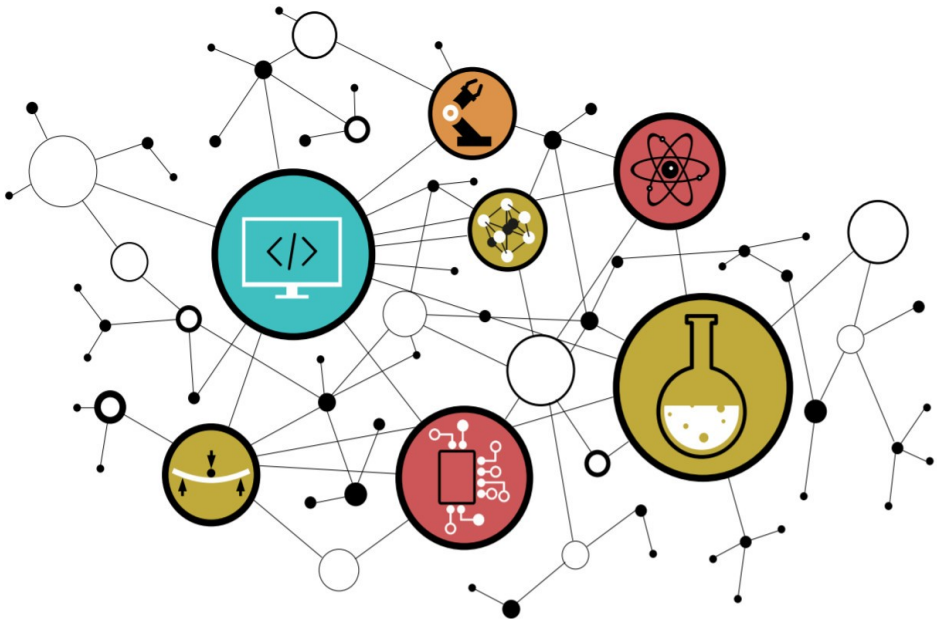
ARCHITECTURE DES ORDINATEURS

TRAVAUX PRATIQUES



CONTACTS

Équipe enseignante

hugo descoubes
hugo.descoubes@ensicaen.fr
+33 (0)2 31 45 27 61

Patrick Lacharme
patrick.lacharme@ensicaen.fr

André Lépine
andre.lepine@ensicaen.fr

Tanguy Gernot
tanguy.gernot@ensicaen.fr

ENSICAEN
6 boulevard Maréchal Juin
CS 45053
14050 CAEN cedex 04

RESSOURCES

Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSICAEN. Télécharger l'archive complète de travail **arch.zip**

https://foad.ensicaen.fr/course/view.php?id=99

ÉVALUATION

- QCM Moodle (30m) - mi-parcours

Questions sous forme de QCM (~30 questions) sur la compilation, l'édition des liens sous GCC et l'analyse de programme assembleur x86_64

- Examen sur table (1h30) - terminal

L'évaluation sur table portera sur les séances de Cours Magistral (potentiellement sur tout point présent dans les supports ou présenté à l'oral) ainsi que sur la trame de Travaux Pratiques. S'aider des conseils et exercices présents dans l'archive de travail en ligne (arch/cm/eval) :

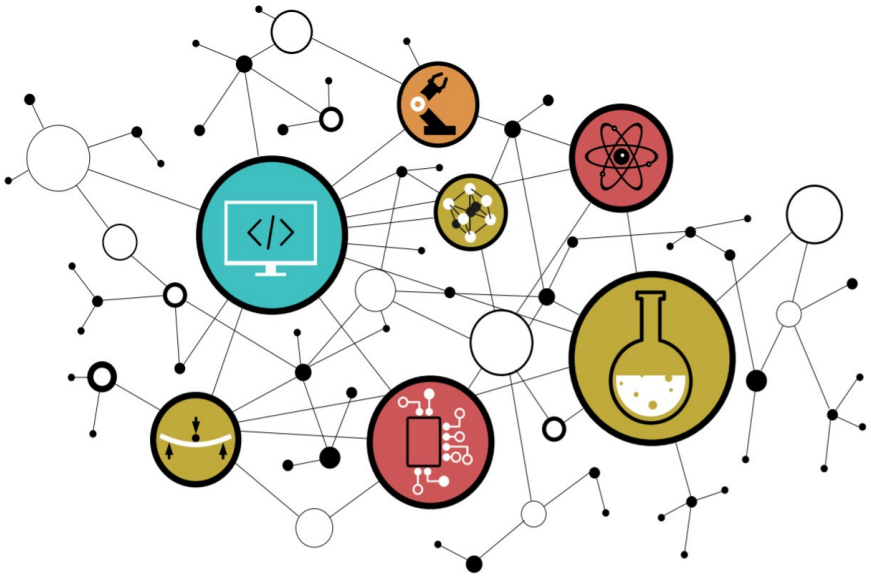
  - SAVOIR – 7pts : Questions de culture générale pouvant traiter sur tout point abordé en séance de cours présentiel ou présent dans le support de travail. *Connaissances fondamentales et culture scientifique de l'ingénieur électronicien.*

  - ANALYSER – 13pts : Exercice de traduction d'un programme assembleur vers un programme C et analyse des mécanismes de gestion de la pile conjointement réalisés par le système d'exploitation, la chaîne de compilation et le processeur. *Comprendre et maîtriser le travail d'un processeur numérique, d'une chaîne de compilation et des mécanismes de gestion de la mémoire. Adaptabilité de l'élève ingénieur.*

Il est possible de s'entraîner très simplement concernant l'exercice d'analyse de code. Éditer un programme C simple (1 à 2 appels de fonctions imbriqués, pas plus de 2 paramètres par fonction). Les fonctions réaliseront des traitements élémentaires. Générer le fichier assembleur 64bits x64 correspondant à votre programme et le fournir à un camarade de promotion sans lui donner le programme C équivalent. Que votre camarade réalise le même travail. En partant du fichier assembleur, essayer de proposer un programme C pouvant générer le même assembleur (plusieurs solutions possibles). De même, proposer le contenu exhaustif de la pile lorsque le pointeur de sommet de pile (SP) se trouve au plus haut sur la pile. Confronter par la suite les solutions proposées.

TRAVAUX PRATIQUES

PRÉLUDE



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2022

SOMMAIRE

La trame de TP minimale que nous considérons comme être les compétences minimales à acquérir afin d'accéder aux métiers de base du domaine suit le séquencement suivant : chapitres 1, 2, 3, 4, 5 et 6 (voire 8 pour une mise en application et 9 pour la gestion des ressources de stockage de masse). Le reste de la trame ne sera pas évalué et représente une extension au jeu de compétences minimales relatif au domaine en cours d'étude (* devant chapitres facultatifs voire complémentaires). Libre à vous d'aller plus loin selon votre temps disponible et votre volonté de mieux comprendre et maîtriser ce domaine !

SEQUENCEMENT PÉDAGOGIQUE

Le plan et le séquencement de la trame de Travaux Pratiques correspondent au chemin proposé afin d'atteindre les objectifs pédagogiques fixés. Ces objectifs ont été choisis au regard des attentes et exigences demandées par les marchés de l'industrie du logiciel et des couches basses des systèmes : systèmes embarqués, systèmes temps réel, développement de systèmes d'exploitation, développement de bibliothèques spécialisées, développement de chaînes de compilation, attaque et sécurité des systèmes, etc, tous des métiers dans divers domaines actuellement exercés par certains de nos anciens élèves. Il s'agit d'un séquencement conseillé qui n'a en aucune façon volonté à être imposé (étudiant comme enseignant encadrant). Pour se dérouler sous les meilleurs hospices, il serait préférable dès le début de l'enseignement de rythmer 1 à 2 heures de travail personnel à la maison en dehors des séances en présentiels avec enseignant. Voici une proposition de séquencement. Lire à minima l'introduction de chaque TP avant de venir en séance :

2. Compilation et éditions de liens

- En session de TP : *2.5 / 2.6 / 2.7 (TP n°1)*
- Hors session de TP : *Lire le document prélude. Rédiger une synthèse de 3 pages sur la compilation et l'édition des liens à déposer sur moodle (avant TP n°1) et 3.1 (avant TP n°2)*

3. Assembleur x86 et bibliothèque statique

- En session de TP : *3.1 / 3.2 / 3.3 / 3.4 / 3.5 / 3.6 (TP n°2)*
- Hors session de TP : *4.1 (avant TP n°3)*

4. Allocations automatiques et segment de pile

- En session de TP : *4.1 / 4.2 (TP n°3) et 4.3 / 4.4 (TP n°4) et 4.5 / 4.6 (TP n°5)*
- Hors session de TP : *4.3 (avant TP n°4) et 4.5 (avant TP n°5) et 5.1 (avant TP n°6)*

5. Allocations statiques et fichier ELF

- En session de TP : *5.1 / 5.2 / 5.3 (TP n°6)*

6. Allocations dynamiques et segment de tas

- En session de TP : *6.1 / 6.2 (TP n°6)*

7. Exceptions et signaux Unix - *Facultatif*

8. Hacking

- En session de TP : *8.1 / 8.2 / 8.3 (TP n°7)*
- Hors session de TP : *8.1 (avant TP n°7)*

9. Mémoire de masse et système de fichiers

- En session de TP : *9.1 / 9.2 / 9.3 (TP n°8)*
- Hors session de TP : *9.1 (avant TP n°8)*

1. PRÉLUDE

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

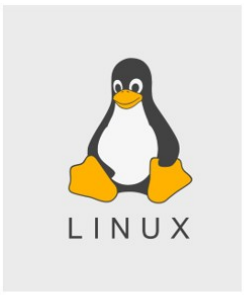
Dennis Ritchie, père du langage C et codéveloppeur d'Unix, à droite sur la photo ci-dessous

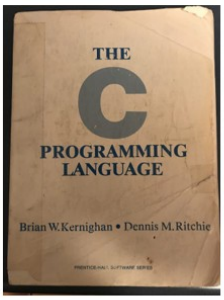
« I think the major good idea in Unix was its clean and simple interface: open, close, read, and write. »

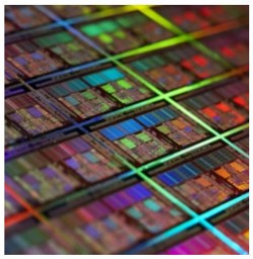
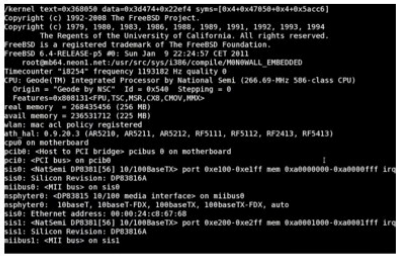
Kenneth Thompson, père du système d'exploitation Unix et du langage B, à gauche sur la photo

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

Linus Torvalds, père de noyau Linux





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

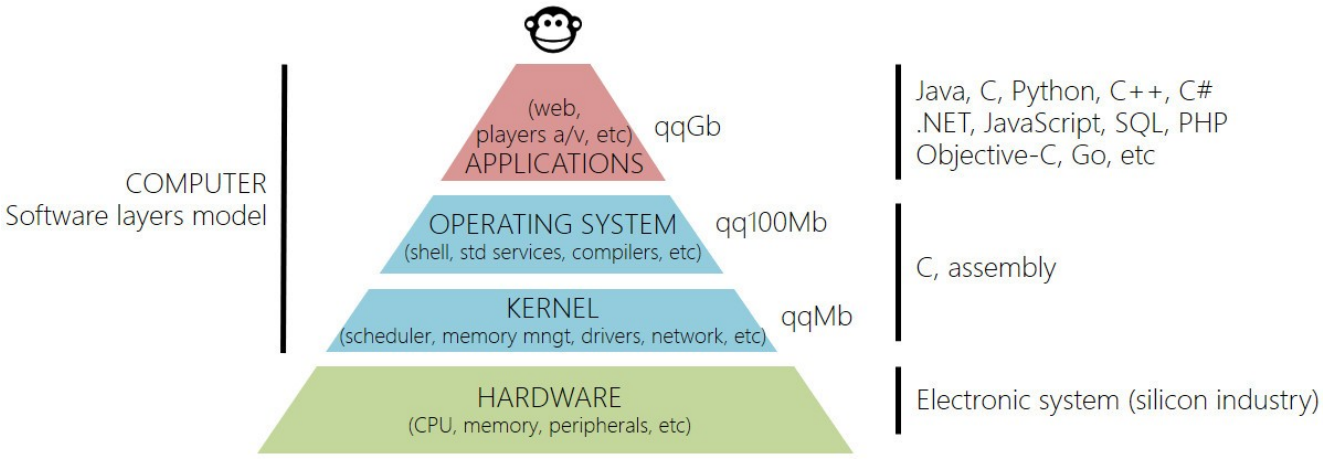
Les technologies matérielles choisies pour illustrer l'enseignement seront quant à elles celles restant toujours à notre époque les plus répandues sur ordinateur, soit les architectures compatibles x86/x64 (32bits/64bits). Ces technologies ont été historiquement développées et spécifiées par Intel (x86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

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

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

Dennis Ritchie

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



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

A notre époque, il reste toujours parmi les langages les plus populaires au monde (n°1 en 2021, source TIOBE index, https://www.tiobe.com/tiobe-index/). Il est d'ailleurs en constante croissance à l'usage, notamment avec l'avènement des systèmes embarqués, de l'IOT (Internet des objets) et des projets « maker » (Raspberry Pi, Arduino, etc). Prenons des objets et logiciels de votre quotidien dans lesquels une partie voire la totalité des développements logiciels ont été développés en langage C : Box internet, télévision, lecteurs CD/DVD/Blu-ray, enceinte Bluetooth, ordinateur et smartphone (systèmes d'exploitation - distributions GNU\Linux telles que Debian/Ubuntu/Redhat/Fedora/Kali/etc, Android, Mac OS X, iOS, Windows, etc) et noyaux (Linux, XNU, Darwin, Mach, NT, etc)), la liste est très très très longue ...

1.1. Objectifs pédagogiques

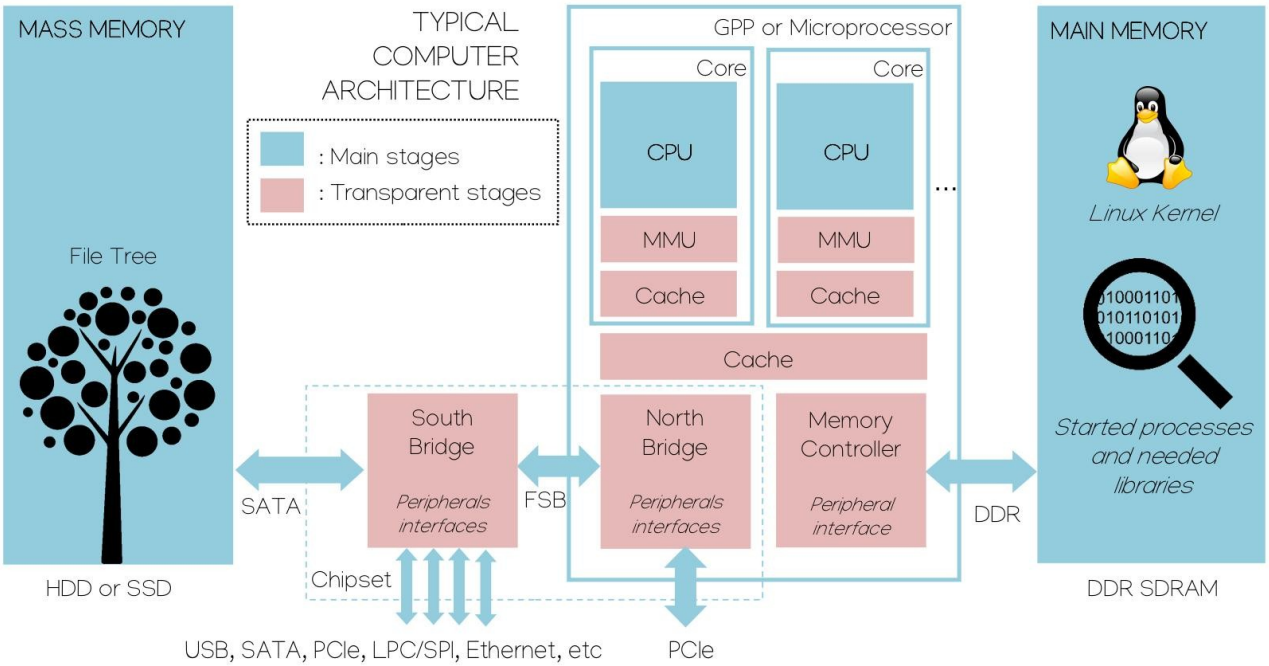
Cette trame d'enseignement ne cherche pas à vous réapprendre un langage de programmation. Il s'agit d'une trame d'analyse de programmes C élémentaires ayant pour objectif terminal d'aboutir à une meilleure représentation et compréhension du fonctionnement de la machine (ordinateur), de la compilation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

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

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

Architecture matérielle (représentation physique)

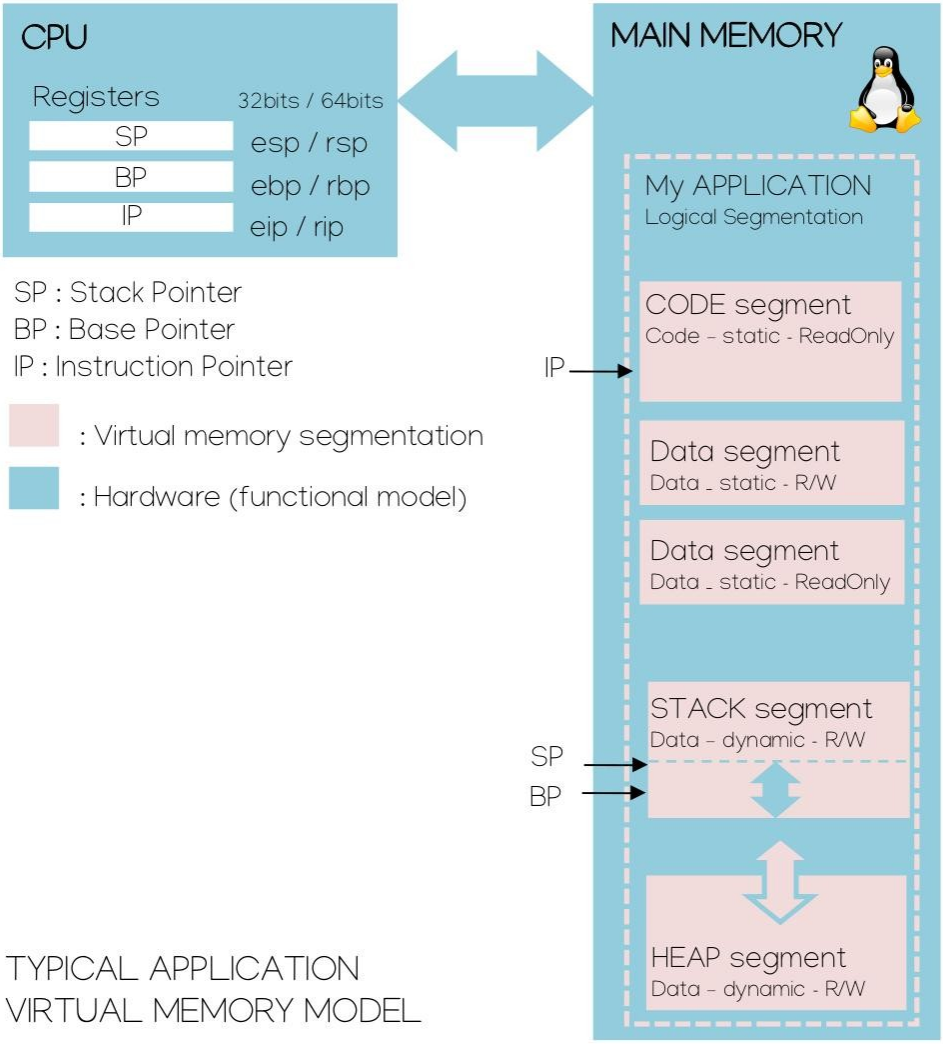
Les systèmes matériels actuels de traitement de l'information, tel qu'un ordinateur, sont devenus d'une grande complexité architecturale et technologique. A titre indicatif, à Caen chez NXP, le développement d'un simple composant sécurisé NFC (chiffrement de l'information) demande le travail direct de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



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

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

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



1.2. Outils et environnement de développement

« Sharing knowledge is the most fundamental act of friendship. Because it is a way you can give something without loosing something. »

Richard Stallman, créateur du projet GNU et fondateur de la Free Software Foundation

L'archive de travail est directement téléchargeable sur la plateforme pédagogique de l'ENSICAEN (https://foad.ensicaen.fr) : *Formation Classique > Spécialité Informatique > 1ère année > Architectures des ordinateurs > Download > «année courant» > arch.zip et disco.zip*. Les programmes à analyser se situent dans le répertoire *arch/tp/disco* (disco pour discovery).

