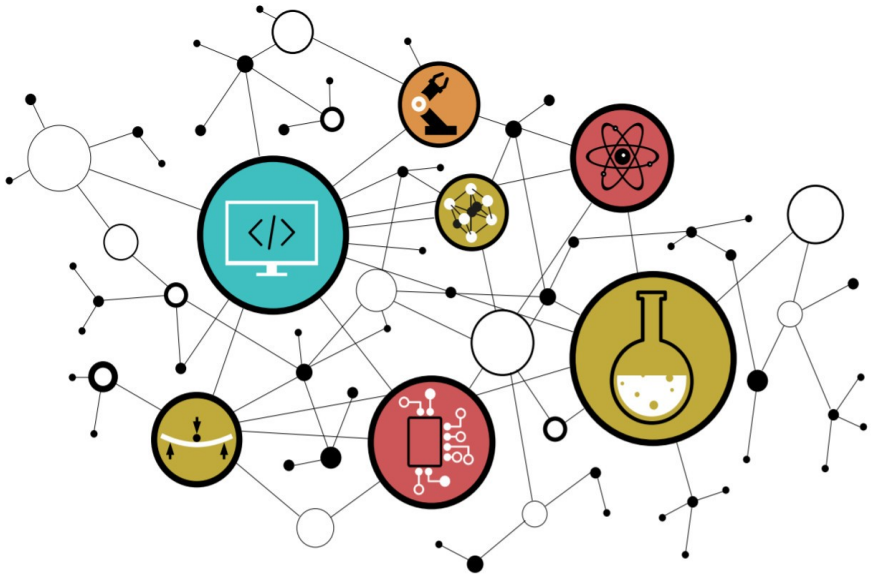
TRAVAUX PRATIQUES

SYNTHÈSE



Hugo Descoubes – neku – GNU\Linux 5.8 - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

Le trame de TP minimale à réaliser regroupant ce que je considère (hugo) comme être les compétences minimales à acquérir suit le séquencent suivant : chapitres 1, 2, 3, 4, 5, 6. La suite de la trame ne sera pas évaluée en examen et représente une extension au jeu de compétences minimales relatif au domaine en cours d'étude.

SYNTHÈSE

1. PRÉLUDE

« Unix is basically a simple operating system, but you have to be a genius to understand the simplicity. »

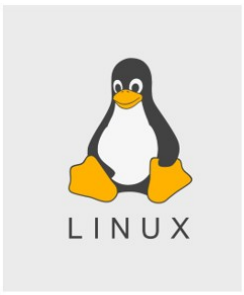
Dennis Ritchie, père du langage C et codéveloppeur d'Unix, à droite sur la photo ci-dessous

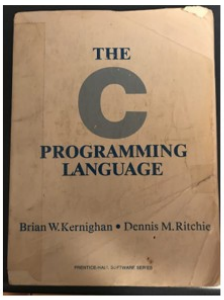
« I think the major good idea in Unix was its clean and simple interface: open, close, read, and write. »

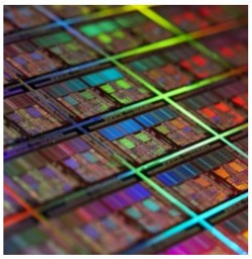
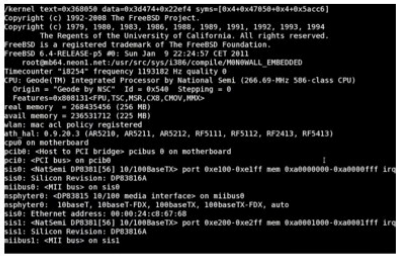
Kenneth Thompson, père du système d'exploitation Unix et du langage B, à gauche sur la photo

« Most of the good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program. »

Linus Torvalds, père de noyau Linux





Cet enseignement sera illustré sur les technologies et les standards logiciels les plus répandus à notre époque dans le monde des couches basses des systèmes numériques de traitement de l'information, notamment sur ordinateur (langages, systèmes d'exploitation, noyaux, compilateurs, systèmes embarqués, etc). Langage C (1972), système d'exploitation GNU (1983), chaîne de compilation GCC (C ANSI, 1986), interface standard POSIX (1988), noyau Unix-like Linux (1991), etc, la plupart de ces standards, projets et technologies sont aux centres de la majorité des systèmes numériques d'information autour de nous de nos jours. Certains ont notamment inspiré et marqué l'émergence des concepts de logiciel libre et d'open source.

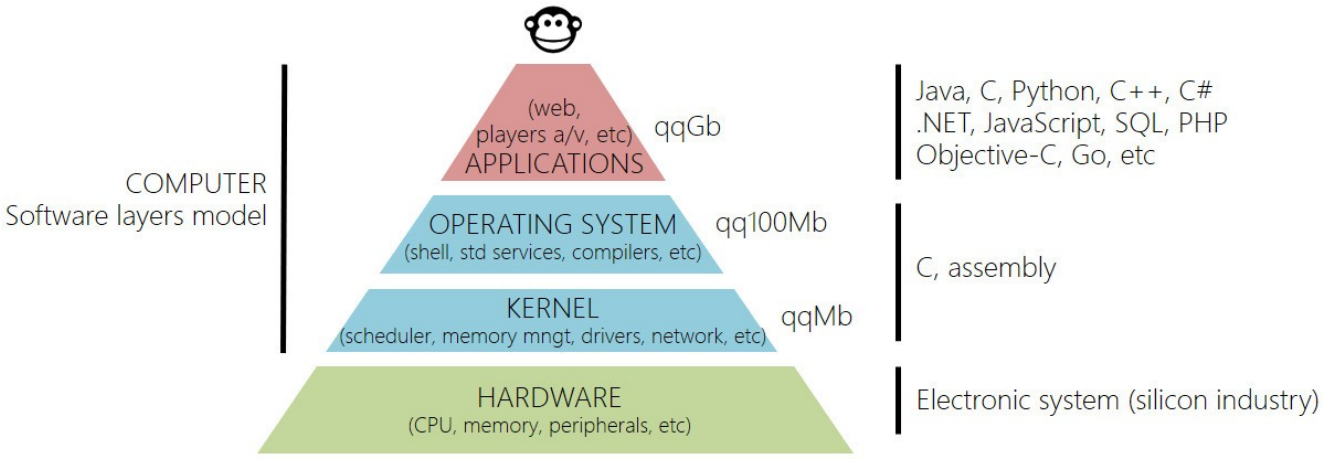
Les technologies matérielles choisies pour illustrer l'enseignement seront quant à elles celles restant toujours à notre époque les plus répandues sur ordinateur, soit les architectures compatibles x86/x64 (32bits/64bits). Ces technologies ont été historiquement développées et spécifiées par Intel (x86 IA-32 en 1985, compatibilité avec le processeur 8086 en 1978) puis AMD (x64 AMD64 compatible x86 en 2003), Intel étant la première société à avoir produit un microprocesseur (Intel 4004 en 1971). Avant de présenter les objectifs de cet enseignement, quelques précisions concernant le langage C, son essence, son histoire et donc sa dominance dans le monde des couches basses des systèmes sont présentées à la suite …

« C is declining somewhat in usage compared to C++, and maybe Java, but perhaps even more compared to higher-level scripting languages. It's still fairly strong for the basic system-type things. »

« The only way to learn a new programming language is by writing programs in it. »

Dennis Ritchie

Le langage C fut historiquement développé par Dennis Ritchie (AT&T, Bell Labs) au début des années 70's afin de réécrire Unix (un système d'exploitation). Le langage C est une évolution du langage B (développé par Ken Thompson, père d'Unix). Rappelons qu'à notre époque les systèmes Unix-Like (Linux, distributions GNU, macOS, Android, BSD, System V, etc) sont les plus répandus et toujours en pleine expansion sur les marchés. Il s'agit d'un langage de programmation impératif de bas niveau (proche de la machine). De nombreux langages plus modernes, mais néanmoins adaptés à d'autres usages, s'en sont inspirés (C++, Java, D, C#, PHP, JavaScript, etc). Même à notre époque, le C continu d'évoluer (normes C ANSI/C89 en 1989, C90, C95, C99, C11 et C18 en 2018, https://www.iso.org/standards.html).



Le langage C a pour intérêt d'avoir été pensé pour les couches basses des systèmes logiciels de supervision des machines numériques. Il offre un faible niveau d'abstraction au matériel et permet des analyses et des développements de plus bas niveau poussés et flexibles (assembleur, binaire, etc). Il s'agit d'un langage ''épuré'' (comparé à d'autres comme le C++, D, etc), versatile et permissif (la compilation et l'exécution tolère beaucoup de marge de manœuvre) offrant un contrôle important sur la machine, notamment au regard de la gestion mémoire (débordements, fuites, traces, etc). Il est donc à manier avec précaution et attention par le développeur. Vous en serez témoin à travers cette trame d'expérimentations. C'est d'ailleurs pour cela qu'il est toujours à notre époque utilisé pour développer les couches basses des systèmes (systèmes d'exploitation, systèmes embarqués, noyaux, compilateurs, interpréteurs, etc). Pour un développeur système, la complexité ne doit pas résider dans le langage mais dans le système !

A notre époque, il reste toujours parmi les langages les plus utilisés au monde (n°1 en 2020, source TIOBE index, https://www.tiobe.com/tiobe-index/). Il est d'ailleurs en constante croissance à l'usage, notamment avec l'avènement des systèmes embarqués, de l'IOT (Internet des objets) et des projets « maker » (Raspberry Pi, Arduino, etc). Prenons des objets et logiciels de votre quotidien dans lesquels une partie voire la totalité des développements logiciels ont été développés en langage C : Box internet, télévision, lecteurs CD/DVD/Blu-ray, enceinte Bluetooth, ordinateur et smartphone (systèmes d'exploitation - distributions GNU\Linux telles que Debian/Ubuntu/Redhat/Fedora/Kali/etc, Android, Mac OS X, iOS, Windows, etc) et noyaux (Linux, XNU, Darwin, Mach, NT, etc)), la liste est très très très longue ...

