

Plan

1. Généralités :

- compilateur, assembleur, éditeur de liens
- compilation séparée
- directives #include, ...
- espace d'adressage d'un programme

2. l'outil make

- cible, dépendance
- fichier Makefile de base
- un peu plus sur le fichier Makefile

3. Annexes

- Structure des fichiers objets
- Structure des fichiers exécutables

Chaîne de production

1. Généralités :

- compilateur, assembleur, éditeur de liens
- compilation séparée
- directive `#include`, bibliothèques

2. l'outil make

- cible, dépendance
- fichier `Makefile` de base
- un peu plus sur le fichier `Makefile`

Chaîne de production d'un programme

Pour exécuter un programme, on passe par les étapes suivantes :

- écriture d'un ou plusieurs fichiers sources qui sont automatiquement rangés sur disque par l'éditeur,
- utilisation, éventuellement complètement transparente, d'OUTILS DE PRODUCTION pour obtenir un fichier exécutable. Ce fichier, comme les fichiers sources, est rangé sur disque.
- pour exécuter ce fichier, on le charge en mémoire. Ceci est fait en "tapant" le nom de ce fichier exécutable qui est alors chargé du disque vers la mémoire par le système d'exploitation.

Pour le fichier exécutable, on parle également de fichier binaire.

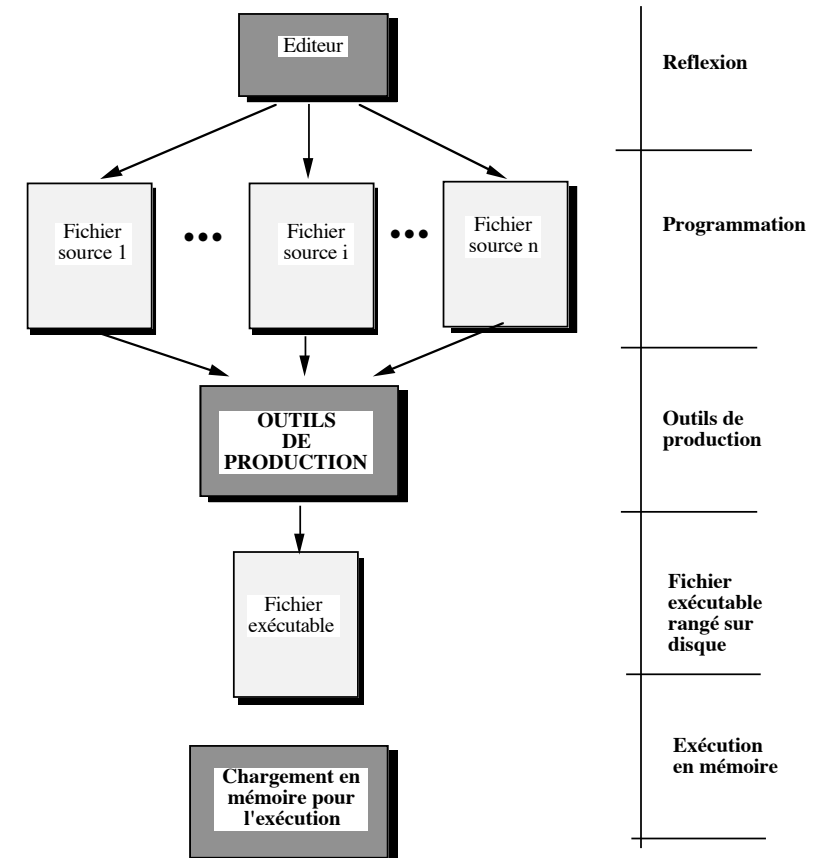
On utilise ce terme pour signifier que ce fichier contient des instructions et des données représentées suivant un format et un codage compréhensibles par le processeur de la machine.

Les fichiers sources, par contre, sont lisibles par l'utilisateur, mais incompréhensibles pour le processeur. Le rôle du compilateur est principalement de traduire le fichier source, écrit en langage de haut niveau, espéranto des programmeurs, dans le langage du processeur qui pilote la machine.

Les outils de productions aident à automatiser ce passage du source à l'exécutable, surtout dans le cas de l'existence de nombreux fichiers sources.

Chaîne de production d'un programme

Place des outils de production



Chaîne de production d'un programme

1-Compilateur

Traduit les modules sources écrits en langage de haut niveau (C, Ada, ...) en modules objets qui contiennent du langage machine mais dans lesquels *les calculs d'adresses ne sont pas résolus*.

2-Editeur de liens (*linker*) :

rassemble les modules traduits séparément par les traducteurs, les relie et produit un module chargeable (ou relogeable - *relocatable* -).

Fonctions de l'éditeur de liens :

Pour fusionner les espaces d'adressage séparés des modules objet en un seul espace linéaire, l'éditeur de liens :

- Construit une table qui indique le nom, la taille et la longueur de tous les modules objet,
- Affecte une adresse de chargement à chaque module objet,
- Effectue la translation en modifiant les instructions qui contiennent une référence mémoire,
- Résout les références externes en insérant l'adresse des procédures à l'endroit où elles sont appelées.

3- Chargeur (*loader*) :

On rappelle que le processeur va chercher en MEMOIRE les informations dont il a besoin, c'est-à-dire le programme et les données. Le rôle du chargeur est donc d'aller chercher sur un disque les fichiers exécutables (le programme) et les données pour les implanter en mémoire.

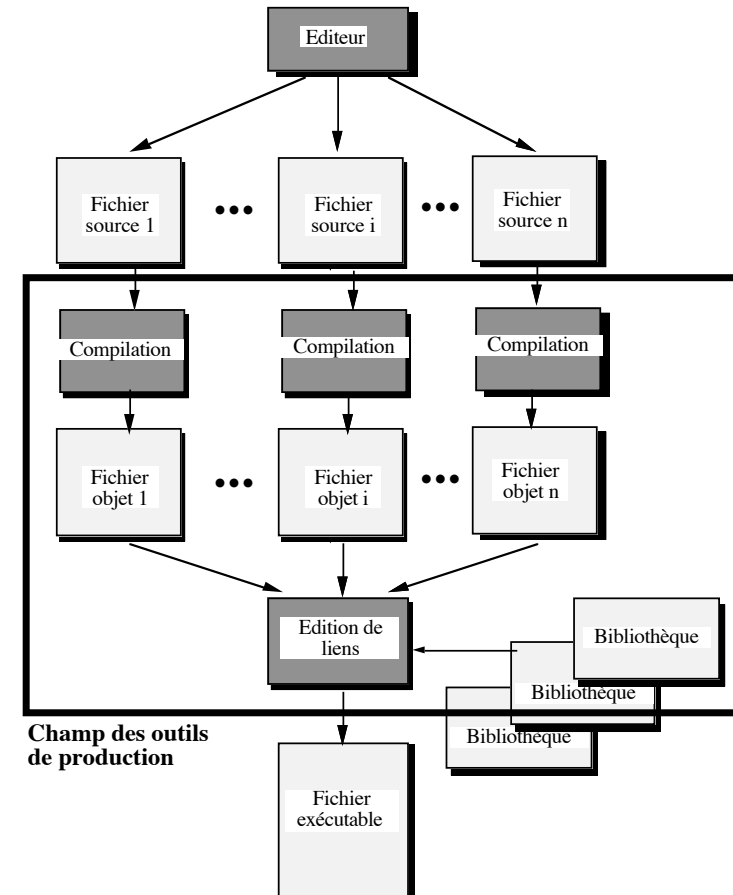
Compilateur (rappel)

Le compilateur :

- analyse le programme au niveau syntaxique et sémantique,
- traduit le langage de haut niveau en langage machine,
- affecte des adresses aux variables.

