

Chapitre 6

Mémoire



Une mémoire peut être classée selon différents critères comme la volatilité (volatile ou non), l'accessibilité (accès aléatoire RAM ou séquentiel), l'adressage (par octet ou associatif), la capacité, les performances ... En voici quelques uns.

Mémoire volatile : mémoire ne conservant pas les informations stockées lorsqu'elle est mise hors tension.

Ex : SDRAM, DDRAM, mémoire cache, registres du processeur.

Mémoire non-volatile : mémoire conservant les données même hors tension.

Souvent des médias de stockage de masse (HDD, SSD, SD-card, DVD, Blu-Ray, ...), mais aussi la mémoire du BIOS ou l'EEPROM des micro-contrôleurs.

Mémoire morte ou ROM (Read-Only Memory): mémoire non-volatile accessible uniquement en lecture, elle est donc pré-programmée.

Ex. : mémoire du BIOS, anciens systèmes embarqués, ...

Mémoire vive ou RAM (Random-Access Memory): mémoire généralement volatile accessible en lecture et écriture.

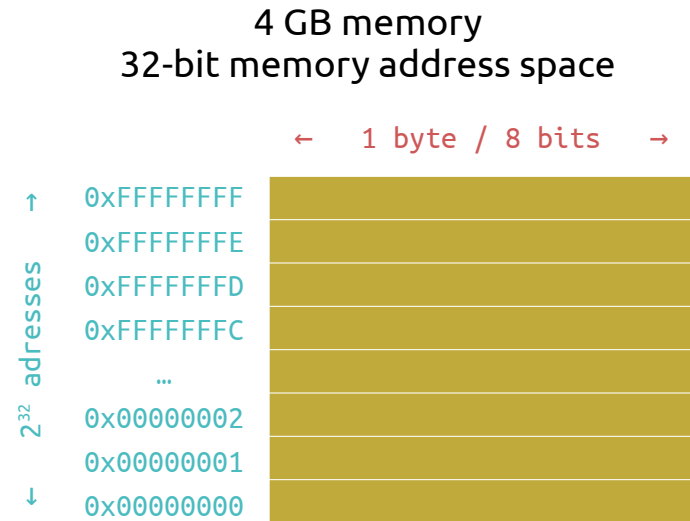
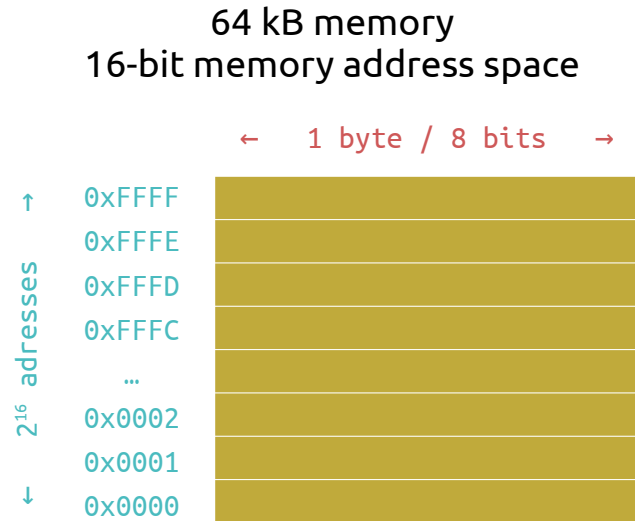
Ex : mémoire principale (SDRAM, DDRAM) d'un ordinateur ou micro-contrôleur.

Attention : beaucoup d'ambiguïtés autour de ces termes.

Dans ce cours nous parlerons de mémoire de stockage de masse (disque dur, ...) et de mémoire principale (RAM).

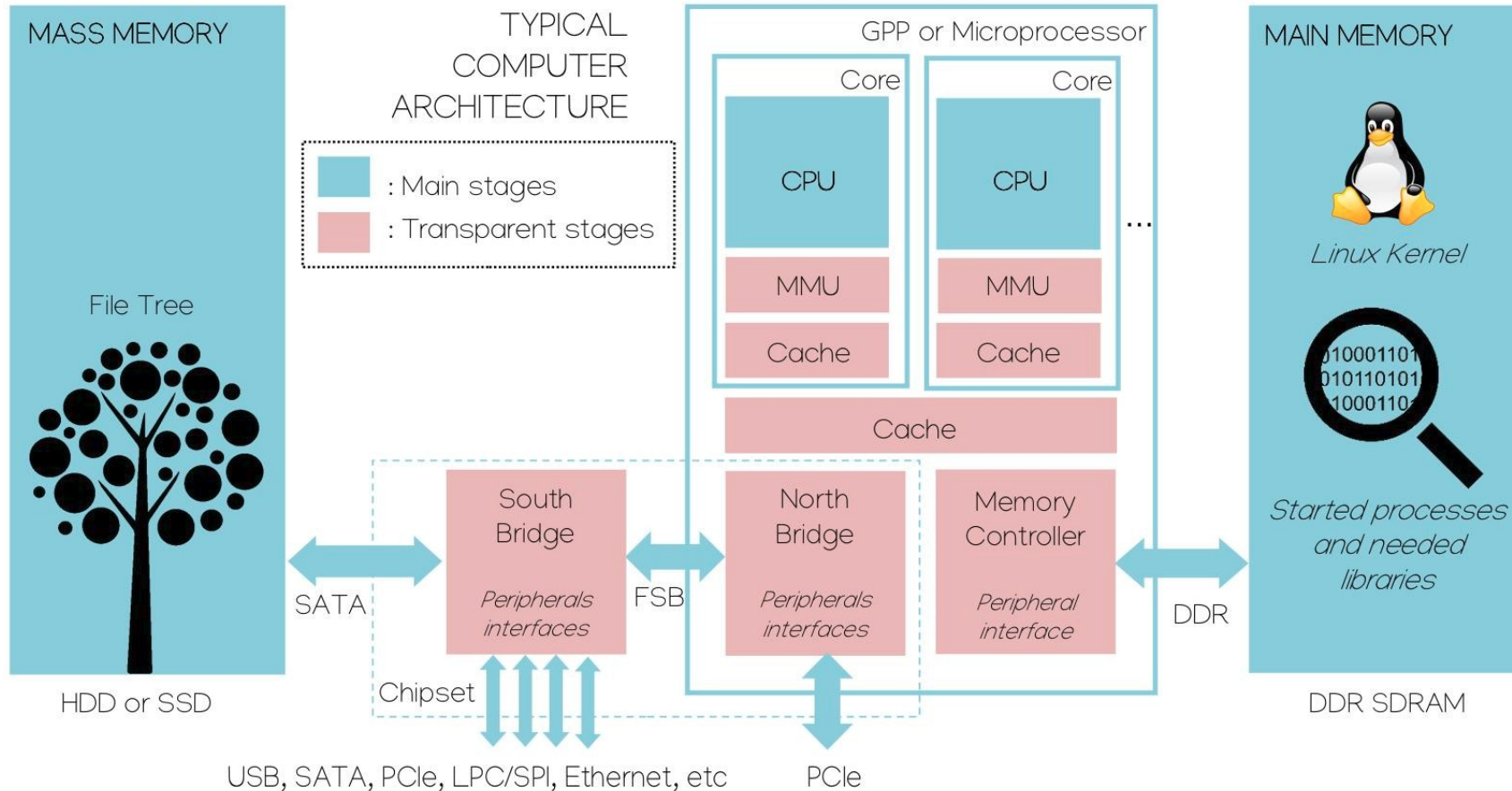
Mémoire adressable par octet : beaucoup de familles de mémoire d'ordinateur ou de processeur embarqué (MCU, DSP, SoC, ...) sont adressables par octets.

À chaque adresse mémoire correspond 1 octet.





Architecture générique d'un ordinateur



Analyses de circuit électroniques : [Deus Ex Silicium](#)

Parfait pour l'analyse d'ordinateurs sur-mesure ;)

Processeur, SSD, RAM, carte mère, South bridge, alimentation ...



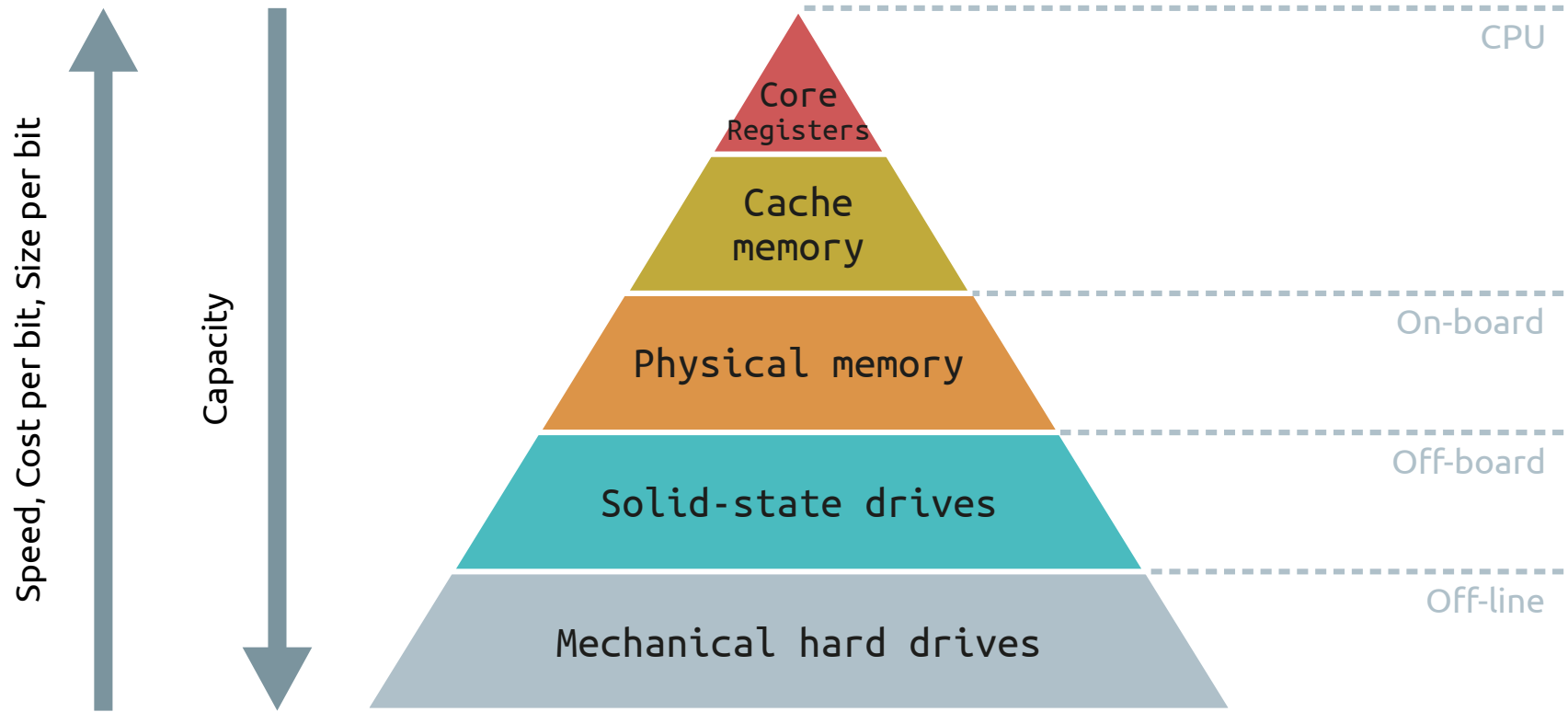
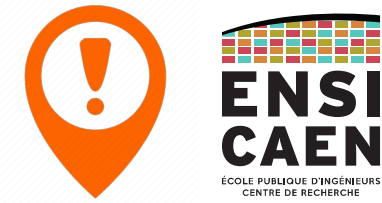
Démontage et remontage d'une Xbox :

[XBox Series X - Décorticage intégral, mesures et analyses](#)

Démontage définitif d'une Nintendo Switch :

[Au plus profond des entrailles de la Nintendo Switch OLED](#)

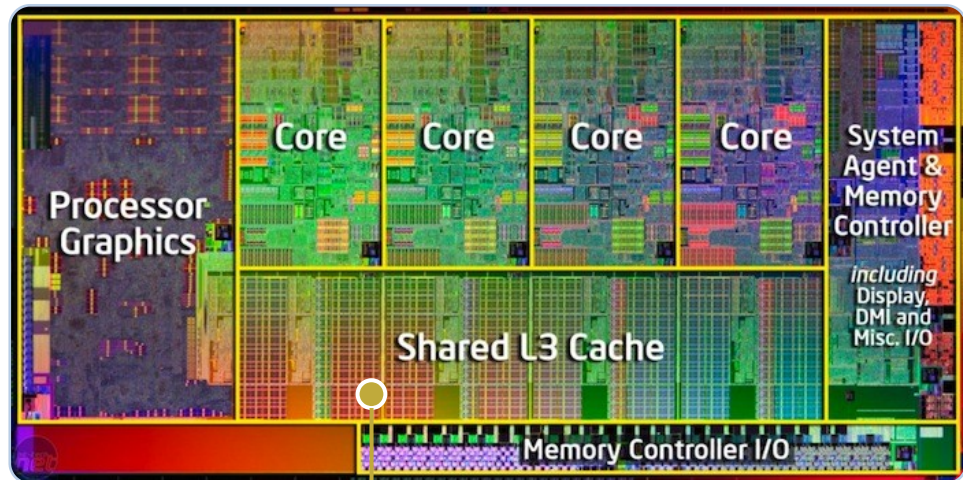
Hiérarchie mémoire (modèle en couches)



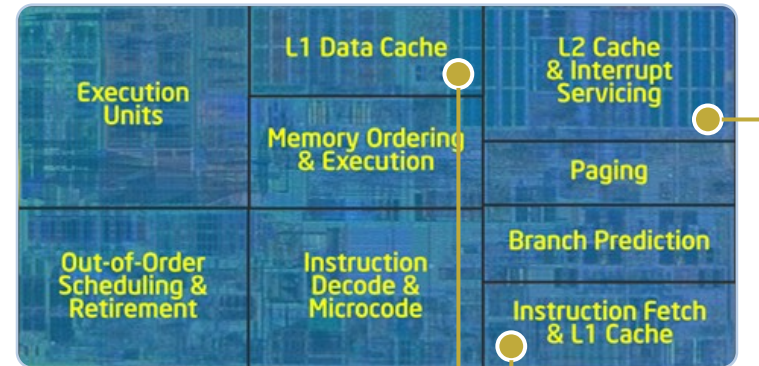
Hiérarchie mémoire (modèle en couches)

Le modèle en couche peut être affiné en fonction des différentes technologies d'intégration. Par exemple, l'empreinte sur silicium de 32 ko de mémoire cache L1 n'est pas la même que pour 32 ko de cache L2 ou encore L3.

Exemple d'un GPP Intel Sandy Bridge (die complet à gauche, zoom sur le core à droite).



6 Mo de cache L3



2x32 ko de cache L1D/L1P

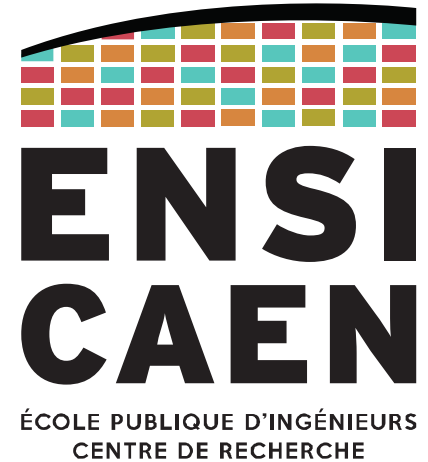
256 ko de cache L2

STOCKAGE DE MASSE

Supports physiques

Volumes logiques

Système de fichiers



Définition

Les mémoires de stockage de masse sont des **mémoires non-volatiles**, destinées à stocker de **grandes quantités de données** sur le long terme y compris **en l'absence d'alimentation** (« grandes » étant à mettre dans le contexte historique ou applicatif).

Les mémoires modernes utilisent une représentation logique des données sous forme de fichiers (voir « *file system* »), mais ça n'a pas toujours été le cas.

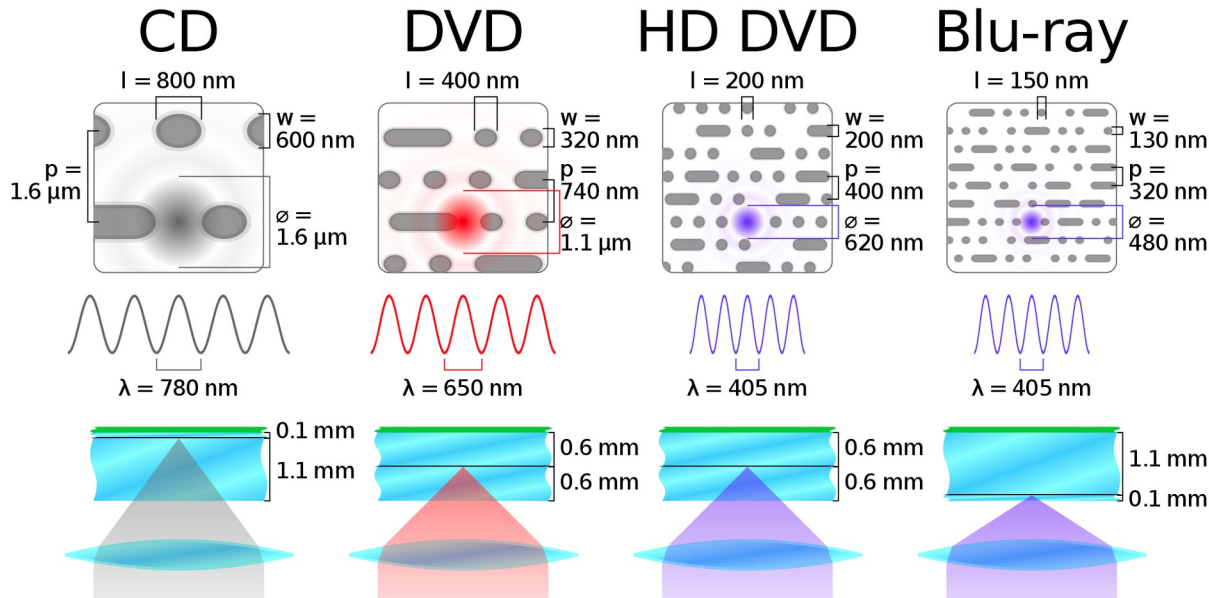
Les principales caractéristiques recherchées sont la capacités de stockage, la vitesse de lecture/écriture, la durée de vie des données et le coût.

La fonction de **stockage de masse** a été effectuée par plusieurs technologies, dont les plus connues sont le stockage magnétique, optique ou électronique.

Supports optiques

Les supports optiques sont généralement des **supports amovibles** et généralement en **lecture seule** (distribution de logiciels, musiques, films, ...).

La gravure sur disque se fait par laser, la densité des informations dépendant de la longueur d'onde, du diamètre et de l'angle d'ouverture du-dit laser.



CD-ROM : 650-700 MB
Licence Sony et Philips

DVD : 4,7 – 8 GB
Licence DVD forum

Blu-ray : 25-50 GB
Blu-ray UHD : 100 GB
Licence Sony

Comparaison des caractéristiques physiques des quelques supports optiques.

Les disquettes (*floppy disks*) et **disques durs (HDD, *Hard Drive Disk*)** utilisent le stockage magnétique.

Les HDD sont moins rapides que les SSD, mais coûtent également bien moins cher à capacité égale.