L'ensemble de la trame de Travaux Pratiques sera réalisée sur système GNU\Linux (Ubuntu, Debian, Fedora, Mageia, etc la liste est longue). 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, http://gcc.gnu.org/). Les logiciels GNU sont tous des logiciels libres et ouverts, soutenus par la Free Software Foundation (fondateur en 1985 Richard Stallman, https://stallman.org/). Pour en savoir plus sur la philosophie de ces projets voire y collaborer, n'hésitez pas à vous rendre sur les sites officiels (https://www.gnu.org/software/). Pour information, les chaînes de compilation GNU-like sont à notre époque les plus répandues, notamment dans le domaine des systèmes embarqués (architectures supportées ARM, MIPS, Open Hardware RISC-V, STMicro, Microchip, etc). Elles s'imposent de plus en plus comme un standard.

```
wget https://foad.ensicaen.fr/<moodle_path>/arch.zip

unzip arch.zip -d <destination_folder>
```

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 virtualisé (https://ubuntu.com/#download). 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 (https://doc.ubuntu-fr.org/cohabitation_ubuntu_windows) ou de virtualiser Ubuntu sur une machine virtuelle. Le projet VirtualBox est par exemple un projet Open Source performant, bien maintenu et stable (https://www.virtualbox.org/wiki/Downloads). 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, etc) et en configurant le réseau.

Une fois votre système installé, voici potentiellement ci-dessous quelques packages complémentaires à installer. A partir de maintenant, la console système (shell), un éditeur de texte (emacs, vim, nano, geany, etc), un compilateur GNU (gcc), un noyau Linux, votre ordinateur et votre volonté de comprendre un peu mieux ce monde obscure du système seront vos principaux alliés ...

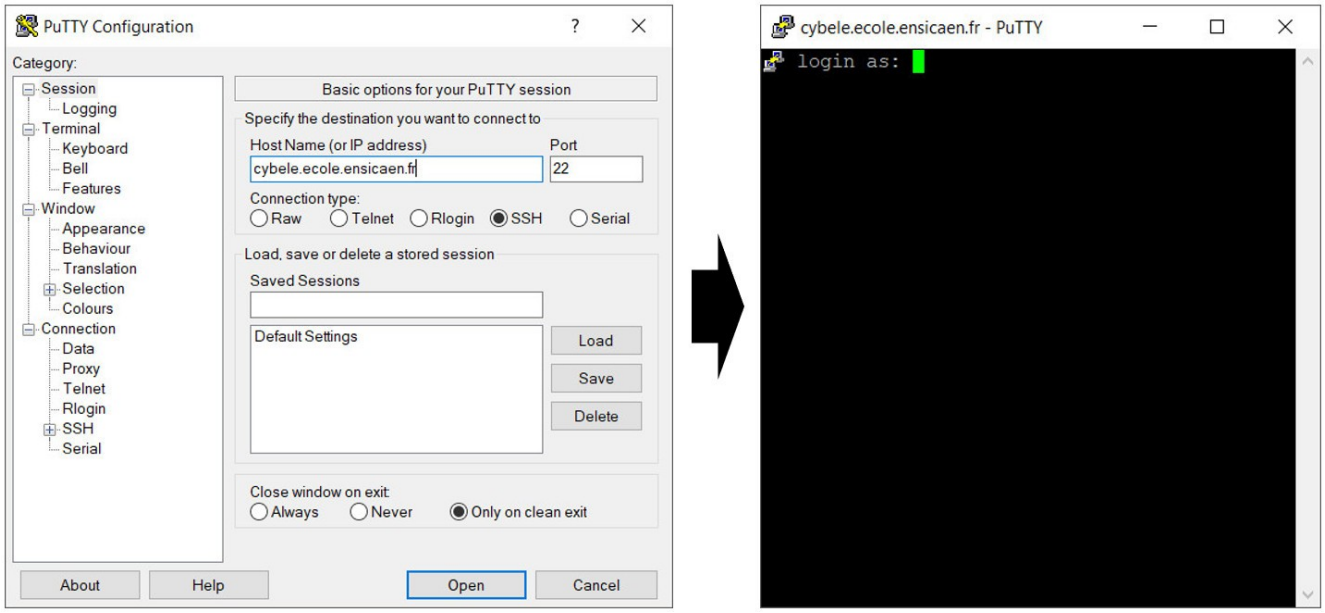
```
sudo apt-get update
sudo apt-get install build-essential gcc-multilib binutils wget unzip
putty
sudo dpkg --add-architecture i386
```

Travail à distance

Pour divers raisons, si vous ne souhaitez ou ne pouvez pas installer un système GNU\Linux sur votre machine personnelle, il vous est également possible de travailler à distance depuis n'importe quelle machine (Windows, Linux, etc) possédant le logiciel PuTTY (ou un autre terminal SSH). PuTTY est un programme léger pouvant notamment vous offrir un terminal et lien de communication sécurisé SSH (Secure Shell) distant avec une autre machine. Dans notre cas, nous l'utiliserons pour nous connecter à notre compte ENSICAEN directement sur le serveur école. Voici le lien officiel de téléchargement :

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

La configuration de l'outil PuTTY est très simple (Host Name : *cybele.ecole.ensicaen.fr*, Port : *22*, Connection type : *SSH*, *Open* puis login et mdp ENSICAEN). Un raccourci pratique à connaître sous PuTTY est le clic droit de la souris permettant de coller une sélection :



Archive de travail pratique et édition en mode console

L'archive *disco.zip* ne contient que les fichiers textes sources de TP à analyser (archive légère de qq10Ko, aucun support PDF ni ODT de CM ni de TP). Une copie de l'archive est présente sur le serveur cybele ENSICAEN au chemin suivant : */home/public/arch/disco.zip*. Nous aurons à utiliser un éditeur de texte en mode console. Nous utiliserons *nano* également porté par le projet GNU (https://www.nano-editor.org/), éditeur le plus simple de sa famille (vi, vim, etc). Lien vers la totalité des raccourcis *nano* : https://www.nano-editor.org/dist/latest/cheatsheet.html

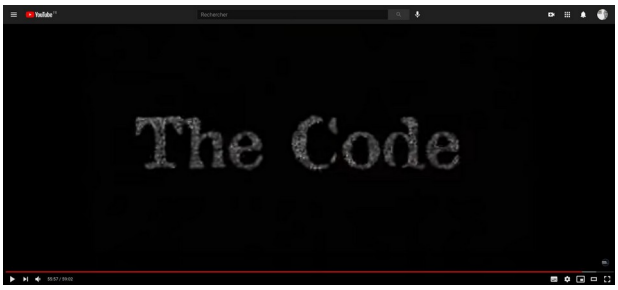
```
cd <working_directory>

cp /home/public/arch/disco.zip .

unzip disco.zip

rm disco.zip
```

1.3. Quelques ressources internet

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

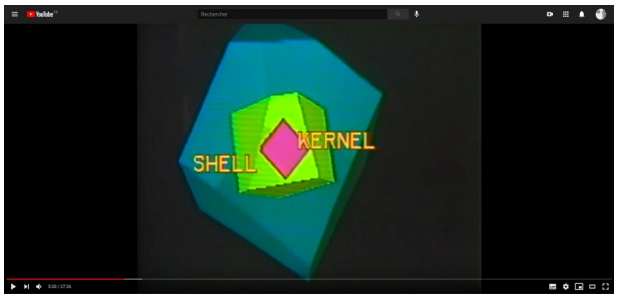
https://www.youtube.com/watch?v=dcN9QXxmRqk

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



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

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



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

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

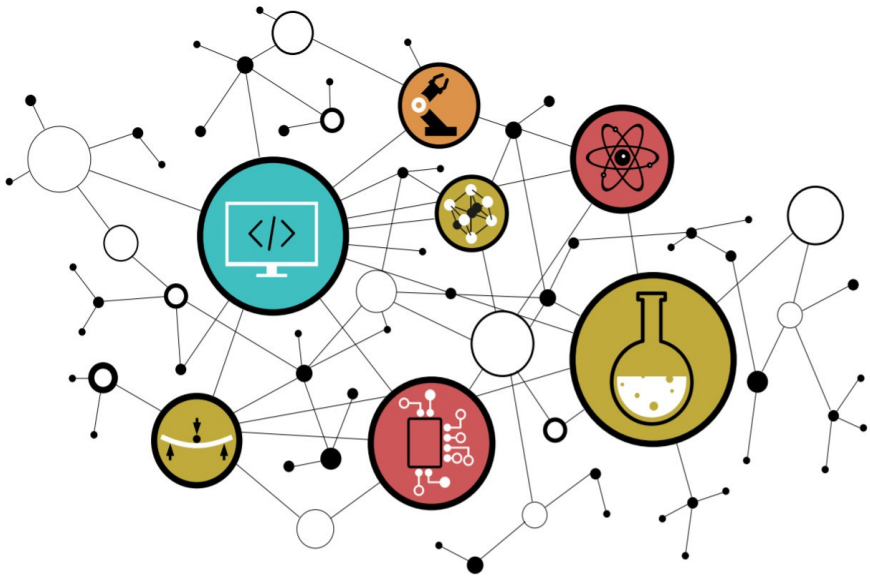
https://www.youtube.com/watch?v=dOakyAhqiVY&t=2140s

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

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

TRAVAUX PRATIQUES

COMPILATION ET ÉDITIONS DES LIENS



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

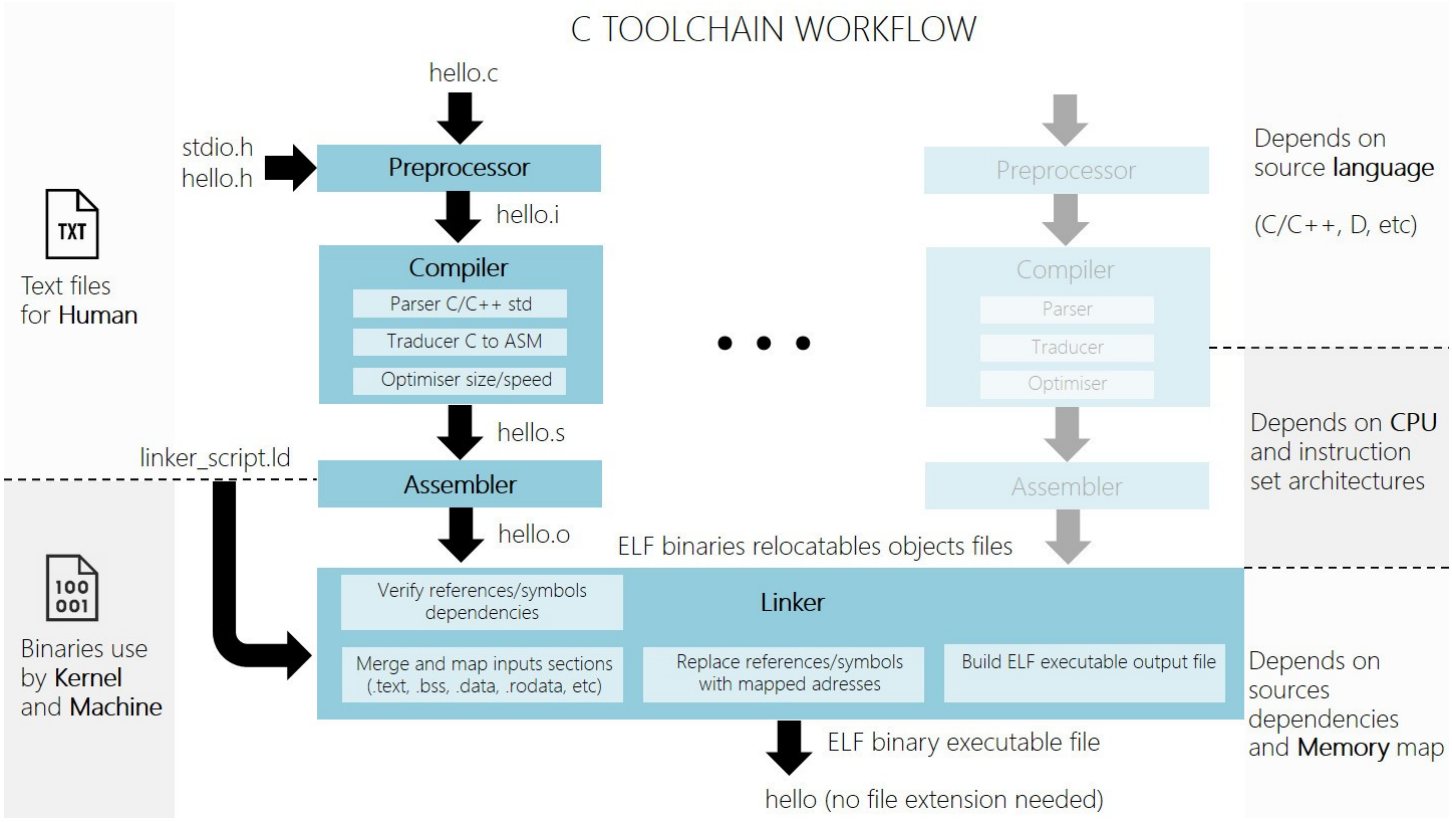
SOMMAIRE

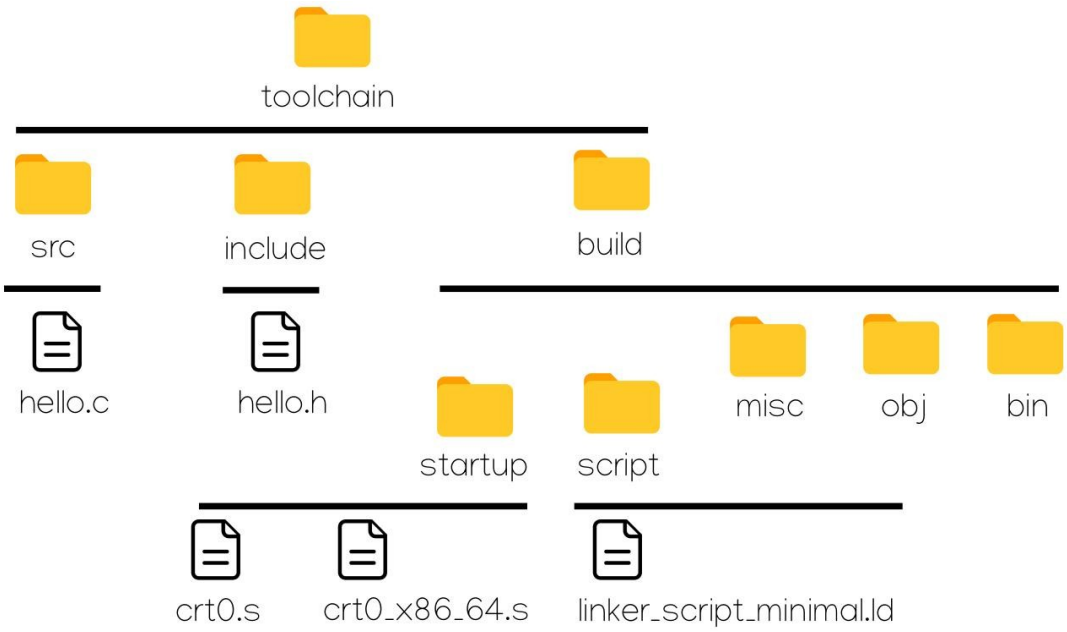
2. COMPILATION ET ÉDITION DES LIENS

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

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

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





- Compiler le programme, analyser la sortie et exécuter le fichier exécutable produit !

```
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 le 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'information de fichier objet
- *readelf (ELF format object file reader)* pour l'affichage d'information de fichier au format ELF
- *strip (symbols cleaner)* pour le nettoyage des symboles dans des fichiers objets
- *objcopy, gold, etc.*

2.1. Preprocessing

A 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 artisan développeur, notamment dans le monde du système. Imaginez un ébéniste ne sachant utiliser un rabot, des gouges ou ciseaux à mortaise …

- Compiler à nouveau le programme en s'arrêtant à l'étage de préparation du code avant la compilation (preprocessing). Ouvrir avec un éditeur de test et analyser le fichier produit en sortie !

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

- Quelles sont les analyses et les traitements réalisés par le préprocesseur du langage C (s'aider d'internet si nécessaire) ?

- Mettre à 0 la valeur de la macro PRINT_HELLO présente dans le fichier d'en-tête *hello.h* puis répéter les tâches précédentes. Analyser 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)

2.2. Analyse et génération de code natif

- Compiler le programme en partant du code précédemment préparé par le préprocesseur du C puis s'arrêter à l'étape d'assemblage. Parcourir le fichier de sortie. Constater que sa lecture reste complexe.

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

- Ne pas hésiter à nettoyer le code assembleur généré par défaut par GCC et incluant du code de sécurité additionnel. Les options suivantes pourront être ajoutées dans la suite de la trame de Travaux Pratiques à chaque génération et analyse de script assembleur. Compiler à nouveau le programme source *hello.c* et analyser la sortie. Normalement, le code assembleur doit être bien plus léger et n'inclure maintenant que le code assembleur utile au besoin de 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 continu 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++(http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html)

- *-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, etc (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, faîtes un vœux !

Si tout ceci semble bien flou, c'est normal. D'ici quelques mois, tout devrait devenir plus clair. Bien observer qu'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). Nous rencontrons ce type de développement à notre époque 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.

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

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

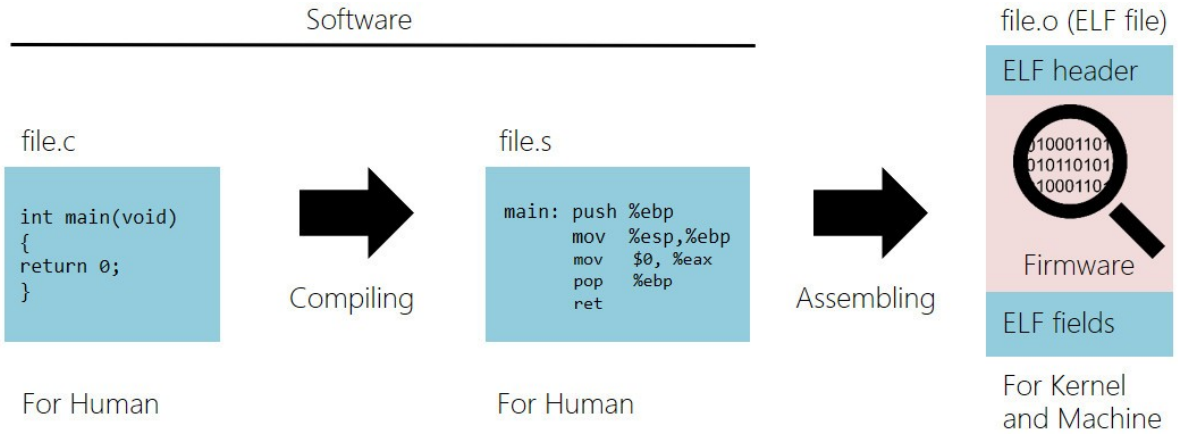
- Qu'est-ce qu'un label en assembleur ? Que représentera plus tard à l'exécution le label *main* (simple chaîne de caractères) ?

- 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 ?

2.3. Assemblage

Assembler le fichier assembleur *build/misc/hello.s* précédemment généré et achever ainsi le processus de compilation. Le fichier binaire résultant est un fichier dit *objet relogeable*, dans notre cas au format ELF 32bits (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*.




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

- Le fichier *hello.o* est un fichier ELF 32bits. 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, préciser l'architecture CPU cible ?

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

- En analysant l'en-tête de fichier ELF, préciser le type de fichier binaire ? Pourquoi nomme-t-on se type de fichier *relogeable* ?

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

2.4. Édition des liens

- Finaliser la génération d'un fichier exécutable par l'édition des liens, puis exécuter 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

readelf -h build/bin/hello
```

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

- 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 ? Pourquoi et surtout comment, la réponse se trouve dans la suite de la trame …

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

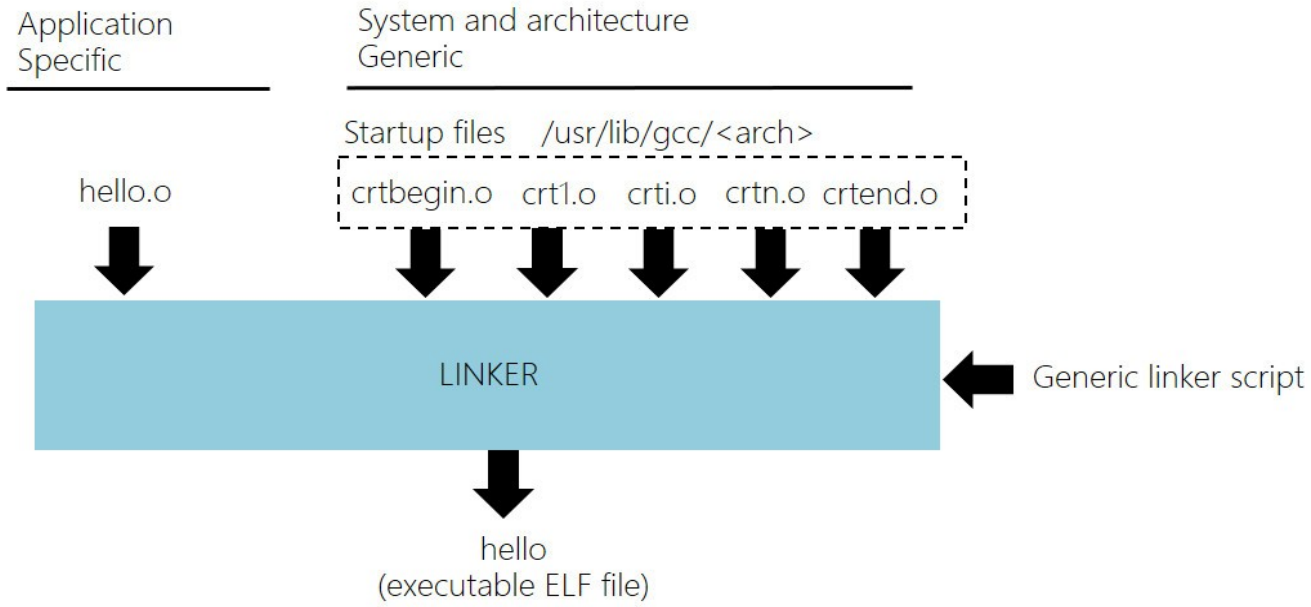
2.5. Startup file

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

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

- Quelle est actuellement la taille sur le support de stockage de masse (HDD, SSD, MMC, etc) du fichier binaire exécutable ELF de sortie ?

