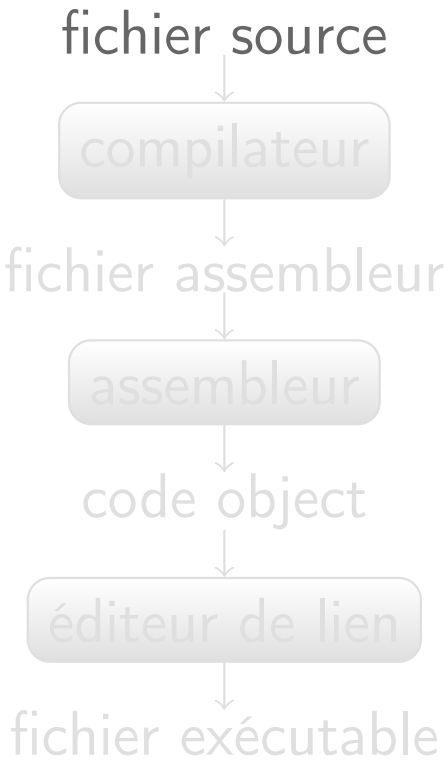
Hugues Cassé <casse@irit.fr>
FSI - Université de Toulouse

Cours 1 : l'édition des liens

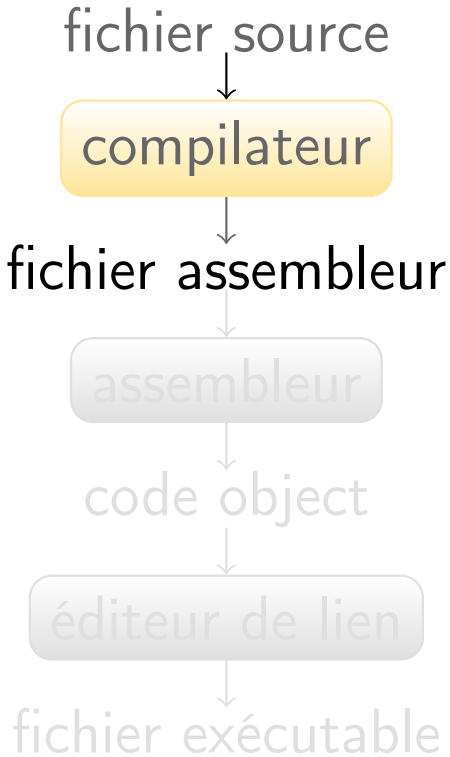
COMC

Modèle d'exécution



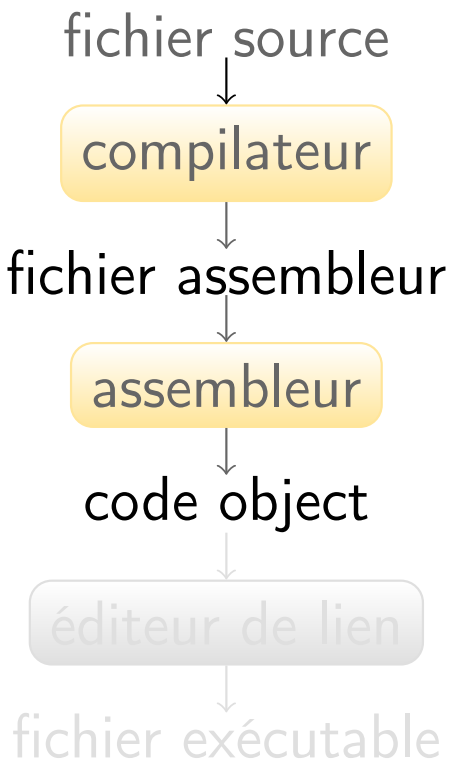

```
int main() {
  int s = 0;
  for(int i = 0; i < 10; i++)
      s += i;
  printf("%d", s);
}
```