Floppy disks

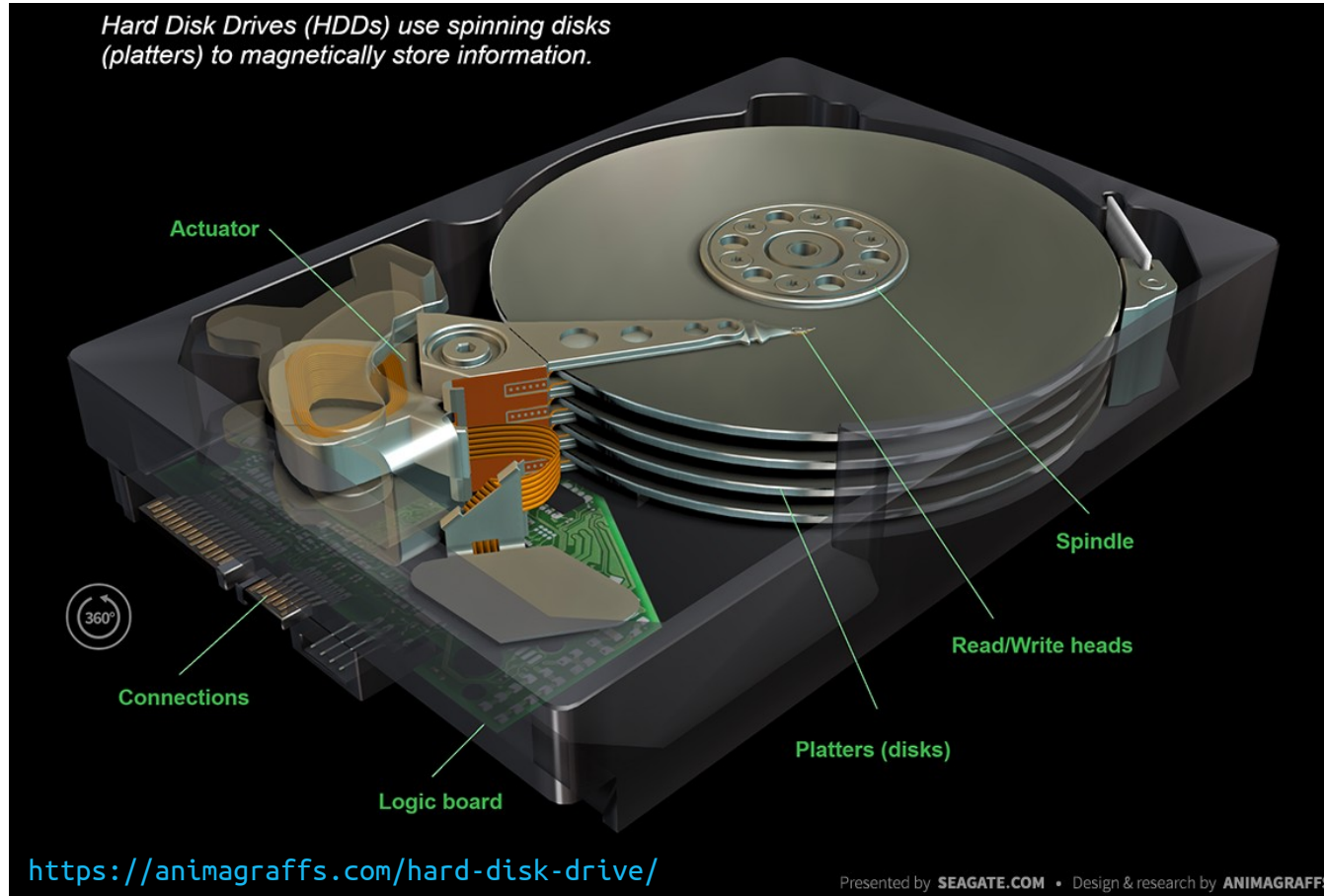
8-inch ~243 kB ; 5^{1/4}-inch = 360 kB ; 3^{1/2}-inch = 1.44 MB



Hard Drive Disk (HDD)

3.5-inch / 2.5-inch

Disque dur mécanique (HDD)



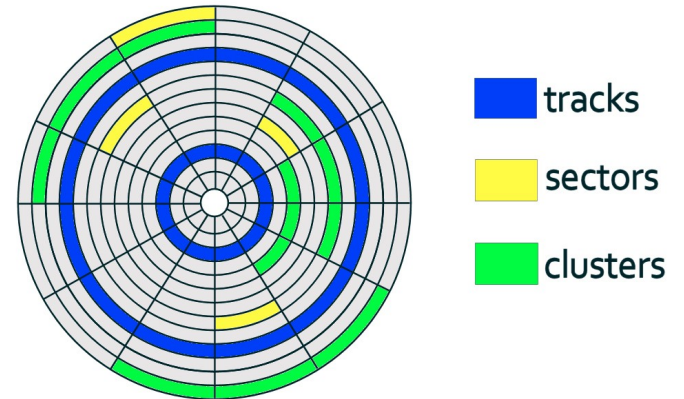
Disque dur mécanique (HDD)

Contrairement aux mémoires adressables par octet, les médias de stockage de masse sont généralement adressables par blocs.

Par exemple, un disque dur (HDD) est découpé en secteurs de 512 octets. Un fichier se trouve vraisemblablement réparti sur plusieurs secteurs, et pas forcément contigus.

Le *file system* a donc aussi le rôle de présenter à l'utilisateur des fichiers entiers (alors qu'ils sont morcelés sur le support physique) et regroupés par dossiers (alors qu'ils sont disséminés sur tout le support physique).

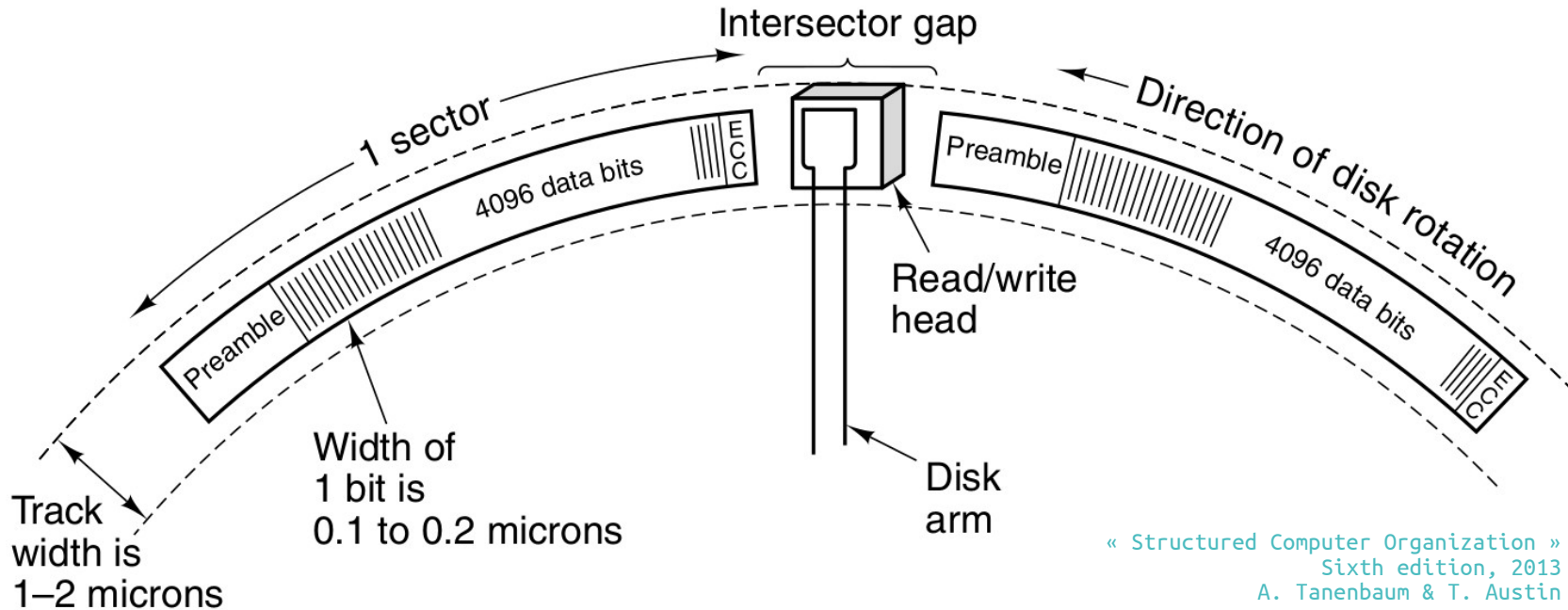
Hard disk drive structure



Disque dur mécanique (HDD)

Le secteur d'un HDD commence par un préambule, chargé de synchroniser la tête de lecture. Arrivent ensuite les 512 octets de données, puis un ECC (*Error-Correcting Code*).

Si on ajoute à cela l'espace inter-secteur (espace résiduel séparant deux secteurs successifs), on peut affirmer que la capacité réelle d'un HDD est environ égale à 85 % de la capacité affichée.

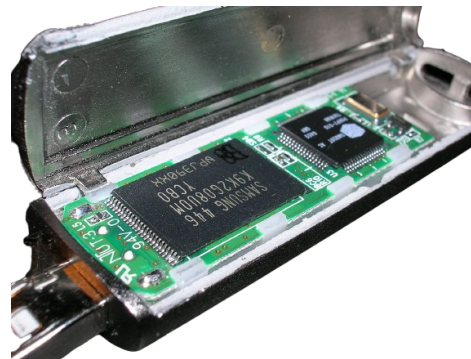


Les **EEPROM** utilisent de la circuiterie basique pour stocker des charges électriques. La technologie EEPROM la plus commune est la **mémoire Flash** (NAND et NOR), qui a un temps constant d'accès à l'information.

Les clés USB, cartes SD, **SSD (Solid-State Drive)** utilisent aussi une technologie Flash.



120 GB, 2.5-inch Samsung SSD



Flash drive / clé USB
(Mémoire à gauche, MCU à droite)

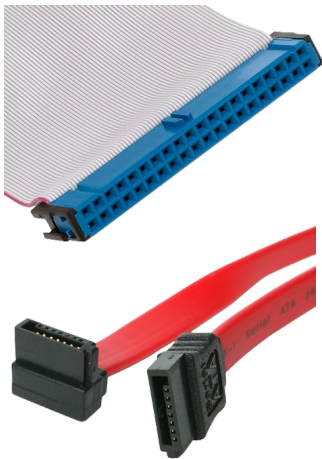


8-GB SD Card
Internal circuit

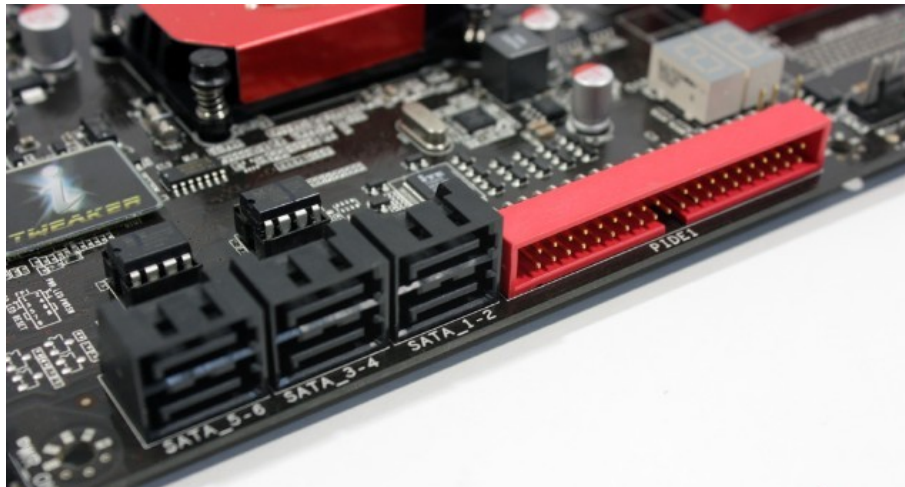
Fin 80-début 2000, les disques durs et autres lecteurs utilisaient la norme **Parallel-ATA** (ou **IDE (*Integrated Drive Electronics*)**) pour communiquer avec la carte mère.

Elle a été remplacée début des années 2000 par **S-ATA (*Serial ATA*)**, toujours utilisée.

Avantages : débit jusqu'à 6 Gb/s (norme III), *hot-plug* et moins de connexions que P-ATA.



Câbles P-ATA (haut)
et S-ATA (bas)

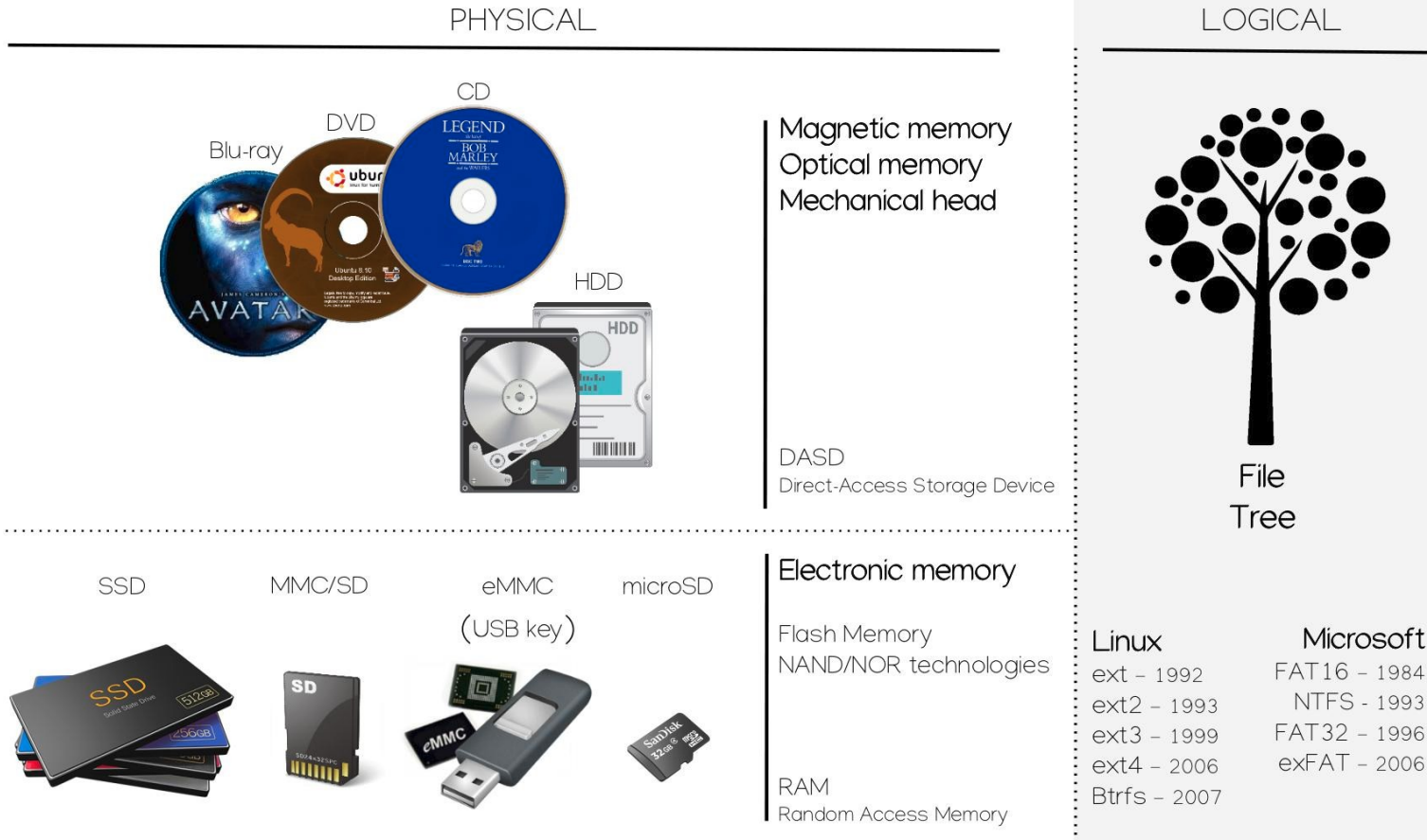


6 ports S-ATA (à gauche)
Et 1 port P-ATA (à droite)



SSD format 2.5-inch (haut)
et format M.2 (bas)

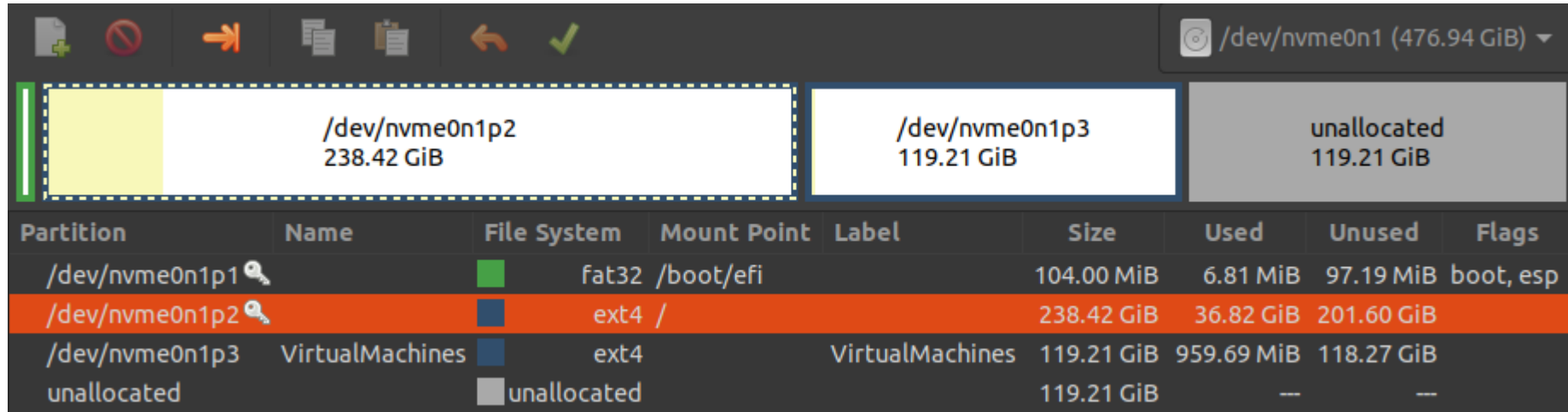
Volume physique vs. Volume logique



Volume physique vs. Volume logique

Un **volume physique** (HDD, clé USB, ...) peut être découpé en plusieurs **volumes logiques (partitions)**.

C'est généralement le cas des ordinateurs en dual-boot, qui vont avoir une partition pour chaque OS voire aussi une partition partagée entre les deux OS, pour les données.



Partition	Name	File System	Mount Point	Label	Size	Used	Unused	Flags
/dev/nvme0n1p1		fat32	/boot/efi		104.00 MiB	6.81 MiB	97.19 MiB	boot, esp
/dev/nvme0n1p2		ext4	/		238.42 GiB	36.82 GiB	201.60 GiB	
/dev/nvme0n1p3	VirtualMachines	ext4		VirtualMachines	119.21 GiB	959.69 MiB	118.27 GiB	
unallocated		unallocated			119.21 GiB	—	—	

Affichage des partitions d'un disque avec gparted (Ubuntu)

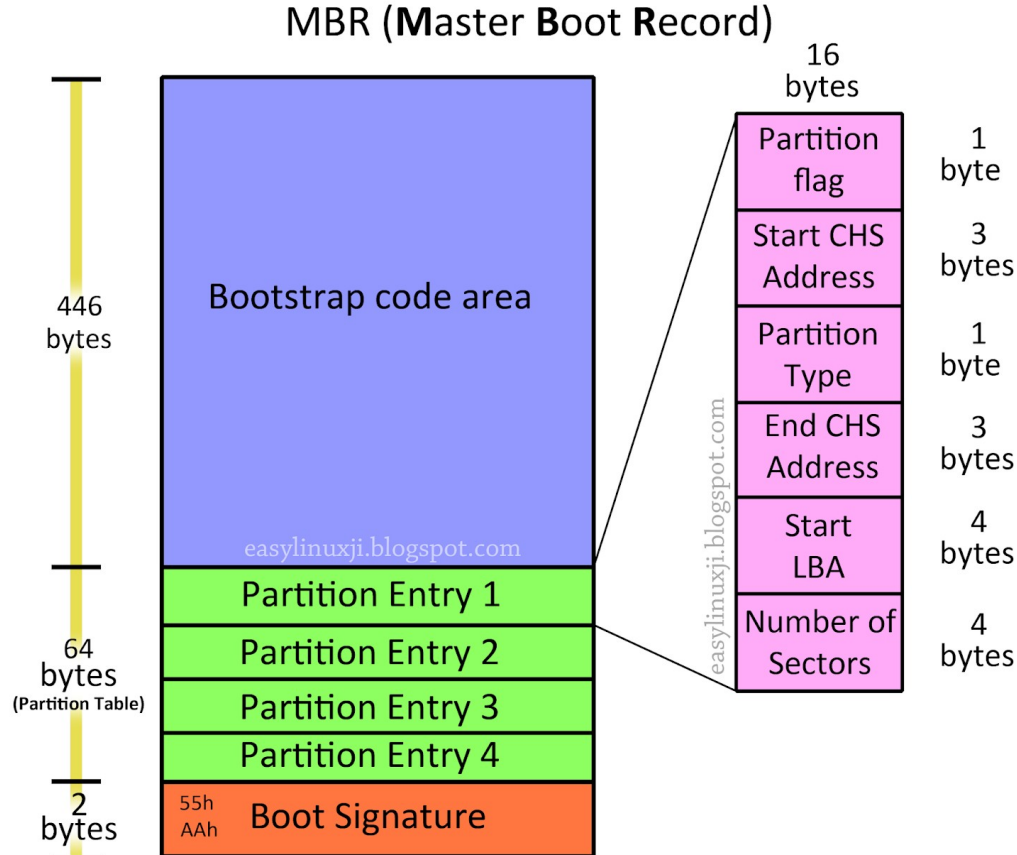
Avec plusieurs partitions sur un même support physique, il faut qu'un système soit capable d'identifier toutes les partitions d'un disque.