Chaîne de production d'un programme

Rôle des outils de production



Chaîne de production d'un programme

Interêt de la compilation séparée :

- Maintenance plus facile,
- Modules réutilisables,
- Par exemple si l'exécutable fifi est construit à partir de f1.c et f2.c et si on modifie f1.c, on ne fera que les actions impliquant f1.c, c'est à dire :

```
gcc -c f1.c
gcc f1.o f2.o -o fifi
```

Quelques options de la commande gcc :

gcc -c compilation et production d'un fichier objet

gcc -o toto compilation et production d'un fichier exécutable appelé toto
Sans l'option -o, gcc range l'exécutable dans un fichier appelé a.out.

gcc -S compilation et production d'un fichier source assembleur

Chaîne de production d'un programme

Compilation séparée

- Contenu des fichiers f1.c et f2.c :

f1.c	f2.c
<pre>int main (argc, argv){ int i = 0; i = carre (4); printf ("i = %d\n", i); return 0 ; }</pre>	<pre>float carre (float var){ return (var * var); }</pre>

- Compilation séparée :

gcc -c f1.c (produit le fichier OBJET f1.o)

gcc -c f2.c (produit le fichier OBJET f2.o)

gcc -o f f1.o f2.o

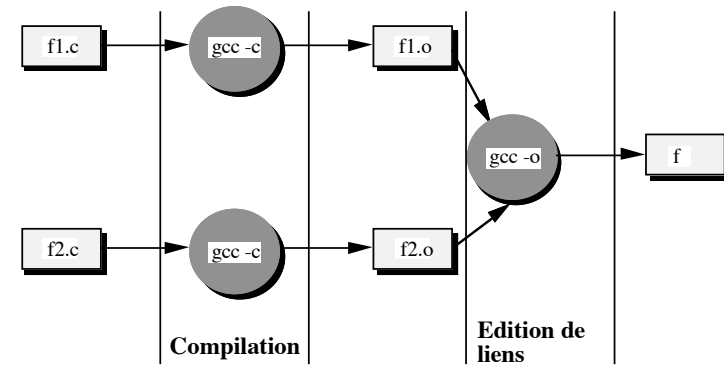
(produit le fichier EXECUTABLE f)

gcc -c	Compilation et production d'un fichier objet
gcc -o	Edition de liens et production d'un fichier exécutable

- **Remarque** : les fonctions printf et carre n'ont pas été prototypées...

**Compilation séparée
(résumé)**

- Production d'un fichier exécutable f à partir des fichiers sources $f1.c$ et $f2.c$:



- Utilisation de gcc dans le schéma ci-dessus :

`gcc -c f1.c` (produit l'objet $f1.o$)

`gcc -c f2.c` (produit l'objet $f2.o$)

`gcc f1.o f2.o -o f` (produit l'exécutable f)

**Compilation : pourquoi
prototyper**

- Reprenons l'exemple précédent :

f1.c	f2.c
<pre>int main (argc, argv){ int i = 0; i = carre (4); printf ("i = %d\n", i); return 0 ; }</pre>	<pre>float carre(float var){ return (var * var); }</pre>

- 1 - L'exécution de : gcc **-Wall** -c f1.c

donne le message suivant :

warning implicit declarations: carre, printf

Pourquoi ? **prototypages** mal faits dans f1.c, en effet il manque le prototype (ou signature) de carre et #include <stdio> pour printf.

- Cependant le fichier objet f1.o est créé et on peut fabriquer un exécutable, mais l'exécution du programme peut donner ce résultat (le résultat va dépendre des machines):

i = 4

Préprocesseur :
#include

- Comme on l'a vu un programme peut ne pas donner un résultat correct parce que l'appel d'une fonction ne correspond pas à sa définition (son implémentation).

Solution :

- Il faut toujours utiliser l'option `-Wall` du compilateur `gcc` qui permet de détecter ces erreurs,
- il faut associer à chaque fichier source (type `.c`) un fichier (type `.h`) qui contient les prototypes des fonctions décrites dans ce fichier source,
- Le programmeur qui utilise des fonctions définies ce fichier source doit inclure le fichier contenant les prototypes pour éviter toute utilisation erronée de ces fonctions

Dans les fichiers <code>.h</code>	Dans les fichiers <code>.c</code>
Prototype, Signature, Déclaration,	Implémentation, Définition,

Préprocesseur : **#include**

- Ici, on crée le fichier f2.h qui contient les prototypes des fonctions décrites dans f2.c.

f1.c	f2.c	f2.h
<pre>#include <stdio.h> #include "f2.h" int main (argc, argv){ int i= 0; i = carre (4); printf("i=%d\n",i); return 0; }</pre>	<pre>float carre(float var){ return (var * var); }</pre>	<pre>float carre(float);</pre>

- Lors de l'exécution de

`gcc -Wall -c f1.c`

1-le préprocesseur inclut f2.h et stdio.h en tête de f2.c,

2-le compilateur connaît donc les types des différents paramètres et applique les conversions de type dans le fichier généré,

- A l'exécution, on obtient :

`i = 16`

Bibliothèques et fichiers #include

- Si le programme contenu dans `f1.c` est à nouveau modifié comme suit :

```
#include <stdio.h>
#include "f2.h"
int main (argc, argv){
    int i;
    i = carre (4);
    printf ("i = %d\n", i);
    i = sqrt (i);
    printf ("i = %d\n", i);
    return 0 ;
}
```

alors, le résultat des commandes :

```
gcc -c -Wall f1.c
gcc -c -Wall f2.c
gcc f1.o f2.o
```

sera :

```
warning: type mismatch
ld : Undefined Symbol sqrt
```

- pour remédier au premier problème (warning) il faut inclure le fichier qui contient les PROTOTYPES des fonctions mathématiques : `math.h`
- pour remédier au second problème (ld ...) : il faut charger la BIBLIOTHEQUE mathématique.

On indique à l'éditeur de liens d'aller chercher une bibliothèque avec l'option `-l`. Pour la bibliothèque mathématique l'option est : `-lm`

***Bibliothèques
et fichiers #include :
exemple***

- Si on modifie maintenant le fichier f1.c comme suit :

```
#include <stdio.h>
#include <math.h>
#include "f2.h"
int main(argc, argv){
    int i;
    i = carre (4);
    printf ("i = %d\n", i);
    i = sqrt (i);
    printf ("i = %d\n", i);
    return 0 ;
}
```

alors, l'exécution des commandes

```
gcc -c -Wall f1.c
gcc -c -Wall f2.c
gcc f1.o f2.o
```

donnera le message d'erreur :

```
ld: fatal: Undefined symbol sqrt
```

(pas d'exécutable, il manque un fichier objet). Il faut ajouter le chargement de la bibliothèque mathématique:

```
gcc f1.o f2.o -o f -lm
```

Chaîne de production d'un programme

- L'exécution de : `gcc -o f f1.o` donne le message:

```
ld :undefined symbol carre
```

```
ld fatal error
```

Pourquoi ?

fichier objet manquant (on n'a pas chargé `f2.o` qui contient le programme de la fonction `carre`!).