Modèle d'exécution



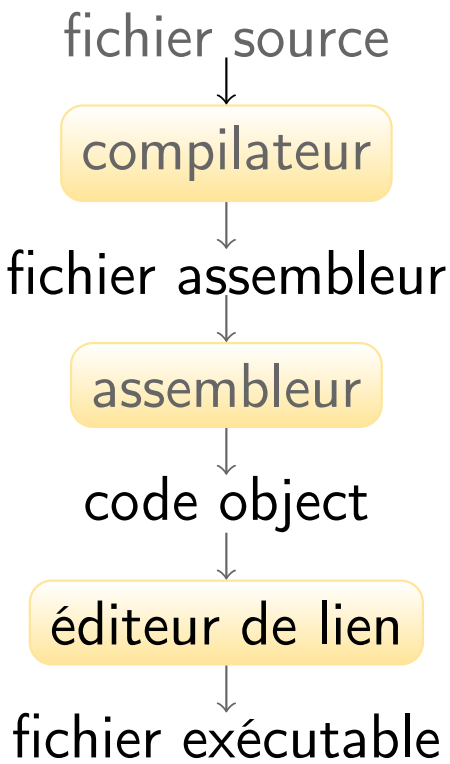

```
main:
  mov R0, #0
  mov R1, #0
label1:
  cmp R1, #10
  bhs label2
  add R0, R0, R1
  add R1, R1, #1
  b label1
label2:
  mov R1, R0
  adr R0, string1
  bl  printf
  mov PC, LR
```

Modèle d'exécution




```
e3a00000
e3a01001
e351000a
2a000002
e0800001
e2811001
eafffffa
e1a01000
e28f0004
ebffffe
e1a0f00e
```

Modèle d'exécution




```
e3a00000
e3a01001
e351000a
2a000002
e0800001
e2811001
eafffffa
e1a01000
e28f0004
ebffffad
e1a0f00e
```