Pour cela, on trouve un **table des partitions** au début du volume physique et en espace non-partitionné (donc hors de toute volume logique).

Il s'agit d'une zone purement binaire (pas de fichier) pouvant être au format **MBR (Master Boot Record)** ou **GPT (GUID Partition Table)**.

C'est cette zone que le BIOS (ou UEFI) ira lire pour détecter la partition à démarrer, ou que l'OS va lire pour afficher l'image de la page précédente.

Table des partitions



Le **MBR (Master Boot Record)** est un vieux standard (1983) développé par PC DOS 2.0 (IBM).

Désigne maximum 4 partitions.

Volumes physiques de taille < 2 TiB (2^{32} x 512-Byte).

Ces limitations ont fait diminuer son usage dans l'informatique, mais restent un avantage pour les systèmes embarqués.

Le **GUID Partition Table (GPT)** est apparu à la fin des années 90, développé par Intel pour s'allier avec son UEFI (remplaçant du BIOS).

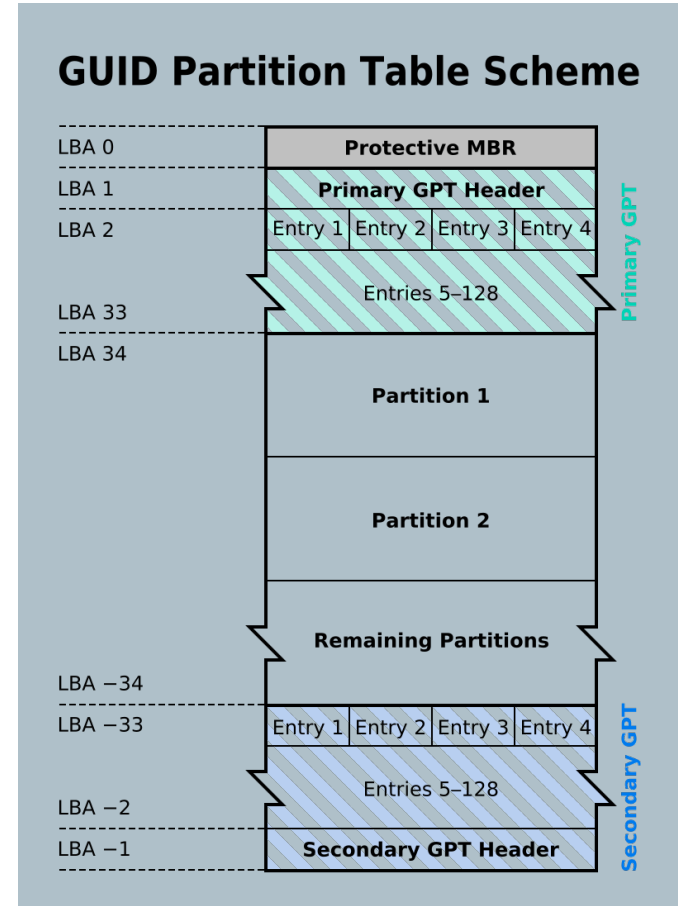
Vu les inconvénients du MBR, le GPT a rapidement remplacé son ancêtre dans l'informatique professionnelle et personnelle.

Maximum 128 partitions

Maximum 8 ZiB (2^{64} x 512-Byte sectors)

CRC32 checksum

GUID = Globally Unique Identifier



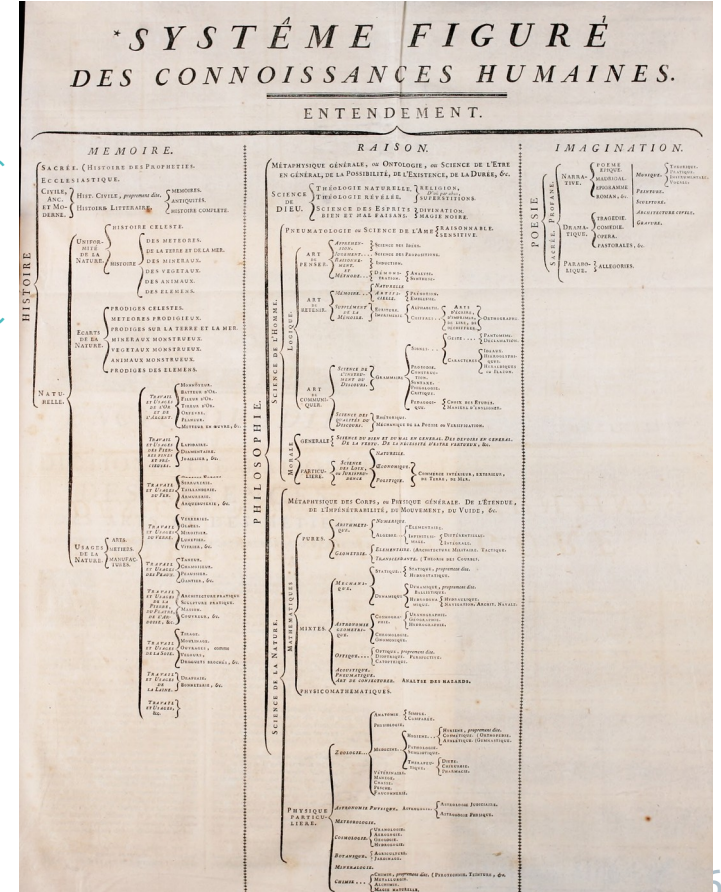
Au sein d'une partition, les fichiers sont organisés selon un **système de fichiers (FS, File System)**.

C'est ce qui permet à l'utilisateur d'observer ses fichiers sous forme d'une arborescence.



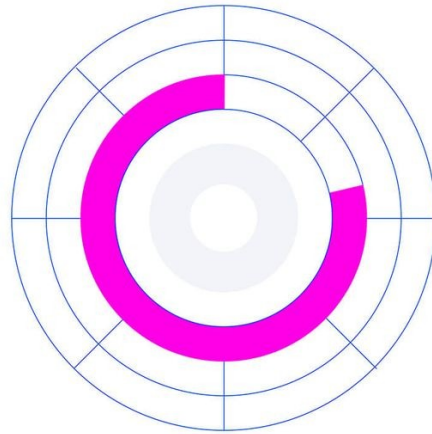
Archivage de cartons contenant jusqu'à 2000 cartes perforées (NARA, 1959).

« Système figuré des connaissances humaines » ou « Arbre de Diderot et d'Alembert » (1751-1772)

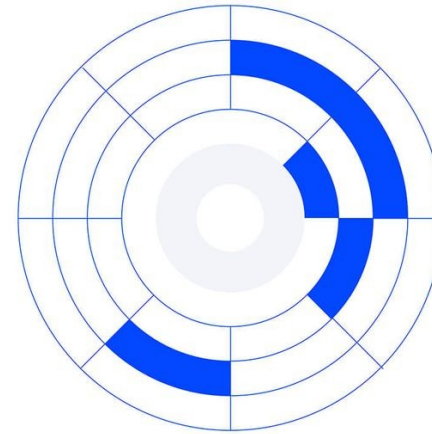


De plus, les fichiers étant généralement répartis sur plusieurs secteurs, il est primordial de connaître l'emplacement successif des différents secteurs formant le fichier.

Ceci est également le rôle du *file system*.

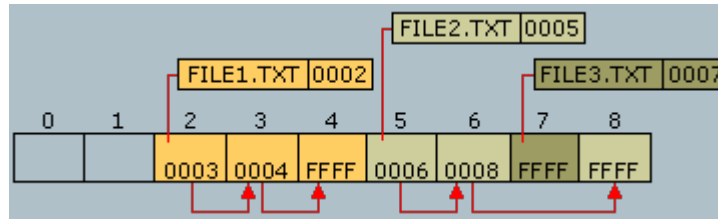


Non-Fragmented file



Fragmented file

FAT (1977, pour Windows) est un *file system* extrêmement simple, travaillant par clusters (groupes de secteurs). Il existe une table liant les noms de fichiers à l'index du premier cluster de chaque fichier. Chaque cluster occupé contient une partie du fichier et l'indice vers le cluster suivant.



Depuis sa naissance, de nombreuses versions sont apparues (FAT16, FAT32, exFAT, ...). Déprécié par Windows au profit de NTFS, il reste très répandu pour les médias amovibles.

File system

Le principal *file system* utilisé par Windows est **NTFS (New Technology File System)** arrivé en 1993 avec Windows NT 3.1.

Pendant longtemps (avant Windows 10), les OS de Microsoft n'ont su gérer que les *file systems* FAT et NTFS.

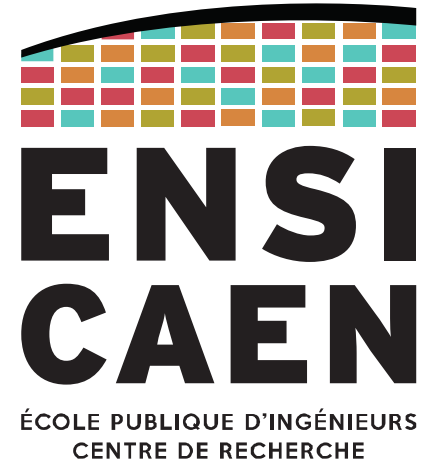
Les *file systems* conçus pour Linux sur ordinateur sont ceux de la famille **ext** (**ext** en 1992, **ext2**, **ext3** et **ext4** depuis 2006), même si Linux a toujours su supporter de nombreux *file systems* (y compris FAT et NTFS).

Depuis l'intégration en 2016 de WSL (*Windows Subsystem for Linux*) dans Windows 10, l'OS de Redmond supporte le *file system* ext4 (même si ce n'est pas si simple).

MÉMOIRE PRINCIPALE

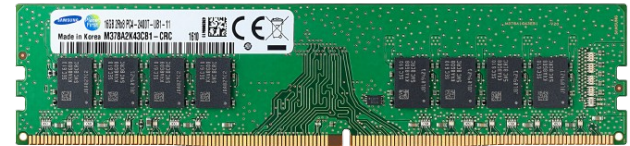
AU NIVEAU COMPOSANT :

- Quelques technologies
- Cellule mémoire
- Mot / *Word*
- Boutisme / *Endianness*



La **mémoire principale** (*main memory*) ou parfois **mémoire de travail** est la mémoire dans laquelle se trouvent les informations en cours d'utilisation par le processeur.

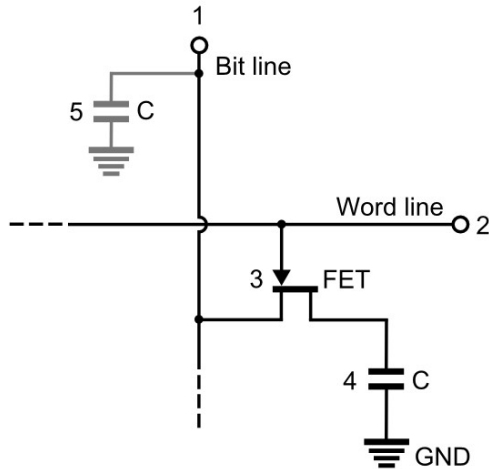
Ces informations peuvent être des données seulement (cas d'un MCU) ou des données ET des instructions (cas d'un smartphone ou ordinateur).



Mémoire principale dans un MCU (en interne), smartphone (chip) et ordinateur (barrette).

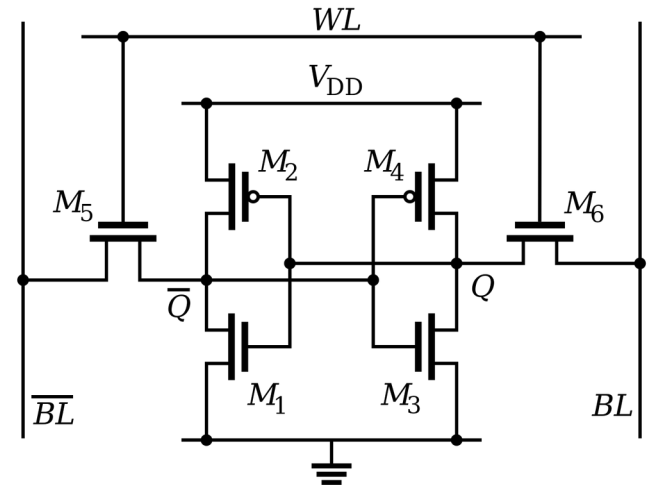
Technologiquement, la mémoire principale est une **RAM (Random Access Memory)** et plus précisément d'une **DRAM (Dynamic RAM)**.

La **DRAM** doit être régulièrement rafraîchie (qq ms) à cause du courant de fuite de ses pico-condensateurs. Utilisée pour la mémoire d'ordinateur. Faible empreinte silicium mais plus lente que la SRAM (voir mém. cache).



DRAM
1 bit requires 1 transistor
and 1 pico-capacitor

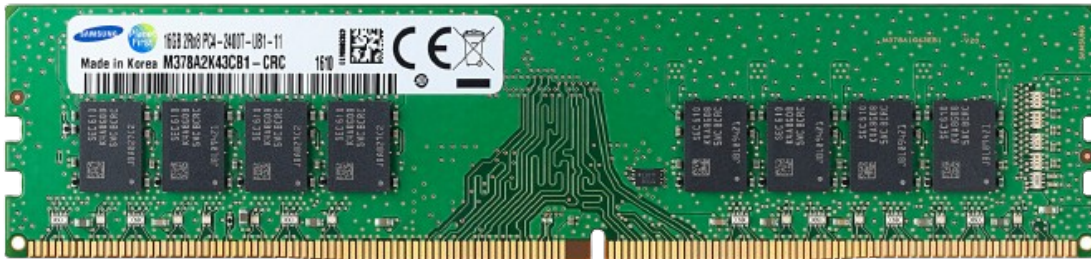
SRAM
1 bit requires 6
CMOS transistors



Les RAM utilisées aujourd'hui fonctionnent en **DDR5 SDRAM** (*5th generation of Double Data Rate Synchronous DRAM*).

Les premières mémoires en DDR (1998) donnaient une bande passante de 1600 Mo/s pour une fréquence d'horloge de 100 MHz (soit 200 MTransferts/s).

Le dernier standard DDR5 (2021, DDR5-7200 de Samsung) donne un débit théorique de 57000 Mo/s avec une horloge cadencée à 3,6 GHz (soit 7200 MTransferts/s).



Samsung 16 GB DDR4 SDRAM



Crucial SO-DIMM 16 GB DDR4 SDRAM

Case mémoire

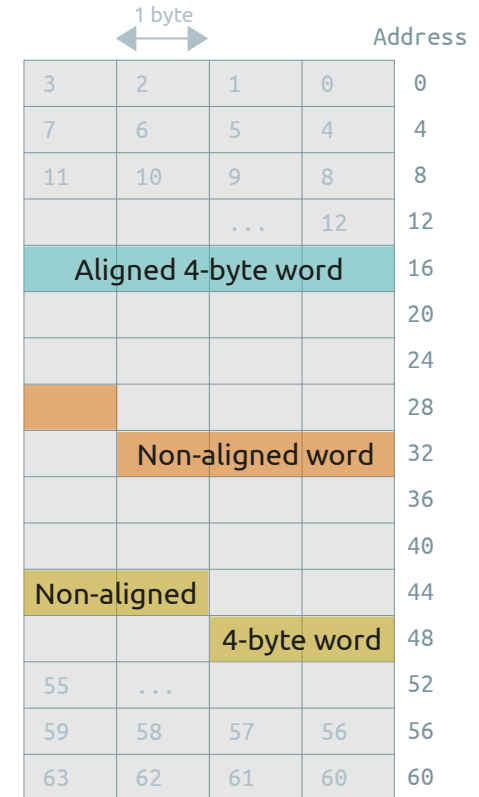
La plus petite quantité de donnée manipulable par un processeur est l'**octet** ou **Byte** (à quelques exceptions près).

Cela signifie que chaque octet en mémoire principale dispose de sa propre adresse : on l'appelle alors **case mémoire** ou **cellule mémoire**.

Plusieurs octets peuvent être associés pour contenir une seule et même donnée : ceux-ci forment un **mot (word)**.

La taille d'un mot varie selon les architectures, les GPP actuels manipulent des mots de 4 ou 8 octets (32 ou 64 bits).

Un mot de n octets est **aligné en mémoire** si son adresse est modulo n , sinon il est dit non-aligné.



64-byte memory

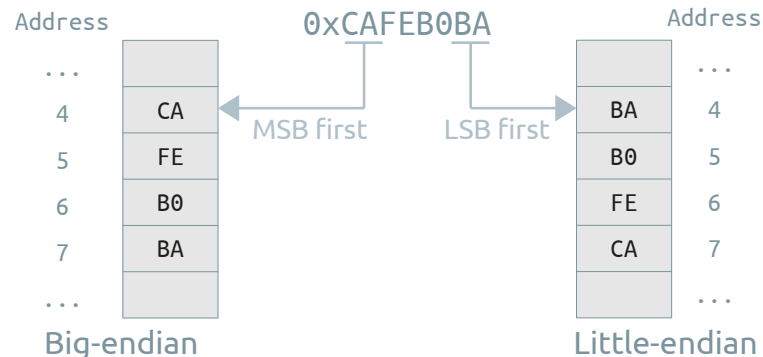
Boutisme

Les octets d'un mot/*word* peuvent être stockés dans un ordre ou dans l'autre. La stratégie employée s'appelle le **boutisme** (*endianness*).

Le **gros-boutisme** (*big-endian*) consiste à stocker l'octet de poids fort (MSB) en premier dans la mémoire, donc à l'adresse la plus petite. C'est par exemple utilisé dans les communications TCP/IP

Le **petit-boutisme** (*little-endian*) consiste à stocker l'octet de poids faible (LSB) en premier dans la mémoire, donc à l'adresse la plus petite. C'est ce qui est classiquement utilisé sur les machines Intel.

L'avantage du petit-boutisme est la simplicité du matériel, alors que le grand-boutisme à l'intérêt d'être plus naturel pour les humains de langues européennes.



