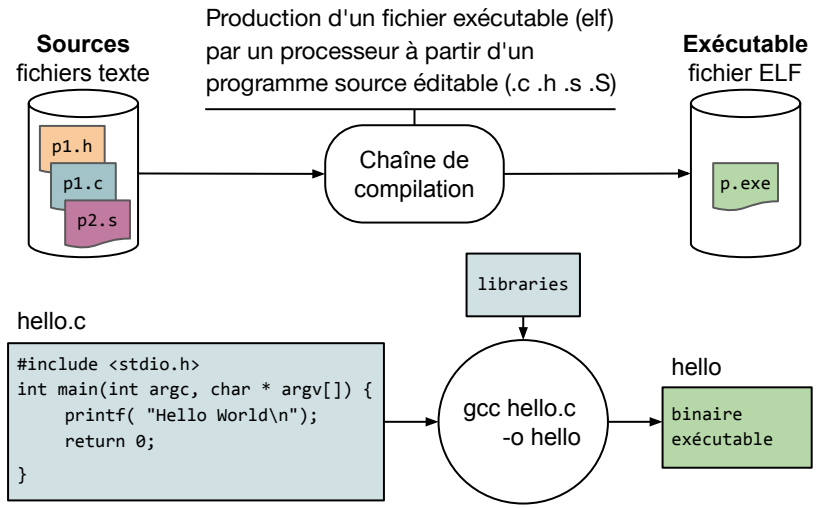
ALMO

chaîne de compilation

Plan

- Chaîne de compilation
- Outils de construction et d'analyse
- Compilation séparée
- Makefile
- Une introduction au rôle du système d'exploitation

Chaîne de compilation



Étapes de compilation

1. Préprocesseur : (.c + .h) → (.c)
   - Expansion des macro-instructions (#define)
   - Ouverture des fichiers inclus (.h) contenant les déclarations externes de variables et de fonctions.
2. Compilateur : (.c) → (.s)
   - Analyse syntaxique et génération de code avec plusieurs niveaux d'optimisation
   - Transformation du code source (.c) en langage d'assemblage (.s)
3. Assembleur (.S ou .s) → (.o)
   - Génération du code objet (code binaire pour le processeur cible)
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

cpp	préprocesseur (gcc -E file.c)
gcc	GNU C compiler (gcc -c file.c)
as	Assembleur (gcc -S file.c)
ld	Éditeur de liens (gcc file.o)

mais pas seulement

objdump	désassembleur
gdb	debugger (pour exécuter en pas à pas)
nm	listage des symboles d'un fichier objet
readelf	affichage du contenu d'un fichier elf

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

→ `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

→ `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`

Préprocesseur du C

Le préprocesseur transforme le code C et produit un nouveau code C

- efface les commentaires
- interprète les directives : `#directive` (le `#` doit être en début de ligne)

Il y a 3 Usages des directives

- Expansion de macro instructions
  permet le remplacement d'un identifiant (macro) par sa définition.
  Ces définitions peuvent être paramétrées avec des arguments
- Compilation conditionnelle
  permet de supprimer une partie des lignes de code source dans certaines conditions
- Inclusion
  permet d'inclure des déclarations de fonctions ou des définitions de macros dans un fichier C
- Directives pour le compilateur
  permet d'informer ou d'interroger le compilateur

Préprocesseur du C : expansion de macro

```
#define MACRO
#define MACRO  DÉFINITION
#define MACRO(a1,a2,..,an)  DÉFINITION_AVEC_ARGUMENTS
#undef  MACRO
```

- Attention, on ne peut pas mettre de commentaire derrière une définition
- Une définition de macro est sur une seule ligne ou utiliser le caractère \
- Exemples

```
#define DEBUG
#define ROUGE   4
#define MAX(a,b) ((a)>(b)?(a):(b))          → noter les ( )
#define INCV(v)  do{v.x++;v.y++;}while(0)   → si plusieurs instructions dans
[…]                                           la définition de la macro
struct v_st {int x, y} v1, v2;
if (MAX(4, 2*i+j) < ROUGE)
    INCV(v1);
else
    INCV(v2);
[…]
```