Plan

Un peu de vocabulaire

source partie de programme en langage source

objet partie de programme en langage binaire

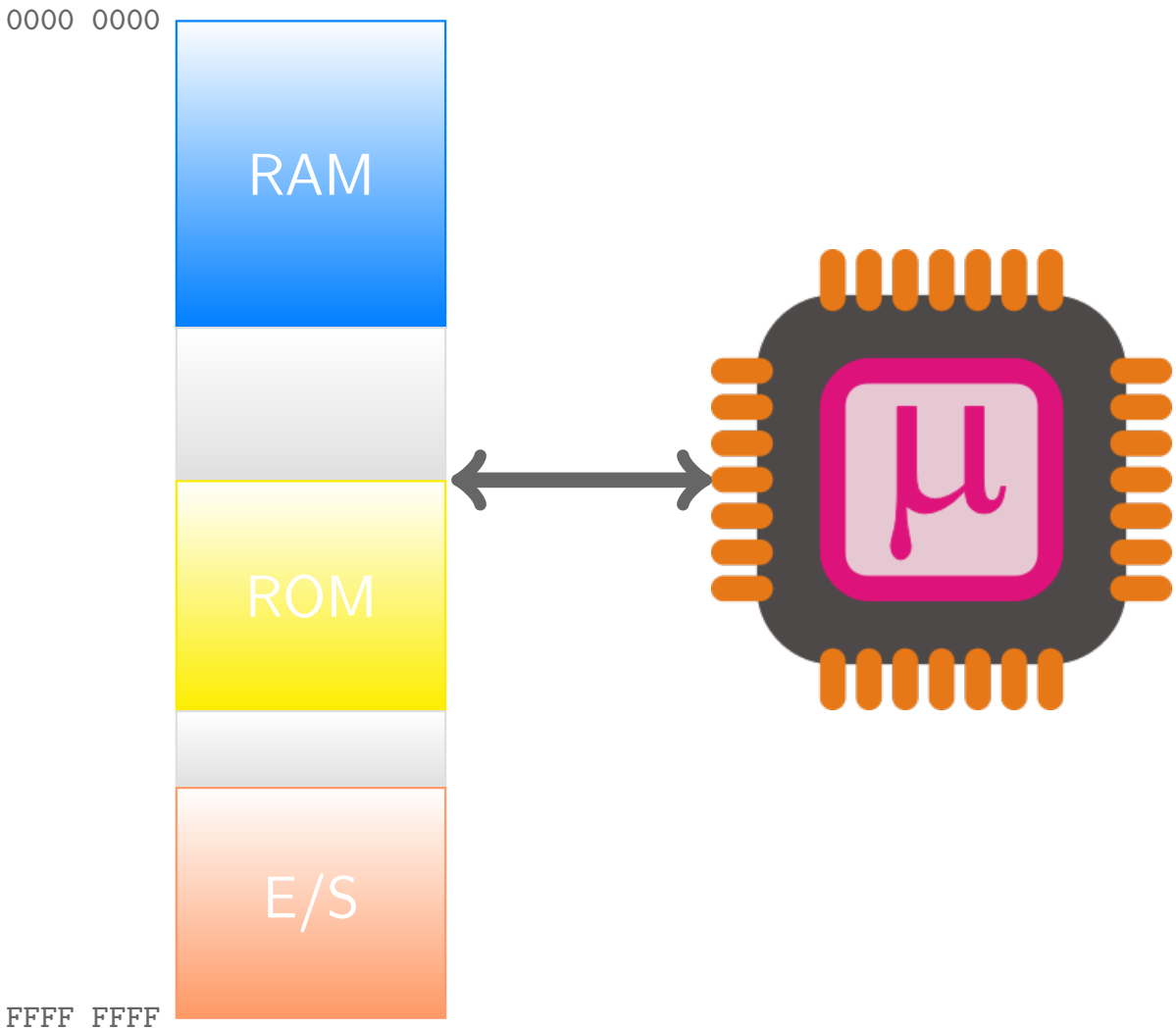
exécutable programme complet prêt à être exécuté

