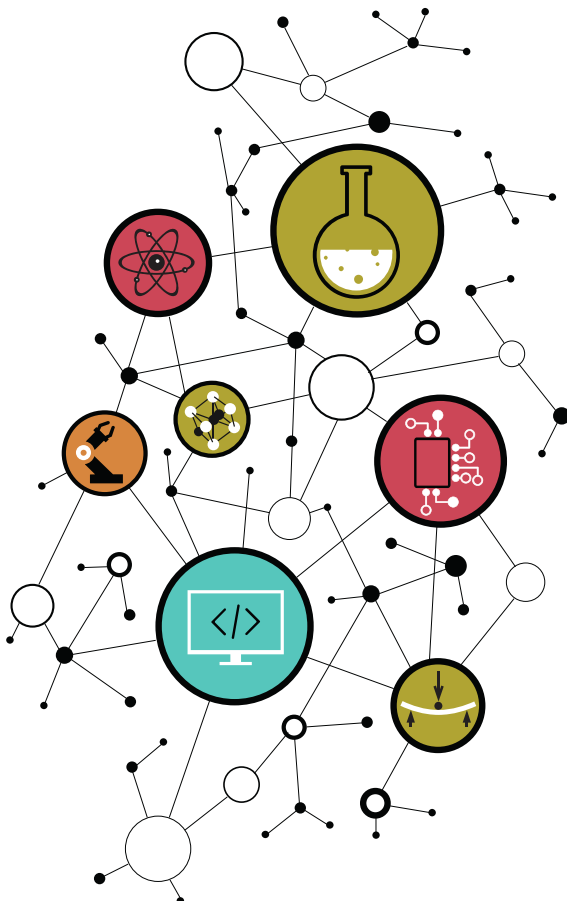


PARTIE 2

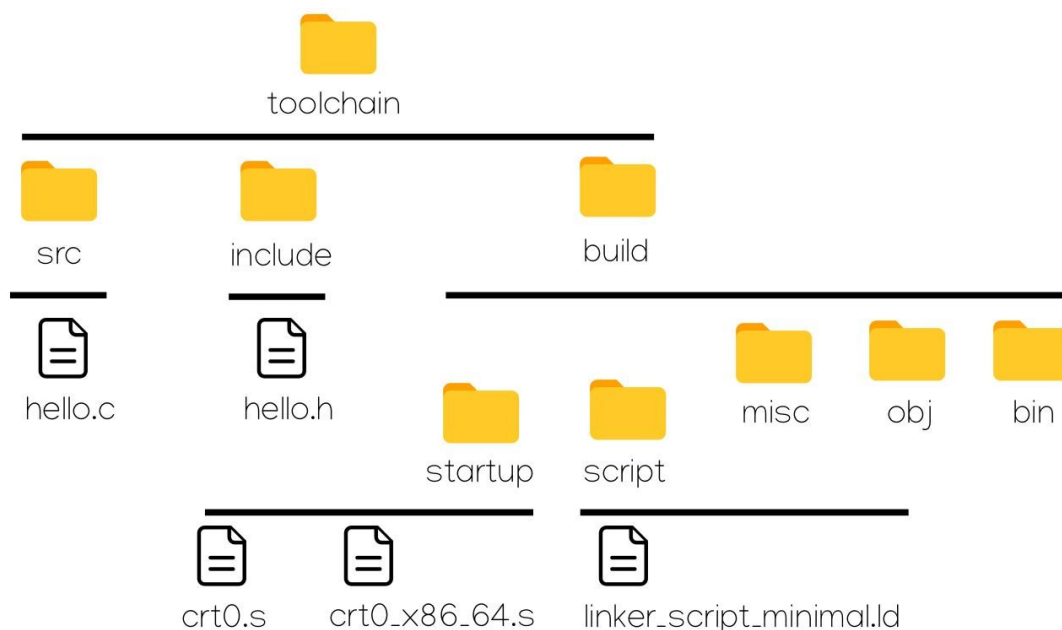
CHAÎNE DE COMPILE



I. Arborescence

Dans ce chapitre, nous allons nous intéresser aux différentes étapes du **processus de compilation** et **d'édition de liens** d'un projet logiciel, dans notre cas développé en langage C. Afin d'appréhender ce *workflow*, nous analyserons la compilation d'un programme élémentaire constitué d'un fichier source unique `disco/toolchain/src/hello.c` incluant un fichier d'en-tête applicatif élémentaire `disco/toolchain/include/hello.h`.