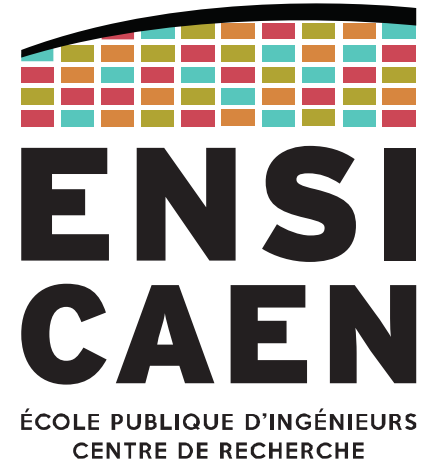
PROCESSUS EN MÉMOIRE VIRTUELLE

Segments mémoire

Segment de code

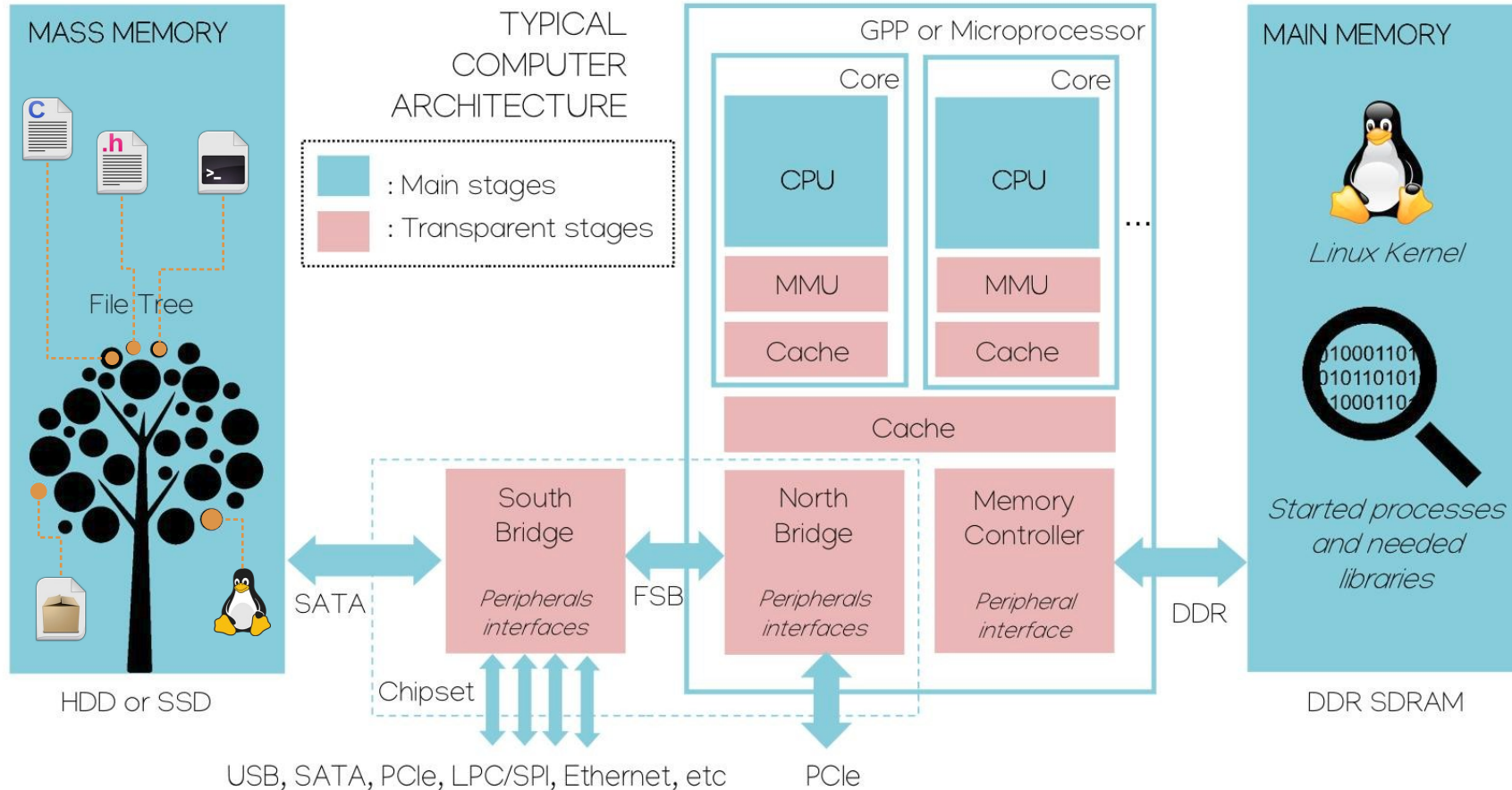
Pile / Stack

Tas / Heap



EXÉCUTION D'UN PROGRAMME

Que se passe-t-il quand on lance un exécutable ?



Sections d'un fichier ELF

Pour rappel, sous Linux, le fichier exécutable est un fichier **binaire** au **format ELF**.
C'est aussi le cas des bibliothèques statiques, dynamiques ou autres objets partagés.

Ce fichier contient différentes sections, dont :

- `.text` : contient le code sous forme binaire ;
- `.data` : les variables globales et statiques initialisées ;
- `.bss` : idem, mais non-initialisées ;
- `.rodata` : les variables en lecture seule (les constantes).

```
dboudier:toolchain$ objdump -s objdump-minimal.o
objdump-minimal.o:      file format elf64-x86-64

Contents of section .text:
0000 f30f1efa 554889e5 4883ec10 488d0500  ...UH..H...H...
0010 00000048 8945f848 8b45f848 89c2488d  ...H.E.H.E.H..H.
0020 35000000 00488d3d 00000000 b8000000  5....H.=.....
0030 00e80000 0000b800 000000c9 c3          .....

Contents of section .data:
0000 426f6e6a 6f757220 6c65206d 6f6e6465  Bonjour le monde
0010 0a00                ..

Contents of section .rodata:
0000 48656c6c 6f20576f 726c640a 00257325  Hello World..%s%
0010 7300                s.
```

Au démarrage de l'application, ces sections passeront de la mémoire de stockage de masse (disque dur) vers la mémoire principale (RAM).

Lors du lancement d'un programme (fichier situé dans la mémoire de masse), le système d'exploitation ne charge en mémoire que les sections essentielles.

La section `.text` devient le **segment de code**.

Et `.data/.bss/.rodata`, des **segments de données**.

Cette phase de chargement, c'est l'**allocation statique** : la taille des segments placés en mémoire est immuable (fixée par la chaîne de compilation).

MAIN MEMORY

CODE segment
Code, static, ReadOnly

DATA segment
Data, static, ReadOnly

DATA segment
Data, static, R/W

Process

Utilisation de la mémoire principale par un processus

Pour chaque **processus** (programme en cours), on trouve deux autres segments, dédiés à l'**allocation dynamique**.

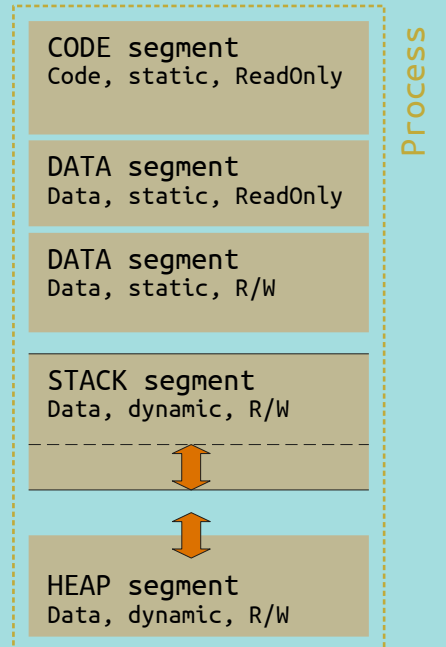
Le **segment de pile (*stack*)** contient les variables locales des fonctions et sauvegardes de contextes.

La *stack* est réservée aux variables locales de taille modeste.

Le **segment de tas (*heap*)** contient les variables allouées dynamiquement : fonctions `malloc()`, `free()`, ...

Très utile pour les tableaux dynamiques par exemple, mais nécessite la présence d'une MMU.

MAIN MEMORY



Utilisation de la mémoire principale par un processus

Les processus peuvent utiliser d'autres zones de la mémoire principale.

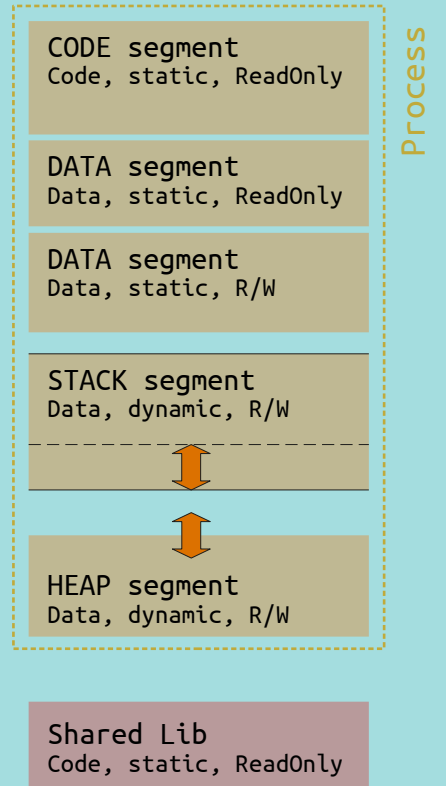
On peut citer par exemple les **bibliothèques partagées** (`glibc` sous Linux) qui sont accessibles par n'importe quel processus (pour un `printf()` par exemple).

Elles sont utilisées grâce à un *linker* dynamique.

On compte aussi des zones de communication entre les processus (IPC, *Inter-Process Communication*).

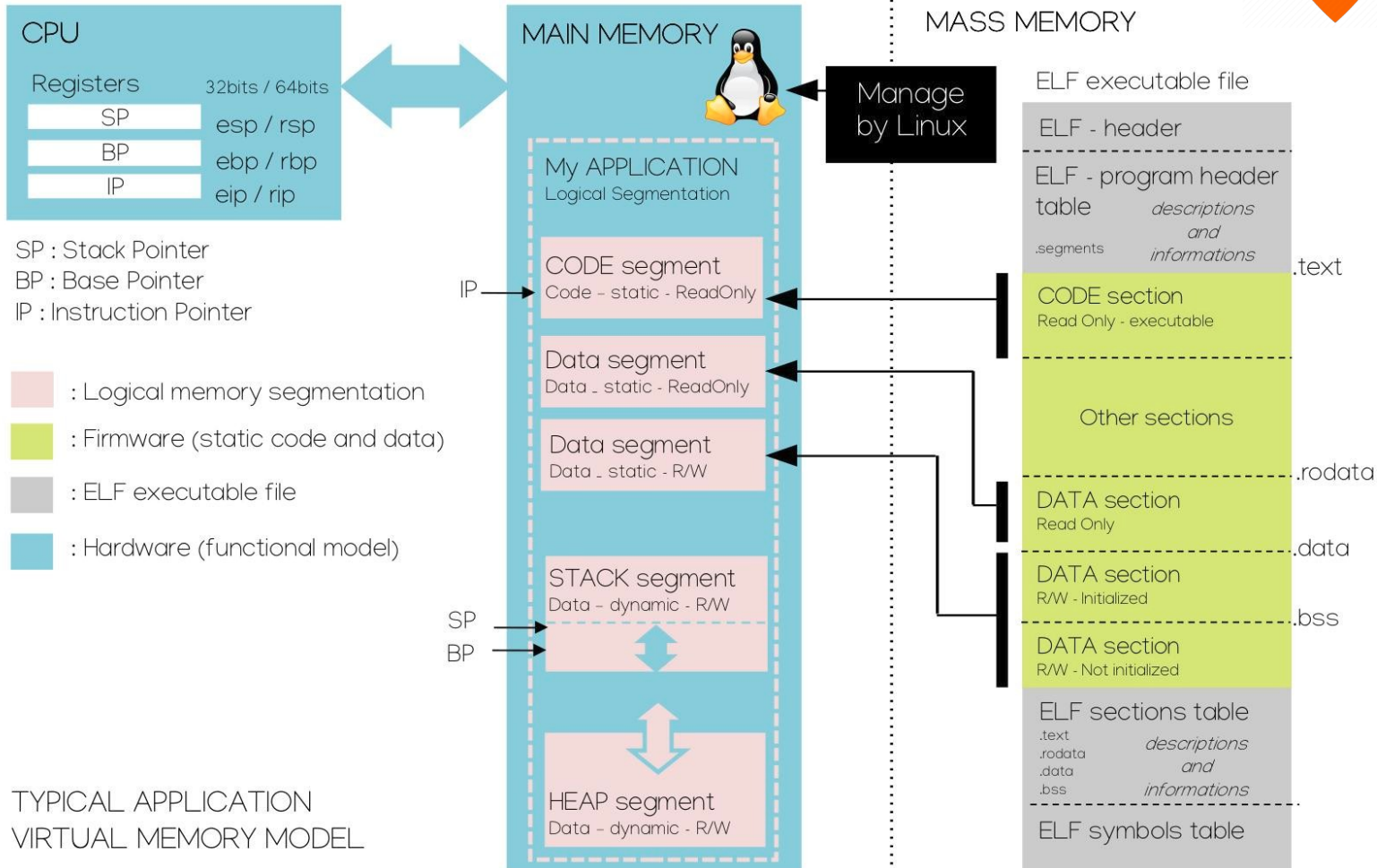
NB : ceci est une représentation logique de ce qu'il se passe en mémoire virtuelle.

MAIN MEMORY



EXÉCUTION D'UN PROGRAMME

Utilisation de la mémoire principale par un processus



Utilisation de la mémoire principale par un processus

```
dboudier:toolchain$ ps -Af
UID          PID     PPID  C  STIME TTY          TIME CMD
root          1         0  0  09:41 ?           00:00:01 /sbin/init splash
root          2         0  0  09:41 ?           00:00:00 [kthreadd]
root          3         2  0  09:41 ?           00:00:00 [rcu gp]
dboudier     24039    1292  0  18:47 ?           00:00:00 /usr/libexec/tracker-store
root         24186     2  0  18:47 ?           00:00:00 [kworker/14:0]
dboudier     24190    12732 99  18:47 pts/0       00:00:06 ./a.out
dboudier     24193    12724  0  18:47 pts/1       00:00:00 bash
dboudier     24211    24193  0  18:47 pts/1       00:00:00 ps -Af
```

ps -Af

Affiche la liste des processus (tronquée ici)

PID = Process Identifier ; CMD = nom de l'exécutable

```
dboudier:toolchain$ sudo cat /proc/24190/maps
```

cat /proc/<PID>/maps

Affiche l'utilisation de la mémoire par le processus <PID>

```
[sudo] password for dboudier:
55faba9de000-55faba9df000 r--p 00000000 08:01 21760669 /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9df000-55faba9e0000 r-xp 00001000 08:01 21760669 /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9e0000-55faba9e1000 r--p 00002000 08:01 21760669 /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9e1000-55faba9e2000 r--p 00002000 08:01 21760669 /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9e2000-55faba9e3000 rw-p 00003000 08:01 21760669 /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba07000-55faba28000 rw-p 00000000 00:00 0 [heap]
7fa9c05f4000-7fa9c0616000 r--p 00000000 103:02 6817925 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c0616000-7fa9c078e000 r-xp 00022000 103:02 6817925 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c078e000-7fa9c07dc000 r--p 0019a000 103:02 6817925 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c07dc000-7fa9c07e0000 r--p 001e7000 103:02 6817925 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c07e0000-7fa9c07e2000 rw-p 001eb000 103:02 6817925 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c07e2000-7fa9c07e8000 rw-p 00000000 00:00 0
7fa9c07fe000-7fa9c07ff000 r--p 00000000 103:02 6817918 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c07ff000-7fa9c0822000 r-xp 00001000 103:02 6817918 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c0822000-7fa9c082a000 r--p 00024000 103:02 6817918 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c082b000-7fa9c082c000 r--p 0002c000 103:02 6817918 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c082c000-7fa9c082d000 rw-p 0002d000 103:02 6817918 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c082d000-7fa9c082e000 rw-p 00000000 00:00 0
7ffe5f5e8000-7ffe5f609000 rw-p 00000000 00:00 0 [stack]
7ffe5f7bc000-7ffe5f7c0000 r--p 00000000 00:00 0 [vvar]
7ffe5f7c0000-7ffe5f7c2000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

Allocation statique

(issus des sections du fichier ELF)

**Librairie partagée (libc)
Linker dynamique (ld)**

**Allocation dynamique
(tas/heap et pile/stack)**

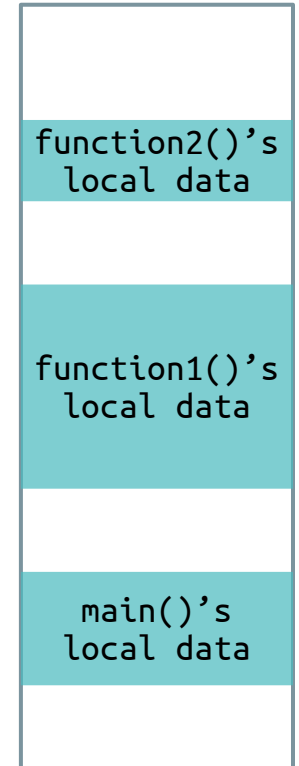
La **stack (ou pile)** d'un processus (ou d'un thread) contient les variables locales des fonctions qui le composent.

Chaque fonction appelée viendra « poser » ses données en sommet de pile, au dessus des autres fonctions.

Cette pile se « videra » progressivement, à chaque retour de fonction.

Ce comportement est celui d'une pile **LIFO (Last In, First Out)**.

Process's stack



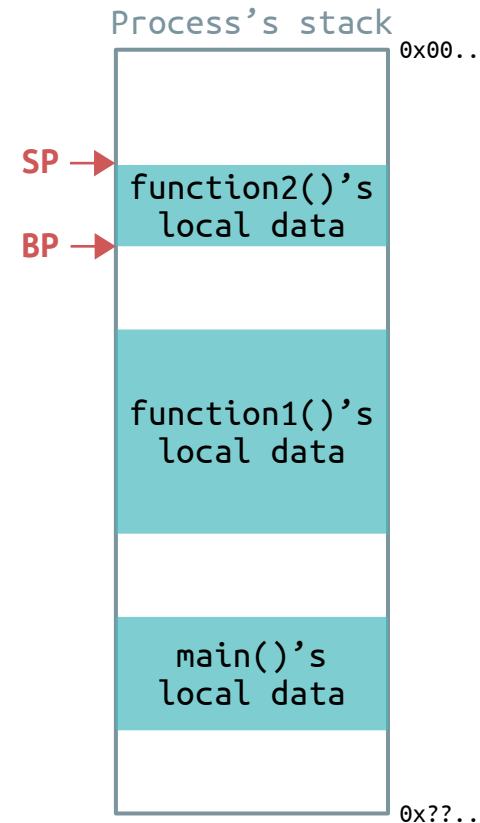
Afin de manipuler la pile, chaque CPU possède deux registres nommés **Base Pointer BP** et **Stack Pointer SP**.

Le **Stack Pointer SP** évolue très fréquemment, suivant le **sommet de pile** selon ce qui y est posé ou retiré.

Le **Base Pointer BP** évolue à chaque appel ou retour de fonction, son rôle étant d'indiquer la zone de données de la **fonction courante**.

Mnémotechnie : SP = Stack Pointer ≈ Sommet de Pile

BP = Base Pointer ≈ Bas de Pile

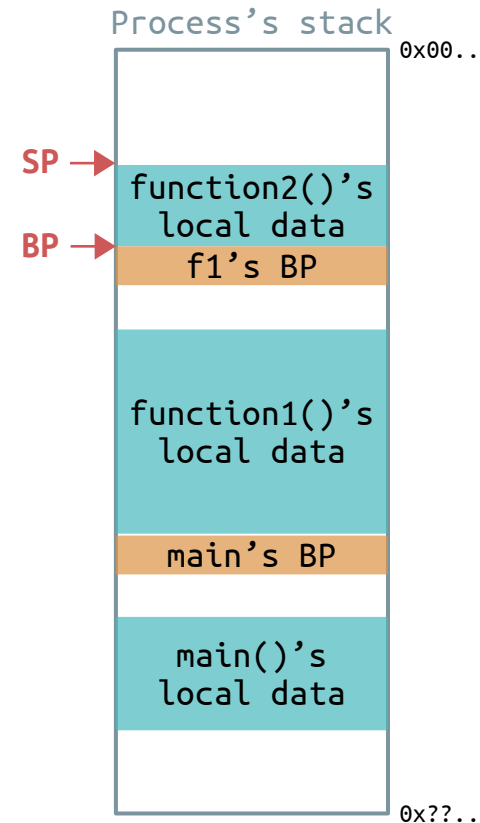


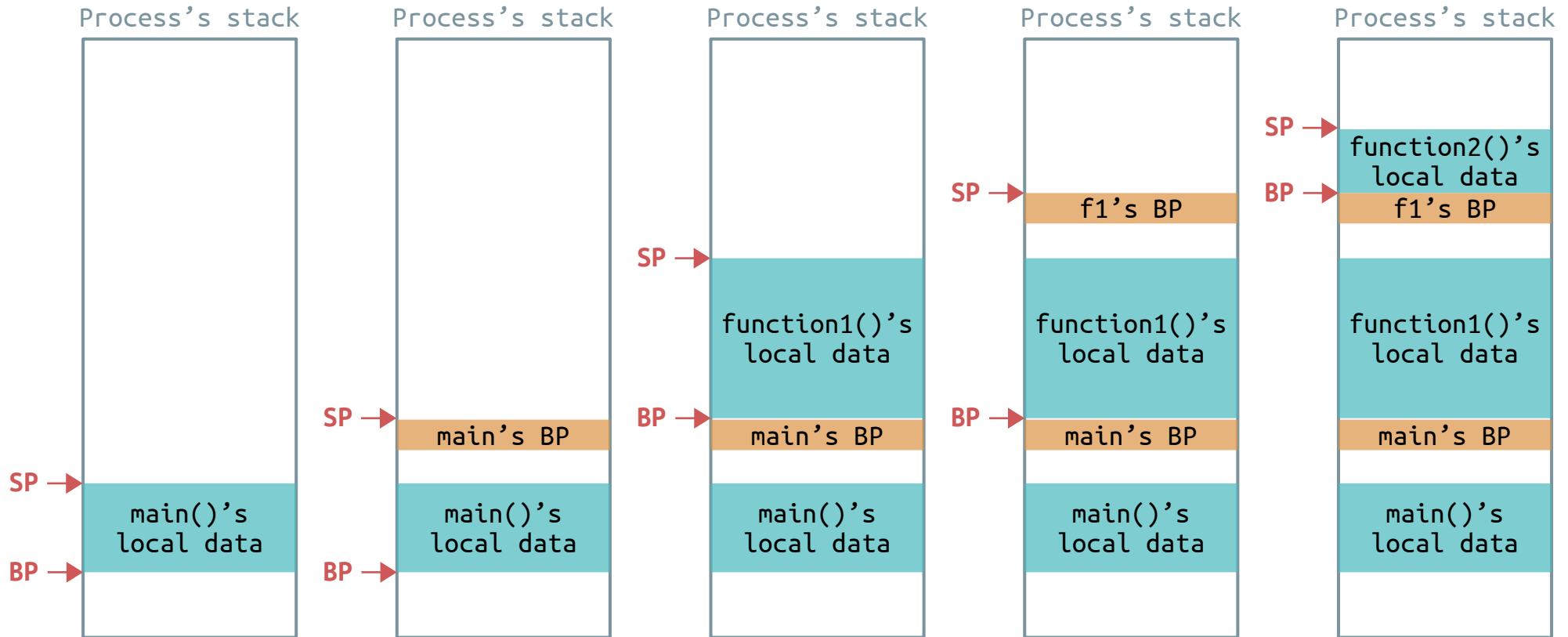
La valeur du **Stack Pointer SP** étant toujours associé au sommet de pile (indépendamment des fonctions), il n'est pas nécessaire de sauvegarder sa valeur.

Le **Base Pointer BP** en revanche est associé à la fonction en cours d'exécution. Quand une nouvelle fonction est appelée, sa valeur sera amenée à changer.

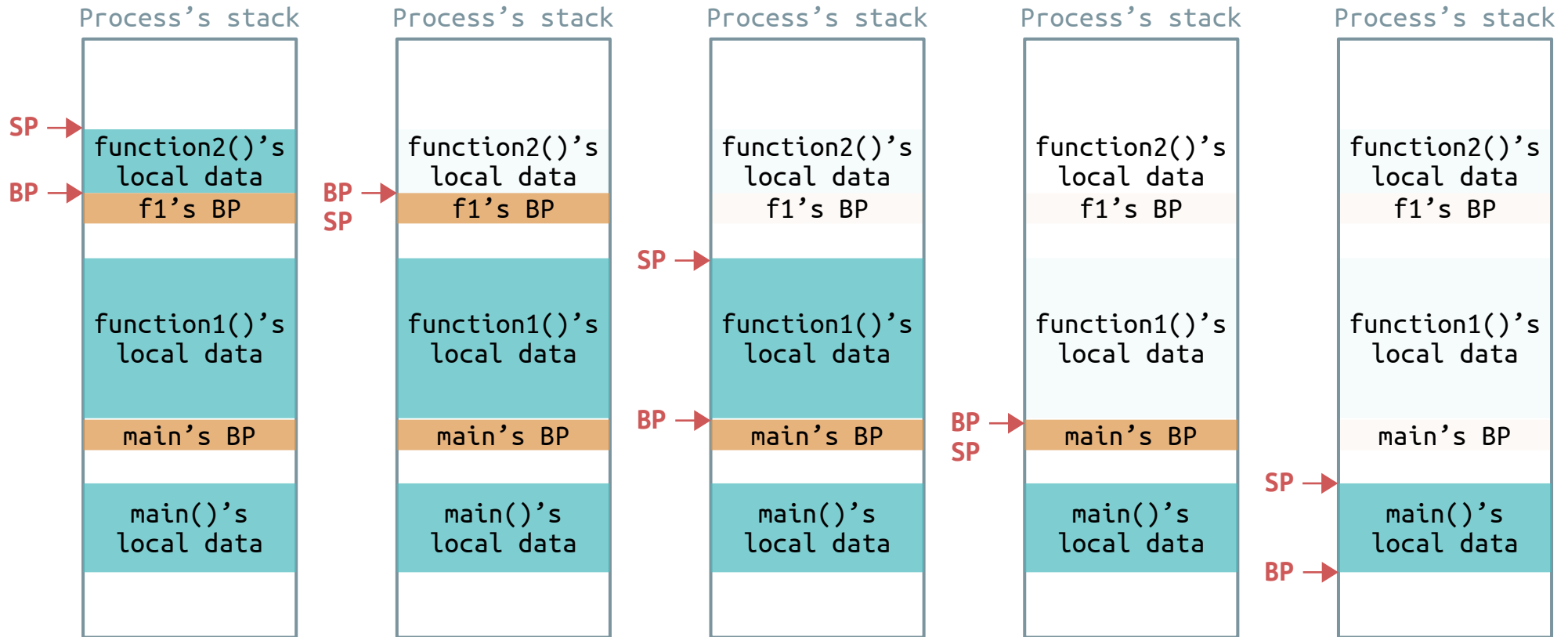
Sa valeur sera donc sauvegardée avant d'être modifiée pour correspondre à la nouvelle fonction.

Ainsi, il sera possible de retrouver la valeur initiale de BP lorsque la fonction appelée effectuera son **return**.





Stack/pile : allocation automatique



Le **Instruction Pointer IP*** contient l'adresse de la prochaine instruction à exécuter.

* IP est parfois appelé **Program Counter PC**, comme chez le PIC18 par exemple.

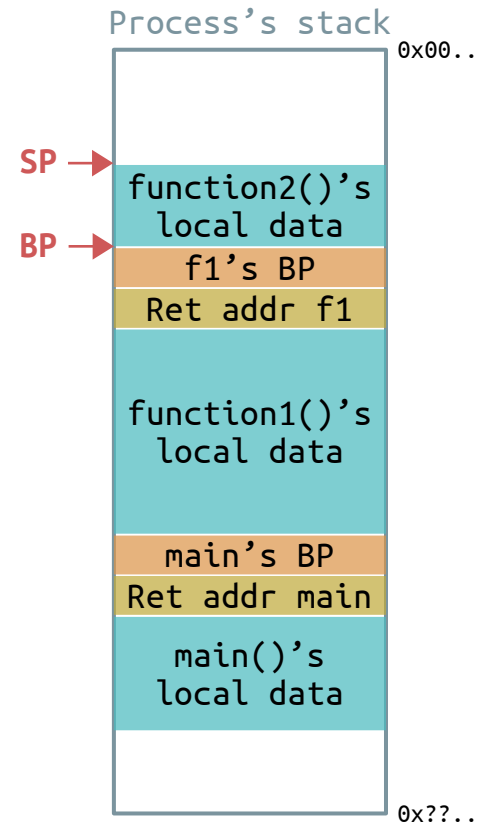
Dans le cas d'un code purement linéaire, il s'incrémente automatiquement pour pointer vers l'instruction suivante.

Lors d'un appel de fonction, sa valeur est écrasée. Il faut alors trouver un moyen de sauvegarder sa valeur : en la posant sur la pile.

