

Chaîne de compilation

Master CHPS

Julien Adam <adamj@paratools.com>



Kahoot !

Organisation du Cours

Cour MATIN, TD après-midi

➡ 10/01: Généralités sur les OS et Utilisation de base

➡ 15/01 : La Chaine de Compilation et l'exécution d'un programme

➡ 16/01 : Les I/Os POSIX et Introduction aux Sockets

➡ 30/01 : Méthodes de communication Inter-Processus

➡ 01/02 : Mémoire avancée (mmap, madvise, pages, TLB, ...)

➡ 06/02 : Programmation et reverse et Q/A projets (journée TD)

➡ 13/02 : TD Débogage (gdb, valgrind) && TD mesure du temps et profilage (perf, kcachegrind) et Q/A projets (journée TD)

➡ Un examen final (25 Mars matin)

Plan du cours

1. Les étapes de compilation
2. Structure d'un binaire
3. Layout d'un processus

Outils utiles

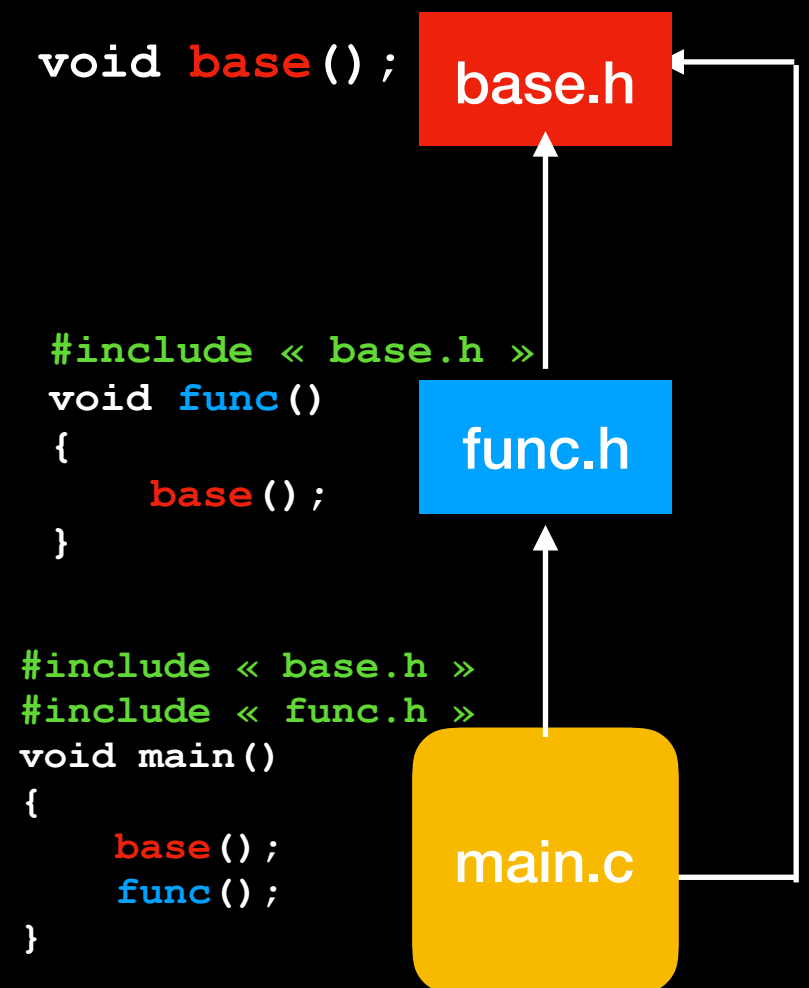
- Compilateurs (C): **gcc**, `icc`, `xlc`, `clang`, `pgcc`...
 - Dont intermédiaires : `cpp`, `as`, `ld/gold`
- Debuggers : **gdb**, `ddt`, `lldb`, `adb`
- Analyse binaire (*disassembling*) :
 - ELF : **readelf**, `hte`, `elfedit`, **nm**
 - Objets : **objdump**, `objcopy`
 - Conversion : `xxd`, `hexdump`,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

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 binaires spécifiques à l'architecture.
- L'édition de liens: Assemblage des différentes TU pour créer un exécutable