Objectifs pédagogiques

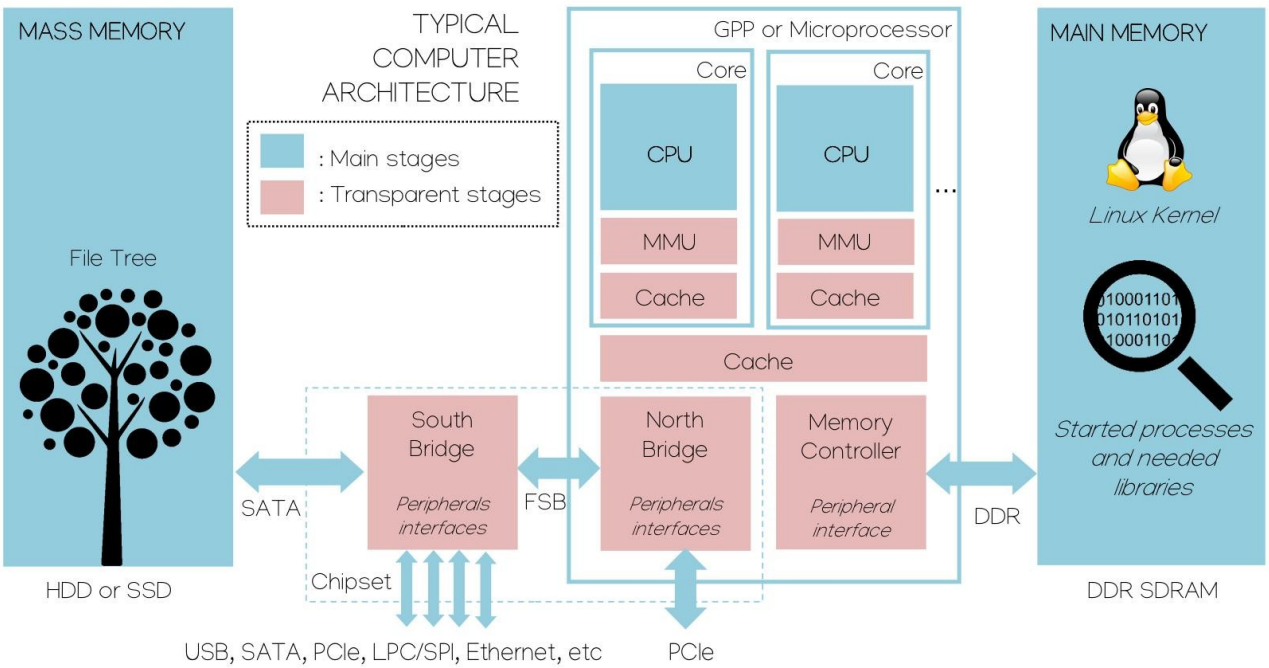
Cette trame d'enseignement ne cherche pas à vous réapprendre un langage de programmation. Il s'agit d'une trame d'analyse de programmes C élémentaires ayant pour objectif terminal d'aboutir à une meilleure représentation et compréhension du fonctionnement de la machine (ordinateur), de la compilation à l'exécution sur cible. L'objectif premier étant de mieux comprendre l'alchimie liant le système d'exploitation (operating system) au système d'exécution (hardware), mais également de maîtriser le processus de génération de micrologiciel (firmware) en partant d'un programme source (software).

Néanmoins, pour les plus clairvoyants, une bonne assimilation des connaissances et compétences enseignées, qu'elles soient architecturales ou techniques, pourrait bien changer à jamais votre façon de voir et d'écrire vos programmes à l'avenir. Ceci sera vrai, que vous deveniez développeur applicatif haut niveau (C++, Java, D, etc), tout comme développeur système bas niveau (C, assembleur).

Il ne s'agit donc pas d'une trame de travaux pratiques visant à apprendre à programmer. Néanmoins, nous nous attarderons sur des points que les enseignements d'initiation à la programmation en C passent souvent très rapidement, car nécessitant une meilleure compréhension des couches basses du système. Prenons les exemples des qualificateurs de type et de fonction spéciaux, des classes de stockage, etc (register, volatile, inline, static , const, restrict, etc).

Architecture matérielle (représentation physique)

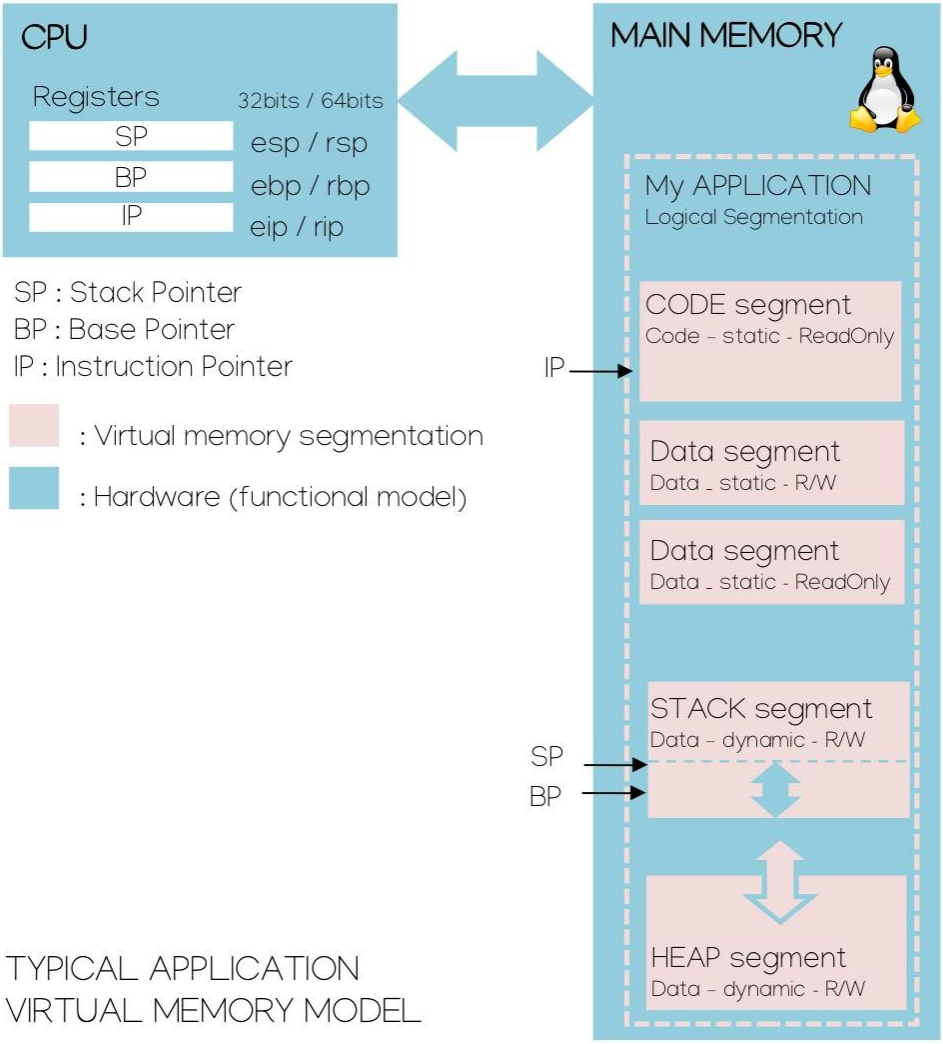
Les systèmes matériels actuels de traitement de l'information, tel qu'un ordinateur, sont devenus d'une grande complexité architecturale et technologique. A titre indicatif, à Caen chez NXP, le développement d'un simple composant sécurisé NFC (chiffrement de l'information) demande le travail direct de près de 700 ingénieurs. Les phases de cours seront là pour mieux comprendre les rôles des principaux éléments constitutifs de l'ordinateur. Nous nous attarderons sur les étages pouvant potentiellement un jour jouer un rôle sur les contres performances de vos programmes (cache processeur, MMU, etc).



Architecture matérielle (représentation logique)
Environnement mémoire segmenté d'une application

Une fois l'environnement mémoire de l'application virtualisé, mappé et protégé (segmentation logique), puis son code et ses données statiques binaires chargées en mémoire principale depuis la mémoire de masse par le noyau du système d'exploitation (Linux par exemple), il ne reste alors plus qu'à exécuter l'application sur le CPU courant. Le modèle de plus bas niveau de représentation de la machine vu du CPU (étage registre) et du développeur (langage d'assemblage) reste à ce niveau relativement simple (cf. schéma ci-dessous). Cette machine minimale est souvent historiquement nommée machine de Von Neumann (mathématicien 1945, projet EDVAC). Il s'agit d'un modèle de machine à mémoire unifiée pour le stockage du code et des données (contrairement aux architectures de Harvard). Durant des phases de développement, une compréhension fine des contraintes amenées par cette segmentation logique représentant les limites environnementales en mémoire d'une application (frontières/périmètre), permet alors de garantir un réel durcissement d'un programme durant l'édition puis l'exécution.

La bonne compréhension des points précédemment cités ainsi que des deux schémas présentés (Architecture matérielle – représentations physique et logique) fait partie des attentes terminales de cet enseignement.