```
0000000000001129 <main>:
1129:  pushq  %rbp
112a:  movq   %rsp, %rbp
112d:  callq  1139 <function_1>
1132:  movl   $0x0, %eax
1137:  popq   %rbp
1138:  retq

0000000000001139 <function_1>:
1139:  pushq  %rbp
113a:  movq   %rsp, %rbp
113d:  subq   $0x10, %rsp
1141:  movl   $0x3, %edx
1146:  movl   $0x2, %esi
114b:  movl   $0x1, %edi
1150:  callq  115b <function_2>
1155:  movl   %eax, -0x4(%rbp)
1158:  nop
1159:  leaveq
115a:  retq

000000000000115b <function_2>:
115b:  pushq  %rbp
115c:  movq   %rsp, %rbp
115f:  movl   %edi, -0x4(%rbp)
1162:  movl   %esi, -0x8(%rbp)
1165:  movl   %edx, -0xc(%rbp)
...
1175:  popq   %rbp
1176:  retq
```



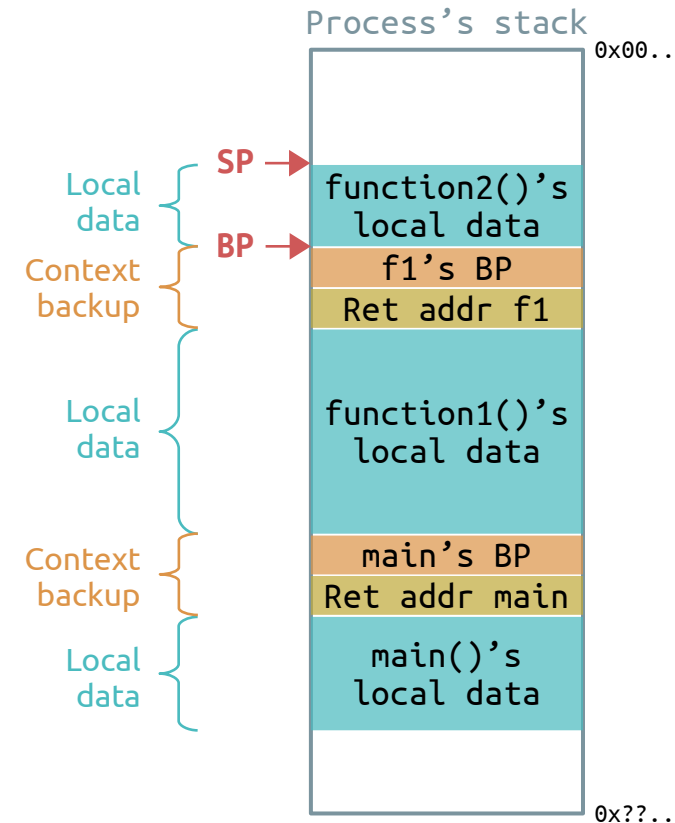
Dans la pile du process, on retrouve donc une alternance de **zones contenant les données locales** des fonctions et de **zones de sauvegarde de contexte** de ces fonctions.

Un **contexte d'exécution** est la valeur des registres du CPU à un instant t . Ces valeurs ne sont valides que dans le contexte (= cadre) de la fonction en cours.

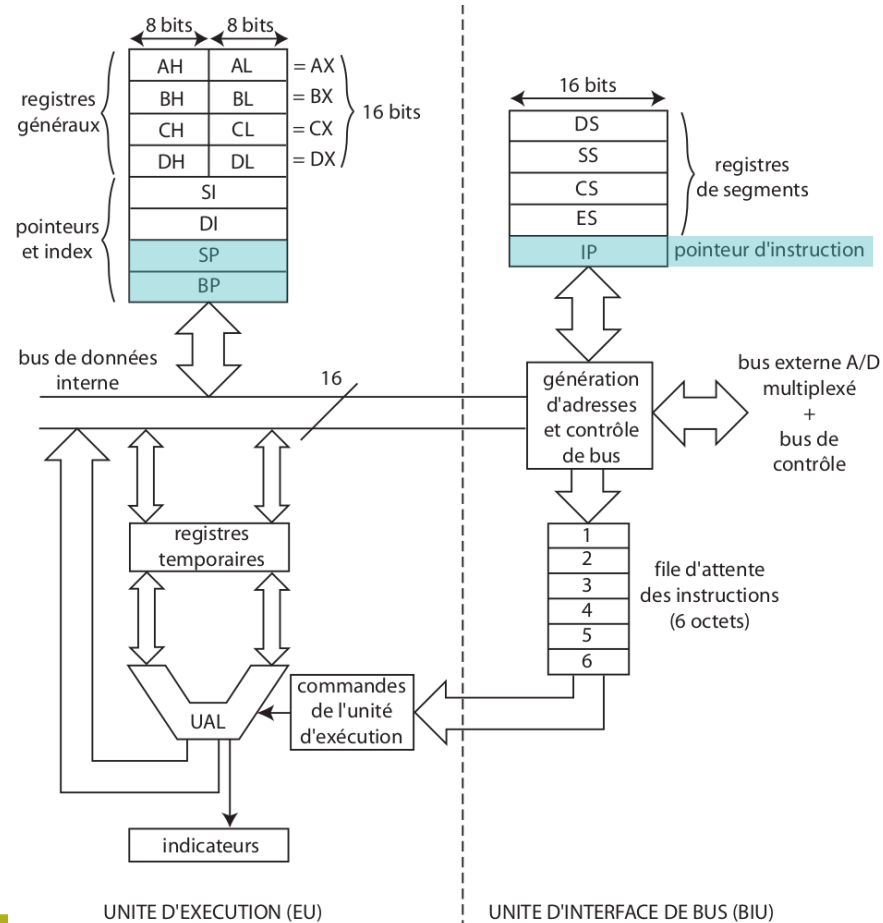
Exemple : les registres IP et BP, mais ça peut aussi être le cas pour les registres de travail.

Si la fonction courante appelle une autre fonction, alors les registres IP et BP qu'elle utilise seront écrasés par la fonction appelée.

On sauvegarde donc la valeur de ces registres afin de restituer le contexte de la fonction appelante au retour de la fonction appelée.



Exemple du Intel 8086



Stack/pile : allocation automatique

```
int main(void)
```

```
{
    function_1();
    return 0;
}
```

```
void function_1 (void)
```

```
{
    int ret_1;
    ret_1 = function_2 (1, 2, 3);
}
```

```
int function_2 (int a_2, int b_2, int c_2)
```

```
{
    return a_2 + b_2 + c_2;
}
```

① ③ ⑤

Sauvegarde du BP de
la fonction appelante

② ④ ⑥

Restauration du BP de
la fonction appelante

⑦ ⑨

Sauvegarde du IP de
la fonction appelante

⑧ ⑩

Restauration du IP de
la fonction appelante

Passage d'arguments
par les registres CPU

```
00000000000001129 <main>:
```

```
① 1129: pushq %rbp
112a: movq %rsp, %rbp
⑦ 112d: callq 1139 <function_1>
1132: movl $0x0, %eax
② 1137: popq %rbp
1138: retq
```

```
00000000000001139 <function_1>:
```

```
③ 1139: pushq %rbp
113a: movq %rsp, %rbp
113d: subq $0x10, %rsp
1141: movl $0x3, %edx
1146: movl $0x2, %esi
114b: movl $0x1, %edi
⑨ 1150: callq 115b <function_2>
1155: movl %eax, -0x4(%rbp)
1158: nop
④ 1159: leaveq
⑧ 115a: retq
```

Allocation pour
variable locale

ret_1 : adresse
relative à BP

```
0000000000000115b <function_2>:
```

```
⑤ 115b: pushq %rbp
115c: movq %rsp, %rbp
115f: movl %edi, -0x4(%rbp)
1162: movl %esi, -0x8(%rbp)
1165: movl %edx, -0xc(%rbp)
...
⑥ 1175: popq %rbp
⑩ 1176: retq
```

Variables locales :
adr. relatives à BP

Stack/pile : allocation automatique

La taille de la pile est gérée par l'OS. Par défaut elle est de **8 MB** sous Linux, mais il est possible de la modifier (minimum de 128 kB) : soit depuis le programme en utilisant les fonctions `getrlimit()` et `setrlimit()` ; soit depuis l'OS avec la commande `ulimit`.

```
dboudier:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size          (kbytes, -d) unlimited
scheduling priority    (-e) 0
file size              (blocks, -f) unlimited
pending signals        (-i) 62349
max locked memory      (kbytes, -l) 65536
max memory size        (kbytes, -m) unlimited
open files             (-n) 1024
pipe size              (512 bytes, -p) 8
POSIX message queues   (bytes, -q) 819200
real-time priority     (-r) 0
stack size             (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes     (-u) 62349
virtual memory         (kbytes, -v) unlimited
file locks             (-x) unlimited
```

Cette taille relativement petite rend l'usage de la stack inintéressant pour les grandes quantités de données.

Dans ce cas, on lui préférera le tas/heap.

```
dboudier:~$ ulimit -s
8192
dboudier:~$ ulimit -s 16384
dboudier:~$ ulimit -s
16384
```



La stack/pile en bref :

- Allocation dynamique (faite à l'exécution ou run-time)
- Allocation automatique (instructions ajoutées par la *toolchain*, non par le dév.)
- Pile LIFO (Last In, First Out)
- Pour les données locales aux fonctions
- Pour les sauvegardes de contexte (BP, IP)
- Passage d'arguments de fonction par les registres CPU
- De taille modeste (par défaut 8 Mo)

Passons maintenant à l'étude du segment de **tas ou heap**. Chaque processus possède son propre tas (partagé entre les threads d'un même processus).

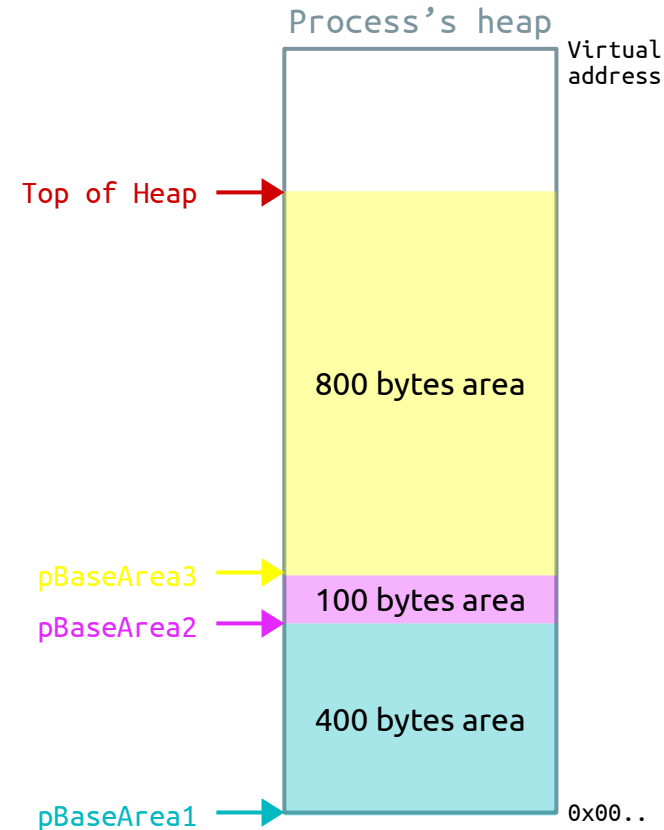
Les ressources y sont allouées dynamiquement, par demande explicite du développeur avec les fonctions dédiées du langage :

- `calloc()`, `malloc()`, `free()` pour le langage C
- `new`, `delete` pour le langage C++
- ...



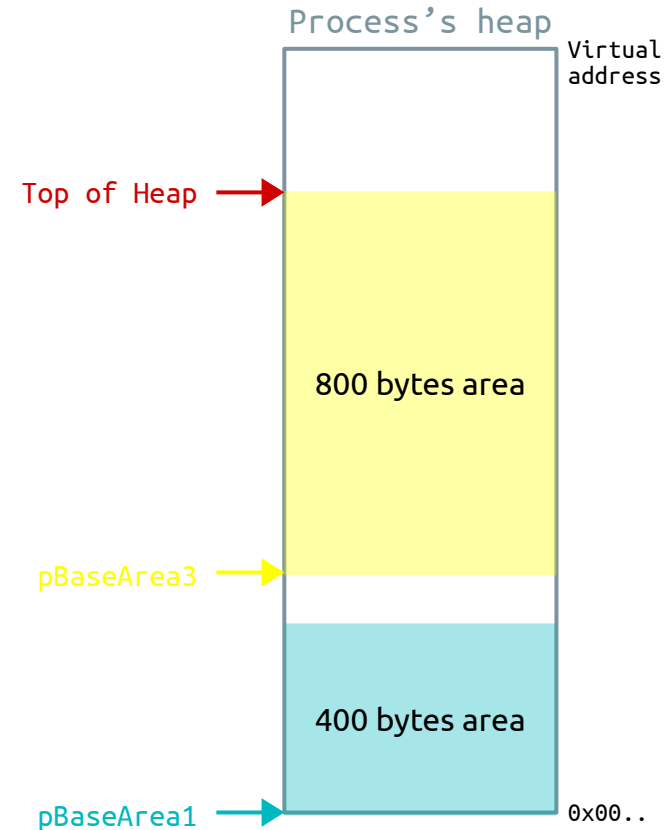
Chaque demande d'allocation renvoie un pointeur vers la base de la zone effectivement allouée.