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++, etc. 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 (sources des fichiers de startup utilisés par GCC en x64 : https://github.com/gcc-mirror/gcc/tree/master/libgcc/config/ia64). Ils sont donc fortement dépendant de la chaîne de compilation et du système d'exploitation utilisés (schéma ci-dessous sur GCC v7).



- Observer 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
```

```
        .global _start

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

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

- Assembler le nouveau fichier d'amorçage et l'ajouter manuellement durant l'édition des liens en appelant directement le linker *ld* (GNU linker). A ce stade là de nos productions, 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*. Bien penser à placer la macro PRINT_HELLO à 0 dans le fichier d'en-tête *disco/toolchain/include/hello.h*.

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

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

- Analyser 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 fichier binaire désassemblé, quelles sont les adresses des fonctions *main* et *_start* ?

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

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

- Verdict, le fichier *hello* est-il toujours exécutable (sans défaut de segmentation à l'exécution) ?

```
readelf -h build/bin/hello

./build/bin/hello
```

2.6. 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* (https://sourceware.org/binutils/docs/ld/Scripts.html , extension .ld), lui permettant de définir l'ossature du binaire (ou firmware) du fichier ELF exécutable de sortie.

- Parcourir juste à titre indicatif le *linker script* générique utilisé par défaut par l'éditeur de liens de GCC et permettant notamment d'organiser le positionnement des codes binaires des fichiers de *startup* dans le firmware de sortie (fichier riche en informations). La lecture est complexe, c'est normal, ne pas chercher à tout comprendre, ce n'est pas le but !

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

Nous allons maintenant analyser 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 observons-nous ?

- Préciser 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
```

- Ouvrir maintenant le fichier *build/script/linker_script_minimal.ld* et analyser sont contenu. Bien 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 (x86 i386 dans notre cas), le point d'entrée du programme (_start) et le format de fichier de sortie (ELF 32bits intel compatible architecture CPU 386 dans notre cas).

```
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 }
```

- Déclarer 3 variables statiques globales comme ci-dessous (non initialisée, initialisée et en lecture seule). Recompiler le fichier *hello.c* en s'arrêtant à l'édition des liens et utiliser notre *linker script* maison. Nous allons analyser la table des sections générée !

```
#include <stdio.h>

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

int main (void) {

return 0 ;
}
```

- Combien de sections observons-nous ?

```
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

objdump -h build/bin/hello
```

- Préciser leur nom, leur nature, leur adresse de départ et leur taille. Est-ce cohérent ?

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

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

- Par élimination, où se situe la variable ''a'' ? La placer sur le schéma ci-contre

- Quel 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). Nous verrons plus en détail dans la suite de la trame de TP ce que nous venons réellement de faire.

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

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

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

- Verdict, le fichier *hello* est-il toujours exécutable ?

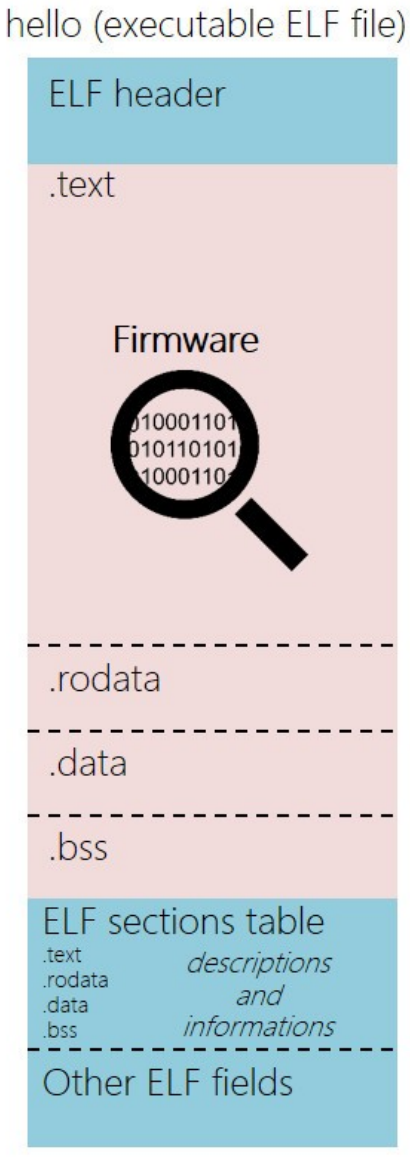
```
./build/bin/hello
```

2.7. Exécution et segmentation

L'exercice touche à sa fin. Cette ultime étape d'observation de notre programme à l'exécution n'a pas pour objectif que d'introduire les exercices suivants. Nous allons introduire le nouveau concept de segment mémoire. Un segment est une zone mémoire contigu virtuelle allouée et mappée par le noyau du système (Linux) en mémoire principale avant exécution d'un programme. En résumé, pour une application en cours d'exécution, il s'agit des emplacements du code et des données en mémoire vive. Ces segments sont alloués dynamiquement et seront différents pour un même programme d'une exécution à une autre.

```
#include <stdio.h>

int main (void) {

     while(1) ;

return 0 ;
}
```

- Modifier le fichier source *disco/toolchain/src/hello.c* en ajoutant une boucle infinie en fin de fonction principale (cf. ci-dessus) afin de rester bloqué dans le programme à l'exécution. Bien penser à placer la macro PRINT_HELLO à 0 dans le fichier d'en-tête *disco/toolchain/include/hello.h*. Recompiler et exécuter le programme. [CTRL] + [c] pour le tuer en fin d'exercice.




```
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

strip build/bin/hello

./build/bin/hello
```

- Récupérer le PID (Process IDentifier) de notre programme en cours d'exécution et observer le mapping en mémoire principale alloué par Linux. Nous pouvons voir un PID comme le nom donné par Linux à un programme durant son exécution en mémoire principale.

```
ps -a

cat /proc/<hello_pid>/maps
```

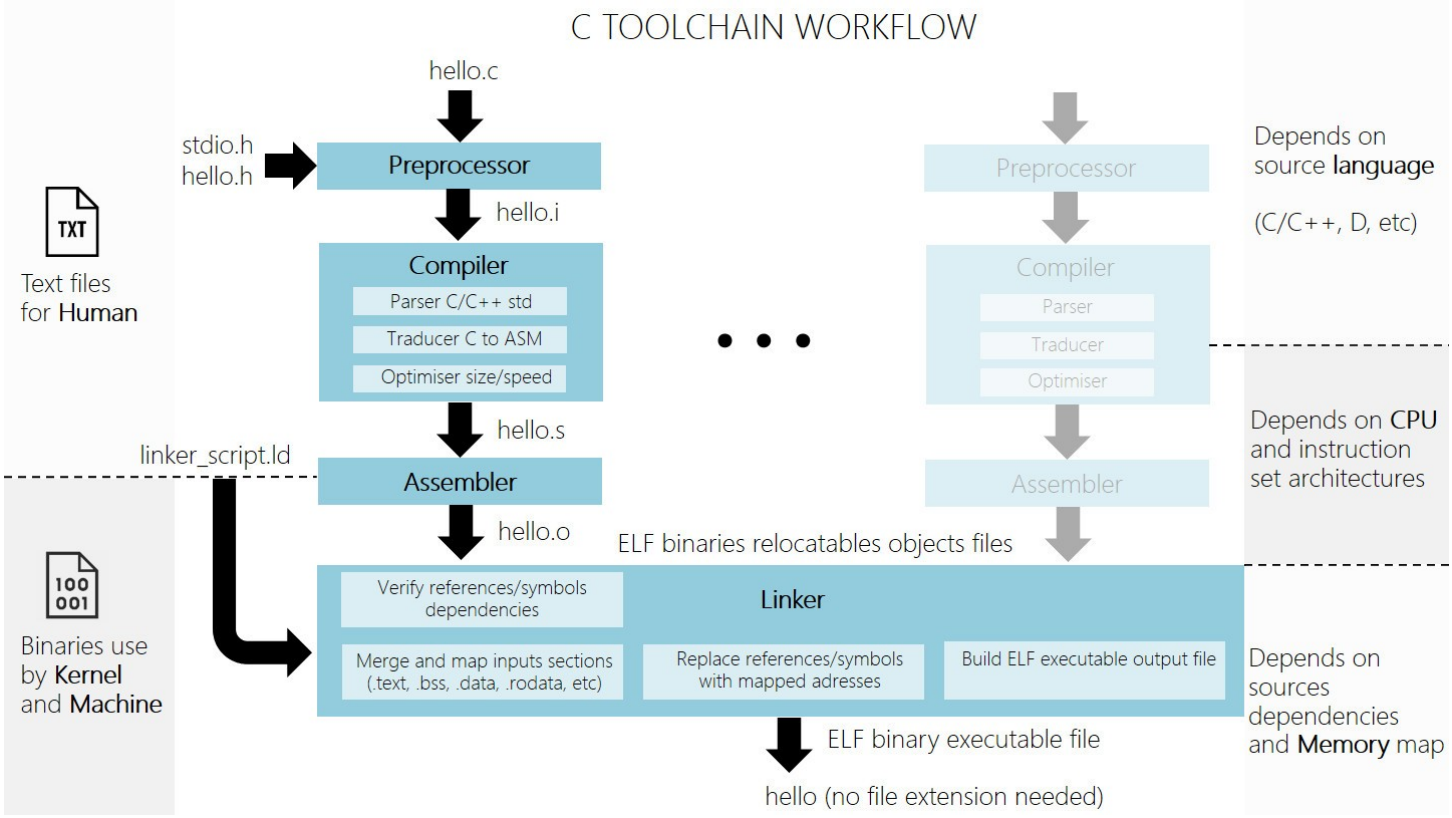
Vous devriez observer 4 segments mémoire distincts (les plages d'adresses virtuelles sont propres à votre application) : /<your_path>/disco/toolchain/build/bin/hello (code binaire), [stack], [vVAR] et [vDSO]. Les segments vVAR et vDSO (virtual Dynamic Shared Object) sont des ponts potentiels entre le noyau Linux et votre application (non directement étudiés dans cet enseignement). En revanche, l'usage du segment de pile (stack) sera pleinement étudié dans l'exercice suivant.

- Quelles sont les tailles des segments de pile (stack) et de code (/<your_path>/hello) ?

2.8. Synthèse

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, etc) sur un système GNU\Linux 32bits et sur architecture matérielle 32bits 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 la compétence. 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 !

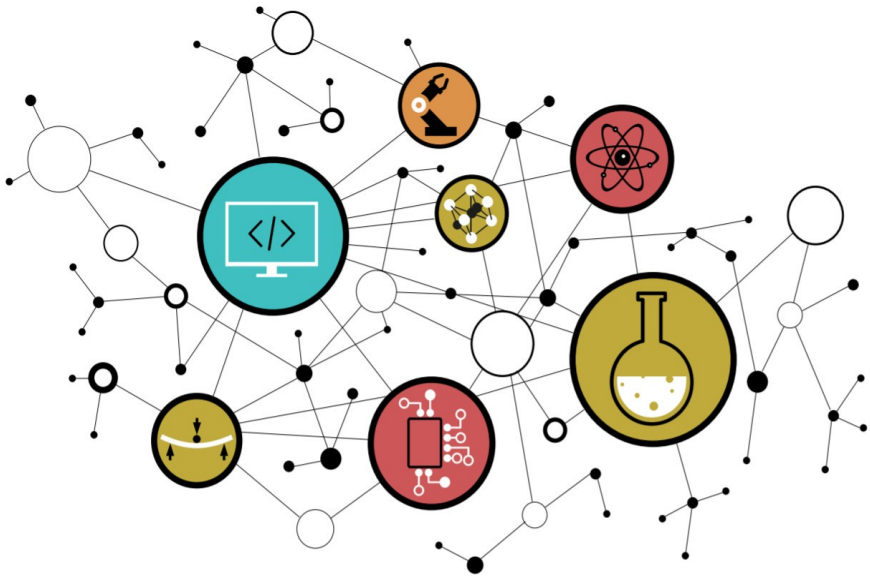



```
gcc -E -m32 -I./inc src/hello.c > build/misc/hello.i

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
```


notes

TRAVAUX PRATIQUES

ASSEMBLEUR X86 ET BIBLIOTHÈQUE STATIQUE



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

SOMMAIRE

3. ASSEMBLEUR X86 ET BIBLIOTHÈQUE STATIQUE

```
int main (void)
{

return 0 ;
}
```

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

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

```
label:    instruction    opérandes
```

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

Syntaxe assembleur AT&T proposée par GNU AS

Syntaxe Intel	Syntaxe AT&T

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

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

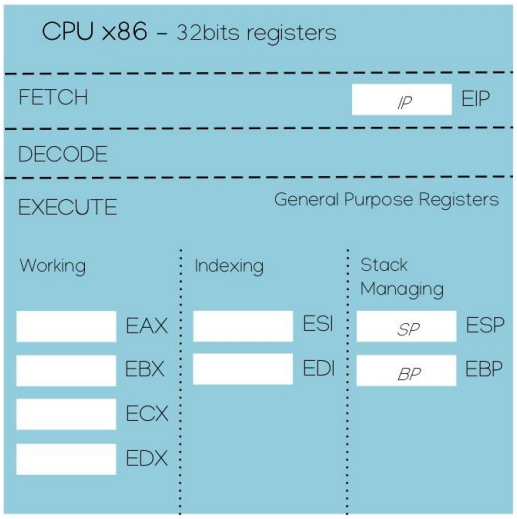
Le langage C et sa syntaxe sont maintenant normalisés et standardisés depuis des décennies même si la norme continue d'évoluer (normes ANSI C, C89, C99, C18, etc). En revanche, différentes syntaxes assembleur liées à la chaîne de compilation et aux outils de développement (ouverts ou fermés, libres ou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

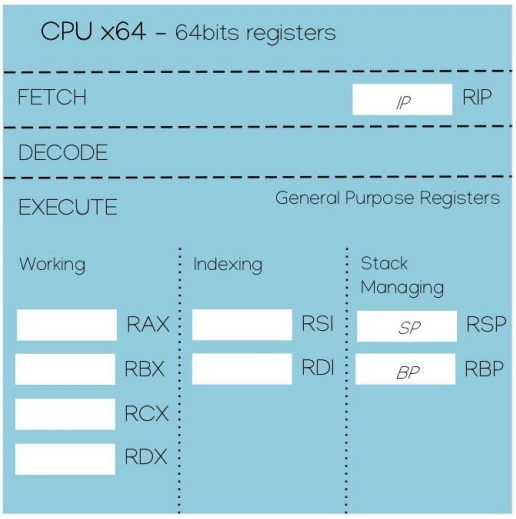
- % : signifie que l'opérande qui suit est un registre CPU (mode d'adressage registre)
- $: signifie que l'opérande que suit est une constante (mode d'adressage immédiat)
- suffixe d'instruction : signifie que l'instruction manipule des opérandes d'une certaine longueur en octets : b=byte=8bits=1octet,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

```
    .global main

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

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

Registres CPU x86-64 et environnement d'exécution assembleur

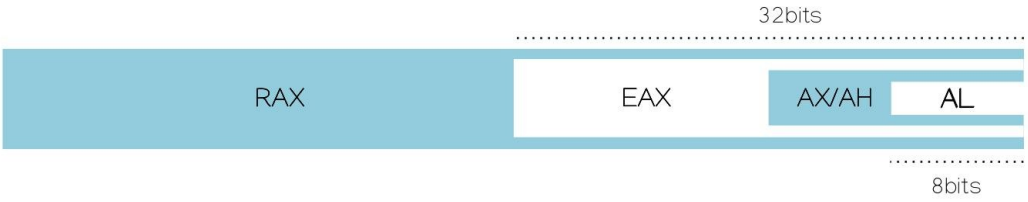




Les schémas ci-dessus présentent les principaux registres d'usages généralistes présents dans les CPU d'architectures compatibles x86 (32bits) et x64 (64bits). Rappelons que ces architectures restent rétrocompatibles avec le CPU 16bits 8086 datant de 1978. Ces registres sont présents dans chaque CPU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

- Registre d'état FLAGS (CF, PF, ZF, etc) : registre contenant les différents flags d'états associés aux unités d'exécution arithmétiques et logiques (carry, zero, sign, etc). Utilisé notamment afin d'implémenter des sauts conditionnels (if, else if, while, for, etc).
- Registres de contrôle (CR0 à CR7) : registres permettant de contrôler les services matériels du CPU (mode protégé, etc), ainsi que du CACHE et de la MMU associés au cœur courant.
- Registres vectoriels (XMM0 à XMM15 128bits extension ISA SSE, YMM0 à YMM15 256bits extension ISA AVX et ZMM0 à ZMM15 512bits extension ISA AVX-512) : registres de travail pour les instructions vectorielles dans un contexte d'optimisation algorithmique
- Registres de segments (CS, DS, ES, SS, FS, GS) : registres historiquement utilisés lorsque la segmentation logique d'une application nécessitait un support d'adressage physique. La segmentation est maintenant virtualisée et gérée entièrement logiquement par le noyau du système à travers la gestion de la PMMU (Paged MMU ou unité de pagination).
- Registres de test (TR3 à DR7), registres de debug (DR0 à DR7) et registres d'accès à la table de pagination mémoire (GDTR, LDTR, IDTR)

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



3.1. 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
```

- Se placer dans le répertoire de travail *disco/asm*. Assembler les fichiers *logic_arithmetic.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal). En continuité du chapitre sur la compilation et l'édition des liens, utiliser le script *linker* minimal et réaliser l'édition des liens du projet. Analyser le programme assembleur *logic_arithmetic.s*

```
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
```

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

- Des deux instructions implémentant un même traitement proposées à la question précédente (XOR vs MOV $0), laquelle offre un 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 ?

3.2. Debugger GDB

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 GNU (http://www.gdbtutorial.com/).

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

- Assembler jusqu'à l'édition des liens incluse le fichier *logic_arithmetic.s*. Ne pas oublier d'ajouter l'option de debug *-g* à l'assemblage. Lancer ensuite une session de debug avec GDB

```
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

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 élémentaires mais très puissants d'analyse. A force de persévérance, aucun *bug* ne peut se cacher indéfiniment !