Pour travailler, ouvrez un terminal et placez-vous dans le répertoire `/disco/toolchain/` afin d'appliquer la totalité des commandes qui suivent. Le fichier `README.md` contient la séquence d'exécution complète de l'exercice.



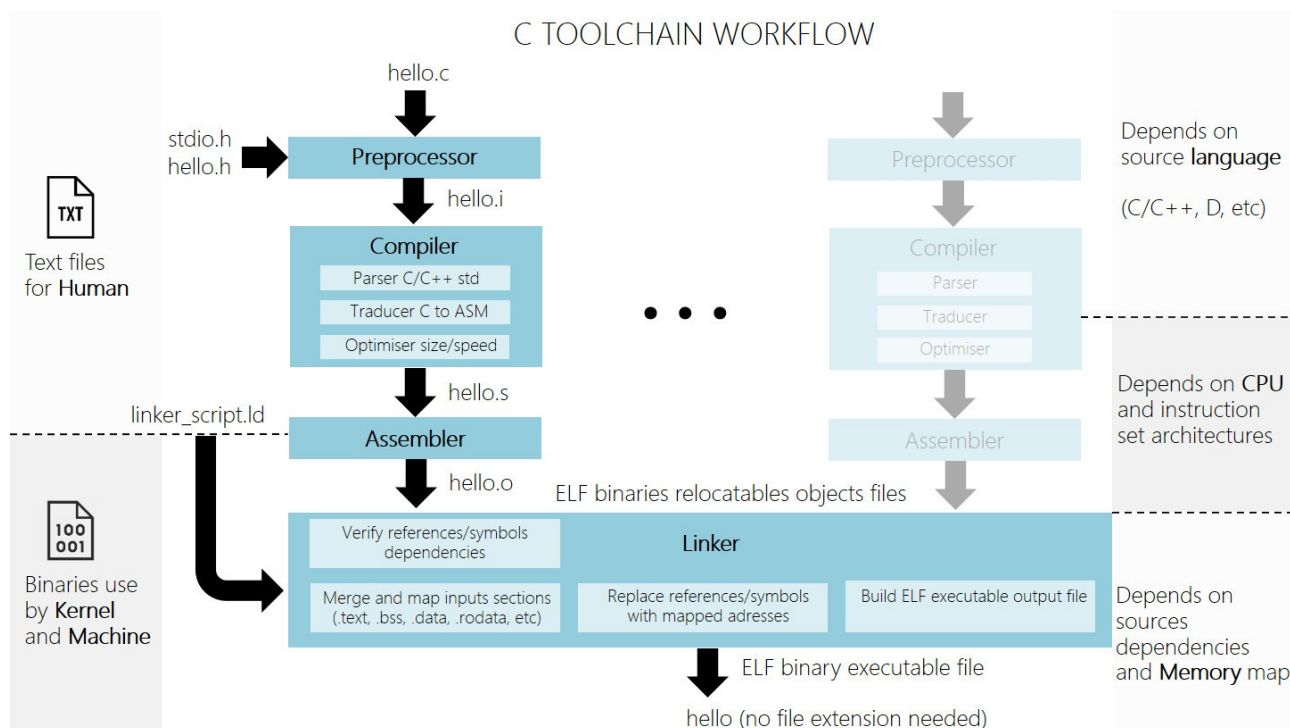
II. GCC Toolchain

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

1. le **prétraitement** (préparation du code avant compilation – analyse lexicale, supprime, copie, colle, remplace) ;
2. la **compilation** proprement dite (analyse syntaxique et sémantique, traduction vers l'assembleur de l'architecture CPU cible et optimisation optionnelle) ;
3. et enfin l'**assemblage** (conversion d'un programme assembleur vers son équivalent binaire pour la machine cible, sans résolution des adresses mémoire en gardant des références symboliques et génération d'un fichier binaire ré-adressables au format ELF).

