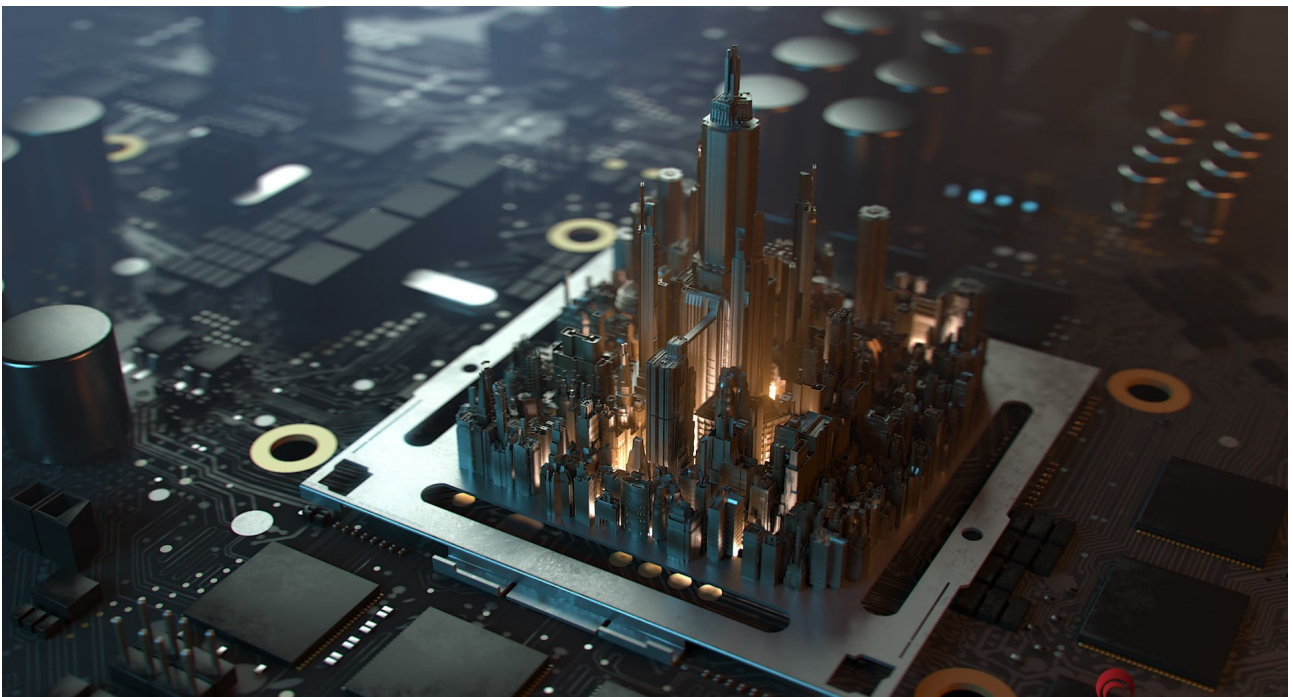


Architecture des Ordinateurs

Support de Travaux Pratiques



Contacts

Dimitri Boudier – Responsable du module – CM, TP

dimitri.boudier@ensicaen.fr

Philippe Lefebvre – TP

philippe.lefebvre@ensicaen.fr

Ressources

Toutes les ressources (supports CM, TP et outils) sont sur la page Moodle du cours :

<https://foad.ensicaen.fr/course/view.php?id=213>



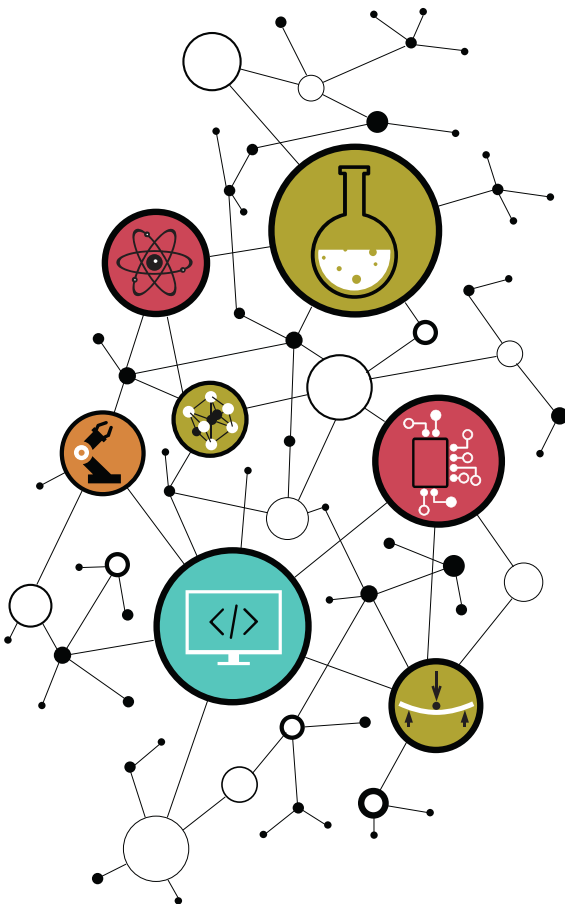
Except where otherwise noted, this work is licensed under
<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Sommaire

Contacts.....	2
Ressources.....	2
Partie 1 Prélude.....	7
I. L'enseignement, en bref.....	8
II. Le langage C.....	9
III. Objectifs pédagogiques.....	10
IV. Outils et environnement de développement.....	12
V. Découpage temporel.....	13
Partie 2 Chaîne de compilation.....	15
I. Arborescence.....	16
II. GCC Toolchain.....	17
III. Compilation.....	18
IV. Édition des liens.....	23
V. Synthèse sur l'exercice.....	29
VI. Librairie statique.....	30
VII. Make et Makefile.....	32
Partie 3 Allocation statique et fichier ELF.....	35
I. Compilation et allocation statique.....	36
II. Variables globales.....	37
III. Variables locales statiques.....	40
IV. Chaînes de caractères.....	41
V. Synthèse.....	42
Partie 4 Assembleur Intel x86.....	45
I. L'assembleur sous architecture x86.....	46
II. Instructions arithmétiques et logiques.....	49
III. GNU Debugger.....	50
IV. Fonction de conversion entier vers ASCII.....	52
V. Fonction d'affichage printf.....	53
VI. Suite de Fibonacci.....	54
Partie 5 Allocation automatique et segment de pile.....	57
I. Segments de code et de pile.....	58
II. Fonction main.....	60
III. Variables locales initialisées.....	62
IV. Variables locales non-initialisées.....	65
V. Appel et paramètres de fonction.....	66
VI. Fonction inline et optimisation.....	69
VII. Limites de la pile.....	72
VIII. Synthèse.....	73

Partie 6 Allocation dynamique et segment de tas.....	75
I. Segments de code, de pile et de tas.....	76
II. Process en exécution.....	77
III. Fuites mémoire.....	78
IV. Limites du tas.....	79
V. Synthèse globale sur les stratégies d'allocations.....	80
Partie 7 Mémoires cache.....	83
I. Principes de localité.....	84
II. Cache hit et cache miss.....	85
III. Cohérence de cache et false sharing.....	86
IV. Synthèse.....	87
Partie 8 Exceptions matérielles et Signaux UNIX.....	89
I. Exception vs Signal.....	90
II. Lecture seule.....	92
III. Pointeur nul.....	93
IV. Signal Unix.....	94
Partie 9 Hacking.....	97
I. Les bidouilleurs.....	98
II. Appels système.....	99
III. Exécution d'un shellcode sur la pile.....	103
IV. Exploitation de vulnérabilité.....	105
V. White-hat.....	107
Partie 10 Mémoire de masse et stockage des fichiers.....	109
I. Du support physique à la représentation logique.....	110
II. Table des partitions.....	112
III. Système de fichiers.....	114
IV. Point de montage.....	115
V. Outil graphique GParted.....	116
VI. Kali sur clé USB bootable.....	118

PARTIE 1 PRÉLUDE



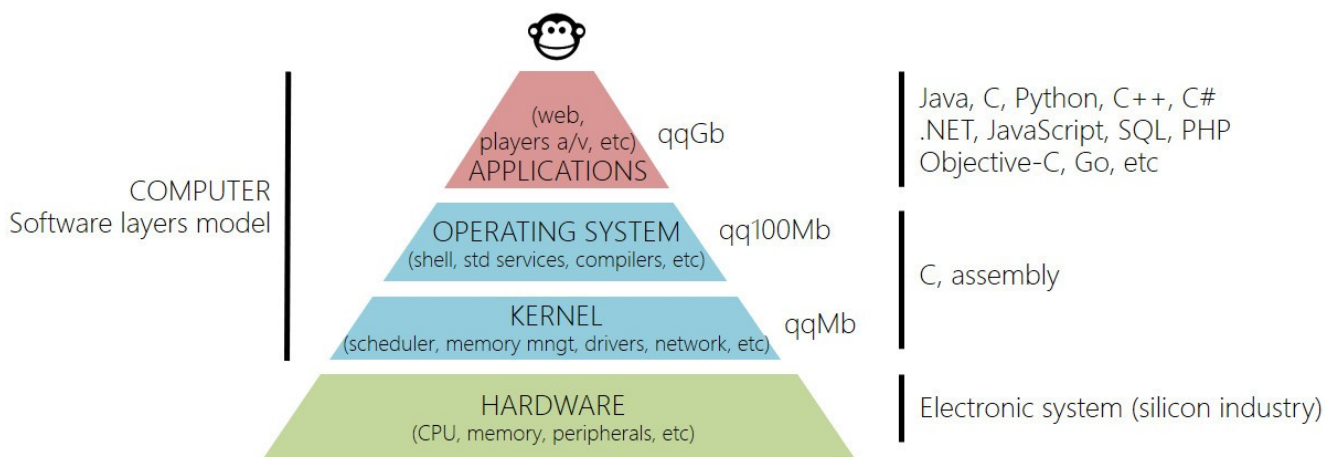
I. L'enseignement, en bref

Pour résumer ces exercices de travaux pratiques en une phrase, c'est de comprendre comment un programme initialement écrit en langage C fini par s'exécuter sur un ordinateur.

« 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 co-développeur d'Unix

Nous travaillons donc ici sur la partie applicative, dans ce qu'on appelle le **user-space**. Il s'agit de l'espace de la machine réservée à l'utilisateur (comme son nom l'indique), donc ce qui est laissé à disposition de n'importe quel programme. Par opposition et dans les couches logicielles plus basses, on trouve le **kernel-space** dans lequel le noyau du système d'exploitation travaille. C'est lui qui gère notamment les accès aux matériels de la machine. Cette deuxième partie sera étudiée au second semestre, dans l'**enseignement de Systèmes d'Exploitation et Réseaux**. Avec **Architecture des Ordinateurs**, ces deux enseignements complémentaires permettent d'appréhender une grande partie du fonctionnement des ordinateurs.



D'un point de vue **logiciel**, nous travaillerons avec les technologies et les standards les plus répandus à notre époque, dans le monde des couches basses des systèmes numériques de traitement de l'information : langage C (1972) ; système d'exploitation GNU (1983) ; chaîne de compilation GCC (C ANSI, 1986) ; interface standard POSIX (1988) ; noyau Linux (1991).

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.

D'un point de vue **matériel**, les technologies 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 (32-bit/64-bit). 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).

II. Le langage C

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 ...

Le langage C fut historiquement développé par Dennis Ritchie (*AT&T, Bell Labs*) au début des années 70 afin de ré-écrire **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, ...) 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, ...). Même à notre époque, le C continue d'évoluer (normes C ANSI/C89 en 1989, C90, C95, C99, C11, C17 et dernièrement le C23).

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). Il s'agit d'un langage "épuré" (comparé à d'autres comme le C++), versatile et permissif (la compilation et l'exécution tolèrent beaucoup de marge de manœuvre) offrant un contrôle important sur la machine, notamment au regard de la gestion mémoire (débordements, fuites, traces, ...). 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, ...). Pour un développeur système, la complexité ne doit pas résider dans le langage mais dans le système !

À notre époque, il reste toujours parmi les langages les plus populaires au monde¹. 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, ...). 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, Android, MacOS X, iOS, Windows, ...) et noyaux (Linux, XNU, Darwin, Mach, NT, ...), la liste est très très très longue ...

¹ <https://www.tiobe.com/tiobe-index/>

III. Objectifs pédagogiques

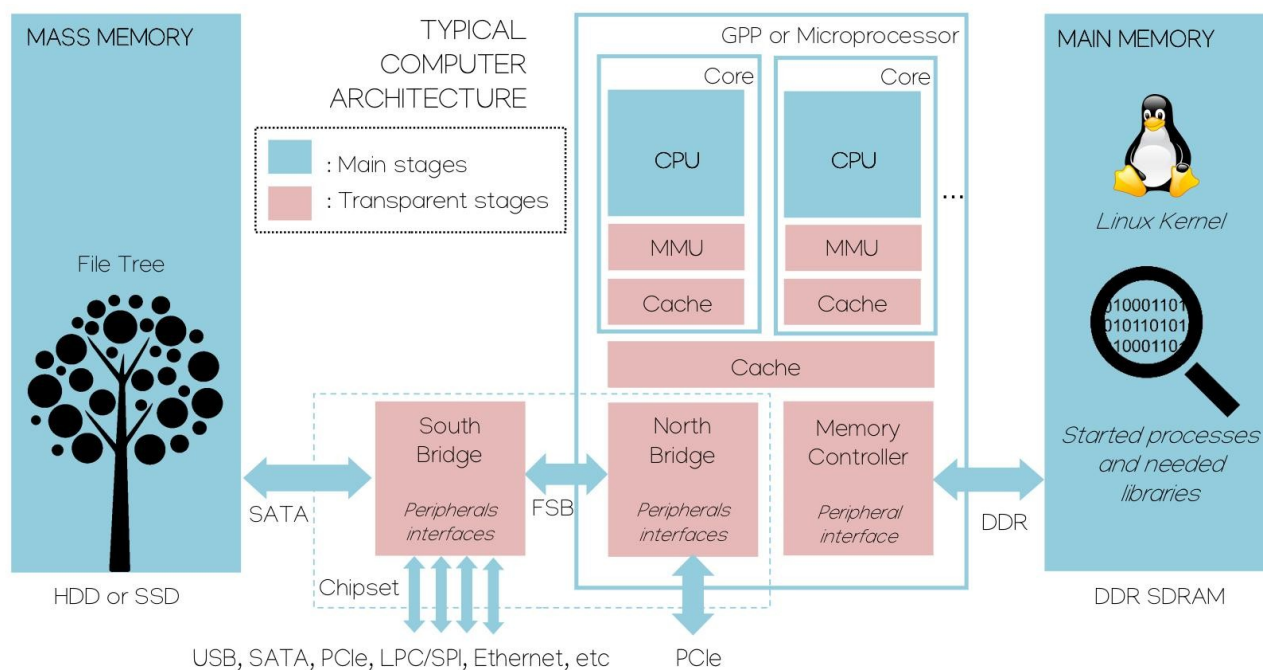
Cette trame d'enseignement ne cherche pas à vous ré-apprendre un langage de programmation. Il s'agit d'une trame d'analyse de programmes C élémentaires ayant pour objectif terminal d'aboutir à une meilleure représentation et compréhension du fonctionnement de la machine (ordinateur), de la compilation 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 très rapidement, car nécessitant une meilleure compréhension des couches basses du système. Prenons les exemples des qualificatifs de type et de fonction spéciaux, des classes de stockage, ... : `register`, `volatile`, `inline`, `static`, `const`, `restrict`, ...

III.1. Représentation physique de l'architecture matérielle

Les systèmes matériels actuels de traitement de l'information, tel qu'un ordinateur, sont devenus d'une grande complexité architecturale et technologique. À 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 contre-performances de vos programmes (MMU et mémoires cache dans notre cas).

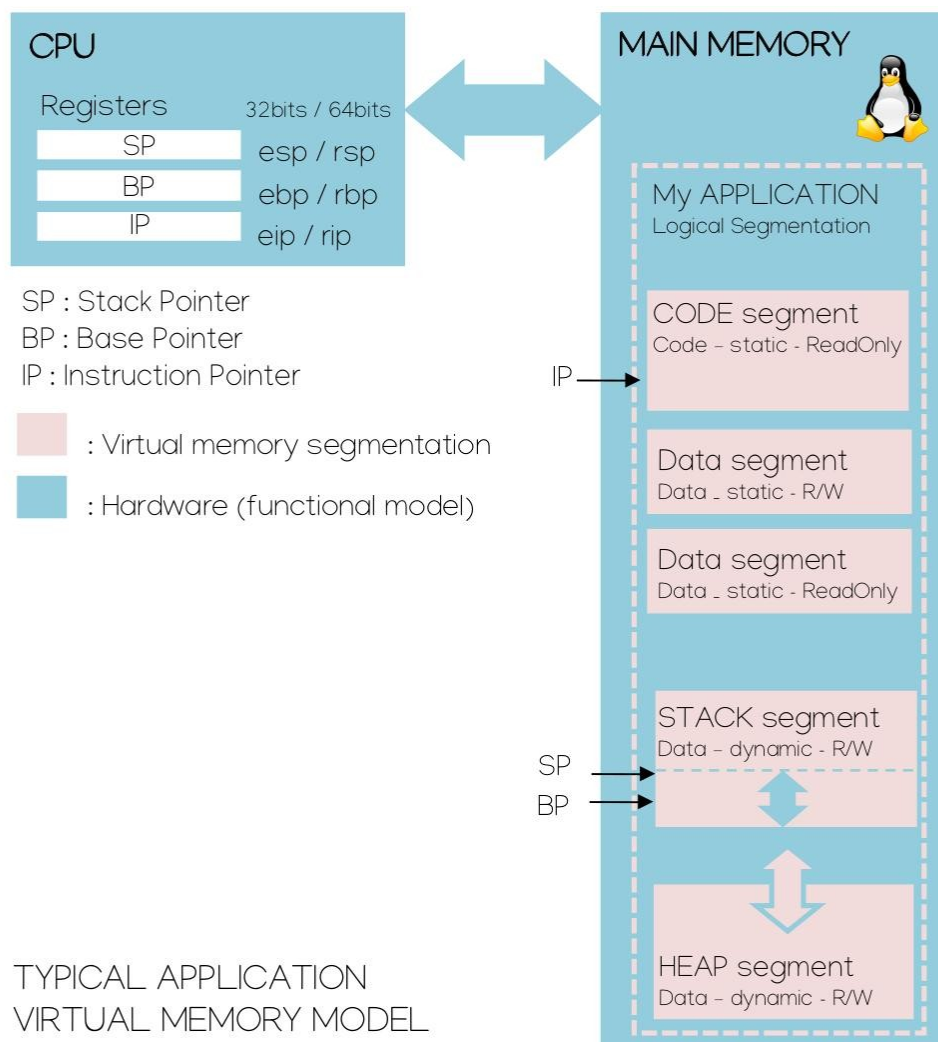


III.2. Représentation logique de l'architecture matérielle

Une fois l'environnement mémoire de l'application virtualisé, mappé et protégé (segmentation logique), puis son code et ses données statiques binaires chargés 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.

Il sera donc essentiel de revenir sur ces deux schémas vers la fin des enseignements, afin de vous assurer d'avoir compris ces deux figures.



IV. Outils et environnement de développement

L'archive de travail est directement téléchargeable sur la plateforme pédagogique de l'ENSICAEN :

- 2A GPSE SATE : <https://foad.ensicaen.fr/course/view.php?id=696>
- 2A GPSE FISA : <https://foad.ensicaen.fr/course/view.php?id=1022>

L'ensemble de la trame de Travaux Pratiques sera réalisée sur système GNU/Linux (Ubuntu dans notre cas, mais n'importe quelle distribution fera l'affaire). La compilation et l'édition des liens se feront donc sur une chaîne de compilation, outils de développement et ABI (*Application Binary Interface*) GNU GCC (GNU Collection Compiler)². Les logiciels GNU sont tous des logiciels libres et ouverts, soutenus par la *Free Software Foundation* (fondateur en 1985 Richard Stallman). Pour information, les chaînes de compilation GNU-like sont à notre époque les plus répandues, notamment dans le domaine des systèmes embarqués (supportant notamment les architectures Microchip, ARM, MIPS, Open Hardware RISC-V, ...). Elles s'imposent de plus en plus comme un standard.

Installation du système et des packages

Si vous souhaitez installer les outils sur votre machine personnelle et garder une configuration système proche des machines de l'école, nous vous conseillons d'installer un système Ubuntu LTS (*Long-Term Support*) réel ou virtualisé³. La version actuellement utilisée à l'école est Ubuntu 22.04 LTS. Néanmoins, même si cela est recommandé, il ne s'agit en rien d'une obligation, tant qu'un GCC est présent dans le système et que Linux supervise la machine hôte.

De même, si vous possédez déjà un système Windows sur votre machine, vous avez notamment la possibilité d'installer un Linux en *dual-boot* à côté de Windows ou de virtualiser Ubuntu sur une machine virtuelle. Le projet *VirtualBox*⁴ est par exemple un projet Open Source performant, bien maintenu et stable. Vous trouverez de nombreux tutoriels permettant d'installer un Ubuntu sur *VirtualBox*. Ne pas oublier de paramétrer votre configuration système sous *VirtualBox* en offrant les ressources suffisantes à votre système virtualisé à l'usage (virtualisation matérielle VT-x/AMD-V, 3/4 des ressources CPU, 3/4 des ressources RAM, ...) et en configurant le réseau.

Une fois votre système installé, voici potentiellement ci-dessous quelques packages complémentaires à installer.

```
sudo apt-get update
```

```
sudo apt-get install build-essential gcc-multilib binutils putty
```

² <http://gcc.gnu.org/>

³ <https://ubuntu.com/#download>

⁴ <https://www.virtualbox.org/wiki/Downloads>

V. Découpage temporel

Le document imprimé ne contient que les chapitres étudiés en TP.

Le document numérique contient l'intégralité des chapitres (dont ceux traités uniquement avec les INFO). Ces chapitres sont laissés à votre disposition pour compléter votre compréhension de l'enseignement et vous pouvez évidemment demander un support à vos enseignants pour ces parties hors TP.

Séances 1 et 2 :

Partie 1 Prélude.....	7
Partie 2 Chaîne de compilation.....	15

Séance 3 :

Partie 3 Allocation statique et fichier ELF.....	35
Partie 4 Assembleur Intel x86.....	45

Séance 4 et 5 :

Partie 5 Allocation automatique et segment de pile.....	57
---	----

Séance 6 :

Partie 6 Allocation dynamique et segment de tas.....	75
Partie 7 Mémoires cache.....	83

Non-abordé en TP :

Partie 8 Exceptions matérielles et Signaux UNIX.....	89
--	----

Séance 7 :

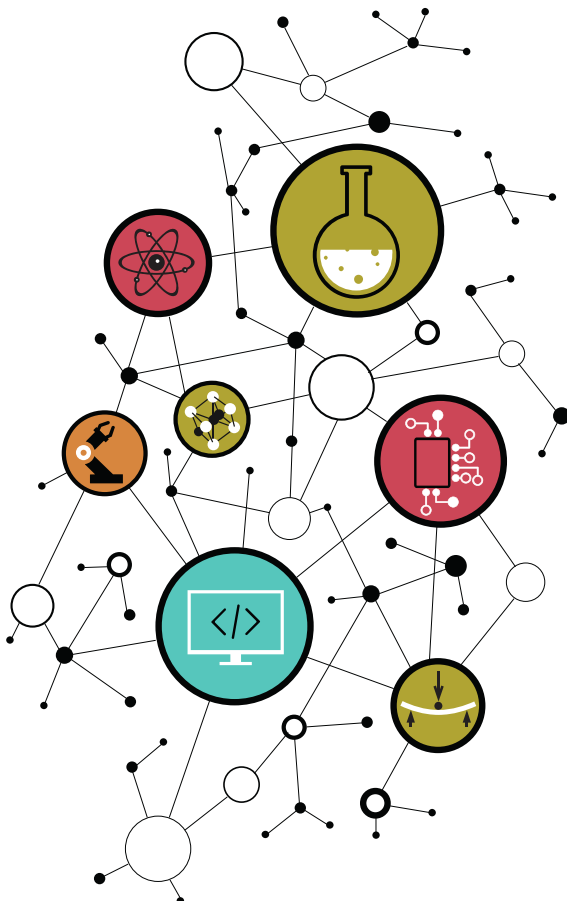
Partie 9 Hacking.....	97
-----------------------	----

Non abordé en TP :

Partie 10 Mémoire de masse et stockage des fichiers.....	109
--	-----

PARTIE 2

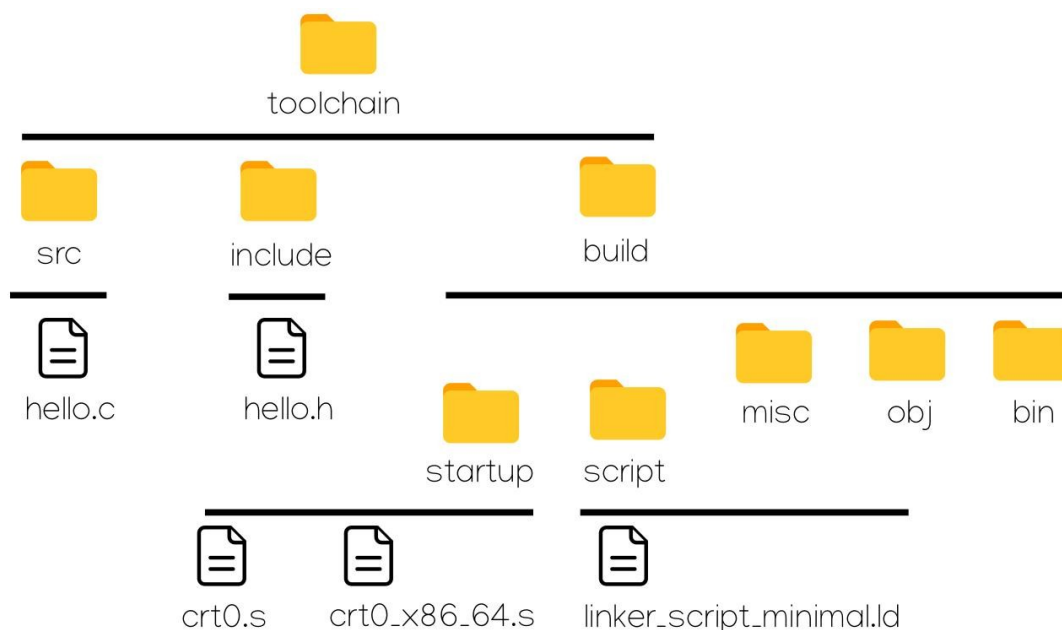
CHAÎNE DE COMPILATION



I. Arborescence

Dans ce chapitre, nous allons nous intéresser aux différentes 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 travailler, ouvrez un terminal et placez-vous 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.