- Suivre et comprendre la séquence de commandes GDB proposée ci-dessous

```
(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
```

3.3. 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
```

- Modifier le fichier *logic_arithmetic.s* afin d'appeler la fonction *itoa* (integer to string conversion). Assembler les fichiers *logic_arithmetic.s*, *itoa.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal) puis réaliser l'édition des liens du projet. Analyser le projet assembleur

```
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* ? Préciser son contenu au démarrage du programme. Proposer une écriture équivalente en langage C. *Constater 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), vérifier la valeur contenu dans EAX après conversion par la fonction *itoa* et avant la fin de la fonction *main* (instruction RET). Vérifier 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
```

3.4. 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
```

- Modifier le fichier *logic_arithmetic.s* afin d'appeler la fonction *printf*. Assembler les fichiers *logic_arithmetic.s, itoa.s, printf.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal) puis réaliser l'édition des liens du projet. Exécuter le binaire de sortie et analyser 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 3 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 32bits x86 permettant de donner la main au noyau Linux en lui passant des arguments par registres. Il s'agit d'une interruption logicielle (interrompre l'exécution synchrone d'un programme). Un appel système en assembleur 64bits x64 est implémenté par l'instruction *syscall*. 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, etc) sur laquelle est portée le Système d'exploitation (Distributions GNU\Linux, Mac OS X, Windows, etc). Observons les opérandes en x86 sous Linux (http://www.lxhp.in-berlin.de/lhpsysc0.html) :

- EAX : Fixe la fonction noyau à exécuter et donc la nature de l'appel système. Fonction *write* dans notre cas
- EBX : Modes 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

3.5. 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
```

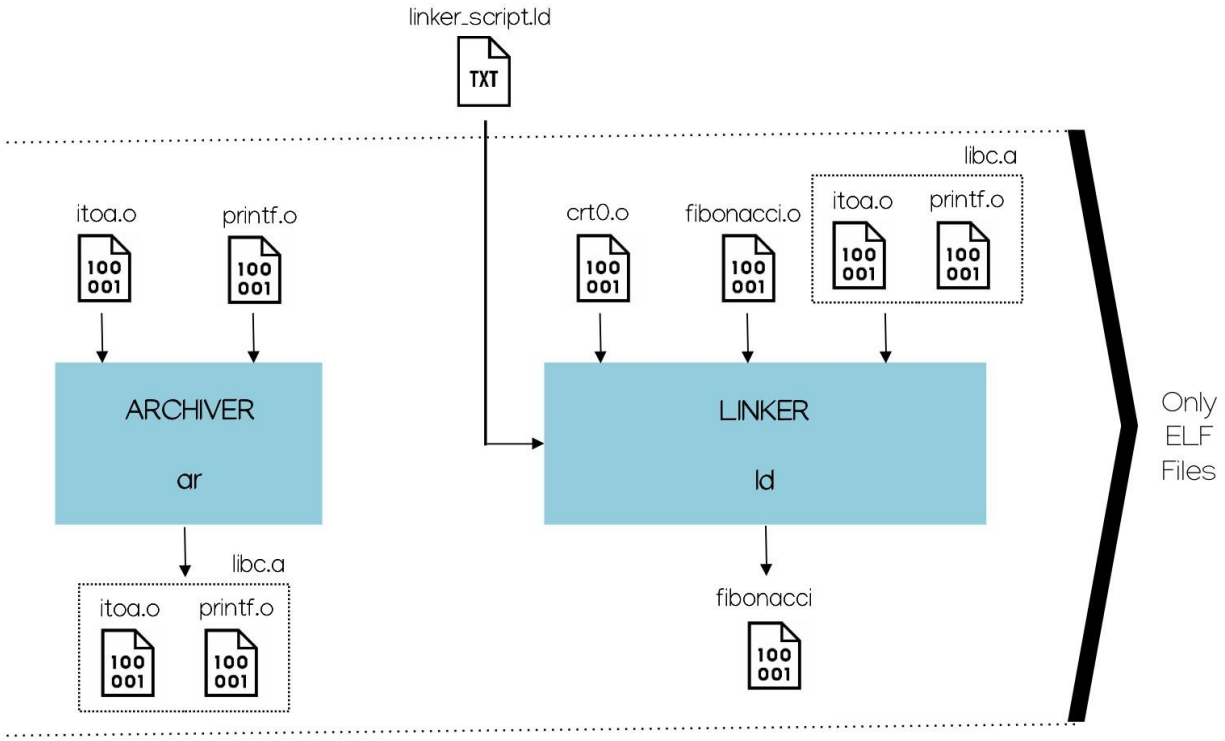
- Assembler les fichiers *fibonacci.s, itoa.s, printf.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal) puis réaliser l'édition des liens du projet. Exécuter le binaire de sortie et analyser le projet assembleur. S'aider de GDB pour votre analyse

```
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 *iota* et *printf* ? *Constater que l'utilisation d'une macro allège la lecture du programme*

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

3.6. Bibliothèque statique



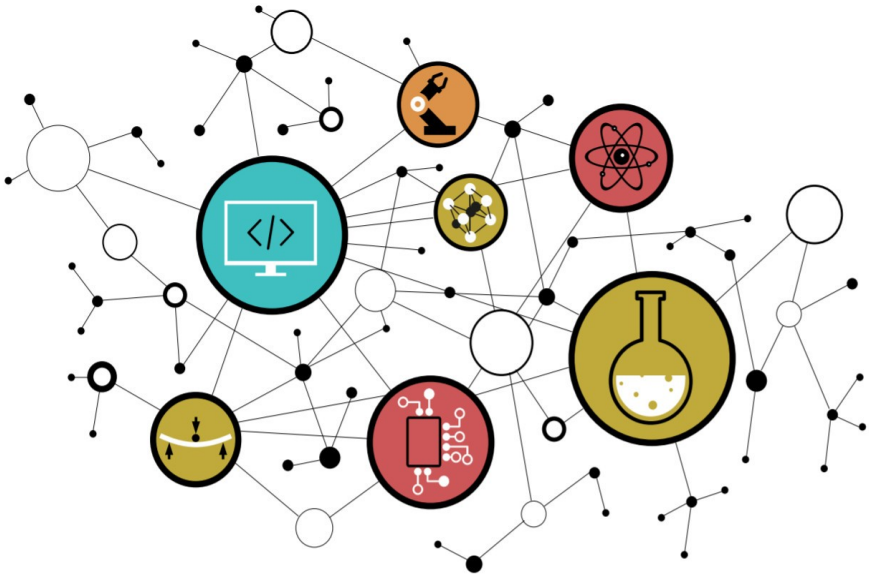
- Générer une bibliothèque statique (GNU AR ou *archiver*) et la lier manuellement durant l'édition des liens. Une bibliothèque statique est une archive (généralement avec une extension .a comme archive), soit la concaténation de fichiers objets binaires ELF ré-adressables. Une bibliothèque statique n'est pas un fichier ELF, mais en revanche elle regroupe de fichiers ELF. Analyser le processus de compilation et d'édition des liens du projet. Valider l'exécution du binaire de sortie. *Observer avec le service readelf de binutils le contenu de notre modeste bibliothèque statique nommée pour l'occasion libc.a, à l'image de la bibliothèque standard du C libc.so rangée dans /lib dans une arborescence de fichiers Unix-like. Au démarrage du système, la bibliothèque standard libc.so est chargée en mémoire principale puis partagée par tous les programmes en cours d'exécution sur la machine ayant besoin d'exécuter des fonctions standards (dont le processus init).*

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

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

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

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

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

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

readelf -h lib/libc.a
```


notes

TRAVAUX PRATIQUES

ALLOCATIONS AUTOMATIQUES ET SEGMENT DE PILE

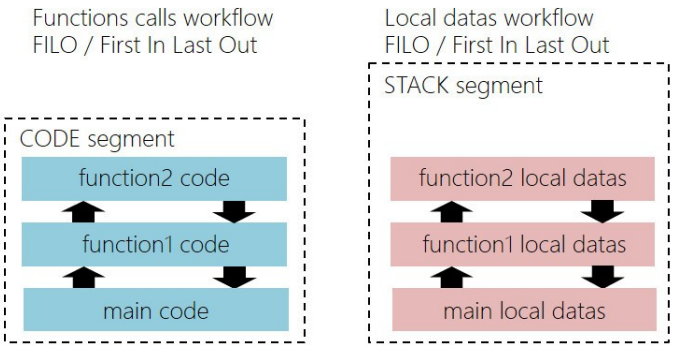


Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

SOMMAIRE

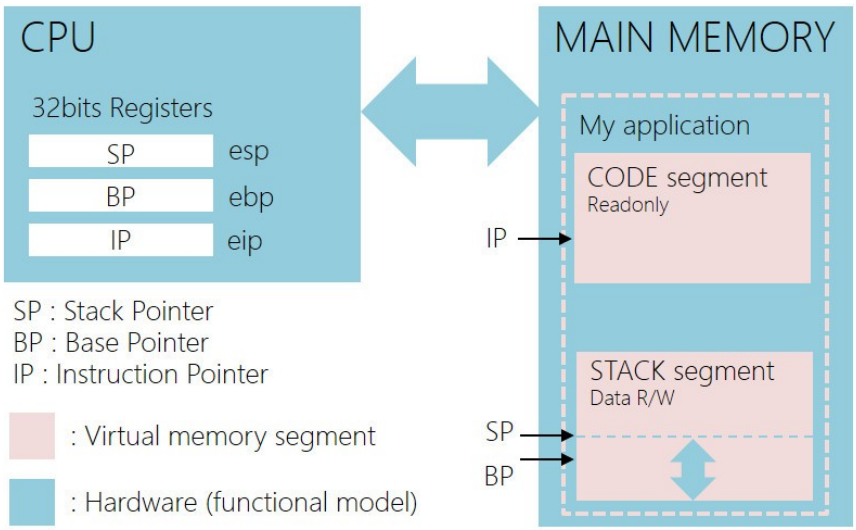
4. ALLOCATIONS AUTOMATIQUES ET SEGMENT DE PILE

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

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

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



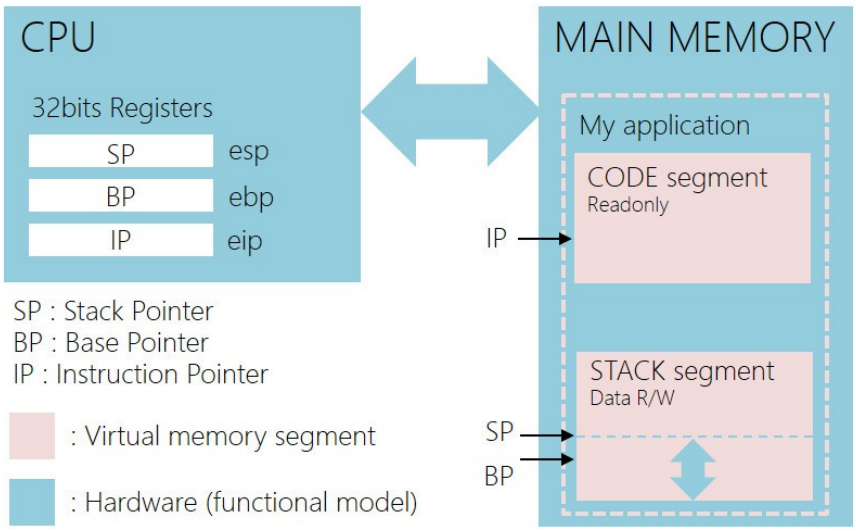
Les allocations de ressources mémoire réalisés dynamiquement à l'exécution durant la manipulation de variables locales se font sur le segment de pile ou *stack*. Ce segment est présent en mémoire principale durant l'exécution d'un programme. *Un segment est une zone logique contigu virtuelle allouée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

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



4.1. Fonction main

Se placer dans le répertoire *disco/stack/* afin d'appliquer la totalité des commandes qui suivent. Nous analyserons de l'assembleur 32bits x86 (option -m32). Par la suite, nous n'analyserons plus que de l'assembleur 64bits x64. L'objectif étant de pouvoir aisément distinguer les 2 technologies et de constater qu'il n'existe pas de grande différence fondamentale à la lecture de script assembleur élémentaire.



Il est important de noter que le segment de pile possédera toujours une taille fixe (8Mo par défaut sous Linux). Sur machine x86 32bits, SP (Stack Pointer) est toujours sauvé dans le registre CPU 32bits *ESP* et pointe toujours le sommet de la pile. BP (Base Pointer ou Frame Pointer) est toujours sauvé dans le registre CPU 32bits *EBP* et est propre à chaque fonction. Tant que nous nous trouvons dans une fonction, ce pointeur ne sera pas modifié et pointera vers la même zone mémoire. IP (Instruction Pointer ou PC ou Program Counter) sera toujours sauvé dans le registre CPU 32bits *EIP* et pointera toujours la future instruction à exécuter. Pour information, par convention, nous représentons toujours un *mapping* mémoire avec les adresses basses en haut de page et les adresses hautes en bas en bas de page (cf. schéma suivant).

- Compiler le fichier *main.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré. S'aider d'internet, de vos voisins de table, de l'enseignant encadrant, etc.

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

- Compléter (au crayon) 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.

- Proposer 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*

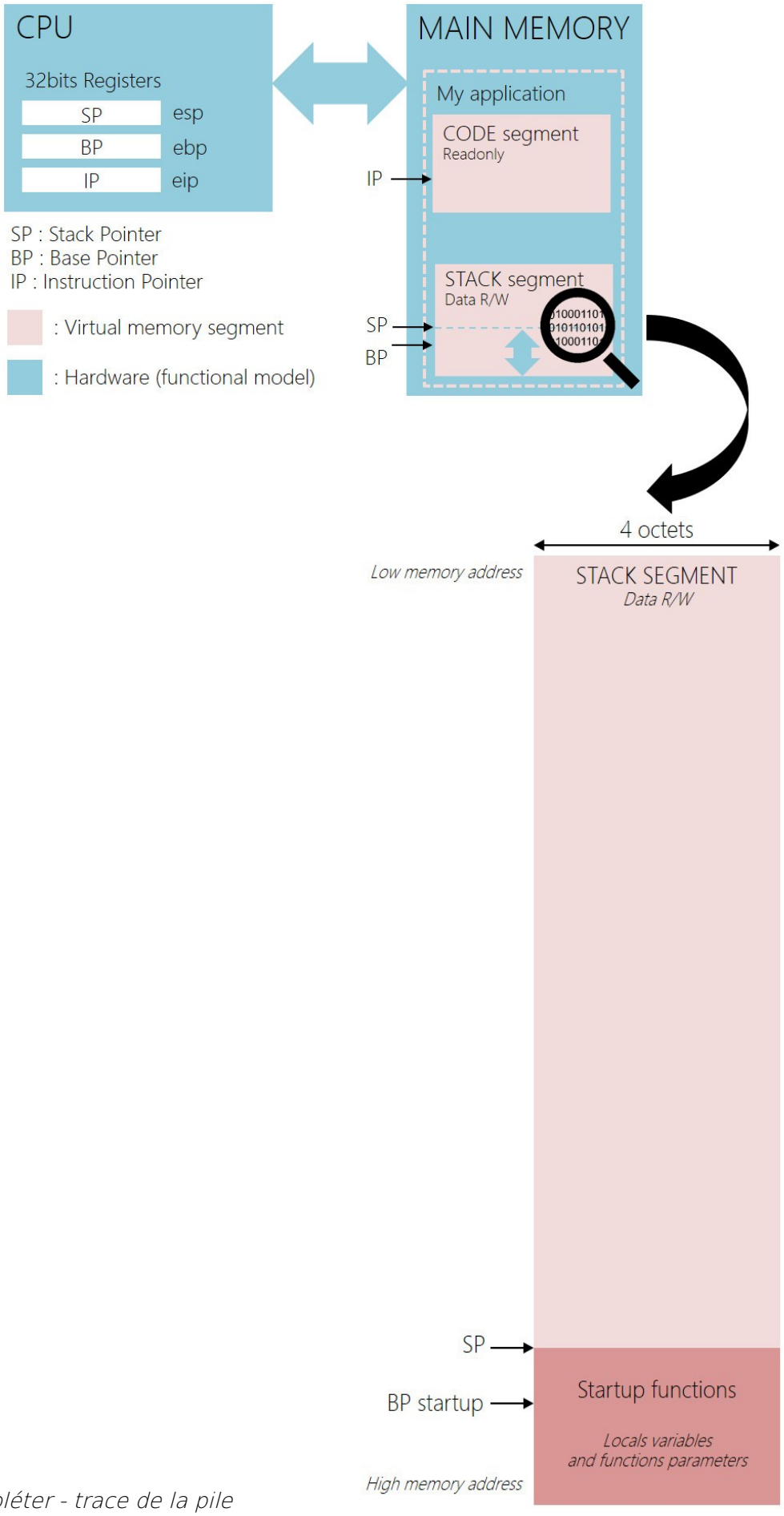


Schéma à compléter - trace de la pile

4.2. Variables locales initialisées

- Ouvrir puis compiler le fichier *local_variable_init.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

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

- Compléter (au crayon) le schéma de la pile sur la page précédente en précisant quel est son contenu suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*.
- Préciser les tailles ou empreintes mémoire des variables locales a,b,c et d
- Préciser 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 ?
- Dé-commenter le *cast* (transtypage) présent dans le programme, compiler et analyser le fichier assembleur de sortie. Analyser le résultat

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

- Qu'est-ce qu'une extension de signe en arithmétique entière signée Cà2 (Complément à 2) ?
- Qualifier le type de la variable *a* de *const*. Compiler le programme puis interpréter le résultat. Que constatons-nous ?
- Quel est le rôle du qualificateur de type *const* et donc son usage ?

4.3. Variables locales non-initialisées

- Ouvrir puis compiler le fichier *local_variable_uninit.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

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

- Que constatons-nous ?

- Ajouter le qualificateur de type *volatile* devant chaque déclaration, compiler le programme puis interpréter le résultat.

- Quel est le rôle de ce qualificateur de type *volatile* et donc son usage ? S'aider d'internet

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).

4.4. Appel et paramètres de fonction

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

- Ouvrir puis compiler le fichier *function_parameters.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

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

- Compléter (au crayon) le schéma de la pile sur la page suivante en précisant quel est son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*.

Les instructions d'appel de fonction comme *call* empilent également l'adresse de retour sur la pile en plus de réaliser un saut vers le code de la fonction appelée. Ce mécanisme permet d'assurer les appels et surtout les retours de fonctions. L'adresse de retour est l'adresse de l'instruction suivant spatialement le *call* en mémoire programme. Durant l'exécution d'un call, il s'agit donc du pointeur *IP* contenu dans le registre CPU *RIP* pointant toujours l'instruction suivante à exécuter. Ceci permettra après exécution de l'instruction *ret* présent dans la fonction appelée de revenir exactement dans la fonction appelante à l'instruction suivant spatialement en mémoire l'instruction *call*. L'instruction *ret* présente dans la fonction appelée dépile l'adresse de retour précédemment sauvée et la restaure dans le registre d'instruction du CPU (registre *RIP* dans notre cas). Rappelons que sur architecture x86/x64 (32bits/64bits), les registres EIP/RIP sont utilisés par l'étage de *fetch* du CPU afin d'aller chercher en mémoire programme les futures instructions à exécuter. 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 ? 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 ?

- Observer 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 ?

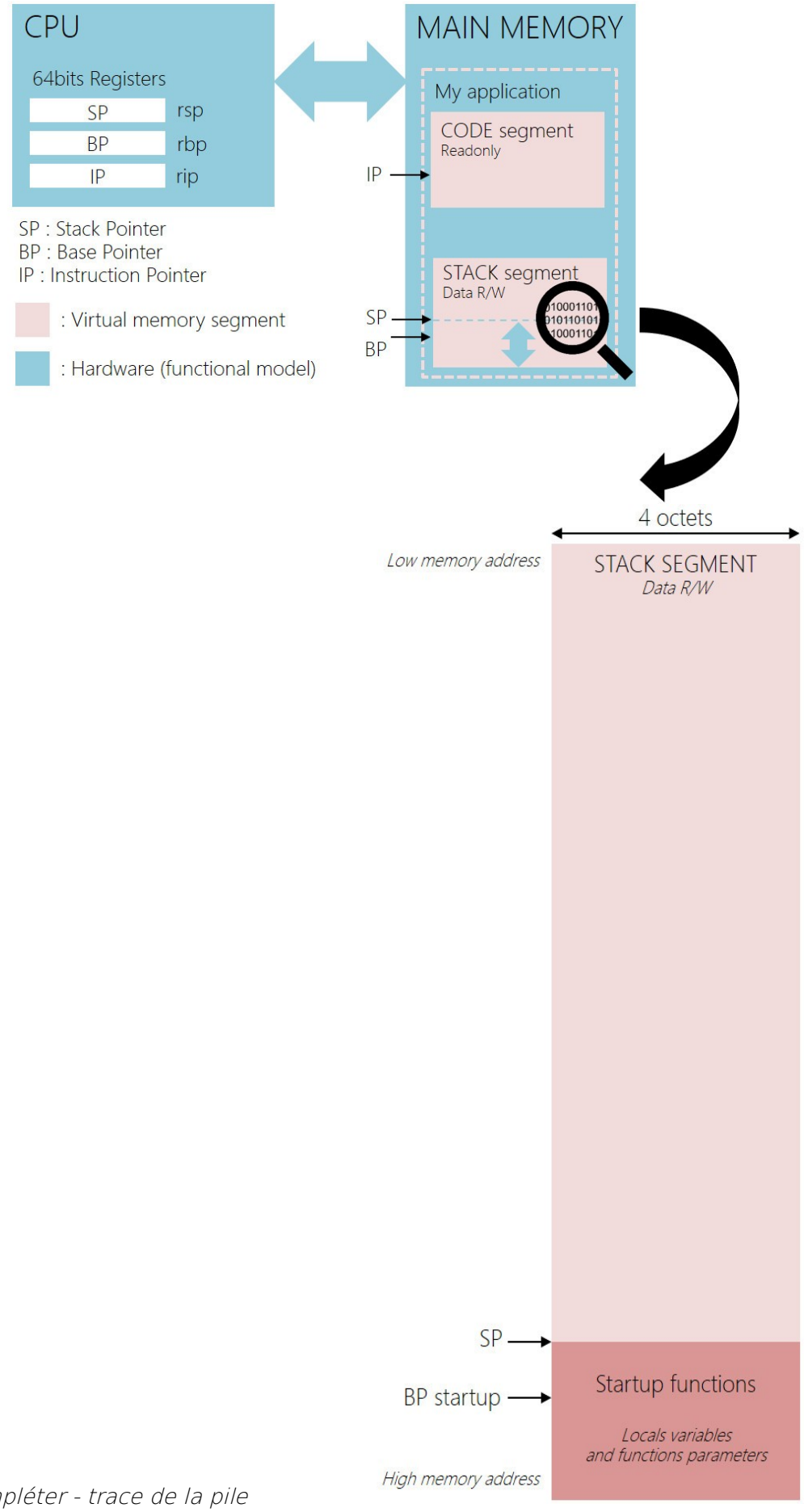


Schéma à compléter - trace de la pile

4.5. 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.

- Ouvrir puis compiler le fichier *function_inlining.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

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

- Compléter (au crayon) le schéma de la pile sur la page suivante en précisant quel est son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*.

- Dé-commenter le prototype de la fonction *swap* utilisant la classe de stockage *register* pour la déclaration des paramètres de fonction et commenter le prototype générique. Faire de même au niveau de la définition de fonction. Qualifier également la variable *tmp* dans la fonction *swap* de *register* (à laisser jusqu'à la fin de l'exercice). compiler et analyser 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 qualificateur 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).

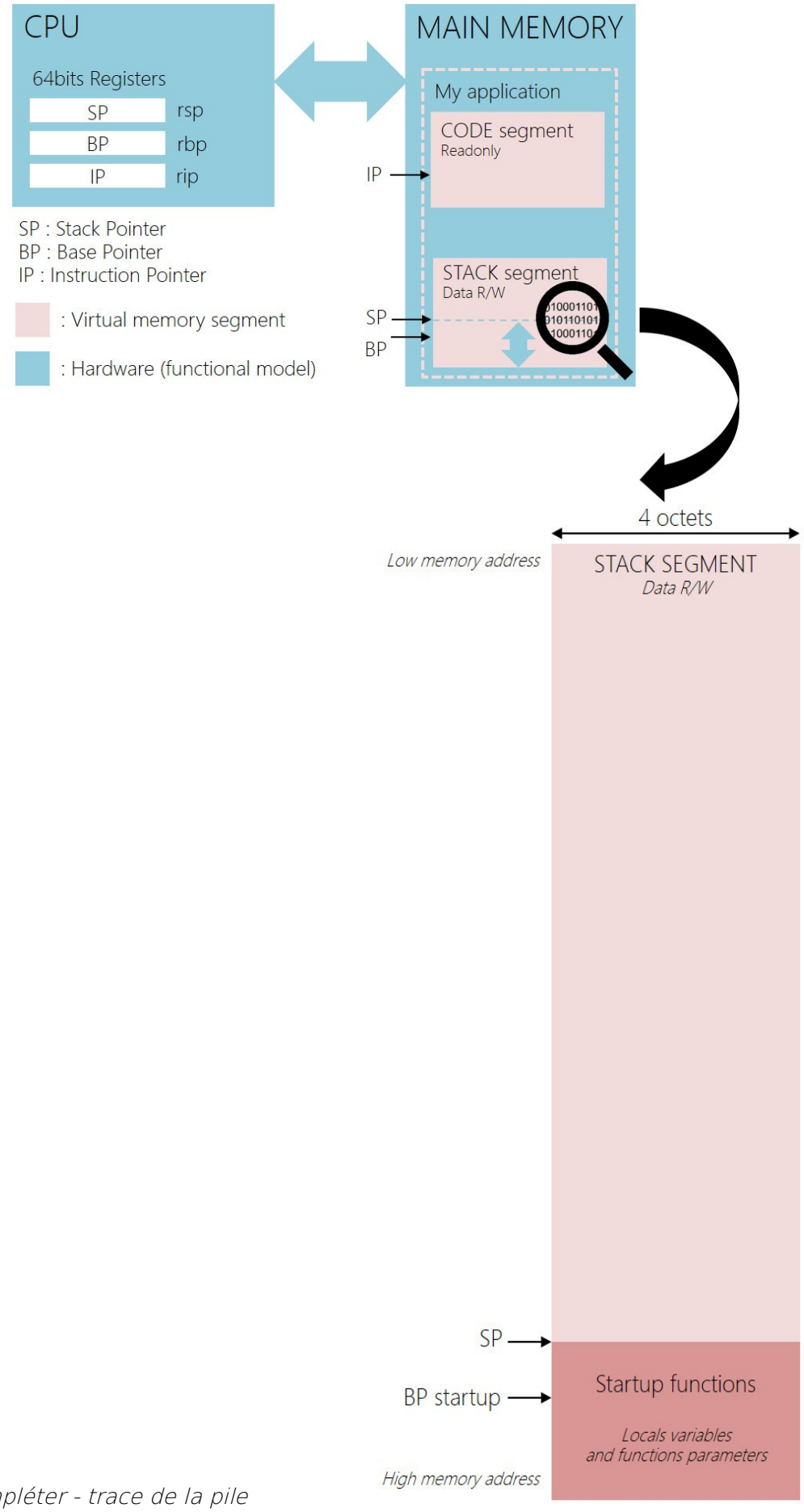


Schéma à compléter - trace de la pile

- Dé-commenter le prototype de la fonction *swap* qualifiée de *inline* (seulement à la déclaration) et commenter les prototypes génériques. Analyser 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* ?