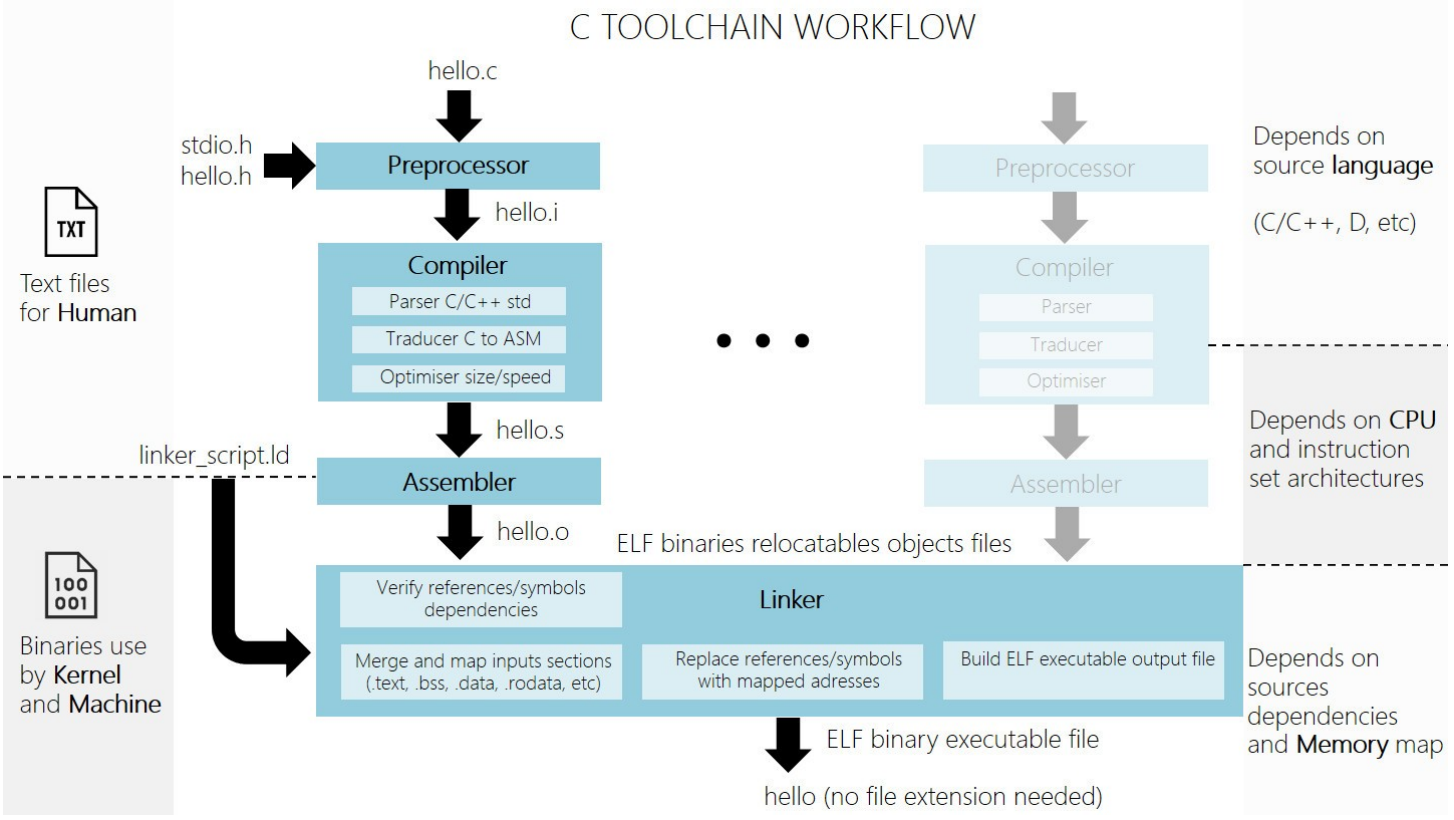


2. COMPILATION ET ÉDITION DES LIENS

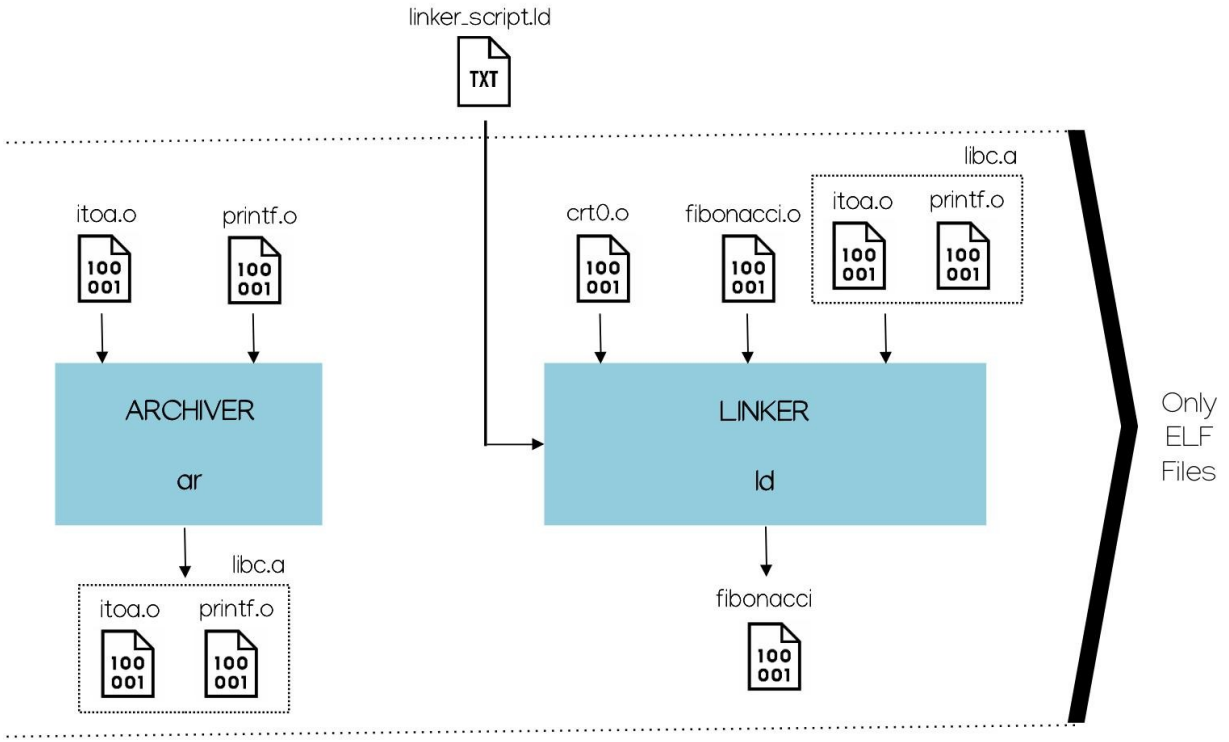
Dans ce chapitre, nous allons nous intéresser aux différences étapes du processus de compilation et d'édition de liens d'un projet logiciel, dans notre cas développé en langage C. Afin d'appréhender ce workflow, nous analyserons la compilation d'un programme élémentaire constitué d'un fichier source unique *disco/toolchain/src/hello.c* incluant un fichier d'en-tête applicatif élémentaire *disco/toolchain/include/hello.h*. Pour la suite de cet exercice, se placer dans le répertoire */disco/toolchain/* afin d'appliquer la totalité des commandes qui suivent. Le fichier *README.md* contient la séquence d'exécution complète de l'exercice.

La compilation (travail d'analyse et de traduction) se décompose en 3 grandes étapes. Le prétraitement (préparation du code avant compilation – analyse lexicale, supprime, copie, colle, remplace), la compilation proprement dite (analyse syntaxique et sémantique, traduction vers l'assembleur de l'architecture CPU cible et optimisation optionnelle) et enfin l'assemblage (translation d'un programme assembleur vers son équivalent binaire pour la machine cible, sans résolution des adresses mémoire en gardant des références symboliques et génération d'un fichier binaire ré-adressables au format ELF). L'étape suivant la compilation est l'édition des liens (analyse et validation des dépendances entre fichiers et références symboliques, placement mémoire et résolution des adresses des symboles statiques, génération dans un format donné ELF/COFF/HEX/etc du fichier binaire exécutable de sortie).

Le schéma ci-dessous, qui aura à être assimilé en fin d'exercice, représente ces différentes étapes exécutées séquentiellement par la toolchain. La compilation est un processus indépendant au sens où si vous avez plusieurs fichiers source à compiler dans un même projet en développement, le processus de compilation (preprocessing, compiling et assembling) sera réalisé indépendamment pour chaque fichier source d'un projet avant l'édition des liens (linking) travaillant avec l'ensemble des fichiers objets binaires ré-adressables par références symboliques (relocatable) générés à la compilation. Le résultat de ce processus statique étant la génération d'un fichier binaire exécutable, dans notre cas au format binaire ELF (Executable and Linkable Format) sur système Unix.



Bibliothèque statique



Une bibliothèque statique est une archive (généralement avec une extension .a comme archive), soit la concaténation de fichiers objets binaires ELF ré-adressables. Une bibliothèque statique n'est pas un fichier ELF, mais en revanche elle regroupe de fichiers ELF.

```
as --32 -g ../toolchain/build/startup/crt0.s -o obj/crt0.o

as --32 -g fibonacci.s -o obj/fibonacci.o

as --32 -g itoa.s -o obj/itoa.o

as --32 -g printf.s -o obj/printf.o

ar -rcs lib/libc.a obj/itoa.o obj/printf.o

ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld
obj/crt0.o obj/fibonacci.o lib/libc.a -o bin/fibonacci
```

3. ASSEMBLEUR X86

```
main:
     pushl   %ebp
     movl    %esp,%ebp
     movl    $0,%eax
     popl    %ebp
     ret
```

L'assembleur (assembly) ou langage d'assemblage est le langage de programmation de plus bas niveau sur la machine. Il est la conversion directe lisible voire éditable par l'homme (texte) du programme exécutable par le CPU de la machine (binaire). Il est de ce fait, le langage le moins universel au monde, car dépendant du jeu d'instructions supporté par le CPU cible. Entre les marchés des ordinateurs et des systèmes embarqués, un grand nombre de fabricants implémentent des modèles d'exécution (RISC ou CISC, Von Neumann ou Harvard, 8-16-32-64bits, entier voire flottant, VLIW ou superscalaire, vectoriel ou scalaire, etc) sur des technologies différentes (x86, x64, ARM, MIPS, C6000, PIC18, etc). L'assembleur présenté ci-dessus est par exemple de l'assembleur 32bits IA-32 (Intel Architecture 32 bits) souvent nommé x86 et supporté par des technologies CPU compatibles (IA-32 chez Intel et AMD). Comme tout langage de programmation, un programme assembleur se lit de haut en bas, à l'image du modèle d'exécution séquentiel d'un CPU. Même si l'assembleur n'est pas universel, certaines représentations liées au langage peuvent néanmoins être généralisées :

```
label:    instruction    opérandes
```

- Label : un label ou étiquette, est une référence symbolique (simple chaîne de caractères) représentant l'adresse mémoire logique (emplacement) de la première instruction suivant celui-ci. Les labels sont remplacés par les adresses logiques voire physiques à l'édition des liens. Un label se termine par : afin de le différencier d'éventuelles directives d'assemblage voire d'instructions.
- Instruction : une instruction est un traitement élémentaire à réaliser par le CPU. Par exemple, charger/load ou sauver/store une donnée depuis ou vers la mémoire principale, réaliser une opération arithmétique ou logique, déplacer une donnée de registre à registre, etc. L'ensemble des instructions exécutables par un CPU est nommé jeu d'instructions (ISA ou Instruction Set Architecture). l'ISA représente ce que sait faire nativement un CPU.
- Opérandes : les opérandes, lorsque l'instruction en utilise, sont les données ou les emplacements de données (registres ou adresses en mémoire principale) manipulées par l'instruction. Nous distinguons les opérandes sources, utilisées comme entrées avant l'exécution de l'instruction, de l'opérande de destination pour sauver le résultat. Les méthodes d'accès aux données en assembleur sont nommées des modes d'adressage :
    - Mode d'adressage registre : l'opérande est un registre CPU dans lequel est sauvé une donnée. Par exemple, les instructions *push*, *mov* et *pop* ci-dessus.
    - Mode d'adressage immédiat : l'opérande est une constante dont la valeur sera sauvée dans le code binaire de l'instruction. Par exemple, l'instruction *mov $0* ci-dessus
    - Mode d'adressage direct (accès mémoire) : l'opérande est directement l'adresse de la case mémoire de la donnée
    - Mode d'adressage indirect (accès mémoire) : l'opérande est l'adresse de la case mémoire de la donnée stockée indirectement dans un registre CPU.