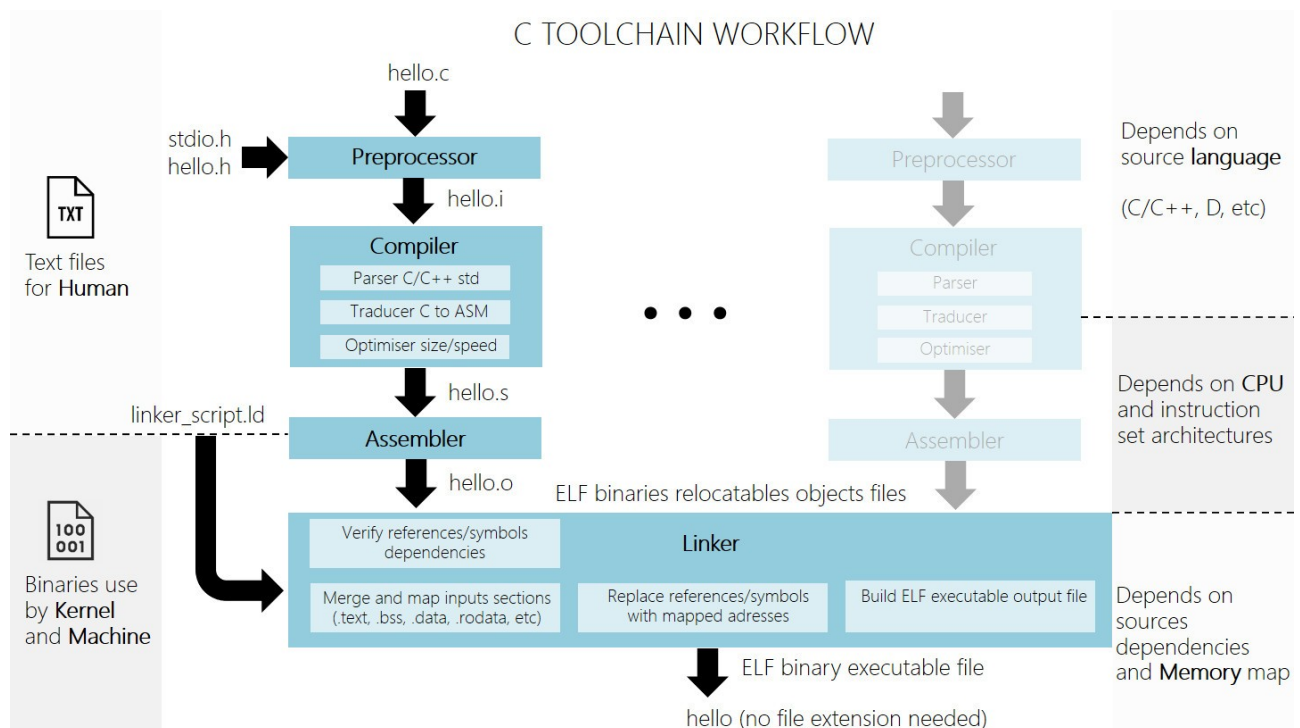
II. GCC Toolchain

La **compilation** (travail d'analyse et de traduction) se décompose en 3 grandes étapes :

1. le **prétraitement** (préparation du code avant compilation – analyse lexicale, supprime, copie, colle, remplace) ;
2. la **compilation** proprement dite (analyse syntaxique et sémantique, traduction vers l'assembleur de l'architecture CPU cible et optimisation optionnelle) ;
3. et enfin l'**assemblage** (conversion 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, représente ces différentes étapes exécutées séquentiellement par la *toolchain*. La compilation est un processus indépendant pour chaque fichier source à compiler. L'éditeur des liens (*linker*) travaillera alors avec l'ensemble des fichiers objets binaires générés à la compilation. Le résultat de ce processus statique est la génération d'un fichier binaire exécutable, dans notre cas au format binaire ELF (*Executable and Linkable Format*) sur système Unix.



III. Compilation

Avant d'étudier la chaîne de compilation étage par étage, procédons à la compilation complète du fichier source `hello.c`, puis exécutons le fichier généré par la chaîne de compilation.

```
gcc -m32 -I./include src/hello.c -o build/bin/hello
./build/bin/hello
```

En appelant l'utilitaire `objdump` sur notre programme exécutable de sortie comme ci-dessous, nous pouvons observer le désassemblage (*reverse engineering*, traduction binaire vers assembleur x86) du fichier binaire précédemment produit. Nous pouvons d'ailleurs constater qu'il y a plus de code binaire généré que notre simple programme élémentaire implémentant un `printf` dans le *firmware* de sortie. Il s'agit du code des fichiers de *startup*. Ce point sera étudié dans la suite de cet exercice.

```
objdump -S ./build/bin/hello
```

Quelles sont les adresses virtuelles des labels `main` (point d'entrée de l'application) et `_start` (point d'entrée des fichiers de *startup* et du firmware dans son ensemble) ? Nous retrouverons et comprendrons plus précisément ces adresses par la suite.

L'utilitaire `objdump` est un service standard de *binutils*, projet GNU proposant une boîte à outils pour la génération, l'analyse et la manipulation de fichiers binaires notamment au format ELF (*Executable and Linkable Format*) compatible sur système *Unix-like* (<https://www.gnu.org/software/binutils/>). Ce package est standard sur système GNU/Linux et est souvent nativement porté sur la majorité des systèmes d'exploitations de bureautique de cette même famille (Ubuntu, Debian, Fedora, etc). Ils sont des programmes outils primordiaux d'une chaîne de compilation et seront utilisés dans cette trame d'enseignement. Il comprend notamment les services suivants :

- *ld* (*GNU linker*) pour l'édition des liens
- *as* (*GNU assembler*) pour l'assemblage
- *ar* (*GNU archiver*) pour la génération de bibliothèque statique
- *objdump* (*object file reader*) pour l'affichage d'informations de fichier objet
- *readelf* (*ELF format object file reader*) pour l'affichage d'informations de fichier ELF
- *strip* (*symbols cleaner*) pour le nettoyage des symboles dans des fichiers objets
- *objcopy*, *gold*, etc.

III.1. Preprocessing

À partir de maintenant, nous allons décomposer les différentes étapes du processus de compilation et d'édition de liens. Une bonne compréhension et maîtrise des outils de développement est un point central pour un développeur, notamment dans le monde du système. L'intention ici est de vous faire comprendre ce qu'il se passe derrière le joli bouton « Build » fourni par tout IDE du marché.

En partant du même fichier source `hello.c`, appelez uniquement le premier étage de la chaîne de compilation avec la commande suivante.

```
gcc -E -m32 -I./include src/hello.c > build/misc/hello.i
```

Que fait l'option `-E` de la commande `gcc` ?

<code>man gcc</code>	# Pour lire le manuel de gcc
<code>/-E</code>	# Pour chercher dans le manuel (/), chercher <code>-E</code>
<code>n</code>	# n = next result (N = previous result)

Affichez le contenu du fichier `hello.i` sur la console et analysez-le.

```
cat build/misc/hello.i
```

Quelles sont les analyses et les traitements réalisés par le préprocesseur du langage C ?

Mettez à 0 la valeur de la macro `PRINT_HELLO` présente dans le fichier d'en-tête `hello.h` puis répétez les tâches précédentes. Analysez le code source de sortie.

/!\ Important : poursuivre la trame de TP en laissant cette macro à 0 /!

Préciser les rôles des directives de précompilation C/C++ suivantes : `#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else` et `#endif` (s'aider d'internet si nécessaire)

III.2. Analyse et génération de code natif

En partant du fichier préparé par le préprocesseur de la *toolchain*, franchissez le deuxième étage de la chaîne de compilation. Affichez et parcourez le fichier de sortie.

```
gcc -S -Wall -m32 build/misc/hello.i -o build/misc/hello.s
```

Vous pouvez constater que sa lecture reste complexe. GCC ajoute par défaut de code de sécurité additionnel, mais nous pouvons lui demander de ne pas les incorporer avec les options suivantes. Ainsi le fichier de sortie ne contiendra désormais que le code utile à l'application.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall -m32 build/misc/hello.i -o build/misc/hello.s
```

La sécurité fait partie des aspects les plus critiques dans l'évolution actuelle des systèmes numériques de traitement de l'information. Depuis maintenant des années, et cela continue toujours d'évoluer, les outils de compilation ajoutent à la traduction du code et techniques de protection et de robustification en surcouche au code applicatif utile afin de durcir la sécurité globale du système. Voici ci-dessous 4 options à passer à GCC afin de retirer le code de protection et de sécurité additionnel (dans le cadre des TP) si nous souhaitons observer uniquement le code assembleur applicatif utile dans le cadre de cet enseignement :

-fno-asynchronous-unwind-tables afin de retirer les directives d'assemblage `.cfi` et faciliter la relecture du code. Les directives `.cfi` sont des informations additionnelles ajoutées à la compilation pour la gestion d'exceptions en C/C++ ⁵.

-fno-pie (*Position Independent Executables*) permet d'invalider la capacité du système à générer aléatoirement un modèle mémoire adressable pour l'application ainsi compilée (ASLR ou *Address Space Layout Randomization*)

-fno-stack-protector (*Stack Smashing Protector*) permet de déployer des mécanismes de protection afin d'éviter des débordements de tampon. En effet, cette capacité (*stack smashing protector*), activée par défaut sur le GCC sous Ubuntu permet au compilateur d'insérer du code de protection au code applicatif, notamment pour la protection d'éventuelles corruptions de la pile par des programmes d'attaque. Cette option peut-être typiquement retirée à la compilation durant la création de bibliothèques partagées (pour ouvrir la possibilité d'émettre des hypothèses sur la pile) ou lorsque nous sommes soucieux des performances temporelles de notre programme.

-fcf-protection=none permet d'autoriser ou d'invalider l'instrumentation de code par contrôle de flux afin d'améliorer la sécurité du système. Par exemple en vérifiant la validité d'appels indirects de fonctions, de sauts indirects, d'adresses de retour de fonctions, ... ⁶.

⁵ http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

⁶ <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

Pour beaucoup, il doit s'agir de votre première lecture d'un programme en langage d'assemblage. Si c'est le cas, voilà, c'est fait, faites un vœux !

Un fichier assembleur est avant tout un fichier texte, et donc à destination d'un humain. Nous pouvons si nécessaire développer en assembleur, il s'agit du langage de programmation de plus bas niveau sur la machine (hors binaire). À notre époque, nous ne rencontrons ce type de développement que dans certains cas spécifiques (optimisations spécifiques à une architecture CPU donnée, diminution de l'empreinte mémoire d'un programme sur système contraint, bibliothèques spécialisées de calcul, hacking et pénétration de système, etc). Tous les ans, quelques élèves ont à réaliser du développement assembleur dans des entreprises ayant l'un des besoins spécifique cité précédemment.

Repérez ci-dessous les éléments suivants : **label** (ou étiquette) en rouge, **instructions** en vert et **opérandes** en bleu

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

Qu'est-ce qu'un label en assembleur ? Que représentera plus tard à l'exécution le label `main` (simple chaîne de caractères) ?

Qu'est-ce qu'une instruction assembleur pour la machine ?

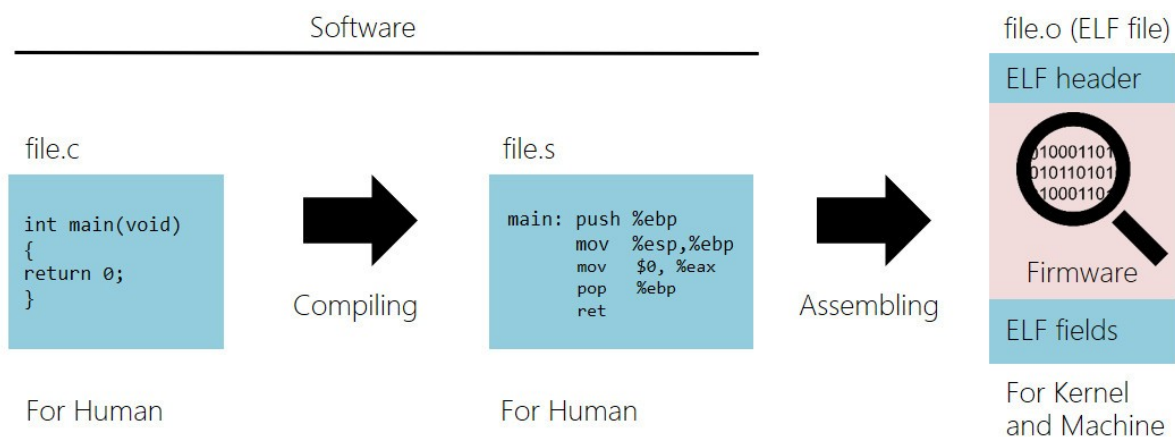
Qu'est-ce qu'un registre ? Combien de registres CPU utilisés observez-vous dans ce programme assembleur ?

III.3. Assemblage

Partant du fichier assembleur `build/misc/hello.s` précédemment généré, passez à travers l'étage d'assemblage et achevez ainsi le processus de compilation.

```
as --32 build/misc/hello.s -o build/obj/hello.o
```

Le fichier binaire résultant est un fichier dit objet **relogeable**, dans notre cas au **format ELF 32-bits (Executable and Linkable Format)**. Ce format de fichier binaire est généralisé sur système *Unix-like*, il s'agit donc du standard le plus répandu au monde à notre époque (applications, drivers/modules et bibliothèques aux formats binaires). Maintenant, nous ne pouvons plus utiliser un éditeur de texte afin d'analyser son contenu. Mais vous pouvez tout de même essayer. Il nous faudra utiliser des utilitaires dédiés à l'analyse du format de fichier ELF, par exemple `objdump` ou `readelf` également proposés dans le package `binutils`.



Le fichier `hello.o` est un fichier ELF 32-bit. En analysant le fichier binaire désassemblé, quelle est l'adresse relative de la fonction `main()` ?

```
objdump -S build/obj/hello.o
```

En analysant l'en-tête de fichier ELF, précisez l'architecture CPU cible ?

```
readelf -h build/obj/hello.o
```

Précisez le type de fichier binaire ? Pourquoi nomme-t-on ce type de fichier « relogeable » ?

Le fichier `hello.o` est-il exécutable ? Pourquoi ? Essayer de l'exécuter.

IV. Édition des liens

Finalisez la génération d'un fichier exécutable par l'édition des liens, puis exécutez le programme. Nous étudierons plus en détail l'édition des liens dans la suite de l'exercice.

```
gcc -m32 build/obj/hello.o -o build/bin/hello
```

En analysant l'en-tête de fichier ELF, précisez le type de fichier binaire ?

```
readelf -h build/bin/hello
```

En analysant l'en-tête de fichier ELF, déduire quelle est l'adresse d'entrée du programme ?

En analysant le fichier binaire désassemblé, quelle est l'adresse relative de la fonction `main` ?

```
objdump -S build/bin/hello
```

En analysant le fichier binaire désassemblé, quelle est l'adresse de la fonction `_start` ?

Verdict, le fichier `hello` est-il exécutable ?

```
./build/bin/hello
```

Pourquoi et surtout comment, la réponse se trouve dans la suite de la trame ...

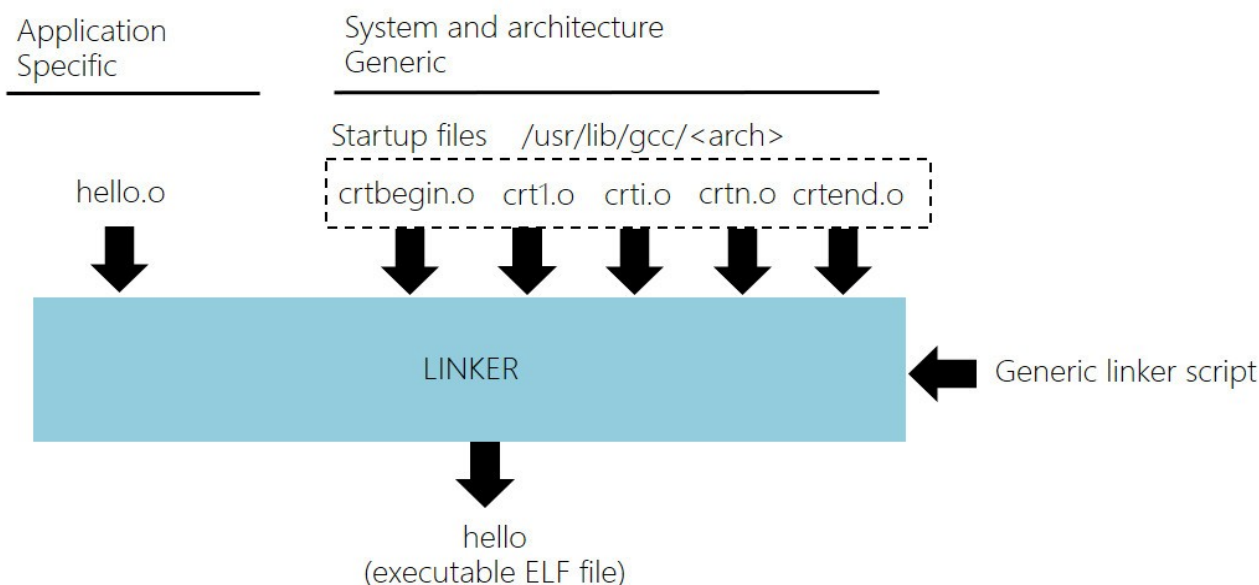
IV.1. Startup file

Nous allons maintenant jouer à un jeu technique et technologique : chercher à obtenir un fichier binaire ELF exécutable sur machine x86 (32-bit) et système GNU/Linux de taille minimale. Pour ce faire, nous allons jouer avec l'étape d'édition des liens, et réaliser étape par étape les différents traitements du *linker* manuellement. Le tout avec quelques ajustements maison.

Relevez la taille du fichier binaire `hello.o`, et celle du fichier binaire `hello`. Qu'en conclure ?

```
ls -l build/obj
ls -l build/bin
```

En langages C/C++, la fonction `main()` n'est jamais le premier point d'entrée réel d'un programme. Tout programme C/C++ débute par un autre programme générique d'amorçage. Ces programmes sont souvent nommés **fichiers de startup (démarrage)**. Leur nom est le plus souvent préfixé par `crt` (*C startup routine*). Ils réalisent quelques initialisations nécessaires à l'application (lier les bibliothèques dynamiques par exemple), ils offrent notamment la possibilité au développeur d'ajouter du code avant le début et en fin d'application (sections `.init` et `.fini`), initialisent les constructeurs en C++, ... L'entrée par défaut typique d'un programme sur système GNU/Linux est la fonction étiquetée `_start`. Nous verrons par la suite que cette entrée peut être aisément renommée si nécessaire. Ce ou ces fichiers d'amorçage restent toujours les mêmes utilisés à l'édition des liens et sont pré-compilés pour une architecture cible donnée avant d'être portés sur le système hôte (sources⁷ des fichiers de *startup* utilisés par GCC en x64). Ils sont donc dépendants de la chaîne de compilation et du système d'exploitation utilisés (schéma ci-dessous sur GCC v7).



Observez les fichiers de *startup* ajoutés à l'édition des liens durant l'étape précédente.

```
gcc -v -Wall -m32 build/obj/hello.o -o build/bin/hello
```

7 <https://github.com/gcc-mirror/gcc/tree/master/libgcc/config/ia64>

Nous allons utiliser un fichier d'amorçage minimal développé par nos soins. Ouvrez le fichier assembleur `build/startup/crt0.s` et parcourez le programme (cf. ci-dessous). Ce fichier se veut minimaliste en taille, même s'il nous est encore possible de gagner quelques octets.

```
.global _start

.text
_start:
    push    %ebp
    mov     %esp, %ebp
    call    main
    mov     $1, %eax
    int     $0x80
```

Assemblez ce nouveau fichier d'amorçage

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

... et ajoutez-le manuellement durant l'édition des liens en appelant directement le *linker* `ld`.

```
ld -melf_i386 build/obj/crt0.o build/obj/hello.o -o build/bin/hello
```

Notre fichier d'amorçage maison permet effectivement de nous rapprocher de notre objectif de diminution de la taille de l'exécutable produit, mais en contrepartie nous ne pouvons plus utiliser de bibliothèques liées dynamiquement, comme la bibliothèque standard `libc` du langage C. Nous ne pouvons donc plus utiliser la fonction `printf()` (d'où la nécessité de mettre à '0' la macro `PRINT_HELLO` dans le fichier d'en-tête `disco/toolchain/include/hello.h`).

Analysez le fichier binaire désassemblé de sortie à l'aide de l'utilitaire `objdump`. Normalement, vous devez pouvoir retrouver toutes les instructions assembleur précédemment analysées dans les fichiers `hello.s` et `crt0.s`. Nous observons l'ensemble du contenu de notre firmware minimaliste.

```
objdump -S build/bin/hello
```

En analysant le binaire désassemblé, quelles sont les adresses des fonctions `main` et `_start` ?

Quelle est maintenant la taille sur le disque du fichier binaire ELF de sortie ?

```
ls -l build/bin
```

Le fichier `hello` est-il toujours exécutable (sans défaut de segmentation à l'exécution) ?

```
readelf -h build/bin/hello
./build/bin/hello
```

IV.2. Linker script

Nous pouvons constater à cette étape que l'empreinte mémoire disque du programme binaire exécutable commence à être grandement réduite. Néanmoins, l'éditeur de liens continue à architecturer le fichier ELF de sortie avec des sections génériques, dont certaines sont maintenant inutiles à nos besoins (simple exécution de notre programme). Une **section** est une zone logique présente dans le fichier ELF de sortie. Il lui est notamment associé une nature de l'information d'accueil (*code*, *data* ou autre), des droits d'accès à l'information (*R/W* ou *ReadOnly*), une adresse de départ et une taille de section. Le *linker* utilise un fichier texte nommé **linker script**⁸ (extension `.ld`), lui permettant de définir l'ossature du binaire (ou *firmware*) du fichier ELF exécutable de sortie.

Affichez le **linker script** générique utilisé par défaut par l'éditeur de liens de GCC.

```
gcc -m32 -Wl,--verbose
```

La lecture est complexe, c'est normal, mais il ne vous est pas demandé de tout comprendre, ce n'est pas le but ! Notez simplement que ce *linker script* permet notamment d'organiser le positionnement des codes binaires des fichiers de *startup* dans le firmware de sortie (fichier riche en informations).

Faisons le lien entre ce *linker script* et les sections présentes dans notre programme exécutable ELF de sortie. Nous les nettoierons par la suite en enlevant voire concaténant les sections inutiles à une simple exécution !

```
objdump -h build/bin/hello
```

Combien de sections observez-vous ? Précisez leur nom, leur nature, leur adresse de départ et leur taille !

Que contient la section `.text` ?

```
objdump -S build/bin/hello
```

Que contient la section `.comment` ?

```
objdump -s build/bin/hello
```

⁸ <https://sourceware.org/binutils/docs/ld/Scripts.html>

Ouvrez le fichier `build/script/linker_script_minimal.ld` et analysez son contenu. Vous pourrez constater qu'il s'agit d'un élagage du *linker script* utilisé par défaut par GCC. Ce fichier définit voire retire des sections existantes. Il définit également la machine ciblée par le firmware (i386 dans notre cas), le point d'entrée du programme (`_start`) et le format de fichier de sortie (ELF 32-bit, compatible architecture Intel 386 dans notre cas).

	File: build/script/linker_script_minimal.ld
1	OUTPUT_FORMAT("elf32-i386")
2	OUTPUT_ARCH(i386)
3	ENTRY(_start)
4	
5	SECTIONS
6	{
7	. = SEGMENT_START("text-segment", 0x08048000) + SIZEOF_HEADERS;
8	.text :
9	{
10	*(.text)
11	}
12	.rodata :
13	{
14	*(.rodata)
15	}
16	.data :
17	{
18	*(.data)
19	}
20	.bss :
21	{
22	*(.bss)
23	}
24	/DISCARD/ :
25	{
26	*(.comment)
27	*(.note.GNU-stack)
28	*(.eh_frame)
29	}
30	}

Modifiez le fichier `src/hello.c` pour y déclarer trois variables statiques globales (non-initialisée, initialisée et en lecture seule) comme ci-dessous.

```
#include <stdio.h>

int a;
int b = 1;
const int c = 2;

int main (void)
{
    return 0;
}
```

Recompilez le fichier en vous arrêtant après l'assemblage, puis réalisez l'édition des liens en utilisant notre *linker script* maison. Nous allons analyser la table des sections générée !

```
gcc -c -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -m32 -I./include src/hello.c -o build/obj/hello.o

ld -melf_i386 -T build/script/linker_script_minimal.ld build/obj/crt0.o build/obj/hello.o -o build/bin/hello
```

Combien de sections observons-nous ? Précisez leur nom, leur nature, leur adresse de départ et leur taille. Est-ce cohérent ?

```
objdump -h build/bin/hello
```

hello (executable ELF file)

ELF header

.text

Firmware

010001101
010110101
1000110

.rodata

.data

.bss

ELF sections table

.text
.rodata
.data
.bss

descriptions
and
informations

Other ELF fields

Que contiennent les sections `.data` et `.rodata` ? Placez les variables concernées sur le schéma ci-contre.

Par élimination, où se situe la variable `a` ? Placez-la sur le schéma ci-contre.

Quelle est maintenant la taille sur le disque du fichier binaire ELF de sortie ?

```
ls -l build/bin
```

Nous allons maintenant conclure l'exercice par un ultime nettoyage du fichier ELF de sortie (nettoyage de la table des symboles). Vous comprendrez avec le prochain chapitre de TP l'action réalisée par cette commande.

```
strip build/bin/hello
```

Quelle est maintenant la taille sur le disque du fichier binaire ELF de sortie ?