Donc, pas de création de fichier exécutable...

Chaîne de production d'un programme

Erreurs : récapitulatif

	Effet
- pas de <code>#include</code>	pas d'exécutable
- pas de chargement de bibliothèque	
- pas de <code>#include</code>	production d'un exécutable mais problèmes possibles à l'exécution
- chargement de bibliothèque	
- avec <code>#include</code>	pas d'exécutable
- pas de chargement de bibliothèque	

- Mise au point d'une application (*debugging*, débogage) :

Les programmes ne fonctionnent généralement pas correctement dès la première exécution. Lors de la mise au point d'une application volumineuse, la technique du `printf` s'avère vite fastidieuse et inefficace.

Pour une mise au point efficace, on utilise l'option "debug" qui permet d'exécuter un programme en mode pas à pas, de visualiser le contenu des variables, leur adresse, etc.

Cette option "debug" (option `-g`) doit être indiquée à la compilation et à l'édition de liens :

```
gcc -g -c f1.c  
gcc -g -c f1.c  
gcc -g f1.o f2.o -o f
```

***Le préprocesseur :
la directive #define***

- Cette directive effectue des **substitutions** de chaînes de caractères, elle ne définit pas de constantes,
- On peut constater son effet en utilisant une des options de gcc : **gcc -E**

seul est activé le préprocesseur, il traite toutes les lignes commençant par #

Fichier titi.c :	Effet de gcc -E titi.c
<pre>#define MAX 100 #define double1(x) (2*x) #define double2(x) (2*(x)) int main (argc, argv){ long int i, j, k; i = MAX; j = 0; k= double1(1+2); k= double2(1+2); return 0; }</pre>	<pre>int main (argc, argv){ long int i , j , k; i = 100; j = 0 ; k = (2 * 1 + 2); k = (2 *(1 + 2)); return 0 ; }</pre>

- Attention aux effets de bord : ici, les parenthèses qui encadrent x les évitent

***Le préprocesseur :
la directive #ifndef
(exemple)***

- Si on ne peut éviter d'inclure plusieurs fois un fichier du type « .h », le compilateur va détecter des redéfinitions. Pour les éviter, on utilise la directive : `#ifndef`.
- Exemple :

Fichier fonctions.h	Fichier fonctions.h modifié
<pre>void fonc1(int i, int j); int fonc2(float r);</pre>	<pre>#ifndef FONC1_H #define FONC1_H void fonc1(int i, int j); int fonc2(float r); #endif</pre>

- Lors de la première inclusion de `fonctions.h`, `FONC1_H` n'est pas définie, donc le préprocesseur traite les lignes comprises entre la directive `#ifndef` et le `#endif` associé.
- Lors des inclusions suivantes, `FONC1_H` est définie, le préprocesseur ignore les lignes comprises entre ce `#ifndef` et le `#endif` associé.

***Le préprocesseur :
la directive #ifdef
(exemple)***

- Les traces sont utiles lors de la mise au point, mais peuvent être indésirables au moment d'une présentation. Pour les conserver et les exécuter seulement si nécessaire, on peut utiliser la directive `#ifdef`.

• Exemple :

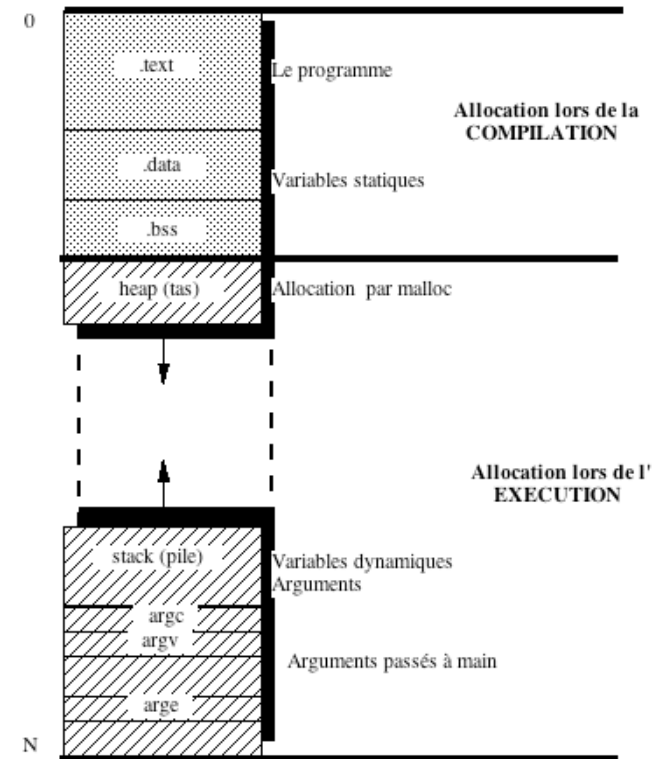
Fichier <code>essai.c</code>	Exemple d'exécutions
<pre>#include <stdio.h> int main (void){ int i; i = 4; #ifdef DEBUG printf("i=%d\n", i); #endif return 0; }</pre>	<p><u>Exécution 1:</u> \$ gcc -Wall <code>essai.c</code> -o <code>essai</code> \$ <code>essai</code></p> <p><u>Exécution 2:</u> \$ gcc -Wall -DDEBUG <code>essai.c</code> -o <code>essai</code> \$ <code>essai</code> i=4</p>

Chaîne de production d'un programme

Chaîne de production d'un programme

Espace d'adressage d'un programme

- Allocation de la mémoire pour un programme:



Visibilité (portée) et durée de vie des variables

- Variables locales (déclarées dans une fonction) :
 - par défaut : allocation sur la pile lors de l'**exécution**, la variable disparaît à la sortie de la fonction,
 - déclarée `static`, variable rémanente : allocation en zone « data » ou « bss » lors de la **compilation** (adresse fixe).
- Variables globales (déclarées en dehors de toute fonction) :
 - en C, par défaut, une variable déclarée globale dans un fichier est visible par l'ensemble des fonctions réunies au moment de l'édition de liens,
 - `static` indique que la variable n'est visible que par les fonctions qui se trouvent dans le même fichier,
- Et aussi :
 - allocation dynamique sur le « tas » (heap) avec **malloc**,
 - `volatile` : indique au compilateur qu'il ne faut pas optimiser l'allocation (registres d'entrées-sorties, ...)