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();
```

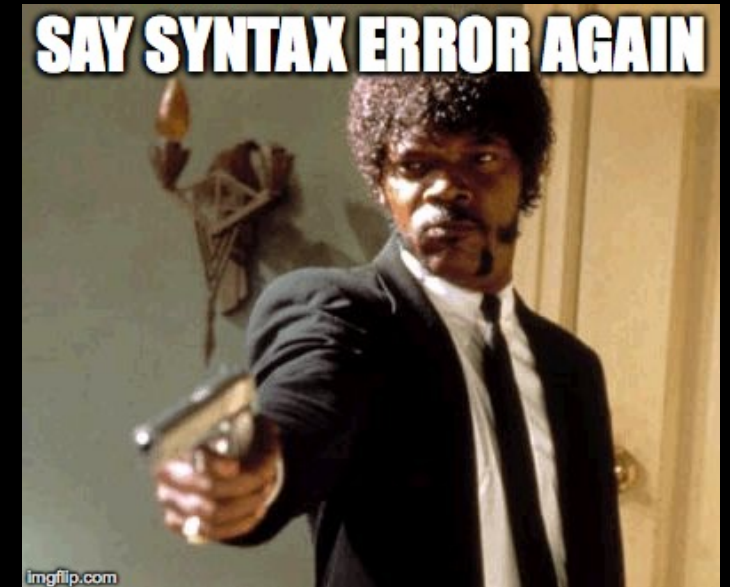
base.h

```
#include « base.h »
#include « base.h »
void func()
{
    base();
}
void main()
{
    base();
    func();
}
```

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...` :
- Résolution des chemins d'inclusion selon 2 règles :
 - Les « quotes » pour un chemin relatif depuis le répertoire courant
 - Les <chevrons> pour un chemin absolu défini par défaut
- Certaines constantes existent (`__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`), et extensions selon l'architecture (`__WIN32`, `__APPLE`, `__linux__`)



```
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;
}
```

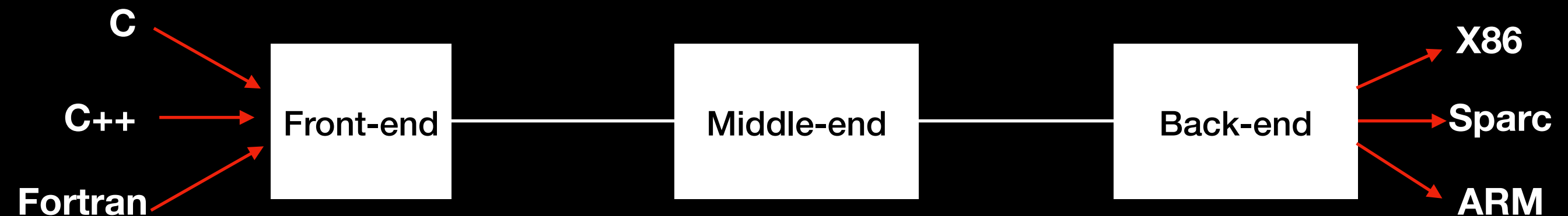
```
➡ gcc -E main.c -o main.i
main.c:1:10: erreur fatale: base.h : Aucun fichier ou dossier de ce type
#include <base.h>
      ^~~~~~
compilation terminée.
```

Démo ?



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

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

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`

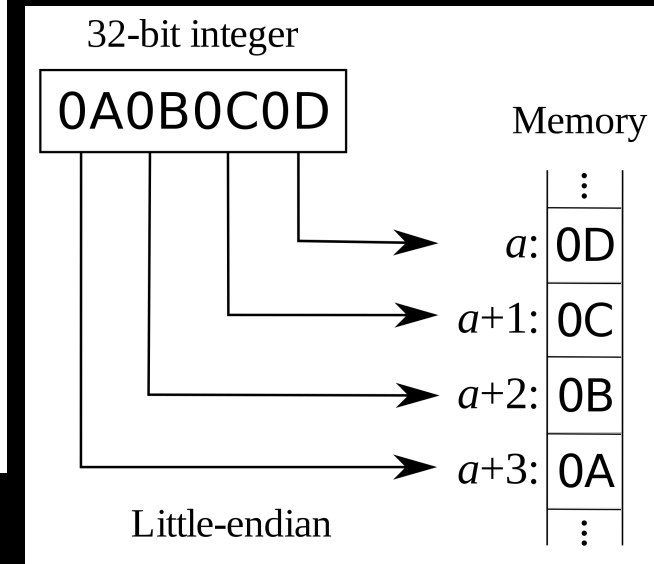
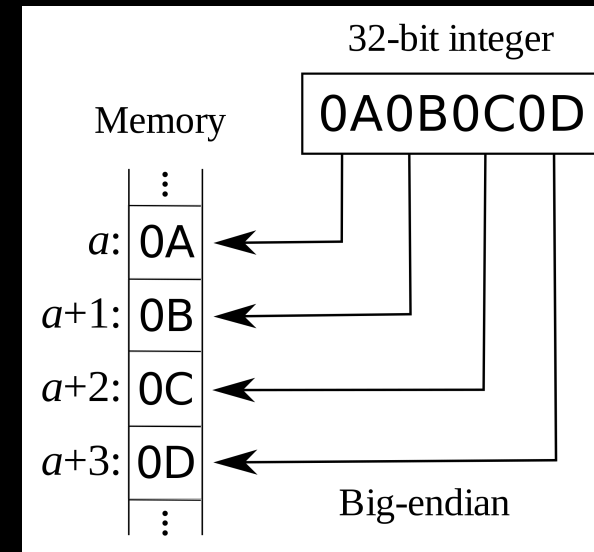
```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 14
.globl _func
.p2align 4, 0x90
_func:
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
movb $0, %al
callq _base
popq %rbp
retq
.cfi_endproc