L'étape suivant la compilation est l'**édition des liens** (analyse et validation des dépendances entre fichiers et références symboliques, placement mémoire et résolution des adresses des symboles statiques, génération dans un format donné ELF/COFF/HEX/etc du fichier binaire exécutable de sortie).

Le schéma ci-dessous, représente ces différentes étapes exécutées séquentiellement par la *toolchain*. La compilation est un processus indépendant pour chaque fichier source à compiler. L'éditeur des liens (*linker*) travaillera alors avec l'ensemble des fichiers objets binaires générés à la compilation. Le résultat de ce processus statique est la génération d'un fichier binaire exécutable, dans notre cas au format binaire ELF (*Executable and Linkable Format*) sur système Unix.



III. Compilation

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

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

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

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

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

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

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

III.1. Preprocessing

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

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

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

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

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

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

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

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

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

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

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

III.2. Analyse et génération de code natif

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

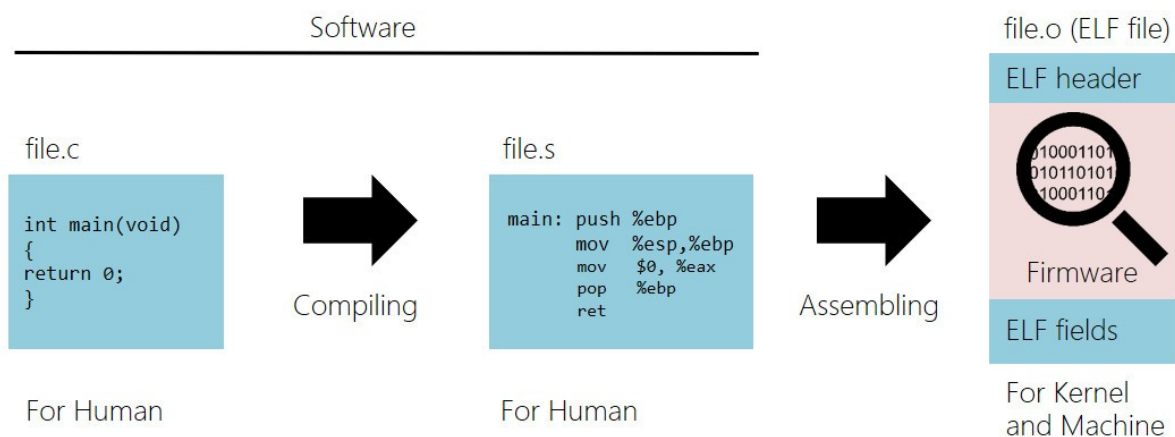
Qu'est-ce qu'un registre ? Combien de registres CPU utilisés observez-vous dans ce programme assembleur ?