Préprocesseur du C : compilation conditionnelle

Permet de sélectionner le code à inclure pour adapter le code à la machine ou traiter le debug

`Directives : #if  #ifdef  #ifndef  #else  #elif  #endif`

```
#ifdef DEBUG
    code 1
#else
    code 2
#endif
```

```
#if !(defined DEBUG)
    code 1
#else
    code 2
#endif
```

```
#if VERBOSE > 1
    code 1
#elif VERBOSE > 0
    code 2
#else
    code 3
#endif
```

```
#if 0
    code
#endif
```

```
#if !(defined __MIPS__ || defined MIPS32)
    code 1
#endif
```


Préprocesseur du C : inclusion

`#include` permet d'inclure un fichier dans un autre

- `#include <`*`fichier.h`*`>` ou `#include "`*`fichier.h`*`"`
  - inclut *`fichier.h`* dans le fichier contenant la directive `#include`
  - avec " " : *`fichier.h`* est recherché dans le répertoire courant
  - avec < >: *`fichier.h`* est recherché dans les répertoires standards tel que /usr/include et dans les répertoires donnés par l'argument `-I`
- Recommandation
  - N'inclure que des .h, pas des .c
  - Se prémunir contre la double inclusion (évite la redéfinition des macros)

```
#ifndef _FICHIER_H_
#define _FICHIER_H_
    …
#endif
```


Préprocesseur du C : autres directives et macros

```
#error     "message"  : affiche "message" et stoppe la compilation
#warning   "message"  : affiche "message"
__FILE__              : MACRO contenant le nom du fichier courant
__LINE__              :   "       "     le numéro de ligne
__DATE__              :   "       "     la date
__TIME__              :   "       "     l'heure
__FUNC__              :   "       "     la fonction courante
```

mise entre guillemet:

```
        #define str(s)      #s      : str(bonjour)   → "bonjour"
```

concaténation

```
        #define concat(a,b)  a##b    : concat(A,B)  → AB
```

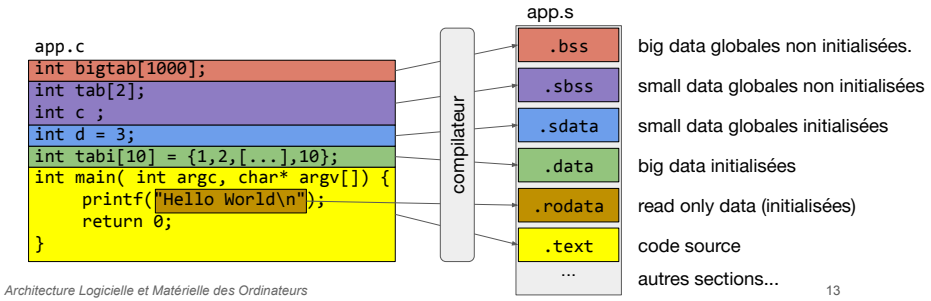

Compilateur C : sections

- Le compilateur vient après le préprocesseur et produit le code assembleur pour le processeur cible. La compilation se déroule en plusieurs étapes :
  - les analyses : lexicale, syntaxique et sémantique
  - la génération d'un code intermédiaire (indépendant du processeur)
  - les optimisations qui produisent un code pour le processeur cible
- Nous n'allons pas détailler ces étapes parce que cela sort du cadre de ce module. Nous allons voir le résultat de la compilation et quelques arguments du compilateur.
- Nous n'allons pas voir le langage C, mais nous verrons plus tard quelques spécificités nécessaires pour la programmation système (c.-à-d. de l'OS)
- La sortie du compilateur est du code assembleur (.s) pour le processeur cible
- En général, on lance la génération de code binaire (.o) produit par l'assembleur dans la foulée du compilateur
- Dans tous les cas, le fichier généré n'est pas exécutable parce que le fichier source .c ne contient pas la définition de toutes les fonctions. Il faudra faire l'édition de liens, c'est ce que nous allons voir après.