```
ls -l build/bin
```

Le fichier `hello` est-il toujours exécutable ?

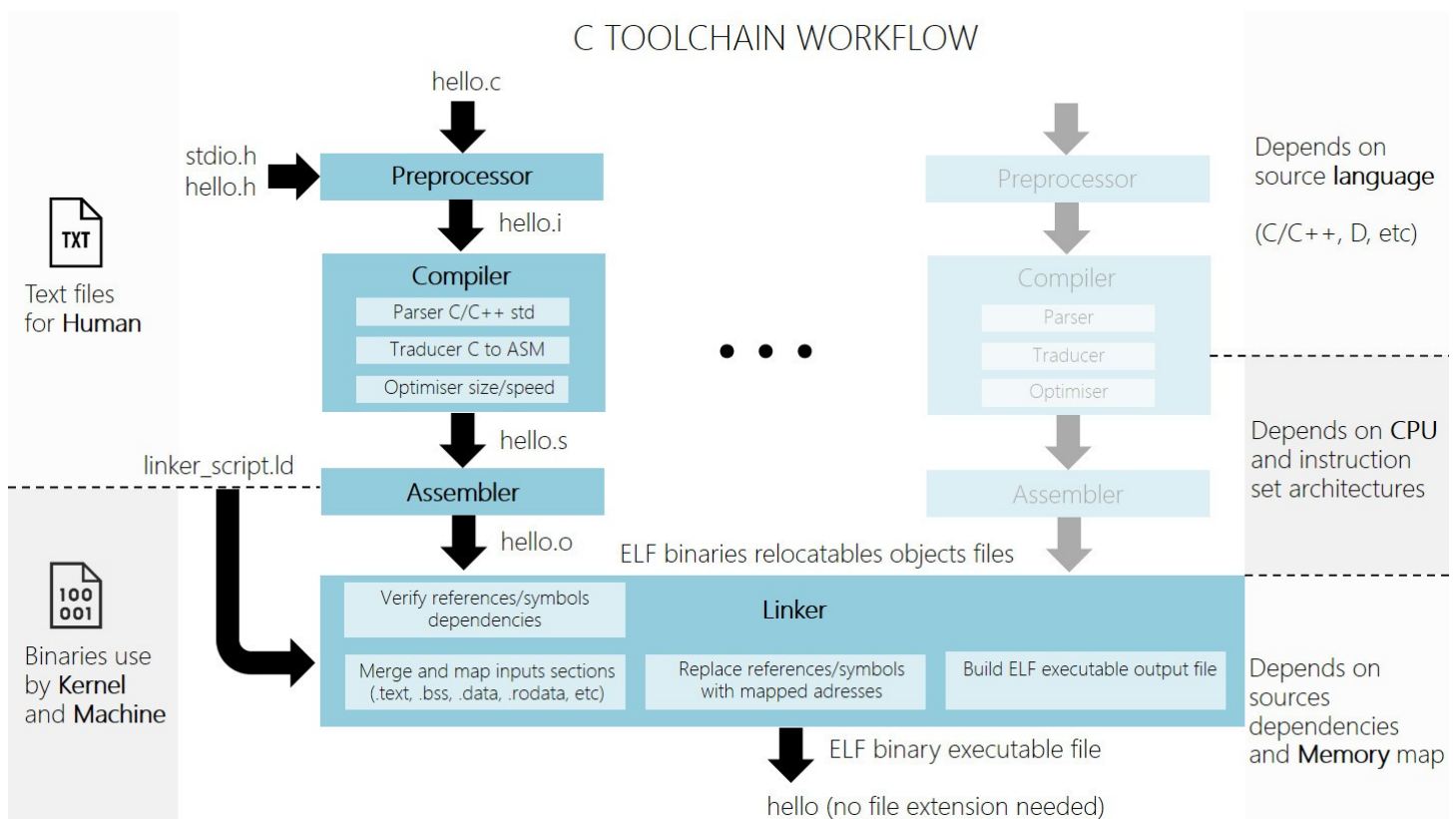
```
./build/bin/hello
```

V. Synthèse sur l'exercice

Voilà, l'exercice est maintenant terminé. Vous venez d'obtenir probablement l'un des firmware les plus légers que l'on puisse exécuter sans erreur (exception matérielle, défaut de segmentation logique, ...) sur un système GNU/Linux 32-bit et sur architecture matérielle 32-bit x86 (à quelques dizaines d'octets près). Rustique, mais il s'exécute sans générer de défaut processeur ni de défaut système ! C'est magique ...

Comprendre l'essence de cet exercice peut prendre du temps et nécessitera probablement de repasser sur le chapitre. Cependant, elle vous permettra à l'avenir d'aborder sereinement bien des problèmes rencontrés en compilation sur des projets complexes et conséquents en taille. Le gain en temps et en énergie peut être considérable !

C TOOLCHAIN WORKFLOW



```
gcc -E -m32 -I./inc src/hello.c > build/misc/hello.i
gcc -S -Wall -m32 build/misc/hello.i -o build/misc/hello.s
as --32 build/misc/hello.s -o build/obj/hello.o
ld -melf_i386 -T build/script/linker_script.ld build/obj/crt0.o build/obj/hello.o -o
build/bin/hello
./build/bin/hello
```

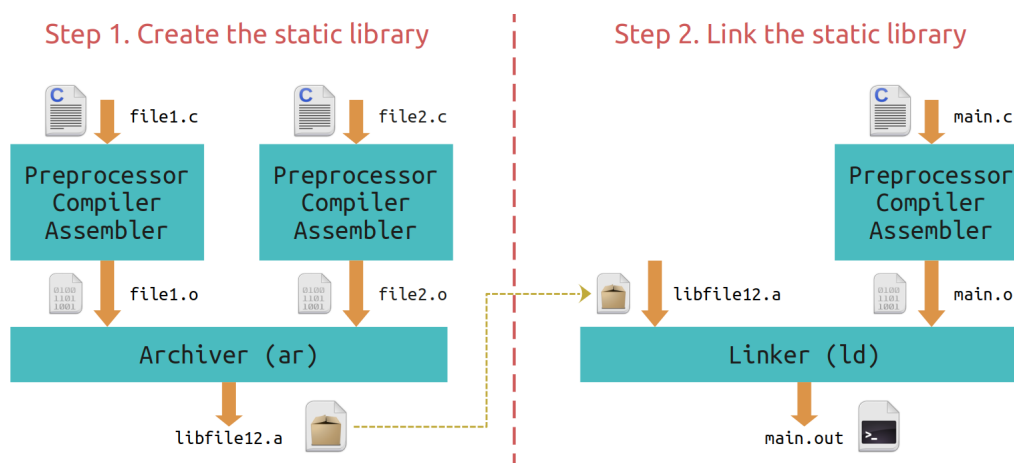
VI. Librairie statique

Vous avez pu constater sur les exercices précédents que le processus de compilation se fait indépendamment pour chaque fichier source, où chaque fichier donne naissance à un fichier objet. Ensuite l'éditeur de lien associe tous ces fichiers `.o` pour fabriquer un fichier exécutable. Toutefois ce *workflow* ne reflète qu'une partie des cas d'usages.

Dans le vaste monde de l'informatique et de l'embarqué, certains développeurs vont travailler pour d'autres en développant par exemple des fonctionnalités vouées à être réutilisées (bibliothèques, drivers, middleware, ...) par d'autres programmes (qu'on appellera code applicatif). Dans ces cas, les premiers développeurs ne fournissent pas directement leurs sources en C (même si c'est possible), mais vont plutôt fournir une version pré-compilée du code. Or pour un gros projet logiciel, cela ferait une grande quantité de fichiers objets à transmettre. C'est pourquoi les développeurs se tournent vers les bibliothèques statiques.

Le rôle de la **bibliothèque statique** (également appelée **archive**) est de contenir tout le code précompilé dans un seul fichier, qui pourra être réutilisé plus tard dans n'importe quel autre projet logiciel. Les fichiers nécessaires sont compilés en fichiers objets, puis archivés dans un fichier binaire `.a`. Cependant il ne s'agit pas d'un format ELF, mais d'une concaténation de fichiers.

La figure ci-dessous illustre ce propos, en prenant pour exemple les fichiers `main.c`, `file1.c` et `file2.c` de l'archive de TP (répertoire `disco/toolchain/`).



Compilez les fichiers `file1.c` et `file2.c` en vous arrêtant après l'étape d'assemblage.

```
gcc src/file1.c -c -o build/obj/file1.o
gcc src/file2.c -c -o build/obj/file2.o
```

Retrouvez et indiquez le type des fichiers binaires générés.

```
readelf -h build/obj/file1.o build/obj/file2.o
```

Observez le désassemblage de leur section `.text` (code binaire opératoire des instructions).

```
objdump -S build/obj/file1.o build/obj/file2.o
```

Construisez une librairie statique (ou archive) qui rassemble ces deux fichiers objets.

```
ar rcs build/lib/libfile12.a build/obj/file1.o build/obj/file2.o
```

Le programme **ar** (*GNU Archiver*) crée une archive (ou librairie statique), les options **r**cs permettant respectivement de (**r**) insérer/remplacer les membres dans l'archive, (**c**) créer l'archive et (**s**) mettre à jour la table des symboles (cf prochain chapitre). L'archive est également un fichier binaire au format ELF, nous allons donc analyser son contenu avec les utilitaires adéquats.

Les commandes suivantes renvoient-elles des informations sur l'archive à proprement parler ? Que peut-on en déduire sur la constitution d'une archive (ou librairie statique) ?

```
readelf -h build/lib/libfile12.a
objdump -S build/lib/libfile12.a
```

Compilez maintenant le projet logiciel complet, en partant du code applicatif (fichier **main.c**) et en liant le code pré-compilé contenu dans l'archive **libfile12.a**.

```
gcc src/main.c -L. build/lib/libfile12.a -o build/bin/main
```

Exécutez le fichier généré et constatez que le code contenu dans les fichiers **file1.c** et **file2.c** a bien été intégré dans le fichier exécutable final.

```
./build/bin/main
```

Vous pouvez le confirmer en observant le désassemblage de l'exécutable.

```
objdump -S build/bin/main
```

L'exercice s'arrête ici sur les librairies statiques, mais il existe également les **librairies dynamiques** (aussi appelées **shared objects**, d'extension **.so**). Contrairement au cas précédent, le code des librairies dynamiques n'est pas directement intégré dans l'exécutable final (autrement dit, le *linker* de la *toolchain* ne travaille pas avec les librairies dynamiques). À la place un **linker dynamique** viendra lier les librairies dynamiques au programme uniquement au moment de son exécution du programme.

C'est par exemple le cas de la librairie **libc-2.31.so** (contenant le code de nombreuses fonctions standard) qui sont liées par le *linker* dynamique **ld**. Ce dernier a travaillé quand votre tout premier programme a effectué son `printf("Hello World!")`, ou quand vous avez réalisé des allocations dynamiques de données avec un `malloc()`.

VII. Make et Makefile

Pour automatiser le processus de compilation sur des projets logiciels de complexité modérée, on utilise généralement un **Makefile**⁹. Pour faire simple, un **Makefile** est un fichier qui contient toutes les commandes de fabrication d'un projet logiciel, allant des commandes de compilation de chaque fichier source (du **.c** vers le **.o**) aux commandes de *linking* (des fichiers objets au fichier exécutable). Il est ensuite appelé par la commande **make** qui interprète ces commandes en fonction notamment de l'heure de dernière modification des fichiers.

Dans cet exercice il n'est pas question d'apprendre à écrire un **Makefile**, cependant si cela vous intéresse vous avez à disposition le fichier `tp/doc/fr_tutoriel_makefile.pdf`. L'idée de cet exercice est plutôt de comprendre comment les règles du **Makefile** s'exécutent en fonction de l'état courant du projet logiciel (et de ses fichiers sources).

Placez-vous dans le répertoire `disco/make/`, ouvrez les fichiers à disposition et analysez le projet logiciel. Décrivez brièvement le contenu du fichier **Makefile**.

Lancez la construction du projet avec la commande **make** (si aucun argument n'est donné à la commande, c'est la première règle du fichier **Makefile** qui est évaluée).

```
make
```

Qu'est-il affiché sur la console ?

Affichez les fichiers par ordre de dernière modification. Constatez que les fichiers objets sont plus récents que les fichiers sources et que l'exécutable est encore plus récent.

```
ls -lt
```

Relancez **make**. Qu'est-il affiché sur la console ? Pourquoi ?

⁹ Pour des projets de plus grande ampleur, on utilise plutôt CMake, mais ce ne sera pas étudié ici.

Que fait la règle `make clean` ? Lancez-la pour confirmer et regardez les fichiers du répertoire.

Relancez `make`. Qu'est-il affiché sur la console ? Pourquoi ?

Modifiez le fichier `file2.c` en retirant le commentaire de la ligne 8.

Relancez `make`. Qu'est-il affiché sur la console ? Pourquoi ?

Modifiez le fichier `file2.c` en retirant le commentaire de la ligne 8.

Relancez `make`. Qu'est-il affiché sur la console ? Pourquoi ?

Modifiez le fichier `file1.h` en retirant le commentaire de la ligne 4.

Relancez `make`. Qu'est-il affiché sur la console ? Pourquoi ?

Modifiez le fichier `file2.h` en retirant le commentaire de la ligne 4.

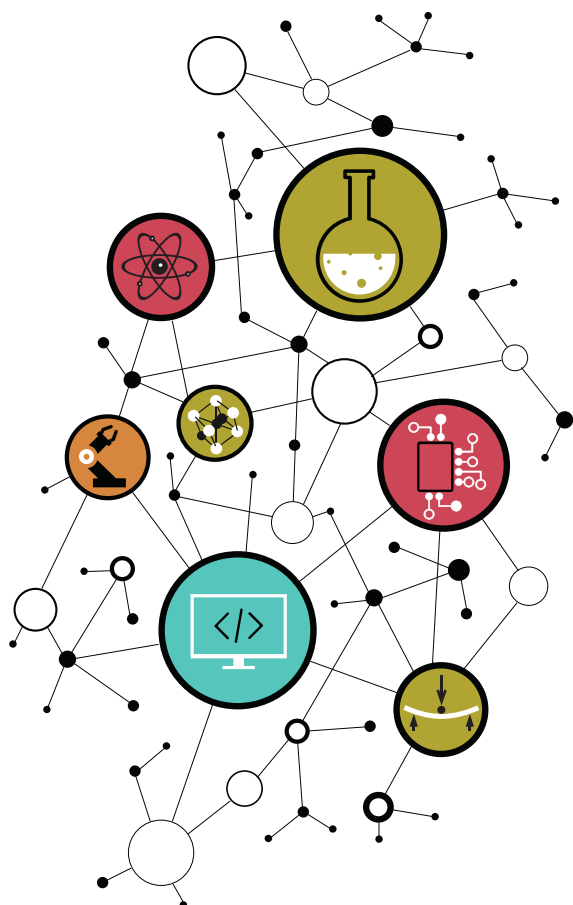
Relancez `make`. Qu'est-il affiché sur la console ? Pourquoi ?

Le `Makefile` contient l'intégralité des commandes de fabrication (au sens compilation et édition des liens) d'un projet logiciel.

La commande `make` interprète ce fichier de sorte à ne recompiler que le strict nécessaire, en fonction des dernières modifications apportées aux fichiers sources. L'idée est d'effectuer une compilation sélective des fichiers afin de limiter le temps de compilation. Imaginez le temps de fabrication d'un exécutable tel qu'un jeu vidéo (exécutable de plusieurs GB), il serait aberrant de compiler l'intégralité des fichiers si seule une ligne de code a été modifiée.

PARTIE 3

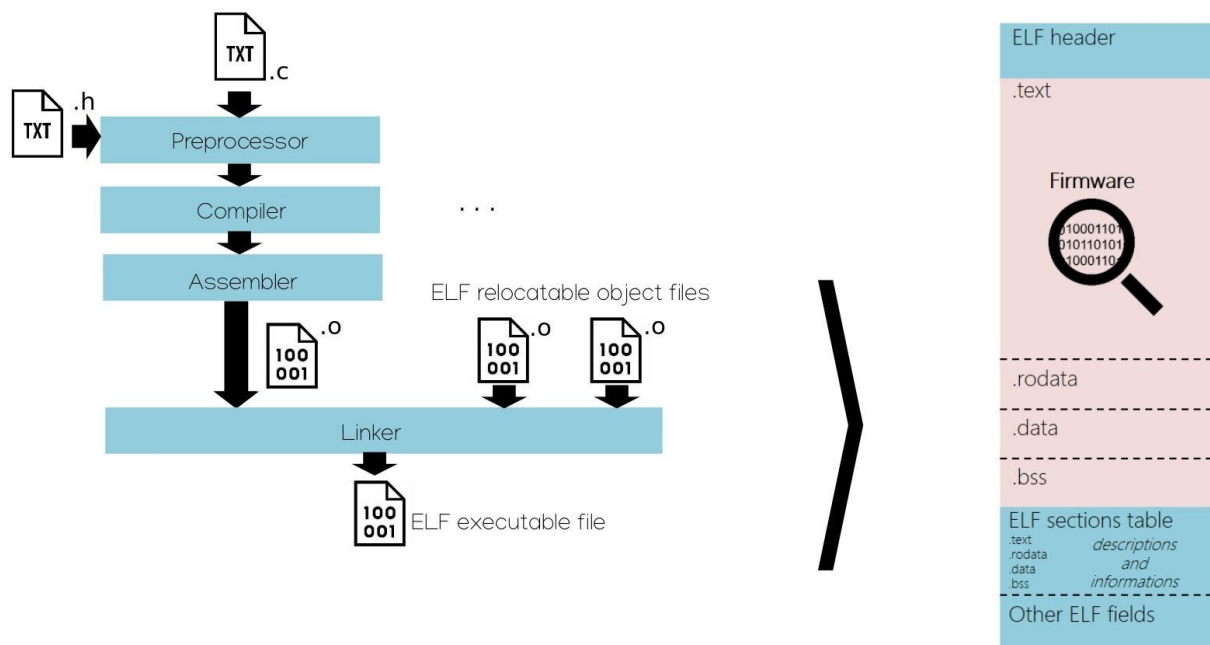
ALLOCATION STATIQUE ET FICHIER ELF



I. Compilation et allocation statique

Le premier chapitre était centré autour du processus de compilation, prenant en entrée un fichier texte générique (en C, C++, ...) ou architecture spécifique (en assembleur), pour créer un fichier binaire exécutable sur le processeur cible. Dans ce chapitre nous allons analyser le contenu des fichiers binaires générés afin notamment d'étudier un des mécanismes d'allocation des données : l'allocation statique.

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 média de stockage de masse (HDD, SSD, ...) 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 (tant qu'il n'y a ni nouvelle compilation et ni édition des liens du projet logiciel source). En d'autres termes, ce chapitre traite de l'allocation des **fonctions et variables statiques**. En revanche ce chapitre ne traite ni des variables locales (dont l'allocation automatique ou dynamique est gérée sur le segment de pile) ni des données allouées avec un `malloc/calloc/new/...` (dont l'allocation dynamique est gérée sur le segment de tas), puisque ces deux mécanismes auront chacun leur propre chapitre.



Dans cet enseignement, nous désignerons par ***firmware*** le code (forcément statique) et les données statiques binaires strictement utiles au fonctionnement d'un programme. Ce ***firmware*** est constitué de plusieurs zones logiques appelées « **sections** », que nous étudierons dans les prochaines pages.

Le ***firmware*** est encapsulé dans un cartouche au format ELF (en-tête, table des sections, en-tête du programme, ...) 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, ...) au moment du lancement de l'application.

II. Variables globales

Ouvrez un terminal et placez-vous dans le répertoire de travail `disco/static/`. Nous travaillons pour le moment avec le fichier `global_variable.c`.

```
char tab[1000000] = {'g','n','u'};

int main(void)
{
    tab[0] = 'G';

    return 0;
}
```

Compilez le fichier `global_variable.c` en vous **arrêtant à l'assemblage**.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-
protection=none -Wall global_variable.c
```

```
cat global_variable.s
```

Parmi toutes celles du fichier assembleur, les lignes qui concernent le tableau `tab` sont les suivantes. Expliquez celles-ci.

```
.globl tab
.data
.align 32
.type tab, @object
.size tab, 1000000
tab:
.string "gnu"
.zero 999996
```

Précisez le nom de la référence symbolique (label ou étiquette) représentant l'adresse du tableau `tab`. Le tableau n'est pas explicitement placé en mémoire (ce sera le travail de l'édition des liens) mais en revanche il est explicitement spécifié dans le script assembleur qu'il se situe dans la section `.data`.

Dans quelle section se trouve le label (ou étiquette ou référence symbolique) `main`? Vous noterez qu'un label peut pointer aussi bien sur une section de code (par exemple `.text`) que sur une section de donnée (par exemple `.bss`, `.rodata` ou `.data`).

Compilez le fichier `global_variable.c` en vous **arrétant à l'édition des liens** et précisez la taille du fichier objet binaire ELF relogeable de sortie.

```
gcc -c -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall global_variable.c -o global_variable.o
```

```
objdump -h global_variable.o
```

Précisez pour le nom et la taille de chaque section applicative. Vous observerez qu'après la compilation, aucune section n'est mappée en mémoire (adresse de base nulle). Elle seront relogées/mappées à l'édition des liens.

Dans quelle section se trouve le tableau statique `tab` ? Justifiez votre réponse.

Observez le contenu binaire du fichier objet. Comme il contient une grande quantité d'informations, nous allons l'exporter dans un fichier texte que vous ouvrirez.

```
objdump -s global_variable.o > global_variable.o.txt
```

```
gedit global_variable.o.txt &
```

Que contiennent les sections affichées ici ?

Compilez le fichier `global_variable.c` jusqu'à l'**édition des liens incluse**, et relevez la taille du fichier binaire ELF exécutable de sortie.

```
gcc -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none
-Wall global_variable.c -o global_variable
```

Ajoutez (ou enlevez) le qualificateur `static` devant la déclaration de la variable globale, recompilez et indiquez à nouveau la taille du fichier exécutable.

Vous constaterez que dans ce cas le qualificateur `static` n'a aucun impact sur la taille de l'exécutable (et en réalité sur l'ensemble du *firmware*), puisque les variables globales sont implicitement et par essence des variables allouées statiquement (donc stockées dans le *firmware*).

En observant la table des symboles (`objdump -t`, table contenant des informations sur toutes les références symboliques statiques dont leurs adresses logiques), quelle est l'adresse relative du tableau `tab` après édition des liens ?

```
objdump -t global_variable
```

Analysez le code du programme après désassemblage (`objdump -S`) et retrouvez l'adresse précédemment trouvée dans le code.

```
objdump -S global_variable
```

Observez la taille du fichier exécutable de sortie, le nettoyer (stripper) puis ré-observez sa taille sur le média de stockage de masse. Observez également la table des symboles après *stripping*. Quel traitement a été réalisé ? Le firmware a-t-il été modifié ? Il est à noter qu'il est possible de réaliser un stripping à l'édition des liens en passant l'option `-s` à GCC.

```
ls -l
strip global_variable
ls -l
objdump -t global_variable
```

III. Variables locales statiques

Par opposition aux variables globales existent les variables locales. Celles-ci sont définies et n'existent qu'au sein d'un *scope* (ou portée, délimité en C par deux accolades). Ces variables locales sont par défaut allouées dynamiquement, mais il existe la possibilité de les allouer statiquement.

Affichez, compilez et exécutez le programme `reminder_static_variable.c`.

```
gcc what_is_a_static_variable.c -o what_is_a_static_variable
./what_is_a_static_variable
```

Rappelez la différence entre une variable locale et une variable locale statique. Rappelez la différence entre une variable locale statique et une variable globale.

Compilez le fichier `local_static_variable.c` en vous **arrêtant à l'assemblage**. Affichez le contenu du fichier assembleur généré. Quel est le nom de la référence symbolique (label ou étiquette) représentant l'adresse de la variable `a` ? Dans quelle section se trouve-t-elle ?

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall local_static_variable.c
```

Compilez le fichier `local_static_variable.c` en vous **arrêtant à l'édition des liens**. Affichez respectivement la table des symboles (`objdump -t`), la table des sections (`objdump -h`) et le contenu binaire du fichier objet (`objdump -s`).

```
gcc -c -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall local_static_variable.c -o local_static_variable.o
objdump -t local_static_variable.o
objdump -h local_static_variable.o
objdump -s local_static_variable.o
```

Quelle est la taille de la section contenant la variable locale statique `a` ? Est-ce cohérent ?

Quelle est le contenu binaire de la variable locale statique `a` (valeur stockée dans le firmware, sur le média de stockage de masse) ? Est-ce cohérent ?

Reprenez l'exercice, en supprimant cette fois l'initialisation (`= 1`) lors de la déclaration de la variable `a`. Vous devriez remarquer que le nom de la section contenant la variable `a` a changé.

IV. Chaînes de caractères

Compilez le fichier `string.c` en s'arrêtant après la compilation mais avant l'édition des liens.

```
gcc -c -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall string.c -o string.o
```

Observez le contenu binaire du fichier objet.

```
objdump -s string.o
```

Dans quelle section se trouve la chaîne de caractères `GNU's design is Unix-like but differs by being free software and containing no Unix code !` ? Il s'agit d'une section, ce qui signifie que cette chaîne de caractères a été allouée statiquement, dans le fichier binaire ELF.

Dans quelle section semble se trouver la chaîne de caractères `GNU's Not Unix !` ? Peut-on réellement parler d'allocation ?

Observez le désassemblage de la section `.text` (instructions du *firmware*).

```
objdump -S string.o
```

Confirmez que les caractères de la chaîne `GNU's Not Unix !` sont directement encodés dans les instructions.

Indiquez et justifiez la taille de la variable `gnu_tab`. Ce tableau est alloué dynamiquement, sur le segment de pile. Ceci permet de pouvoir modifier le contenu du tableau.

Justifiez la taille de la variable `gnu_pointer` ? Ce pointeur est alloué dynamiquement, sur le segment de pile. Le contenu du pointeur est modifiable. En revanche il pointe vers une chaîne de caractères stockée dans la section `.rodata`, ce qui fait que le texte pointé n'est lui pas modifiable.

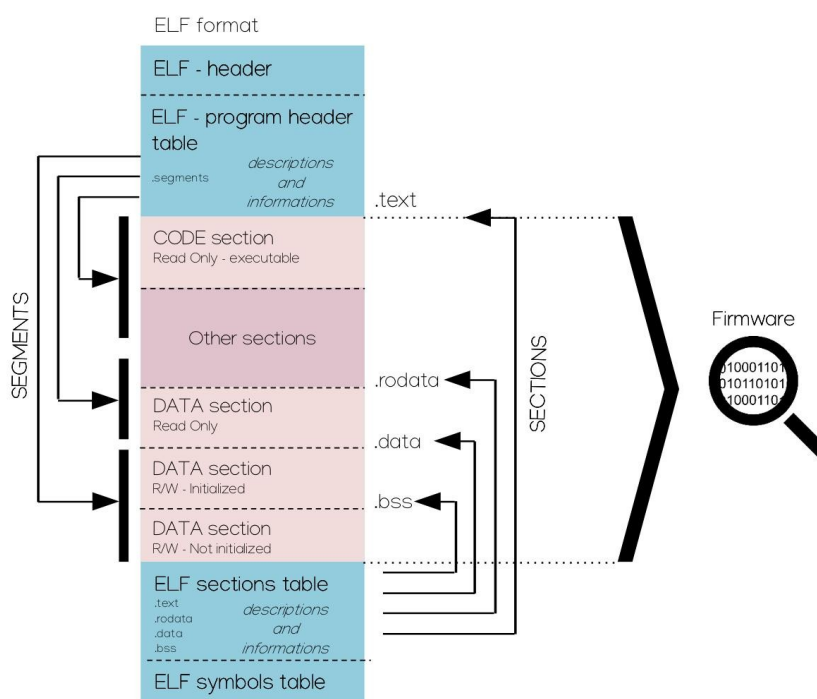
Note : il peut être intéressant de repasser sur cet exercice une fois que le chapitre sur l'allocation dynamique a été traité, puisqu'ici cohabitent allocation statique (chaîne de caractères `GNU's design ...`) et allocation dynamique (`gnu_pointer`, pointeur vers cette chaîne de caractères).

V. Synthèse

Pour conclure, bien se souvenir que dans un fichier binaire exécutable (formats ELF, COFF, PE, ...), nous ne trouvons pas que du code binaire. Les données allouées statiquement sont également présentes (variables globales, variables locales statiques et chaînes de caractères). Ces données existent donc déjà sur le média de stockage de masse (HDD, SSD, MMC, ...) avant même que le programme soit exécuté en mémoire principale.

Sauf si un développeur crée explicitement de nouvelles sections en spécifiant des attributs spécifiques durant une déclaration d'une variable¹⁰, une application pourra comporter au plus quatre 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

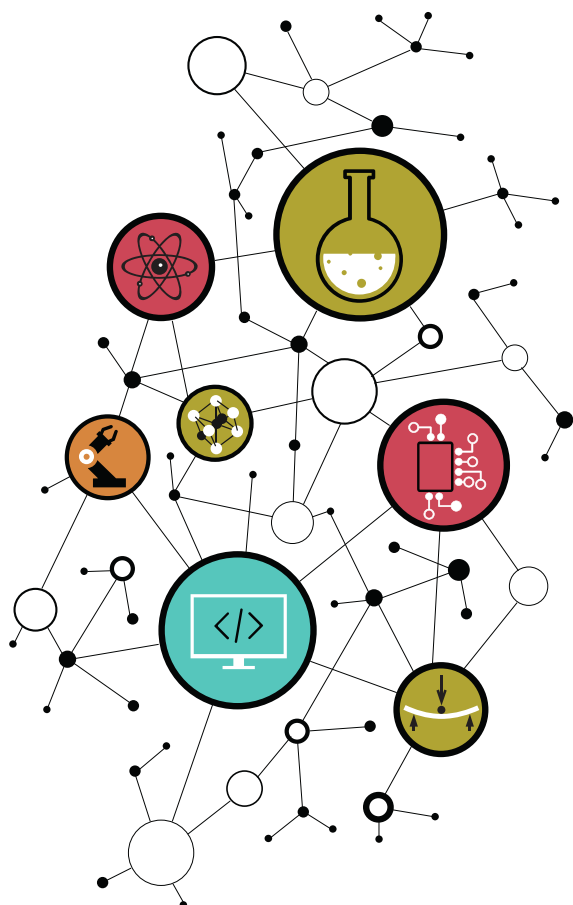


Lorsque nous exécutons un programme, le noyau du système (Linux, GNU Hurd, Mach, XNU, etc) va analyser les différents champs du fichier ELF exécutable (header, header du programme pour définir les futurs segments en mémoire vive, etc). Après analyse, le système va mapper et allouer en mémoire vive les segments nécessaires pour la bonne exécution de l'application puis charger (initialiser) les segments statiques avec les contenus associés dans le fichiers ELF toujours présent sur le média de stockage de masse.

Une fois les segments mémoire mappés, le code et les données statiques chargées en mémoire principale dans les segments associés, le système passe la main au code de l'application qui peut s'exécuter sur le CPU courant en commençant par le code des programmes de startup puis celui du main.

¹⁰ <https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html>

PARTIE 4 ASSEMBLEUR INTEL x86



I. L'assembleur sous architecture x86

I.1. Le langage d'assemblage

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 éditée 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, 8-16-32-64-bit, entier voire flottant, VLIW ou superscalaire, vectoriel ou scalaire, ...) sur des technologies différentes (x86, x64, ARM, MIPS, C6000, PIC18, ...).

