Torvalds, *Collection d'emails commentés retraçant la genèse de Linux*,1992. `https://www.cs.cmu.edu/~awb/linux.history.html`

[8] Luiz André Barroso, Jeffrey Dean, Urs Hölzle, WEB SEARCH FOR A PLANET : THE GOOGLE CLUSTER ARCHITECTURE, Published by the IEEE Computer Society

[9] *The original sources of MS-DOS 1.25 and 2.0*, `https://github.com/microsoft/MS-DOS`

[10] Félix Cloutier, *x86 and amd64 instruction reference*, 2019. `https://www.felixcloutier.com/x86/index.html`

[11] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`

[12] Adam Finch, *Moon Machines - 3. « L'ordinateur de navigation »*, 2008. `https://www.youtube.com/watch?v=DWcITjqZtpU`

[13] *nInvaders*. `http://ninvaders.sourceforge.net`

[14] Busybox. `https://busybox.net`

[15] David Both, *An introduction to the Linux boot and startup processes*, 2017. `https://opensource.com/article/17/2/linux-boot-and-startup`

[16] QEMU. `https://www.qemu.org`

[17] Rust Book. `https://doc.rust-lang.org/book/`

[18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Three Easy Pieces*. `https://pages.cs.wisc.edu/~remzi/OSTEP/`

[19] Searchable Linux Syscall Table for x86 and x86_64. `https://filippo.io/linux-syscall-table/`

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