Compilateur C

Le compilateur place le code et les données globales dans des sections typées

- Une section est un segment d'adresses consécutives.
  - Le code est dans une section .text
  - Les données globales sont placées dans des sections différentes en fonction de leur type, de taille et du fait quels sont initialisées ou pas.
  - Les données non initialisées seront initialisées au lancement de l'application.
- Il n'y a pas de sections pour les variables locales ou les données dynamiques car ce sont des données qui n'existent qu'à l'exécution du programme.
- Les **sections** commencent toutes **à l'adresse 0**. Elles auront une adresse dans l'espace d'adressage au moment de la génération du programme exécutable (cf. éditeur de liens)



Assembleur

Vous avez déjà vu les bases du langage assembleur MIPS (il vous manque le langage des macro-instructions mais il n'est pas utile dans ce module).
Le programme assembleur génère du code objet à partir du code assembleur.

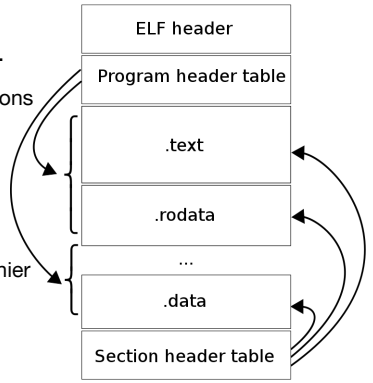
```
as -MIPS32 <file.s> -o <file.o>
```

Principales directives (https://sourceware.org/binutils/docs/as/)

- `.text .data` : indique la section à remplir
- `.section` *`name[, "flags"][, "type"]`* (il y a d'autres flags et types)
  - *`flags`* : `wax` ⇒ `writable, allocated, executable`
  - *`type`* : `@prognobit | @nobits` ⇒ resp. avec et sans data
- `.globl` *`label`* : indique que ce label est visible des autres fichiers
- `.word|.ascii|.asciiz|.byte` : permet d'allouer de la place, c'est suivi des valeurs
- `.space` *`size`* : alloue de la place non initialisée
- `.align` *`n`* : déplace le ptr. de remplissage à la prochaine adresse 2^n
- `.func` *`<func_name>`* : informe du début d'une fonction (pour le debug)
- `.endfunc` : fin de fonction
- `.size` *`name, expression`* : permet de donner une taille (en octets) au symbole *`name`*
  p.ex. : `.size func_name,.-func_name`

Format elf (Executable and Linkable Format)

L'assembleur (ou directement le compilateur) produit un fichier objet ou exécutable au format elf.



crédit : wikipedia

- ELF header : identifiants, types de fichier et dimensions
- Program header table :
  description des segments du programme exécutable sous forme d'un tableau de structures pointant dans le fichier
- Section header tbl. : description des sections du fichier
- section .text : code binaire du processeur cible
- plusieurs sections de données
  - data initialisées !=0 : .sdata, .data, .rodata, …
  - data non initialisée : .sbss, .bss
- section contenant des tables de relocation
  - permet de connaître les symboles utilisés mais non défini localement

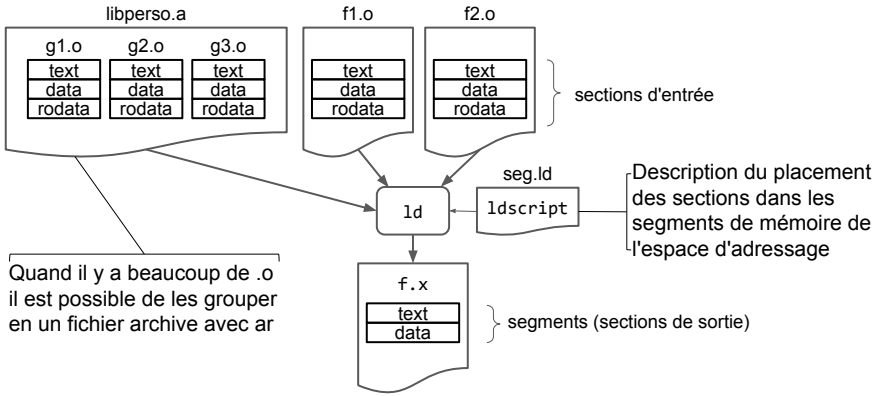
readelf / objdump / nm

Ces applications permettent d'afficher le contenu d'un fichier objet.