Exemple : mauvaise utilisation

- Exemple de mauvaise utilisation :

```
#include <stdio.h>
int main (void){
    char *Tab[2] ;
    void fonc (char *T);
    char Message[] = "bonjour";

    Tab[0] = Message;
    fonc (Tab[1]);
    printf ("Dans main, Tab[0] : %s\n", Tab[0]);
    printf ("Dans main, Tab[1] : %s\n", Tab[1]);
    return 0;
}

void fonc (char *T) {
    char Message [] = "bonjour";
    T = Message;
    printf ("Dans fonc, T : %s\n", T);
}
```

- On obtient le résultat suivant :

```
Dans fonc, T : bonjour
Dans main, Tab[0] : bonjour
Dans main, Tab[1] : (null)
```

En effet, T est alloué sur la pile, et perdu lors de la sortie de la fonction `fonc`.

Allocation lors de la compilation Ou allocation lors de l'exécution

- Soit le programme :

```
#include <stdio.h>
int main (void){
    int i;
    void fonc (void);

    for (i = 1; i < 5; i++) {
        printf ("Appel numero %d\n", i);
        fonc();
    }
    return 0;
}

void fonc(){
    static int Compteur1 = 0;
    int Compteur2 = 0;

    Compteur1 = Compteur1 + 1 ;
    Compteur2 = Compteur2 + 1 ;
    printf("Compteur1(adr.: %p)=%d, Compteur2(adr.:%p)=%d\n",
        &Compteur1, Compteur1,&Compteur2, Compteur2);
    if (Compteur1 < 3) fonc();
}
```

- La variable `Compteur1` est allouée une fois pour toutes en mémoire (dans `.data`) au moment de la **compilation**. Elle conserve la valeur qui lui a été affectée au dernier appel à la fonction (variable rémanente).
- `Compteur1` est **réallouée sur la pile à chaque appel** à la fonction, donc réinitialisée à chaque appel.

Résultat obtenu

• Résultats :

Appel numero 1

Compteur1(adress.: 0x3080)=1, Compteur2(adress.: **0xbffffbf8**)=1
Compteur1(adress.: 0x3080)=2, Compteur2(adress.: **0xbffffb98**)=1
Compteur1(adress.: 0x3080)=3, Compteur2(adress.: **0xbffffb38**)=1

Appel numero 2

Compteur1(adress.: 0x3080)=4, Compteur2(adress.: **0xbffffbf8**)=1

Appel numero 3

Compteur1(adress.: 0x3080)=5, Compteur2(adress.: **0xbffffbf8**)=1

Appel numero 4

Compteur1(adress.: 0x3080)=6, Compteur2(adress.: **0xbffffbf8**)=1

• Commentaires :

- Premier appel : tant que Compteur1 ne vaut pas 4, la fonction se rappelle elle-même et on constate les différentes occurrences de Compteur2 sur la pile, **réallouée et réinitialisée** à des adresse (décroissantes) différentes

- Appel suivants : Compteur2 est **réallouée et réinitialisée** à la même adresse à chaque appel,

Chaîne de production d'un programme

Chaîne de production d'un programme

Langage C : *Où sont rangées les variables ?* *Récapitulatif*

Type de variable			Zone d'implantation			
			.data	.bss	.rodata	pile
Variable globale	statique (déclarée <i>static</i>)	initialisée à la déclaration	■			
		non initialisée		■		
	dynamique (non statique)	initialisée à la déclaration	■			
		non initialisée		■		
Variable locale	statique (déclarée <i>static</i>)	initialisée à la déclaration	■			
		non initialisée		■		
	dynamique (non statique)	initialisée à la déclaration				■
		non initialisée				■
	constante			■		

Chaîne de production d'un programme

L'éditeur de liens

La plupart des éditeurs de liens travaillent en deux passes :

Première passe :

- on lit tous les fichiers objets,
- on construit la table des noms et la longueur des modules
- on construit la table des symboles globaux (points d'entrées et références externes).

Deuxième passe :

- on translate les modules objets,
- on les relie en un seul module en satisfaisant les références externes.

Chaîne de production d'un programme

Fichier objet

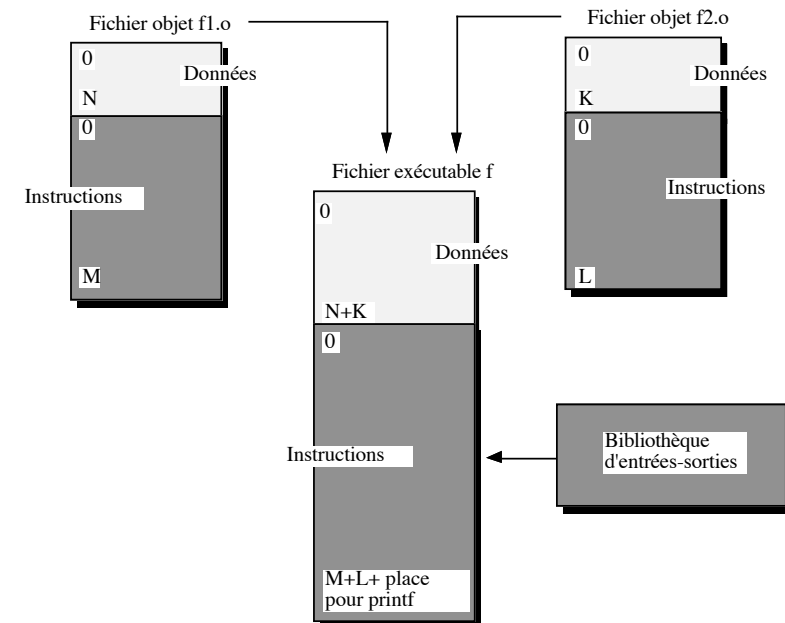
- Un fichier objet se décompose en plusieurs parties :
 - **un en-tête** (taille, emplacement en mémoire de la zone objet, le nom du fichier source, date de création...),
 - **l'espace objet proprement dit**, divisé en plusieurs sections :
 - le code binaire des instructions (section **text**)
 - les zones de données : sections **data** (données initialisées par le programme) et **bss** (données non initialisées par le programme, pouvant être mises à zéro par le système);
 - **la table des symboles** : informations sur les symboles locaux, les symboles de bibliothèque et les références non satisfaites;
 - des **informations** pour le chargement et la mise au point : numéros de lignes, description des structures de données...

Chaîne de production d'un programme

Chaîne de production d'un programme

- Des fichier objets au fichier exécutable

- On reprend l'exemple précédent. Chaque fichier objet comprend une partie données et une partie instructions.



- Nous sommes dans le cas de l'édition de liens statique : la partie nécessaire de la bibliothèque d'E/S est incorporée au fichier l'exécutable

Fichier exécutable

- Les informations concernant les symboles restent présentes dans le fichier exécutable. Pour obtenir un exécutable plus compact, on utilise la commande `strip`.

