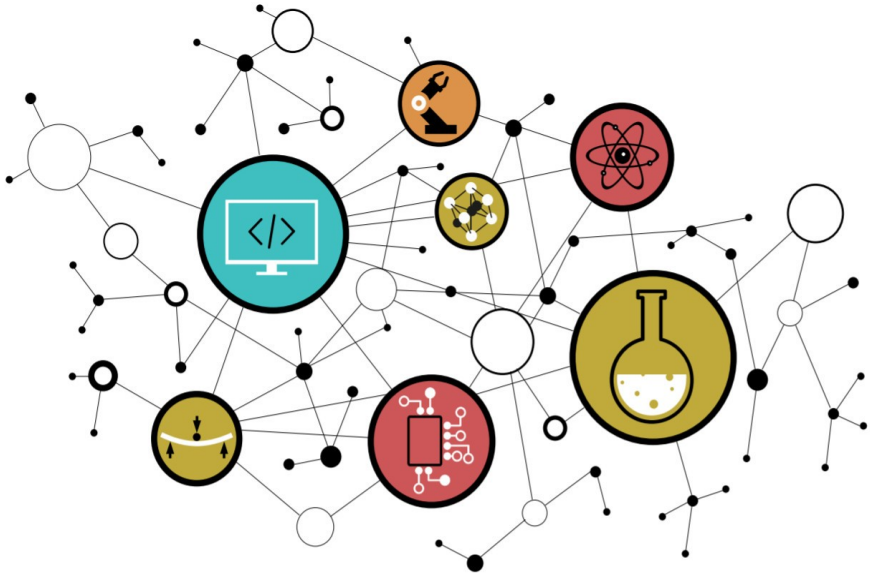
TRAVAUX PRATIQUES

COMPILATION ET ÉDITIONS DES LIENS



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

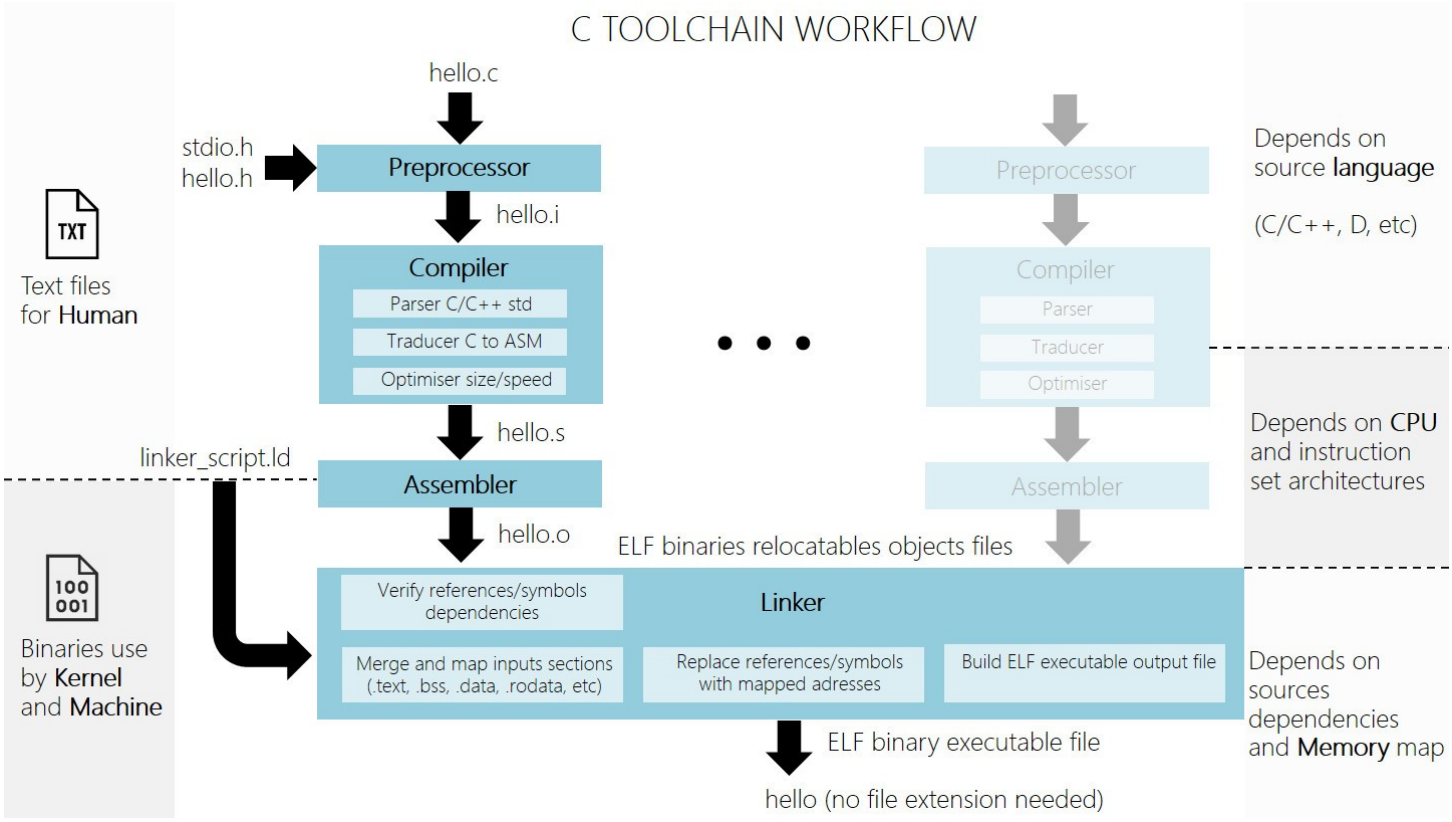
SOMMAIRE

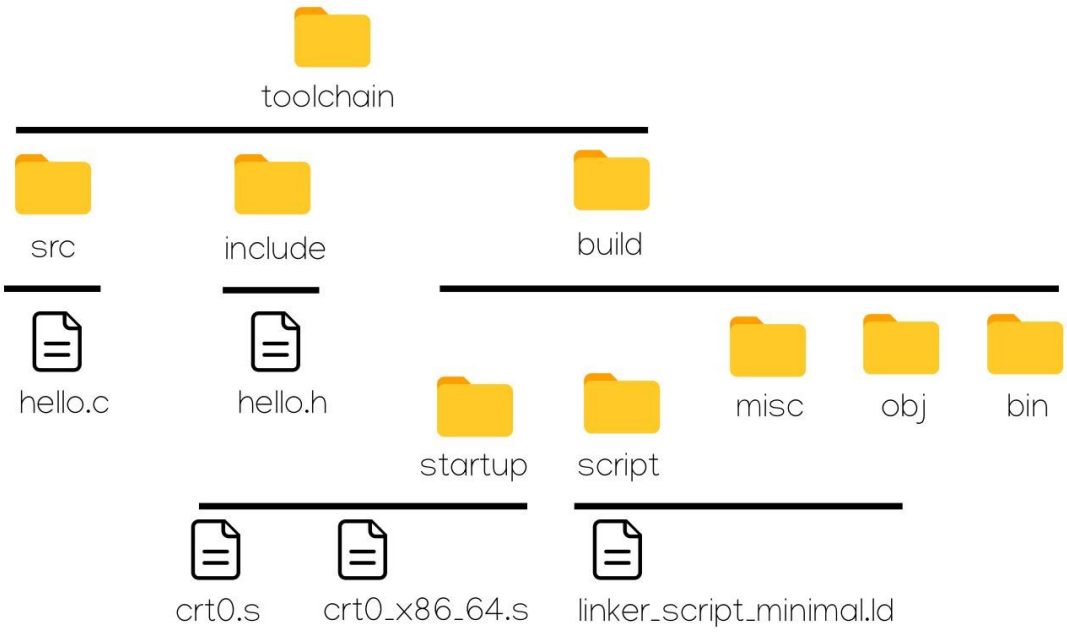
2. COMPILATION ET ÉDITION DES LIENS

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 teste et analyser le fichier produit en sortie !