III.3. Assemblage

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

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

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



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

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

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

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

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

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

IV. Édition des liens

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

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

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

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

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

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

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

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

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

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

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

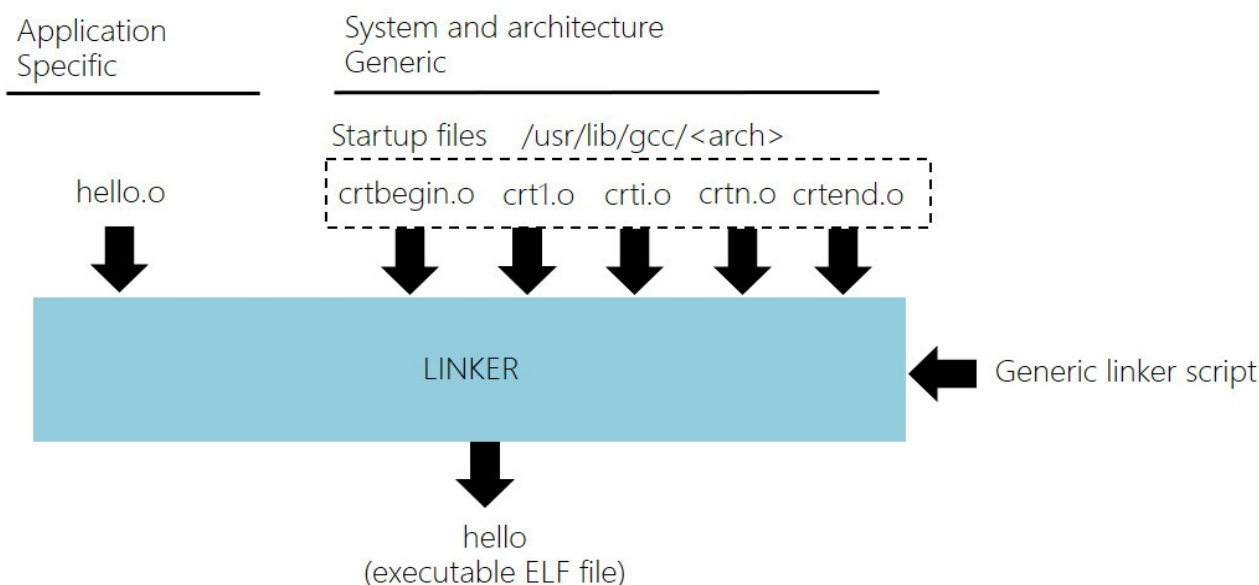
IV.1. Startup file

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

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

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

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



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

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

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

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

```
.global _start

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

Assemblez ce nouveau fichier d'amorçage

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

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

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

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

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

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

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

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

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

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

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

IV.2. Linker script

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

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

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

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

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

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

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

Que contient la section `.text` ?

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

Que contient la section `.comment` ?

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

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

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

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

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

```
#include <stdio.h>

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

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

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

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

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

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

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

hello (executable ELF file)

ELF header

.text

Firmware

010001101
010110101
1000110

.rodata

.data

.bss

ELF sections table

.text
.rodata
.data
.bss

descriptions
and
informations

Other ELF fields

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

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

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

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

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

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

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

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

Le fichier `hello` est-il toujours exécutable ?