Syntaxe assembleur AT&T proposée par GNU AS

Syntaxe Intel	Syntaxe AT&T

main:
push ebp
mov ebp,esp
mov eax, 0
pop ebp
ret
	main:
	pushl %ebp
	movl %esp,%ebp
	movl $0,%eax
	popl %ebp
	ret

L'assembleur x86 est une ISA développée historiquement par Intel pour son CPU 16bits 8086 produit en 1978. Les générations suivantes de processeurs Intel et AMD sorties dans les années 80 (80286, 80386, etc) sont restées compatibles avec leur prédécesseur. Ce langage d'assemblage s'est donc nommé au fil du temps x86. Il est à noter que les processeurs 64bits compatibles x64 (IA-64 chez Intel et AMD64 chez AMD) restent également rétrocompatibles x86. Ceci reste également toujours vrai à notre époque (technologies Pentium, Core2, Corei, etc).

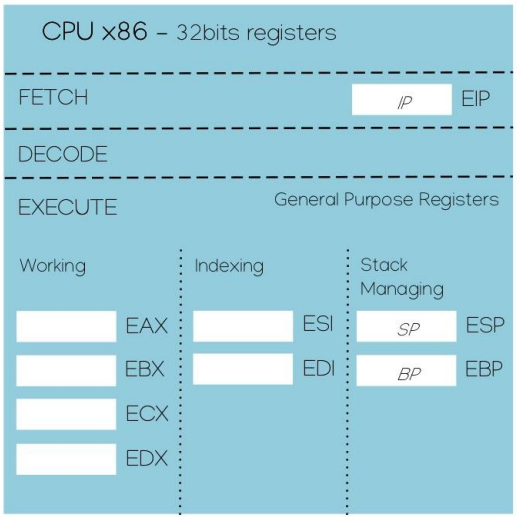
Le langage C et sa syntaxe sont maintenant normalisés et standardisés depuis des décennies même si la norme continue d'évoluer (normes ANSI C, C89, C99, C18, etc). En revanche, différentes syntaxes assembleur liées à la chaîne de compilation et aux outils de développement (ouverts ou fermés, libres ou propriétaires) existent sur le marché. Prenons l'exemple ci-dessus d'un même programme assembleur en syntaxe AT&T (généré par AS ou GAS ou GNU AS – étage d'assemblage de GCC et généralisé sur système Unix-like) et en syntaxe Intel (généré par ICC – Intel C++ Compiler). Nous pouvons constater ci-dessus que les opérandes sources et de destinations ne sont pas placées dans le même sens (destination à droite en syntaxe AT&T). Observons quelques particularités de la syntaxe AT&T :

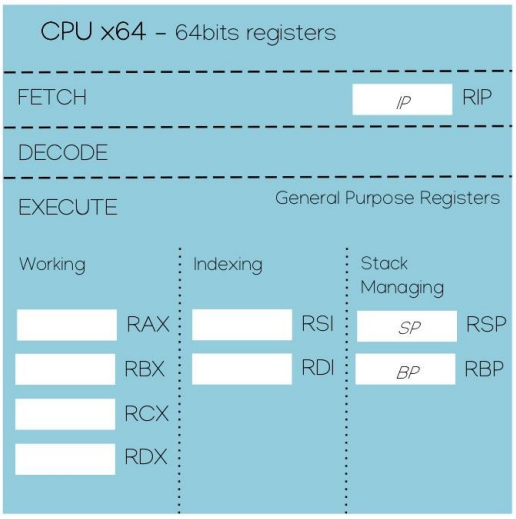
- % : signifie que l'opérande qui suit est un registre CPU (mode d'adressage registre)
- $: signifie que l'opérande que suit est une constante (mode d'adressage immédiat)
- suffixe d'instruction : signifie que l'instruction manipule des opérandes d'une certaine longueur en octets : b=byte=8bits=1octet, s=short=16bits=2octets, l=long=32bits=4octets et q=quad=64bits=8octets. Ce suffixe est facultatif à l'édition.
- (%registre) ou ($adresse) : signifie que l'instruction nécessite de charger ou de sauver une donnée depuis ou vers la mémoire principale (respectivement modes d'adressages indirect et direct). Par exemple, en mode d'adressage indirect, si le pointeur BP (adresse) est sauvé dans EBP (registre), l'opérande avec offset suivante -4(%ebp) se lit en pseudo-code *(BP – 4), soit accès (lecture ou écriture) à la case mémoire pointée par l'adresse BP – 4o

```
    .global main

    .text
main:
    pushl   %ebp
    movl    %esp,%ebp
    movl    $0,%eax
    popl    %ebp
    ret
```

Un programme assembleur intégrera également certaines directives d'assemblage préfixées par un point (.global, .text, .macro, etc). Celles-ci renseignent l'outil chargé de convertir l'assembleur en binaire (également nommé assembleur en français ou *assembler* en anglais) ainsi que l'éditeur des liens. Ces directives fixent par exemple les sections du firmware où ranger le code et les données statiques (.section, .text, .data, .rodata, etc), étendent les portées de labels à l'éditeur des liens (.global), définissent des macros (.macro, .endm), etc (https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html).

Registres CPU x86-64 et environnement d'exécution assembleur





Les schémas ci-dessus présentent les principaux registres d'usages généralistes présents dans les CPU d'architectures compatibles x86 (32bits) et x64 (64bits). Rappelons que ces architectures restent rétrocompatibles avec le CPU 16bits 8086 datant de 1978. Ces registres sont présents dans chaque CPU d'une machine multicœurs (un cœur est l'ensemble CPU, MMU et Caches locaux). Cependant, d'autres registres aux usages plus spécifiques existent dans chaque CPU. Présentons-les succinctement (non vus en enseignement) :

- Registre d'état FLAGS (CF, PF, ZF, etc) : registre contenant les différents flags d'états associés aux unités d'exécution arithmétiques et logiques (carry, zero, sign, etc). Utilisé notamment afin d'implémenter des sauts conditionnels (if, else if, while, for, etc).

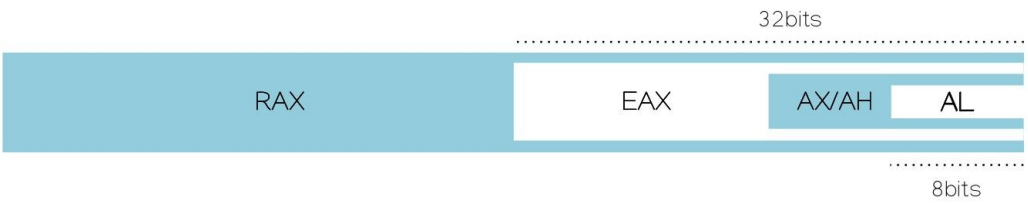
- Registres de contrôle (CR0 à CR7) : registres permettant de contrôler les services matériels du CPU (mode protégé, etc), ainsi que du CACHE et de la MMU associés au cœur courant.

- Registres vectoriels (XMM0 à XMM15 128bits extension ISA SSE, YMM0 à YMM15 256bits extension ISA AVX et ZMM0 à ZMM15 512bits extension ISA AVX-512) : registres de travail pour les instructions vectorielles dans un contexte d'optimisation algorithmique

- Registres de segments (CS, DS, ES, SS, FS, GS) : registres historiquement utilisés lorsque la segmentation logique d'une application nécessitait un support d'adressage physique. La segmentation est maintenant virtualisée et gérée entièrement logiquement par le noyau du système à travers la gestion de la PMMU (Paged MMU ou unité de pagination).

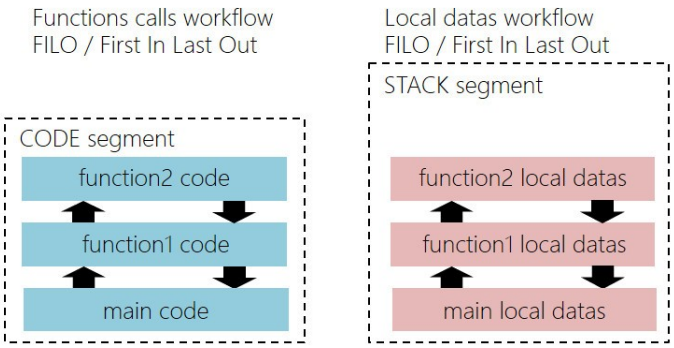
- Registres de test (TR3 à DR7), registres de debug (DR0 à DR7) et registres d'accès à la table de pagination mémoire (GDTR, LDTR, IDTR)

Pour des soucis de rétrocompatibilité, toute nouvelle architecture compatible x64 doit rester compatible avec les générations antérieures x86. Cette rétrocompatibilité remonte jusqu'au 8086. Par exemple, les registres 16bits et 8 bits de ce processeur sont toujours supportés à notre époque. Prenons l'exemple du registre à usage général A (Accumulator), déjà présent sur Intel 4004 en 1971 (premier microprocesseur), ainsi que ses déclinaisons 8-16-32-64bits imbriquées les unes dans les autres sur architectures x86-64 (respectivement AL/AH 8bits, AX 16bits, EAX 32bits et RAX 64bits). L'exemple donné ci-dessous sur le registre 64bits RAX peut être étendu aux registres de travail généralistes du CPU.



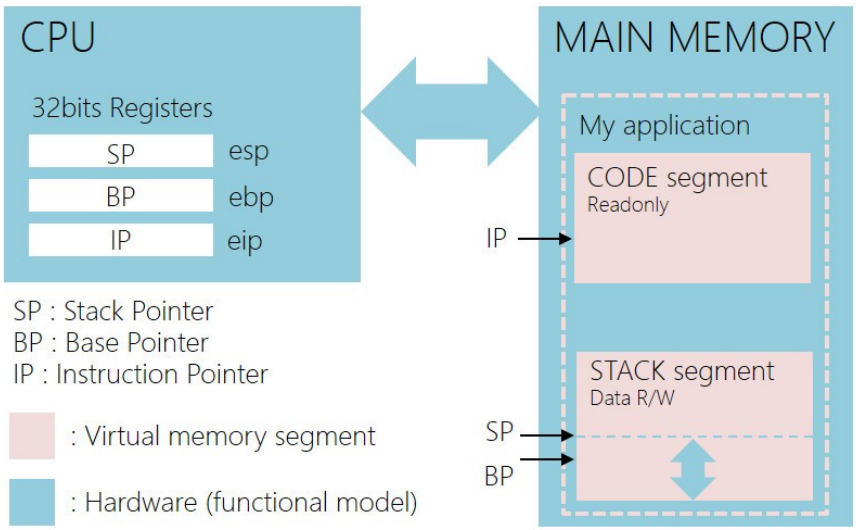
4. ALLOCATIONS AUTOMATIQUES ET SEGMENT DE PILE

```
int main (void)
{
    /* local user code and datas */
    function1();
   return 0;
}

void function1 (void)
{
    /* local user code and datas */
   function 2();
}

void function2(void)
{
    /* local user code and datas */
}
```