- Commande `strip`:

```
$ gcc f1.c f2.c -o prog
$ ls -l prog
-rwx----- 1 dupont  eleves  6692 Aug 30 15:33 prog*
$ file prog
prog: ELF 32-bit MSB executable, dynamically linked, not
stripped
```

```
$ strip prog
$ ls -l prog
-rwx----- 1 dupont  eleves  4188 Aug 30 15:34 prog*
$ file prog
prog: ELF 32-bit MSB executable, dynamically linked,
stripped
$
```

- On peut vérifier avec la commande `objdump` :

```
$objdump -t prog
SYMBOL TABLE:
no symbols
```

Chaîne de production d'un programme

Chaîne de production d'un programme

Edition de liens

- Reprenons cet exemple :

f1.c	f2.c
<pre>#include <stdio.h> int main (void){ float carre (float) ; int i = 0; i = carre (4); printf ("i = %d\n", i); return 0 ; }</pre>	<pre>float carre (float var){ return (var * var); }</pre>

- Extrait du contenu des tables de symboles, obtenue par les commandes :

```
nm f1.o
nm f2.o
nm f
```

Fichier f1.o	Fichier f2.o	Fichier f
U carre 00000000 T main U printf	00000000 T carre	00010274 T carre 0001023c T main 000105dc T printf

- **Remarque**

Dans les fichiers objets, les adresse sont calculées à partir de zéro.

Dans le fichier exécutable, elles sont calculées à partir de l'adresse de chargement.

Chaîne de production d'un programme

Edition de liens dynamique.

Exemple sous Unix :

```
$ cat bidon.c
#include <stdio.h>
int main (void){
    printf ("Coucou !\n");
    return 0 ;
}

$ gcc -Wall -c bidon.c
$ gcc -static bidon.o -o bidonS
$ gcc      bidon.o -o bidonD

$ ls -l bidon*
-rw-r--r--  1 dupont grpel      66 Aug 25 15:04 bidon.c
-rw-r--r--  1 dupont grpel     892 Aug 25 15:08 bidon.o
-rwxr-xr-x  1 dupont grpel    4461 Aug 25 15:06 bidonD
-rwxr-xr-x  1 dupont grpel   222331 Aug 25 15:08 bidonS
```

Ce mécanisme est généralement mis en œuvre pour les bibliothèques système (X11, e/s standards,...).

C'est ce qu'on appelle des DLL (*dynamic loadable libraries*) dans le monde Microsoft.

Chaîne de production d'un programme

Edition de lien statique et dynamique

• Sur les systèmes Unix, la commande :

gcc -o f f1.o f2.o

donnera des tables de symboles contenant ces informations :

Fichier f1.o	Fichier f2.o	Fichier f
U carre	00000000 T carre	000106d8 T carre
00000000 T main		000106a0 T main
U printf		U printf

- On remarque que printf est toujours une référence non satisfaite (`U printf`) : on se trouve dans le cas de l'édition de liens dynamique :
 - Les modules objets manquants ne sont pas ajoutés lors de l'édition de liens.
 - Les références vers ces modules sont remplacées par des pointeurs vers des fichiers **exécutables**.
 - Le fichier exécutable ainsi créé est moins volumineux, les modules nécessaires ne sont chargés que lors de l'**exécution** du programme.

***Edition de lien statique
et dynamique***

- Différence entre fichiers exécutables de type « statique » et « dynamique » :

```
$gcc -o fstatic -static f1.o f2.o  
$gcc -o fdyna f1.o f2.o
```

```
$ file fdyna fstatic  
fdyna: ELF 32-bit executable, dynamically linked  
fstatic: ELF 32-bit executable, statically linked
```

```
$ ls -l fdyna fstatic  
-rwx----- 1 dupont grpe 6764 Jun 30 fdyna  
-rwx----- 1 dupont grpe 372472 Jun 30 fstatic
```

- La taille du fichier construit en édition de liens dynamique (6764) est beaucoup plus petite.

(Les sorties des commandes `ls` et `file` ont été simplifiées)

- Inconvénients :
 - les modifications faites dans les procédures contenues dans les bibliothèques « dynamiques » peuvent perturber le fonctionnement de l'application
 - temps d'exécution (encore plus) difficile à estimer,

Chaîne de production

1. Généralités :

- compilateur, assembleur, éditeur de liens
- compilation séparée

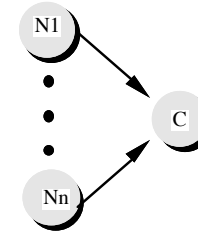
2. l'outil make

- cible, dépendance
- fichier Makefile de base
- un peu plus sur le fichier Makefile

Cible, dépendance

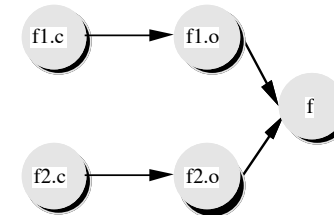
- Graphe de dépendance :

si le fichier A dépend du fichier B, il y aura un arc de B vers A.
Par exemple, si C dépend de N1, N2, ... Nn :



On dira que C est une cible qui dépend de N1, ... Nn.

- Traçons le graphe de dépendance de l'application précédente :



Ici, la cible f dépend de f1.o et f2.o, qui sont eux-mêmes des cibles dépendant de f1.c et f2.c. L'ensemble des sommets précédents f s'appelle la règle de dépendance de f.

Chaîne de production d'un programme

La commande `make` :

`make` est une commande qui permet, en particulier, de maintenir à jour un programme obtenu à partir de modules objets différents.

Pour atteindre cet objectif, `make` prend en compte :

- les dates de modification des différents modules composant l'application,
- les dépendances des différents modules.

`make` trouve les informations de dépendance dans un fichier appelé `Makefile` qui contient :

- la description des dépendances des modules (fichiers) composant l'application
- les actions à entreprendre pour remettre à jour les modules

Utilisation de `make` :

`make` va chercher un fichier `Makefile` pour y trouver le graphe de dépendance de l'application.

Sans argument, la commande `make` va chercher la première cible de ce fichier, sinon elle traite d'abord celle dont on lui a donné le nom en argument.

Chaîne de production d'un programme

Fichier Makefile

- Un fichier `Makefile` contient la représentation du graphe de dépendance d'une application sous forme de texte. La notation est la suivante :

`cible : dépendance 1 ... dépendance n`

- Il contient aussi, à la suite de chaque règle de dépendance, les actions à entreprendre pour passer des dépendances à la cible. Un fichier `Makefile` est donc une succession de lignes de dépendance et de lignes d'action :

ligne de dépendance	<code>cible : dépendance₁ dépendance_n</code>
ligne d'action	commande(s) à exécuter pour maintenir cible si une dépendance _i est modifiée

Chaîne de production d'un programme

Dans l'exemple ci-contre, la première ligne de dépendance indique que, si f1.o ou f2.o est plus récent que f il faudra faire :

```
gcc -o f f1.o f2.o
```

la seconde ligne de dépendance indique que, si f1.c est plus récent que f1.o il faudra faire :

```
gcc -c f1.c.
```

Donc, si on modifie f1.c est que l'on appelle make :

1- make refait gcc -c f1.c.(f1.c est plus récent que f1.o)

2- mais alors make refait gcc -o f f1.o f2.o.(f1.o est plus récent que f)

Attention :

- une ligne d'action commence par le caractère <TAB>
- une ligne de commentaires commence par #