```
int main (void)
{
    return 0;
}
```

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

L'assembleur présenté ci-dessus est par exemple de l'assembleur 32-bit IA-32 (Intel Architecture 32-bit) 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** : un label ou étiquette, est une référence symbolique (simple chaîne de caractères) représentant l'adresse mémoire logique (emplacement) de la première information 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 ou 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, ... 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 sauvée 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.

1.2. Syntaxe assembleur AT&T proposée par GNU AS

L'assembleur x86 est une ISA développée historiquement par Intel pour son CPU 16-bit 8086 produit en 1978. Les générations suivantes de processeurs Intel et AMD sorties dans les années 80 (80286, 80386, ...) 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 64-bit 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, Core-i, etc).

Syntaxe Intel

```
main:
    push    ebp
    mov     ebp, esp
    mov     eax, 0
    pop     ebp
    ret
```

Syntaxe AT&T

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

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, ...). 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 placés 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 qui suit est une constante (mode d'adressage immédiat) ;
- suffixe d'instruction : précise la taille en octet des données manipulées : **b**=byte=1octet, **s**=short=2octets, **l**=long=4octets et **q**=quad=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 à la case mémoire pointée par l'adresse BP, moins 4 octets.

```
.global main

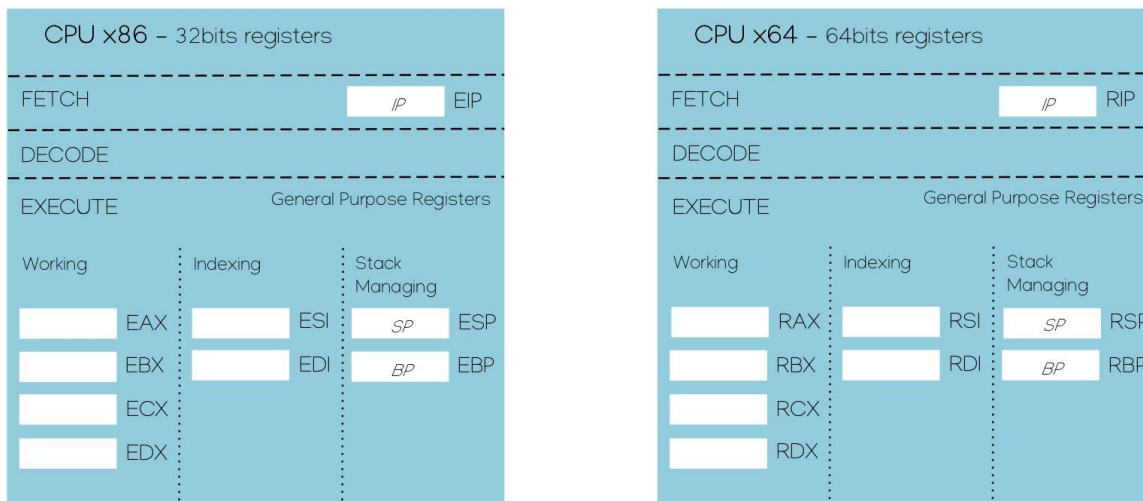
.text
main:
    ...
    ret
```

Un programme assembleur intégrera également certaines directives d'assemblage¹¹ préfixées par un point (**.global**, **.text**, **.macro**, ...). 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**), ...

¹¹ https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html

I.3. Registres CPU x86-64

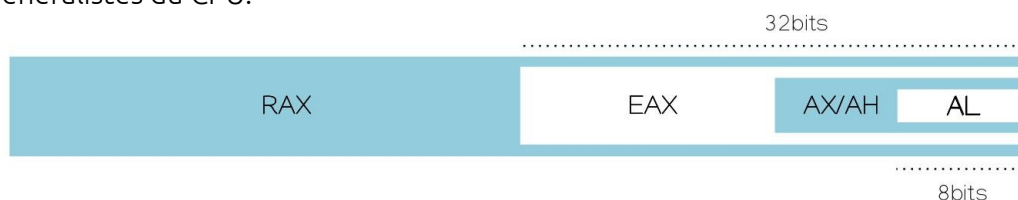
Les schémas ci-dessous présentent les principaux registres d'usages généralistes présents dans les CPU d'architectures compatibles x86 (32-bit) et x64 (64-bit). Rappelons que ces architectures restent rétrocompatibles avec le CPU 16-bit 8086 datant de 1978. Ces registres sont présents dans chaque CPU d'une machine multicœurs (un cœur est l'ensemble CPU, MMU et caches locaux).



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, ...) : registre contenant les différents *flags* d'états associés aux unités d'exécution arithmétiques et logiques (*carry, zero, sign, ...*). Utilisé notamment afin d'implémenter des sauts conditionnels (*if, else if, while, for, ...*) ;
- **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 ;
- **Registres vectoriels** (XMM0 à XMM15 128-bit extension ISA SSE, YMM0 à YMM15 256-bit extension ISA AVX et ZMM0 à ZMM15 512-bit 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).

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 16-bit et 8-bit 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 8-16-32-64-bit imbriquées les unes dans les autres sur architectures x86-64 (respectivement AL 8-bit, AX 16-bit, EAX 32-bit et RAX 64-bit). L'exemple donné ci-dessous sur le registre 64bits RAX peut être étendu à tous les registres de travail généralistes du CPU.



II. Instructions arithmétiques et logiques

```

--- crt0.s ---

.global _start
.text
_start:
    push    %ebp
    mov     %esp, %ebp
    call    main
    mov     $1, %eax
    int     $0x80

```

```

--- logic_arithmetic.s ---

.global main
.text
main:
    xor     %eax, %eax
    mov     $9, %ebx
    add     %ebx, %eax
    sub     $2, %eax
    ret

```

Ouvrez un terminal et placez-vous dans le répertoire de travail `disco/asm/`. Assemblez les fichiers `logic_arithmetic.s` et `disco/toolchain/build/startup/crt0.s` (fichier de startup minimal). Puis réalisez l'édition des liens en utilisant le linker script minimal utilisé dans le premier chapitre de TP.

```

as --32 -g ../toolchain/build/startup/crt0.s -o obj/crt0.o
as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o
ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o -o bin/logic_arithmetic

```

Affichez et analysez le programme assembleur `logic_arithmetic.s`.

Quel traitement implémente l'instruction `XOR` ? Comment aurait-on pu écrire autrement ce même traitement ?

Des deux instructions implémentant un même traitement proposées à la question précédente (`XOR` vs `MOV $0`), laquelle offre une implémentation plus optimisée et pourquoi ?

Analyser le firmware pour vous aider.

```
objdump -S bin/logic_arithmetic
```

Quel est le contenu du registre `eax` à la fin de l'exécution de la fonction `main` ?

III. GNU Debugger

Nous allons maintenant découvrir quelques outils complémentaires pouvant vous aider à l'analyse et au développement d'un script assembleur. Nous allons découvrir quelques une des principales commandes de GDB, le débogueur ou *debugger* du projet GNU¹².

Commandes (raccourcis)	Descriptions (nom complet)
<code>l</code>	(<i>list</i>) Affiche le listing avec numéros de lignes du programme C
<code>la a</code>	(<i>layout assembly</i>) Affiche le listing assembleur du binaire désassemblé
<code>b label</code>	(<i>break</i>) place un point d'arrêt (<i>breakpoint</i>) sur un label connu
<code>b file.c:line_nb</code>	(<i>break</i>) place un point d'arrêt sur une ligne spécifique du programme C
<code>i b</code>	(<i>info break</i>) Liste tous les point d'arrêts du programme
<code>r</code>	(<i>run</i>) exécute le programme jusqu'au premier point d'arrêt
<code>c</code>	(<i>continue</i>) reprend l'exécution jusqu'au prochain point d'arrêt (ou la fin)
<code>s</code>	(<i>step</i>) Exécute l'instruction ASM suivante (pas à pas)
<code>ni</code>	(<i>next instruction</i>) Exécute l'instruction C suivante (pas à pas)
<code>p variable</code>	(<i>print</i>) Affiche le contenu d'une variable
<code>i reg</code>	(<i>info</i>) Affiche le contenu de tous les registres de l'architecture
<code>i reg names</code>	(<i>info</i>) Affiche le contenu des registres demandés
<code>x/NFb 0xaddress</code>	Examine N octets de la mémoire au format F spécifié (d ou x)
<code>q</code>	(<i>quit</i>) Quitter la session de debug courante
<code>more ...</code>	http://www.gdbtutorial.com/gdb_commands

Assemblez jusqu'à l'édition des liens incluse le fichier `logic_arithmetic.s`. Vous aurez peut-être remarqué l'option `-g` passée à GCC, qui lui demande d'intégrer des directives de *debug* au sein du code compilé.

```
as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o

ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o -o bin/logic_arithmetic
```

¹² <http://www.gdbtutorial.com/>

Lancez maintenant l'exécutable au travers du *debugger* GDB.

```
gdb ./bin/logic_arithmetic
(gdb) ... wait for gdb commands !
```

Une console de *debug* vient maintenant de s'ouvrir. GDB est un programme capable de prendre le contrôle sur d'autres programmes. Nous pouvons placer des points d'arrêts, puis analyser pas à pas l'exécution d'un programme (*dump* mémoire, trace d'exécution, contenu de registres CPU, etc). Un *debugger* s'utilise durant les phases de développement ou de déverminage d'un programme. Il propose des outils d'analyse élémentaires mais très puissants. À force de persévérance, aucun bug ne peut se cacher indéfiniment !

Suivez la séquence de commandes GDB proposée ci-dessous, et interprétez-la à l'aide du tableau de la page précédente.

```
(gdb) la a
(gdb) b _start
(gdb) r
(gdb) b main
(gdb) c
(gdb) s
(gdb) i reg eax
(gdb) s
(gdb) i reg eax ebx
(gdb) s
(gdb) i reg eax ebx
(gdb) s
(gdb) i reg eax ebx
(gdb) s
(gdb) s
... until end of program
(gdb) s
```

IV. Fonction de conversion entier vers ASCII

```

--- logic_arithmetic.s ---

.global main

.text
main:
    xor    %eax, %eax
    mov    $9, %ebx
    add    %ebx, %eax
    sub    $2, %eax
    call   itoa
    ret

```

```

--- itoa.s ---

.global itoa
.global tab

.data
tab:
    .zero    1
    .string  "\n"

.text
itoa:
    add     $48, %eax
    mov     %al, tab
    ret

```

Modifiez le fichier `logic_arithmetic.s` afin d'appeler la fonction `itoa` (*integer to string conversion*). Assemblez `logic_arithmetic.s`, `itoa.s` et `disco/toolchain/build/startup/crt0.s` (fichier de startup minimal) puis réalisez l'édition des liens du projet.

```

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

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

ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o obj/itoa.o -o bin/logic_arithmetic

```

Quelle taille fait le tableau d'adresse `tab` ? Précisez son contenu au démarrage du programme et proposez une écriture équivalente en langage C. Constatez qu'un label peut pointer sur du code (`_start`, `main` ou `itoa`) ou des données statiques (`tab`).

Que représente la constante décimale 48 et en quoi cela assure une conversion entier vers ASCII (uniquement pour un chiffre compris entre 0 et 9) ?

En utilisant GDB (mode pas à pas), vérifiez la valeur contenu dans `EAX` après conversion par la fonction `itoa` et avant la fin de la fonction `main` (instruction `RET`). Vérifiez également le contenu du tableau `tab` avant et après exécution de la fonction `itoa`.

```

(gdb) la a
(gdb) b main
(gdb) r
(gdb) x/3xb 0x<tab_address>
(gdb) s
... Go to and execute itoa function
(gdb) x/3xb (char*)&tab
(gdb) s

```

V. Fonction d'affichage printf

```
--- logic_arithmetic.s ---
.global main

.text
main:
    xor    %eax, %eax
    mov    $9,    %ebx
    add    %ebx, %eax
    sub    $2,    %eax
    call   itoa
    call   printf
    ret
```

```
--- printf.s ---
.global printf

.text
printf:
    mov    $3,    %edx
    mov    $tab, %ecx
    mov    $1,    %ebx
    mov    $4,    %eax
    int    $0x80
    ret
```

Modifiez le fichier `logic_arithmetic.s` afin d'appeler la fonction `printf`. Assemblez les fichiers `logic_arithmetic.s`, `itoa.s`, `printf.s` et `disco/toolchain/build/startup/crt0.s` (fichier de startup minimal) puis réalisez l'édition des liens du projet. Exécutez le binaire de sortie et analysez le projet assembleur en s'aidant de GDB.

```
as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o
as --32 -g printf.s -o obj/printf.o
ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o obj/itoa.o obj/printf.o -o bin/logic_arithmetic
./bin/logic_arithmetic
```

La fonction standard du langage C `printf` réalise trois traitements distincts. Une première étape optionnelle de conversion de valeurs typées aux formats entiers ou flottants vers une chaîne de caractères (`%d`, `%u`, `%lu`, `%llu`, `%f`, `%lf`, etc). La seconde étape consiste à construire une chaîne de caractères à transmettre vers la sortie standard du système `stdout` (shell courant ou autre sortie en mode texte). La dernière étape consiste à transmettre la chaîne de caractères construite au noyau du système chargé de l'envoyer vers le flux standard de sortie `stdout`. Nous allons nous intéresser à cette dernière étape durant cet exercice, en envoyant la chaîne de caractères précédemment construite par la fonction `itoa`.

L'instruction `int 0x80` implémente un appel système 32-bit x86¹³ permettant de donner la main au noyau Linux en lui passant des arguments par registres. Il s'agit d'une interruption logicielle (interrompt l'exécution synchrone d'un programme). Les registres utilisés en x86 (`EAX`, `EBX`, `ECX` et `EDX`) sont documentés et seront toujours les mêmes pour un appel système sur noyau Linux. Ces registres dépendent donc de la technologie de noyau (Linux, Hurd, XNU, Minix, NT, ...) sur laquelle est portée le système d'exploitation (distribution GNU/Linux, Mac OS X, Windows, ...). Observons les opérandes en x86 sous Linux :

- `EAX` : fixe la fonction noyau à exécuter et donc la nature de l'appel système. Fonction `write` dans notre cas
- `EBX` : Mode d'accès au fichier
- `ECX` : Pointeur vers la chaîne de caractères à transmettre, soit `tab` l'adresse du tableau statique `tab[3] = "?\n"` dans notre cas
- `EDX` : Taille en octet de la chaîne de caractères à transmettre, soit 3 dans notre cas

¹³ En assembleur 64-bit x64, l'appel système est implémenté par l'instruction `syscall` (et non `int 0x80`).

VI. Suite de Fibonacci

```
.global main

.macro
CONVERT_TO_ASCII_AND_PRINT
    mov    %eax,tmp_eax
    mov    %edx,tmp_edx
    call   itoa
    call   printf
    mov    tmp_eax,%eax
    mov    tmp_edx,%edx
.endm

.comm tmp_eax, 4
.comm tmp_edx, 4

.text
main:
    xor    %eax, %eax
    CONVERT_TO_ASCII_AND_PRINT
    mov    $1, %eax
    CONVERT_TO_ASCII_AND_PRINT
    xor    %esi, %esi
    xor    %edx, %edx
.L0:
    mov    %eax, %edi
    add    %esi, %eax
    mov    %edi, %esi
    CONVERT_TO_ASCII_AND_PRINT
    add    $1, %edx
    cmp    $5, %edx
    jb     .L0
    ret
```

Assemblez `fibonacci.s`, `itoa.s`, `printf.s` et `disco/toolchain/build/startup/crt0.s` (fichier de *startup* minimal) puis réalisez l'édition des liens du projet. Exécutez le binaire de sortie et analysez le projet assembleur en vous aidant de GDB.

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

```
ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/fibonacci.o obj/itoa.o obj/printf.o -o bin/fibonacci
```

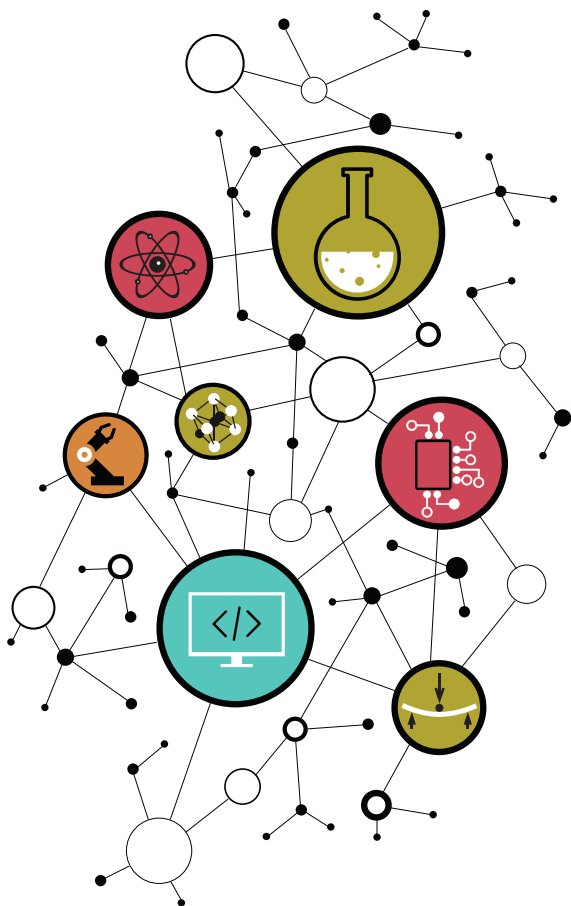
```
./bin/fibonacci
```

Pourquoi être passé par une sauvegarde puis restitution de contexte vers deux variables non initialisées (`.comm`) `tmp_eax` et `tmp_edx` avant d'appeler les fonctions `itoa` et `printf` ? Constatez que l'utilisation d'une macro allège la lecture du programme

En vous aidant de la documentation technique (*datasheet*) constructeur Intel présente dans `tp/doc/architectures-software-developer-manuals-vol2.pdf` à la page 474/1284, expliquez le fonctionnement de l'instruction `JB` (*Jump if Below*).

PARTIE 5

ALLOCATION AUTOMATIQUE ET SEGMENT DE PILE

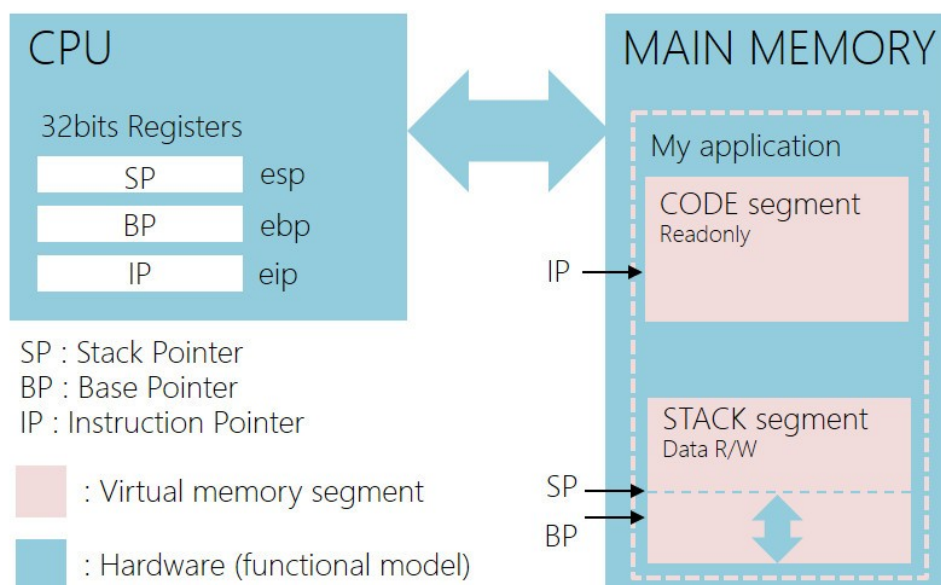


I. Segments de code et de pile

Nous avons vu avec les chapitres précédents que les instructions (sous forme binaire) et certaines données (variables globales, variables locales statiques et chaînes de caractères) sont **allouées statiquement à la compilation** et sont donc stockées dans le fichier binaire au format ELF (dans les sections `.text`, `.data`, `.bss` et `.rodata`). D'autres données (notamment les variables locales non-statiques) sont quant à elles **allouées dynamiquement pendant l'exécution** du programme. Dans ce chapitre, nous nous intéresserons aux allocations de ressources mémoire réalisées 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 contigue virtuelle allouée en mémoire principale par le noyau du système avant l'exécution d'un programme. Par exemple le segment **code** contient le code binaire des instructions du processus en cours d'exécution. Il est construit en mémoire principale au lancement du programme, à partir de la section `.text` du fichier exécutable. Il contient donc les mêmes instructions et la taille du segment de code dépend du nombre d'instructions. Le segment de **pile** (ou **stack**) qu'on étudiera dans ce chapitre ne contient que des données accessibles en lecture et en écriture. Sa taille est toujours fixe (8 MB par défaut sous système Linux). D'autres segments existent, nous les aborderons dans de futurs chapitres.

Ces segments sont spatialement séparés les uns des autres. 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.



Le segment de *stack* est référencé par deux pointeurs. Le **Stack Pointer SP** pointe toujours vers le sommet de pile (il évolue donc fréquemment), et le **Base Pointer BP** (aussi appelé *Frame Pointer*) indique le début de la zone mémoire utilisée par la fonction courante (il n'évolue donc qu'à chaque appel ou retour de fonction). Pour les CPU 32-bit x86, ces pointeurs sont respectivement stockés dans les registres CPU `esp` et `ebp`. Pour les CPU 64-bit, ces pointeurs sont respectivement stockés dans les registres CPU `rsp` et `rbp`.

Le **Instruction Pointer IP** (parfois appelé *Program Counter PC*) est un pointeur dédié au segment de *code*, désignant la prochaine instruction à exécuter. Sa valeur est sauvée dans le registre `eip` (pour CPU 32-bit x86) ou `rip` (pour CPU 64-bit x86).

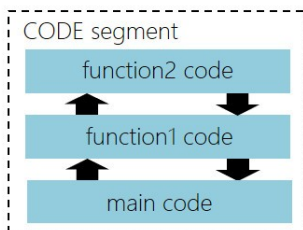
En langage C, comme dans beaucoup d'autres langages (C++, Java, D, etc), le point d'entrée d'une application est la fonction `main()`. De même, si nous réalisons des appels de fonctions imbriqués depuis le `main()` (cf. exemple ci-dessous) et si nous souhaitons revenir à la fonction principale de façon 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. À 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 8 MB potentiel d'espace mémoire de stockage.

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

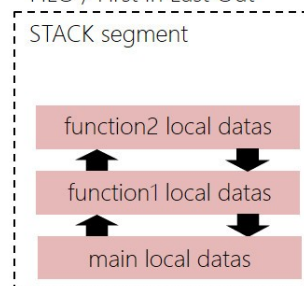
void function1 (void)
{
    /* local user code and datas */
    function 2();
}

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

Functions calls workflow
FILO / First In Last Out



Local datas workflow
FILO / First In Last Out



II. Fonction main

Pour suivre ce chapitre de TP, placez-vous dans le répertoire `disco/stack/`.

Compilez le fichier `main.c` vous arrêtant à la phase d'assemblage.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall -m32 main.c
```

Affichez le code assembleur généré, puis expliquez chacune des instructions en vous aidant de la documentation Intel ([tp/doc/architectures-software-developer-manuals-vol2.pdf](https://www.intel.com/content/www/us/en/tp/doc/architectures-software-developer-manuals-vol2.pdf)¹⁴).

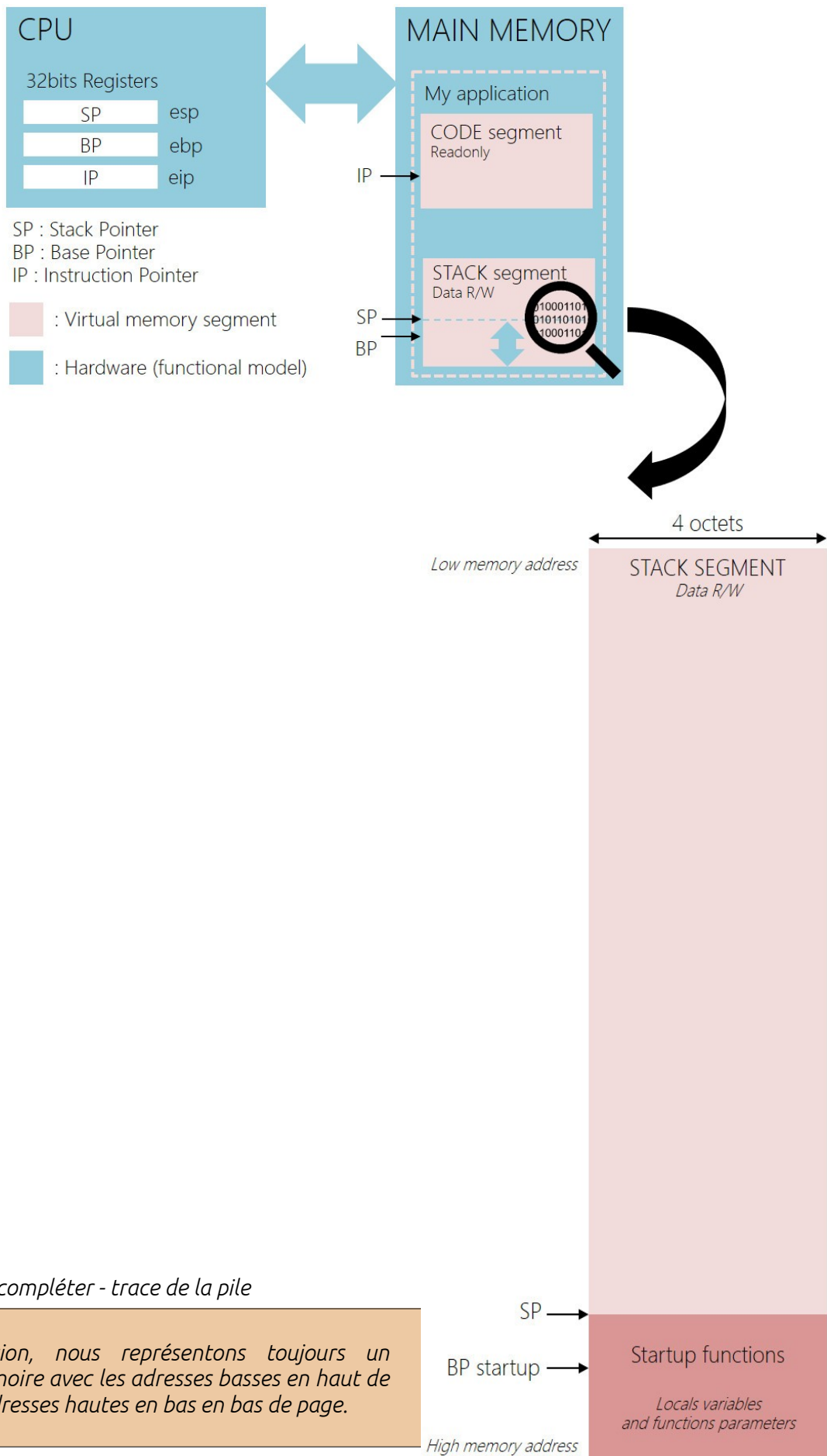
Complétez instruction par instruction le schéma de la pile sur la page suivante en précisant quel est son contenu suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`. Ne pas oublier qu'avant le début de notre programme, le code de la fonction de `startup` s'est exécuté.