```
//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));
```

- Dé-commenter le prototype de la fonction *swap* qualifiée de *inline* au niveau de la définition de la fonction *swap* et commenter les prototypes génériques. Analyser le code assembleur généré. Verdict ?

```
//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.

- Lever les options d'optimisation de niveau 1 (-O1) de GGC en laissant dé-commentée la fonction qualifiée de *inline* (déclaration et définition). Compiler et analyser la sortie. Observer 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
```

- Essayer le niveau maximal d'optimisation -O3 et analyser le programme de sortie.

4.6. Limites de la pile

- Nous allons maintenant tester les limites de notre pile applicative. Se déplacer dans le répertoire *disco/except/. C*ompiler le fichier *stack_overflow.c* puis l'exécuter.

```
gcc stack_overflow.c –o stack_overflow

./stack_overflow
```

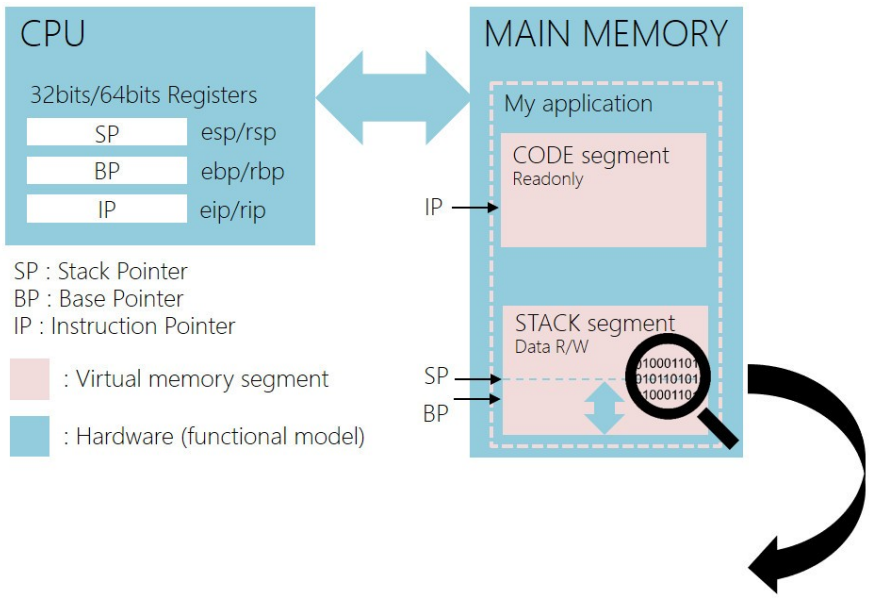
- Quel segment logique mémoire virtuel applicatif a subit un débordement et a causé la fin de notre programme ? Question facile !

- Qu'elle 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-commenter la section de code présente dans le programme. Compiler à nouveau le fichier *stack_overflow.c* puis l'exécuter. Qu'elle est la nouvelle taille de la pile associée à notre programme ? Analyser le code dé-commenté.

La fonction *setrlimit* (set application ressources 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.

4.7. Synthèse

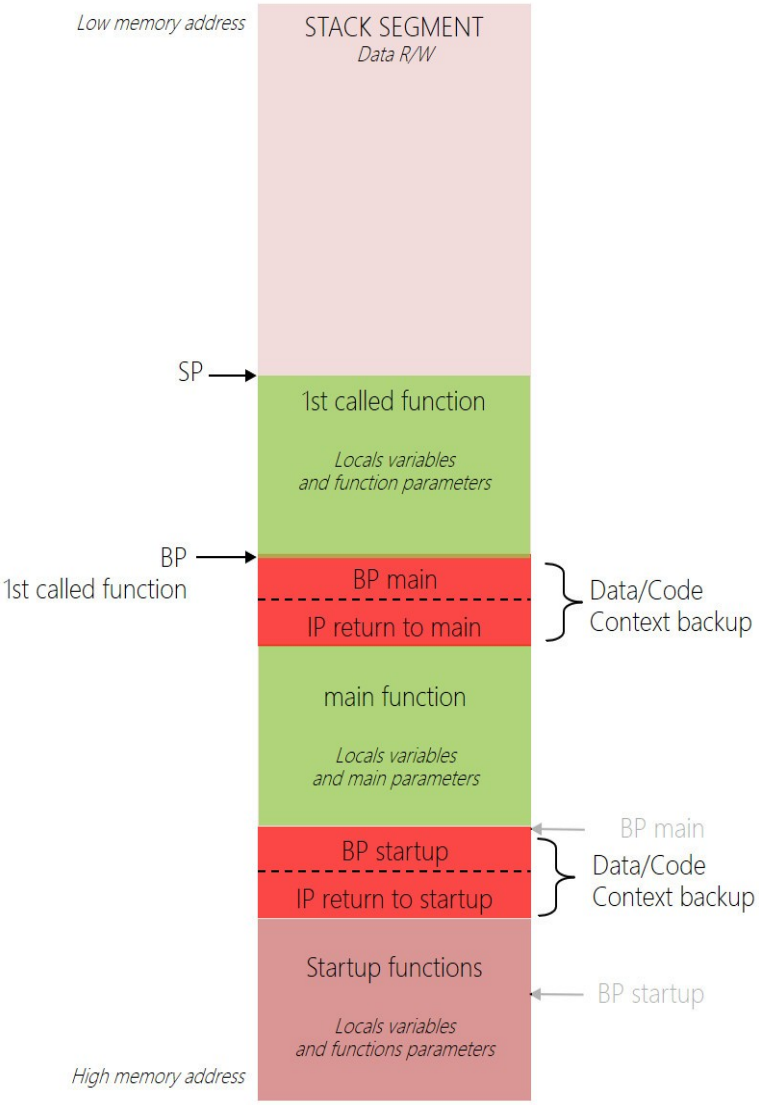


Les processus de gestion des variables locales et des paramètres de fonction (eux même des variables locales paramétrées) sont conjointement réalisés à la compilation par les outils de développement et exploités à l'exécution par la machine.

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

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

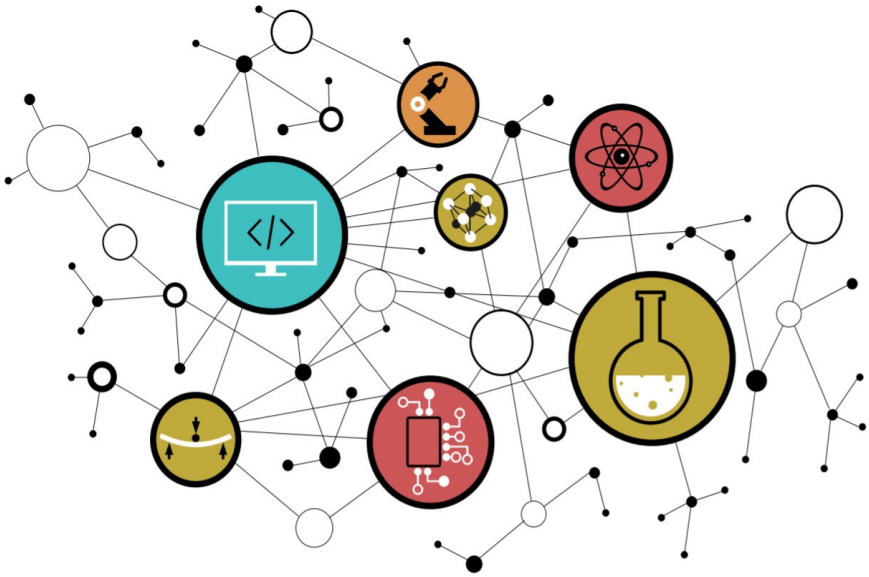
Quant à lui, le segment de pile possédera toujours une taille fixe. Sur ordinateur, rappelons qu'un segment est une zone virtuelle contigu allouée en mémoire principale par le noyau du système à l'exécution. Ce segment n'admet aucune existence sur les médias de stockage de masse (HDD, SSD, etc).




notes

TRAVAUX PRATIQUES

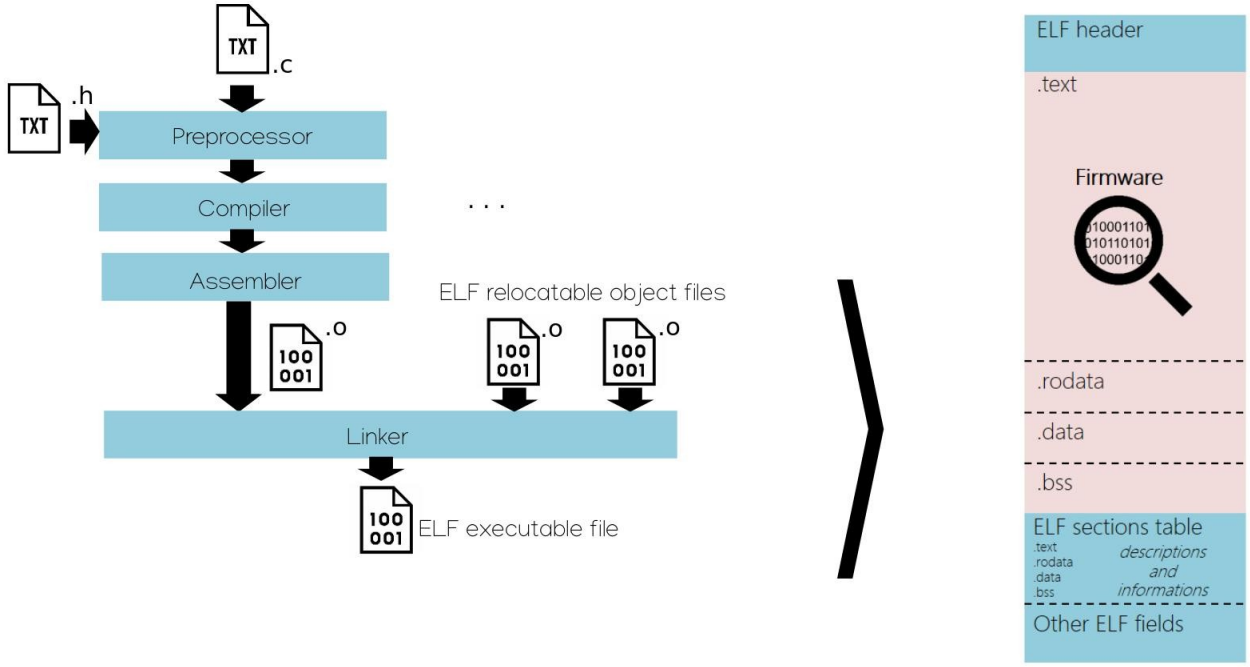
ALLOCATIONS STATIQUES ET FICHIER ELF



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

SOMMAIRE

5. ALLOCATIONS STATIQUES ET FICHIER ELF



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

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

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

5.1. Variables globales

- Se placer dans le répertoire de travail *disco/static*. Compiler le fichier *global_variable.c* jusqu'à l'édition des liens incluse. Préciser la taille du fichier binaire ELF exécutable de sortie ? *Constater par la suite qu'enlever ou mettre le qualificateur de type static à la déclaration de la variable globale n'a aucun impact sur le code généré. En effet, les variables globales sont implicitement et par essence des variables allouées statiquement.*

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

ls -l
```

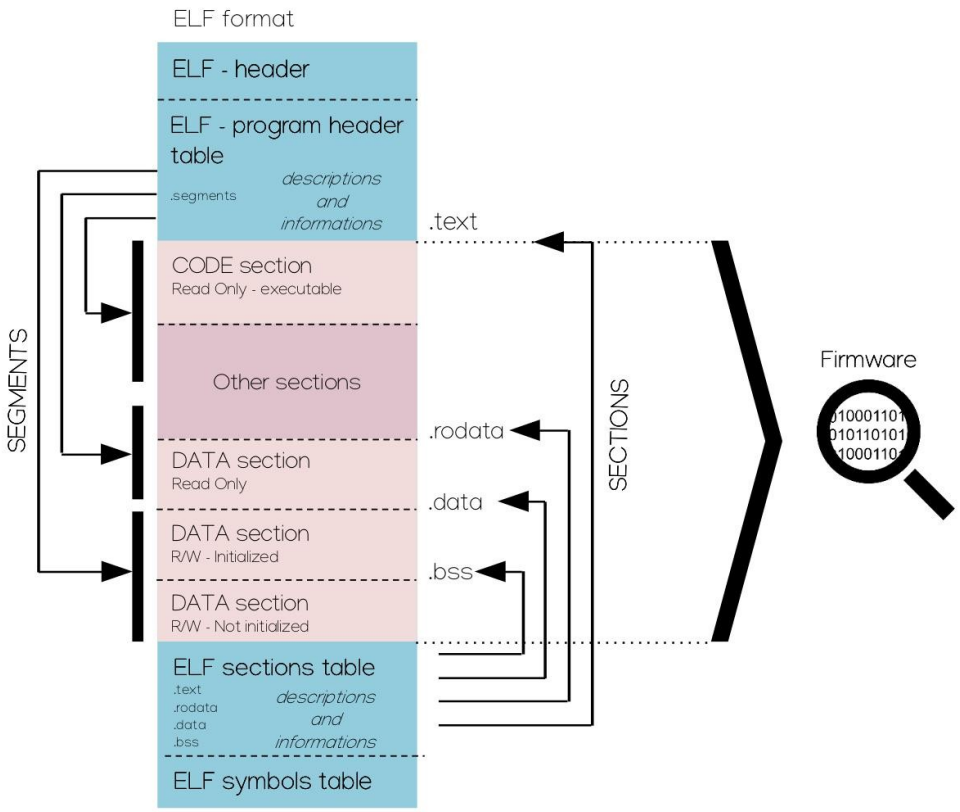
- Compiler le fichier *global_variable.c* en s'arrêtant à l'édition des liens. Préciser la taille du fichier objet binaire ELF relogeable de sortie ? Préciser le nombre de sections applicatives, leurs noms ainsi que leurs tailles ? Dans quelle section se trouve le tableau statique *tab* ? *Constater qu'après la compilation, aucune section n'est mappée en mémoire (adresse de base nulle). Elle seront relogées/mappées à l'édition des liens.*

```
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
```

- Compiler le fichier *global_variable.c* en s'arrêtant à l'assemblage. Préciser le nom de la référence symbolique (label ou étiquette) représentant l'adresse du tableau *tab* ? *Constater que le tableau n'est pas explicitement placé en mémoire (travail de l'édition des liens) mais qu'il est en revanche explicitement spécifié dans le script assembleur qu'il se situe dans la section .data.*

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

- Le *label* (ou étiquette ou référence symbolique) *main* se situe dans quelle section ? *Constater 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).*



- Compiler le fichier *global_variable.c* jusqu'à l'édition des liens incluse. En observant la table des symboles (objdump -t, table contenant des informations sur toutes les références symboliques statiques dont leurs adresses logiques), préciser l'adresse relative du tableau *tab* après édition des liens ? Analyser le code du programme après désassemblage (objdump -S) et retrouver l'adresse précédemment trouvée dans le code ?

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

objdump -t global_variable

objdump -S global_variable
```

- Observer la taille du fichier exécutable de sortie, le nettoyer (stripper) puis ré-observer sa taille sur le média de stockage de masse. Observer 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
```

5.2. Variables locales statiques

- Compiler le fichier *local_static_variable.c* en s'arrêtant à l'assemblage. Préciser le nom de la référence symbolique (label ou étiquette) représentant l'adresse de la variable *a* ?

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

- La variable locale statique *a* se situe-t-elle sur la pile ?

- Dans quelle section se situe la variable locale statique *a* ?