Contenu des fichiers f1.c et f2.c

f1.c	f2.c
<pre>int main(void){ int i; i = carre(4); printf("i = %d\n", i); return 0; }</pre>	<pre>int carre(int var){ return(var*var) }</pre>

Chaîne de production d'un programme

Exemple de fichier Makefile

- Voici un fichier Makefile pour effectuer le travail représenté sur le schéma précédent. (<TAB> indique qu'il FAUT une tabulation, en début de ligne d'action, elle ne sera plus représentée par la suite) :

```
### f dépend de f1.o et f2.o:
```

```
f : f1.o f2.o
```

```
<TAB> gcc f1.o f2.o -o f
```

```
### f1.o dépend de f1.c :
```

```
f1.o : f1.c
```

```
gcc -c f1.c
```

```
### f2.o dépend de f2.c :
```

```
f2.o : f2.c
```

```
gcc -c f2.c
```

- Après une modification de l'un des fichiers impliqués dans le Makefile, il suffit de donner la commande :

```
make f
```

ou :

```
make
```

pour refaire l'exécutable f en ne gérant que les actions nécessaires, (make sans paramètre traite la première cible).

Makefile :
améliorations

- Voici le contenu du fichier `Makefile` pour créer `f` :

```
f : f1.o f2.o
    gcc f1.o f2.o -o f
f1.o : f1.c f2.h
    gcc -c -Wall f1.c
f2.o : f2.c
    gcc -c -Wall f2.c
```

- On a ajouté l'option `-Wall` et le fichier `f2.h` dans les dépendances de `f1.c`

***Les variables de
make***

• les VARIABLES de make :

Rôle	Nom	Exemple d'initialisation
option de compilation	CFLAGS	CFLAGS=-c -g -Wall
option d'éd. de liens	LDFLAGS	LDFLAGS= -g -lm
fichiers objets	OFILES	OFILES= f1.o f2.o
fichiers sources	CFILES	CFILES= f1.c f2.c
nom du compilateur	CC	CC= gcc
nom de l'éd. de liens	LD	LD= gcc
commande rm	RM	RM= /bin/rm
nom du programme	PROG	PROG= f

• Remarque : la cible n'est pas forcément un fichier, ici `make clean` va nettoyer le répertoire courant :

...

`clean :`

`$(RM) $(OFILES) core`

...

Chaîne de production d'un programme

Pour changer de compilateur, il suffit de changer la variable CC.

On ajoute des bibliothèques en modifiant LDFLAGS, par exemple :

```
LDFLAGS = -lm -lX11 -lsocket
```

Si on veut être sûr que LD et CC soient les mêmes, faire :

```
LD = $(CC)
```

Pour savoir ce que fait make, utiliser l'option -d, ce qui donne dans notre exemple :

```
make -d f
```

Chaîne de production d'un programme

Makefile et variables

• Nouvelle version, paramétrée, du fichier Makefile :

```
#####  
BINDIR = /usr/local/bin  
CFLAGS = -c -g -Wall  
LDFLAGS = -g -lm  
OFILES = f1.o f2.o  
CC = $(BINDIR)/gcc  
LD = $(BINDIR)/gcc  
RM = /bin/rm -f  
PROG = f  
  
#####  
f: $(OFILES)  
 $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)  
  
f1.o: f1.c f2.h  
 $(CC) $(CFLAGS) f1.c  
  
f2.o: f2.c  
 $(CC) $(CFLAGS) f2.c  
  
#####  
clean:  
 $(RM) $(OFILES) core
```

Chaîne de production d'un programme

Attention :

malgré les règles de suffixe, il faut laisser la règle :

```
f1.o : f1.c f2.h
      gcc -c -Wall f1.c
```

à cause du fichier f2.h.

La commande **makedepend** permet d'ajouter automatiquement les dépendances dues aux fichiers d'inclusion. Nous n'en parlerons pas dans ce document.

Chaîne de production d'un programme

Règle de suffixe

- Si on ne lui donne pas d'indications, make utilise ses "règles de suffixe" pour construire la cible. Par exemple, la règle de suffixe pour passer d'un fichier source à un objet est :

.c.o :

```
$(CC) $(CFLAGS) $<
```

- Les règles suivantes sont donc inutiles :

```
f1.o : f1.c
```

```
$(CC) $(CFLAGS) f1.c
```

- On peut réécrire le Makefile :

```
#####
BINDIR = /usr/local/bin
CFLAGS = -c -g -Wall
LDFLAGS = -g -lm
OFILES = f1.o f2.o
CC = $(BINDIR)/gcc
LD = $(BINDIR)/gcc
RM = /bin/rm -f
PROG = f

#####
f: $(OFILES)
  $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)

f1.o: f1.c f2.h
  $(CC) $(CFLAGS) f1.c

#####
clean:
  $(RM) $(OFILES) core
```

Chaîne de production d'un programme

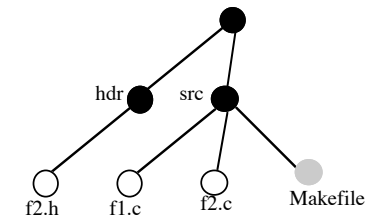
L'option `-I` indique à `gcc` où aller chercher les fichiers `#include` si ils ne sont pas trouvés dans le répertoire courant.

Chaîne de production d'un programme

Meilleure organisation de l'application

- Lorsque les fichiers constituant une application deviennent nombreux, il est souhaitable de les organiser en plusieurs répertoires. En général on sépare les fichiers sources et objets des fichiers "include".
- Ici, on adopte l'organisation suivante :

fichiers	répertoire
sources et objets correspondants	<code>src</code>
"include"	<code>hdr</code>



- la seule modification à faire est d'indiquer à `gcc` de chercher les fichier de type `.h` dans le répertoire `hdr` en utilisant l'option `-I`. On ne modifie pas les sources.

`CFLAGS= -c -g -Wall -I../hdr`

Plan

1. Généralités :

- compilateur, assembleur, éditeur de liens
- compilation séparée
- directives #include, ...
- espace d'adressage d'un programme

2. l'outil make

- cible, dépendance
- fichier Makefile de base
- un peu plus sur le fichier Makefile

3. Annexes

- Structure des fichiers objets
- Structure des fichiers exécutables

Chaîne de production d'un programme

Commentaires sur la sortie de objdump :

- VMA : Virtual Memory Address, l'adresse à l'exécution
- LMA : Load Memory Address, l'adresse de chargement, en général, on LMA = VMA
- Algn : alignment, indiqué en puissance de 2, c'est à dire $2^{**3}=8$.