Proposez une réécriture des instructions CISC-like (*Complex Instruction Set Computing*) `push` et `pop` à l'aide des instructions RISC-like (*Reduce Instruction Set Computing*) `sub`, `add` et `mov`.

À partir de maintenant, il faudra avoir deux niveaux de lecture d'un code assembleur.

- 1) Comprendre ce que fait littéralement l'instruction (ex : `pushl %ebp` copie le contenu du registre ebp en sommet de pile).
- 2) Comprendre la finalité de l'instruction (ex : `pushl %ebp` sauvegarde la valeur actuelle du registre ebp sur la pile).

¹⁴ Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C). Intel



III. Variables locales initialisées

Compilez le fichier `local_variable_init.c` en vous arrêtant à la phase d'assemblage.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall -m32 local_variable_init.c
```

Analysez le fichier assembleur généré et complétez (au crayon) le schéma de la pile sur la page suivante en précisant son contenu suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`.

Précisez les tailles ou empreintes mémoire des variables locales `a`, `b`, `c` et `d`.

Précisez les adresses relatives des variables locales `a`, `b`, `c` et `d`.

Combien faut-il de pointeurs, et donc de registres, afin d'adresser et de gérer un nombre quelconque de variables locales ?

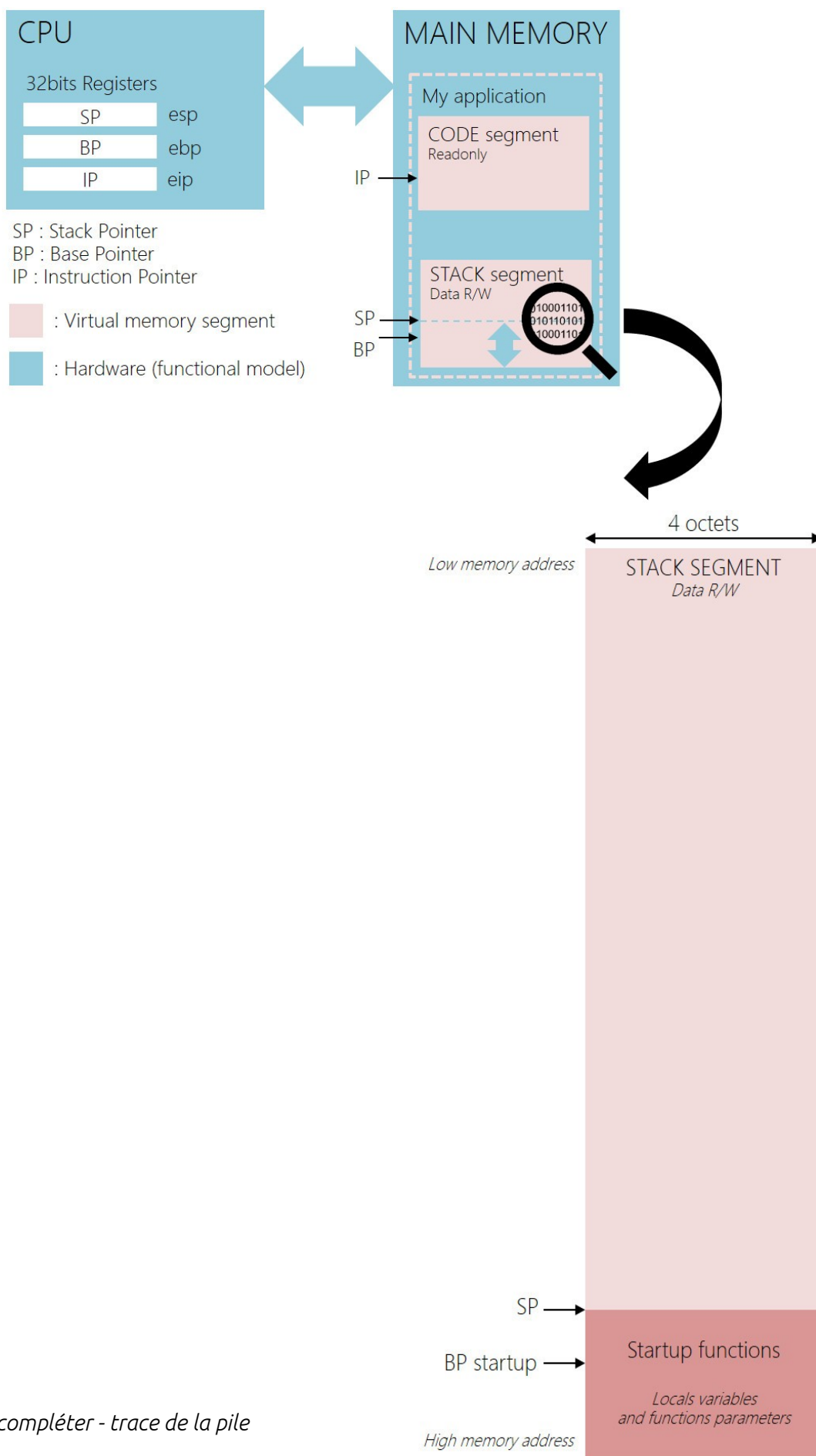


Schéma à compléter - trace de la pile

Dé-commentez le *cast* (transtypage) présent dans le programme, compilez et analysez le fichier assembleur de sortie. Analysez le résultat.

```
c = (int) a;
```

Qu'est-ce qu'une extension de signe en arithmétique entière signée Cà2 (Complément à 2) ? Aidez-vous de l'instruction **MOVSX** dans la documentation Intel.

Modifiez le fichier et qualifiez le type de la variable **a** de **const**. Compilez le programme puis interprétez le résultat. Que constatez-vous ?

Modifiez le fichier et qualifiez le type de la variable **c** de **const**. Compilez le programme puis interprétez le résultat. Que constatez-vous ?

Quel est le rôle du qualificateur de type **const** et donc son usage ?

IV. Variables locales non-initialisées

Ouvrez et compilez le fichier `local_variable_uninit.c` en vous arrêtant à la phase d'assemblage.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall -m32 local_variable_uninit.c
```

Analysez le fichier assembleur généré, que constatez-vous ?

Ajoutez le qualificateur de type `volatile` devant chaque déclaration, compilez le programme puis interprétez le résultat. Quel nouvelle instruction est apparue ? Quels sont les deux niveaux de lectures que vous accordez à cette instruction ?

Quel est le rôle de ce qualificateur de type `volatile` et donc son usage ?

Au regard du système cible (alchimie matérielle et logicielle), le compilateur est susceptible de réarranger des lectures et écritures sur des emplacements mémoire pour des raisons de performances. Les variables qualifiées de `volatile` ne sont pas soumises à ces optimisations (à la compilation comme à l'exécution). Ce mot clé peut notamment être utile en programmation multi-threads ou en programmation événementielle (interruptions, exceptions, etc). Le *qualifier* ou qualificateur de type `volatile` force le compilateur à n'opérer aucune optimisation sur les variables ainsi déclarées et laisse alors la porte ouverte à des modifications volatiles potentielles de la ressource par d'autres entités. Ceci est utilisé dès qu'il s'agit de manipuler des variables critiques comme des ressources partagées (*buffer* d'échanges, *flags*, périphériques, etc) et que nous sommes amenés à lever les options d'optimisation à la compilation.

Un qualificateur de type doit être vu comme une directive de compilation sciemment écrite par le développeur afin d'aiguiller voire forcer le compilateur à opérer des traitements privilégiés sur une variable. De façon générale, il s'agit de stratégies de durcissement ou robustification (verrouiller ou forcer un usage) voire d'optimisation (vitesse ou taille).

V. Appel et paramètres de fonction

À partir de maintenant, nous analyserons de l'assembleur 64-bit compatible pour architectures x64, en retirant l'option `-m32` à la compilation. Il s'agit de l'assembleur généré par défaut sur système GNU/Linux 64-bit porté sur machine 64-bit. Cela vous permettra d'apprécier les différences 32-bit/64-bit sur des programmes assembleurs élémentaires.

Ouvrez et compilez le fichier `function_parameters.c` en s'arrêtant à la phase d'assemblage.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall function_parameters.c
```

Analysez le fichier assembleur et complétez instruction par instruction le schéma de la pile sur la page suivante. Précisez son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`.

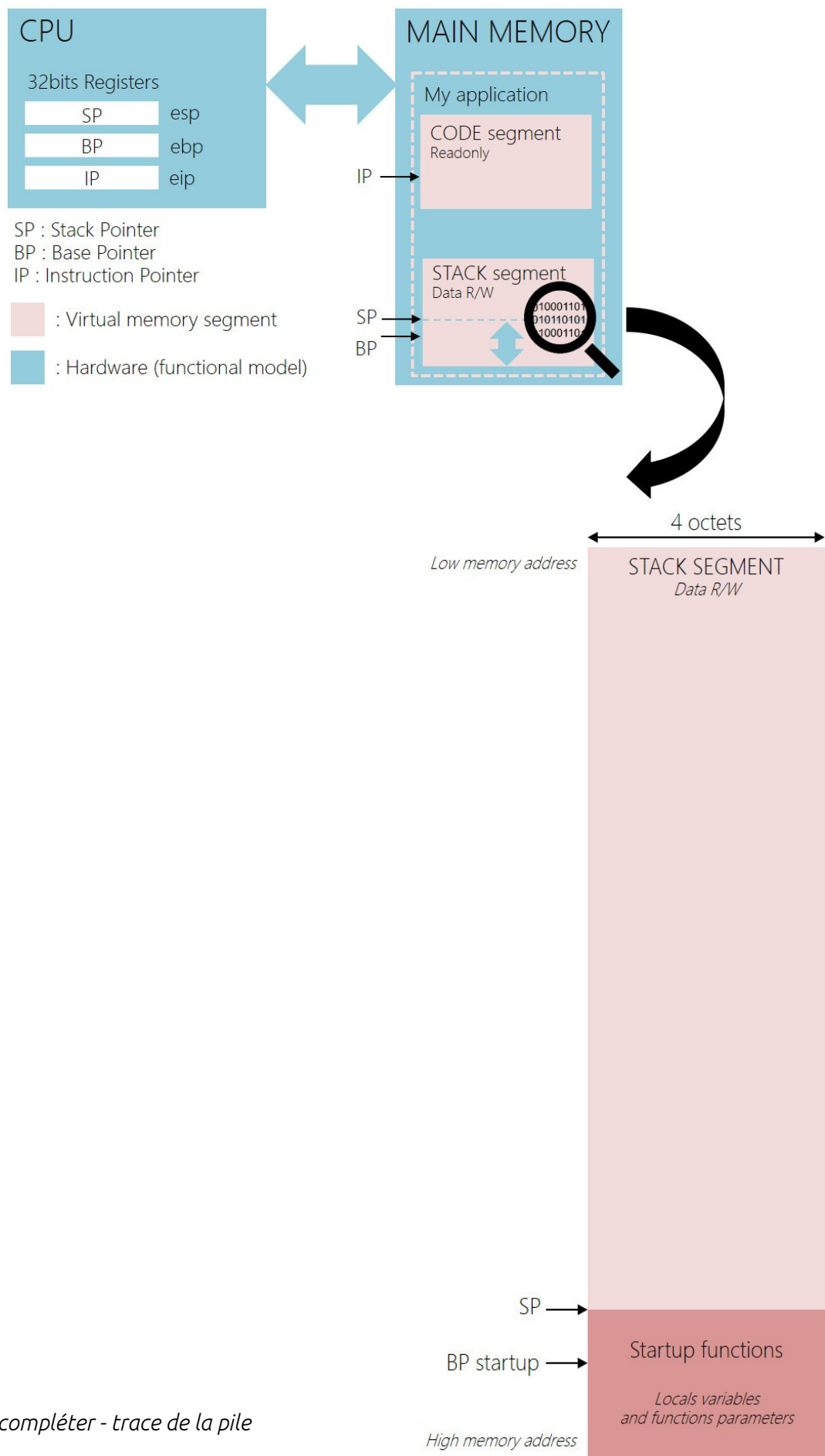
L'instruction `call` réalise un appel de fonction : elle stocke l'adresse de retour sur la pile et effectue un saut vers le code de la fonction appelée.

L'adresse de retour est celle de l'instruction située spatialement après le `call` en mémoire programme. Or pendant l'exécution du programme le *Instruction Pointer* IP (dont la valeur est stockée dans le registre `eip/rip`) contient l'adresse de la future instruction à exécuter. C'est à dire qu'au moment de l'exécution du `call`, le registre `eip/rip` contient l'adresse de retour de fonction. Empiler cette valeur en sommet de pile permet de pouvoir revenir à cette instruction en fin de fonction appelée.

Pour effectuer un saut vers le code de la fonction appelée, l'adresse visée sera écrite dans le *Instruction Pointer*.

L'instruction `ret` présente dans la fonction appelée dépile l'adresse de retour précédemment sauvée et la restaure dans le registre d'instruction `eip/rip` du CPU. Observons ci-dessous une réécriture en pseudo-code RISC des instructions `call` et `ret` :

CALL	called_function_label	→	PUSH rip
			MOV called_function_label, rip
RET		→	POP rip



Pour le passage d'arguments de type entier, quels sont respectivement les 3 registres CPU utilisés par GCC pour passer les paramètres à une fonction appelée ? Notez qu'il s'agit d'une stratégie proposée par GCC, cela ne dépend pas directement du langage ou de l'architecture CPU.

Quel est par défaut le registre utilisé pour passer une valeur de retour entière ?

Quelles sont les adresses relatives des variables `ret_1` (variable locale à `function_1`) et `a_2` (variable locale à `function_2`) ? Pourquoi ne sont-elles pas spatialement au même emplacement mémoire sur la pile ?

Observez la définition de la fonction `function_2` utilisant la syntaxe K&R (Kernighan & Ritchie) originelle du langage C . Au final, qu'est-ce qu'un paramètre de fonction ?

VI. Fonction inline et optimisation

Nous allons profiter de ce dernier exercice d'analyse de gestion de la pile pour étudier un appel de fonction avec passage d'arguments par pointeur. De même, nous analyserons les effets de quelques optimisations réalisées par GCC à la compilation.

Ouvrez puis compilez le fichier `function_inlining.c` en s'arrêtant à la phase d'assemblage.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-  
protection=none -Wall function_inlining.c
```

Analysez le fichier assembleur généré. Complétez le schéma de la pile sur la page suivante en précisant son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`.

Dé-commentez le prototype de la fonction `swap` qui utilise la classe de stockage `register` pour la déclaration des paramètres de fonction, et commentez le prototype générique. Faites de même au niveau de la définition de fonction, et qualifiez également la variable `tmp` dans la fonction `swap` de `register` (à laisser jusqu'à la fin de l'exercice). Compilez et analysez le code assembleur de la fonction `swap`. Quel est le rôle et l'intérêt de cette classe de stockage ?

```
//void swap(int* pt_a, int* pt_b);  
void swap(register int* pt_a, register int* pt_b);  
//inline void swap(int* pt_a, int* pt_b) __attribute__((always_inline));
```

Le mot clé `register` est une classe de stockage demandant (sans l'imposer) à la chaîne de compilation de manipuler les variables ainsi qualifiées par registre CPU et non en mémoire principale par la pile. Ce qualificatif ne peut-être géré que si les ressources matérielles le permettent (nombre de registres disponibles). Il s'agit d'un mécanisme simple d'optimisation permettant de limiter légèrement l'empreinte mémoire d'un programme mais pouvant augmenter significativement ses performances en évitant des lectures/écritures avec la pile présente en mémoire principale (technologie lente de transfert DDR sur stockage DRAM en comparaison aux registres en technologie SRAM travaillant à la même fréquence que le CPU).

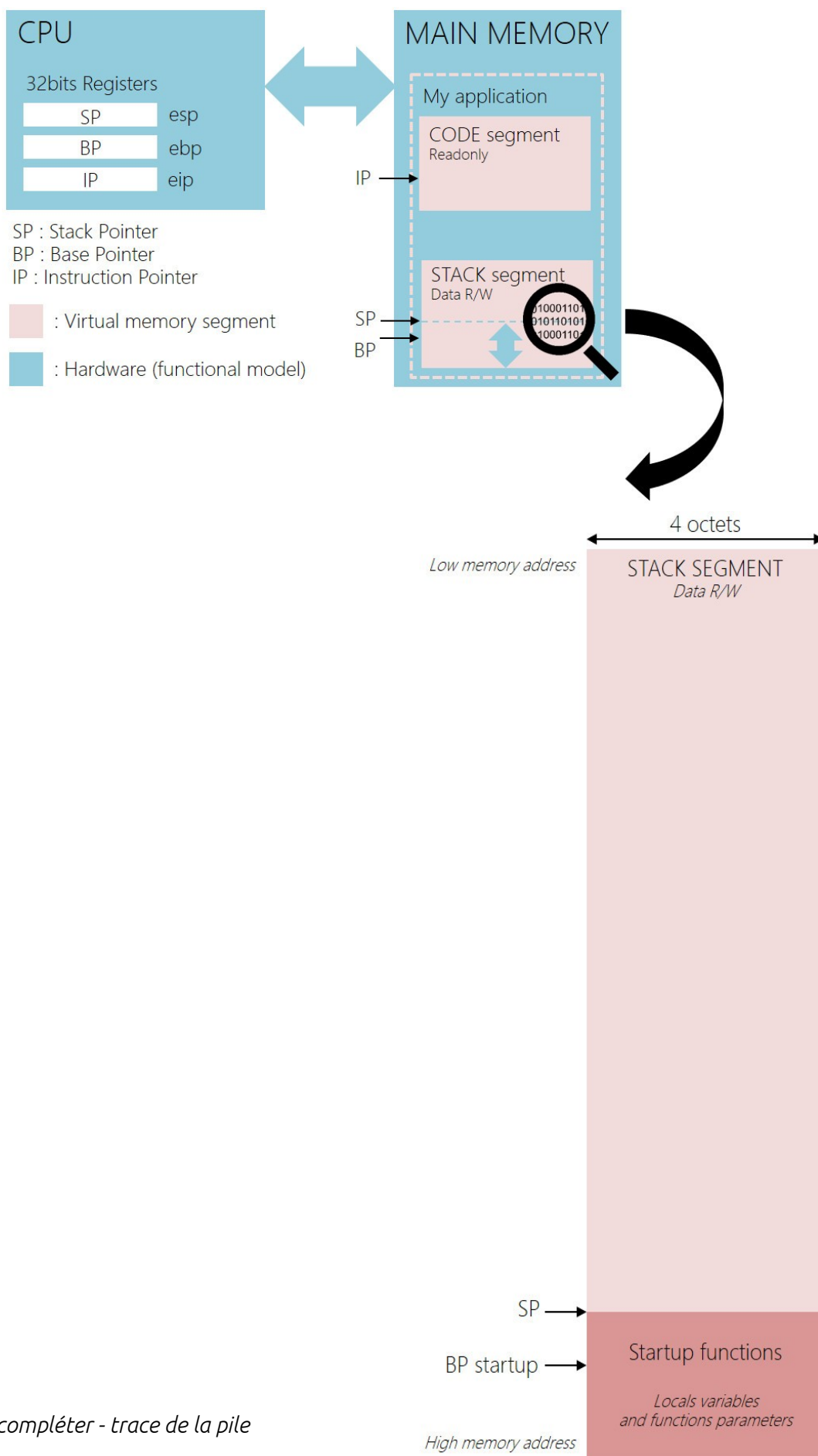


Schéma à compléter - trace de la pile

Dé-commentez le prototype de la fonction `swap` qualifiée de `inline` (seulement à la déclaration) et commentez les prototypes génériques.

```
//void swap(int* pt_a, int* pt_b);
//void swap(register int* pt_a, register int* pt_b);
inline void swap(int* pt_a, int* pt_b) __attribute__((always_inline));
```

Compilez et analysez le code assembleur de la fonction `main`. Quel est le rôle du mot clé `inline` arrivé avec la norme C99 ?

Pourquoi le code de la fonction `swap` est-il toujours présent dans le programme assembleur et donc à terme dans le firmware alors que la fonction n'est plus appelée depuis le `main` ?

Dé-commentez le prototype de la fonction `swap` qualifiée de `inline` au niveau de la définition de la fonction `swap` et commentez les prototypes génériques. Analysez le code assembleur généré.

```
//void swap(int* pt_a, int* pt_b)
//void swap(register int* pt_a, register int* pt_b)
inline void swap(int* pt_a, int* pt_b)
{
    ...
}
```

Le mot clé `inline` demande au compilateur d'insérer le code d'une fonction appelée dans le code de l'appelant en retirant l'*overhead* d'appel de fonction (`call`, `push`, `pop`, `ret`, etc). Ce mot clé s'applique donc à des fonctions courtes et permet d'augmenter les performances d'un programme. Néanmoins, le code étant dupliqué, en cas d'appels multiples ceci peut impacter la taille du firmware.

Laissez votre code tel quel (fonction `swap()` déclarée et définie avec le qualificateur `inline`) et ajoutez l'option d'optimisation de niveau 1 (`-O1`) de GCC. Compilez et analysez la sortie. Observez les décisions radicales prises par GCC.

```
gcc -S -O1 -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-
protection=none -Wall function_inlining.c
```

Essayez le niveau maximal d'optimisation `-O3` et analysez le programme de sortie.

VII. Limites de la pile

Nous allons maintenant tester les limites de notre pile applicative. Placez-vous dans le répertoire `disco/except/`. Compilez le fichier `stack_overflow.c` puis exécutez-le.

```
gcc stack_overflow.c -o stack_overflow
./stack_overflow
```

Le segment de pile a débordé. Expliquez l'erreur indiquée.

Quelle est la taille par défaut de la pile associée à notre programme ? Quelle entité sur la machine fixe et impose la taille de la pile associée à un programme ?

Dé-commentez la section de code présente dans le programme. Compilez à nouveau le fichier `stack_overflow.c` puis exécutez-le. Quelle est la nouvelle taille de la pile associée à notre programme ? Analysez le code dé-commenté.

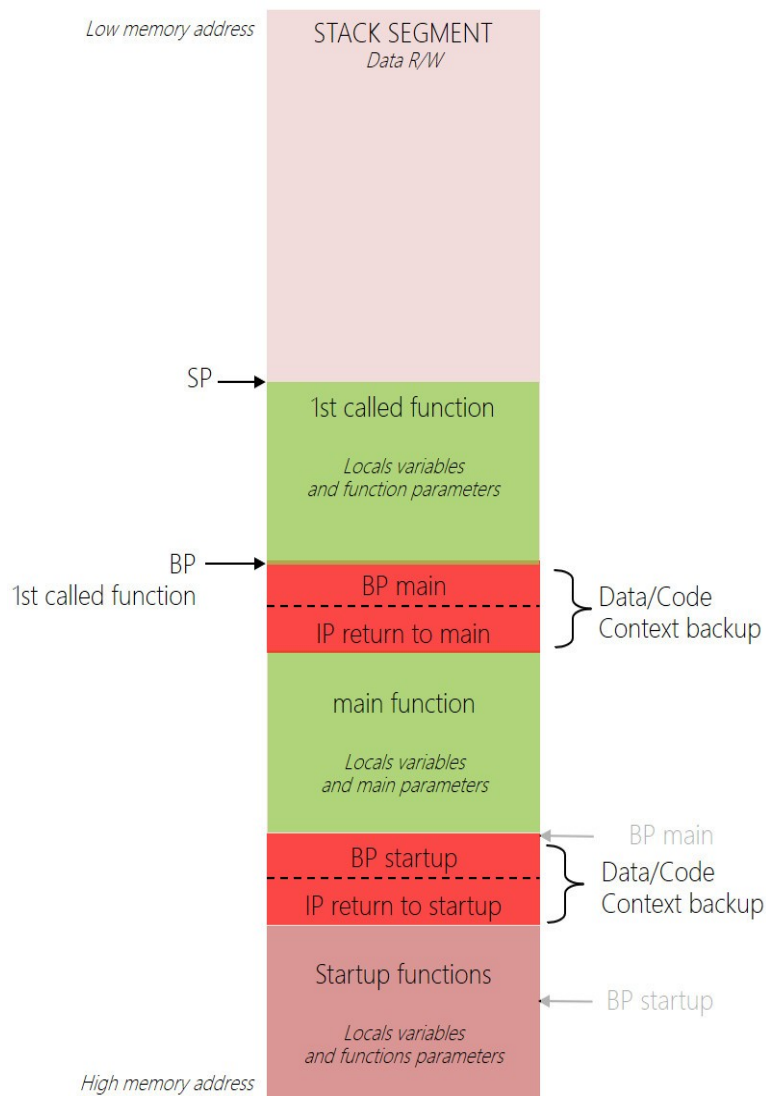
La fonction `setrlimit` (*set application resources limits*) réalise un appel système au noyau Linux en lui demandant d'allouer à notre programme un segment logique virtuel de pile plus large que celui alloué par défaut. Rappelons que Linux est le gestionnaire et le garant du bon fonctionnement des ressources matérielles de la machine ainsi que de ses limites physiques comme logiques. Nous appelons souvent un système d'exploitation un superviseur, sous entendu de l'ordinateur. Tant qu'une application n'est pas trop gourmande en ressource, le *kernel* Linux s'efforcera de répondre à ses requêtes.

VIII. Synthèse

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 alloués 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.

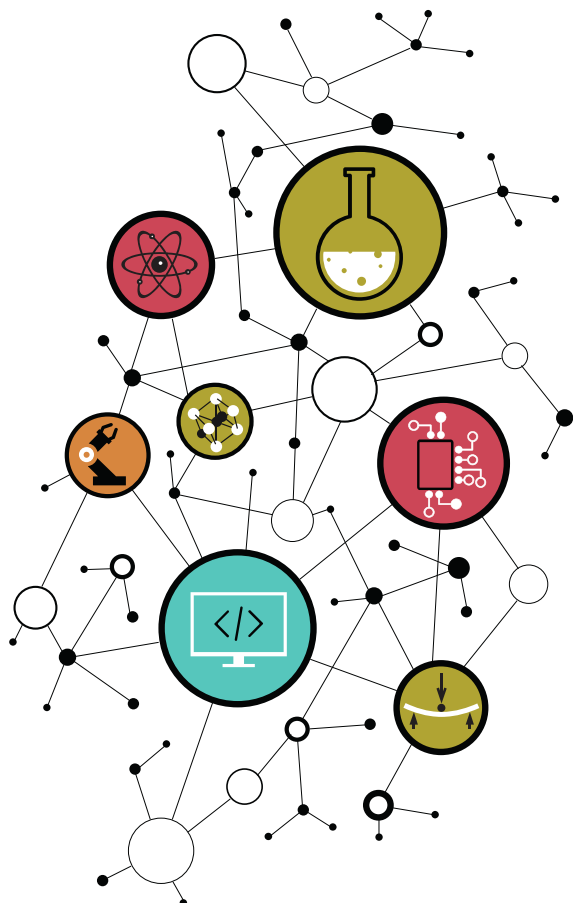
Une fois dans une fonction, les pointeurs BP et SP encapsulent le plus souvent la zone mémoire sur la pile comprenant les variables locales et paramètres propres à cette même fonction. Tant que nous restons dans cette fonction, BP ne sera pas modifié. De ce fait, toutes les variables locales et paramètres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



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 ou SSD).

