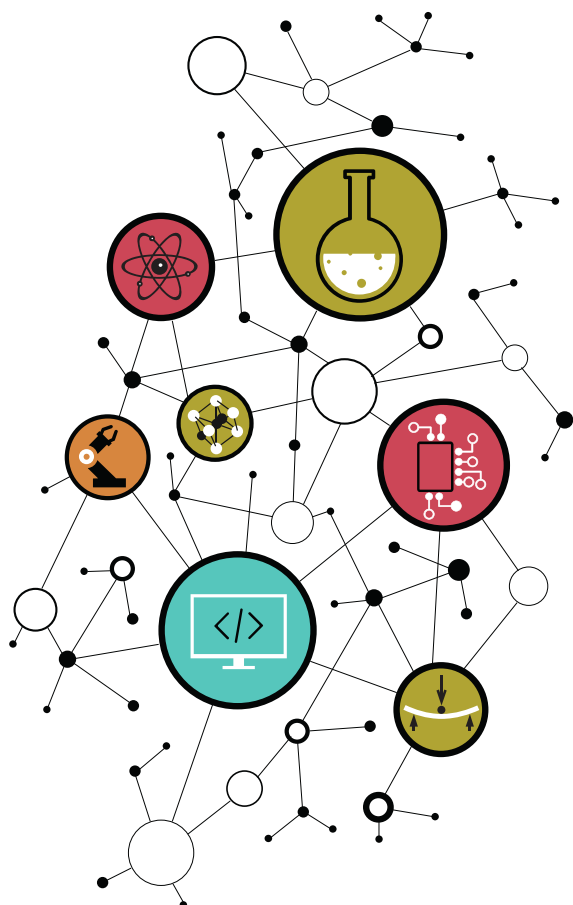


PARTIE 3

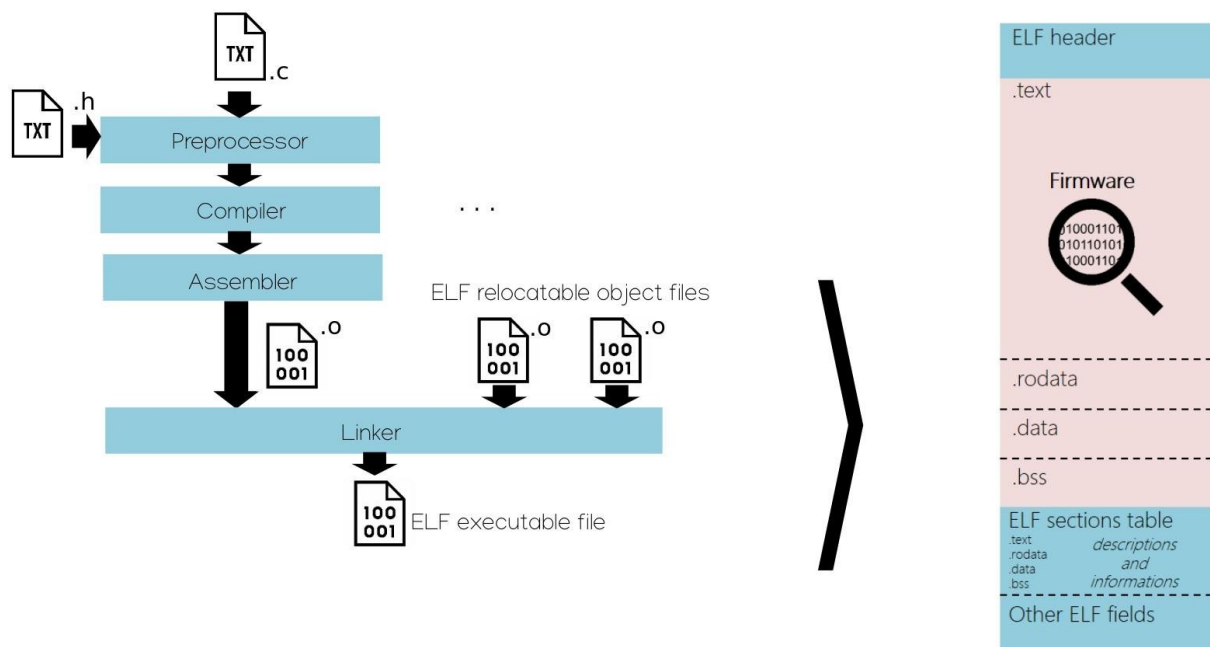
ALLOCATION STATIQUE ET FICHIER ELF



I. Compilation et allocation statique

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

Les **allocations statiques** représentent toutes les allocations de ressources mémoire **réalisées à la compilation** et donc présentes dans le fichier binaire ELF de sortie. Les variables statiques admettent donc une existence sur un média de stockage de masse (HDD, SSD, ...) avant même l'exécution d'un programme en mémoire principale. Les références symboliques, ou adresses logiques, de chaque fonction et variable statique sont donc inchangées (statiques) durant la totalité de la vie d'un programme binaire (tant qu'il n'y a ni nouvelle compilation et ni édition des liens du projet logiciel source). En d'autres termes, ce chapitre traite de l'allocation des **fonctions et variables statiques**. En revanche ce chapitre ne traite ni des variables locales (dont l'allocation automatique ou dynamique est gérée sur le segment de pile) ni des données allouées avec un `malloc/calloc/new/...` (dont l'allocation dynamique est gérée sur le segment de tas), puisque ces deux mécanismes auront chacun leur propre chapitre.



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

Le ***firmware*** est encapsulé dans un cartouche au format ELF (en-tête, table des sections, en-tête du programme, ...) proposant au noyau du système (Linux) une description et des informations sur le micrologiciel afin d'aider à sa manipulation (préparation des segments mémoire, association de propriétés et privilèges, ...) au moment du lancement de l'application.

II. Variables globales

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

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

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

    return 0;
}
```

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

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

```
cat global_variable.s
```

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

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

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

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

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

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

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

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

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

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

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

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

Que contiennent les sections affichées ici ?