- `readelf`
  - nécessite le format elf
  - décrit en détail tout le contenu du fichier
  - p.ex: `readelf -a file.o`
- `objdump`
  - gère plusieurs formats objet (elf, coff, mach-O, etc).
  - donne moins d'informations sur le contenu du fichier
  - mais désassemble les sections en code assembleur
  - p.ex: `objdump -D file.o`
- `nm`
  - affiche les symboles définis ou utilisés dans un fichier elf
  - p.ex: `nm -a file.o`

Edition de liens

Le compilateur et l'assembleur ont produit des fichiers objets avec du code binaire mais chacun est incomplet et les sections ne sont pas mappées dans l'esp. d'adr. Il faut les lier (les unir) pour produire un fichier exécutable (au format elf pour gcc).




```
ld -o f.x f1.o f2.o -lperso -Tseg.ld
```


Archives de fichiers objet

- Lorsque l'on fait l'édition de lien avec un file.o, tout le fichier est copié dans le programme exécutable. Si file.o contient un grand nombre de fonctions et que seules quelques-unes sont utilisés, il va y avoir du code inutile dans l'exécutable.
- On peut décomposer le code en mettant très peu de fonctions par fichier, cela évite le problème précédent mais cela augmente le nombre de fichiers objet.
- On peut grouper tous les fichiers `objet .o` dans une archive avec l'archiver `ar`

```
ar [OPTIONS] archive_name files
OPTIONS:c   crée l'archive si elle n'existe pas
        r   ajoute les fichiers dans l'archive
        s   crée ou met à jour un index dans l'archive
```

- Quand on crée une archive pour l'éditeur de lien on la nomme : `libname.a` et pour l'utiliser on écrit juste : `-lname` en retirant le préfixe `lib` et l'extension `.a`
- L'éditeur de liens va ouvrir l'archive et prendre les fichiers objets dans lesquels se trouvent les fonctions ou les variables dont il a besoin.
- Il s'agit ici de bibliothèque statique, c'est-à-dire que les fonctions sont vraiment ajoutées dans l'exécutable, il est possible de faire une édition de lien dynamique pour que le code des fonctions ne soit ajouté que lorsqu'elles sont utilisées.

Edition de lien : fichier ldscript

Le fichier **ldscript** décrit comment remplir l'espace d'adressage, c'est un script pour ld.

déclaration de variable dans ld

. représente le pointeur de remplissage

on définit ce pointeur aux adresses des segments mémoire

fichier objet, * est un jocker qui prend tout

contenu de la section

seg_data est une section de sortie qui est un segment d'adresse dans la mémoire

Les sections sont prises dans l'ordre

```
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)
    }
}
```

Il y a plusieurs syntaxe possibles, vous en voyez une, ce qui est important est de comprendre que le ldscript décrit l'espace d'adressage et comment il est rempli avec les sections produites par le compilateur.

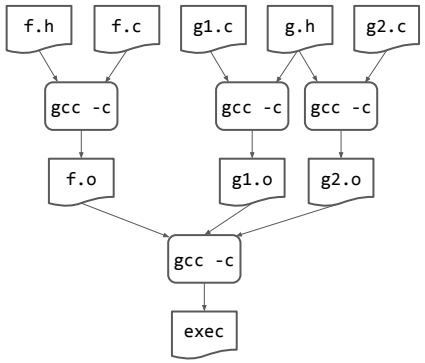
Compilateur C : arguments

L'application gcc permet d'appeler le préprocesseur, le compilateur, l'assembleur et l'éditeur de liens. La documentation du compilateur 9.2.0 fait presque 1000 pages. Nous n'allons voir que quelques arguments… (ceux utilisés en TME)

```
gcc -E                  : stoppe gcc après le préprocesseur
    -c                  : stoppe gcc après le compilateur produit un .o ou un .s
    -S                  : stoppe gcc après l'assembleu
    -o <file>           : nom du fichier de sortie <file>
    -g                  : ajoute du code pour le debugger
    -I <dir1:dir2..>    : ajoute <dir1:dir2..> aux répertoires de recherche des #include
    -L <dir1:dir2..>    : ajoute <dir1:dir2..> aux répertoires de recherche des libraries
    -l <lib>            : ajoute la library lib
    -T <ldscript>       : informe l'éditeur de lien du nom du ldscritp (s'il est invoqué)
    -Wall               : tous les warnings du langage C (ou presque) sont relevés
    -Werror             : tous les warnings sont considérés comme des erreurs
    -mips32             : informe le compilateur du type de MIPS
    -fomit-frame-pointer : demande d'utiliser $29 comme seul pointeur de pile
    -ffreestanding      : demande au compilateur de ne pas
    -mno-gpopt          : demande de ne pas utiliser de global pointer pour les
                          variables globale (gp utilise $28 qui pointe dans .data
                          et cela permet un accès relatif aux variables : lw $x, imm($28)
```