- Une variable locale statique, tout comme une variable locale standard, ne possède qu'une portée locale à la fonction où elle a été déclarée. En revanche, une variable locale statique mémorise la dernière valeur affectée, même si l'on quitte la fonction et qu'on la rappelle après avoir exécuter le code d'autres fonctions. Ceci est impossible avec une variable locale standard. Pourquoi est-ce possible avec une variable locale statique ?

5.3. Chaînes de caractères

- Compiler le fichier *string.c* en s'arrêtant à l'assemblage.

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

- Justifier la taille de la variable *gnu_tab* ? Où est allouée ce tableau ?

- Où (segment ou section) est allouée la chaîne de caractères *GNU's Not Unix !* ? Ma question est-elle rigoureusement formulée ?

- Justifier la taille de la variable *gnu_pointer* ?

- Où (segment ou section) est allouée la chaîne de caractères *GNU's design is Unix-like but differs by being free software and containing no Unix code !* ? S'agit-il d'une allocation statique ou d'une allocation dynamique sur la pile ?

- Observer les chaînes de caractères en observant les contenus binaires des sections applicatives du fichier ELF relogeable après compilation mais avant édition des liens.

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

objdump -s string.o
```

Pour conclure, bien se souvenir que dans un fichier binaire exécutable (formats ELF, COFF, PE, etc), 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, etc) avant même que le programme soit exécuter en mémoire principale.

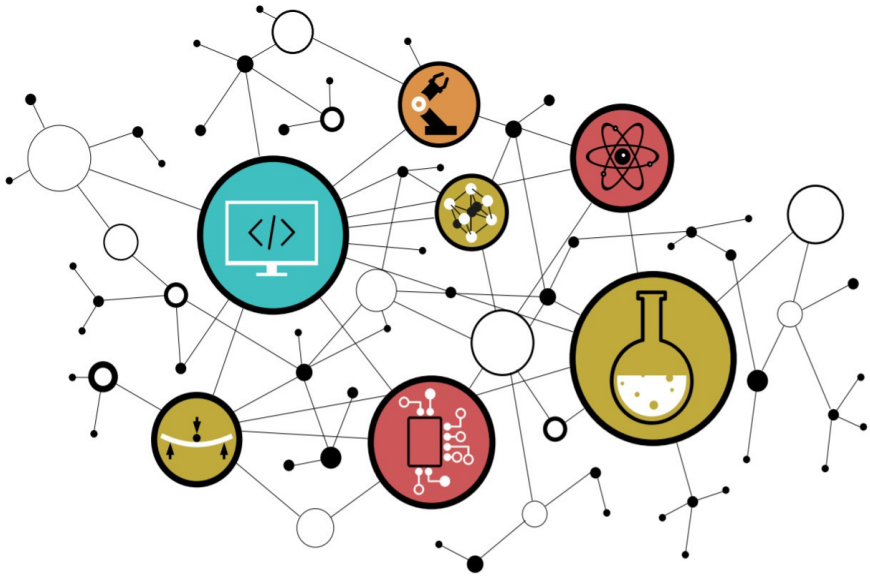
Lorsque nous exécutons un programme, le noyau du système (Linux, GNU Hurd, Mach, XNU, etc) va analyser les différents champs du fichier ELF exécutable (header, header du programme pour définir les futurs segments en mémoire vive, etc). Après analyse, le système va mapper et allouer en mémoire vive les 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 mappé, le code et les données statiques chargées en mémoire principales 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.


notes

TRAVAUX PRATIQUES

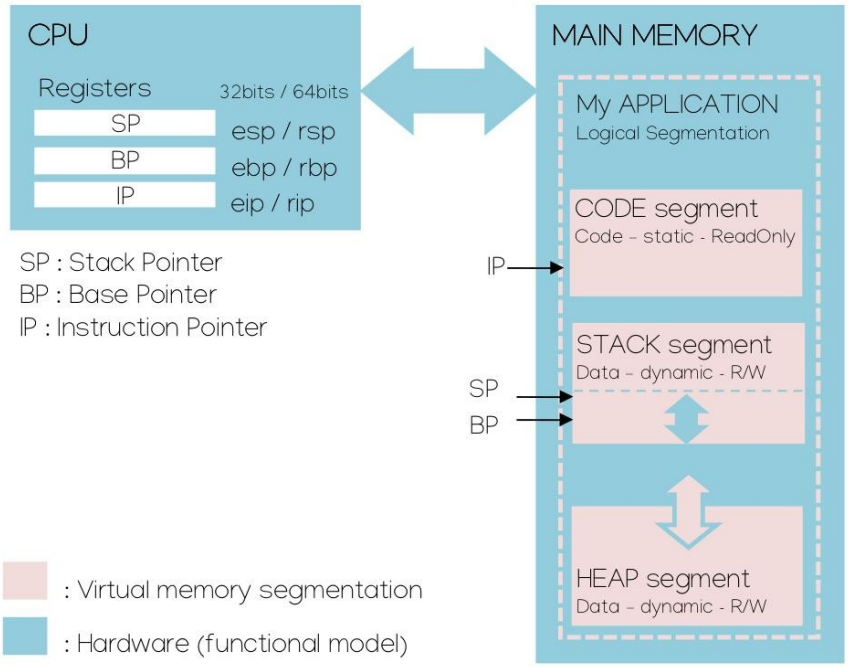
ALLOCATIONS DYNAMIQUES ET SEGMENT DE TAS



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

SOMMAIRE

6. ALLOCATIONS DYNAMIQUES ET SEGMENT DE TAS



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

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

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

6.1. Gestion du tas

- Se placer dans le répertoire *disco/heap*. Compiler le fichier *heap.c* jusqu'à l'édition des liens incluse et exécuter le programme. Récupérer le PID (Process Identifier) du programme dans une nouvelle console et observer le mapping mémoire du programme. Constater que les 3 pointeurs affichés via le *printf* dans le programme *heap.c* pointent vers 3 segments mémoire distincts (CODE, STACK et HEAP). *Pour faciliter l'analyse, nous compilerons le programme en statique (-static) en incluant donc dans le firmware de sortie le code binaire de la fonction malloc et des fonctions systèmes dépendantes (évite les dépendances avec la bibliothèque partagée standard du C ou libc).*

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

./heap

[CTRL] + [c]
```

```
ps -a

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

- Qu'elle est la taille du segment de pile ?

- Qu'elle est la taille du segment de tas ?

- Qu'elle est la taille du segment de code (seul segment statique dont les propriétés permettent l'exécution, propriétés --x-) ? *Constater que chaque segment en mémoire principale possède une taille multiple de 4Ko, la taille d'une page gérée par l'unité de pagination MMU.*

- Compiler le fichier *heap.c* en s'arrêtant à l'assemblage. Analyser le programme assembleur généré. *A ce stade de l'enseignement, vous devez pouvoir être capable d'analyser des script assembleur de ce type. Si c'est le cas, ce que j'espère, bravo, du chemin a été fait !*

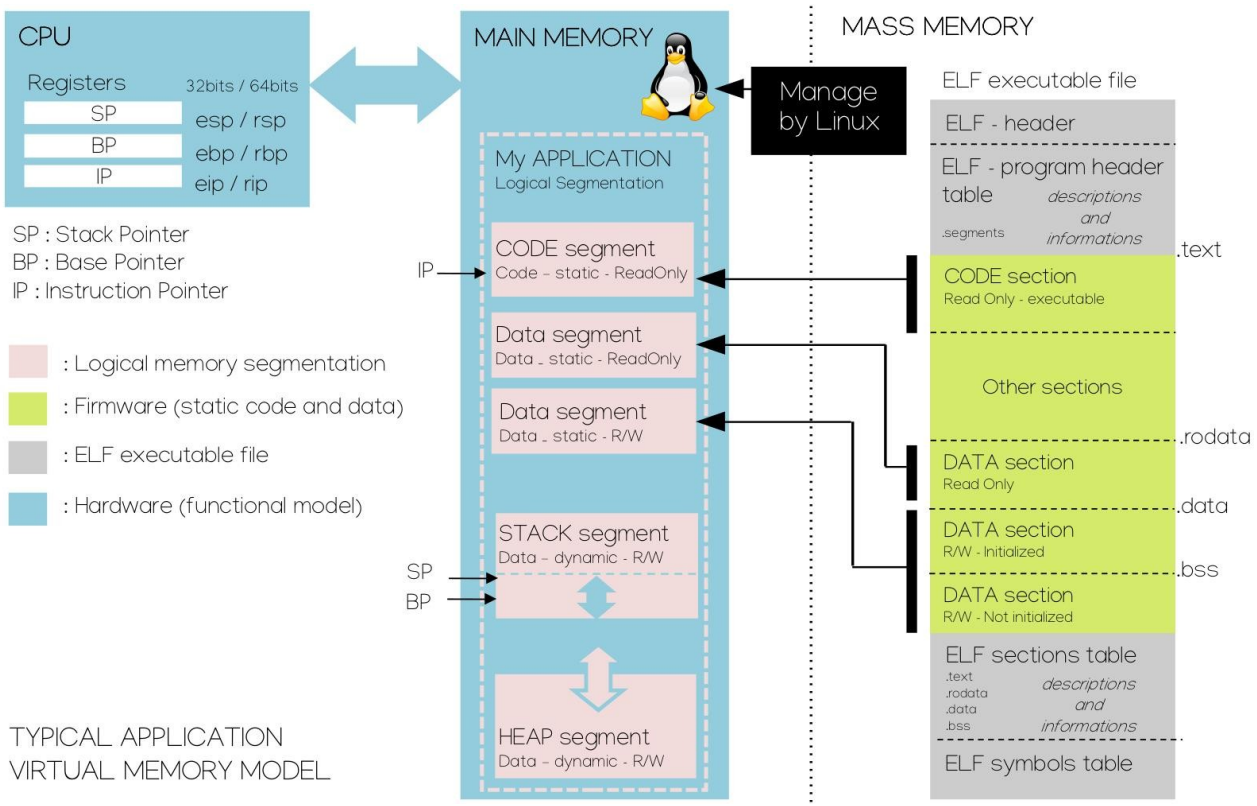
```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall heap.c
```

6.2. Limites du tas

- Se placer dans le répertoire *disco/except*. Compiler le fichier *heap_overflow.c* jusqu'à l'édition des liens incluse et l'exécuter. Analyser le programme et observer les limites du tas. Comparer aux informations proposées par le système.

```
gcc heap_overflow.c -o heap_overflow

./heap_overflow

free

cat /proc/meminfo
```

6.3. Synthèse globale sur les stratégies d'allocations



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

Compilation et édition des liens

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

Allocation des segments et chargement en mémoire par le noyau

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

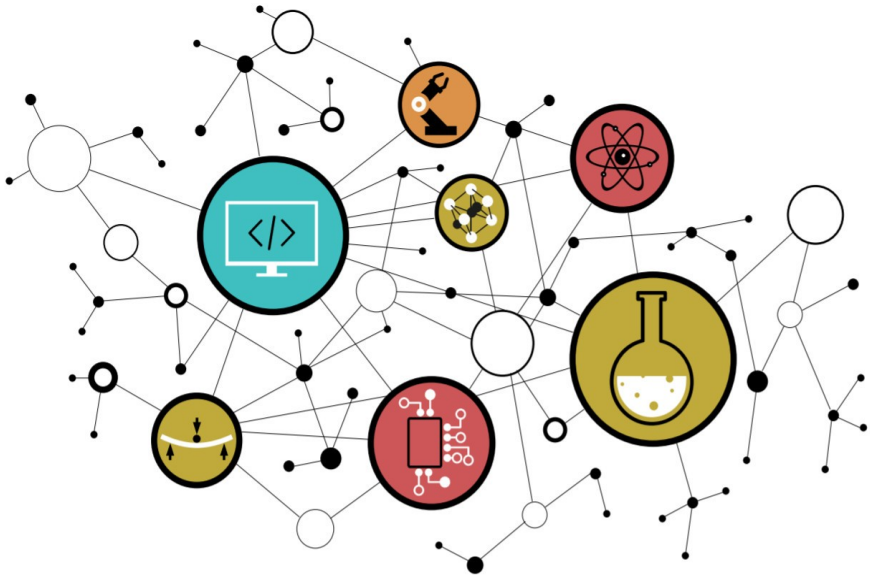
Allocations mémoire et segments associés

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


notes

TRAVAUX PRATIQUES

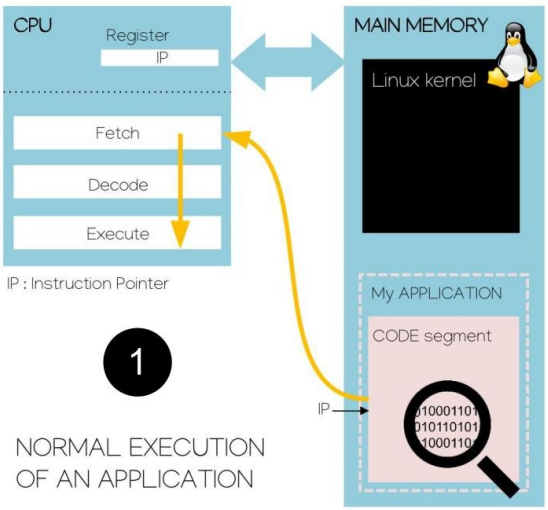
EXCEPTIONS MATÉRIELLES ET SIGNAUX UNIX

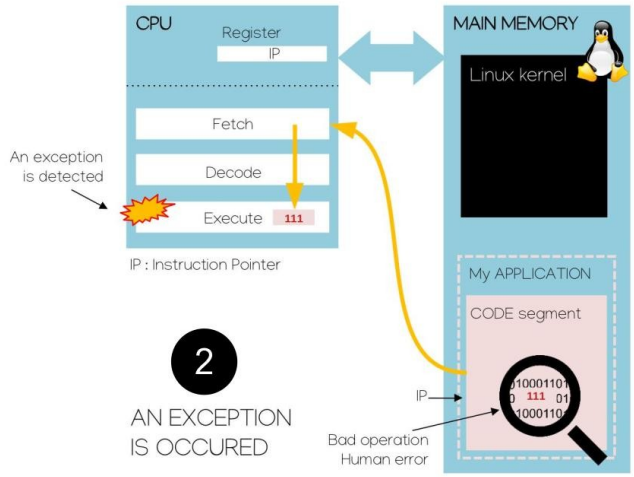


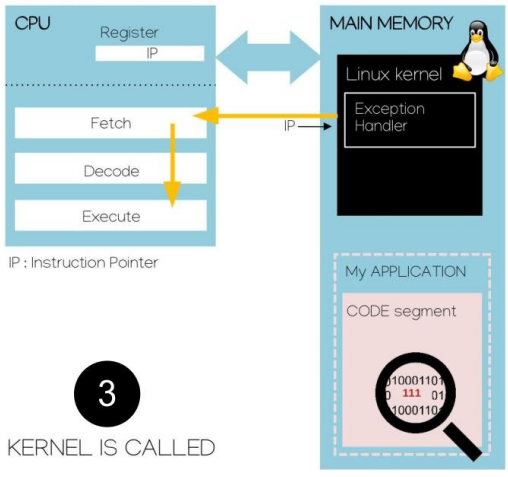
Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

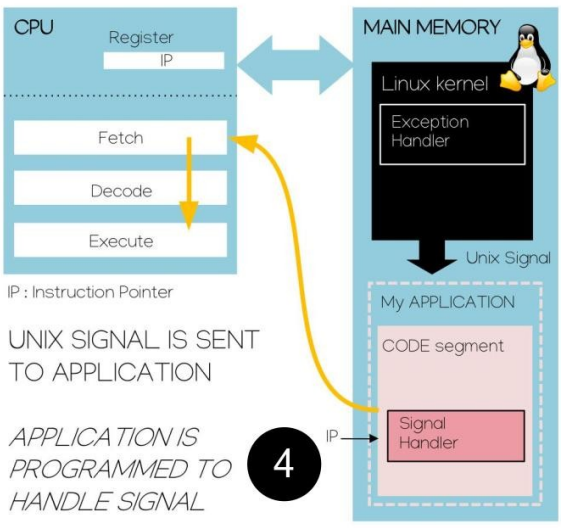
SOMMAIRE

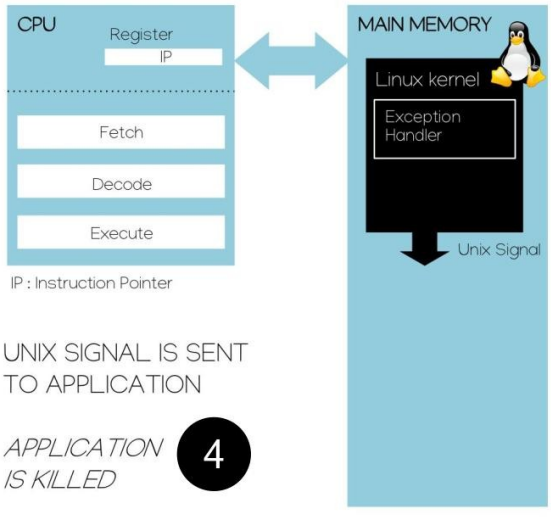
7. EXCEPTIONS MATÉRIELLES ET SIGNAUX UNIX











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

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

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

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

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

   ou

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

7.1. Lecture seule

- Se placer dans le répertoire *disco/except*. Compiler le fichier *except_readonly.c* jusqu'à l'édition des liens incluse et exécuter le programme. Interpréter et expliquer l'exception matérielle ainsi que le défaut logiciel à la racine de cette exception. Proposer un schéma commenté.

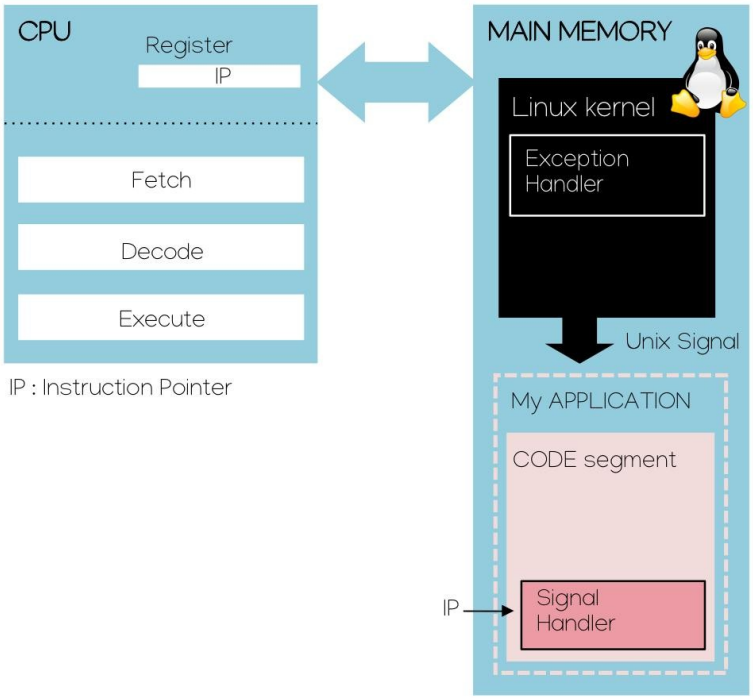
```
gcc except_readonly.c -o except_readonly

./except_readonly

Segmentation fault (core dumped)
```

7.2. Pointeur nul

- Compiler le fichier *except_null_pointer.c* jusqu'à l'édition des liens incluse et exécuter le programme. Interpréter et expliquer l'exception matérielle ainsi que le défaut logiciel à la racine de cette exception. Proposer un schéma commenté.

```
gcc except_null_pointer.c -o except_null_pointer

./except_null_pointer

Segmentation fault (core dumped)
```

7.3. Signal Unix



- Compiler le fichier *signal_handler.c* jusqu'à l'édition des liens incluse et exécuter le programme. Interpréter et expliquer l'exception matérielle ainsi que le défaut logiciel à la racine de cette exception. Analyser le script assembleur du programme et préciser l'instruction ainsi que le registre à la source de l'exception. *La séquence qui suit propose de remplacer le contenu du registre avec une valeur acceptable par le système pour la bonne exécution du processus.*

```
gcc signal_handler.c -o signal_handler

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

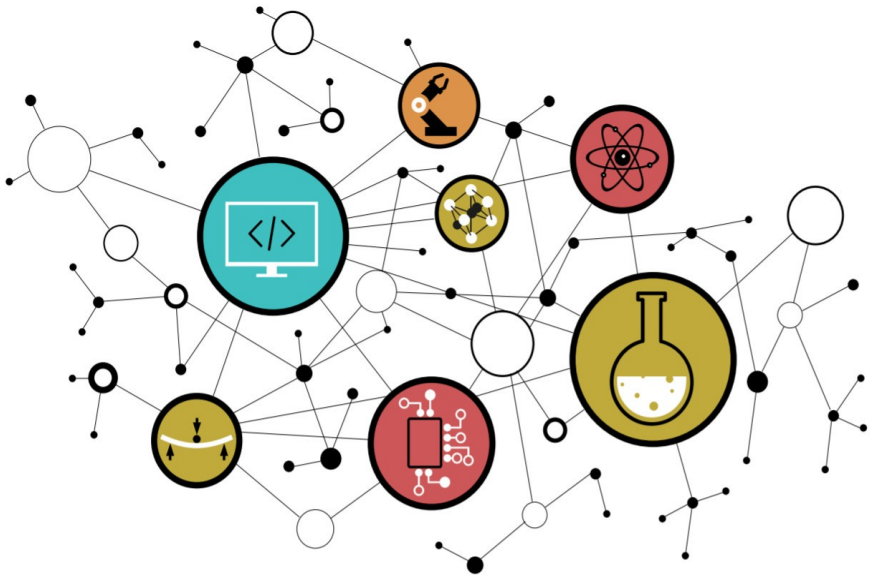
- Dé-commenter la section de code assurant la configuration et la gestion des signaux système reçus par l'application. Analyser le fonctionnement du programme.