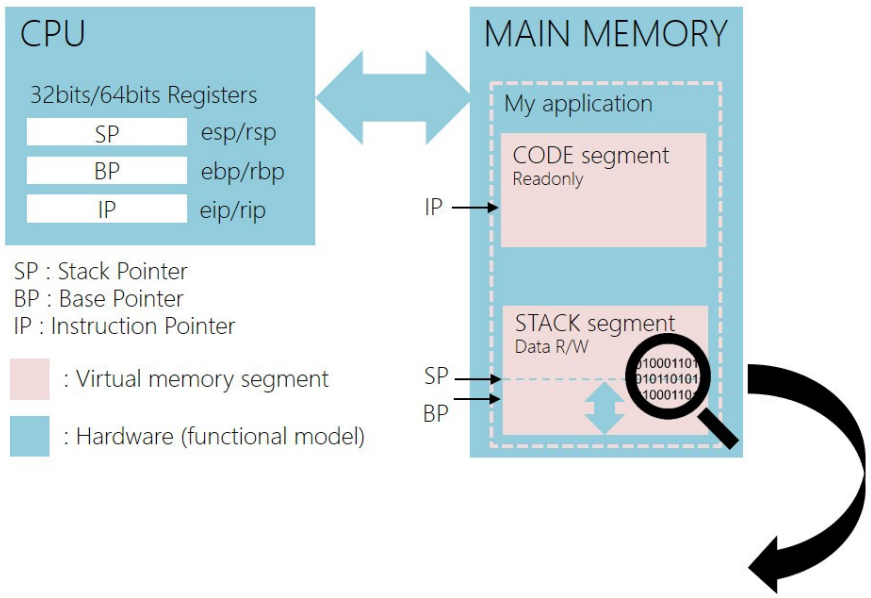


Les allocations de ressources mémoire réalisés dynamiquement à l'exécution durant la manipulation de variables locales se font sur le segment de pile ou *stack*. Ce segment est présent en mémoire principale durant l'exécution d'un programme. *Un segment est une zone logique contigu virtuelle allouée en mémoire principale par le noyau du système avant l'exécution d'un programme*. Rappelons que pour des raisons de robustesse (spatialité des usages), les variables locales sont les variables les plus couramment utilisées dans le monde du logiciel. Contrairement aux variables globales à n'utiliser qu'en cas d'absolue nécessité (ressources partagées).

En langage C, comme dans beaucoup d'autres langages (C++, Java, D, etc), le point d'entrée d'une application est la fonction *main*. De même, si nous réalisons des appels de fonctions imbriqués depuis le *main* (cf. exemple ci-dessus) et si nous souhaitons revenir à la fonction principale de façon conventionnelle, nous aurons à quitter dans l'ordre d'appel toutes les fonctions respectivement appelées. Les appels de fonctions sont gérés telle une pile de papier (LIFO, Last In First Out) et il en va de même pour les variables locales. Les variables locales à une fonction seront allouées automatiquement en entrée de fonction à l'exécution. A l'usage, toutes les variables locales seront stockées dans le segment de pile, qui sera toujours de taille fixe. Chaque application possède sa propre pile. Il existe au minimum autant de piles que de programmes chargés et démarrés (processus) en mémoire principale par le noyau Linux. Par défaut sur ordinateur sous Linux, chaque pile applicative offre 8Mo potentiel d'espace mémoire de stockage. Le segment de code, contenant le code binaire exécutable du programme, est donc spatialement séparé du segment de pile (cf. schéma ci-dessous), contenant uniquement des données accessibles en lecture et écriture. Ce cloisonnement spatial est nécessaire à la robustesse globale de l'ordinateur et est conjointement réalisé et supervisé par l'unité matérielle de pagination (PMMU ou Paged Memory Management Unit) qui est elle même exploitée par le noyau du système.



Synthèse sur la gestion du segment de pile

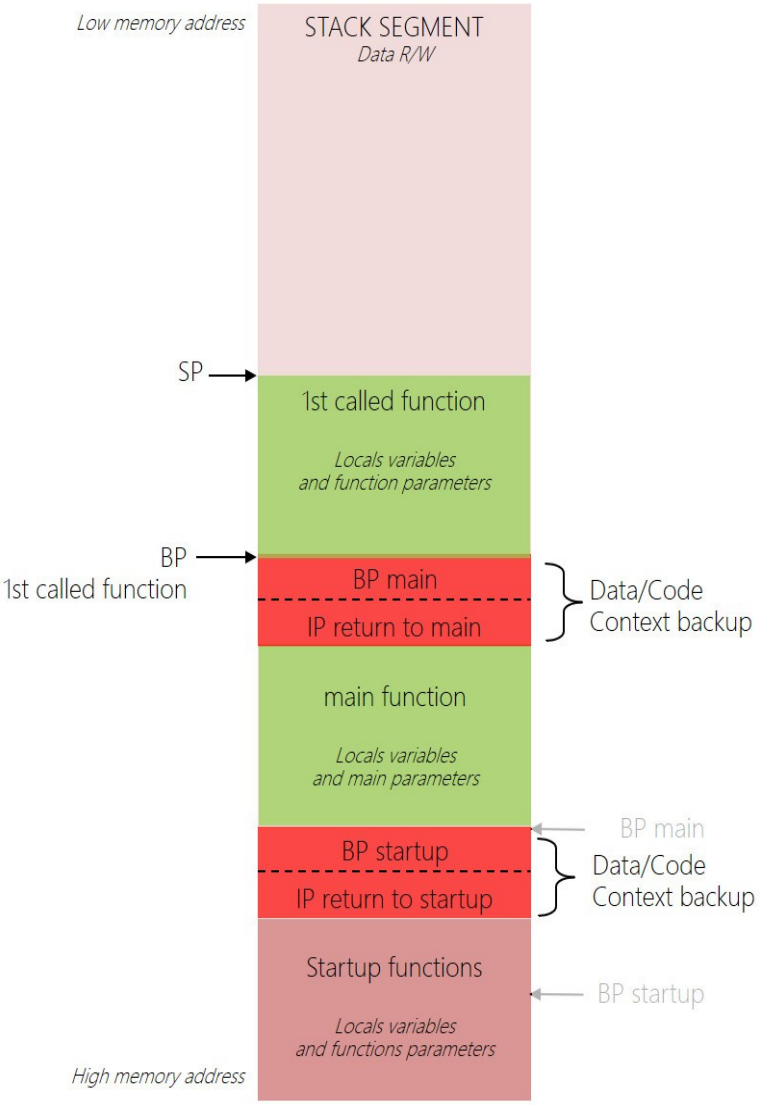


Les processus de gestion des variables locales et des paramètres de fonction (eux même des variables locales paramétrées) sont conjointement réalisés à la compilation par les outils de développement et exploités à l'exécution par la machine.

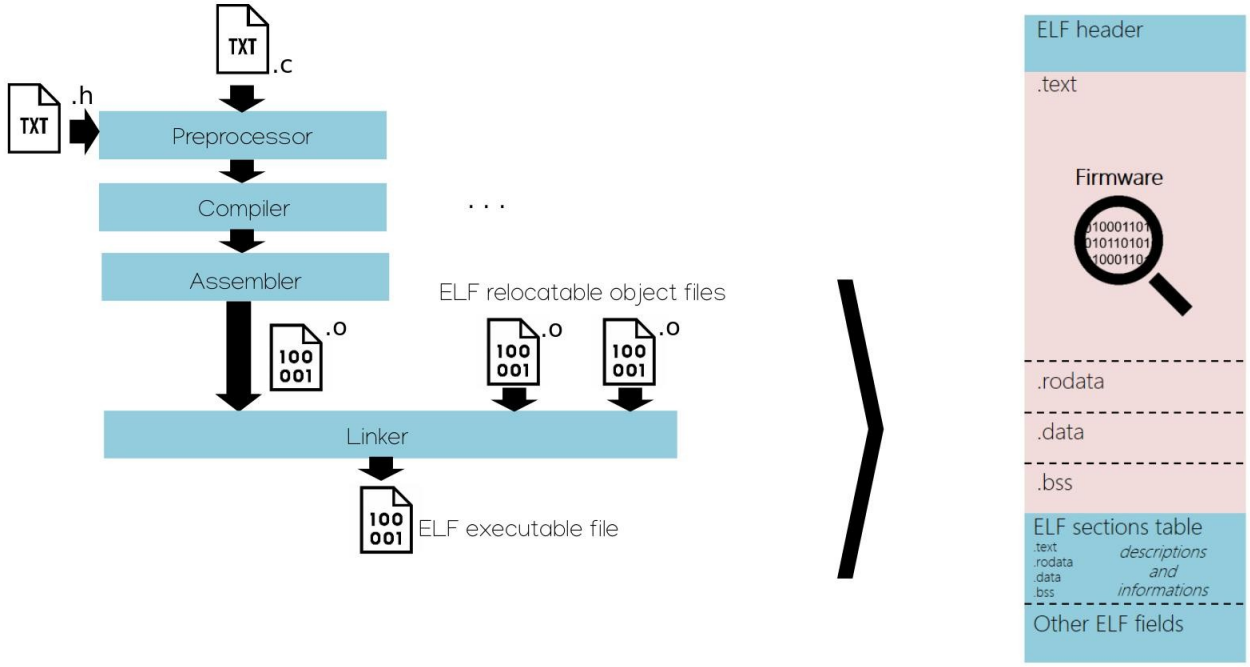
Variables locales et paramètres de fonction sont allouées dynamiquement à l'exécution suite à l'appel et durant l'entrée dans le code d'une fonction. Les premières lignes de code implémentant une fonction servent donc à sauvegarder le contexte d'exécution de la fonction appelante (BP pour les données et IP pour le code) puis à allouer sur la pile les ressources mémoire données nécessaires à son exécution.

Une fois dans une fonction, les pointeurs BP et SP encapsulent le plus souvent la zone mémoire sur la pile comprenant les variables locales et paramètres propres à cette même fonction. Tant que nous restons dans cette fonction, BP ne sera pas modifié. De ce fait, toutes les variables locales et paramètres de fonction possèdent une adresse mémoire relative au pointeur BP. Un seul et même registre (EBP/RBP 32bits/64bits x86/x64) permet donc indirectement d'adresser un nombre quelconque de variables locales.

Quant à lui, le segment de pile possédera toujours une taille fixe. Sur ordinateur, rappelons qu'un segment est une zone virtuelle contigu allouée en mémoire principale par le noyau du système à l'exécution. Ce segment n'admet aucune existence sur les médias de stockage de masse (HDD, SSD, etc).