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

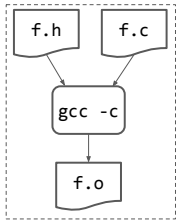
Un makefile est un fichier contenant la méthode de construction d'un fichier cible à partir de ces sources. Un Makefile est interprété par **make**

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

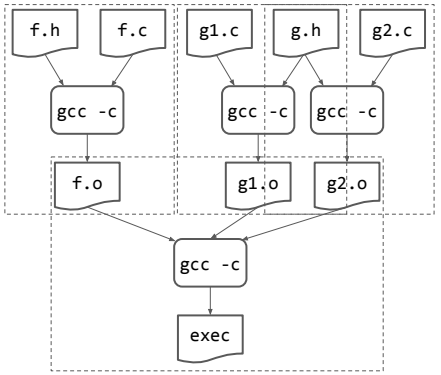
Un makefile est constitué de règles

```
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...

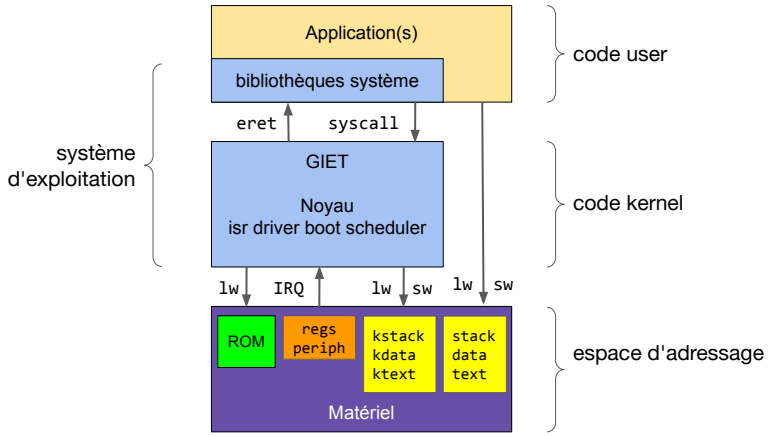
⇒ https://www.gnu.org/software/make/manual/

```
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
```

En résumé, nous avons vu :

- De quoi est composé la chaîne de compilation
- Ce que signifie compilation native et compilation croisée
- A quoi servent les étapes des :
  - préprocesseur
  - compilation
  - edition de liens
- Ce qu'est une section, un segment et à quoi ils servent
- A quoi sert une archive de fichier objet et comment l'utiliser
- Quelques outils de la chaîne de compilation et quelques paramètres
- A quoi sert un Makefile et deux exemples simples

Place du système d'exploitation



Rôles et devoirs du système d'exploitation

Rôles :

1. Gérer les ressources matérielles et les services pour les applications
   - commande aux périphériques : terminal, disque, réseau, etc.
   - allocation de mémoire
   - allocation du processeur
2. Gérer les événements émanant des périphériques
   - fin de commande aux périphériques
   - Tick d'horloge pour réaliser des opérations périodiques
3. Gérer les erreurs de programmation où les allocations à la volée
   - division par 0
   - violation de privilège
   - instruction illégale
   - faute de page, erreur de segmentation (bus error)

Devoirs : l'OS doit garantir

- la **qualité de service** (QoS) (équité, priorité ou respect des échéances),
- la **sécurité** des applications (confidentialité, intégrité, et accès aux services (DoS))
- la **sûreté** de fonctionnement du matériel (pour éviter la casse)

Attention : Sécurité est différent de Sûreté

Prochaine séance