- Dé-commenter la ligne de code dans la fonction *signal_handler* proposant une solution alternative afin d'acquitter l'instruction à la source de l'exception (dans le *main*) et analyser le résultat. *Voilà, vous venez de générer volontairement une exception (défaut de segment), de capter le signal Unix envoyé par le système, d'acquitter l'erreur par remplacement du contenu d'un registre (RAX) et de redonner une chance à l'application de s'exécuter sans défaut.*


notes

TRAVAUX PRATIQUES

HACKING



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

SOMMAIRE

8. HACKING

Le terme *hacker* a été maintenant depuis bien des années détourné par les médias pour faire référence à la notion de pirate informatique. Cependant, à la racine, derrière ce mot se cache des utilisateurs développeurs fouineurs poussés par la passion, le jeu, le plaisir, l'échange et le partage. Des passionnés devenus experts dans la compréhension et la maîtrise des systèmes numériques d'information. Depuis la naissance d'internet, dans les années 90, le concept de cybercriminalité a émergé. 2 familles de hackers apparaissent, les *black hat* , exerçant des activités qualifiées de criminelles, et les *white hat* d'une vision éthique, cherchant à aider à la sécurité des systèmes d'informations.

Un code éthique du hacker a d'ailleurs été formalisé au MIT et publié dans un livre de Steven Levy en 1984 (*Hackers : Heros of the Computer Revolution*). Il s'agit d'un ensemble de concepts et de visions provenant de la relation symbiotique entre hackers et machines. Celui-ci comprend 6 règles :

- L'accès aux ordinateurs - ainsi que tout ce qui peut permettre de comprendre comment le monde fonctionne - doit être universel et sans limitations. Il ne faut pas hésiter à se retrousser les manches pour surmonter les difficultés.
- Toute information doit être libre.
- Se méfier de l'autorité - encourager la décentralisation.
- Les hackers doivent être jugés selon leurs *hacks*, et non selon de faux critères comme les diplômes, l'âge, l'origine ethnique ou le rang social.
- On peut créer l'art et le beau à l'aide d'un ordinateur.
- Les ordinateurs peuvent améliorer notre vie.

Durant cet exercice, nous allons humblement chercher à entrouvrir le domaine du *hacking* et de l'expertise système. Nous allons explorer des solutions simples d'injection de code sur la pile par *shellcode*. Nous vous invitons bien entendu à explorer le sujet, en regardant par exemple des solutions plus avancées comme la ROP chain (Return-Oriented Programming) et bien d'autres. Vous trouverez d'ailleurs de nombreuses exploitations de vulnérabilités sur internet, notamment sur des applications ayant des failles connues publiées sur le site CVE (Common vulnerabilities and Exposures, https://cve.mitre.org/index.html).

8.1. Exécution d'un shellcode sur la pile

Se placer dans le répertoire */disco/hack/* afin d'appliquer la totalité des commandes qui suivent. Nous allons chercher à invoquer une console système d'interface avec le noyau (*shell*) en injectant puis en exécutant du code depuis la pile d'un applicatif. Le programme standard permettant d'invoquer un *shell* (*/bin/sh*) utilise la fonction *execve* et réalise un appel système au noyau Linux (instruction *syscall* en 64bits et *int 0x80* en 32bits). Observons quelques informations concernant cette fonction et son implémentation en *user space* (distribution GNU\Linux Debian-like comme Ubuntu) et *kernel space* (Linux) :

- Manuel Ubuntu pour la fonction *execve* (user space) :

https://manpages.ubuntu.com/manpages/precise/fr/man2/execve.2.html

- Source code officiel Linux, fichier */fs/execve.c* (kernel space) :

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/exec.c?h=v5.4-rc7

- Vulnérabilités connues sur le fichier du noyau Linux */fs/execve.c* :

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

- Ouvrir puis compiler le fichier *shellcode.c*. Exécuter le programme.

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

./bin/shellcode
```

- Que constatons-nous ?

- Observer l'en-tête de programme du fichier ELF exécutable de sortie, et préciser les droits (rwx ou read write executable) sur le futur segment de pile qui sera associé à notre programme à l'exécution ?

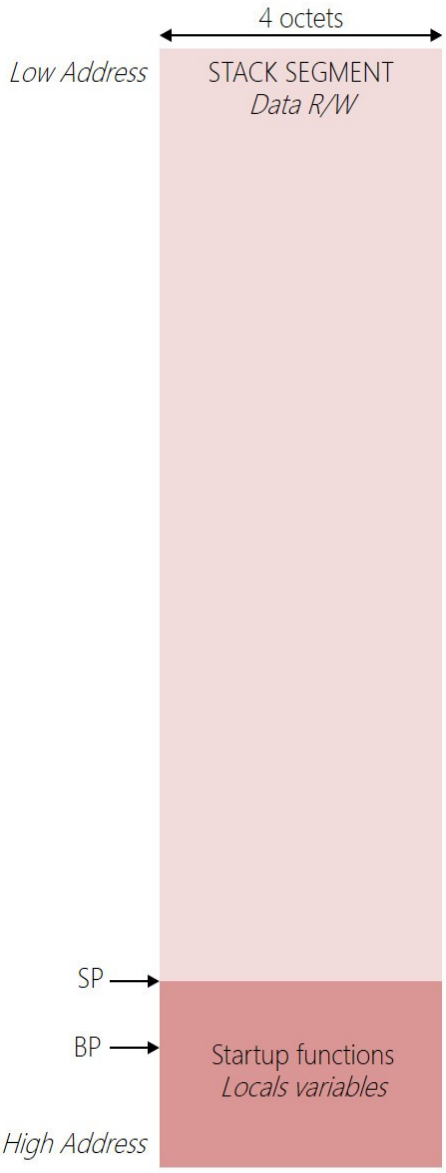
```
objdump -fp bin/shellcode
```

- 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). Rendre la pile exécutable, vérifier les droits sur le segment de pile et exécuter le programme. Que constatons-nous ? *Entrer exit pour quitter le shell*

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

./bin/shellcode

objdump -fp bin/shellcode
```

- Compiler et analyser le programme assembleur. Analyser ensuite l'exécution du programme en ouvrant une session de *debug* avec *gdb*. Décrire 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
...etc
(gdb) ni
$
$ exit
(gdb) q
```



- Après analyse du script assembleur, représenter ci-contre le contenu de la pile avant exécution de l'instruction *call* *%rdx implémentant l'appel de fonction *(*(void(*)()) shellcode)()* . 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
```

8.2. Extraction et édition d'un shellcode

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

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

  exit(EXIT_FAILURE);
}
```

Nous allons maintenant nous intéresser à une méthodologie afin d'identifier et d'éditer un *shellcode*. Pour cela, nous allons partir de la solution générée par défaut par *gcc* à partir d'un appel standard d'un programme depuis un applicatif (cf. ci-dessus, appel de */bin/sh*). La compilation et l'édition des liens se feront en statique (option -*static*) de façon à pouvoir observer l'implémentation binaire et assembleur (après désassemblage) de l'appel de la fonction système *execve*.

- Ouvrir puis compiler le fichier *shell.c*. Exécuter le programme.

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

./bin/shell
```

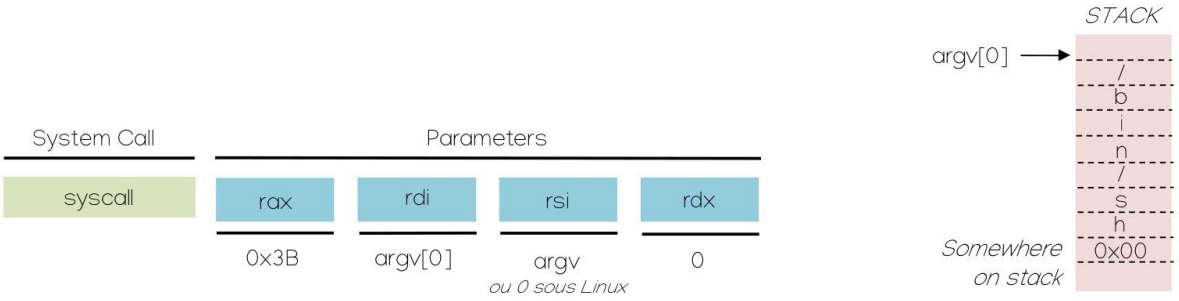
- Que constatons-nous ?

- Analyser le programme assembleur ([CTRL] + [f] puis rechercher *execve*) et comprendre l'implémentation réalisée par *gcc*, notamment les passages de paramètres par registres. Nous allons nous efforcer d'en reproduire une version allégée en assembleur puis en binaire.

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

gdb ./bin/shell
(gdb) la a
(gdb) b main
(gdb) r
(gdb) b execve
(gdb) c
(gdb) ni
...etc
(gdb) ni
$
$ exit
(gdb) q
```

- Que réalise l'instruction *syscall* ?

- Préciser ci-dessous les 4 registres utilisés par le compilateur pour passer des arguments à l'appel système *syscall* ?



Après analyse, nous pouvons en déduire les registres utilisés et les valeurs des arguments à passer avant l'appel système. Pour information, seulement sous Linux, le troisième argument passé peut être nul. Attention, cette implémentation ne respecte pas le standard Unix. Nous allons faire ceci afin d'alléger l'implémentation de notre *shellcode* (instruction XOR pour une mise à zéro rapide de registre et une empreinte minimale du code). Nous allons chercher à réaliser l'implémentation minimale proposée graphiquement ci-dessous.

- Observer le contenu du fichier *shell_asm.s* afin d'implémenter la solution illustrée ci-dessus. Valider l'assemblage et l'exécution du programme

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

ld obj/shell_asm.o -o bin/shell_asm

./bin/shell_asm
```

- Identifier et extraire le shellcode en observant l'implémentation binaire du programme. Analyser à nouveau le *shellcode* binaire du fichier *shellcode.c*. Il est possible d'en trouver une grande quantité déjà édités sur internet (http://shell-storm.org/shellcode/). Écrire le *shellcode* identifié ci-dessous (24 octets)

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

8.3. Édition d'un exploit

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 le 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. Le tableau ci-dessous présente une écriture en pseudo-code des instructions *call* et *ret*.

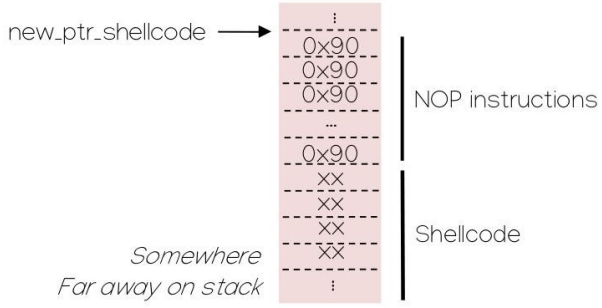
Instruction	CALL *function_address*	RET	
Pseudo-code	RSP ← RSP – 8 *(RSP) ← RIP RIP ← *function_address*	*tmp* ← *(RSP) RSP ← RSP + 8 RIP ← *tmp*	*<--- shellcode address here !* *<--- goto shellcode !*

Nous allons développer ces deux actions sur un programme simple, soit changer l'adresse de retour d'une fonction puis déplacer un *shellcode* sur la pile. Ces techniques permettent d'ouvrir à une infinité d'autres *exploit* possibles. Mais encore faut-il trouver des failles, s'assurer que les bonnes conditions soient réunies et enfin réussir à l'exploiter !

- Modifier 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. Valider la bonne exécution du programme. Ne pas hésiter à activer la macro PRINT_DEBUG durant vos tests, mais penser à la désactiver afin de simplifier l'analyse de l'exploit et de valider son bon fonctionnement.

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

./bin/exploit

objdump -S bin/exploit
```

- Modifier le second commentaire *TODO* afin de déplacer le *shellcode* sur la pile. Nous placerons des instructions NOP (No Operation, 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. Valider la bonne exécution du programme.



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

8.4. White Hat

L'exercice de travaux pratiques s'arrête ici. J'offre avec plaisir un café et la possibilité d'insérer son code dans les ressources de TP à tout étudiante ou étudiant réussissant à développer un exploit permettant d'ouvrir une console root sur sa machine personnelle (sans avoir à user des droits root). Voici sinon les livrables attendus :

- Code source de l'exploit avec Makefile propre à la racine
- Conditions et procédure de test et validation. Fichier texte README.md à la racine
- Démonstration sur machine personnelle (contre café ou autre breuvage à mes frais)
- Contre-mesure et solution (montée de version, patch système, etc) pour bloquer cette faille avec procédure de déploiement sur le système cible

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, etc) ou au niveau kernel Linux avec par exemple une gestion aléatoire des mapping mémoire de certains segments applicatifs (pile, tas, bibliothèques partagées et vDSO, etc), plusieurs contre-mesures sont déjà à l'œuvre afin de contraindre le travail des Black Hat dans leur volonté de pénétrer les systèmes.

Prenons l'exemple de la fonction execve. comme précisé dans la documentation officielle, 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'appelé. 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 !

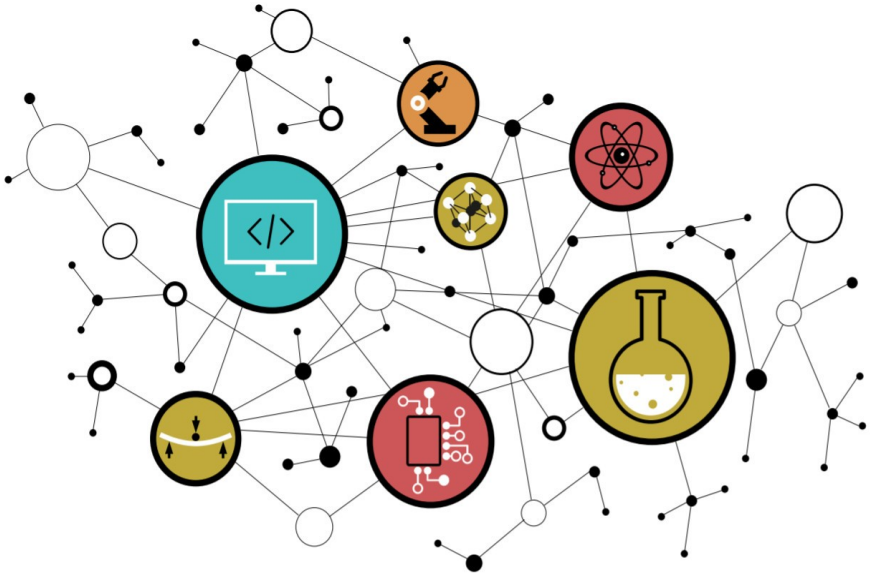
Voilà, et sinon, un peu de musique avec votre café avant d'attaquer le *hack* ...

https://www.youtube.com/watch?v=_AdBdCz3tVs


notes

TRAVAUX PRATIQUES

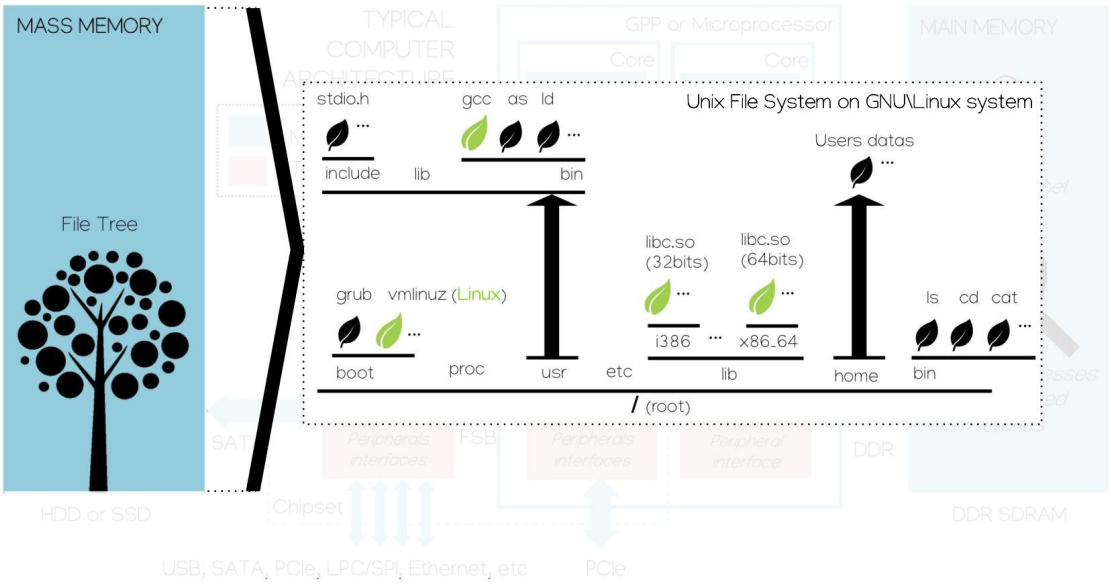
MÉMOIRE DE MASSE ET SYSTÈME DE FICHIERS



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

SOMMAIRE

9. MÉMOIRE DE MASSE ET SYSTÈME DE FICHIERS



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

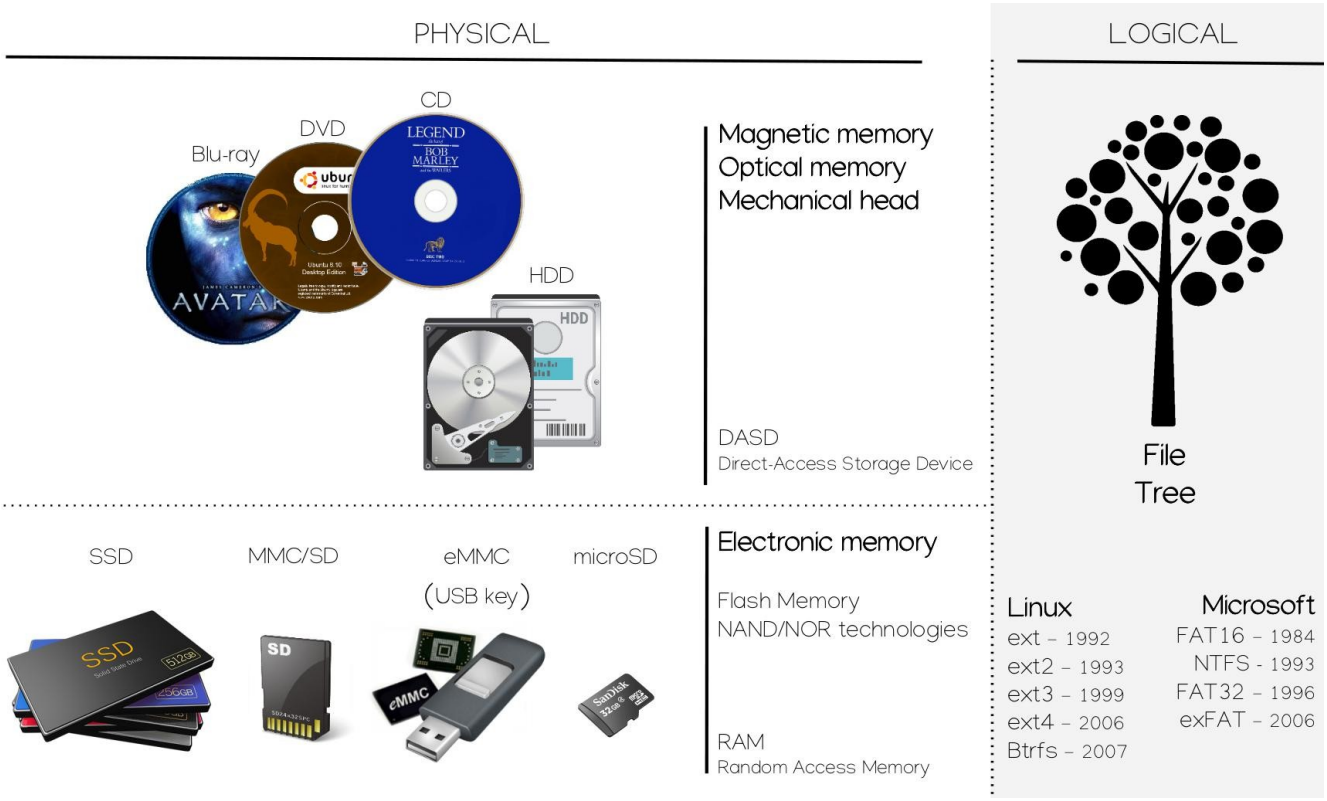
- Mémoire cache : *mémoire adressable par association (associative) de technologie SRAM*. A chaque octet copié en cache (niveaux L1\L2\L3) depuis la mémoire principale est associé un emplacement (par indexage) dans une ligne de cache.
- Mémoire principale : *mémoire adressable par octet de technologie DRAM*. Chaque octet possède une adresse mémoire unique typiquement représentée au format hexadécimal (32bits, 64bits, exemple 39bits physical et 48bits virtual, etc – *cat /proc/cpuinfo*). Sur processeur intégrant une MMU (GPP, AP, etc) 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, etc*. Cette mémoire stock l'information en implémentant les concepts de classification de fichiers dans des répertoires. Un fichier possède donc une adresse nommée chemin (*path*) dans une arborescence dont la base est nommée racine (/ ou *root* sur système Unix-like).

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

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

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

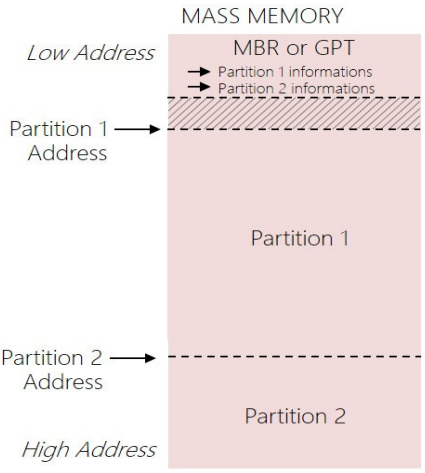


Plusieurs technologies de mémoire de masse sont actuellement en usage (HDD, SSD, MMC SD, MMC microSD, CD, DVD, Blu-ray, etc), tout comme certaines sont maintenant tombées en désuétude (K7 audio, disquette, VHS, VHS-c, carte perforée, tore magnétique, etc). Chaque technologie offre son lot hérité d'avantages,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

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

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

9.1. Table des partitions



Une partition représente une zone logique contigu de la mémoire physique. Pour un support donné et en fonction des besoins, il est bien entendu 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 dominent le marché :

- MBR (Master Boot Record, https://doc.ubuntu-fr.org/mbr). Nous ne pouvons créer que 4 partitions primaires maximum, toutes de tailles inférieures à 2To. Technologie encore très utilisée à notre époque, notamment dans les Systèmes Embarqués et ordinateur personnel

- 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 !***

- Se placer dans le répertoire */disco/mass/* puis connecter la clé USB à votre machine. Identifier son nom (*/dev/sdx*) et ses caractéristiques.