```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc( 100 * sizeof(float) );  
    pBaseArea2 = (char*) malloc( 100 * sizeof(char) );  
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );  
  
    // user application ...  
  
    free( pBaseArea2 );  
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );  
  
    // user application ...  
  
    free( pBaseArea1 );  
    free( pBaseArea2 );  
    free( pBaseArea3 );  
  
    return 0;  
}
```



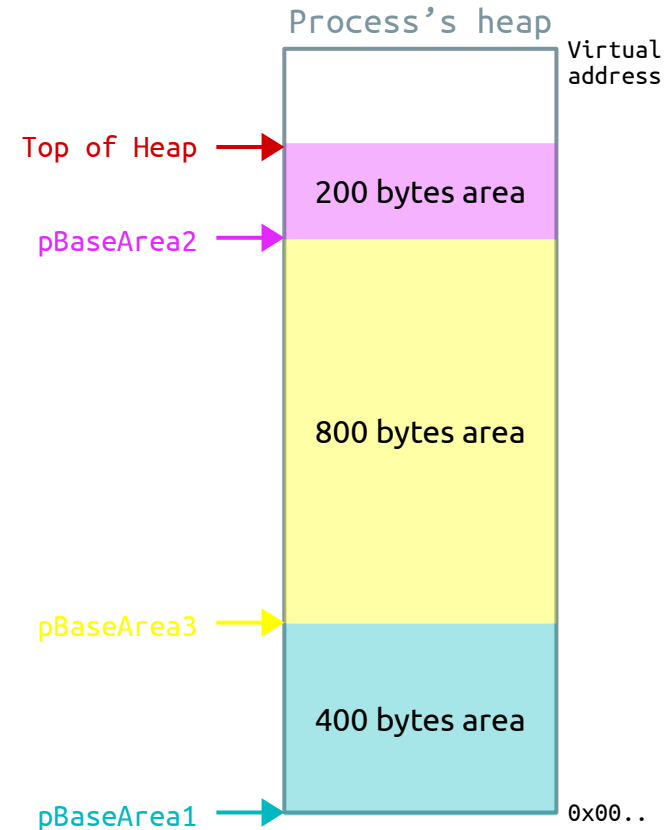
Une demande de désallocation dérèfèrence le pointeur.

```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc( 100 * sizeof(float) );  
    pBaseArea2 = (char*) malloc( 100 * sizeof(char) );  
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );  
  
    // user application ...  
  
    free( pBaseArea2 );  
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );  
  
    // user application ...  
  
    free( pBaseArea1 );  
    free( pBaseArea2 );  
    free( pBaseArea3 );  
  
    return 0;  
}
```



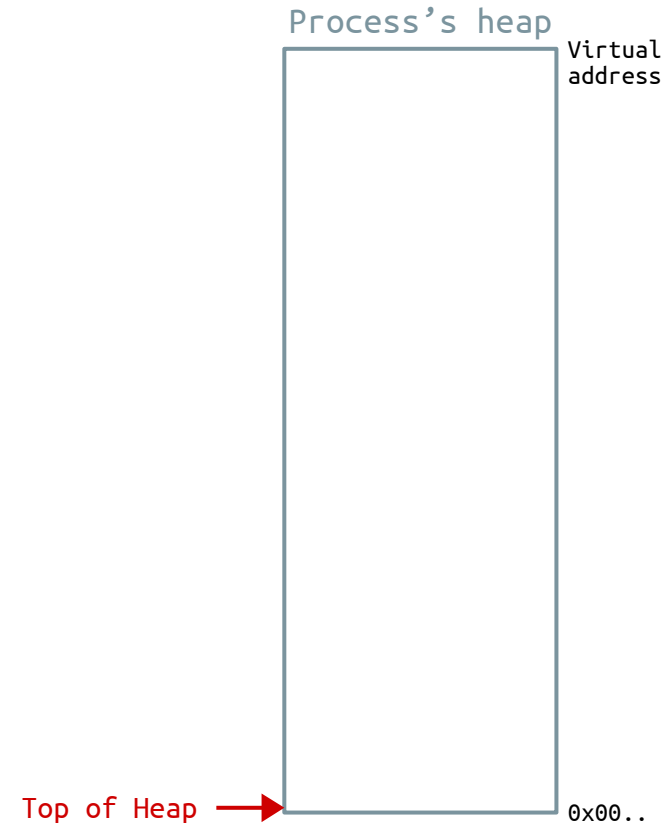
L'espace de mémoire virtuelle sera toujours contigu,
même si l'unité de pagination de la MMU gère en réalité la fragmentation
de la mémoire physique.

```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc( 100 * sizeof(float) );  
    pBaseArea2 = (char*) malloc( 100 * sizeof(char) );  
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );  
  
    // user application ...  
  
    free( pBaseArea2 );  
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );  
  
    // user application ...  
  
    free( pBaseArea1 );  
    free( pBaseArea2 );  
    free( pBaseArea3 );  
  
    return 0;  
}
```



Il faut toujours libérer les ressources mémoires après utilisation, sinon les données encombrant la mémoire.

```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc( 100 * sizeof(float) );  
    pBaseArea2 = (char*) malloc( 100 * sizeof(char) );  
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );  
  
    // user application ...  
  
    free( pBaseArea2 );  
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );  
  
    // user application ...  
  
    free( pBaseArea1 );  
    free( pBaseArea2 );  
    free( pBaseArea3 );  
  
    return 0;  
}
```



Heap/tas : allocation dynamique

Comme pour la pile, des débordements de tas (*heap overflow*) sont possibles.

Mais contrairement à la pile, **le tas dispose de quasiment tout l'espace de la mémoire principale**, ce qui permet d'y allouer de graaaandes quantités de données.

