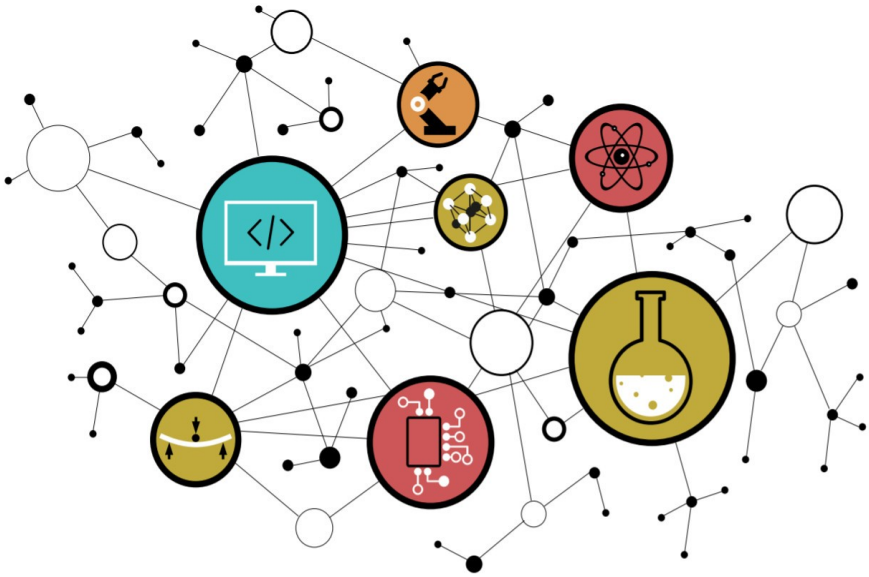
TRAVAUX PRATIQUES

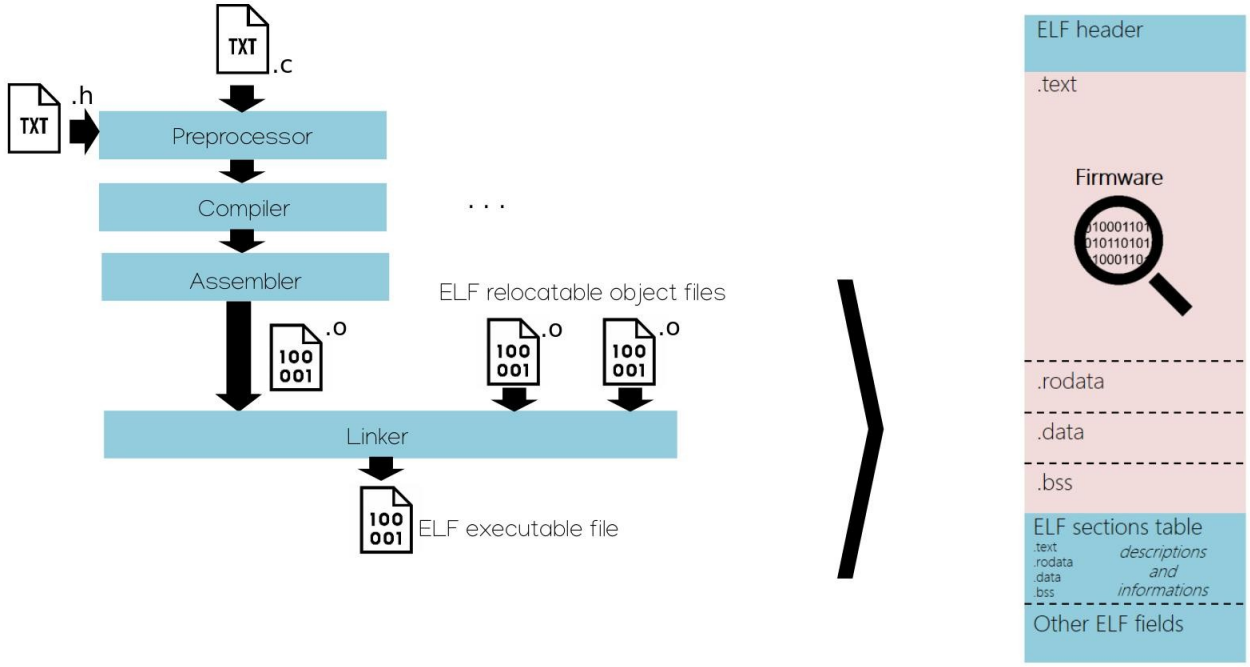
ALLOCATIONS STATIQUES ET FICHIER ELF



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.*

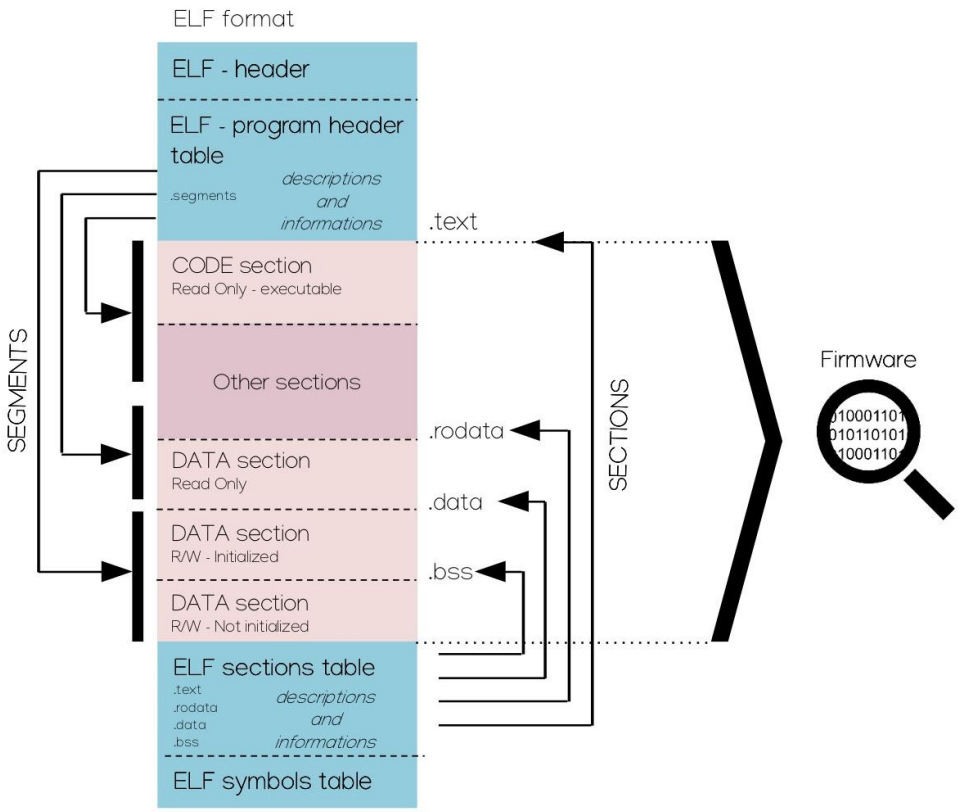
```
gcc -c -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none -Wall global_variable.c -o global_variable.o

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