.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.o`
- 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 (compressé), 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

Artefacts compilo-spécifiques

- `crt1.o` / `crt0.o`...
- `crti.o` / `crtn.o`
- `crtbegin.o` / `crtend.o`
- `crtbeginS.o` / `crtendS.o`
- `crtbeginT.o` / `crtendT.o`

• 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>:
```

Démo ?



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/libcc1.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

- 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)

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

- 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 (0x00007ff6e0cf2000)  
/lib64/ld-linux-x86-64.so.2 (0x00007ff6e10b1000)
```

```
└─▶ 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
```

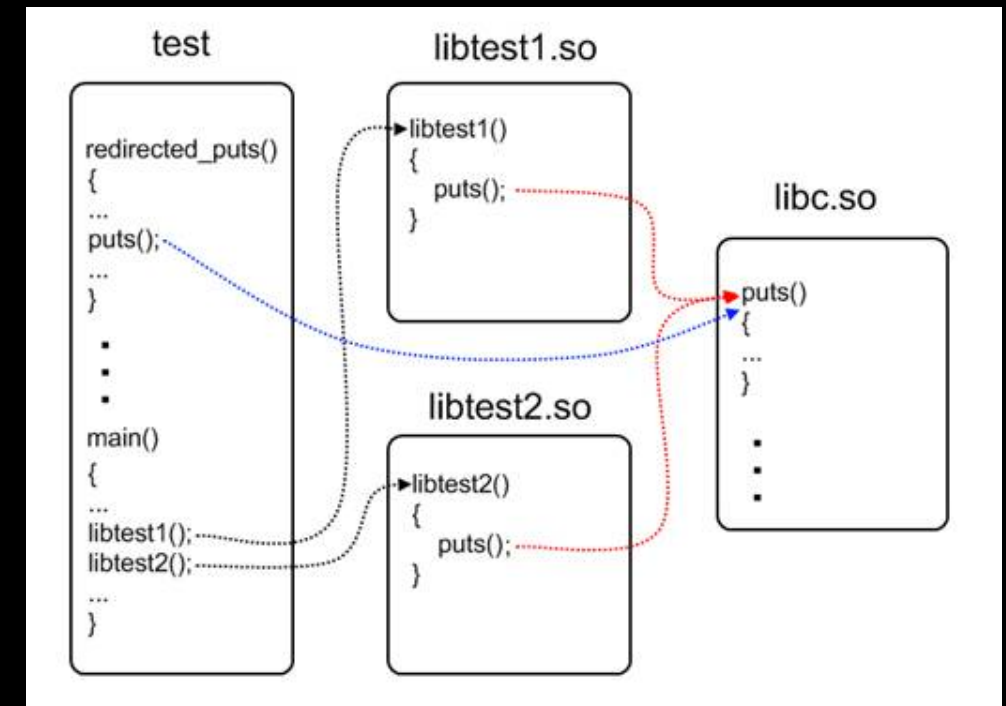
```
└─▶ readelf -dW ./a.out  
Section dynamique à l'offset 0xe60 contient 20 entrées :  
Étiquettes Type Nom/Valeur  
0x0000000000000001 (NEEDED) Bibliothèque partagée: [libc.so.6]  
0x000000000000000c (INIT) 0x400390  
0x000000000000000d (FINI) 0x400534  
0x0000000000000019 (INIT_ARRAY) 0x600e50  
0x000000000000001b (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 redéfinis.
- 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)
```


Démo ?