Chaîne de production d'un programme

Fichier objet (1)

- Nous allons visualiser le contenu des différentes sections du fichier objet associé au fichier source suivant :

```
$ cat flg.c
int j;
int k =4;
int main (argc, argv){
    int i = 1;
    i = carre (4);
    printf ("i = %d\n", i);
    return 0 ;
}
```

- Pour ce faire, on utilise la commande objdump :

```
$ objdump -x flg.o
```

```
flg.o:      file format elf32-sparc flg.o
architecture: sparc, flags 0x00000011: HAS_RELOC, HAS_SYMS
start address 0x00000000
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.data	00000004	00000000	00000000	00000078	2**2
			CONTENTS, ALLOC, LOAD, DATA			
1	.rodata	00000008	00000000	00000000	00000080	2**3
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
2	.text	00000040	00000000	00000000	00000088	2**2
			CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE			
3	.comment	0000005c	00000000	00000000	000001b8	2**0
			CONTENTS, READONLY			

SYMBOL TABLE:

```
00000000 l  df *ABS* 00000000 flg.c
00000000 l  d  .data 00000000
00000000 l  d  .text 00000000
00000000 l  d  .rodata 00000000
00000004  O *COM* 00000004 j
00000000 g  O  .data 00000004 k
00000000  *UND* 00000000 printf
00000000  *UND* 00000000 carre
00000000 g  F  .text 00000040 main
```

Chaîne de production d'un programme

Contenu des différentes sections, `data` contient la valeur pour `j`, `rodata` (read only data) la chaîne de caractères du format, et comment les versions de logiciels utilisées.

```
$ objdump -s flg.o
```

```
flg.o:      file format elf32-sparc
```

```
Contents of section .data:
```

```
0000 00000004          ....
```

```
Contents of section .rodata:
```

```
0000 69203d20 25640a00          i = %d..
```

```
Contents of section .text:
```

```
0000 9de3bf88 82102001 c227bfec 90102004  ....'.....
0010 40000000 01000000 82100008 c227bfec  @.....'..
0020 03000000 90106000 d207bfec 40000000  .....~.....@...
0030 01000000 01000000 81c7e008 81e80000  .....
```

```
Contents of section .comment:
```

```
0000 0061733a 2053756e 20576f72 6b53686f  .as: Sun WorkSho
0010 70203620 75706461 74652032 20436f6d  p 6 update 2 Com
0020 70696c65 7220436f 6d6d6f6e 20362e32  piler Common 6.2
0030 20536f6c 61726973 5f395f43 42452032  Solaris_9_CBE 2
0040 3030312f 30342f30 320a0047 43433a20  001/04/02..GCC:
0050 28474e55 2920332e 332e3200          (GNU) 3.3.2.
```

```
$objdump -d flg.o
```

```
Disassembly of section .text:
```

```
00000000 <main>:
0: 9d e3 bf 88      save %sp, -120, %sp
4: 82 10 20 01      mov 1, %g1
8: c2 27 bf ec      st %g1, [ %fp + -20 ]
c: 90 10 20 04      mov 4, %o0
10: 40 00 00 00      call 10 <main+0x10>
14: 01 00 00 00      nop
18: 82 10 00 08      mov %o0, %g1
1c: c2 27 bf ec      st %g1, [ %fp + -20 ]
20: 03 00 00 00      sethi %hi(0), %g1
24: 90 10 60 00      mov %g1, %o0 ! 0 <main>
28: d2 07 bf ec      ld [ %fp + -20 ], %o1
2c: 40 00 00 00      call 2c <main+0x2c>
30: 01 00 00 00      nop
34: 01 00 00 00      nop
38: 81 c7 e0 08      ret
3c: 81 e8 00 00      restore
```

Chaîne de production d'un programme

Fichier objet (2)

• Commentaires sur les sections et la table des symboles:

```
Sections:
```

Idx Name	Size	VMA	LMA	File off	Algn
0 .data	00000004	00000000	00000000	00000078	2**2
		CONTENTS, ALLOC, LOAD, DATA			
1 .rodata	00000008	00000000	00000000	00000080	2**3
		CONTENTS, ALLOC, LOAD, READONLY, DATA			
2 .text	00000040	00000000	00000000	00000088	2**2
		CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE			
3 .comment	0000005c	00000000	00000000	000001b8	2**0
		CONTENTS, READONLY			

```
SYMBOL TABLE:
```

00000000	l	df	*ABS*	00000000	flg.c
00000000	l	d	.data	00000000	
00000000	l	d	.text	00000000	
00000000	l	d	.rodata	00000000	
00000004	O	*COM*	00000004	j	
00000000	g	O	.data	00000004	k
00000000		*UND*	00000000	printf	
00000000		*UND*	00000000	carre	
00000000	g	F	.text	00000040	main

- `j`, variable globale non initialisée est rangée dans la zone `COMMON` qui sera intégrée à `bss` à l'édition de liens,
- `k`, variable globale initialisée est dans `data`,
- `carre` et `printf` sont indéfinies, on attend l'édition de liens pour satisfaire ces références.
- `i` est allouée sur la pile,
- le symbole `main` est bien dans la section de programme, la section `text`.
- le symbole `rodata` contient la chaîne de caractères constante du format de `printf`.

Chaîne de production d'un programme

Dans l'exemple suivant on montre l'utilisation de variables qualifiées de static

Compteur1 est allouée une fois pour toute en mémoire, donc elle n'est pas perdue à chaque sortie de la fonction. Elle conserve la valeur qui lui a été affectée au dernier appel à la fonction (variables rémanentes).

Compteur2 est **réallouée sur la pile à chaque appel** à la fonction, donc réinitialisée à chaque appel.

```
$ cat compteur.c
#include <stdio.h>

int main (){
    int i;
    void fonc (void);
    for (i = 0; i < 3; i++) fonc();
    return 0;
}

void fonc(){
    static int Compteur1 = 0;
    int Compteur2 = 0;
    Compteur1 = Compteur1 + 1 ;
    Compteur2 = Compteur2 + 1 ;
    printf (" Appel numero %d (Compteur1), appel numero %d
(Compteur2)\n", Compteur1, Compteur2);
}

$ gcc -Wall compteur.c

$ ./a.out
Appel numero 1 (Compteur1), appel numero 1 (Compteur2)
Appel numero 2 (Compteur1), appel numero 1 (Compteur2)
Appel numero 3 (Compteur1), appel numero 1 (Compteur2)
```

Chaîne de production d'un programme

Fichier objet (3)

- Modification du fichier source précédent. On ajoute des variables de type static.

```
$ cat flgs.c
int j;
int k =4;
int main (void){
int i = 1;
static int j2;
static int k2 = 8;
i = carre (4);
printf ("i = %d\n", i);
}
```

```
$ objdump -x flgs.o
```

```
flgs.o:      file format elf32-sparc flgs.o
architecture: sparc, flags 0x00000011: HAS_RELOC, HAS_SYMS
start address 0x00000000
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.data	00000008	00000000	00000000	0000007c	2**2
			CONTENTS, ALLOC, LOAD, DATA			
1	.rodata	00000008	00000000	00000000	00000088	2**3
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
2	.text	00000040	00000000	00000000	00000090	2**2
			CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE			
3	.bss	00000004	00000000	00000000	000000d0	2**2
			ALLOC			
4	.comment	0000005c	00000000	00000000	000001f8	2**0
			CONTENTS, READONLY			

SYMBOL TABLE:

00000000	l	df	*ABS*	00000000	flgs.c
00000000	l	d	.data	00000000	
00000000	l	d	.bss	00000000	
00000004	l	O	.data	00000004	k2.1
00000000	l	d	.text	00000000	
00000000	l	O	.bss	00000004	j2.0
00000000	l	d	.rodata	00000000	
00000004	O	*COM*	00000004	j	
00000000	g	O	.data	00000004	k
00000000		*UND*	00000000	printf	
00000000		*UND*	00000000	carre	
00000000	g	F	.text	00000040	main

Chaîne de production d'un programme

Commentaires sur la commande nm :

Mnémonique	Signification
B	bss, section des variables non initialisées.
D	data, section des variables initialisées par l'utilisateur
T	Text, section de programme
U	Référence non satisfaite
C	common. Variables non initialisées, implantées dans bss à l'édition de liens.