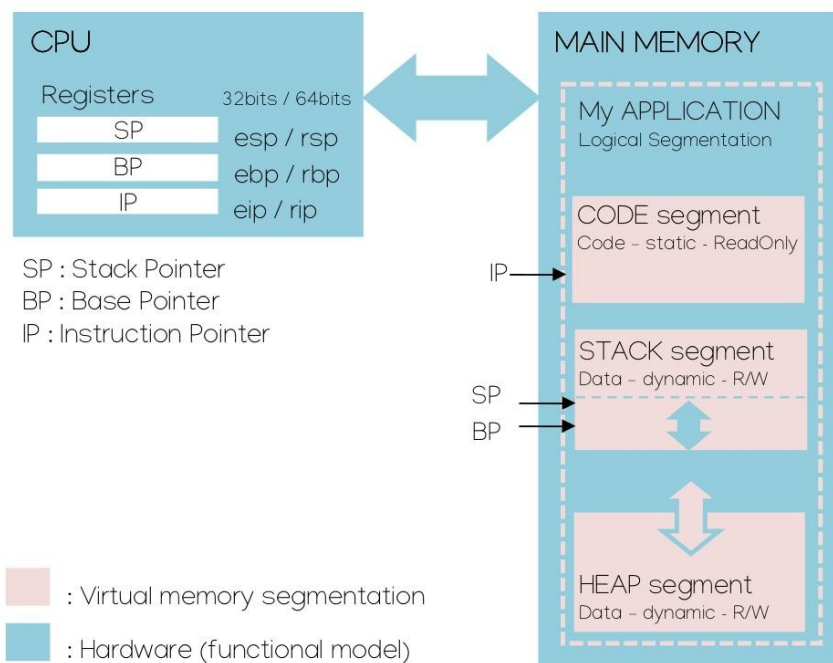
PARTIE 6

ALLOCATION DYNAMIQUE ET SEGMENT DE TAS



I. Segments de code, de pile et 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 du chapitre précédent 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 possède une taille extensible pouvant aller jusqu'aux limites physiques des ressources de stockage en mémoire principale de la machine (mémoire principale DDR SDRAM + potentiel SWAP système sur média de stockage de masse HDD/SSD). 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**, ou encore **new** en C++). Tant que l'application est active en mémoire principale, l'espace demandé restera alloué. Une fois la ressource mémoire utilisée, il est de la responsabilité du développeur de libérer les allocations précédentes avec appel de la fonction **free** (ou **delete** en C++). 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 sans MMU (MCU, DSP, ...). 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). Il faut 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 vers un bug certain de l'application.

II. Process en exécution

Placez-vous dans le répertoire `disco/heap/`. Compilez `heap.c` et exécutez le programme.

```
gcc -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none
-Wall -static heap.c -o heap

./heap

[CTRL]+[c] pour arrêter l'exécution du processus
```

Pendant que le processus est en cours d'exécution, ouvrez une nouvelle console pour récupérer le **PID (*Process Identifier*)** du-dit processus. Observez ensuite son *mapping* mémoire.

```
ps -a

cat /proc/<pid_of_heap_program>/maps
```

Quels éléments sont affichés par les trois `printf()` du programme `heap.c` ?

Dans quel segment mémoire est théoriquement situé chacun de ces éléments ?

Observez le *mapping* mémoire du processus et ses segments. Cela confirme-t-il les réponses précédentes ?

Quelle est la taille du segment de pile ? Quelle est la taille du segment de tas ? Quelle est la taille du segment de code (seul segment statique dont les propriétés permettent l'exécution, propriétés `r-x-`) ? Chaque segment en mémoire principale possède une taille multiple de 4 KB, la taille d'une page gérée par l'unité de pagination MMU.

Compiler le fichier `heap.c` en s'arrêtant à l'assemblage. Analysez le programme assembleur généré. À ce stade de l'enseignement, vous devez pouvoir être capable d'analyser des script assembleur de ce type.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-
protection=none -Wall heap.c
```

III. Fuites mémoire

Les mauvaises gestions de la mémoire représentent un très (très) grand nombre de sources d'erreurs de logiciels en cours d'exécution. Pour limiter ces erreurs il est nécessaire que le développeur ne cause aucune **fuite mémoire** en s'assurant de la libération des données qui ont été allouées mais qui ne sont plus nécessaires. Un outil indispensable pour tout développeur est **valgrind**, installable sur n'importe quelle distribution (déjà installé sur les machines de l'école).

Compilez le fichier **memory_leak.c**, exécutez-le une première fois et constatez qu'il n'y a pas de message d'erreur lors de l'exécution. Exécutez l'exécutable cette fois-ci par l'intermédiaire de **valgrind** et observez le résultat.

```
gcc memory_leak.c -o memory_leak
./memory_leak

valgrind --leak-check=full ./memory_leak
```

Qu'indique la sortie de **valgrind** ? Quelle est la cause des fuites mémoire ?

Modifiez le fichier **memory_leak.c**, recompilez et relancez l'exécution jusqu'à résolution complète des fuites mémoires.

Compilez le fichier **memory_leak_in_array.c**, exécutez-le une première fois et constatez qu'il n'y a pas de message d'erreur lors de l'exécution. Puis ré-exécutez au travers de **valgrind**.

```
gcc memory_leak_in_array.c -o memory_leak_in_array
./memory_leak_in_array
valgrind --leak-check=full ./memory_leak_in_array
```

Qu'indique la sortie de **valgrind** ? Quelle est la cause des fuites mémoire ?

Modifiez le fichier **memory_leak_in_array.c**, recompilez et relancez l'exécution jusqu'à résolution complète des fuites mémoires.

Même si l'exécution du processus ne provoque pas directement d'erreur, les fuites mémoires répétées (en quantité par un processus, et ce sur plusieurs processus) mèneront inévitablement à un bug. Un exemple très connu est le bug de la sonde spatiale Mars Pathfinder (1997), dont les fuites mémoires bloquaient les tâches critiques provoquant un redémarrage en boucle du système. Un bug à plusieurs millions de Dollars, corrigé à distance.

Valgrind est un outil puissant d'analyse, permettant de cibler les données perdues (allouées mais non libérées à la fin de l'exécution du processus). Il indique notamment les fonctions en cause ainsi que la quantité de données perdues (nombres et tailles des blocs).

IV. Limites du tas

Pour étudier les limites du segment de tas, placez-vous dans le répertoire `disco/except/`.

Ouvrez le moniteur système (ou *system monitor*) et observez l'onglet « *Resources* » pendant l'exécution du programme.

Compilez le fichier `heap_overflow.c` jusqu'à l'édition des liens incluse et l'exécutez-le.

```
gcc heap_overflow.c -o heap_overflow
./heap_overflow
```

Analysez le programme et observez les limites du tas. Comparez aux informations proposées par le système.

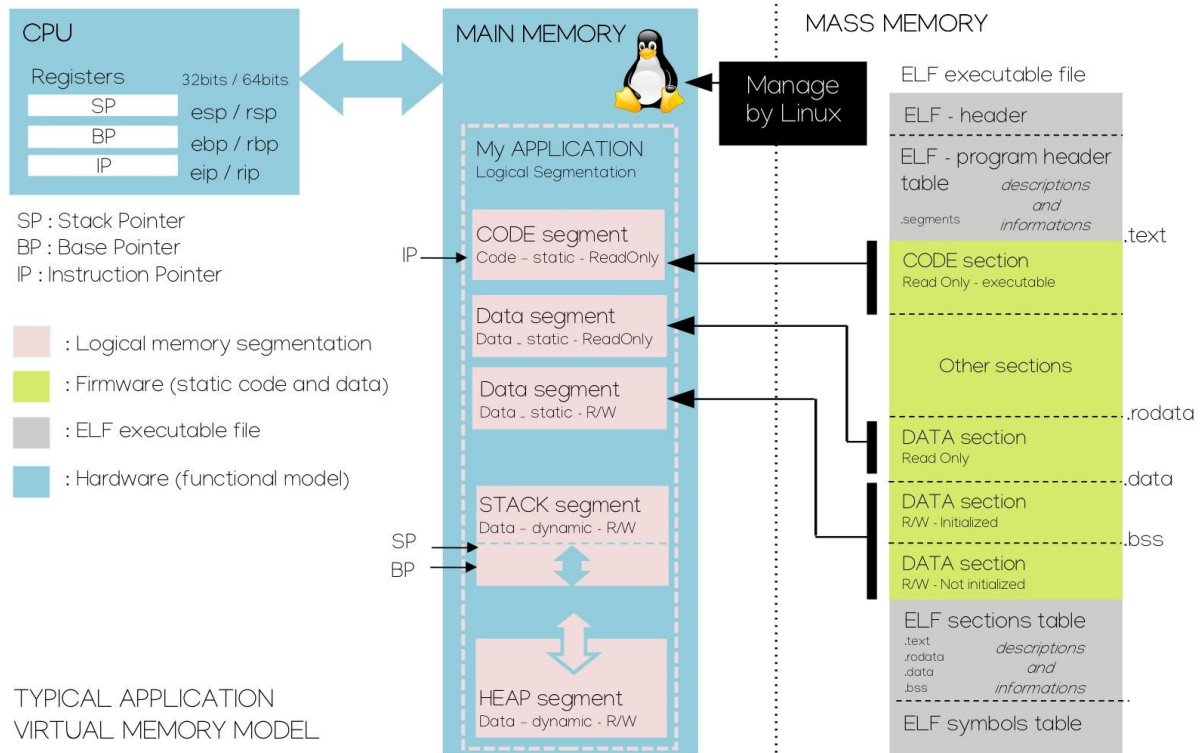
```
free
cat /proc/meminfo
```

La capacité du segment de tas est limité par la **mémoire principale** du système. Cette mémoire principale inclut évidemment la **RAM**, mais également le **swap**. Ce dernier est une partition d'échange présente sur le support de stockage de masse (HDD ou SSD), qui fait office d'extension de la RAM. Ainsi quand la RAM est pleine, le *swap* entre en jeu pour mettre à disposition une portion supplémentaire d'espace. L'avantage est donc d'étendre la quantité maximale de données allouables, mais avec en contrepartie de piètres performances en terme de débit des données manipulées.

Le segment de tas peut donc s'étendre au maximum sur toute la RAM et tout le *swap*, en prenant en compte ce qui est déjà alloué sur la mémoire principale (système d'exploitation, autres processus, autres segments du processus, ...).

V. Synthèse globale sur les stratégies d'allocations

Le schéma ci-dessous rappelle 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, ...) est chargée de traduire le programme d'un langage source (C, C++, ...) en langage machine binaire (x86, x64, ARM, MIPS, ...). Le format de fichier standard d'encapsulation de firmware sur système Unix-like est le format **ELF**. Le *firmware* est découpé en **sections**, des zones logiques séparant l'information (code et données) en partie de même natures et propriétés (code, donnée, lecture seule, lecture/écriture, exécutable, ...).

Allocation des segments et chargement en mémoire par le noyau

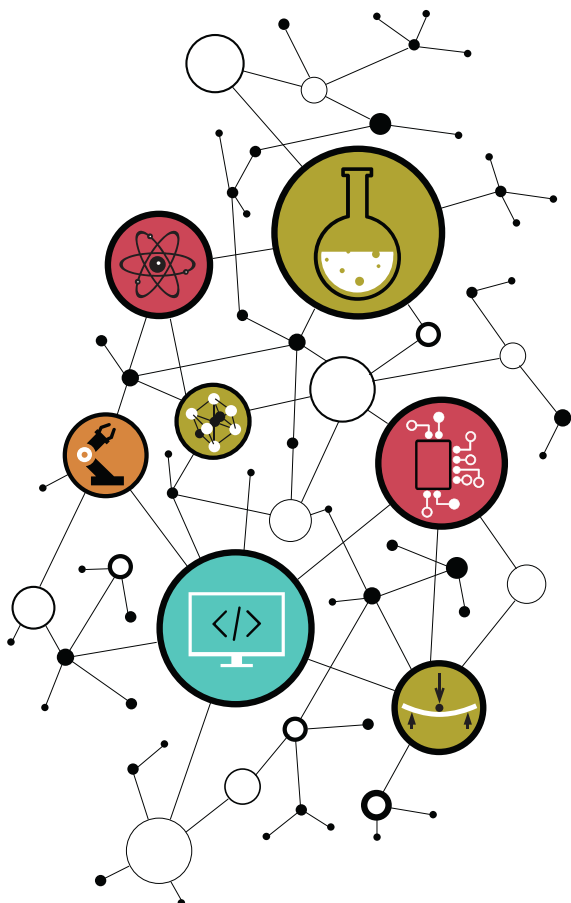
Si l'utilisateur demande l'exécution d'un programme, le noyau analyse alors le contenu du fichier exécutable (application à exécuter) grâce aux différents en-têtes et tables du format ELF. Il alloue alors des segments mémoire logiques contigus (virtualisation) ne pouvant se chevaucher (*Virtual Memory 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) les sections statiques du firmware vers les segments associés en mémoire principale (DDR_x 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 un *Segmentation fault* (*core dumped*) sera retourné par le système et l'application sera retirée (*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 allocations dynamiques sur tas. Chaque type d'allocation possède un segment mémoire dédié.

PARTIE 7

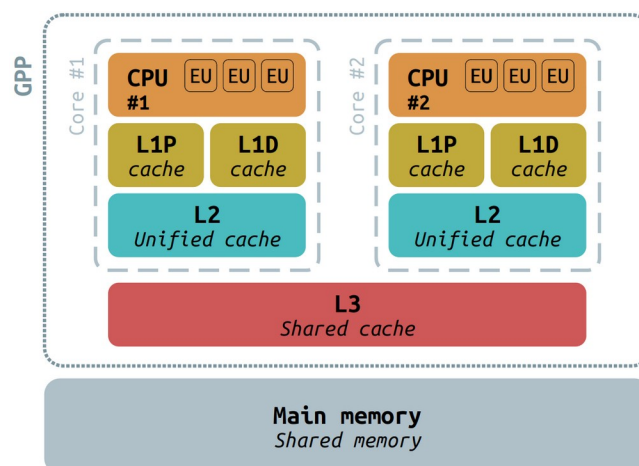
MÉMOIRES CACHE



I. Principes de localité

Les **mémoires cache** (ou parfois simplement **caches**) ne sont pas des zones mémoires accessibles par le système d'exploitation, car l'OS (ou le processus faisant une demande d'accès mémoire) n'a à sa disposition que l'espace de la mémoire principale constituée de la RAM et voire du swap.

Les caches se trouvent physiquement à l'intérieur du processeur et constituent une technique d'**accélération matérielle**, ou plus précisément d'accélération du temps d'accès à l'information en **dupliquant** ces informations au plus proche du CPU. Plus les informations sont spatialement proches des unités de traitement (EU, *Execution Unit*) du CPU, plus l'accès à ces informations est rapide. À ce titre on distingue plusieurs niveaux de cache, du plus lent (mais de plus grande capacité) au plus rapide (mais de capacité moindre) : la cache L3 propre au processeur, la cache L2 propre à chaque cœur (un cœur étant l'association d'un CPU et de ses caches attitrées) et les caches L1D et L1P (pour les *Data* et le *Program* respectivement).



Ces mémoires cache s'encapsulent : la L3 contenant une copie d'une portion de RAM, la L2 contenant une copie d'une portion de L3, la L1 contenant une copie d'une portion de L2. Si l'information demandée par le CPU est présente en cache L1, bingo : c'est un **cache hit** et le CPU dispose instantanément de l'information voulue. Si l'information demandée par le CPU n'est pas présente en cache L1, c'est un **cache miss** : on va alors la chercher en L2 pour la recopier en L1, voire la chercher en L3 si absente en L2, voire en mémoire principale en dernier recours.

Les avantages d'utiliser du silicium supplémentaire dans un processeur afin de gagner en temps d'accès à l'information s'explique par deux principes de localité :

- localité temporelle : une information juste manipulée va sûrement être réutilisée sous peu
 - d'où l'intérêt de garder l'information proche du CPU
- localité spatiale : les prochaines données à manipuler sont sûrement proches de la dernière
 - d'où l'intérêt de copier les informations par « bloc », qu'on appelle une ligne

Bien que les mémoires caches sont censées améliorer la vitesse d'exécution d'un programme, nous verrons qu'elles peuvent également être sources de contre-performances.

Complétez le schéma ci-dessus en y apposant les caractéristiques de votre processeur :

lscpu

II. Cache hit et cache miss

Placez-vous dans le répertoire `disco/cache/`.

Nous allons ici exécuter quatre exécutables très similaires : tous allouent (puis libèrent) un grand tableau de données, tous écrivent dans ce tableau grâce à une boucle `for`, et tous mesurent le temps d'exécution de cette boucle `for`. Ainsi le nombre d'instructions exécutées par ces quatre programmes est quasiment identique.

Les différences se font en deux points :

- les programmes `cache_v1` et `cache_v3` ne travaillent que sur les 10 premiers échantillons du tableau tandis que `cache_v2` et `cache_v4` travaillent sur l'intégralité du tableau ;
- les programmes `cache_v1` et `cache_v2` réalisent un grand nombre d'itération avec deux instructions C dans la boucle `for`, tandis que `cache_v3` et `cache_v4` réalisent 1000 fois moins d'itérations tout en faisant 1000 fois plus d'instructions dans la boucle `for`.

Pour tous ces programmes, vous pouvez jouer sur la macro constante `ARRAY_SIZE` pour « ajuster » le temps d'exécution de sorte à ce qu'il soit raisonnable. En revanche vous devez utiliser la même valeur pour les quatre fichiers : pensez donc à la mettre à jour partout si vous la modifiez !

Compilez chaque programme, puis exécutez-les avec l'outil d'analyse de cache de `valgrind`.

```
gcc cache_v1.c -o cache_v1
valgrind --tool=cachegrind --cachegrind-out-file=/dev/null ./cache_v1
```

Qu'observez-vous au niveau des temps d'exécution des programmes ?

Note : la différence entre `cache_v1` et `cache_v2` est plus probante avec une grande valeur pour `ARRAY_SIZE` mais votre machine va sacrément morfler avec le programme `cache_v4` ...

Qu'observez-vous au niveau accès aux mémoires cache (notamment les *cache misses*) ?

Liez les résultats de `valgrind` avec les quantités d'instructions et de données manipulées par chaque programme.

III. Cohérence de cache et false sharing

Le programme `threads.c` crée deux **threads**, qui accéderont en lecture et écriture à un tableau partagé. À notre niveau on considère que les *threads* sont deux sous-processus, les processus et *threads* seront étudiés en S8 en *Systèmes d'Exploitation et Réseaux*. La seule chose à savoir ici est que les *threads* vont s'exécuter simultanément, chacun sur un CPU différent.

Les *threads* `thread_write_first_half` et `thread_write_second_half` se séparent le tableau en deux moitiés égales, tandis que les *threads* `thread_write_even` et `thread_write_odd` gèrent chacun un échantillon sur deux, respectivement les échantillons pairs et impairs.

Compilez et exécutez le programme `threads.c`.

```
gcc threads.c -lpthread -o threads
./threads
```

Comment expliquer l'écart de performances entre les deux versions, alors que la quantité de données traitées et le nombre d'instructions sont très similaires ?

Sur les processeurs d'architecture Intel x86 ou x64, la taille d'une ligne de cache est généralement de 64 octets. Dans notre exemple, cela correspond à 16 cases du tableau ($16 * \text{sizeof(int)} = 64$ octets).

Si le CPU0 copie 16 cases du tableau dans sa cache L1D et que le CPU1 copie 16 autres cases dans sa cache L1D, chaque CPU peut travailler indépendamment sur son lot de données. C'est grosso modo ce qui se passe avec les versions `first_half` et `second_half`.

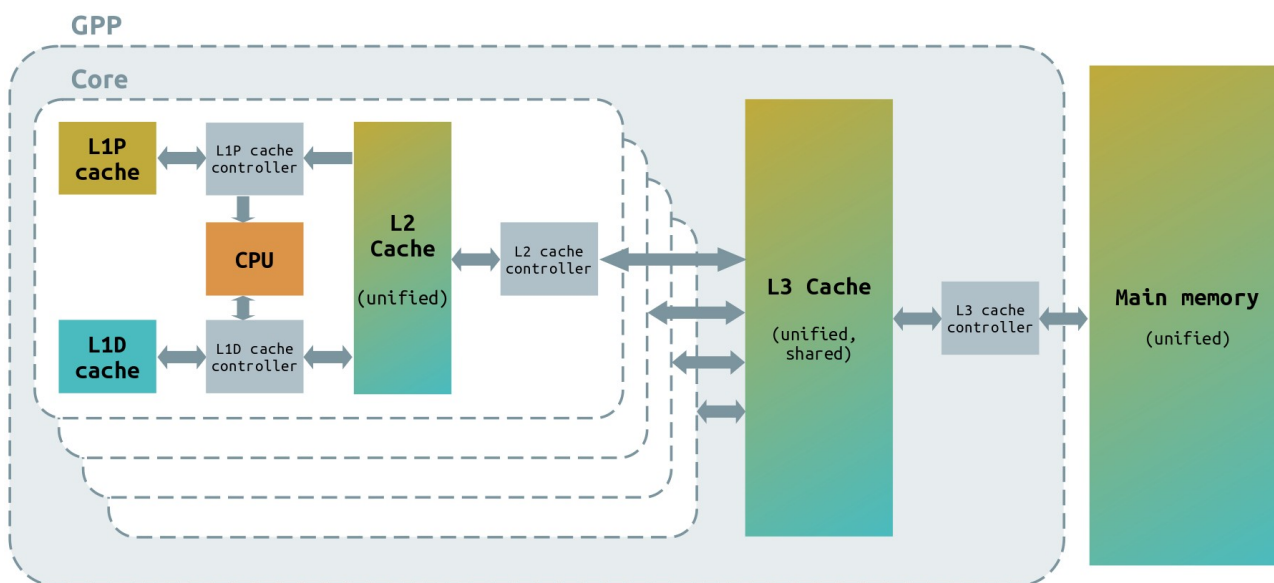
En revanche si le CPU0 copie 16 cases du tableau dans sa cache L1D et que le CPU1 copie les 16 mêmes cases du tableau dans sa cache L1D, alors chaque CPU doit vérifier si les copies présentes dans sa cache L1D sont à jour, puisqu'elles pourraient avoir été modifiées par l'autre CPU. C'est ce qui se produit avec les versions `even` et `odd`, où les données manipulées par les deux CPU sont **adjacentes** en mémoire.

Cet exercice met donc en avant un problème de **cohérence de cache**, ce qui signifie que les données contenues dans une cache L1 ne sont pas forcément les données à jour. Ce cas précis est même un cas particulier de cohérence appelé le **false sharing** (faux partage), où deux CPU accèdent en même temps à des données situées sur la même ligne de cache (puisque les informations en mémoire principales sont adjacentes).

IV. Synthèse

Bien que l'objectif des exercices précédents est de montrer les problèmes de performances liés aux mémoires cache (et donc ce qu'on observe sont les dégradations de performances des programmes), il faut quand même mettre ces résultats en perspective avec le taux de réussite d'accès à l'information.

En effet dans le premier exercice, même si le nombre de *cache misses* peut augmenter d'un facteur 1000, le taux de *cache misses* reste quant à lui très faible. Ces performances sont notamment dues à la politique de gestion des mémoires caches, assurant le chargement des données depuis la mémoire principale vers une cache et la restitution en mémoire principale par la suite. Ces politiques sont gérées par du matériel appelé contrôleur de cache et présent à chaque niveau de mémoire cache à l'intérieur du processeur.

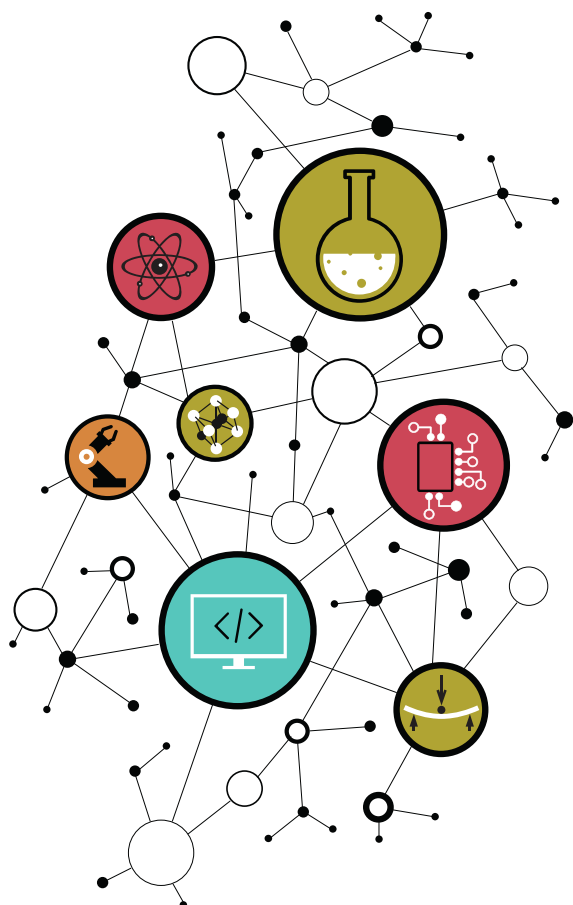


Le deuxième exercice montre lui que le choix de l'algorithme impactera sensiblement les performances du programme. Dans les faits les développeurs haut-niveau ou orientés application n'auront pas à se soucier de la cache en particulier, ni même du matériel en général. En revanche les développeurs d'algorithme (au sens traitement mathématiques appliqués sur une grande quantité de données) devront quand à eux s'intéresser aux couches matérielles du système (les mémoires cache, mais aussi les différentes unités d'exécution du CPU, les étages du pipeline, ...) puisque ces éléments seront fortement sollicités pendant l'exécution de l'algorithme. Un léger gain (ou au contraire un léger délai) s'appliquant à une donnée, se répercutera sur la totalité des données et donc générera un fort gain (ou forte dégradation) à l'ensemble du programme.

En bref, les caractéristiques des mémoires caches (temps d'accès, capacité, taille d'une ligne, politique de remplacements des lignes ...) sont autant de paramètres permettant d'améliorer les performances d'un algorithme en terme de temps d'exécution. Même si ce mécanisme matériel est transparent pour le développeur (au même titre que le *pipeline* par exemple), il faut dans certains cas être capable de descendre au niveau *hardware* pour comprendre ses forces et (parfois) ses faiblesses.

PARTIE 8

EXCEPTIONS MATÉRIELLES ET SIGNAUX UNIX

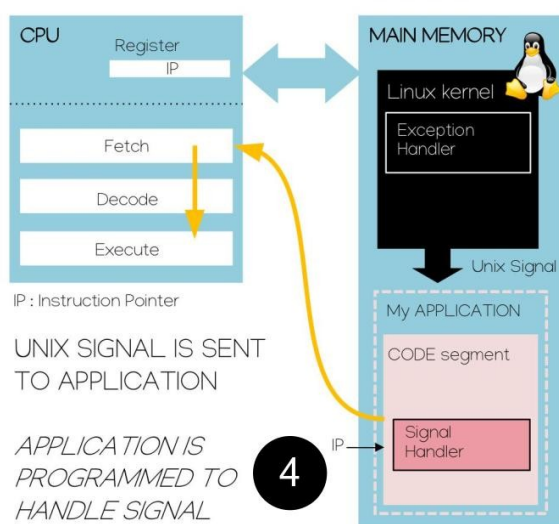
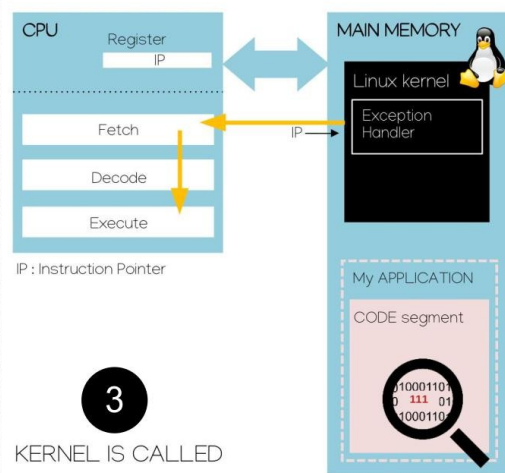
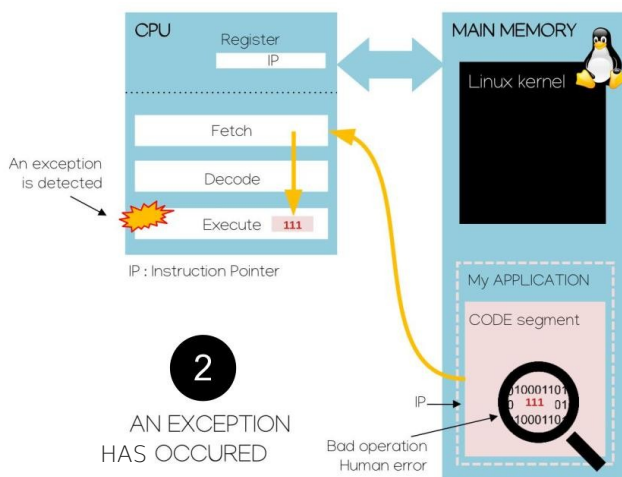
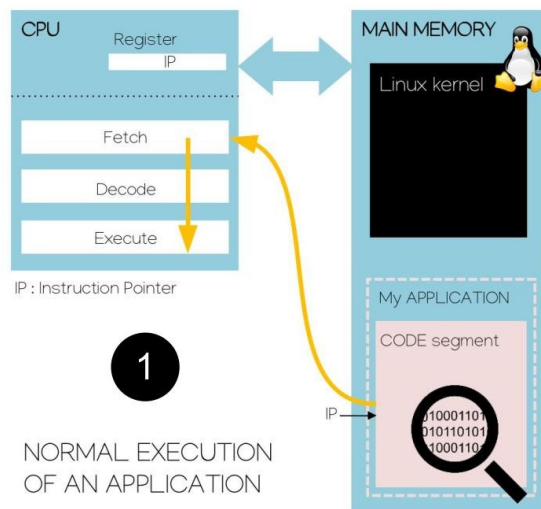


I. Exception vs Signal

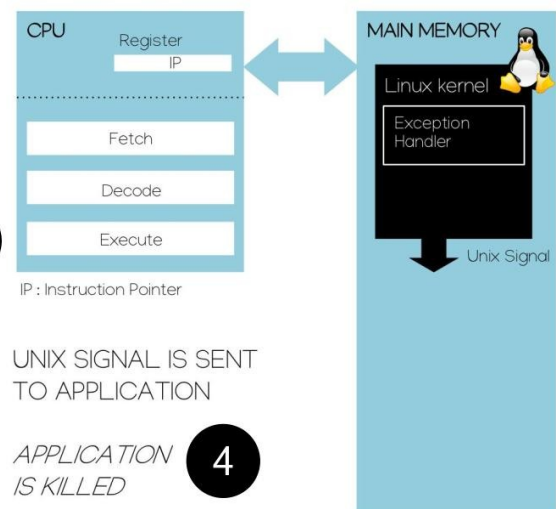
Une **exception matérielle** est un événement matériel synchrone généré par le CPU voire la MMU. Nous parlons d'événement synchrone au regard du fonctionnement d'un CPU dont les traitements restent synchronisés sur une référence d'horloge et non au regard de la probabilité d'occurrence. Une exception 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 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 arriver. Elles sont le plus souvent le fruit d'une erreur de programmation. Détaillons la séquence graphique présentée sur la page suivante :

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. Le pointeur d'instruction IP est alors redirigé (par lecture et exécution d'un tableau de vecteurs) vers une fonction noyau dédiée 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* Linux¹⁵). 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 ayant généré le défaut.
4. Le noyau du système vient d'envoyer un signal logiciel au processus. Deux cas de figure :
 - (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.
 - (schéma de gauche) Si le processus traite le signal Unix (le développeur a rédigé dans l'application un code spécifique au traitement de ces signaux Unix), l'application peut alors tenter d'acquiescer 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.

¹⁵ <https://www.kernel.org/>



or



II. Lecture seule

Placez-vous dans le répertoire `disco/except/`. Compiler le fichier `except_readonly.c` jusqu'à l'édition des liens incluse et exécutez le programme.

