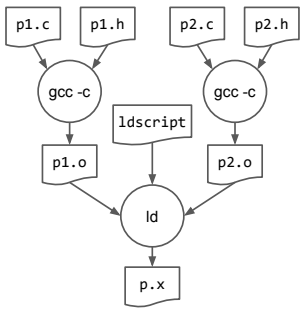
ALMO

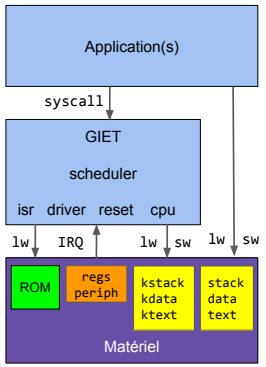
chaîne de compilation
introduction du GIET

Plan

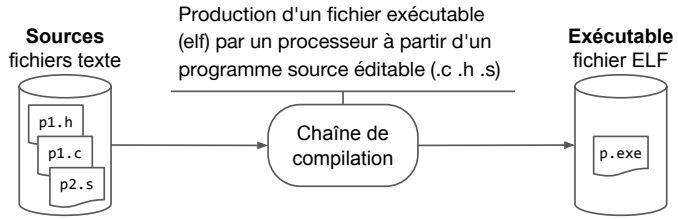
Chaîne de compilation



GIET

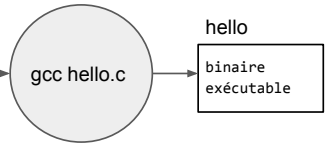


Objectif de la chaîne de compilation



hello.c

```
#include <stdio.h>
int main(int argc, char * argv[]) {
     printf( "Hello World\n");
     return 0;
}
```



Étapes de compilation

1. Préprocesseur : (.c + .h) → (.c)
   - Remplacement des macro-instructions (#define)
   - Ouverture des fichiers inclus (.h) contenant les déclarations externes de variables et de fonctions.
2. Compilateur : (.c) → (.s)
   - Transformation du code source (.c) en langage d'assemblage (.s)
   - Analyse syntaxique et génération de code avec plusieurs niveaux d'optimisation
3. Assembleur (.S ou .s) → (.o)
   - Génération du code binaire
4. Editeur de lien (.o) → (.x)
   - Unification de tous les fichiers objets pour produire un exécutable

Compiler un fichier source C signifie réaliser les 3 premières étapes :
⇒ *Preprocessing*, compilation et génération du code binaire
L'édition de lien est la dernière étape

Chaîne de compilation GNU

GNU propose une chaîne de compilation permettant de produire un exécutable à partir de programme source

gcc	GNU C compiler
as	Assembleur
ld	Editeur de liens
cpp	Préprocesseur

mais pas seulement

objdump	désassembleur
gdb	débugger
nm	listage des symbole d'un fichier objet

compilation native et croisée

Compilation native

lorsque le compilateur produit du code pour la machine et le système d'exploitation sur laquelle a été faite la compilation a été effectuée

→ `gcc, as, ld, objdump`

Compilation croisée

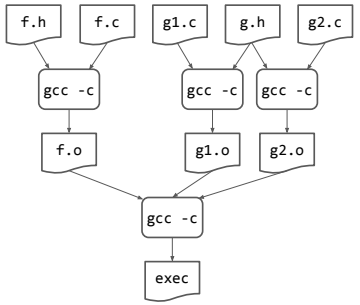
lorsque le compilateur produit du code pour une autre machine et un autre système d'exploitation que la machine sur laquelle a été faite la compilation a été effectuée.

→ `cpu-os-format-tool`

- assembleur `as` `mipsel-mars-elf-as`
- compilateur `gcc` `mipsel-mars-elf-gcc`
- linker `ld` `mipsel-mars-elf-ld`
- désassembleur `objdump` `mipsel-mars-elf-objdump`

Compilation séparée

Compilation de plusieurs fichiers sources indépendants
Puis réunion en un seul fichier exécutable




```
gcc -c -Wall f.c
gcc -c -Wall g1.c
gcc -c -Wall g1.c
gcc f.o g1.o g2.o -o exec
```


Makefile

Un makefile est un fichier contenant la méthode de construction d'un fichier cible à partir de ces sources. Un Makefile est interprété par **make**

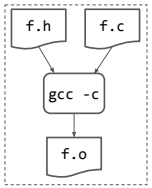
Un makefile a deux objectifs :

1. Capitaliser la méthode de construction permettant de la rejouer après un changement des sources.
2. Permettre une reconstruction sélective en n'exécutant que les étapes de construction nécessaires qui dépendent des sources modifiées

Un makefile est constitué de règles

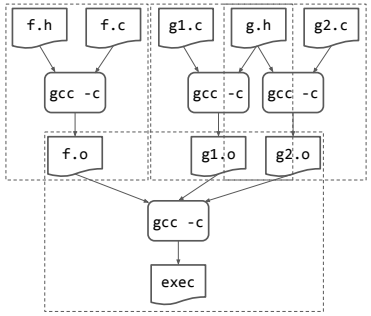
```
cible : dependances…
	commandes
```

tab



Makefile : exemple 1

4 règles



bash> make exec

Makefile

```
f.o : f.c f.h
    gcc -c -Wall f.c

g1.o : g1.c g.h
    gcc -c -Wall g1.c

g2.o : g2.c g.h
    gcc -c -Wall g1.c

exec : f.o g1.o g2.o
    gcc f.o g1.o g2.o -o exec
```

make lit le Makefile entièrement, construit le graphe de construction et produit la cible demandée uniquement si sa date en antérieure à l'une de ces dépendances.