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`

Makefile

- Fichier de configuration de l'outil **Make**
- Facilite la compilation et le link de programmes
-  **Le nom du fichier est toujours Makefile (sauf explicite)**
- Vue du Shell, on compile avec la commande **make** dans le répertoire où se trouve le Makefile. Cela uniformise la compilation de tout programme (et donc la nôtre et la vôtre !)
- Reproductibilité de compilation (il est facile d'oublier un flag de compilation)
- Principaux arguments à la commande make :
 - **-f myMakefile** : changer le nom du fichier
 - **-C <chemin_du_projet>** : pour compiler sans être dans le répertoire
 - Tout argument non-Make est ensuite utilisé comme nom de cible (voir slide suivant)

Une seule des centaines de fois que GCC est appelé pour compiler... GCC :

```
libtool: compile: gcc -DTIME_WITH_SYS_TIME=1 -DHAVE_INTTYPES_H=1 -DHAVE_STDINT_H=1 -DHAVE_LOCALE_H=1 -DHAVE_WCHAR_H=1 -DHAVE_STDARG=1 -DHAVE_SYS_TIME_
H=1 -DHAVE_STRUCT_LCONV_DECIMAL_POINT=1 -DHAVE_STRUCT_LCONV_THOUSANDS_SEP=1 -DHAVE_ALLOCA_H=1 -DHAVE_STDINT_H=1 -DHAVE_VA_COPY=1 -DHAVE_SETLOCALE=1 -DH
AVE_GETTIMEOFDAY=1 -DHAVE_LONG_LONG=1 -DHAVE_INTMAX_T=1 -DMPFR_HAVE_INTMAX_MAX=1 -DMPFR_HAVE_FESETRound=1 -DHAVE_DENORMS=1 -DHAVE_SIGNEDZ=1 -DHAVE_ROUND
D=1 -DHAVE_TRUNC=1 -DHAVE_FLOOR=1 -DHAVE_CEIL=1 -DHAVE_NEARBYINT=1 -DHAVE_LDOUBLE_IEEE_EXT_LITTLE=1 -DMPFR_USE_THREAD_SAFE=1 -DMPFR_USE_C11_THREAD_SAFE
=1 -DHAVE_CLOCK_GETTIME=1 -DLT_OBJDIR=\".libs/\" -DHAVE_ATTRIBUTE_MODE=1 -DHAVE___GMPN_ROOTREM=1 -DHAVE___GMPN_SBPI1_DIVAPPR_Q=1 -I. -I../..../mpfr/src
c -I$BUILD_PATH/x86_64/x86_64/gcc-7.2.0/build/gmp -DNO_ASM -g -O2 -MT ai.lo -MD -MP -MF .deps/ai.Tpo -c ../..../mpfr/src/ai.c -o ai.c
```



Makefile

- Une cible définit un fichier à construire ou une action
- Une règle est l'ensemble des commandes à exécuter pour réaliser cette cible (lancées dans un Shell différent). Chaque règle commence par une **tabulation**
- La première cible est celle exécutée par défaut
- Une cible peut avoir des dépendances, des cibles à résoudre **avant**.
- La résolution de dépendances fonctionne par horodatage. Permet par exemple de recompiler les .o pour lesquels le fichier .c a été modifié en amont

<code>\$@</code>	Nom de la cible
<code>\$<</code>	Nom de la 1ere dépendance
<code>\$^</code>	Nom de toutes les dépendances
<code>\$?</code>	Nom des dépendances plus récentes que la cible
<code>\$*</code>	Nom du fichier sans suffixe (voir .SUFFIXES)

```
main.bin:
    gcc -o main.bin main.c
```

```
main.o: main.c
    gcc -c main.c -o main.o

main.bin: main.o
    gcc -o main.bin main.o
```

```
main.o: main.c
    gcc -c main.c -o $@

main.bin: main.o
    gcc -o $@ main.o
```

```
main.o: main.c
    gcc -c $< -o $@

main.bin: main.o
    gcc -o $@ $<
```

```
%.o: %.c
    gcc -c $< -o $@

main.bin: main.o
    gcc -o $@ $<
```