```
dboudier:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size          (kbytes, -d) unlimited
scheduling priority    (-e) 0
file size              (blocks, -f) unlimited
pending signals        (-i) 62349
max locked memory      (kbytes, -l) 65536
max memory size        (kbytes, -m) unlimited
open files             (-n) 1024
pipe size              (512 bytes, -p) 8
POSIX message queues   (bytes, -q) 819200
real-time priority     (-r) 0
stack size             (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes     (-u) 62349
virtual memory         (kbytes, -v) unlimited
file locks             (-x) unlimited
```

Mémoire virtuelle : illimitée (en théorie).

En pratique, limitée par la taille réelle de la RAM (moins l'espace occupé par l'OS, ~ 1 Go).

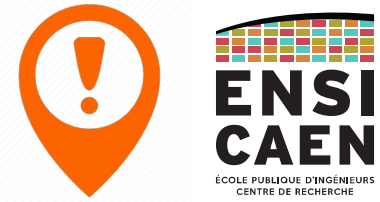
Vu par le CPU, l'espace des données allouées sur le tas est toujours contigu.

Ceci est dû au travail de la **MMU (*Memory Management Unit*)** qui gère la fragmentation de la mémoire physique d'un côté (avec son unité de pagination) et la présente majestueusement au CPU.

Les processeurs sans MMU (typiquement, les micro-contrôleurs) et leurs *toolchains* associées gèrent difficilement les mécanismes de fragmentation de leur mémoire physique.

Dans ces cas, l'utilisation des fonctions `malloc()` et `free()` doit être prohibé, ou a minima surveillé avec de grandes précautions.

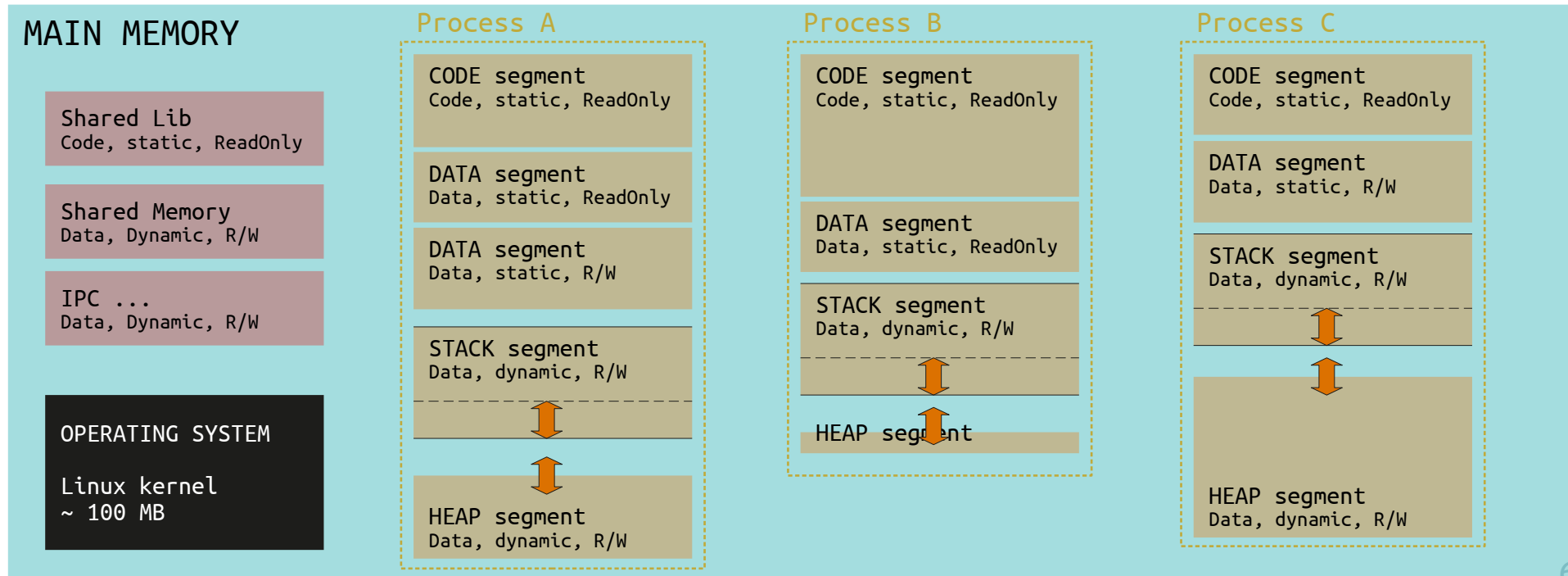
Pas de MMU → pas d'allocation dynamique



Le heap/tas en bref :

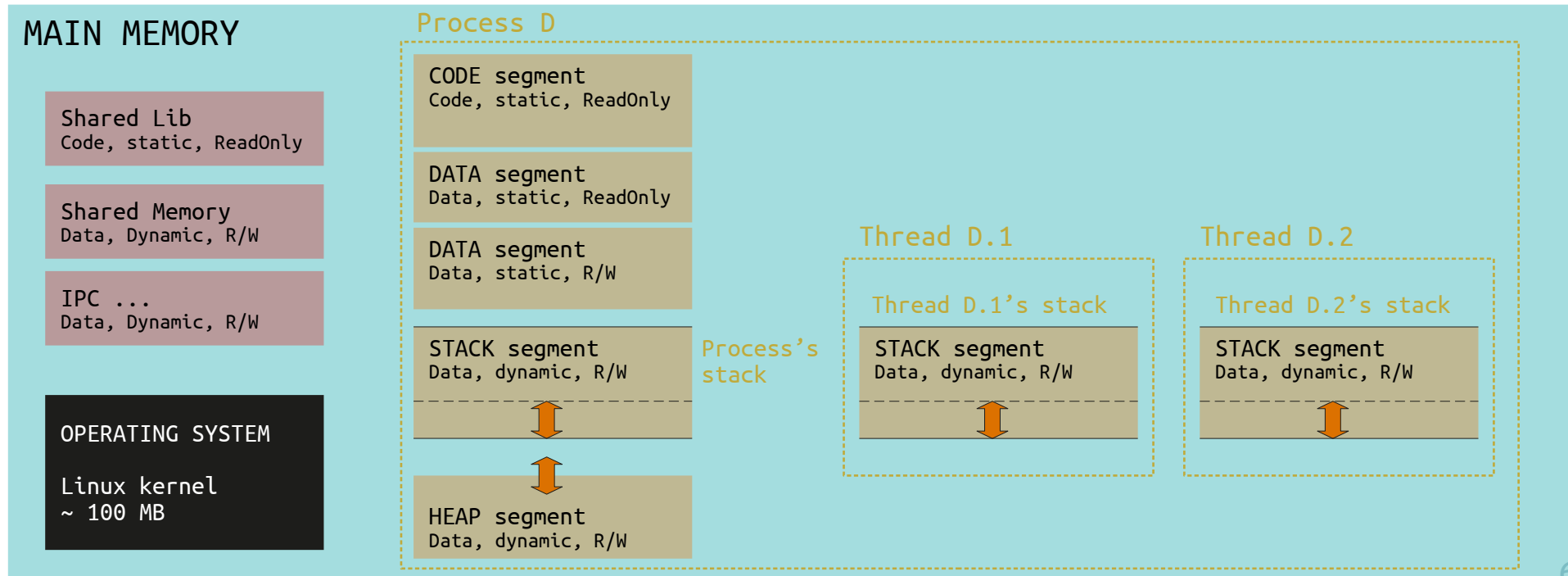
- Allocation dynamique (faite à l'exécution ou run-time)
- Allocation explicite (fonctions appelées par le développeur)
 - `malloc()`, `free()`
- Pour les grandes quantités de données (tableaux dynamiques, ...)
- Toujours libérer les ressources allouées après utilisation
- Nécessité d'avoir une MMU (Memory Management Unit)
- De taille infinie en théorie, limitée à la taille de la RAM en pratique

Sur un ordinateur équipé d'un système d'exploitation, plusieurs processus sont actifs en même temps. Ils possèdent chacun leur espace mémoire dédié, auquel s'ajoutent les zones partagées (bibliothèques dynamiques, IPC, ...)



Plusieurs threads en exécution

Un processus typique possède plusieurs **threads** (processus légers). Chaque thread dispose de sa propre *stack*, mais tous les autres segments du processus sont partagés.



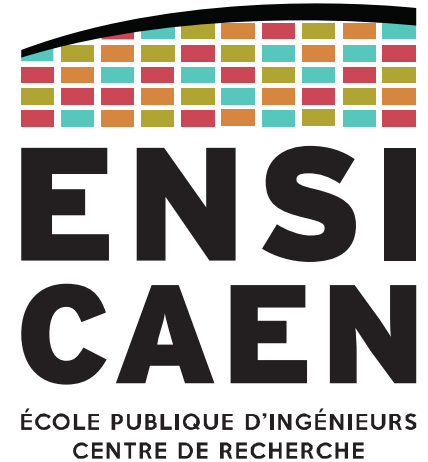
MEMORY MANAGEMENT UNIT

Mémoire virtuelle vs Mémoire physique

Pagination

Segmentation

Protection de la mémoire

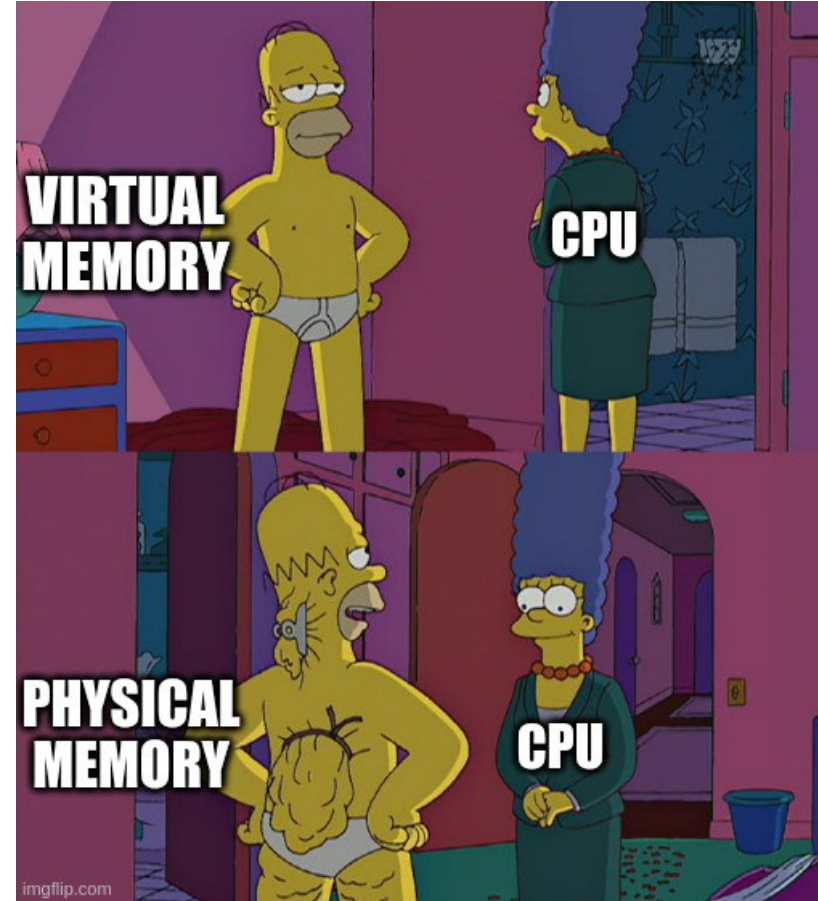


Mémoire physique vs mémoire virtuelle

Tout ce qui a été vu dans la section précédente (segments de *code*, de *data*, *heap*, *stack*, ...) se trouve en **mémoire virtuelle**.

Le CPU voit un espace mémoire infiniment grand, dans lequel tous les segments sont en un seul morceau.

En réalité l'utilisation de la **mémoire physique** (la RAM) est bien plus chaotique. Chaque segment est découpé en plusieurs morceaux, placés à des endroits bien divers. Et la taille de la mémoire physique est en pratique limitée par le composant RAM.



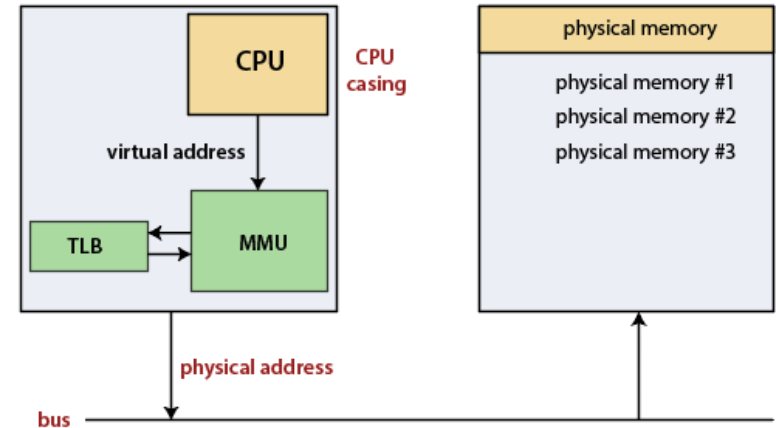
Mémoire physique vs mémoire virtuelle

L'intermédiaire entre la **mémoire virtuelle** (vue par le CPU) et la **mémoire physique** (RAM) est la **MMU** (*Memory Management Unit* ou Unité de Gestion de la Mémoire).

La MMU est un composant matériel propre à chaque cœur d'un GPP ou AP. Tout OS un tant soit peu évolué est capable de l'exploiter.

Elle intègre plusieurs services :

- *Pagination Unit*
- *Segmentation Unit*
- *Memory Protection Unit (MPU)*
- Arbitrage des bus
- ...



CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

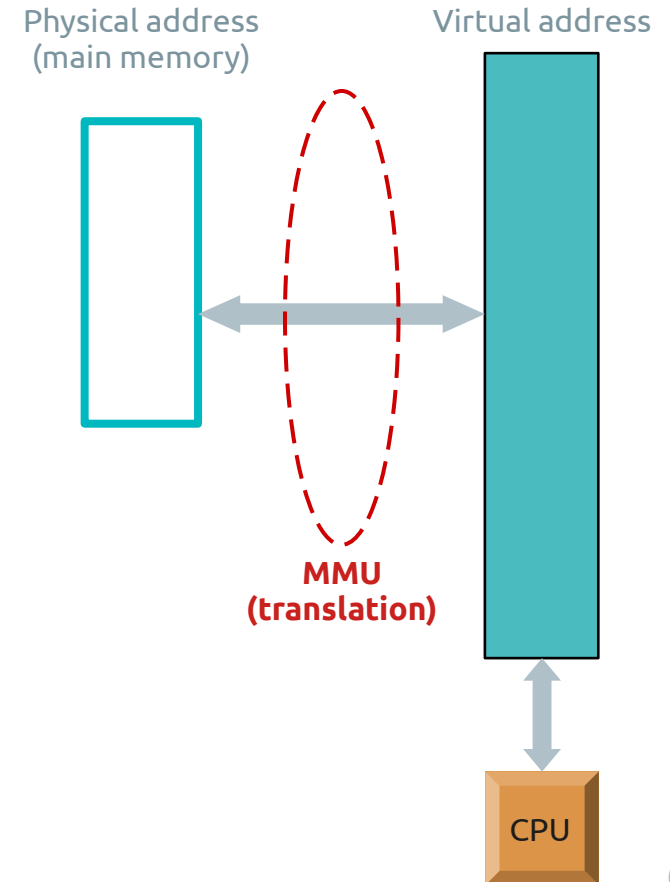
Pagination : adresse physique vs adresse virtuelle

Historiquement, les programmes volumineux qui ne rentraient pas dans la mémoire principale étaient découpés en plusieurs branches, qui étaient stockées dans une mémoire secondaire.

Pendant l'exécution, les branches se chargeaient tour à tour dans la mémoire principale. Le découpage était fait « à la main » par les développeurs qui tenaient un registre dédié.

Aujourd'hui le mécanisme a quelque peu évolué et est automatiquement géré par la MMU. D'un côté cette dernière utilise l'espace d'adressage physique, c-à-d les adresses disponibles de la mémoire physique (la RAM).

De l'autre côté la MMU offre au CPU un espace d'adressage virtuel, soit un ensemble des adresses auxquelles le programme peut faire référence. L'espace d'adressage virtuel est plus grand que l'espace d'adressage physique.



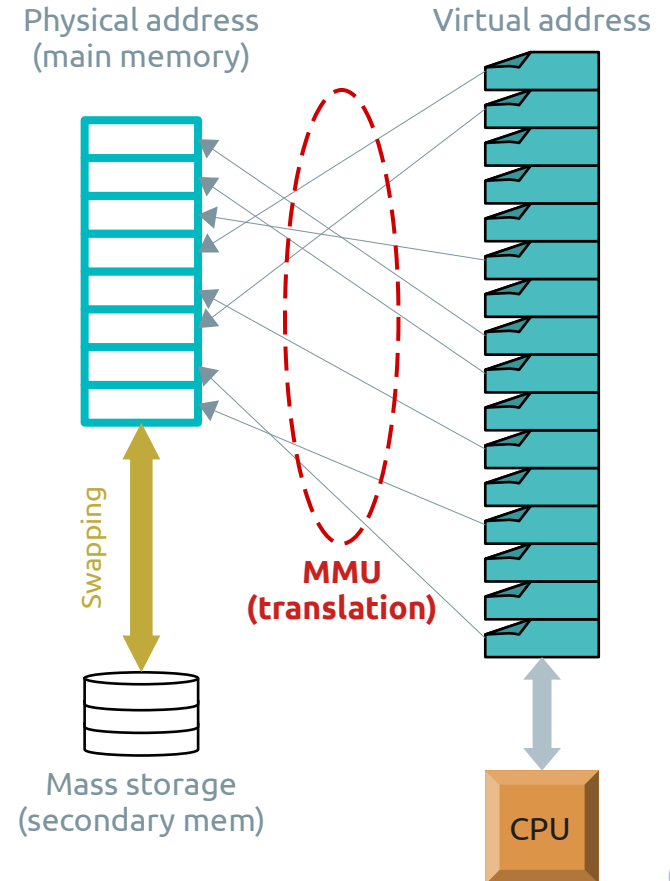
Pagination : adresse physique vs adresse virtuelle

La MMU découpe cet espace d'adressage virtuel en **pages** de même taille ; elle découpe également la mémoire physique en **frames** (cadres) de taille identique.

Chaque *frame* de la mémoire physique peut accueillir une page de la mémoire virtuelle.

Mathématiquement, il y a plus de pages en mémoire virtuelle que de *frames* en mémoire physique. Quand elles ne sont pas utilisées, ces pages sont stockées sur l'**espace d'échange** (*swap*) du disque dur.

Cette méthode permet de faire croire au CPU qu'il accède à une mémoire virtuelle bien plus grande que la mémoire physique.



Pagination : translation

La **translation** (conversion d'une adresse virtuelle en adresse physique) se fait très simplement en consultant une simple table de pages (*page table*).

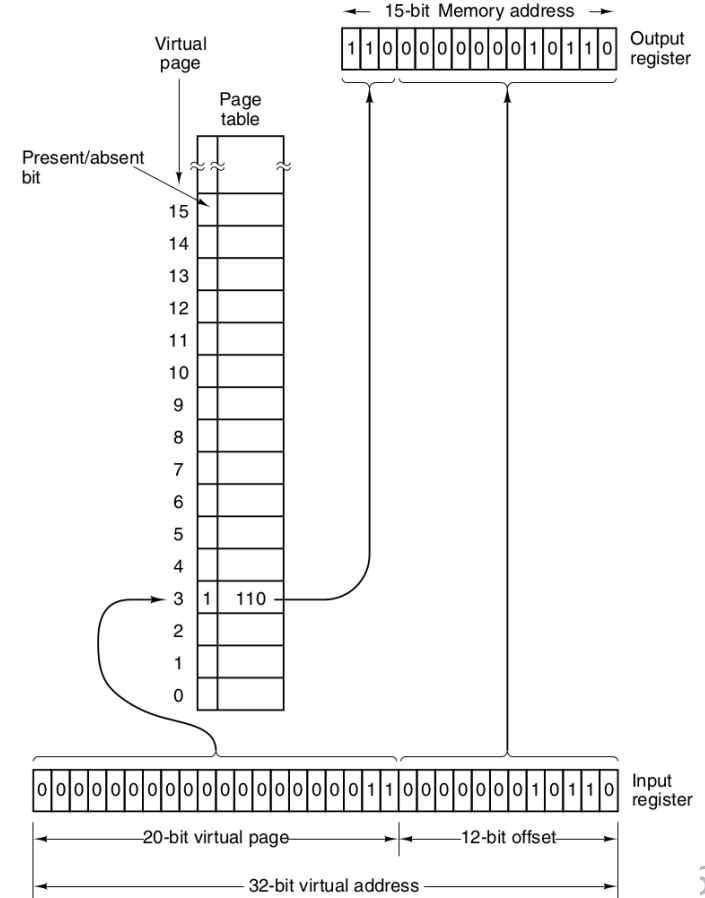
Soit une mémoire physique de 32 ko (15 bits) découpée en 8 *frames* (3 bits) de 4 ko (12 bits).

Soit un espace d'adressage virtuel de 4 Go (32 bits), découpé en 2 millions de pages (20 bits) de 4 ko (12 bits).

Les 12 bits de poids faible de l'adresse virtuelle sont directement mappés sur les 12 bits de poids faible de l'adresse physique. Ils constituent l'offset, c-à-d l'adresse de la donnée dans une page.

Les 20 bits de poids fort de l'adresse virtuelle servent à sélectionner une entrée du TLB, qui contient le numéro du *frame* à manipuler.

Ce mécanisme est transparent : le développeur n'en connaît rien.



Pagination : remplacement de page

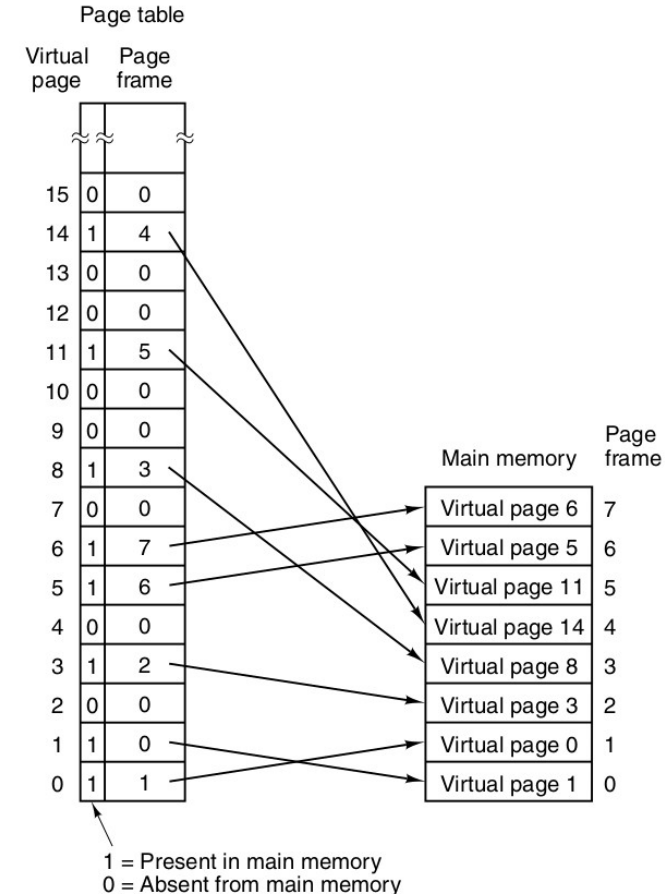
Si la page est déjà présente en mémoire principale (le *present/absent bit* de la table est à '1'), alors la donnée est immédiatement accédée grâce à la translation d'adresse.

Mais si l'adresse virtuelle indique une page qui n'est pas en mémoire physique (le *present/absent bit* de la table est à '0'), alors il faut écraser une page existante pour y placer celle demandée.

Afin de ne pas perdre en efficacité, il faut déterminer quel *frame* devra contenir la page demandée. Avec une stratégie FIFO, le cadre contenant la page la plus ancienne est choisi. Avec une stratégie LRU, le cadre qui a été utilisé il y a le plus longtemps est écrasé. Cette dernière est plus efficace mais plus chère en ressources matérielles.

De plus, à chaque cadre est associé un bit pour savoir si la page contenue a été modifiée. Si c'est le cas, le contenu du *frame* est recopié sur la page virtuelle (dans le *swap*).

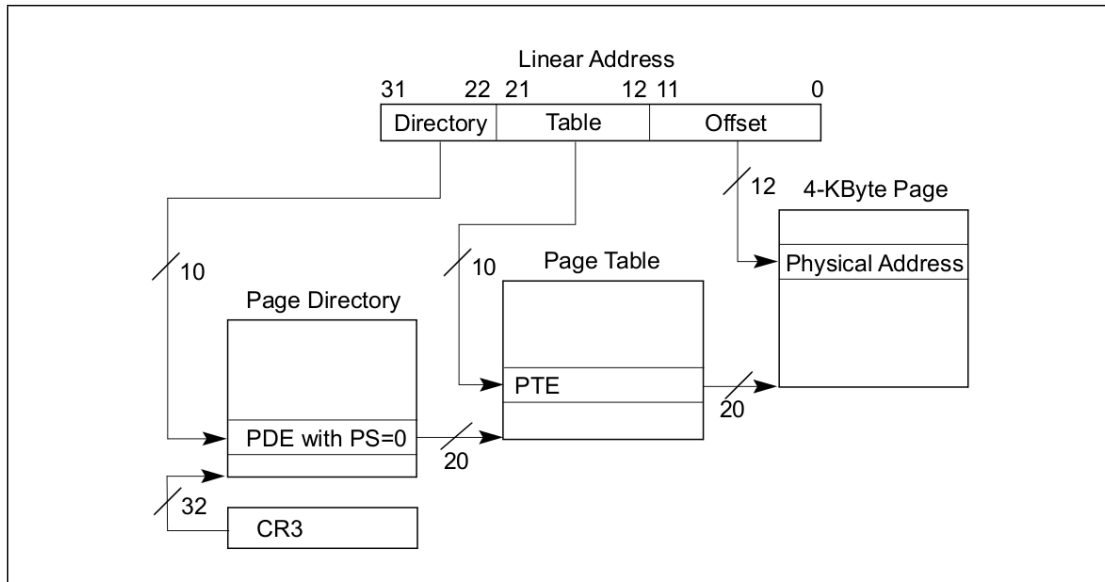
Depuis le *swap*, on peut alors récupérer la page demandée pour l'écrire dans le *frame*.



Pagination : Multilevel page table

Dans l'exemple des deux diapos précédentes, la table de pages occupe 2^{20} octets. Or la table de pages doit se trouver en mémoire physique, qui ne fait que 32 ko !

En réalité, seul un répertoire est stockée en mémoire physique. Il pointe vers de petites tables de pages stockées sur le swap, pour enfin référencer les pages.



Page directory : répertoire stockée en mémoire physique.

Page Table : la table sélectionnée par le répertoire, parmi toutes les tables de pages stockées en swap (ou mém phys).

4-kB page : page stockée dans un frame.

Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Pagination : Fragmentation

Un programme de 26000 B occupera par exemple 7 pages de 4096 B. Mais la dernière page ne contiendra que 1424 B utiles, les autres étant perdus.



Ce phénomène est appelé **fragmentation interne**.

En utilisant des pages de k octets, la fragmentation entraîne en moyenne une perte de $k/2$ octets sur la dernière page.

Pour réduire les pertes liées à la fragmentation interne, il suffit alors d'utiliser des pages de taille réduite.

Cependant cela fait plus de pages à stocker pour un espace virtuel de même taille, et donc la table de page est plus grande (plus complexe, plus de matériel).

Pagination : Multilevel page table

Depuis, les architectures IA-32 supportent la pagination à 4 niveaux, pour des pages de différentes tailles (4 ko, 2 Mo, 4 Mo, 1 Go, en fonction du mode de pagination).

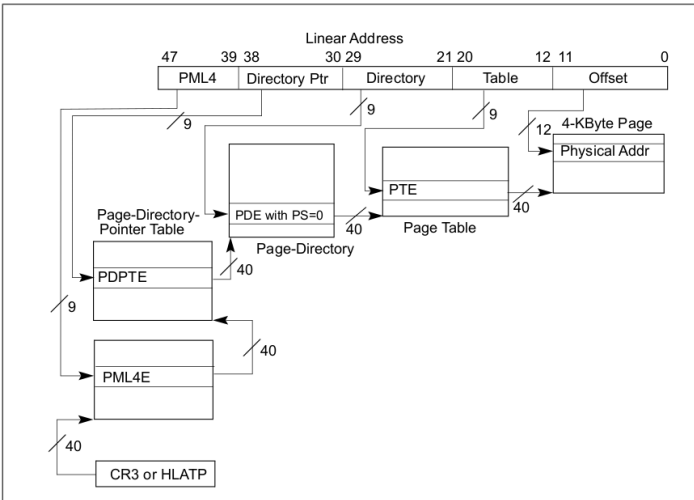


Figure 4-8. Linear-Address Translation to a 4-KByte Page Using 4-Level Paging

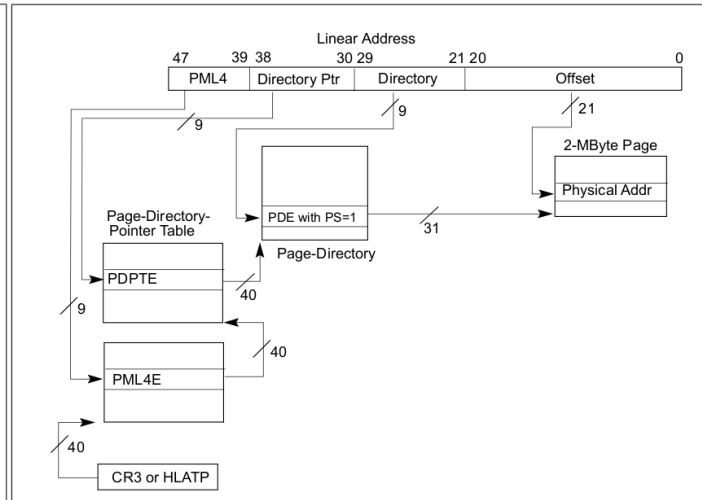


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

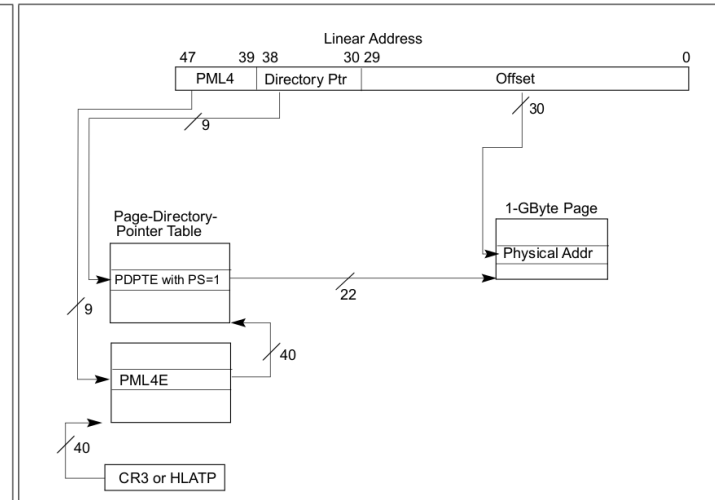
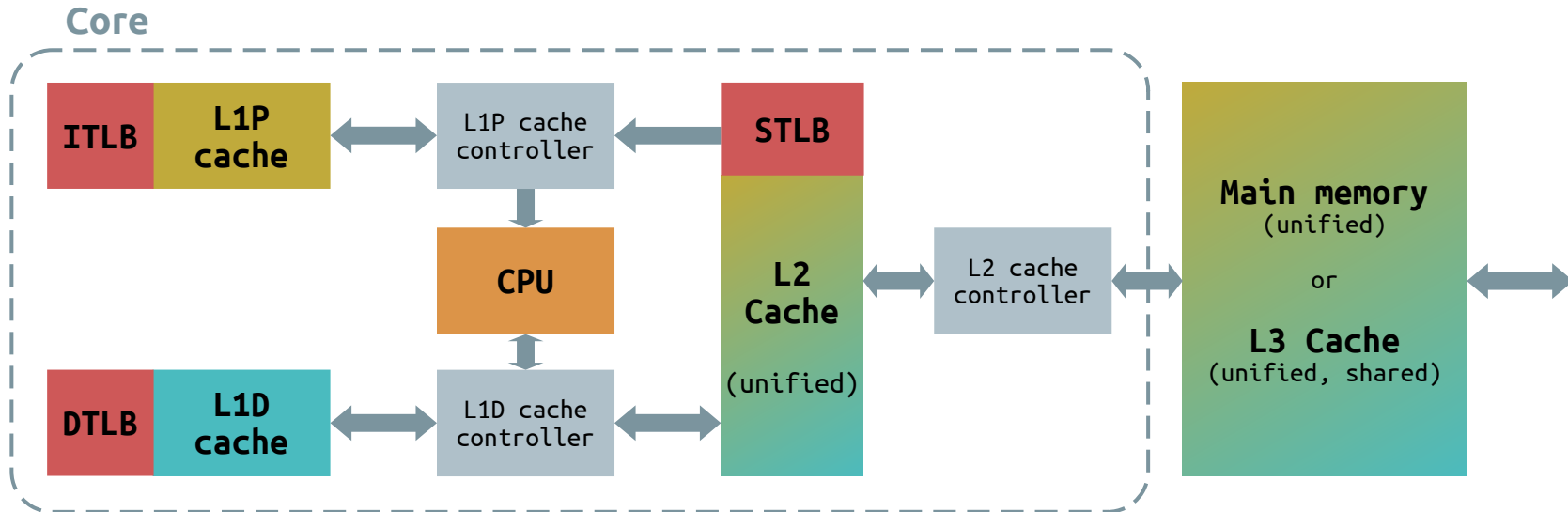


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

Pagination : Translation Lookaside Buffer

Un dernier mécanisme d'optimisation utilisé afin d'accélérer les mécanismes de translation d'adresse est d'utiliser de petites mémoires cache associatives (**Table Lookaside Buffer** ou **TLB**) chargées de sauver les entrées les plus couramment appelées. Les stratégies de remplacement des entrées des TLB sont semblables aux techniques de gestion des caches processeur (LRU, random, FIFO ...).



Pagination : Translation Lookaside Buffer

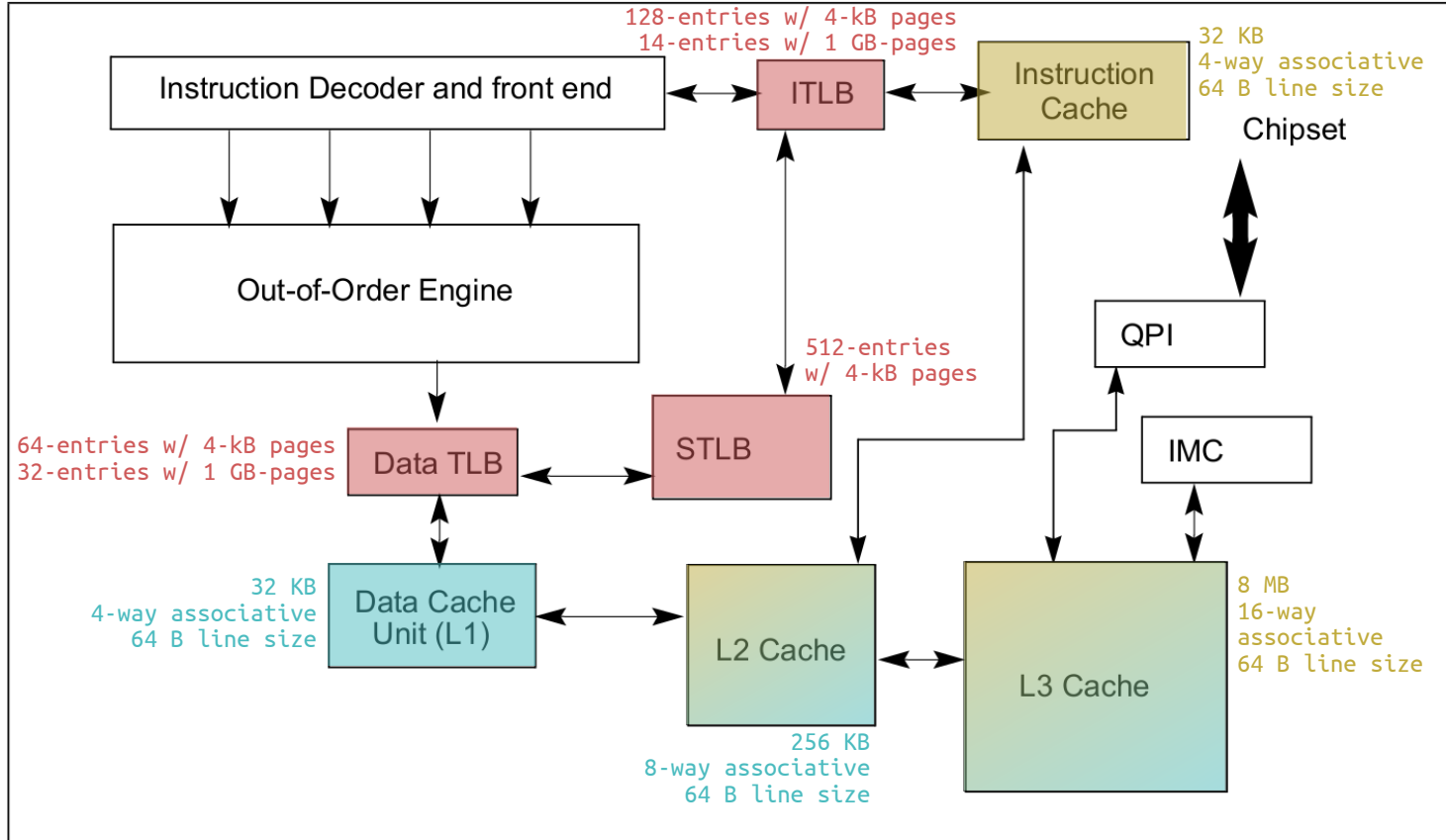


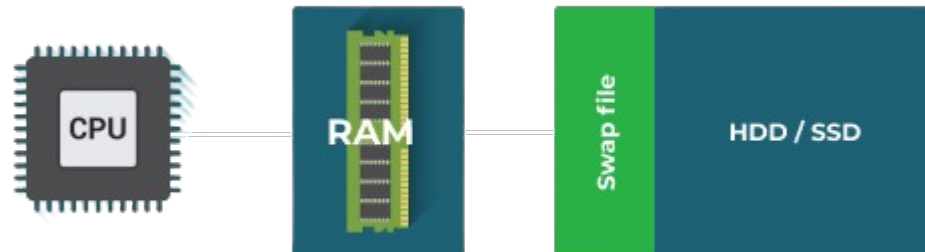
Figure 12-2. Cache Structure of the Intel Core i7 Processors

Swap

L'espace d'adressage virtuel pouvant être supérieur à l'espace d'adressage physique, le système d'exploitation utilise une zone du disque dur : le **swap (espace d'échange)**.

Cette zone doit être vue comme une extension de la **mémoire principale (RAM)** située sur la **mémoire secondaire (disque dur)**.

Il s'agit d'une zone d'échange de données, et non pas d'une mémoire cache (copie).

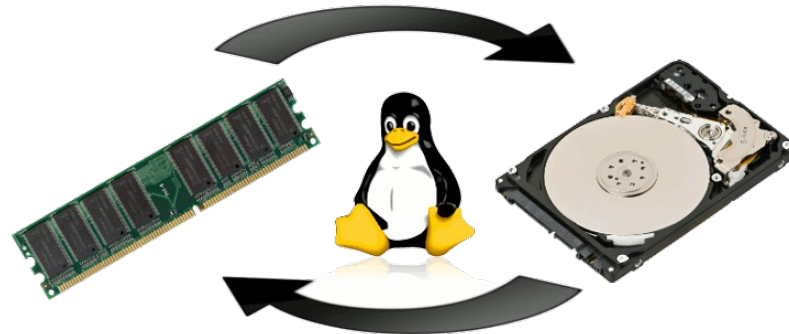


Swap

Sous Windows, le swap est un fichier nommé « pagefile.sys ». Sous Linux, il s'agit d'une partition logique du disque dur.

Dans les deux cas, cette zone doit être exploitée le moins possible afin de ne pas dégrader les performances du système. Le swap sera utilisé pour les applications gourmandes en mémoire, durant la mise en veille. Pour Linux, on conseille une taille de swap comprise entre 1x et 1,5x la taille de la mémoire principale.

Le choix du disque sur lequel se trouve le swap impacte également les performances.



Swap

Il est possible de jouer sur la politique de remplacement des pages en forçant Linux à swap le moins possible et exploiter au mieux les ressources en mémoire principale.

Pour ce faire il faut modifier le paramètre système `/proc/sys/vm/swappiness`. Par défaut il vaut 60, il est modifié à 10 dans l'exemple ci-dessous.