Makefile : exemple 2

Le format Makefile est très riche

- Il permet d'exprimer des règles génériques
  - La dépendance de la cible utilise le caractère % qui représente un nom de fichier quelconque
  - Les commandes utilisent des variables automatiques dont la valeur est extraite de la ligne de dépendance de la règle
    - $@ : cible
    - $< : première dépendance
    - $^ : toutes les dépendances
    - $* : %
- on peut ajouter des règles de commandes
  - clean ou all et le .PHONY pour les désigner
- on peut utiliser des variables
  - CFLAGS, EXE
- On peut même faire des boucles...

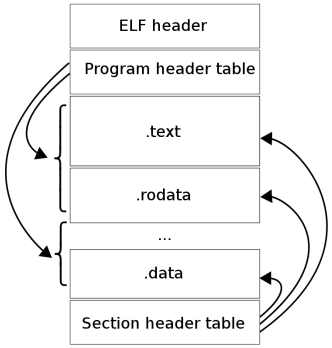
⇒ https://www.gnu.org/software/make/manual/

```
CFLAGS = -Wall
OBJ = f.o g1.o g2.o
EXE = exec
.PHONY = all clean

all: clean $(EXE)

%.o : %.c
    gcc -c $(CFLAGS) $<.c
$(EXE) : $(OBJ)
    gcc $^ -o $@

clean:
    rm $(OBJ)

f.o : f.c f.h
g1.o : g1.c g.h
g2.o : g2.c g.h
```


Format elf (Executable and Linkable Format)

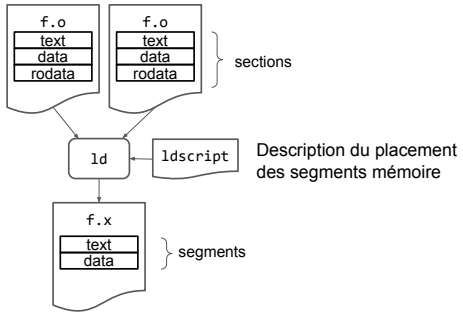
Après compilation on obtient un fichier objet ou exécutable au format elf.

- code binaire du processeur cible
- données globales
  - plusieurs sections
    - initialisées != 0
      - data
      - rodata
      - sdata
      - …
    - non initialisée
      - bss
      - sbss
- Table relocation
  - permet de connaître les symboles utilisé mais non défini localement



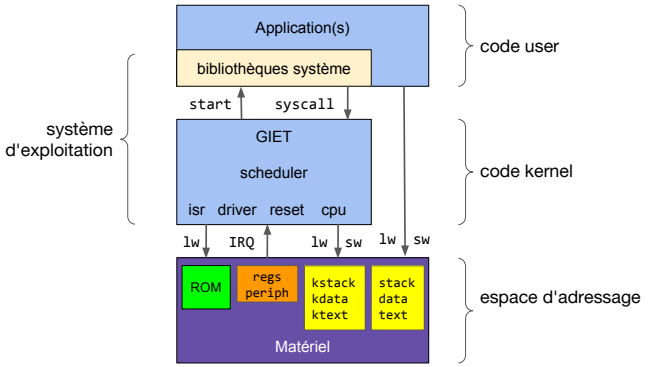
crédit : wikipedia

Edition de lien




```
seg_code_base   = 0x00400000;
seg_data_base   = 0x10000000;

SECTIONS
{
    . = seg_code_base;
    seg_code : {
        *(.text)
    }
    . = seg_data_base;
    seg_data : {
        *(.ctors)
        *(.rodata)
        *(.rodata.*)
        *(.data)
        *(.lit8)
        *(.lit4)
        *(.sdata)
        *(.bss)
        *(COMMON)
        *(.sbss)
        *(.scommon)
    }
}
```

Piles logicielles



Rôles du noyau du système d'exploitation

1. Gestion des services proposés aux applications : syscall (Trap)
   - accès aux périphériques : terminal, disque, réseau, etc.
   - allocation de mémoire
   - allocation du processeur
2. Gestion des interruptions émanant des périphériques
   - fin de commande
   - interruption d'horloge pour la gestion du temps partagé.
3. Gérer les erreurs de programmation où les allocations à la volée
   - division par 0
   - violation de privilège
   - instruction illégale
   - faute page

⇒ GIET : Gestionnaire des Interruptions, Exceptions et Trap

Prochaine séance