```
gcc -E -m32 -I./inc 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

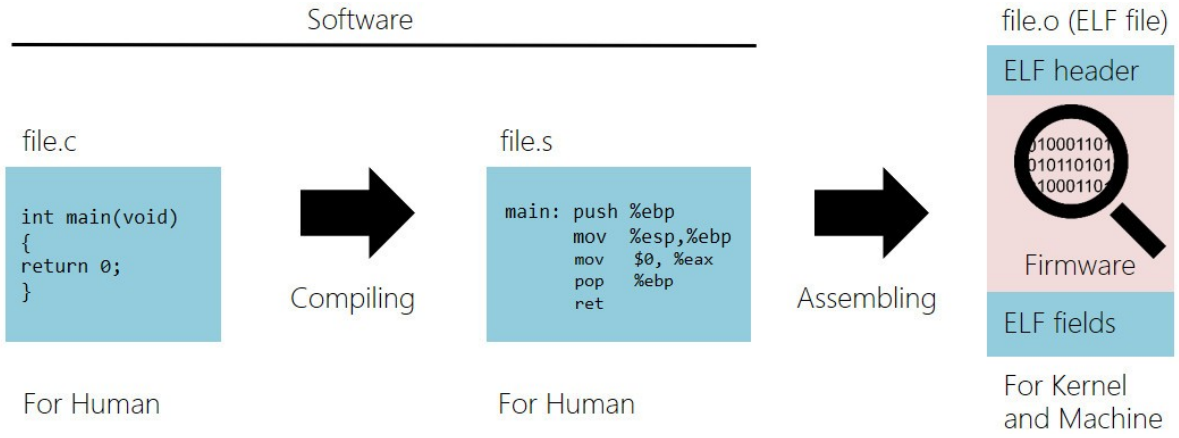
- Qu'est-ce qu'un label en assembleur ? Que représentera plus tard à l'exécution le label *main* (simple chaîne de caractères) ?

- 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
```