5. ALLOCATIONS STATIQUES ET FICHIER ELF

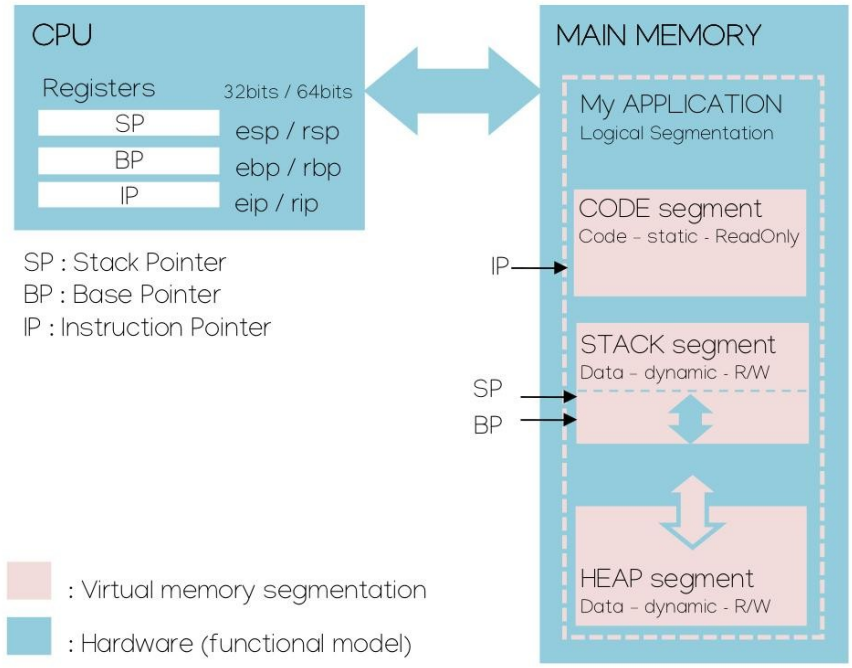


Les allocations statiques représentent toutes les allocations de ressources mémoire réalisées à la compilation et donc présentes dans le fichier binaire ELF de sortie. Les variables statiques admettent donc une existence sur un media de stockage de masse (HDD, SSD, etc) avant même l'exécution d'un programme en mémoire principale. Les références symboliques, ou adresses logiques, de chaque fonction et variable statique sont donc inchangées (statiques) durant la totalité de la vie d'un programme binaire sans nouvelle compilation et édition des liens du projet logiciel source. Contrairement aux variables locales (allocations automatiques ou dynamiques sur le segment de pile) et allocations dynamiques sur le segment de tas, pour lesquelles les allocations de ressources mémoire sont réalisées à l'exécution du programme dans des segments logiques dédiés.

Une *section* est une zone logique du *firmware*. Un *Firrmware* peut être également nommé micrologiciel en Français. Le *firmware* représente dans cet enseignement le code (forcément statique) et les données statiques binaires strictement utiles au fonctionnement d'un programme. Le *firmware* est quant à lui encapsulé dans un cartouche au format ELF (en-tête, table des sections, en-tête du programme, etc) proposant au noyau du système (Linux) une description et des informations sur le micrologiciel afin d'aider à sa manipulation (préparation des segments mémoire, association de propriétés et privilèges, etc). Sauf si un développeur créé explicitement de nouvelles sections en spécifiant des attributs spécifiques durant une déclaration d'une variable (https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html), une application pourra comporter au plus 4 sections applicatives par défaut afin de gérer l'ensemble des besoins standards en allocations statiques de ressources (les noms suivants sont hérités d'Unix) :

- .text (CODE - Read Only - executable) : section encapsulant le code binaire statique du programme
- .rodata (DATA - Read Only – not executable) : section encapsulant les données statiques accessibles en lecture seule
- .data (DATA - Read/Write – not executable) : section encapsulant les données statiques initialisées accessibles en lecture et écriture
- .bss (DATA - Read/Write – not executable) : section encapsulant les données statiques non-initialisées accessibles en lecture et écriture

6. ALLOCATIONS DYNAMIQUES ET SEGMENT DE TAS

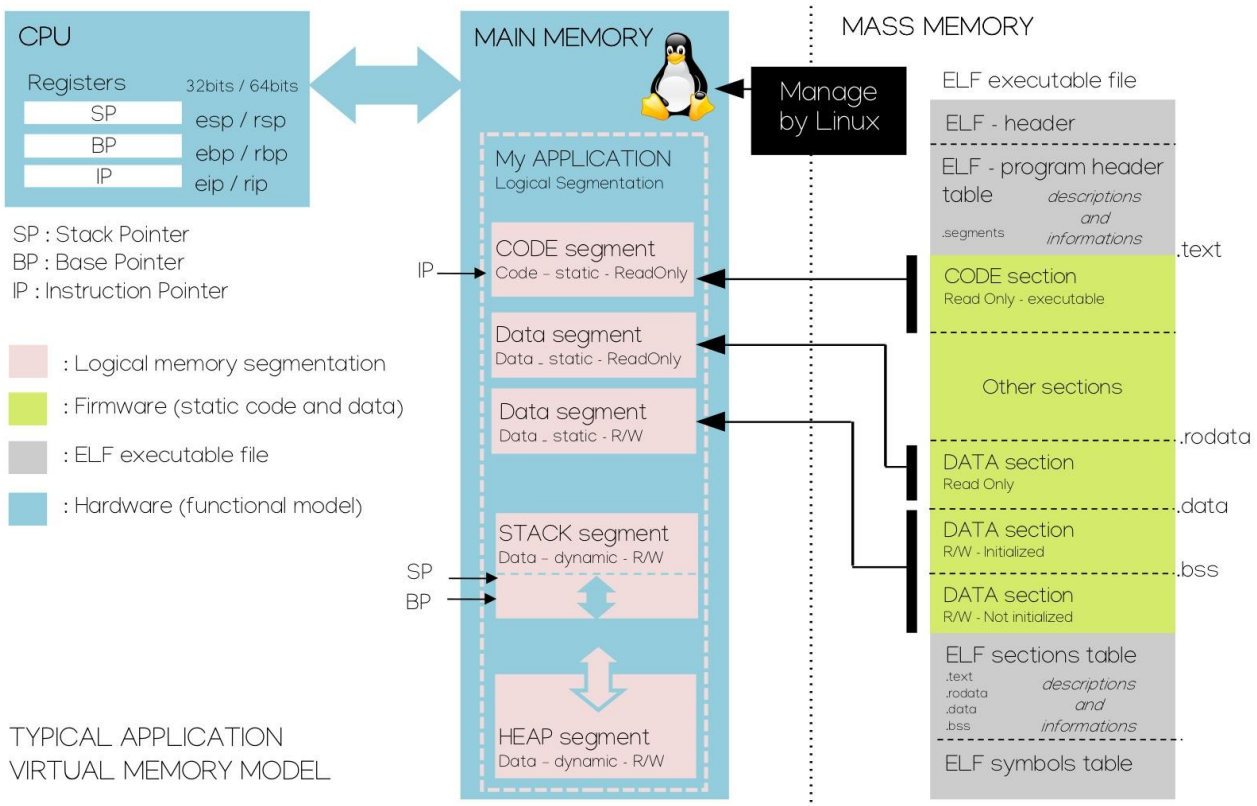


Le tas (ou *heap*) est l'un des deux segments mémoire utilisé pour les allocations dynamiques durant l'exécution d'un programme. Nous avons découvert le premier dans les exercices précédents à travers l'analyse de la gestion des variables locales et des allocations dynamiques sur la pile aussi nommées allocations automatiques. Ces allocations sont nommées ainsi car l'allocation mémoire est réalisée automatiquement à l'exécution durant l'entrée dans le code d'une fonction.

Contrairement à la pile ou stack qui possède une taille fixe, le tas (cf. schéma ci-dessus) possède une taille extensible pouvant aller jusqu'aux limites physiques des ressources de stockage en mémoire principale de la machine (mémoire principale DDR SDRAM + potentiellement SWAP système sur média de stockage de masse HDD/SSD/etc). Les allocations dynamiques sur le tas sont exécutées explicitement à la demande du programme sous forme de requêtes envoyées au noyau du système (Linux). Ces requêtes d'allocations mémoire se font par appels de la fonction *malloc* (memory allocation) ou de ses variantes (*calloc*, *realloc* et *aligned_alloc*). Tant que l'application est active en mémoire principale, l'espace demandé restera alloué. Une fois la ressource mémoire utilisée, bien penser à libérer les allocations précédentes avec appel de la fonction *free* (à la responsabilité du développeur). Ceci permet de libérer de l'espace pour les autres applications actives sur la machine. Le phénomène de zones mémoires précédemment allouées non libérées se nomme fuites mémoire et reste des erreurs assez courantes dans le monde du logiciel.

Il est important de noter que la fonction *free* est probablement l'une des fonctions les plus risquées à l'usage du langage C, notamment pour une application dans le domaine des systèmes embarqués sur processeur MCU, DSP, MPPA, etc (sans MMU). En effet, allouer successivement des ressources mémoire dynamiquement sur le tas puis libérer certaines de ces zones amène une fragmentation du tas (zones mortes non utilisées). Éviter d'utiliser la fonction *free* sur un processeur ne possédant pas de MMU (Memory Management Unit) et ne pouvant garantir au noyau du système d'exploitation une virtualisation de la gestion mémoire. Sans quoi, la fragmentation précédemment citée sera inévitable et conduira vers un comportement erratique puis le bug de l'application.

Synthèse globale sur les stratégies d'allocations



Le schéma ci-dessus rappel et synthétise une grande partie des nombreux points abordés dans cet enseignement et cette trame de TP. Maintenant vous le savez, le superviseur de la machine à la gestion des ressources matérielles est le noyau du système d'exploitation (Linux, GNU Hurd, XNU, etc). Il est notamment le gestionnaire de la mémoire.

Compilation et édition des liens

Après développement d'un programme logiciel (*software*), la chaîne de compilation (GCC, Clang, ICC, etc) est chargée de traduire le programme d'un langage source (C, C++, D, etc) en langage machine binaire (x86, x64, ARM, MIPS, etc). Le format de fichier standard d'encapsulation de *firmware* sur système Unix-like est le format ELF (COFF, PE sous windows, etc). Le *firmware* est découpé en sections, des zones logiques séparant l'information (code et donnée) en partie de même natures et propriétés (code, donnée, lecture seule, lecture/écriture, exécutable, etc).