`$ make main.bin`

Makefile

```
# Indiquer quel compilateur est à utiliser
CC      ?= gcc

# Spécifier les options du compilateur
CFLAGS  ?= -g
LDFLAGS ?= -L/usr/lib
LDLIBS  ?= -ldl

# Reconnaître les extensions de nom de fichier *.c et *.o comme suffixes
SUFFIXES ?= .c .o
.SUFFIXES: $(SUFFIXES) .

# Nom de l'exécutable
PROG    = main

# Liste de fichiers objets nécessaires pour le programme final
OBJS    = main.o module.o

all: $(PROG)

# Étape de compilation et d'éditions de liens
# ATTENTION, les lignes suivantes contenant "$(CC)" commencent par un caractère TABULATION et non pas des espaces
$(PROG): $(OBJS)
    $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)

.c.o:
    $(CC) $(CFLAGS) -c $.c
```

Source : <https://fr.wikipedia.org/wiki/Make>

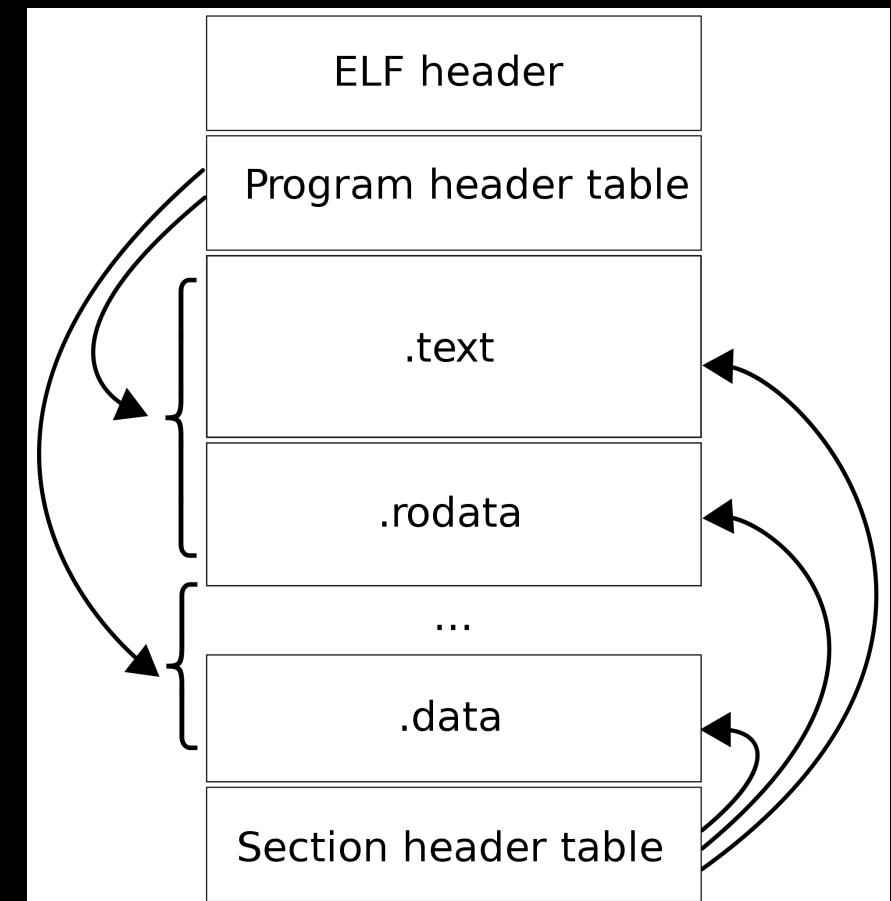
Pour une documentation dense (raccourcis,etc...) : <https://www.gnu.org/software/make/manual/make.html>

Démo ?