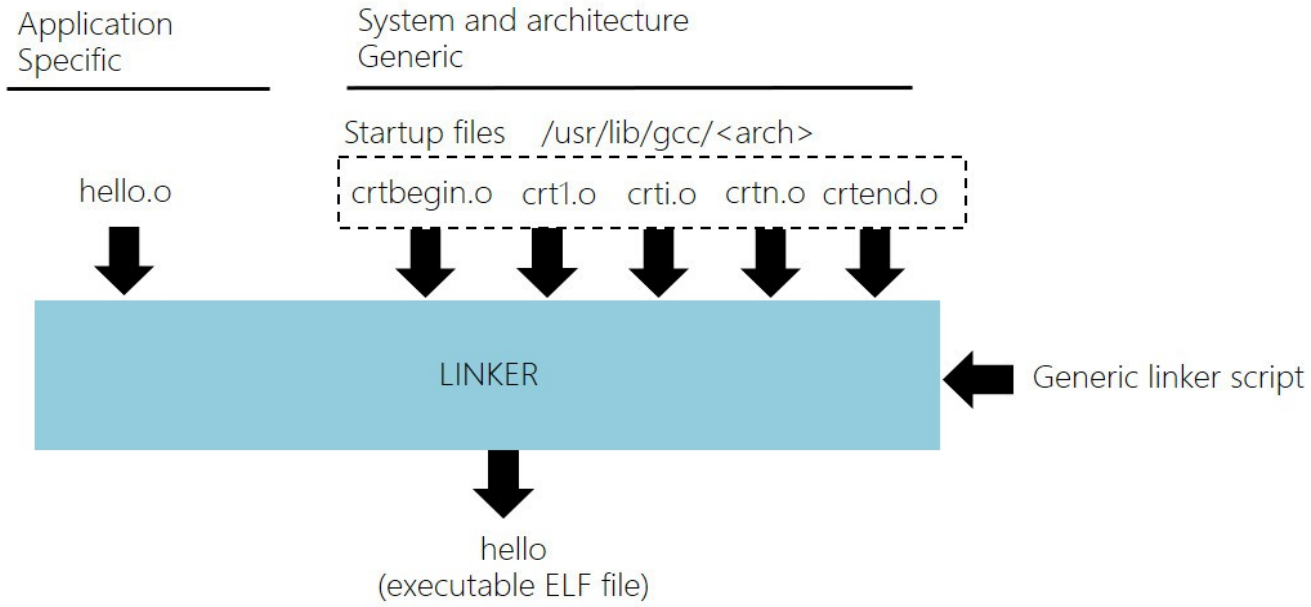
2.5. Startup file

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.

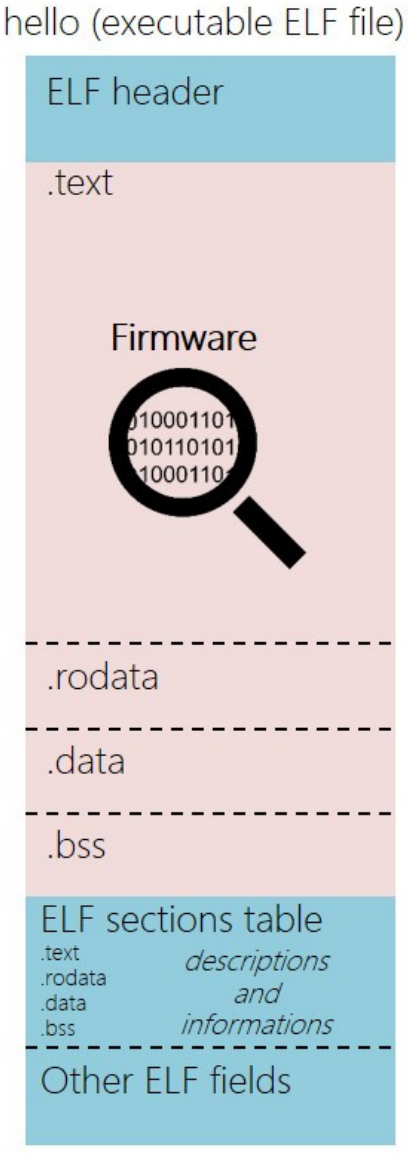
```
#include <stdio.h>

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
```