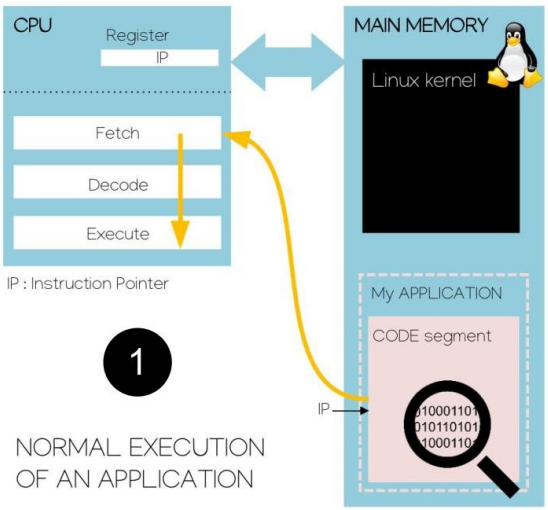
Allocation des segments et chargement en mémoire par le noyau

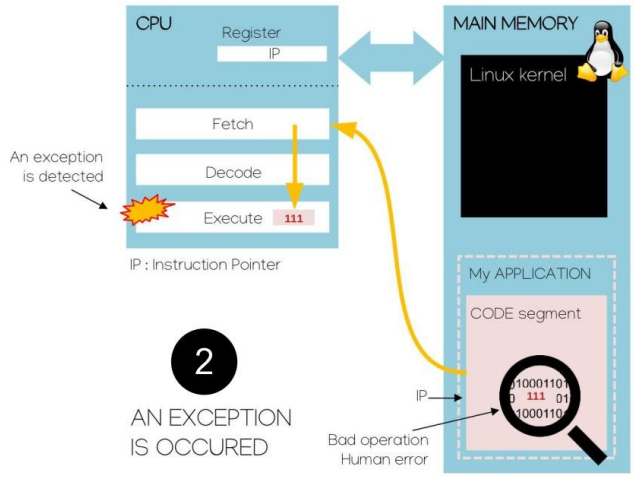
Si l'utilisateur demande l'exécution d'un programme, le noyau analyse alors le contenu du fichier exécutable (application à exécuter) grâce aux différentes en-têtes et tables du format ELF. Il alloue alors des segments mémoire logiques contigus (virtualisation) ne pouvant se chevaucher (Virtual Memory Area ou VMA sous Linux) de tailles et propriétés adaptées en mémoire principale. Une fois les allocations réalisées, par copie il charge du média de stockage de masse (HDD, SSD, MMC, etc) les sections statiques du *firmware* vers les segments associés en mémoire principale (DDRx SDRAM). Une fois cette opération faite, il donne la main à l'application en commençant par exécuter le code des fonctions de *startup*. L'application pourra ensuite s'exécuter dans le respect des limites des segments mémoire alloués par le système. Sinon *Segmentation fault (core dumped)* sera retourné par le système et l'application sera retiré (*kill*) de la mémoire principale.

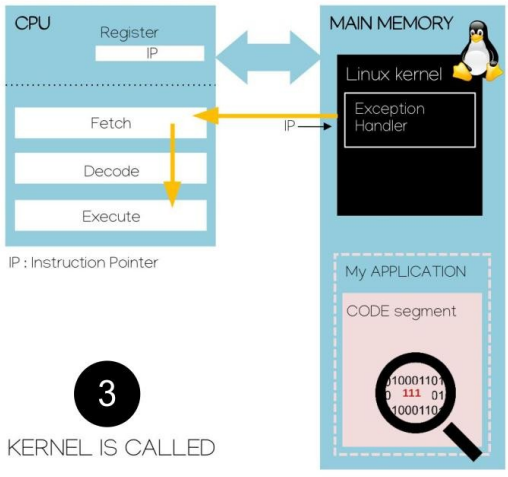
Allocations mémoire et segments associés

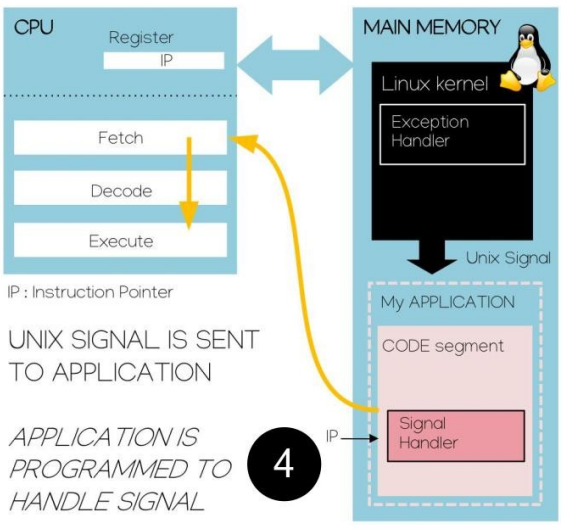
En résumé, il existe donc 3 types d'allocation de ressource mémoire sur les langages compilés, les allocations statiques, les allocations dynamiques sur la pile (ou automatiques) et les allocation dynamiques sur tas. Chaque type d'allocation possède un segment mémoire dédié.

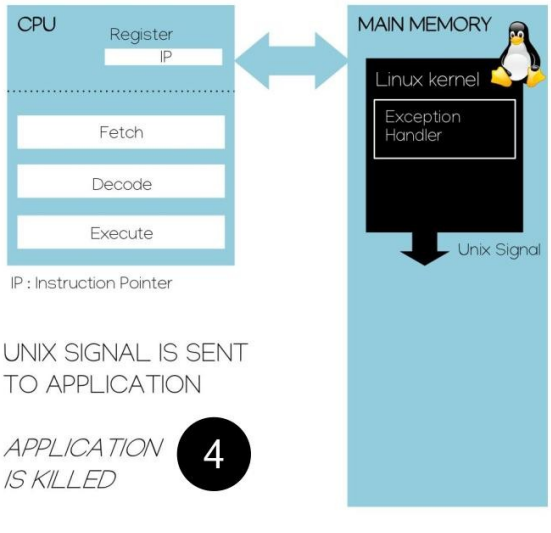
7. EXCEPTIONS MATÉRIELLES ET SIGNAUX UNIX











Une exception matérielle est un événement matériel synchrone généré par le CPU voire la MMU. Nous parlons d'événement synchrone au regard du fonctionnement d'un CPU dont les traitements restent synchronisés sur une référence d'horloge et non au regard de la probabilité d'occurrence. Une exception peut interrompre l'exécution d'une application à tout moment. Ces événements sont relevés par le CPU lorsque celui-ci détecte une condition prédéfinie et documentée non conventionnelle faisant exception durant l'exécution d'une instruction (violation de privilège, division flottante par zéro, accès illégal en mémoire, etc). Contrairement aux *interruptions* (générées par les périphériques) provoquées par des causes externes au programme (asynchrone), les *exceptions* sont provoquées par des causes internes au programme (synchrone). Toute exception est une opération connue ne devant généralement en aucun cas arrivé. Elles sont le plus souvent le fruit d'une erreur de programmation. Détaillons la séquence graphique présentée sur la page précédente :

1. L'application s'exécute normalement sur le CPU courant. Le noyau est au repos en attente d'un événement requérant son travail

2. Le programme en cours d'exécution implémente une opération binaire erronée. Durant son exécution par le CPU, l'instruction génère une exception au fonctionnement normal du CPU. Le pointeur d'instruction IP est alors redirigé (par lecture et exécution d'un tableau de vecteurs) vers une fonction noyau dédié au traitement des exceptions. Le noyau du système s'exécute alors sur le CPU courant

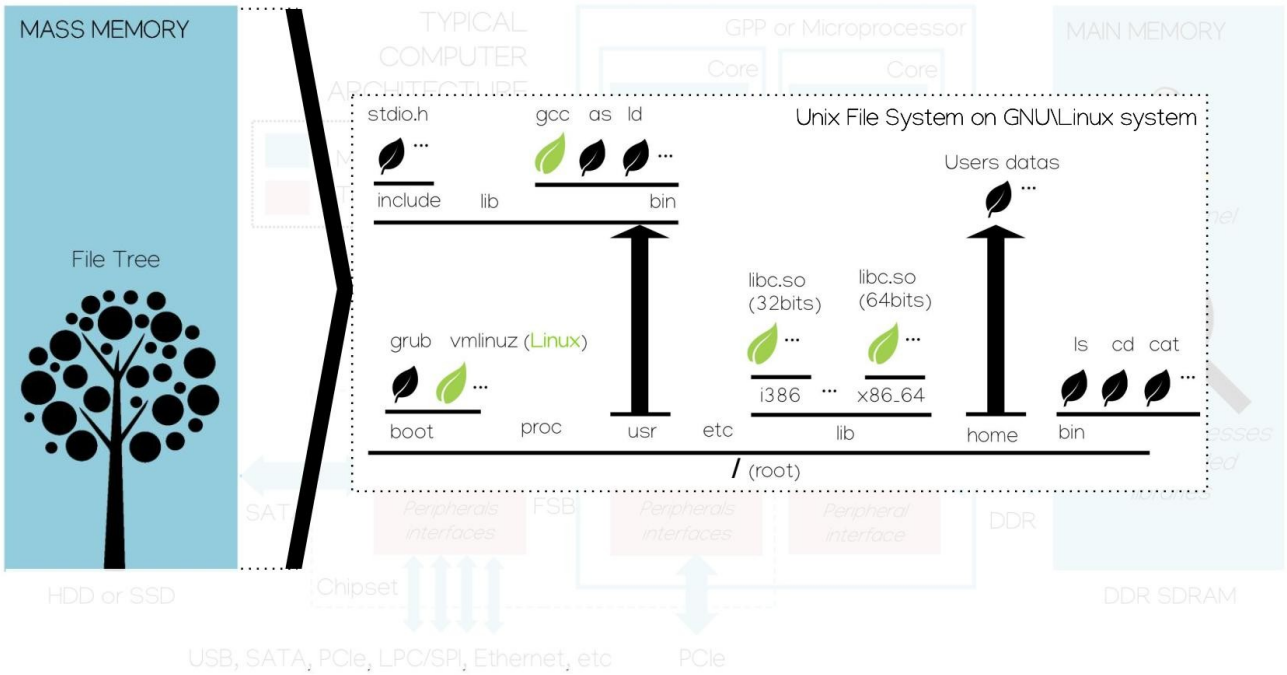
3. Le CPU vient donc de stopper l'exécution de l'application en cours et d'appeler une procédure enfouie du système. Une sauvegarde du contexte CPU (registres de travail internes) est également réalisée et pourra être accessible depuis l'application en espace utilisateur. La procédure de gestion des exceptions est implémentée par la fonction *do_page_fault* dans le cas de Linux (présente dans le fichier */arch/<cpu_architecture>/mm/fault.c* du système de fichier du kernel Linux, https://www.kernel.org/). La fonction de gestion des exceptions (ou *exception handler*) est chargée de relever le type de défaut et de générer, si l'exception le permet, un signal logiciel Unix à destination du processus (application) ayant généré le défaut.