```
gcc except_readonly.c -o except_readonly
```

Interprétez et expliquez l'exception matérielle ainsi que le défaut logiciel à la racine de cette exception. Proposez un schéma commenté.

III. Pointeur nul

Compilez le fichier `except_null_pointer.c` jusqu'à l'édition des liens incluse et exécutez le programme.

```
gcc except_null_pointer.c -o except_null_pointer
```

Interprétez et expliquez l'exception matérielle ainsi que le défaut logiciel à la racine de cette exception. Proposez un schéma commenté.

Le 19 juillet 2024 une panne informatique d'envergure mondiale se produit. CrowdStrike, une société spécialisée en cybersécurité, publie une mise à jour d'un de leurs fichiers (comme ils le font plusieurs fois par jour) sur les machines Windows équipées de leur anti-virus.

Cette fois-ci, le fichier contenait une erreur. Même si les sources divergent au sujet de l'origine réelle du bug, toutes tendent vers le même constat : dans un des fichiers de CrowdStrike rédigés en C++, un pointeur `null` a été déréférencé.

Cette exception matérielle non traitée provoquait systématiquement le *kill* du processus au démarrage des ordinateurs, ce qui menait au fameux **Blue Screen Of Death** (BSOD) puis forçait un redémarrage en boucle du système.

Au final, près de 8.5 millions de machines Windows ont été touchées à travers le monde, annulant notamment plusieurs milliers de vols commerciaux sur plusieurs jours. Le préjudice pourrait atteindre près de 10 milliards de \$¹⁶, les plus touchés étant dans l'ordre les industries de santé, le secteur bancaire et les compagnies aériennes¹⁷.

¹⁶ <https://www.reuters.com/technology/insurers-face-business-interruption-claims-after-global-tech-outage-2024-07-19/>

¹⁷ <https://edition.cnn.com/2024/07/24/tech/crowdstrike-outage-cost-cause/index.html>

IV. Signal Unix

Une fois l'**exception matérielle** levée (*Null Pointer exception* dans l'exercice précédent), le noyau du système d'exploitation envoie un **Unix signal** (ici SIGSEGV, soit *Segmentation Violation*) au processus en cause. Le processus doit impérativement traiter ce signal, sans quoi il sera *kill*.

Compilez le fichier `signal_handler.c` jusqu'à l'édition des liens incluse et exécutez le programme.

```
gcc signal_handler.c -o signal_handler
```

Interpréter et expliquer l'exception matérielle ainsi que le défaut logiciel à la racine de cette exception.

Analysez le script assembleur du programme et précisez l'instruction ainsi que le registre à la source de l'exception.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall signal_handler.c
```

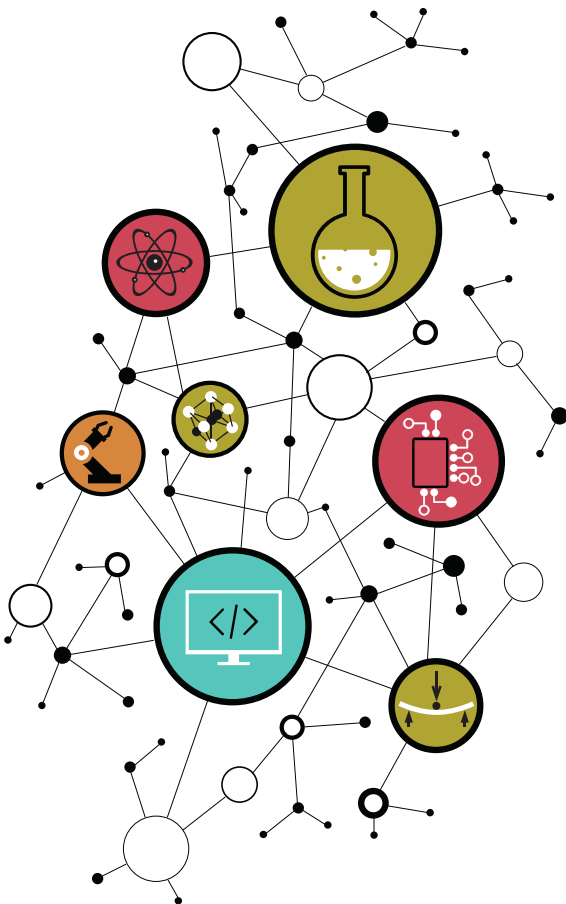
La séquence qui suit propose de remplacer le contenu du registre avec une valeur acceptable par le système pour la bonne exécution du processus.

Dé-commentez la section de code se trouvant dans la fonction `main()`. Ces instructions configurent la gestion des signaux système reçus par l'application. Compilez le fichier C jusqu'à l'édition des liens incluse, exécutez-le et analysez le fonctionnement du programme.

Dé-commentez la ligne de code dans la fonction `signal_handler()`. Celle ci propose une solution afin de corriger la source de l'exception (dans le `main`). Compilez, exécutez et analysez le résultat.

Voilà, vous venez de générer volontairement une exception (défaut de segment), de capter le signal Unix envoyé par le système, d'acquiescer l'erreur par remplacement du contenu d'un registre (RAX) et de redonner une chance à l'application de s'exécuter sans défaut.

PARTIE 9 HACKING



I. Les bidouilleurs

Le **hacking** est un terme aux significations multiples ayant évolué avec les années et les disciplines concernées. À l'origine il est utilisé par des radio-amateurs qui « *hackaient* » (au sens bricoler, bidouiller¹⁸) les systèmes afin de les améliorer. Il a ensuite été repris par des étudiants du MIT (*Massachusetts Institute of Technology*) pour qui l'étude des ordinateurs et réseaux informatiques se faisait au travers du démontage des matériels et désassemblage des logiciels, sous forme de jeux et de défis. Aujourd'hui, on parlerait de rétro-ingénierie (ou *retro-engineering*) même si ce terme apporte une notion moins artisanale à l'activité d'analyse des systèmes.

De nos jours le terme « *hacker* » est quelque peu galvaudé et aurait (du point de vue du grand public) tendance à désigner un pirate informatique au sens *black-hat* du terme. Un **black-hat** est une personne malveillante, agissant dans l'illégalité et s'introduisant dans un système d'informations (ordinateur, serveur, réseau, ...) afin d'en tirer un bénéfice (dégradations des services, obtentions d'informations sensibles, demandes de rançons, ...). Par opposition on trouvera le **white-hat** qui est un hacker travaillant en toute légalité, recherchant une vulnérabilité ou une faille de sorte à la rendre publique et voire proposer un correctif à l'éditeur. Entre les deux le **grey-hat** serait plutôt un *hacker* avec une éthique mais agissant dans l'illégalité. Il peut se rapprocher du *white-hat* au sens où il recherche des vulnérabilités et en informe l'éditeur concerné, ou il peut désigner le *hacktiviste* prônant une certaine idéologie (par exemple *Anonymous* militant pour la liberté d'expression et le respect de la vie privée).



Dans ce chapitre, nous allons nous introduire au monde du *hacking* en insérant un code malveillant dans un programme existant. Ce code est un **shellcode** : il aura pour objectif d'ouvrir un *shell* (un console), outil indispensable pour ensuite avoir un accès quasi-illimité à l'ensemble de la machine infectée. Ce *shellcode* sera inséré sur la pile afin de ne pas être détecté dans le code binaire du fichier exécutable.

Bien évidemment ceci n'est qu'une brève démonstration d'un monde bien plus vaste. De nombreuses autres solutions de *hacking* existent, comme la *ROP-chain* (*Return-Oriented Programming*) ou l'exploitation de vulnérabilités (*exploit*). Pour ce dernier cas, le site CVE (*Common Vulnerabilities and Exposures*)¹⁹ rassemble toutes les vulnérabilités publiques pour tout un tas d'application.

¹⁸ D'ailleurs le terme français-québécois associé à « *hacker* » est « bidouilleur ».

¹⁹ <https://cve.mitre.org/index.html>

II. Appels système

II.1. Fonction `execve()`

Vous n'avez peut-être pas réalisé qu'il est possible de lancer un fichier exécutable depuis un autre exécutable, mais vous l'avez fait à plusieurs occasions : en lançant un de vos programmes depuis le `bash` (qui est effectivement un programme exécutable), ou dans `gdb` et `valgrind` par exemple. Avec le programme qui suit, vous allez découvrir une des fonctions qui permet de lancer depuis votre programme un autre exécutable : `execve()`²⁰. Nous pourrions alors analyser le code assembleur généré afin de le réutiliser avec un objectif de *hacking*.

```
int main(void)
{
    char *argv[] = { "/bin/sh" , NULL };

    execve( argv[0], argv, NULL );

    exit(EXIT_FAILURE);
}
```

La fonction `execve()` permet de lancer n'importe quel exécutable depuis le processus courant. Pour cela il faut lui passer en paramètres :

1. l'adresse de la chaîne de caractères comprenant le chemin complet de l'exécutable ;
2. l'adresse du tableau contenant la liste des arguments (dont le nom de l'exécutable, puis la liste complète des arguments jusqu'à la valeur `NULL`) ;
3. un troisième argument (également un tableau) qui ne nous intéresse pas ici.

Dans l'exemple ci-dessus, le programme compilé démarrera puis lancera l'exécutable `/bin/sh`, qui est un *shell*. Au final vous allez donc lancer un *shell* dans un *shell*, ce qui n'a pas grand intérêt. En revanche, ouvrir un terminal depuis un logiciel frauduleux permet au *hacker* d'accéder à une grande partie du système cible, ce qui revient à avoir les clés de la machine !

Placez-vous dans le répertoire `disco/hack/`.

Compilez le fichier `shell.c` en ajoutant les options de *debug* (`-g`). L'option `-static` permettra d'embarquer le code binaire de la fonction `execve()` dans notre exécutable afin de l'étudier.

```
gcc -g -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-
protection=none -static shell.c -o bin/shell

./bin/shell
```

Que se passe-t-il à l'exécution du programme ?

²⁰ <https://www.man7.org/linux/man-pages/man2/execve.2.html>

Analysons le désassemblage du fichier exécutable binaire.

```
objdump -S bin/shell > misc/shell_disassembly.md
```

Dans le fichier `misc/shell_disassembly.md`, cherchez la fonction `<main>` et tracez le contenu de la pile jusqu'à l'exécution de `callq execve` inclus.

Quelles sont les deux valeurs stockées dans le contexte de la fonction `<main>`? À quoi correspondent-elles dans le programme écrit en C?

Vers quelle donnée pointe la première case du tableau `argv[0]`? Confirmez avec ceci :

```
objdump -s -j .rodata bin/shell | head
```



Que contiennent les registres `rdx`, `rsi` et `rdi`?

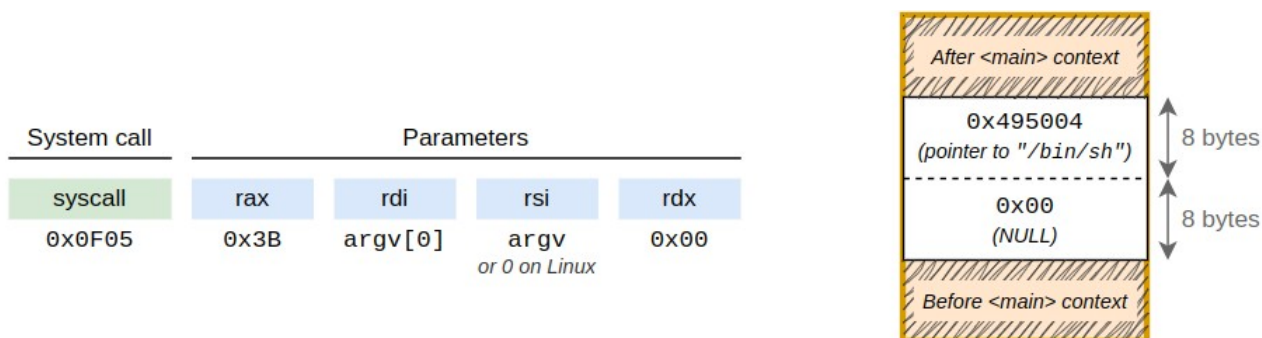
II.2. Instruction syscall

Cherchez dans le fichier `misc/shell_disassembly.md` la fonction `execve`.

Quel est le code binaire de l'instruction `syscall` ?

Quelle est la valeur du registre `rax` juste avant le `syscall` ?

L'instruction `syscall`²¹ (en x86_64) réalise un appel système : la main est alors donnée à l'OS qui effectue une action précise, définie par la valeur contenue dans le registre `rax`²². Dans notre cas, vous avez relevé que le registre `rax` contient la valeur `0x3b` (ou `59` en décimal) au moment de l'appel à `syscall`, ce qui correspond à l'appel système `sys_execve`. Avec les instructions placées dans le `main` avant l'appel à `syscall`, les registres `rdi`, `rsi` et `rdx` contiennent les arguments nécessaires à l'exécution de ce programme, comme indiqué en début de cet exercice.



Avec nos connaissances, il nous est possible d'écrire directement cet appel système (avec ses paramètres) en assembleur. Il « suffit » de placer la valeur `0x3b` dans le registre `rax`, de placer l'adresse de la chaîne de caractère `"/bin/sh"` dans le registre `rdi`, et de mettre les registres `rsi` et `rdx` à `0`. C'est le but de l'exercice page suivante.

²¹ Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C) ; p. 982.

²² https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

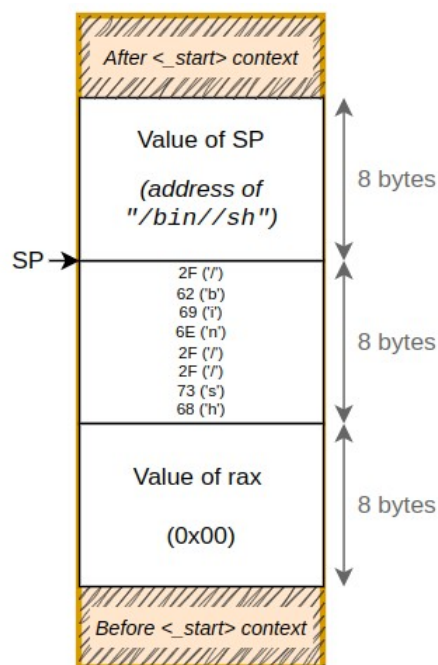
II.3. Extraction du shellcode

Observez le contenu du fichier `shell_asm.s`.

```
_start:
    pushq    %rax
    movq     $0x68732f2f6e69622f, %rbx
    pushq    %rbx
    xorq     %rdx,%rdx
    xorq     %rsi,%rsi
    pushq    %rsp
    popq     %rdi
    movb     $0x3b, %al
    syscall
```

Vous devriez observer qu'il mène au résultat de la figure ci-dessous. Notez que cette fois-ci la chaîne de caractère `"/bin/sh"` n'est pas allouée statiquement dans la section `.rodata` de l'exécutable (contrairement au cas du fichier `shell.c`), mais qu'elle est stockée sur le segment de pile, dans le contexte de la fonction `_start`. Ainsi son adresse relative est connue puisqu'il s'agit de l'adresse désignée par le *Stack Pointer*.

System call	Parameters				
<code>syscall</code>	<code>rax</code>	<code>rdi</code>	<code>rsi</code>	<code>rdx</code>	
0x0F05	0x3B	SP = address of "/bin/sh"	0x00	0x00	



Compilez et exécutez le programme pour confirmer son fonctionnement.

```
as shell_asm.s -o obj/shell_asm.o
ld obj/shell_asm.o -o bin/shell_asm
./bin/shell_asm
```

Observez l'implémentation binaire du programme et extrayez le *shellcode* (recopiez l'intégralité du code binaire du programme).

```
objdump -S obj/shell_asm.o
```

III. Exécution d'un shellcode sur la pile

L'exercice précédent nous a permis d'identifier un code binaire opératoire relativement léger (24 octets) permettant de lancer un shell à partir d'un exécutable. Dans l'esprit du *hacking*, nous chercherons à réutiliser ce **shellcode** (littéralement un *code qui lance un shell*) en exploitant une vulnérabilité du système. Ainsi le programme pourra lancer un *shell* accessible au hacker alors même que le logiciel d'origine ne le permettait pas.

Ouvrez le fichier `shellcode.c` et analysez le contenu du tableau `shellcode[]`. Que constatez-vous ? Qu'est censé faire ce programme ?

Compilez et exécutez le programme.

```
gcc shellcode.c -o bin/shellcode
./bin/shellcode
```

Le programme peut-il s'exécuter ?

Observez l'en-tête de programme du fichier ELF exécutable de sortie, et précisez les droits (`rwX`) sur le futur segment de pile qui sera associé à notre programme à l'exécution ?

```
objdump -fp bin/shellcode
```

Durant l'édition des liens, le *linker* a la capacité de préparer les droits associés aux futurs segments mémoire de l'application, notamment le segment de pile. Ce travail est réalisé à la construction du fichier ELF (en-tête du programme). Compilez à nouveau le fichier `shellcode.c` et rendant la pile exécutable, vérifiez les droits sur le segment de pile et exécutez le programme.

```
gcc -fno-stack-protector -z execstack shellcode.c -o bin/shellcode
objdump -fp bin/shellcode
./bin/shellcode
```

Le programme peut-il s'exécuter ?

Que fait le programme ?

Compilez et analysez le programme assembleur. Analysez ensuite l'exécution du programme en ouvrant une session de *debug* avec **gdb**. Décrivez le fonctionnement du programme en proposant un schéma commenté !

```
gcc -S -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o misc/shellcode.s

gcc -g -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o bin/shellcode

gdb ./bin/shellcode
(gdb) la a
(gdb) b main
(gdb) r
(gdb) s
(gdb) ni
...
(gdb) ni
$                               ← Ici, le shellcode est en cours d'exécution
$ exit
(gdb) q
```

Après analyse du script assembleur, représentez ci-contre le contenu de la pile avant exécution de l'instruction `call %rdx` implémentant `return(0)` de la fonction `main`. S'aider des outils et des commandes suivantes.

```
gcc -c -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o obj/shellcode.o

objdump -S ./obj/shellcode.o

objdump -s ./obj/shellcode.o
```

À ce stade, vous avez réussi à récupérer le code binaire d'un programme permettant de lancer un *shellcode* (premier exercice). De nombreux exemples de *shellcodes* sont disponibles sur Internet²³. Puis vous avez stocké le code binaire de ce *shellcode* sur la pile de sorte à l'exécuter par la suite.

Avec l'exercice suivant, vous verrez comment insérer ce *shellcode* dans un programme existant afin de le détourner de son fonctionnement normal.



²³ <http://shell-storm.org/shellcode/>

IV. Exploitation de vulnérabilité

Nous allons dans cette dernière partie donner quelques briques et techniques permettant l'ouverture au développement d'un *exploit* (exploitation d'une vulnérabilité). Une technique classique consiste à remplacer l'adresse de retour d'une fonction par celle d'un code malveillant (*shellcode*) caché sur la pile. Rappelons que l'adresse de retour d'une fonction est sauvée par défaut sur la pile durant l'appel de la fonction (*call*). Ainsi, lorsque la fonction courante souhaitera se terminer en exécutant l'instruction *ret*, qui dépile l'adresse de retour de la fonction appelante depuis la pile pour la placer dans le registre CPU d'instruction *rip*, la fonction la remplacera par l'adresse du code malveillant à exécuter. Voilà ci-dessous une écriture en pseudo-code des instructions *call* et *ret*.

CALL <i>function_address</i>	<=>	RSP ← RSP - 8 *(RSP) ← RIP RIP ← <i>function_address</i>
RET	<=>	tmp ← *(RSP) RSP ← RSP + 8 RIP ← tmp

Ouvrez le fichier *disco/hack/exploit.c* et analysez-le. Pour l'instant on considère qu'il s'agit d'une application classique, dans laquelle on a ajouté le *shellcode*. Compilez et exécutez le fichier.

```
gcc -fno-stack-protector -z execstack exploit.c -o bin/exploit
./bin/exploit
```

A-t-on accès à un shell (autrement dit, le *shellcode* s'exécute-t-il) ? Pourquoi ?

Pour que le *shellcode* s'exécute, il faut qu'à un certain moment le *Instruction Pointer* pointe vers celui-ci. Pour cela, nous allons falsifier dans la pile l'adresse de retour de la fonction ayant appelé le *main()*. L'instruction *ret* se chargera d'affecter l'adresse frauduleuse dans le *Instruction Pointer*.

Modifiez le premier commentaire *TODO* dans le fichier *disco/hack/exploit.c* par une ligne de code permettant de remplacer l'adresse de retour de la fonction appelante par l'adresse du *shellcode* sur la pile. Compilez et validez la bonne exécution du programme.

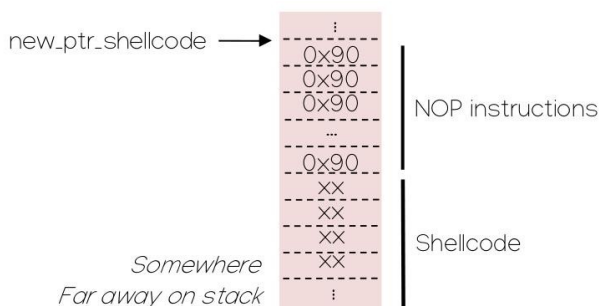
Avez-vous désormais accès à un shell ? Si oui, mission accomplie.

Différents mécanismes de sécurité (logiciels ou matériels) peuvent mettre à mal l'exécution de *shellcodes*, car ceux-ci sont (plutôt) connus et sont donc détectables. Pour éviter qu'un *exploit* soit détecté il existe différentes techniques de camouflage :

- *obfuscation* : le *shellcode* est encodé ou chiffré avant d'être inséré dans le firmware, mais il faut également intégrer une routine de décodage ou déchiffrement ;
- *padding, garbage insertion* : ajout d'instructions **NOP** (*No Operation*) ou de code mort (code sans impact) dans le *shellcode* afin de brouiller la lisibilité de celui-ci
- ...

La méthode que nous allons implémenter ici consiste à insérer le *shellcode* sur la pile, mais suffisamment loin du bas de pile. Les outils automatiques ne font généralement qu'analyser le code et les données proches des informations déjà présentes, ce qui permet d'éviter au *shellcode* d'être détecté. Par contre, des outils d'analyse heuristique pourront remarquer un accès soudain à une zone mémoire encore jamais utilisée, ce qui peut paraître suspect et provoquer une analyse.

Modifiez le second commentaire **TODO** afin de déplacer le *shellcode* plus loin sur la pile. Nous placerons des instructions **NOP** (*opcode* binaire **0x90**) avant le *shellcode*. Cette technique est couramment utilisée afin de faciliter la recherche d'un *shellcode* sur la pile par un *exploit*. Notamment lorsque l'adresse exacte du code malveillant n'est pas précisément connue.



Compilez et testez la solution. Si vous avez accès au shell, alors mission accomplie : vous venez d'insérer un code malveillant dans un code source, tout en le camouflant un minimum \o/

*Note : Les solutions sont cachées dans le répertoire **hack/misc/** !*

V. White-hat

Cet exercice de travaux pratiques rassemble un tas de compétences, dont toutes les notions importantes attendues dans cet enseignement. Si vous avez pu réaliser cet exercice par vos propres moyens, alors vous pouvez être fier.

Vous pouvez aller encore plus loin en cherchant à développer un exploit permettant d'ouvrir une console *root* sur votre machine personnelle (à partir d'un programme *non-root*). Si vous y arrivez, chapeau ! Auquel cas votre exploit pourra être intégré dans cette trame de TP pour les futurs élèves (avec tout le crédit qui va avec, et peut-être même des cafés jusqu'à la fin d'année) !