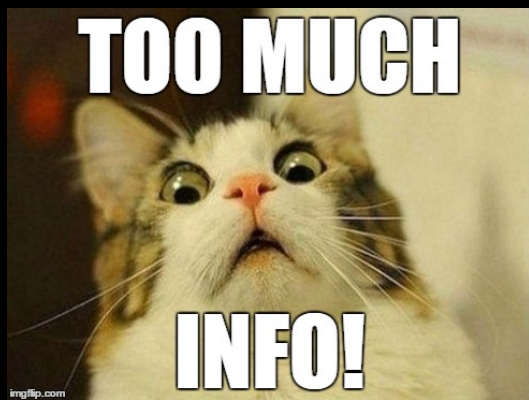


Format ELF

- 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 ou l'autre des tables précitées

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	0	1
[26]	.shstrtab	STRTAB	0000000000000000	0017d8	0000f9	00		0	0	1

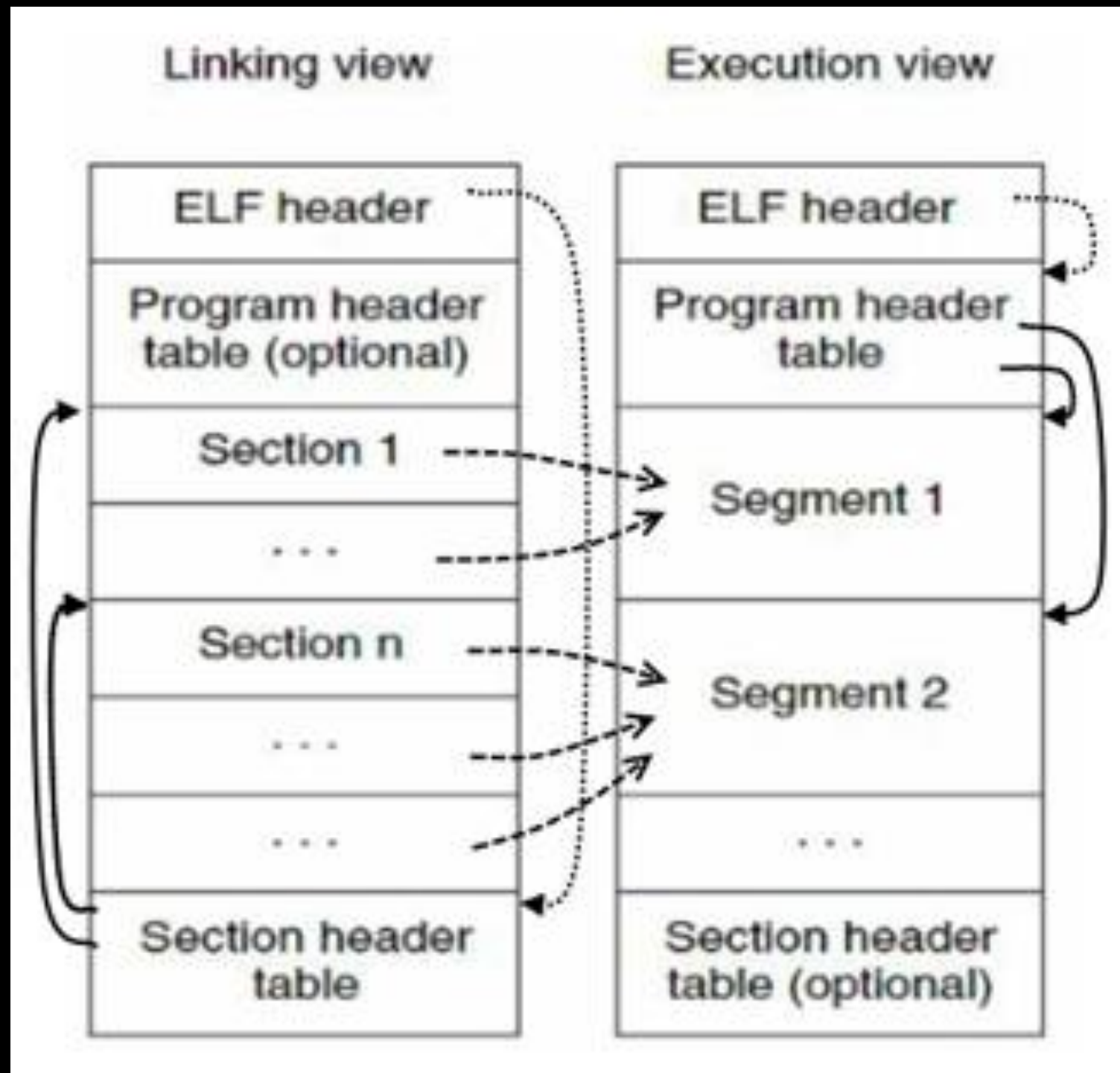
Clé des fanions :