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

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

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

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

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

```
objdump -t global_variable
```

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

```
objdump -S global_variable
```

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

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

III. Variables locales statiques

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

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

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

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

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

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

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

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

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

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

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

IV. Chaînes de caractères

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

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

Observez le contenu binaire du fichier objet.

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

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

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

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

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

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

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

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

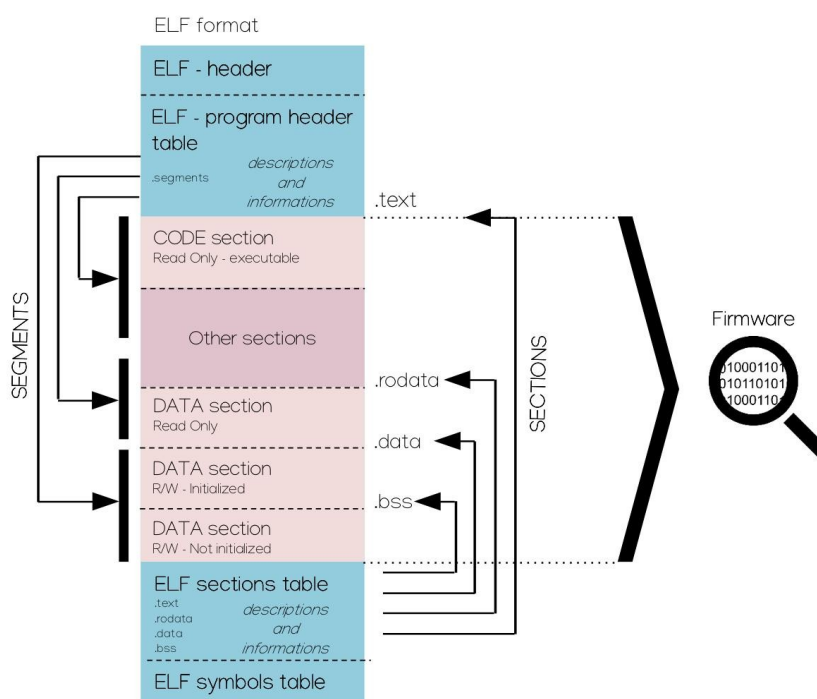
Note : il peut être intéressant de repasser sur cet exercice une fois que le chapitre sur l'allocation dynamique a été traité, puisqu'ici cohabitent allocation statique (chaîne de caractères `GNU's design ...`) et allocation dynamique (`gnu_pointer`, pointeur vers cette chaîne de caractères).

V. Synthèse

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

Sauf si un développeur crée explicitement de nouvelles sections en spécifiant des attributs spécifiques durant une déclaration d'une variable¹⁰, une application pourra comporter au plus quatre sections applicatives par défaut afin de gérer l'ensemble des besoins standards en allocations statiques de ressources (les noms suivants sont hérités d'Unix) :

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



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

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

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