```
dboudier:~$ swapon -s
Filename                Type          Size      Used      Priority
/swapfile                file          2097148  0         -2
```

```
dboudier:~$ cat /proc/sys/vm/swappiness
60
```

```
dboudier:~$ sudo sysctl vm.swappiness=10
vm.swappiness = 10
```

```
dboudier:~$ sudo swapoff -av
swapoff /swapfile
dboudier:~$ sudo swapon -av
swapon: /swapfile: found signature [pagesize=4096, signature=swap]
swapon: /swapfile: pagesize=4096, swappsize=2147483648, devsize=2147483648
swapon /swapfile
```

```
dboudier:~$ cat /proc/sys/vm/swappiness
10
```

Swappiness = 0 :
si possible, usage massif de
la mémoire vive

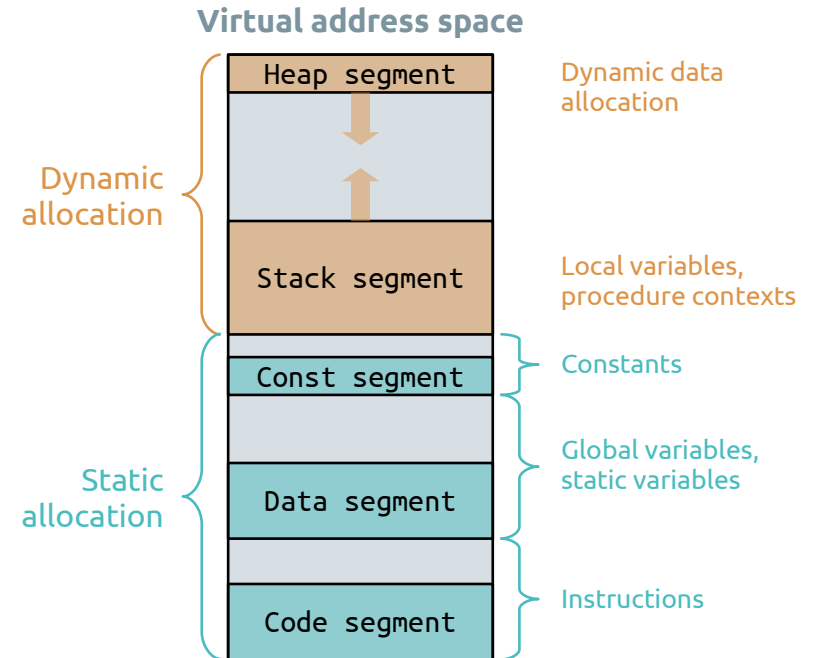
Swappiness = 100 :
usage massif du swap en
mémoire secondaire

Segmentation

L'espace d'adressage offert par la pagination est unidimensionnel : toutes les adresses se suivent commençant de 0x00...00 à une valeur maximale.

Pour un processus par exemple, la taille des segments de Code, Data (r/w) et Data (ro) est ajustée pendant la phase de compilation (**allocation statique**) tandis que la taille des segments de Heap et de Stack évolue au cours de l'exécution (**allocation dynamique**).

Chaque segment a un espace mémoire dédié, mais si le nombre de variables globales ou de variables locales est très élevé, alors deux segments risquent de se chevaucher.



Segmentation

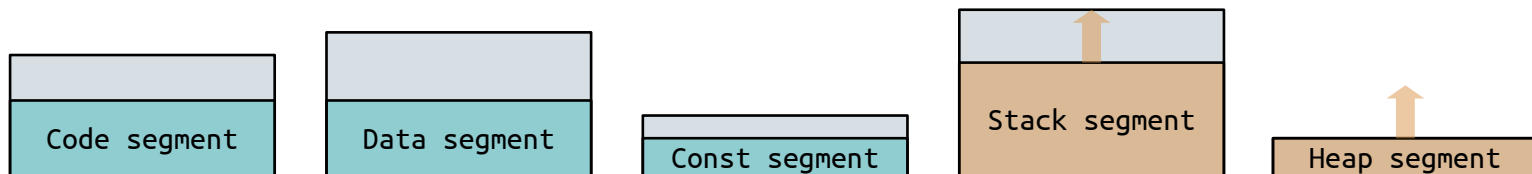
Afin de palier ce problème chaque segment aura son propre espace d'adressage, commençant à $0x00...00$ jusqu'à un certain maximum : c'est la **segmentation**.

Ainsi on passe désormais à un espace d'adressage en deux dimensions, soit une adresse linéaire (comme avant) plus un numéro de segment.

L'avantage apporté par la segmentation est donc que chaque segment commence à l'adresse $0x00...00$, facilitant ainsi le linkage d'une part, et le partage de segments entre processus d'autre part.

De plus, chaque segment peut avoir un niveau de privilège qui lui est propre.

Segmented
memory

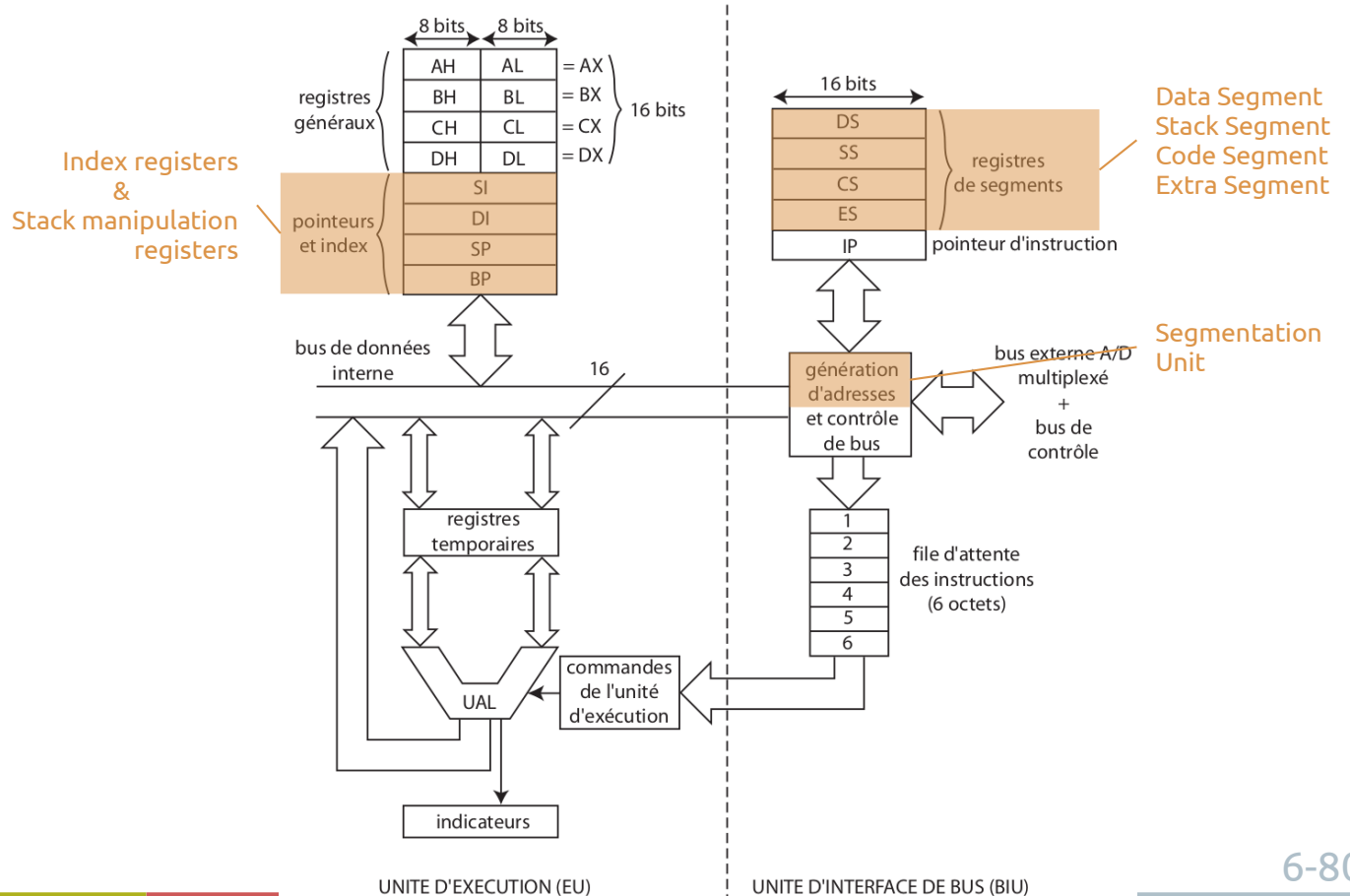


Segmentation

Le Intel 8086 dispose de registres dédiés à la manipulation de 4 segments mémoire de 16 bit d'adresse.

Rappelons que le 8086 dispose de 20 broches pour le bus d'adresse physique.

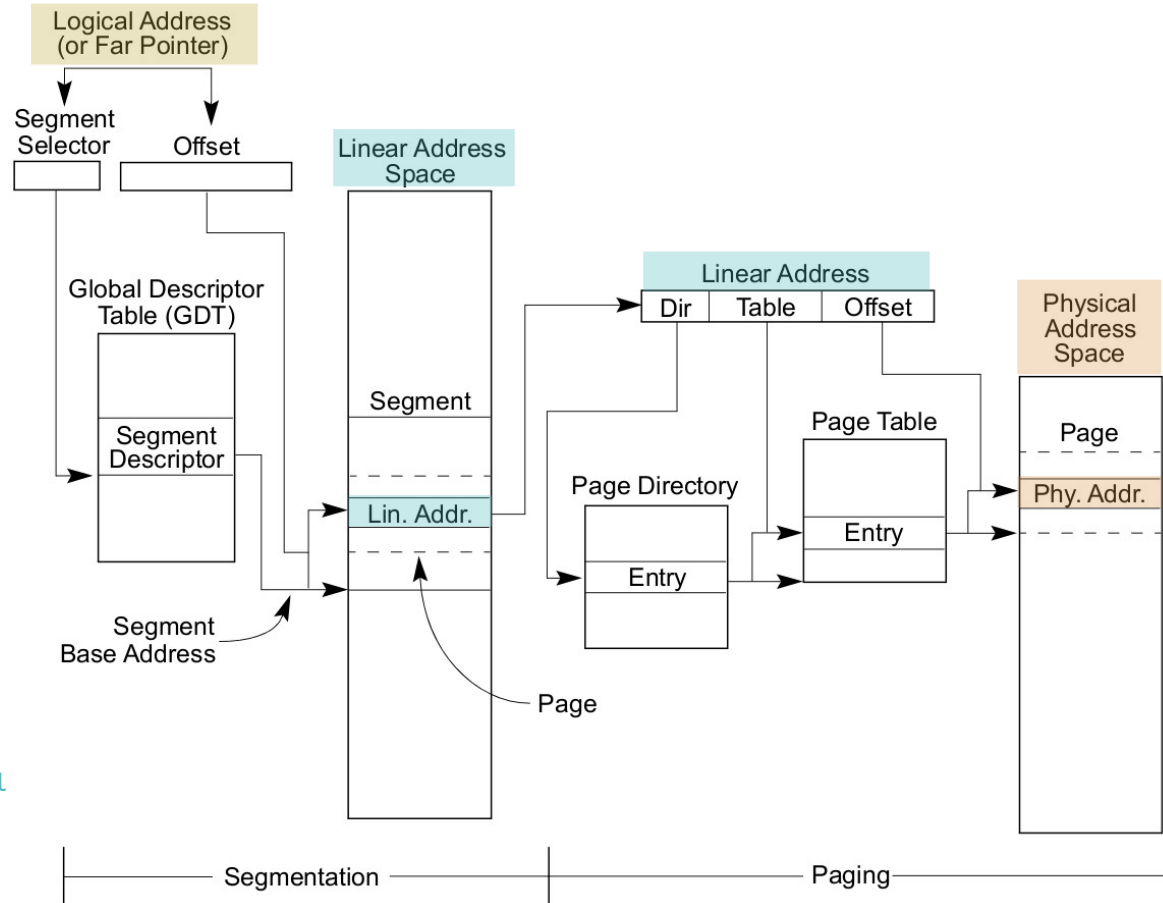
Sur processeurs x86-64 actuels, ces registres sont toujours 16-bit mais de nouveaux registres sélecteurs ont été ajoutés (FS, GS).



Segmentation et Pagination

Segmentation et pagination peuvent être utilisées conjointement.

Sur les Intel Core, la segmentation ne peut être désactivée et la pagination est optionnelle. C'est quand on sait que Linux utilise peut la segmentation est intensément la pagination !



Le noyau Linux utilise très peu la segmentation mémoire, contrairement à la pagination.

En effet Linux a vocation à être multi-plateforme, or de nombreuses architectures RISC ne gèrent pas la segmentation.

De plus le travail de la MMU se trouve grandement simplifiée si elle n'a à gérer qu'un seul espace d'adressage linéaire (et pas plusieurs espaces parallèles).



Résumé

Pagination : Mécanisme offrant au CPU un espace mémoire virtuel continu et contigu masquant la fragmentation de la mémoire physique. L'espace mémoire virtuel paginé peut dépasser la taille de la mémoire physique. Mécanisme de virtualisation très répandu (GPP, AP, certains MCU) et utilisé par Windows et Linux (entre autres).

Swap : Espace d'échange sur la mémoire secondaire (disque dur) permettant de stocker temporairement une partie du contenu de la mémoire principale (RAM). Linux l'implémente sous forme de partition du disque dur, Windows sous forme d'un fichier.

Segmentation : Mécanisme permettant d'offrir plusieurs espaces d'adressage distincts pour différents segments d'un processus (stack, heap, code, data, ...). Mécanisme plutôt historique et peu répandu sur architectures RISC, donc très peu utilisé par Linux.

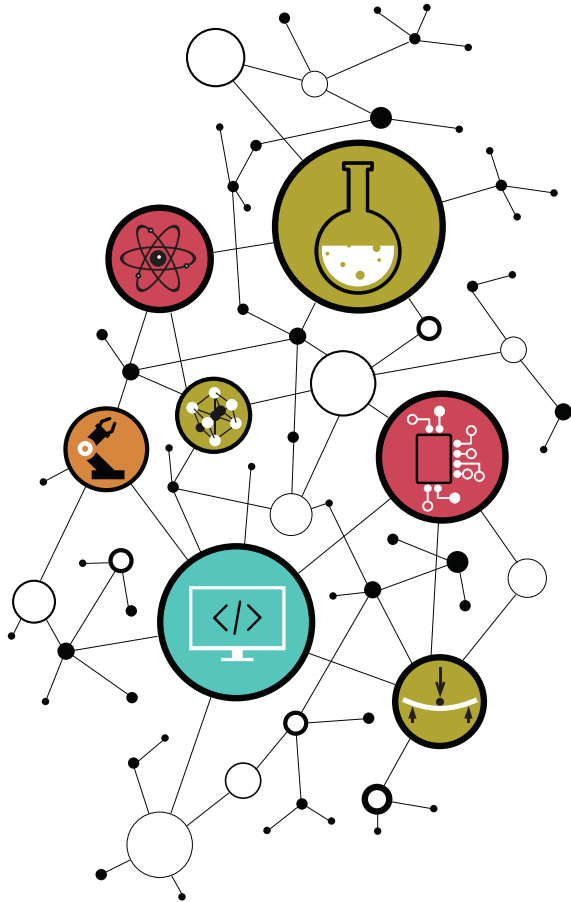
Memory Management Unit, MMU : Unité matérielle, accolée à un CPU (donc plusieurs MMU par GPP) implémentant les mécanismes sus-nommés.

Résumé

Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

Figure 6-9. Comparison of paging and segmentation.

CONTACT



Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)



Except where otherwise noted, this work is licensed under
<https://creativecommons.org/licenses/by-nc-sa/4.0/>