W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
L (ordre des liens), 0 (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	0x00001f8	0x00001f8	R E	0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x000001c	0x000001c	R	0x1
[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x00006b0	0x00006b0	R E	0x200000
LOAD	0x000e40	0x0000000000600e40	0x0000000000600e40	0x00001dc	0x00001e0	RW	0x200000
DYNAMIC	0x000e50	0x0000000000600e50	0x0000000000600e50	0x00001a0	0x00001a0	RW	0x8
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x0000044	0x0000044	R	0x4
GNU_EH_FRAME	0x000550	0x0000000000400550	0x0000000000400550	0x0000044	0x0000044	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x0000000	0x0000000	RW	0x10
GNU_RELRO	0x000e40	0x0000000000600e40	0x0000000000600e40	0x00001c0	0x00001c0	R	0x1

Format ELF



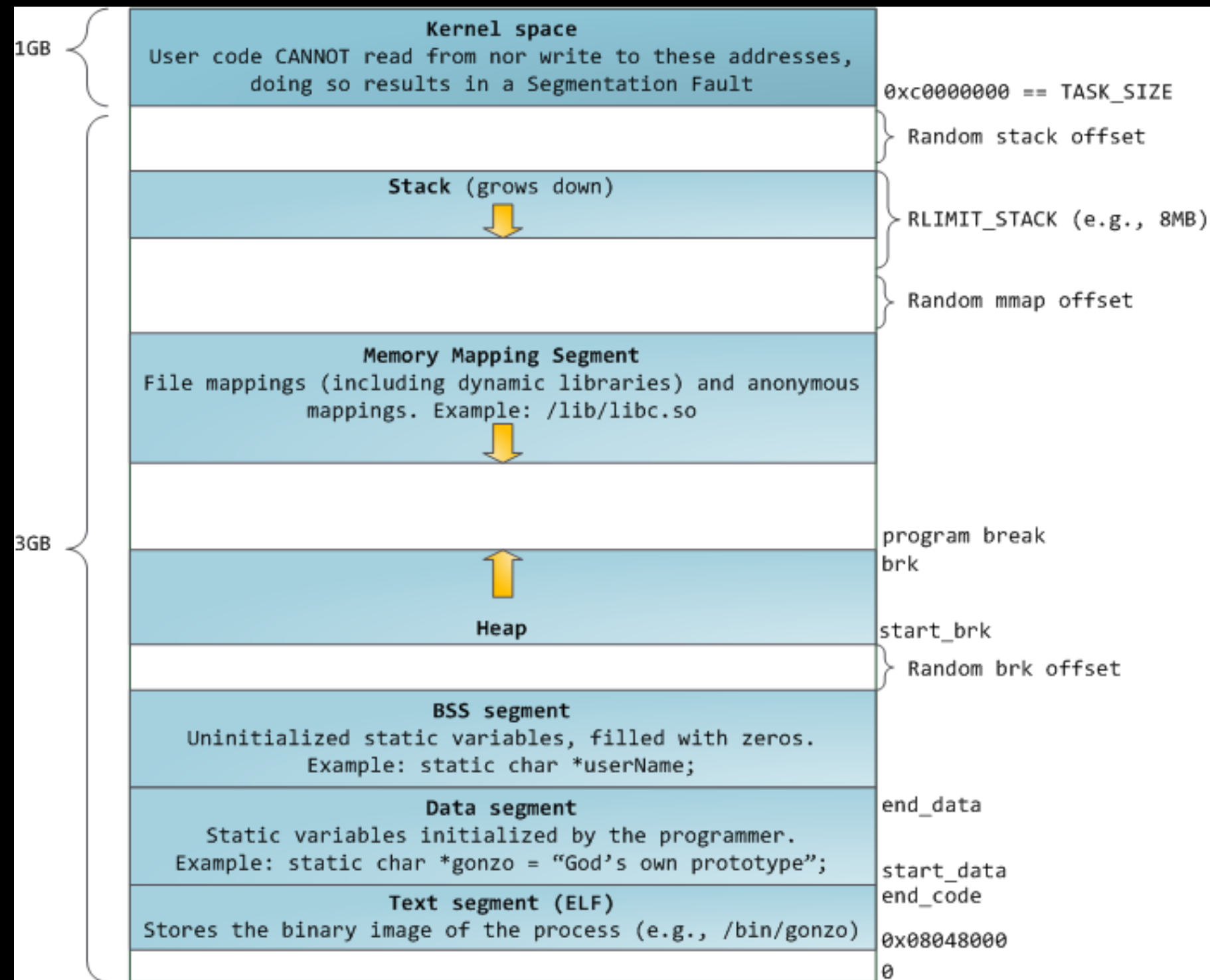
Format ELF

- **.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()**

Démo ?