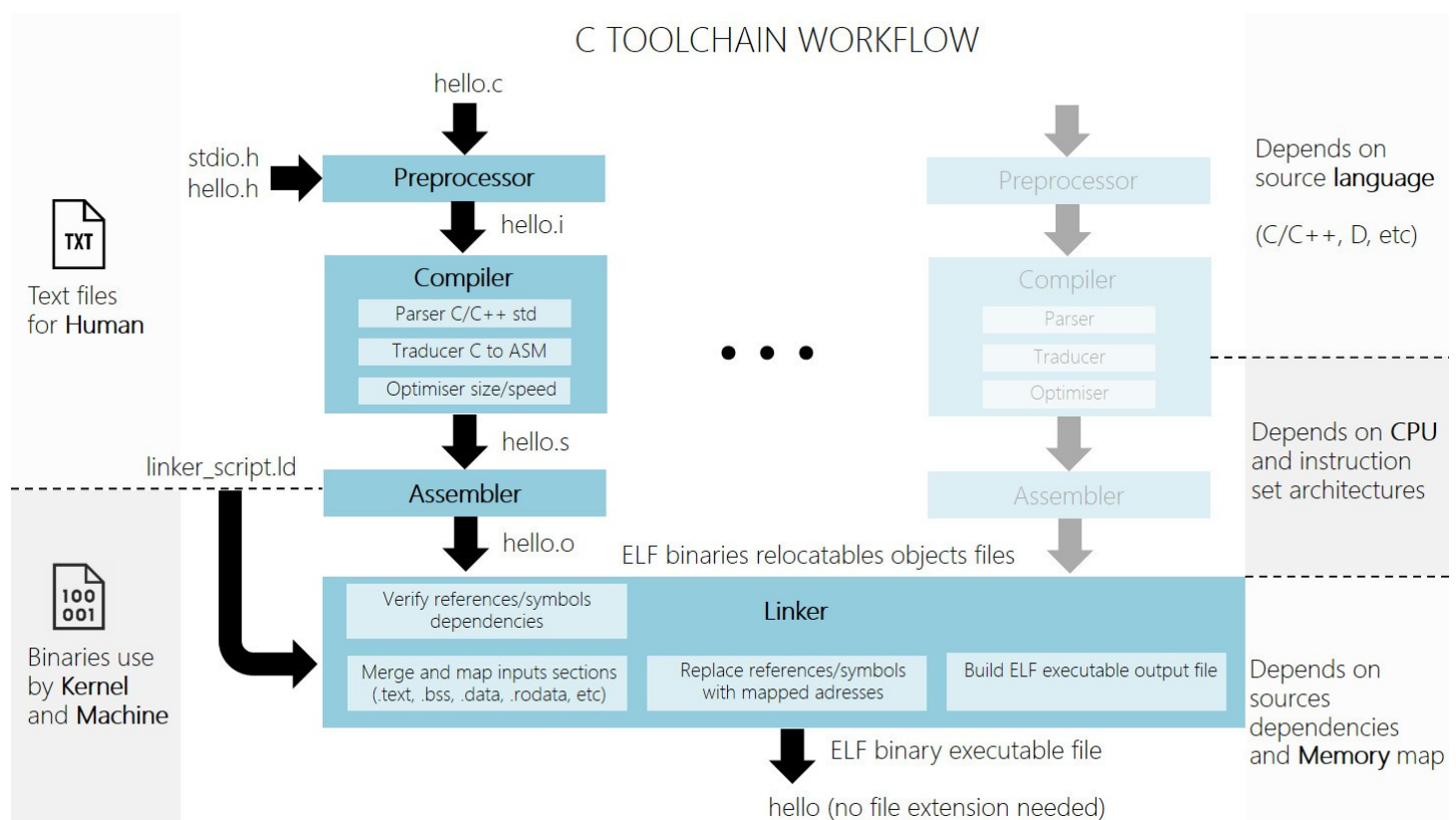
```
./build/bin/hello
```

V. Synthèse sur l'exercice

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

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

C TOOLCHAIN WORKFLOW



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

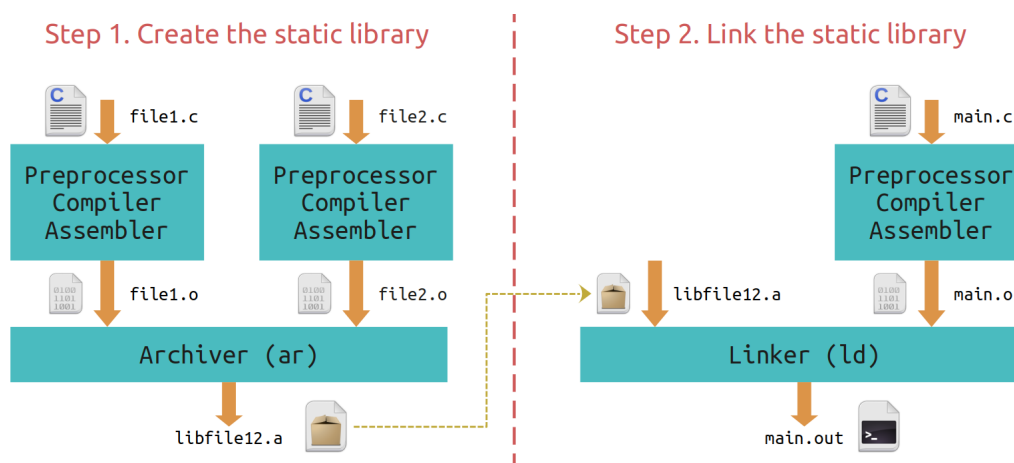
VI. Librairie statique

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

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

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

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



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

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

Retrouvez et indiquez le type des fichiers binaires générés.

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

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

VII. Make et Makefile

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

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

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

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

```
make
```

Qu'est-il affiché sur la console ?

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

```
ls -lt
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