```
Lsblk

lsblk -f

sudo sfdisk -l

export DISK=/dev/<your_device_name>
```

- Identifier le paramètre système *block size* relatif à votre support (valeur dépendant de multiples paramètres), puis effacer l'ancienne table des partitions. Analyser 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 ?

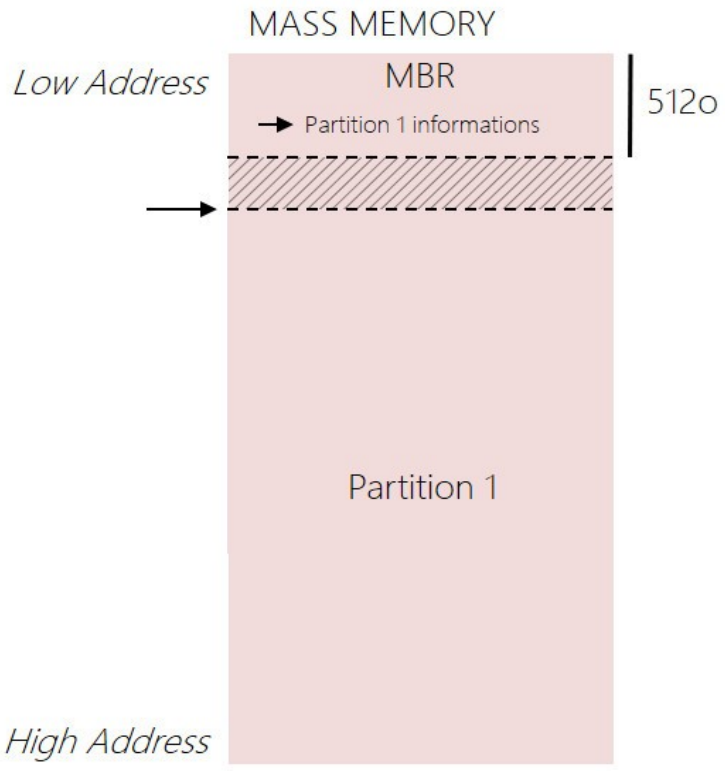
- Nous allons créer un nouvelle table des partitions en utilisant l'utilitaire *sfdisk*. Néanmoins, d'autres utilitaires existent pour créer des tables des partitions (*parted, fdisk, etc*). Créer une nouvelle table des partitions et analyser la sortie.

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

lsblk

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

xxd ./mbr.bin > mbr.txt

file mbr.bin
```

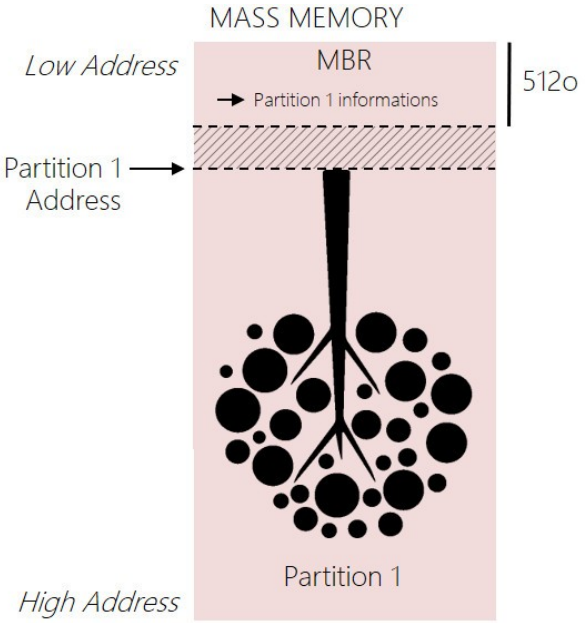
- En s'aidant d'internet, préciser la taille d'un MBR ? Par quelle suite d'octets se termine toujours un MBR ? Vérifier que cette suite binaire est bien présente après création de la table des partitions.

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

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



9.2. 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 peu 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, mûres et offrant du support comme FAT par exemple. Sur ordinateur ou système embarqué (SoC AP ou System on Chip Application Processor) supervisé par Linux et couplé au réseau, préférer par exemple Btrfs, la quintessence des FS sous Linux (ext2, ext3, ext4 étant des technologies transitoires).

- Déployer un système de fichiers Virtual FAT (extension à FAT12, FAT16 et FAT32) sur la partition n°1 et nommer 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
```

- Observer l'implémentation technologique Virtual FAT du système de fichiers à l'adresse de début de la partition. Retrouver votre label ?

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

xxd ./fs_fat32.bin > fs_fat32.txt
```

9.3. 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 (clé USB, MMC SD, etc). 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 !

- Créer un point de montage et respecter le nom du label présent dans la partition n°1 précédemment créée (néanmoins, ce nom pourrait être différent). Vérifier et valider avant de réaliser l'opération, le chemin relatif à votre nom d'utilisateur dans */media* . Valider les opérations réalisées.

```
export MEDIA=/media/<user_name>

sudo mkdir -p ${MEDIA}/root/

sudo mount ${DISK}1 ${MEDIA}/root/

lsblk -f
```

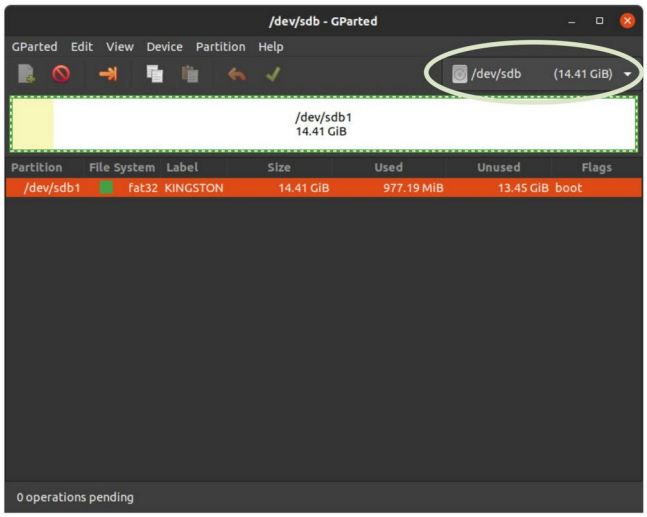
- Réaliser une écriture sur le point de montage, synchroniser point de montage et support physique puis retirer manuellement du système le point de montage. Vous pouvez alors retirer physiquement la clé USB de la machine et valider son bon fonctionnement sur un autre ordinateur ! *Ne pas oublier l'étape de synchronisation, sinon l'écriture sur le média cible ne sera pas active*.

```
echo "Hello World Bro's !" > hello.txt

sudo cp hello.txt ${MEDIA}/root/

sync

sudo umount ${MEDIA}/root
```

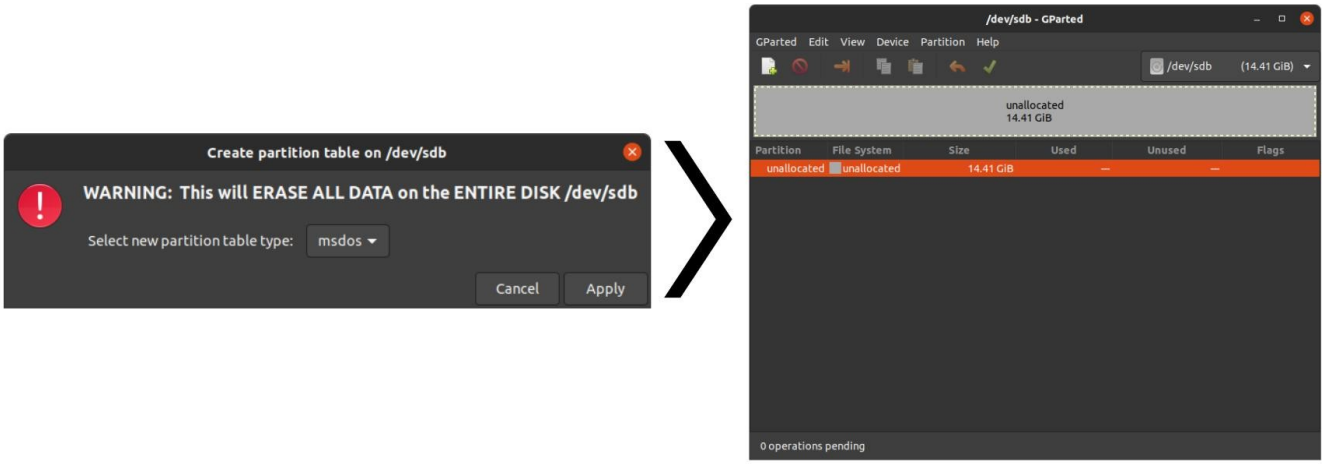
9.4. Outil graphique GParted

Nous allons finaliser l'exercice avec l'utilisation d'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 graphique les plus standard sur système GNU\Linux.

- Sélectionner le périphérique matériel à partitionner et retirer le point de montage existant
    - GParted > Refresh Devices
    - GParted > Devices > /dev/sd? (your device name)
    - Clic droit sur la partition du device (Partition > /dev/sdb1 - ci-dessous) > Unmount

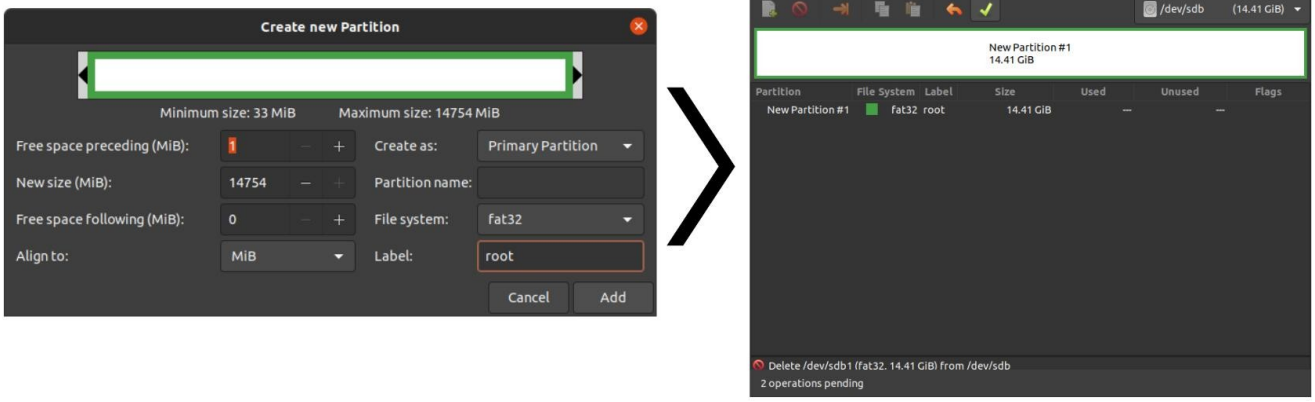


- Créer une table de partitions MSDOS (technologie MBR)
    - Device > Create Partition Table…
    - msdos > Apply



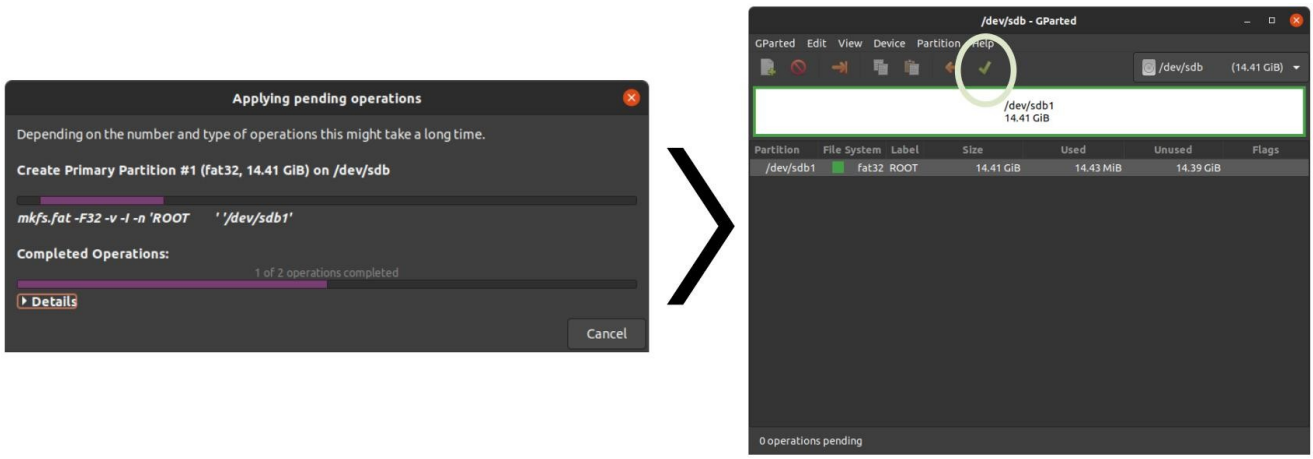
- Créer une nouvelle partition

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



- Appliquer les configurations précédentes

    - Cliquer sur l'icône ''Apply all operations'' entouré ci-dessous
    - Et c'est terminé !



9.5. Kali sur clé USB bootable

Pour celles et ceux étant 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 Linux (https://www.kali.org/) 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, etc).

- Télécharger l'ISO d'un Kali Linux Light 64bits :

  https://www.kali.org/downloads/

- A 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, redémarrer votre ordinateur en interrompant la phase de boot à l'amorçage (appuyer sur F12 sur ordinateur en salles A203/A201 ou F10,F2, etc dépend du fabricant de carte mère), spécifier que vous souhaitez booter (démarrer) sur un support USB et un Kali Linux va se démarrer en quelques seconde

Et voilà, les TP sont terminés !


```
neku$ Pour conclure sur quelques mots bien personnels ...
neku$ Je dirais longue vie à Linux, au projet GNU, aux systèmes Unix ...
neku$ A l'open source, au travail collaboratif décentralisé ...
neku$ A l'agilité dans le travail en équipe et dans les entreprises ...
neku$ A la liberté de s'informer et de partager l'information ...
neku$ A vous et à vos rêves …
neku$ Que nous ne cessions jamais d'œuvrer pour la liberté de tous ...
neku$ My name is nobody
neku$
```

A

- ABI : Application Binary Interface
- ADC : Analog to Digital Converter
- ALU : Arithmetic and Logical Unit
- AMD : Advanced Micro Devices
- ANSI : American National Standards Institute
- AP : Application Processor
- API : Application Programming Interface
- APU : Accelerrated Processor Unit
- ARM : Société anglaise fabless concevant des CPU RISC 32bits
- ASCII : American Standar Code for Information Interchange
- ASIC : Application Specific Integrated Circuit

B

- BP : Base Pointer
- BSL : Board Support Library
- BSP : Board Support Package

C

- CCS : Code Composer Studio
- CEM : Compatibilité ElectroMagnétique
- CISC : Complex Instruction Set Computer
- CMS : Composant Monté en Surface
- CPU : Central Processing Unit
- CSL : Chip Support Library

D

- DAC : Digital to Analog Converter
- DDR : Double Data Rate
- DDR SDRAM: Double Data Rate Synchronous Dynamic Random Access Memory
- DMA : Dual Inline Package (boîtier de composant)
- DMA : Direct Memory Access
- DSP : Digital Signal Processor
- DSP : Digital Signal Processing

E

- EDMA : Enhanced Direct Memory Access
- EUSART : Enhanced Universal Synchronous Asynchronous Receiver Transmitter
- EMIF : External Memory Interface
- EPIC : Explicitly Parallel Instruction Computing

F

- FPU : Floating Point Unit
- FLOPS : Floating-Point Operations Per Second
- FMA: Fused Multiply-Add

G

- GCC : Gnu Collection Compiler
- GLCD : Graphical Liquid Crytal Display
- GNU : GNU's Not UNIX
- GPIO : General Purpose Input Output
- GPGPU : General Purpose GPU
- GPP : General Purpose Processor
- GPU : Graphical Processing Unit

I

- IA-64 : Intel Architecture 64bits
- I2C : Inter Integrated Circuit
- IC : Integrated Circuit
- ICC : Intel C++ Compiler
- ICC : Interface Chaise Clavier (main problem root)
- IDE : Integrated Development Environment
- IDMA : Internal Direct memory Access
- IHM : Interface Homme Machine
- IRQ : Interrupt ReQuest
- ISR : Interrupt Software Routine
- ISR : Interrupt Service Routine

L

- L1D : Level 1 Data Memory
- L1I : Level 1 Instruction Memory (idem L1P)
- L1P : Level 1 Program Memory (idem L1I)
- Lx : Level x Memory
- LCD : Liquid Crytal Display
- LRU : Least Recently Used

M

- MAC: Multiply Accumulate
- MCU : Micro Controller Unit
- MIMD : Multiple Instructions Multiple Datas
- MIPS : Mega Instructions Per Second
- MISD : Multiple Instructions Single Data
- MMU : Memory Managment Unit
- MPLABX : MicrochiP LABoratory 10, IDE Microchip
- MPU : Micro Processor Unit ou GPP
- MPU : Memory Protect Unit
- MPPA : Massively Parallel Processor Array

O

- OS : Operating System

P

- PC : Program Counter
- PC : Personal Computer
- PCB : Printed Circuit Board
- PIC18 : Famille MCU 8bits Microchip
- PIC : Programmable Interrupt Controller
- PLD : Programmable Logic Device
- POSIX : Portable Operating System Interface (norme IEEE 1003)
- PPC : Power PC

R

- RAM : Random Access Memory
- RISC : Reduced Instruction Set Computer
- RS232 : Norme standardisant un protocole série asynchrone
- RTOS : Real Time Operating System
- RTS : Real Time System

S

- SDK : Software Development Kit
- SIMD : Single Instruction Multiple Date
- SIP : System In Package
- SOB : System On Board
- SOC : System On Chip
- SOP : Sums of products
- SP : Stack Pointer
- SP : Serial Port
- SPI : Serial Peripheral Interface
- SRAM : Static Random Access Memory
- SSE : Streaming SIMD Extensions
- STM32 : STMicroelectronics 32bits MCU

T

- TI : Texas Instruments
- TNS : Traitement Numérique du Signal
- TSC : Time Stamp Counter
- TTM : Time To Market

U

- UART : Universal Asynchronous Receiver Transmitter
- USB : Universal Serial Bus

V

- VHDL : VHSIC Hardware Description langage
- VHSIC : Very High Speed Integrated Circuit
- VLIW : Very Long Intruction Word

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES