

Compilation et représentation binaire

M1 - CHPS

Architecture Interne des Systèmes d'exploitations (AISE)

Jean-Baptiste Besnard
<jean-baptiste.besnard@paratools.com>



Julien Adam
<julien.adam@paratools.com>

Programme du Semestre

- ◆ 1 - Généralités sur les OS et Utilisation de base
- ◆ 2 - Processus, Threads & Synchronisation
- ◆ **3 - Compilation et représentation Binaire**
- ◆ 4 - Architecture Mémoire d'un processus
- ◆ 5 - Programmation réseau et entrées/sorties avancées
- ◆ 6 - Virtualisation et Conteneurs
- ◆ 7 - Noyau Linux et bases d'ordonnancement
- ◆ Examen + Démo de projets

Construction d'un programme

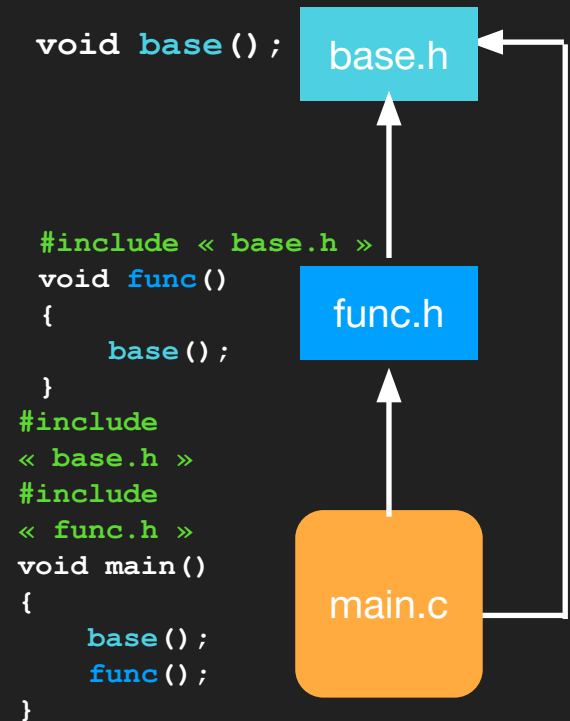
- Unité de compilation : un fichier source (parfois nommé TU pour « Translation Unit »)
- Préprocesseur : préparation d'une TU pour la compilation.
- Compilateur : exécution d'une TU, transformation en un set d'instructions spécifiques à l'architecture.
- Assemblage: Transformation des instructions assembleur en code machine (structure ELF)
- L'édition de liens: Assemblage des différentes TU pour créer un exécutable

Derrière GCC

- GCC signifie “GNU Compiler Collection”, ce n’est donc pas un programme, mais une collection d’outils, un pour chaque phase de la construction d’un programme:
- preprocessing: `cpp file.c > file.i`
- compilation: `cc1 file.i -o file.s`
- Assemblage: `as file.s -o file.o`
- Link: `collect2 file.o -o program`
- `collect2` est un wrapper GNU de `link`, qui repose sur `LD`, qui n’est pas un outil de la collection GNU. `LD` est distribué via les “binutils” et est commun à la majorité des compilateurs (intel, PGI, LLVM...)

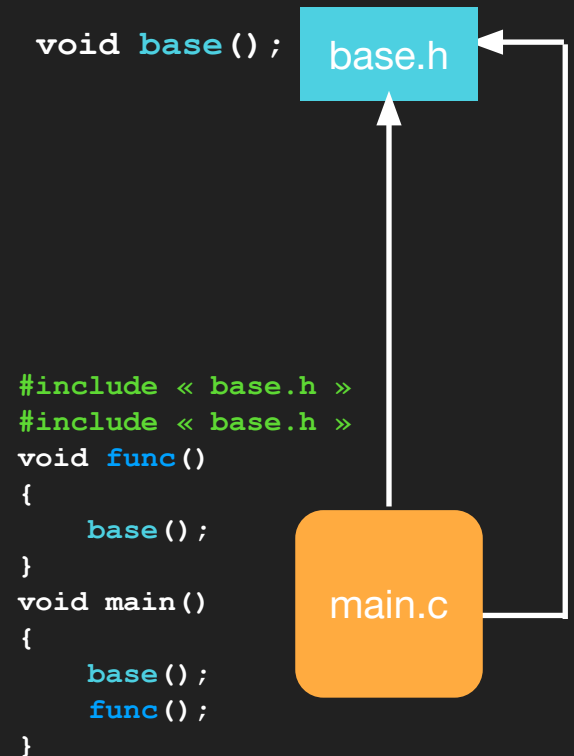
Preprocessing

- Interprétation de directives (#)
 - **#define** / **#undef** : Définition, déclaration de macro (fonctions, constantes...)
 - **#if(n)def** / **#else** / **#endif** : Compilation conditionnelle de sections de code
 - **#include** : Inclusion de fichiers récursives, nécessité des guards
 - **#error** / **#warning** / **#todo** : Influence la sortie de compilation
 - **#pragma...** : gestion compilateur
- Certaines constantes existent (**__FILE__**, **__LINE__**, **__DATE__**, **__TIME__**), et extensions selon l'architecture (**__WIN32**, **__APPLE**, **__linux__**)
- Programme : **cpp main.c main.i**, Résultat obtenu avec : **gcc -E main.c**



Preprocessing

- Interprétation de directives (#)
 - **#define** / **#undef** : Définition, déclaration de macro (fonctions, constantes...)
 - **#if(n)def** / **#else** / **#endif** : Compilation conditionnelle de sections de code
 - **#include** : Inclusion de fichiers récursives, nécessité des guards
 - **#error** / **#warning** / **#todo** : Influence la sortie de compilation
 - **#pragma...** :
- Certaines constantes existent (**__FILE__**, **__LINE__**, **__DATE__**, **__TIME__**), et extensions selon l'architecture (**__WIN32**, **__APPLE**, **__linux__**)
- Programme : **cpp main.c main.i**, Résultat obtenu avec : **gcc -E main.c**



Preprocessing

- Interprétation de directives (#)
 - **#define** / **#undef** : Définition, déclaration de macro (fonctions, constantes...)
 - **#if(n)def** / **#else** / **#endif** : Compilation conditionnelle de sections de code
 - **#include** : Inclusion de fichiers récursives, nécessité des guards
 - **#error** / **#warning** / **#todo** : Influence la sortie de compilation
 - **#pragma...** :
- Certaines constantes existent (**__FILE__**, **__LINE__**, **__DATE__**, **__TIME__**), et extensions selon l'architecture (**__WIN32**, **__APPLE**, **__linux__**)
- Programme : **cpp main.c main.i**, Résultat obtenu avec : **gcc -E main.c**

```
void base();  
void base();  
void func()  
{  
    base();  
}  
void main()  
{  
    base();  
    func();  
}
```

main.c

Preprocessing

- Invocation : `cpp main.c main.i`
- Résultat: `gcc -E main.c [-P]`
- Ajout de chemins :
 - `-I/usr/include`
 - `export C_INCLUDE_PATH`
- `-D` / `-undef`
- `-include`

```
# 1 "main.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 361 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.c" 2
# 1 "./base.h" 1
void base();
# 2 "main.c" 2
# 1 "./func.h" 1
void func()
{
    base();
}
# 3 "main.c" 2
int main(int argc, char const *argv[])
{
    func();
    return 0;
}
```

```
➤ gcc -E main.c -o main.i
main.c:1:10: erreur fatale: base.h : Aucun fichier ou dossier de ce type
#include <base.h>
      ^~~~~~
compilation terminée.
```


Compilation

- Objectif : Transformer un contenu d'un langage **source** vers un langage **destination** (*=target*)
- Un programme source doit suivre un ensemble de règles afin d'être compris par le compilateur : la **grammaire**
- Le processus de compilation se découpe en trois grosses phases principaux (simplifié)



Front-end

- Réalise l'analyse grammaticale du langage source pour produire une représentation intermédiaire (IR)

1. Analyse Lexicale : lecture de la source un caractère après l'autre pour former des mots (=lexème). Toute information superflue est ignorée (espaces...)
2. Analyse Syntaxique : Chacun de ces lexèmes est soumis à validation pour s'assurer qu'il font parti du langage
3. Analyse Sémantique : un ensemble de lexème forme une phrase, qui doit être sémantiquement juste


Token	Example lexeme
const	const
if	if
relop	<, <=
id	pi, count, age
num	3.14, 0
literal	"hello world"

- Tout un pan de l'informatique moderne s'intéresse au formalisme du langage (pour créer son propre langage : lex & yacc)

Middle-end

- Une fois le code généré, on obtient un programme sémantiquement juste mais loin d'être optimisé. De nombreuses passes sont en jeu ici
 - Graphe de control-flow, inlining
 - Élimination de code « mort » (DCE)
 - Transformation de boucles
 - Propagation de constantes
- Ce composant est indépendant de tout langage et de toute architecture. Réutilisation infinie, tant que la grammaire fourni la même sémantique (représentation intermédiaire)

```
int foo(void)
{
    int a = 24;
    int b = 25;
    int c;
    c = a * 4;
    return c;
    b = 24;
    return 0;
}
```



Back-end

- Génération du programme pour la machine cible. Chaque architecture ayant un jeu d'instructions différent
- le code généré possède ses propres optimisations (=machine-dependent Optimisations), Vectorisation (SSE, AVX...)
- Registres, Pipelining, mode d'adressage (Absolute, PC-centric, register-*...), code redondant
- Génération des fichiers contenant le code assembleur (.s)
- Résultat de **gcc -S main.c**

```
.section      __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 14
.globl _func
.p2align     4, 0x90

_func:
.cfi_startproc
## %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq     %rsp, %rbp
.cfi_def_cfa_register %rbp
movb     $0, %al
callq    _base
popq     %rbp
retq
.cfi_endproc

.globl _main
.p2align     4, 0x90

_main:
.cfi_startproc
## %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq     %rsp, %rbp
.cfi_def_cfa_register %rbp
subq     $16, %rsp
movl     $0, -4(%rbp)
movl     %edi, -8(%rbp)
movq     %rsi, -16(%rbp)
callq    _func
xorl     %eax, %eax
addq     $16, %rsp
popq     %rbp
retq
.cfi_endproc
```

Assemblage

Source :

<https://en.wikipedia.org/wiki/Endianness>

- Transformation du code dépendant machine en code binaire
- Prise en compte de « l'Endianness » (Little / big)
- Création d'un fichier objet (.o) suivant le format ELF

■ **.text** : code défini dans le fichier

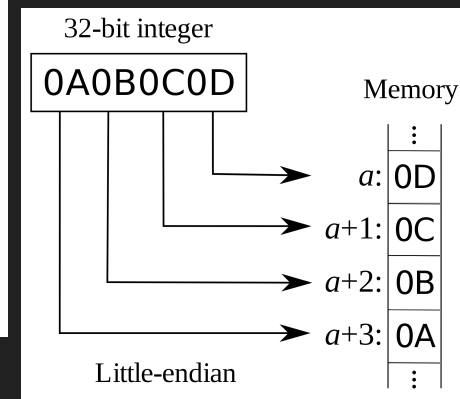
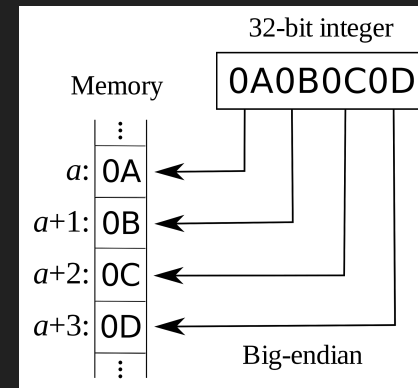
■ **.data / .bss** : variables globales du fichier initialisées / ou non

■ **.rodata** : Constantes

■ **.shstrtab** : tableau des chaînes de caractères

○ Commande : **as main.s -o main.s**

○ Résultat : **gcc -c main.c -o main.o**



```
readelf -SW main.o
Il y a 9 en-têtes de section, débutant à l'adresse de décalage 0x180:

En-têtes de section :
[Nr] Nom                               Type                               Adr                               Décala.Taille ES Fan LN Inf Al
[ 0]                               NULL                               0000000000000000 000000 000000 00  0  0  0
[ 1] .text                             PROGBITS                           0000000000000000 000040 000042 00 AX  0  0  1
[ 2] .data                             PROGBITS                           0000000000000000 000082 000000 00 WA  0  0  1
[ 3] .bss                              NOBITS                             0000000000000000 000082 000000 00 WA  0  0  1
[ 4] .rodata                           PROGBITS                           0000000000000000 000082 00000d 00 A  0  0  1
[ 5] .comment                          PROGBITS                           0000000000000000 00008f 00002d 01 MS  0  0  1
[ 6] .note.GNU-stack                   PROGBITS                           0000000000000000 0000bc 000000 00  0  0  1
[ 7] .eh_frame                        PROGBITS                           0000000000000000 0000c0 000078 00 A  0  0  8
[ 8] .shstrtab                         STRTAB                             0000000000000000 000138 000047 00  0  0  1

Clé des fanions :
W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe),
T (TLS), C (comprimé), x (inconnu), o (spécifique à l'OS), E (exclu),
l (grand), p (processor specific)
```

Édition de liens

- Addition de plusieurs fichiers objets pour créer un exécutable
- Fonction du « linker » : **ld** / **ld.gold** (version GNU)
- Fusion des sections identiques
- « Relocations » de symboles = réarrangement de l'espace d'adressage
- Résultat : **gcc main.c**

```
↳ readelf -h ./a.out
En-tête ELF:
  Magique:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:                                ELF64
  Données:                                complément à 2, système à octets de poids faible d'abord (little endian)
  Version:                                1 (current)
  OS/ABI:                                UNIX - System V
  Version ABI:                            0
  Type:                                EXEC (fichier exécutable)
  Machine:                                Advanced Micro Devices X86-64
  Version:                                0x1
  Adresse du point d'entrée:              0x4003b0
  Début des en-têtes de programme :      64 (octets dans le fichier)
  Début des en-têtes de section :        6360 (octets dans le fichier)
  Fanions:                                0x0
  Taille de cet en-tête:                  64 (octets)
  Taille de l'en-tête du programme:      56 (octets)
  Nombre d'en-tête du programme:          9
  Taille des en-têtes de section:         64 (octets)
  Nombre d'en-têtes de section:           27
  Table d'index des chaînes d'en-tête de section: 26
```

```
Betelgeuse ~
↳ objdump -d ./a.out | grep "<_start>"
00000000004003b0 <_start>:
Betelgeuse ~
↳ objdump -d ./a.out | grep "<main>"
00000000004004ae <main>:
```

Édition de liens

Artefacts compilo-spécifiques:

- `crt1.o` / `crt0.o...` : Symboles de chargement du programme qui contient surtout le point d'entrée (fonction `_start()`) pour le déchargement du Shell.
- `crti.o` / `crti.o`: Symboles `_init` et `_fini`, constructeur et destructeur (old-style). Ils sont conservés par rétrocompatibilité avec les anciens systèmes. Remplacé par les segments des sections `.init_array` et `.fini_array`.
- `crtbegin.o` / `crtend.o`: **C++** constructeurs
- `crtbeginS.o` / `crtendS.o` : remplace son équivalent quand `-fPIC`
- `crtbeginT.o` / `crtendT.o`: remplace son équivalent quand `-static`

Bibliothèque / Module

- Une application contient rarement tout le code dont elle a besoin et repose sur l'inclusion de modules déjà implémentés : réutilisation de code
- Une déclaration de la partie publique du module. C'est le header inclus, exposant variable & fonctions (ex: /usr/include)
 - Inclusion avec `-I`, `C_INCLUDE_PATH`, `-include...`
 - Invocation avec : `#include <mymodule.h>`
- Les code du module précompilé, qui est chargé lors de l'édition de liens pour l'optimisation du binaire final (exemple : /usr/lib[64])
 - Inclusion avec `-L`, `[LD_]LIBRARY_PATH`
 - Invocation avec: `-l<nom du module>` (ex: `-lgcc` pour **libgcc.a**)
 - Pas nécessaire pour les fonctions comme `printf/scanf`, pourquoi ?

```
└─ gcc main.c -I./include -L./lib -lmylib
```

```
└─ tree -L 1 /usr/include
/usr/include
├── aio.h
├── aliases.h
├── alloca.h
├── a.out.h
├── argp.h
├── argz.h
├── ar.h
├── arpa
├── asm
├── asm-generic
├── assert.h
├── bits
├── byteswap.h
├── bzlib.h
├── c++
├── complex.h
├── cpio.h
├── crypt.h
├── ctype.h
├── cursesapp.h
├── cursesf.h
├── ── ls /usr/lib64/*.so -l
├── /usr/lib64/BugpointPasses.so
├── /usr/lib64/eppic_makedumpfile.so
├── /usr/lib64/ld-2.27.so
├── /usr/lib64/libanl-2.27.so
├── /usr/lib64/libanl.so
├── /usr/lib64/libasm-0.174.so
├── /usr/lib64/libbfd-2.29.1-23.fc28.so
├── /usr/lib64/libBrokenLocale-2.27.so
├── /usr/lib64/libBrokenLocale.so
├── /usr/lib64/libbtparse.so
├── /usr/lib64/libbz2.so
├── /usr/lib64/libc-2.27.so
├── /usr/lib64/libccl.so
├── /usr/lib64/libclangAnalysis.so
├── /usr/lib64/libclangApplyReplacements.
├── /usr/lib64/libclangARCMigrate.so
├── /usr/lib64/libclangASTMatchers.so
```


Bibliothèque / Module

- **STATIQUE (extension .a)**

- Le module est lié & injecté à l'application (archive de fichiers .o)
- Avantage : Indépendant de l'exécution
- Inconvénients : Binaire + lourd, fonction externes référencées « en dur » (pas de `dlopen()`)
- Outil : `ar` (-x : extract, -s : create, -t : list) Souvent : `ar rcs libmodule.a module.o`

- **DYNAMIQUE (extension .so)**

- Le module est référencé à la compilation et injecté à **l'exécution**
- Avantage : Binaire plus léger, rien n'est en dur dans le binaire, plus de souplesse à l'exécution
- Inconvénient : crée un overhead au runtime, dépendance entre environnement de compilation & d'exécution
- Outil : `ld`
- Via compilateur : `gcc -shared module.o -o libmodule.so`
- -fPIC indispensable dans 90% des cas de bibliothèques dynamiques (*Position Independent Code*)

Bibliothèque / Module

```
Betelgeuse ~  
➤ gcc -static main.c -I.  
/usr/bin/ld : ne peut trouver -lc  
collect2: error: ld a retourné le statut de sortie 1
```

- Par défaut, il n'y a pas de distinctions entre statique et dynamique à l'édition de liens. Possibilité de forcer un link statique : `gcc -static` (génère une erreur si la version statique n'existe pas)
- Recherche de `.so` à la compilation : `-L` / `-Wl,-rpath`
- Chargement au runtime: `LD_LIBRARY_PATH` / `LD_PRELOAD`

```
Betelgeuse ~  
➤ ldd ./a.out  
linux-vdso.so.1 (0x00007ffffdded6000)  
libc.so.6 => /lib64/libc.so.6 (0x00007ffff6e0cf2000)  
/lib64/ld-linux-x86-64.so.2 (0x00007ffff6e10b1000)
```

```
➤ gcc main.c -I. -Wl,-rpath=/lib64/; \  
> readelf -dW ./a.out
```

Section dynamique à l'offset 0xe50 contient 21 entrées :

Étiquettes	Type	Nom/Valeur
0x0000000000000001	(NEEDED)	Bibliothèque partagée: [libc.so.6]
0x000000000000000f	(RPATH)	Bibliothèque rpath: [/lib64/]
0x000000000000000c	(INIT)	0x400398
0x000000000000000d	(FINI)	0x400394

```
➤ readelf -dW ./a.out
```

Section dynamique à l'offset 0xe60 contient 20 entrées :

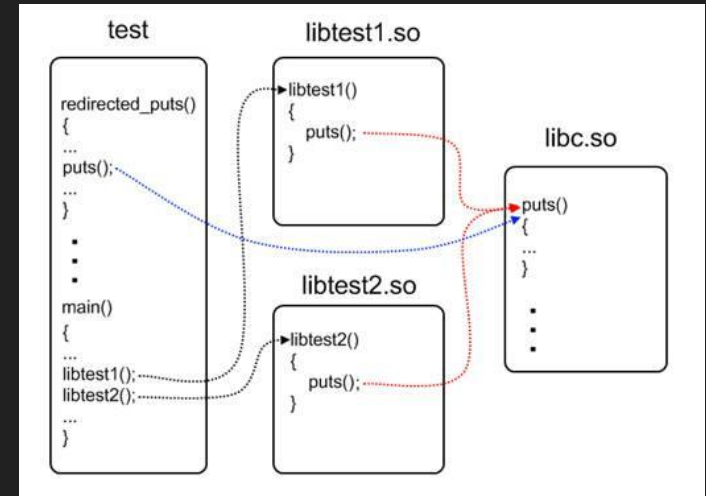
Étiquettes	Type	Nom/Valeur
0x0000000000000001	(NEEDED)	Bibliothèque partagée: [libc.so.6]
0x000000000000000c	(INIT)	0x400390
0x000000000000000d	(FINI)	0x400394
0x0000000000000019	(INIT_ARRAY)	0x600e50
0x000000000000001b	(INIT_ARRAYSZ)	8 (octets)
0x000000000000001a	(FINI_ARRAY)	0x600e58

Bibliothèque / Module

- Chargement dynamique via **libdl.so**
 - **h = dlopen(« mylib.so »)** : Charge une bibliothèque (appel du loader, chargement mémoire, etc...)
 - **dlsym(h, « i »)** : Renvoie l'adresse d'un symbole chargé en mémoire (variable, fonction, etc...)
 - **dlclose(h)** : Ferme la bibliothèque, déchargement...
- Requiert une bibliothèque dynamique !

Bibliothèque / Module

- L'introspection est l'art de charger une bibliothèque à l'exécution, pour venir « écraser » les symboles existants par ceux re-définis.
- Exemple : `LD_PRELOAD=myalloclib.so ./a.out`
- Conserver la cohérence de l'application : rappeler la fonction originale via `dlsym(« func », RTLD_NEXT)` ;
- Le prochain symbole est déterminé par l'ordre des bibliothèques tel qu'indiqué à la compilation



```
$ ldd ./IMB-MPI1
linux-vdso.so.1 (0x00007fff493b1000)
libmpc_framework.so => $INSTALL_PATH//x86_64/x86_64//lib/libmpc_framework.so (0x00007f74b2717000)
libextls.so.0 => $INSTALL_PATH//x86_64/x86_64//lib/libextls.so.0 (0x00007f74b250d000)
libportals.so.4 => /opt/sources/portals4/INSTALL/lib/libportals.so.4 (0x00007f74b22e7000)
libm.so.6 => /lib64/libm.so.6 (0x00007f74b1f53000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f74b1d34000)
librt.so.1 => /lib64/librt.so.1 (0x00007f74b1b2c000)
libsctk_arch.so => $INSTALL_PATH//x86_64/x86_64//lib/libsctk_arch.so (0x00007f74b1929000)
libhwloc.so.5 => $INSTALL_PATH//x86_64/x86_64//lib/libhwloc.so.5 (0x00007f74b16f0000)
libxml2.so.2 => $INSTALL_PATH//x86_64/x86_64//lib/libxml2.so.2 (0x00007f74b138e000)
libmpcgetopt.so.0 => $INSTALL_PATH//x86_64/x86_64//lib/libmpcgetopt.so.0 (0x00007f74b118a000)
libc.so.6 => /lib64/libc.so.6 (0x00007f74b0dcb000)
/lib64/ld-linux-x86-64.so.2 (0x00007f74b2f3a000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f74b0bc7000)
libev.so.4 => /lib64/libev.so.4 (0x00007f74b09b8000)
liblzma.so.5 => /lib64/liblzma.so.5 (0x00007f74b0791000)
```

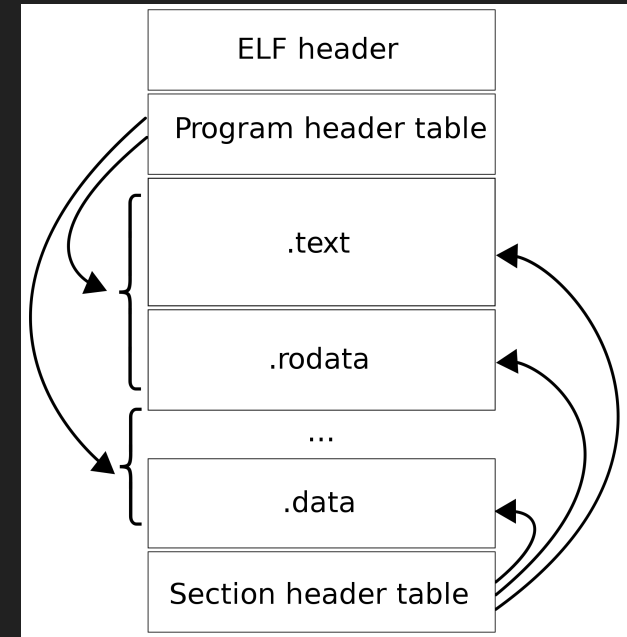
En résumé



- Preprocessing : `gcc -E main.c (cpp)`
 - Ajout de règles : `INCLUDE_PATH= / -I / -include / -D / -undef`
- Compilation : `gcc -S main.c`
- Code objet : `gcc -c main.c (as)`
- Edition de liens : `gcc main.c (ld)`
- Ajout de librairies : `-L<chemin> / -l<chemin> / <chemin absolu>`
 - Statique (.a) : `ar rcs / -static / LIBRARY_PATH`
 - Dynamique (.so) : `-fPIC -shared / -Wl,-rpath / LD_LIBRARY_PATH`

Format ELF

- ELF = *Executable and Linkable Format*
- Décrit comment un binaire doit être représenté pour être compris par le lanceur de processus (`ld-linux.so`)
- Un programme contient beaucoup d'informations. Pour rester cohérent, il est segmenté en plusieurs sections
- Séquence magique : « **7F 45 4C 46** » = 7F « ELF »
- L'entête du ELF contient toutes les informations nécessaires à l'architecture (32/64 bits, endianness, ABI, type de fichier, jeu d'instruction...)



```
En-tête ELF:
Magique: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Classe: ELF64
Données: complément à 2, système à octets
Version: 1 (current)
OS/ABI: UNIX - System V
Version ABI: 0
Type: EXEC (fichier exécutable)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Adresse du point d'entrée: 0x4003b0
Début des en-têtes de programme : 64 (octets dans le fichier)
Début des en-têtes de section : 6360 (octets dans le fichier)
Fanions: 0x0
Taille de cet en-tête: 64 (octets)
Taille de l'en-tête du programme: 56 (octets)
Nombre d'en-tête du programme: 9
Taille des en-têtes de section: 64 (octets)
Nombre d'en-têtes de section: 27
Table d'index des chaînes d'en-tête de section: 26
```

Format ELF

- **Program header** : Stocke les informations nécessaires à la création de l'image du processus. Structure le programme d'un point de vue mémoire
- **Section header** : Regroupe les informations nécessaires au bon fonctionnement du programme. Structure le programme d'un point de vue fonctionnel
- Le reste du ELF est composé de blocs d'instructions, indexées dans l'une et/ou l'autre de ces tables

En-têtes de section :

[Nr]	Nom	Type	Adr	Décala.	Taille	ES	Fan	LN	Inf	Al
[0]		NULL	0000000000000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	0000000000400254	000254	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	0000000000400298	000298	00001c	00	A	5	0	8
[5]	.dynsym	DYNSYM	00000000004002b8	0002b8	000048	18	A	6	1	8
[6]	.dynstr	STRTAB	0000000000400300	000300	000040	00	A	0	0	1
[7]	.gnu.version	VERSYM	0000000000400340	000340	000006	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	0000000000400348	000348	000020	00	A	6	1	8
[9]	.rela.dyn	RELA	0000000000400368	000368	000030	18	A	5	0	8
[10]	.init	PROGBITS	0000000000400398	000398	000017	00	AX	0	0	4
[11]	.text	PROGBITS	00000000004003b0	0003b0	000181	00	AX	0	0	16
[12]	.fini	PROGBITS	0000000000400534	000534	000009	00	AX	0	0	4
[13]	.rodata	PROGBITS	0000000000400540	000540	000010	00	A	0	0	8
[14]	.eh_frame_hdr	PROGBITS	0000000000400550	000550	000044	00	A	0	0	4
[15]	.eh_frame	PROGBITS	0000000000400598	000598	000118	00	A	0	0	8
[16]	.init_array	INIT_ARRAY	0000000000600e40	000e40	000008	08	WA	0	0	8
[17]	.fini_array	FINI_ARRAY	0000000000600e48	000e48	000008	08	WA	0	0	8
[18]	.dynamic	DYNAMIC	0000000000600e50	000e50	0001a0	10	WA	6	0	8
[19]	.got	PROGBITS	0000000000600ff0	000ff0	000010	08	WA	0	0	8
[20]	.got.plt	PROGBITS	0000000000601000	001000	000018	08	WA	0	0	8
[21]	.data	PROGBITS	0000000000601018	001018	000004	00	WA	0	0	1
[22]	.bss	NOBITS	000000000060101c	00101c	000004	00	WA	0	0	1
[23]	.comment	PROGBITS	0000000000000000	00101c	000058	01	MS	0	0	1
[24]	.symtab	SYMTAB	0000000000000000	001078	0005a0	18		25	41	8
[25]	.strtab	STRTAB	0000000000000000	001618	0001c0	00			0	1
[26]	.shstrtab	STRTAB	0000000000000000	0017d8	0000f9	00			0	1

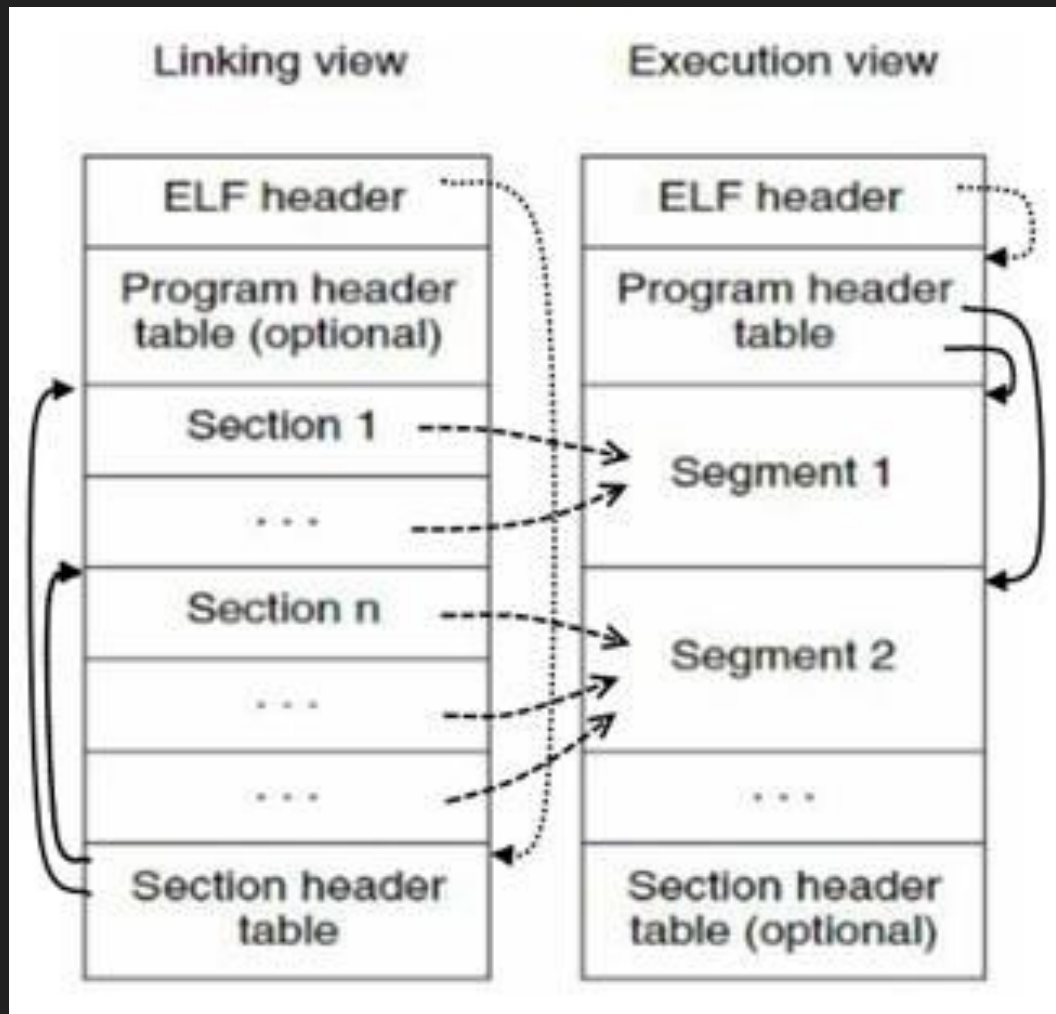
Clé des fanions :

W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe),
T (TLS), C (comprimé), x (inconnu), o (spécifique à l'OS), E (exclu),
l (grand), p (processor specific)

En-têtes de programme :

Type	Décalage	Adr. vir.	Adr.phys.	T.Fich.	T.Mém.	Fan	Alignement
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R	E 0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1
[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x0006b0	0x0006b0	R	E 0x200000
LOAD	0x000e40	0x0000000000600e40	0x0000000000600e40	0x0001dc	0x0001e0	RW	0x200000
DYNAMIC	0x000e50	0x0000000000600e50	0x0000000000600e50	0x0001a0	0x0001a0	RW	0x8
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x000044	0x000044	R	0x4
GNU_EH_FRAME	0x000550	0x0000000000400550	0x0000000000400550	0x000044	0x000044	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10
GNU_RELRO	0x000e40	0x0000000000600e40	0x0000000000600e40	0x0001c0	0x0001c0	R	0x1

Format ELF

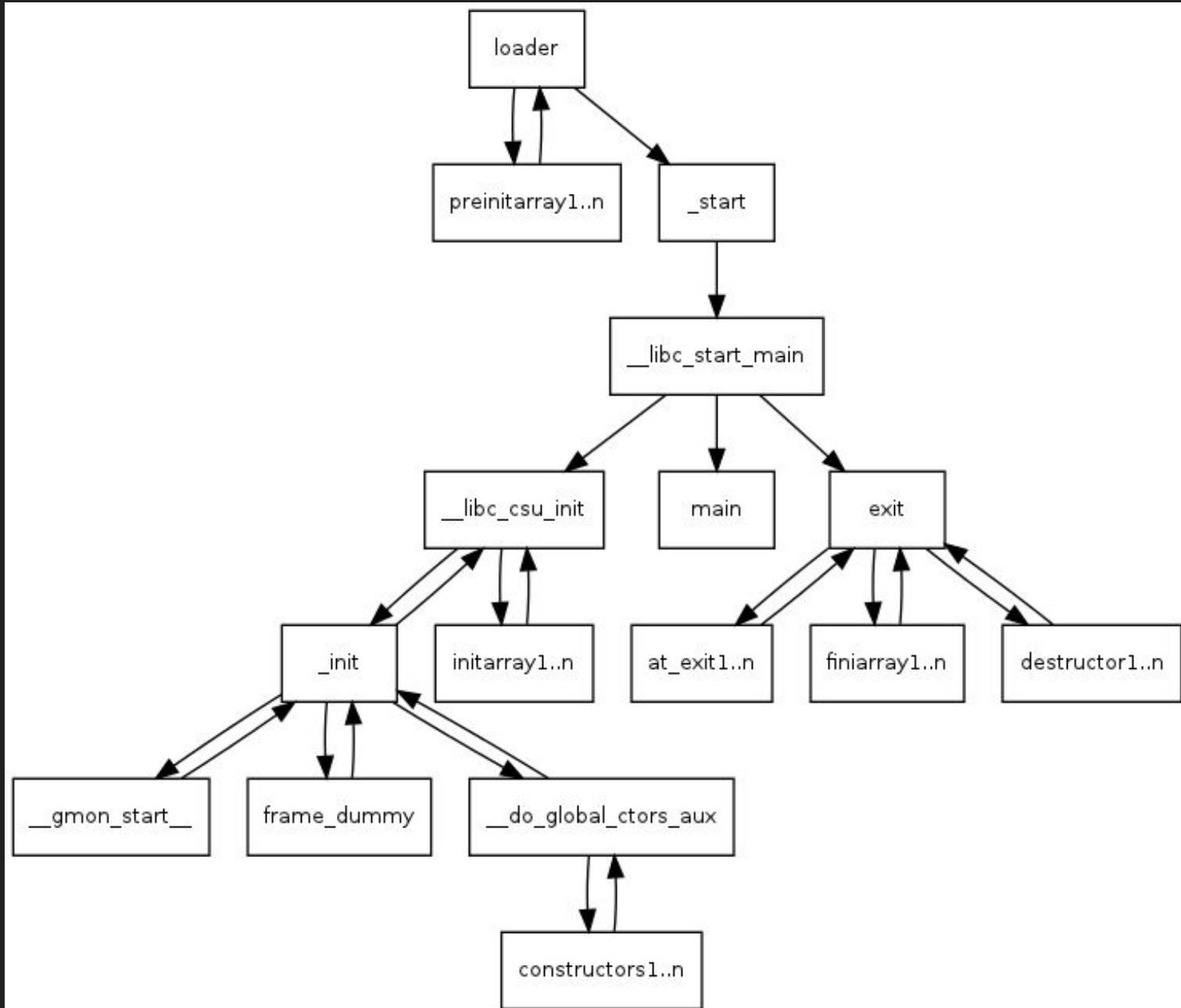


Format ELF

- `.text` : code exécutable
- `.data` / `.bss` : données globales
- `.rodata` : constantes
- `.tdata/.tbss` : Section de données thread-specific (TLS)
- `.got` : Table globale permettant d'avoir un accès indirect aux symboles globaux
- `.got.plt` : GOT pour fonctions dynamiques
- `.rel[a].*` : Symbole repositionnable, à résoudre avant le début du programme
- `.init` : prologue
- `.fini` : épilogue
- `.dynamic` : données utiles au loader pour charger les bibliothèques dynamiques
- `.dynstr` : Chaîne de noms des symboles globaux
- `.dynsym` : Table des symboles globaux
- `.symtab` : table de symbole
- `.c/dtors` : Stockage des routines `pre-main()`

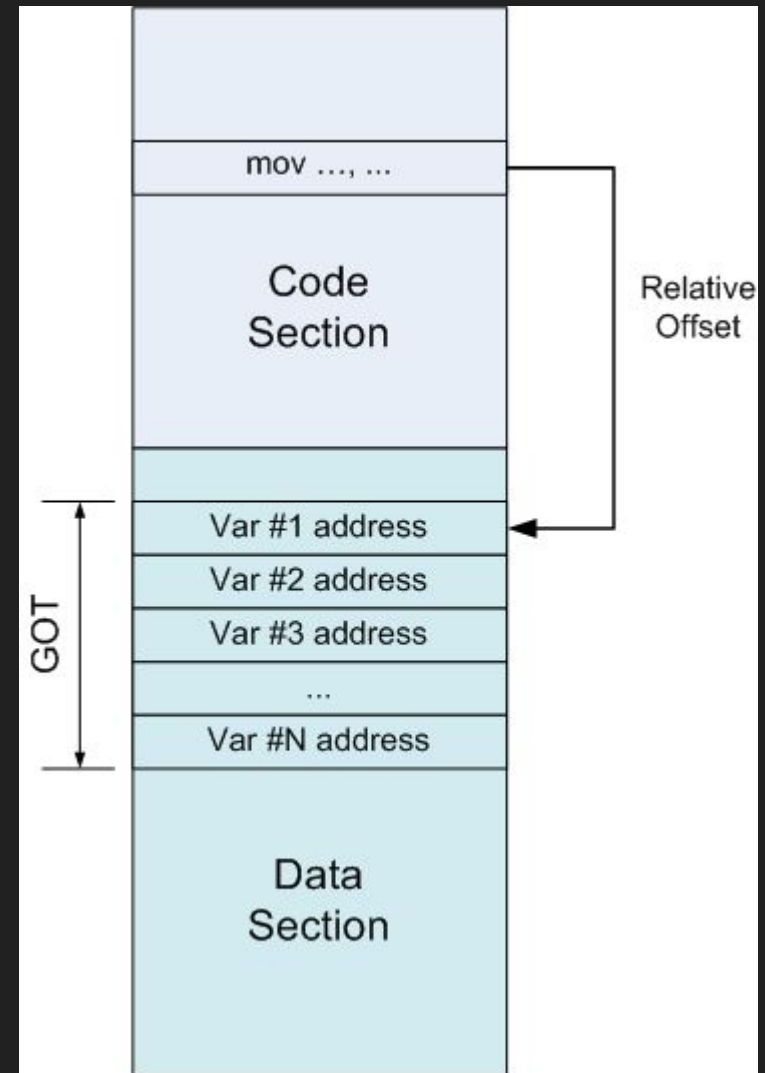
Chargement du programme

- En partant du Shell :
 - a. invocation de `execv*()`
 - b. `search_binary_handler()` : détection du type d'exécutable
 - c. `load_elf_binary()` : mapping du binaire en mémoire (les deux segments `PT_LOAD`)
 - d. Si un interpréteur est nécessaire (section `.interp`), préparation de l'interpréteur pour charger les segments des bibliothèques (`load_elf_interp()`)
 - e. `start_thread()` : la main est enfin donné au code utilisateur
 - f. chargement des bibliothèques dans l'espace mémoire (PIC)
 - g. invocation de la fonction `_start()`
 - h. Invocation de `__libc_start_main()(_init & _fini)`
 - i. Invocation du `main()`



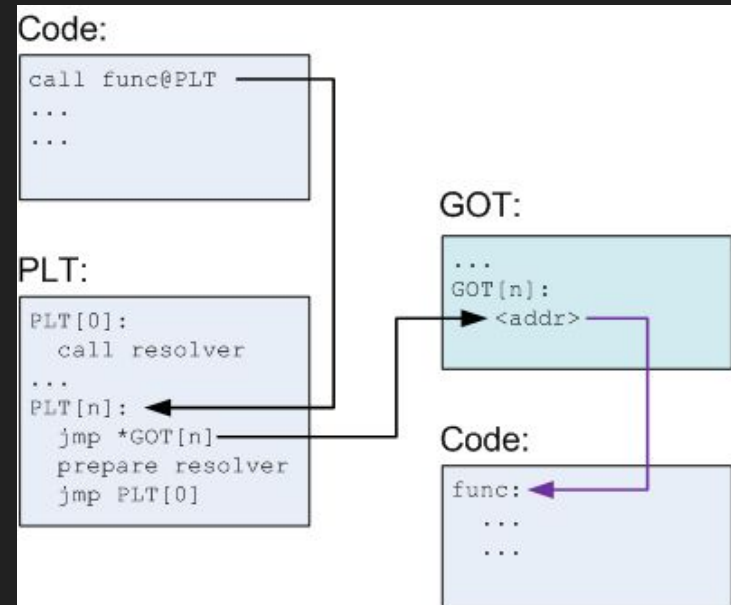
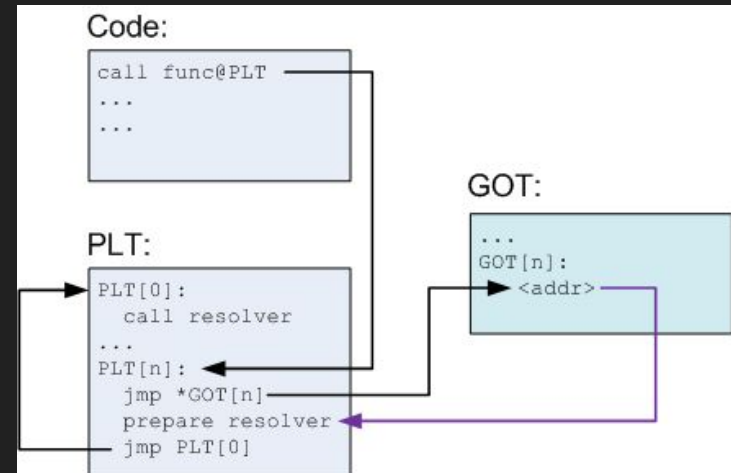
“Relocation” dynamique

- Les codes compilés en “position-independent” (PIC/PIE) sont des codes qui ne possèdent pas de lien mémoire “en dur”. Il faut donc les résoudre pour le programme fonctionne
- Les bibliothèques, qui peuvent être placés n’importe où dans l’espace mémoire, sont toujours en -fPIC.
- Mais quand faire cette “relocation” ?
 - à la compilation
 - au chargement du programme
 - à l’exécution
- GOT = Global Offset Table
- => Indexation de toutes les variables non déterministes



“Relocation” dynamique de fonctions

- Comment gérer ce même comportement pour les fonctions ?
- Résolution de toutes les fonctions au chargement est peu efficace (latence)
- => **Lazy binding !**
- La PLT ajoute un niveau d'indirection supplémentaire à une GOT dédié
- Au 1er appel, la PLT invoque une fonction du loader (lookup).
- L'adresse est mise à jour dans la GOT
- Les futurs appels n'auront qu'un JMP
- Forcer une résolution complète avec `LD_BIND_NOW`



Outils utiles

- Compilateurs (C): **gcc**, `icc`, `xlc`, `clang`, `pgcc`...
 - Dont intermédiaires : `cpp`, `as`, `ld/gold`
- Debuggers : **gdb**, `ddt`, `lldb`, `adb`
- Analyse binaire (*disassembling*) :
 - ELF : **readelf**, `hte`, `elfedit`, **nm**
 - Objets : **objdump**, `objcopy`
 - Conversion : `xxd`, `hexdump`, `base64`
- Bonus : `radare2`, `peda`
- Opcodes x86_64 : <http://ref.x86asm.net/coder64.html>
- Sources : <https://github.com/gweodoo/AISE-20.git>