Variable locale : mnémonique en minuscule

Variable globale : mnémonique en majuscule

Chaîne de production d'un programme

Fichier objet (4)

• Commentaires sur les symboles :

```
Sections:
Idx Name      Size      VMA      LMA      File off  Algn
  0 .data      00000008 00000000 00000000 0000007c 2**2
CONTENTS, ALLOC, LOAD, DATA
  1 .rodata    00000008 00000000 00000000 00000088 2**3
CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .text      00000040 00000000 00000000 00000090 2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  3 .bss       00000004 00000000 00000000 000000d0 2**2
ALLOC
  4 .comment   0000005c 00000000 00000000 000001f8 2**0
CONTENTS, READONLY
```

```
SYMBOL TABLE:
00000000 l df *ABS* 00000000 flgs.c
00000000 l d .data 00000000
00000000 l d .bss 00000000
00000004 l o .data 00000004 k2.1
00000000 l d .text 00000000
00000000 l o .bss 00000004 j2.0
00000000 l d .rodata 00000000
00000004 o *COM* 00000004 j
00000000 g o .data 00000004 k
00000000 *UND* 00000000 printf
00000000 *UND* 00000000 carre
00000000 g F .text 00000040 main
```

- j2, variable static non initialisée est rangée dans bss, sur 4 octets
- k2, variable static initialisée est rangée dans data, sur 4 octets

• On peut aussi utiliser la commande nm, qui donne moins de détails :

```
$ nm flgs.o
U carre
00000004 C j
00000000 b j2.0
00000000 D k
00000004 d k2.1
00000000 T main
U printf
```

Assembleur

- Pour une compréhension plus détaillée des mécanismes précédents, nous allons étudier un exemple en assembleur

Chaîne de production d'un programme

La plupart des assembleurs travaillent en deux passes :

- Première passe :
 - on construit la table des symboles dans laquelle il y a une entrée pour chaque variable ou étiquette rencontrée dans le programme. Tant que les adresses mémoire ne sont pas associées aux variables et étiquettes, celles-ci sont représentées par leur entrée dans cette table des symboles. Ici, Fin sera représentée par 2 tant qu'on ne lui a pas affecté d'adresse.
 - on génère du programme dans lequel les symboles (étiquettes et variables) ont pour adresse leur entrée dans la table des symboles.
 - on remplit la table des symboles : ici on implante les variables après le programme, donc on ne peut associer une adresse à Som qu'après avoir parcouru tout le programme. Par contre on peut tout de suite affecter l'adresse de Boucle puisqu'on a rencontré l'instruction repérée par Boucle avant la référence à Boucle. Dans la cas d'une référence en avant on ne peut pas affecter immédiatement une valeur à l'étiquette.
- Deuxième passe :
 - on relit le programme généré en associant à chaque symbole l'adresse qu'on lui a affectée et qui a été rangée dans la table des symboles

Gestion de la table des symboles

On représente l'évolution de la table des symboles lors de la première passe pour le programme ci-contre.

L'étiquette Boucle est rencontrée :

Table des symboles

1	Boucle	0008
---	--------	------

L'étiquette Fin est rencontrée :

Table des symboles

1	Boucle	0008
2	Fin	0000

Le nom de variable Som est rencontré :

Table des symboles

1	Boucle	0008
2	Fin	0000
3	Som	0000

A la fin de la première passe, les entrées de la table des symboles affectées à Fin et Som peuvent être remplies :

Table des symboles

1	Boucle	0008
2	Fin	0018
3	Som	001C

Chaîne de production d'un programme

Assembleur : première passe(1)

- Exemple de source assembleur :

```
/* Somme des 128 premiers entiers */
store    128,Reg2 ; ranger valeur 128 dans Reg2
store    0,Reg1  ; valeur 0 -> Reg1
Boucle :dec    Reg2 ; décrémenter Reg2
        beq    Fin ; si Reg2 est nul aller à Fin
        add    Reg2,Reg1 ; Reg1 = Reg1 + Reg2
        jump   Boucle ; aller à Boucle
Fin :     store    Reg1,Som ;

Som      long    1 ;
```

- Format des instructions :

- elles sont toutes codées sur 4 octets,
- ce codage est divisé en 3 champs : code opération sur 8 bits et deux opérandes sur 12 bits chacun,

Assembleur :première passe(2)

• Voici le programme généré (représenté en base 16) à la fin de la première passe, il reste des références non satisfaites remplacées par leur entrée dans la table des symboles. Ces références sont en caractères **gras**.

Adresse	Code opération	Opérande 1	Opérande 2
00	01 (store)	080 (128 en base 16)	002 (Reg2)
04	01 (store)	000 (0)	001 (Reg1)
08	03 (dec)	002 (Reg2)	0FF (rien)
0C	06 (beq)	002(Fin ?)	0FF (rien)
10	0D (add)	002 (Reg2)	001 (Reg1)
14	20 (jump)	008 (Boucle)	0FF (rien)
18	11 (store)	001 (Reg1)	003 (Som ?)

Etat de la table des symboles à la fin de la première passe :

Table des symboles

1	Boucle	0008
2	Fin	0018
3	Som	001C

Chaîne de production d'un programme

On remarquera que rien ne permet de dire si cette zone de mémoire :

```
00 01040002
04 01000001
08 030020FF
0C 060180FF
10 0D002001
14 200080FF
18 1100101C
```

contient un programme (ce qui est le cas) ou des données.

Nous sommes dans une architecture de Von Neumann, rien ne permet de distinguer en mémoire les instructions des données, devient zone de programme, la zone mémoire sur laquelle pointe le compteur ordinal.

Chaîne de production d'un programme

Assembleur deuxième passe

- On remplace dans le programme les références non satisfaites. Les variables qui étaient représentées par leur entrées dans la table des symboles sont maintenant représentées par leur adresse.

Adresse	Code opération	Opérande 1	Opérande 2
00	01 (store)	080 (128 en base 16)	002 (Reg2)
04	01 (store)	000 (0)	001 (Reg1)
08	03 (dec)	002 (Reg2)	0FF (rien)
0C	06 (beq)	018 (Fin)	0FF (rien)
10	0D (add)	002 (Reg2)	001 (Reg1)
14	20 (jump)	008 (Boucle)	0FF (rien)
18	11 (store)	001 (Reg1)	01C (Som)

- On rappelle l'état de la table des symboles à la fin de la première passe :

Table des symboles

1	Boucle	0008
2	Fin	0018
3	Som	001C