4. (schéma de droite) Si le processus ne traite pas le signal Unix, il est alors tué par le noyau du système qui assure la libération de toutes les ressources mémoire associées. Aucune autre application n'est impactée. Ceci assure un cloisonnement des défauts et la stabilité du système.

   ou

   (schéma de gauche) Si le processus traite le signal Unix (code spécifique intégré à l'application), l'application peut alors tenter d'acquitter le défaut (restaurer un contexte CPU viable) ou tenter une solution de contournement (redirection vers un autre code de l'application viable, redémarrage ou mode dégradé de l'application, etc). Si cela est possible, un message d'erreur ou un fichier de log pourra être sauvé voire envoyé aux équipes de développement.

9. MÉMOIRE DE MASSE ET SYSTÈME DE FICHIERS



Les médias physiques de stockage de masse offre une stratégie de représentation et de classification de l'information sous forme d'arborescence de fichiers. Il est à noter qu'une mémoire physique (espace de stockage) est toujours pilotée par un périphérique matériel d'interface nommé contrôleur. Celui-ci est chargé d'écouter les requêtes (opération de lecture ou d'écriture, adresse, nombre d'octets, etc) et de répondre à celle-ci en délivrant ou en stockant l'information demandée. Rappelons les 3 familles de mémoires rencontrées sur ordinateur. :

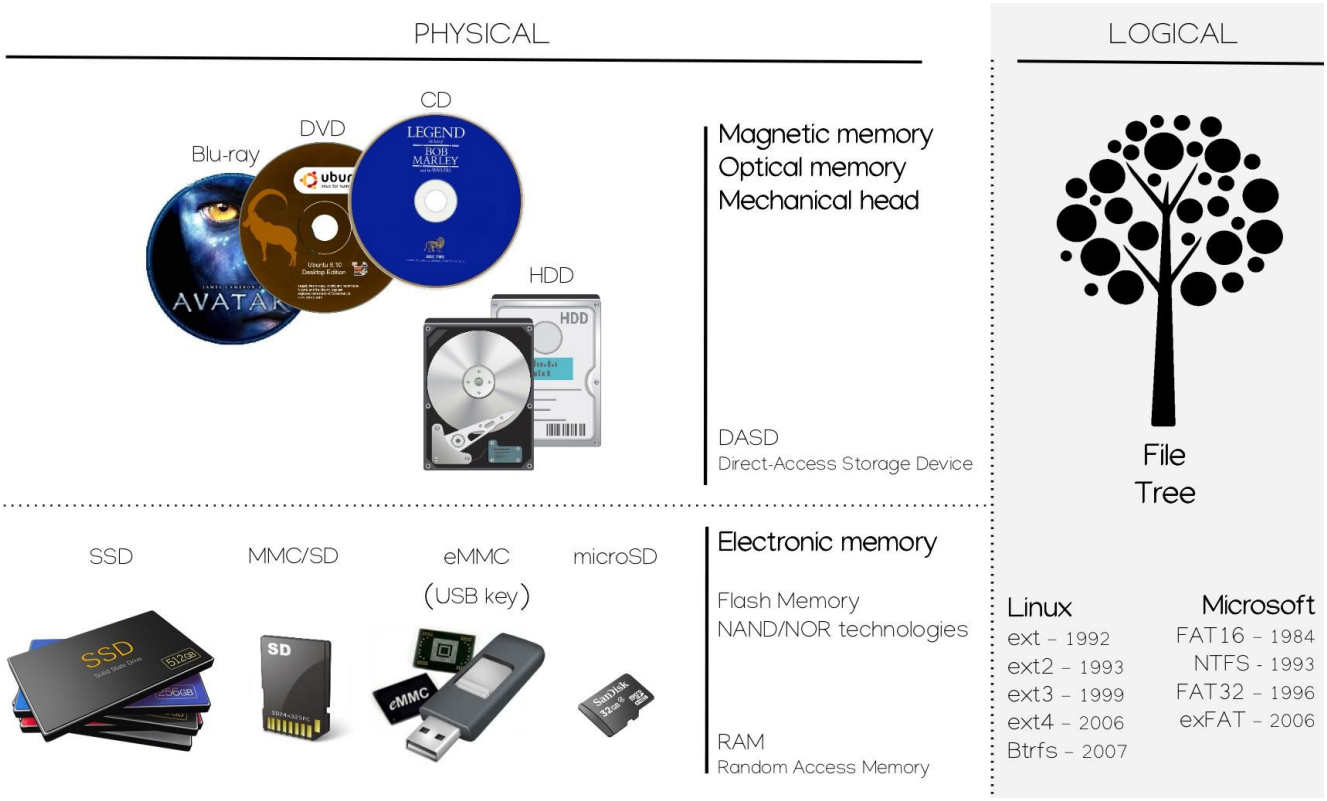
- Mémoire cache : *mémoire adressable par association (associative) de technologie SRAM*. A chaque octet copié en cache (niveaux L1\L2\L3) depuis la mémoire principale est associé un emplacement (par indexage) dans une ligne de cache.
- Mémoire principale : *mémoire adressable par octet de technologie DRAM*. Chaque octet possède une adresse mémoire unique typiquement au format hexadécimal (32bits, 64bits, etc – *cat /proc/cpuinfo*). Dans les langages de programmation offrant une abstraction aux langages machines, les adresses en mémoire principale sont le plus souvent nommées pointeurs voire références. Ces mémoires sont souvent nommées mémoires vives.

- Mémoire de masse : *mémoire adressable par chemin dans une arborescence de fichiers de technologies HDD, SSD, MMC, etc*. Cette mémoire stock l'information en implémentant les concepts de classification de fichiers dans des répertoires. Un fichier possède donc une adresse nommée chemin (*path*) dans une arborescence dont la base est nommée racine (/ ou *root* sur système Unix-like).

Les systèmes Unix-like, comme les systèmes d'exploitation GNU\Linux, héritent pour la plupart d'une arborescence de fichiers Unix (/boot, /proc, /usr, /lib, /bin, /home, etc). Rappelons que le système original Unix se voulait d'une philosophie simple et minimaliste. Sous Unix, tout est fichier avec une gestion élémentaire (open, read, write et close). Le système se veut également implémenter un modèle à 3 couches (kernel, shell et utilities). Nous pouvons observer dans l'illustration ci-dessus quelques un des programmes et bibliothèques les plus connus du système, notamment :

- Linux : firmware brut de qqMo dans */boot* . Vous pouvez vérifier et constater que Linux n'est pas un fichier ELF, mais sait exploiter des fichiers ELF une fois chargé et exécuté en mémoire principale.
- GCC : Chaîne de compilation ayant servie à généré le système lui-même dans */usr/bin*
- LIBC : Bibliothèques standards du langage C (32bits et 64bits) dans */lib*

Dans les systèmes numériques de traitement de l'information actuels, une mémoire de masse est une mémoire persistante dite non volatile (persistance de l'information sans apport d'énergie électrique). Une mémoire de masse est accessible en lecture voire en écriture. Elle sont souvent de plus grandes capacités que les mémoires vives mais restent plus lentes (mémoires vives de technologies volatiles SRAM ou DRAM). Hors communication extérieure au système (Internet, réseau Ethernet, clé USB, etc), une mémoire de masse stocke et représente l'ensemble des savoirs et savoirs-faire statiques d'une machine !



Plusieurs technologies de mémoire de masse sont actuellement en usage (HDD, SSD, MMC SD, MMC microSD, CD, DVD, Blu-ray, etc), tout comme certaines sont maintenant tombées en désuétude (K7 audio, disquette, VHS, VHS-c, carte perforée, tore magnétique, etc). Chaque technologie offre son lot hérité d'avantages, d'inconvénients, de compromis et se trouve adapté à des besoins et des marchés ciblés. Les mémoires de masse actuellement les plus rapides sur le marché, mais toujours les plus coûteuses pour de grandes capacités de stockage, sont les mémoires électroniques de technologie Flash NOR ou NAND (SSD, MMC SD, MMC microSD, eMMC par exemple sur clé USB, etc)

Indépendamment des technologies physiques de stockage utilisées, la représentation logique de l'information offerte par le système se base sur les concepts de partitions (zones logiques contiguës de la mémoire physique) et d'adressage de l'information par arborescence de fichiers. Plusieurs technologies de systèmes de fichiers (FS ou File System) sont en usage à notre époque. Btrfs, ext4, SquashFS, etc sous Linux. NTFS, FAT32, VFAT, etc sous Windows. Bien d'autres technologies existent, notamment pour d'autres systèmes d'exploitation (BSD, Mac OS X, etc). Chaque technologie offre sont lot d'avantages et d'inconvénients. Prenons l'exemple de la technologie FAT32 encore très utilisée à notre époque (systèmes embarqués, clé USB, etc) :

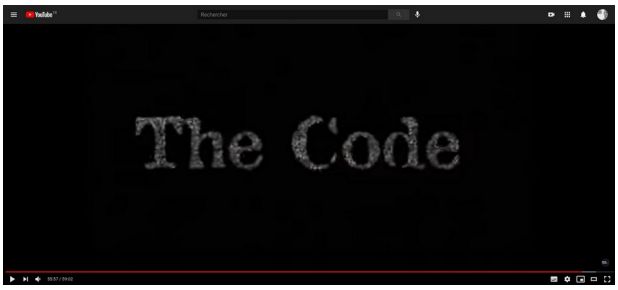
- Taille maximale d'un fichier 4Go
- Taille maximale théorique d'une partition 16To
- Nombre maximal de fichiers 268M
- Nombre maximal de fichiers par répertoire 65534
- etc

Quelques ressources internet

Voici quelques ressources en ligne pouvant vous aider à une meilleure contextualisation des machines, des outils de développement, des langages et des systèmes utilisées à notre époque. Bons visionnages et lectures dans ces voyages dans notre histoire …

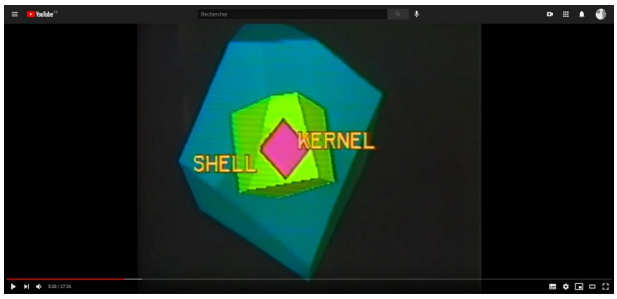
https://www.youtube.com/watch?v=dcN9QXxmRqk

Noyau Linux et système d'exploitation GNU – Nom de code Linux



https://www.youtube.com/watch?v=79_IMeks4wY

Système d'exploitation Unix – AT&T Archives: The Unix Operating System



https://www.youtube.com/watch?v=tc4ROCJYbm0&t=1337s

Sociétés privées Apple, Microsoft et IBM – Les cinglés de l'informatique

https://www.youtube.com/watch?v=dOakyAhqiVY&t=2140s

https://www.youtube.com/watch?v=zywPwbQqshY&t=2364s

https://www.youtube.com/watch?v=m9QUs2Nf3yA