Notamment depuis l'arrivée d'internet, les systèmes numériques d'information se sont grandement complexifiés, mais également durcis. Bien des vulnérabilités ont déjà été explorées et exploitées, et les solutions déjà déployées. Mais comme une recherche inextinguible de trésor, bien des failles restent encore à trouver. Que ce soit au niveau matériel (étage de prédiction avec les failles *Meltdown* et *Spectre*, *NX bit* ou *No eXecute bit* dans la table de translation d'adresses utilisée par la MMU, ...) ou au niveau *kernel* Linux avec par exemple une gestion aléatoire des *mapping* mémoire de certains segments applicatifs (pile, tas, bibliothèques partagées et *vDSO*, ...), plusieurs contre-mesures sont déjà à l'œuvre afin de contraindre le travail des *Black Hats* dans leur volonté de pénétrer les systèmes.

Prenons l'exemple de la fonction `execve`. Comme précisé dans la documentation officielle²⁴, cette fonction travaille par remplacement de segments mémoire de la fonction appelante (`.text`, `.bss`, `.data` et pile). De même, il existe un jeu de conditions possibles et documentées permettant à l'appelant d'hériter des privilèges d'exécution de l'appelé²⁵. Je vous laisse donc mûrir et entrouvrir le spectre du possible, si l'applicatif appelé et le système de fichiers sous-jacent n'a pas été pensé contre !

Et pour les bouineurs et bouineuses, le site RootMe²⁶ donne accès à de nombreux exercices de *hacking*. Les défis relevés sur ce site font même l'objet d'un portfolio qui peut être analysé dans le cadre d'entretiens d'embauches !

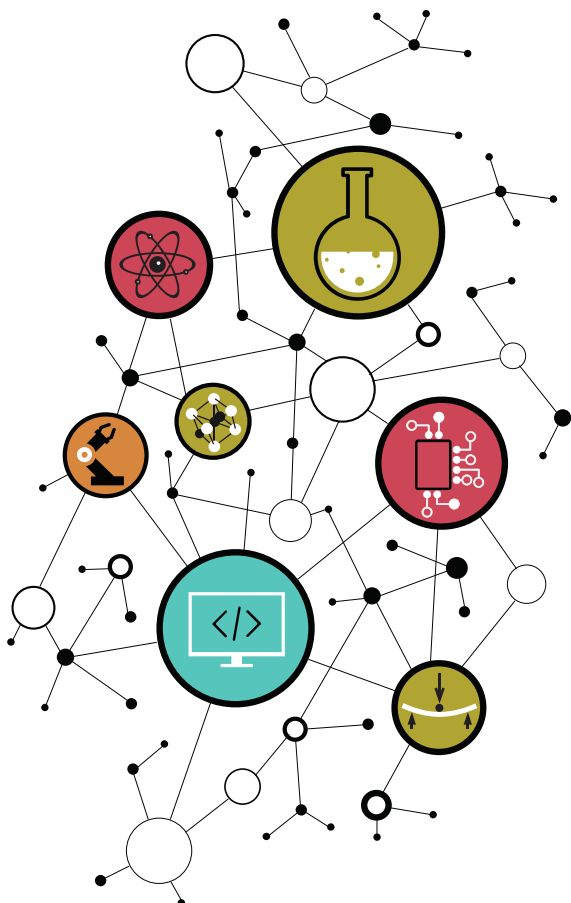
²⁴ <https://www.man7.org/linux/man-pages/man2/execve.2.html>

²⁵ <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=fs%2Fexec.c>

²⁶ <https://www.root-me.org/>

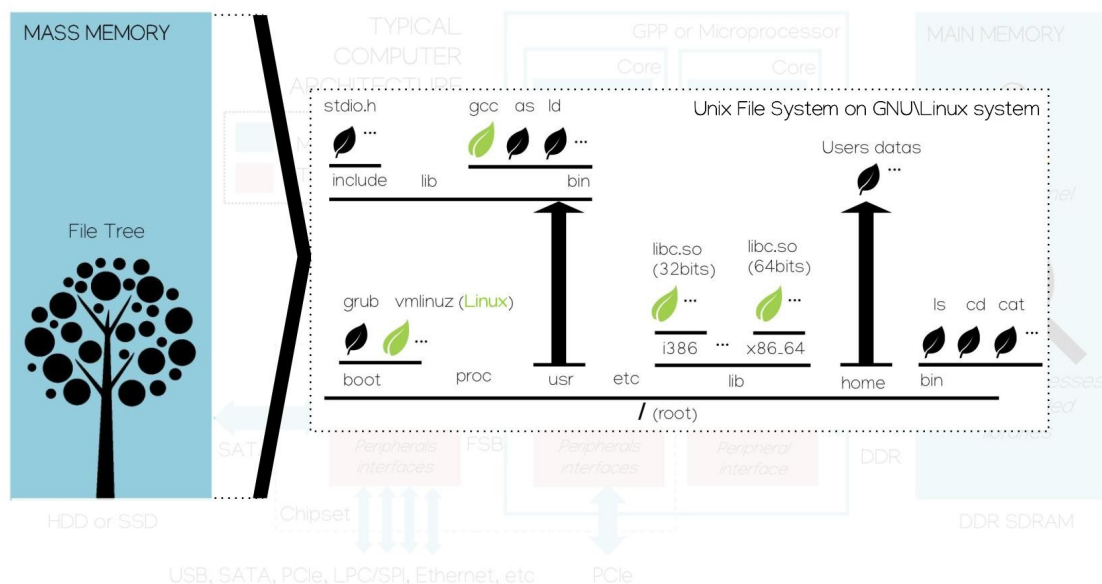
PARTIE 10

MÉMOIRE DE MASSE ET STOCKAGE DES FICHIERS



I. Du support physique à la représentation logique

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 (opérations de lecture ou d'écriture, adresse, nombre d'octets, etc) et de répondre à celle-ci en délivrant ou en stockant l'information demandée.

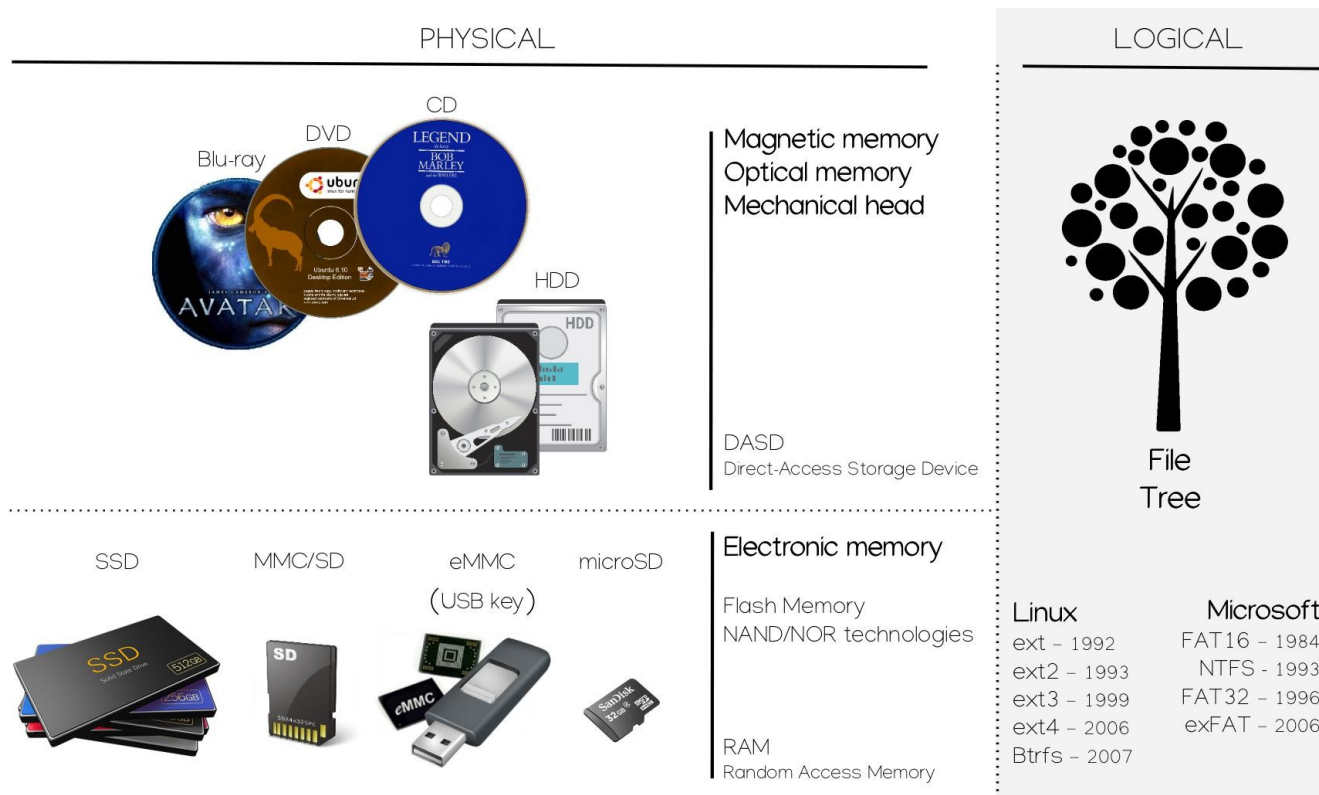


Rappelons les 3 familles de mémoires rencontrées sur ordinateur :

- **Mémoire cache** : mémoire adressable par association (associative) de technologie SRAM. À 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 représentée au format hexadécimal (32-bit, 64-bit, exemple *39-bit physical* et *48-bit virtual*, ...; `cat /proc/cpuinfo`). Sur processeur intégrant une MMU (GPP, AP, ...) nous distinguons une adresse virtuelle (vision système, CPU et développeur) d'une adresse physique (limitation matérielle sur la machine). 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 SSD, HDD, MMC, ... Cette mémoire stocke 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.

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, ...), tout comme certaines sont maintenant tombées en désuétude (K7 audio, disquette, VHS, VHS-c, carte perforée, tore magnétique, ...). Chaque technologie offre son lot hérité d'avantages, d'inconvénients, de compromis et se trouve adaptée 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.

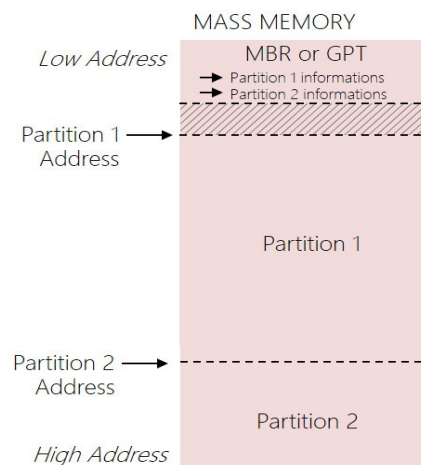
Indépendamment des technologies physiques de stockage utilisées, la représentation logique de l'information offerte par le système se base sur les concepts de partitions (zones logiques contiguës de la mémoire physique) et d'adressage de l'information par arborescence de fichiers. Plusieurs technologies de **systèmes de fichiers (FS ou File System)** sont en usage à notre époque. *Btrfs*, *ext4*, *SquashFS*, ... sous Linux ; *NTFS*, *FAT32*, *VFAT*, ... sous Windows ; et encore bien d'autres notamment pour d'autres systèmes d'exploitation (BSD, Mac OS X, etc). Chaque technologie offre son 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 : 4 GB
- Taille maximale théorique d'une partition : 16 TB
- Nombre maximal de fichiers : 268 M
- Nombre maximal de fichiers par répertoire : 65534

II. Table des partitions

Une partition représente une zone logique contiguë de la mémoire physique. Pour un support donné et en fonction des besoins, il est possible de définir plusieurs partitions, de différentes tailles, natures et à différents emplacements en mémoire. Pour ce faire, deux technologies représentant la table des partitions d'un média dominant le marché :

- **MBR (Master Boot Record)**²⁷, le support physique peut contenir quatre partitions primaires maximum, toutes de tailles inférieures à 2 TB. Technologie encore très utilisée dans les systèmes embarqués mais en voie d'extinction dans les ordinateurs personnels ;
- **GPT (GUID Partition Table)**, rétrocompatible MBR et offrant moins de limitations mais manquant parfois de support sur certains systèmes.



Dans cet exercice, nous allons travailler sur une clé USB 16 Go en supprimant l'ancienne table des partitions et en la remplaçant par un nouveau MBR. Attention, certaines étapes exécutées en tant que super utilisateur **root (sudo)** sont critiques et risqueraient notamment de supprimer la table des partitions du disque système de façon irréversible. Ne surtout pas se tromper dans le choix du périphérique matériel `/dev/sdX` durant les exercices qui suivent !

Placez-vous dans le répertoire `/disco/mass/` puis connectez la clé USB à votre machine. Identifiez son nom (de la forme `/dev/sdX`) et ses caractéristiques.

```
lsblk
lsblk -f

sudo sfdisk -l

export DISK=/dev/<your_device_name>    # À modifier selon le nom de votre composant
```

Identifiez le paramètre système `block size` relatif à votre support (valeur dépendant de multiples paramètres), puis effacer l'ancienne table des partitions. Analysez la sortie.

```
sudo blockdev --getbsz ${DISK}

sudo dd if=/dev/zero bs=1K count=10K of=${DISK}

lsblk

sudo dd if=${DISK} bs=1K count=1 of=./mbr.bin

xxd ./mbr.bin > mbr.txt
```

Que constatons-nous ?

²⁷ <https://doc.ubuntu-fr.org/mbr>

Nous allons créer une nouvelle table des partitions en utilisant l'utilitaire `sfdisk`. Néanmoins, d'autres utilitaires existent pour créer des tables des partitions (`parted`, `fdisk`, ...). Créez une nouvelle table des partitions et analysez la sortie.

```
sudo sfdisk ${DISK}
>>> help -> [ENTER]
>>> ,, -> [ENTER]
>>> quit -> [ENTER]
>>> Y -> [ENTER]

lsblk

sudo dd if=${DISK} bs=1K count=1 of=./mbr.bin

xxd ./mbr.bin > mbr.txt

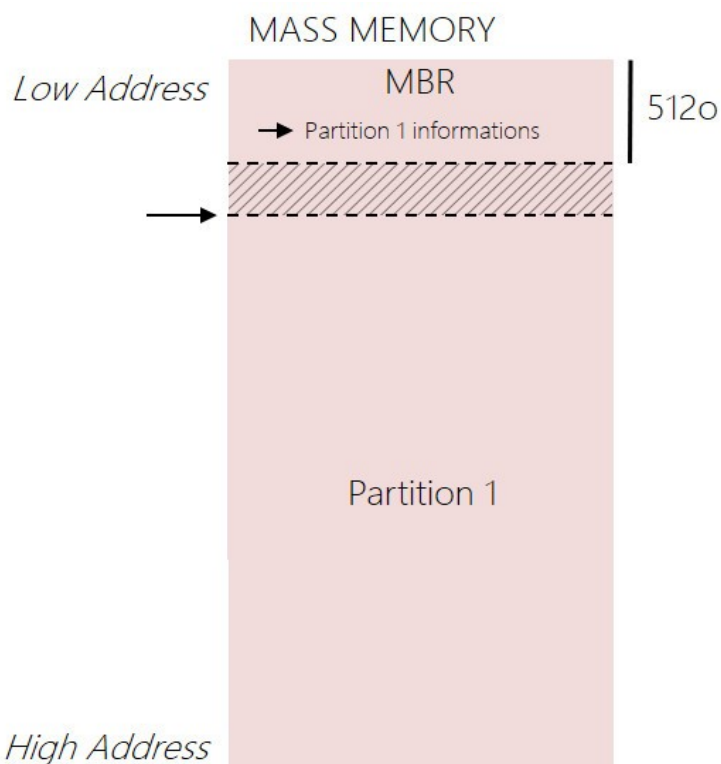
file mbr.bin
```

En s'aidant d'Internet, précisez la taille d'un MBR ?

Par quelle suite d'octets se termine toujours un MBR ? Vérifiez que cette suite binaire est bien présente après création de la table des partitions.

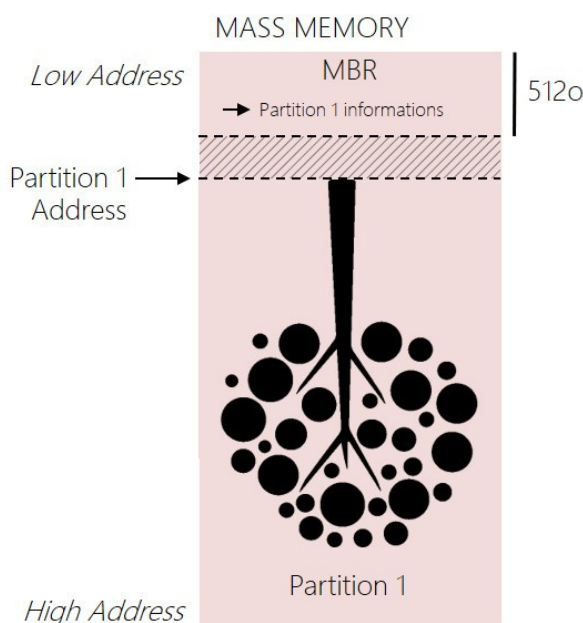
Quelle est la taille d'un bloc logique sur notre clé USB ?

Sur le schéma suivant, précisez à quelle adresse (en octet) débute la partition n°1 précédemment créée.



III. Système de fichiers

Nous allons maintenant déployer un système de fichier sur la partition précédemment créée. Le choix de la technologie du FS (*File System*) choisi peut conditionner les performances voire le bon fonctionnement du système ou du média. Par exemple, une clé USB sera probablement amenée à être utilisée sur machine supervisée par Windows comme par Linux. Windows étant une solution propriétaire et fondamentalement fermée, mieux vaut préférer une FS propriétaire comme **NTFS** ou **FAT** afin d'éviter toute mauvaise surprise et une bonne compatibilité à l'usage. Sur système embarqué (MCU), préférer des FS légers, mûrs et offrant du support comme **FAT** par exemple. Sur ordinateur ou système embarqué (SoC AP) supervisé par Linux et couplé au réseau, préférer par exemple **Btrfs**.



Déployez un système de fichiers **Virtual FAT** (extension à FAT12, FAT16 et FAT32) sur la partition n°1 et nommez cette partition à l'aide d'un label (option **-n**). Ce label sera à l'avenir utilisé par le système pour nommer les futurs points de montage.

```
lsblk -f
sudo mkfs.vfat -n root ${DISK}1
lsblk -f
```

Observez l'implémentation technologique **Virtual FAT** du système de fichiers à l'adresse de début de la partition. Retrouvez votre label ?

```
sudo dd if=${DISK} bs=1K count=2K of=./fs_fat32.bin
xxd ./fs_fat32.bin > fs_fat32.txt
```

IV. Point de montage

Un **point de montage** est un répertoire dans le FS de la machine *host* (ordinateur) représentant l'image logique du contenu du média de stockage de masse externe. Sous Unix, les points de montage sont généralement présents dans les répertoires racine `/mnt/` (point de montage manuel) et `/media/` (points de montage automatiques). En effet, contrairement à Windows qui considère les périphériques externes comme des lecteurs différenciés du lecteur principal, Linux traite les partitions et périphériques de stockage comme des fichiers. Rappelons que sous Unix, tout est fichier !

Créez un point de montage et respectez le nom du label présent dans la partition n°1 précédemment créée (néanmoins, ce nom pourrait être différent). Vérifiez et validez avant de réaliser l'opération, le chemin relatif à votre nom d'utilisateur dans `/media/`. Validez les opérations réalisées.

```
export MEDIA=/media/<user_name>
sudo mkdir -p ${MEDIA}/root/
sudo mount ${DISK}1 ${MEDIA}/root/
lsblk -f
```

Réalisez une écriture sur le point de montage, ...

```
echo "Hello World Bro's !" > hello.txt
sudo cp hello.txt ${MEDIA}/root/
```

... synchronisez point de montage et support physique, ...

```
sync
```

... puis retirez manuellement du système le point de montage.

```
sudo umount ${MEDIA}/root
```

Vous pouvez alors retirer physiquement la clé USB de la machine et valider son bon fonctionnement sur un autre ordinateur ! Ne pas oublier l'étape de synchronisation, sinon l'écriture sur le média cible ne sera pas active.

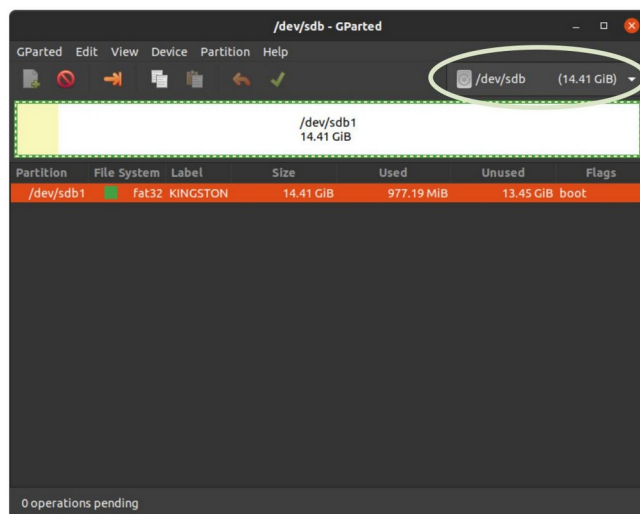
V. Outil graphique GParted

Ce n'est pas parce qu'on travaille sous Linux qu'on doit tout faire à la console, dans l'ombre en sentant le café et les cacahuètes (ne nous jugez pas). Tout ce que nous avons fait ici est réalisable avec un outil graphique, automatisant les étapes précédemment présentées (création de la table des partitions et déploiement d'un système de fichiers). Nous utiliserons **GParted** (*GNOME Partition Editor*) basé sur GNU Parted, l'un des outils graphiques les plus standards sur système GNU/Linux.

Ouvrez GParted.

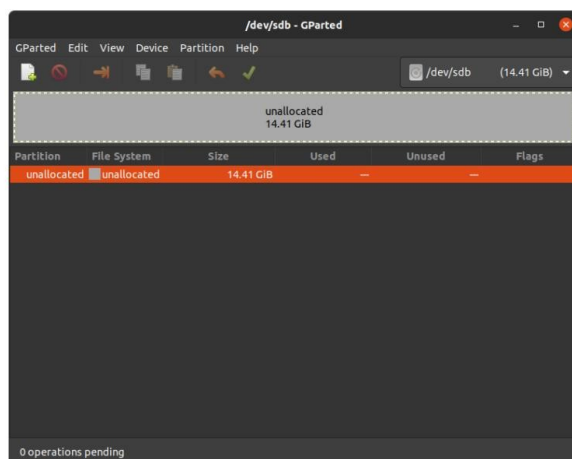
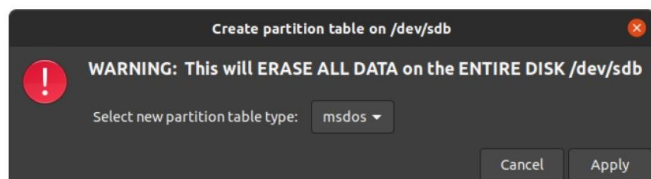
Sélectionnez le périphérique matériel à partitionner et retirez le point de montage existant :

- GParted > Refresh Devices
- GParted > Devices > /dev/sdX (*your device name*)
- Clic droit sur la partition du device (Partition > /dev/sdb1 - ci-dessous) > Unmount



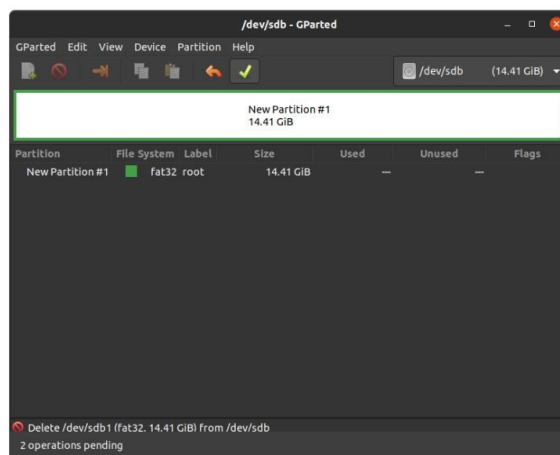
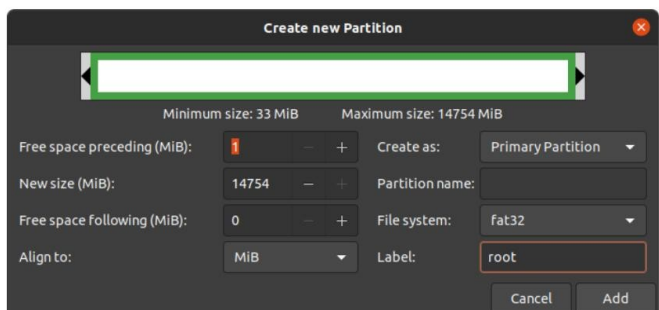
Créez une table de partitions MSDOS (technologie MBR) :

- Device > Create Partition Table...
- msdos > Apply



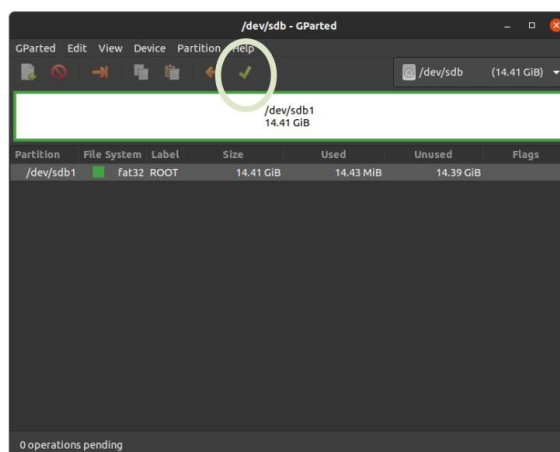
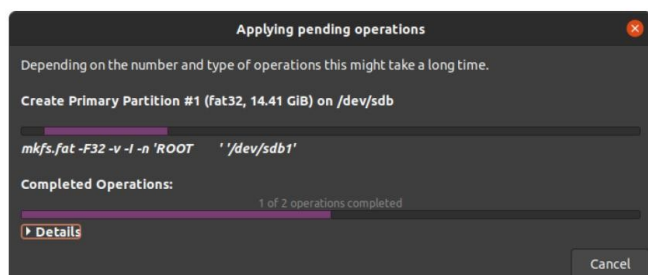
Créez une nouvelle partition :

- Partition > New
- Valeurs par défaut sauf les champs suivants
 - File system : > fat32
 - Label : > root
- Add



Appliquez les configurations précédentes :

- Cliquez sur l'icône "Apply all operations" entourée ci-dessous
- Et c'est terminé !



VI. Kali sur clé USB bootable



Pour celles et ceux arrivés jusqu'ici, cet ultime exercice permet de déployer une image disque ISO sur une clé USB afin de la rendre *bootable* au démarrage et donc de charger dans notre cas au *boot* (phase d'amorçage) un système Kali Linux²⁸ dédié à la pénétration des systèmes.

Une image disque est historiquement une archive correspondant à la copie conforme d'un disque optique ou magnétique. Le format le plus répandu à notre époque est ISO (norme ISO 9660), même si d'autres standards existent (ISZ, IMG, UIF, ...).

Téléchargez l'ISO d'un Kali Linux Light 64-bit : <https://www.kali.org/downloads/>

À ce stade de l'enseignement, vous devez être apte à comprendre le tutoriel proposé sur le site officiel Kali afin de préparer un clé USB bootable :

```
sudo sfdisk -l

export DISK=/dev/<your_device_name>

dd if=kali-linux-<your_version>.iso of=${DISK} bs=4M
```

Une fois l'installation réalisée, redémarrez votre ordinateur en interrompant la phase de *boot* à l'amorçage (appuyez sur **F12** sur ordinateur en salles A203/A201 ou **F10**, **F2**, ... cela dépend du fabricant de carte mère), spécifiez que vous souhaitez démarrer sur un support USB et un Kali Linux va se démarrer en quelques secondes.

²⁸ <https://www.kali.org/>