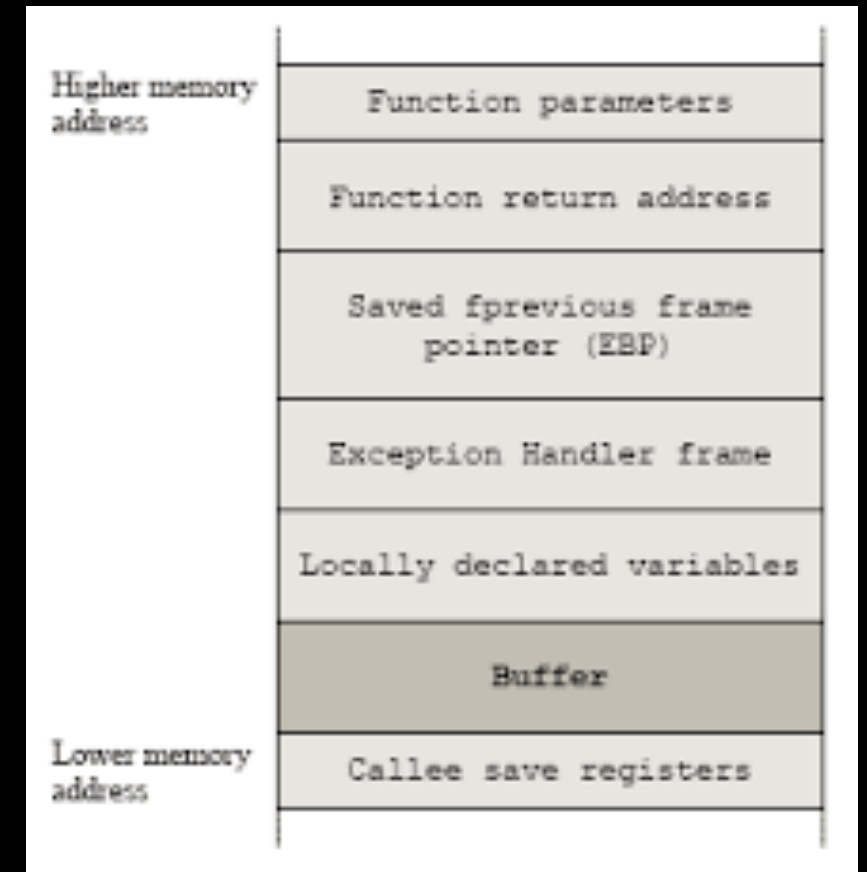
Layout de la mémoire



- 64 bits théoriques
- 48 bits câblés (soit 256 Tio adressables)
- Les adresses de pile décroissent
- Les adresses allouées dynamiquement croissent
- Dernière adresse : `0x00007fffffffffffffff`

Layout de la pile

- La pile est une superposition couches appelées « frame », toutes identiques. Une stackframe est créée à chaque fois qu'une nouvelle fonction est appelée (instruction x86 **call***)
- Dans une stack-frame est stockée :
 - Les arguments de fonctions
 - L'adresse de retour RIP dans la fonction parente (pour *Return Instruction Pointer*)
 - Le pointeur de pile **RBP** de la frame précédente
 - Les variables automatiques (dites « locales »)
- Il existe deux pointeurs de pile



```
gdb >> disas main
Dump of assembler code for function main:
0x000000000040052e <+0>:      push    %rbp
0x000000000040052f <+1>:      mov     %rsp,%rbp
0x0000000000400532 <+4>:      sub     $0x10,%rsp
```

- **RBP** : « Base pointer » = l'adresse où commence la frame courante
- **RSP** : « Stack Pointer » = l'adresse qui suit la dernière adresse accessible pour la frame courante

Manipulation des Registres:

- **r* = 64 bits**
- **e* = 32 bits**
- **l* = 16 bits**

Démo ?