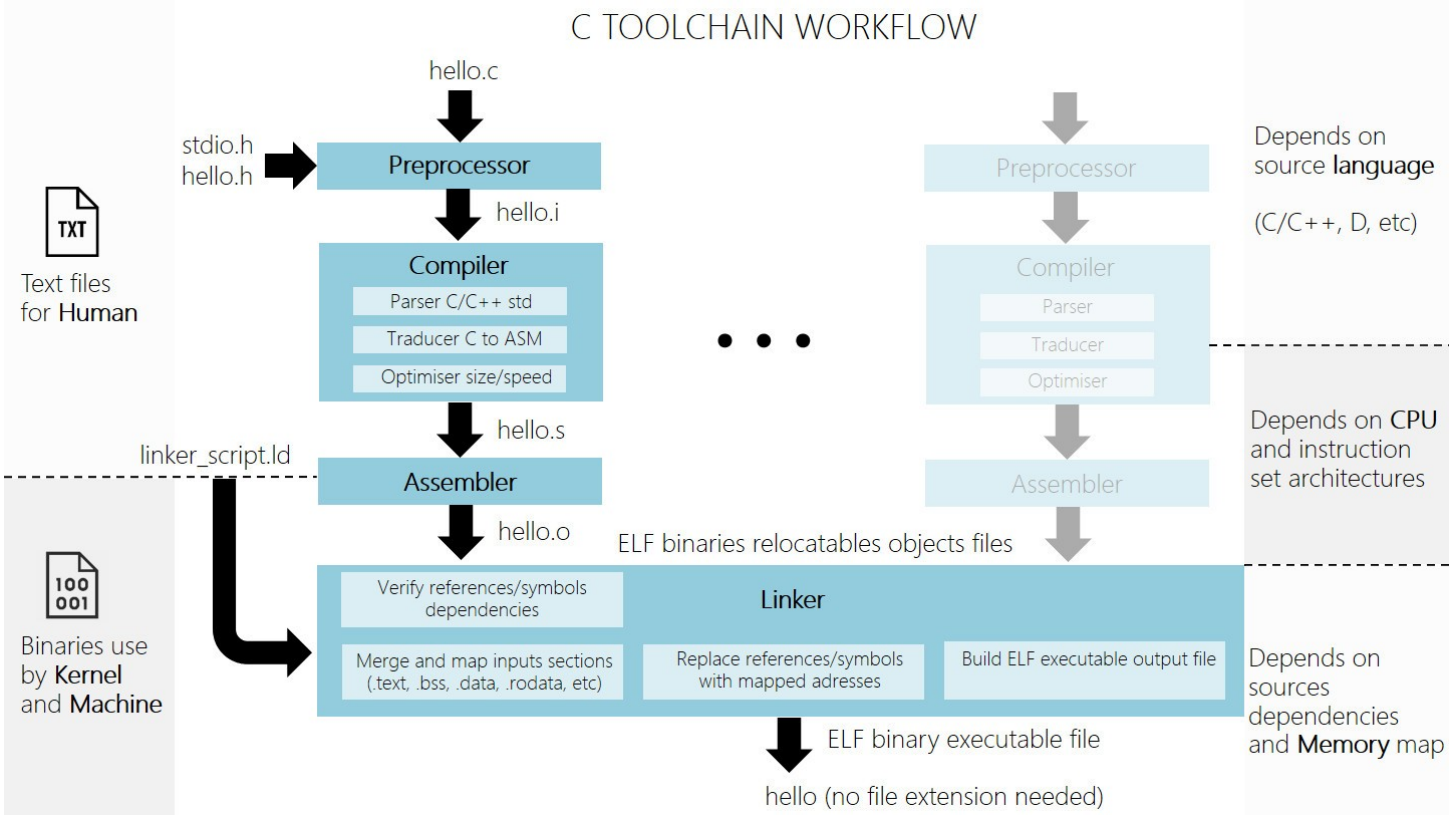
Vous devriez observer 4 segments mémoire distincts (les plages d'adresses virtuelles sont propres à votre application) : /<your_path>/disco/toolchain/build/bin/hello (code binaire), [stack], [vVAR] et [vDSO]. Les segments vVAR et vDSO (virtual Dynamic Shared Object) sont des ponts potentiels entre le 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