bibliothèque (statique) collection de fichiers objet

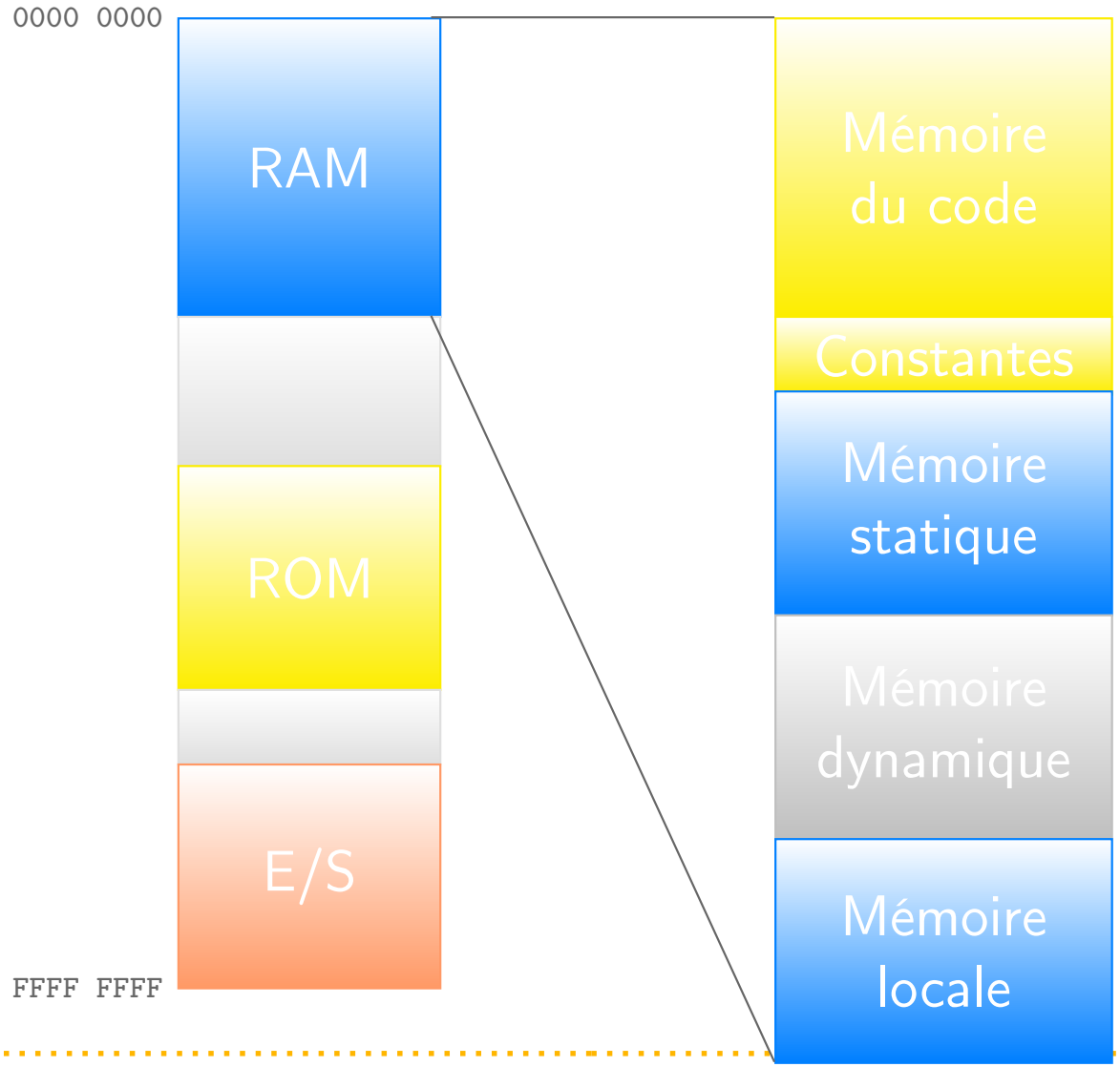
bilbiothèque dynamique (ou partagée) code lié à l'exécution du programme

image (d'un programme) configuration mémoire du programme à son démarrage (code + données)

Structure physique de la mémoire



Structure logique de la RAM



Format binaire

Definition

Format des fichiers servant à stocker des programmes.

Exemples :

ELF	(*Exchange and Linking Format*) – Unix + embarqué
PE-COFF	Windows (hérité du format Unix ECOFF)
Mach-O	MacOSX (plusieurs binaires d'architectures différentes)
byte-code	machine virtuelle Java

Structure d'un fichier ELF

En-tête ELF :

- architecture supportant le binaire
- système supportant le binaire
- endianness (little ou big)
- taille de mot (32-bit ou 64-bit)
- adresse de démarrage

Table des sections : vue logique (éditeur de lien, débogueur, etc)
Table des en-tête de programme : chargement rapide de l'exécutable
Code + données du programme.

La commande `objdump`

`objdump` *options fichier_ELF*

Consultation des fichiers ELF :

- -d désassemblage des sections de code
- -f en-tête ELF
- -g information de débogage
- -h liste des sections
- -l table de correspondance ligne-source / adresse
- -r liste des relocationss
- -t liste des symboles définis et utilisés

Alternative : `readelf` (plus orientée ELF)

En-tête programme

en-tête programme = segment de mémoire dans l'image du programme
Caractéristiques :

- adresse dans l'image
- position dans le fichier
- taille (en octet)
- droits d'accès (exécution, lecture, écriture)

fonction $\Longrightarrow$ lecture + exécution (mémoire du code)
variables globales $\Longrightarrow$ lecture + écriture (mémoire statique)
données constantes $\Longrightarrow$ lecture

Chargement d'un programme

Le chargeur de programme :

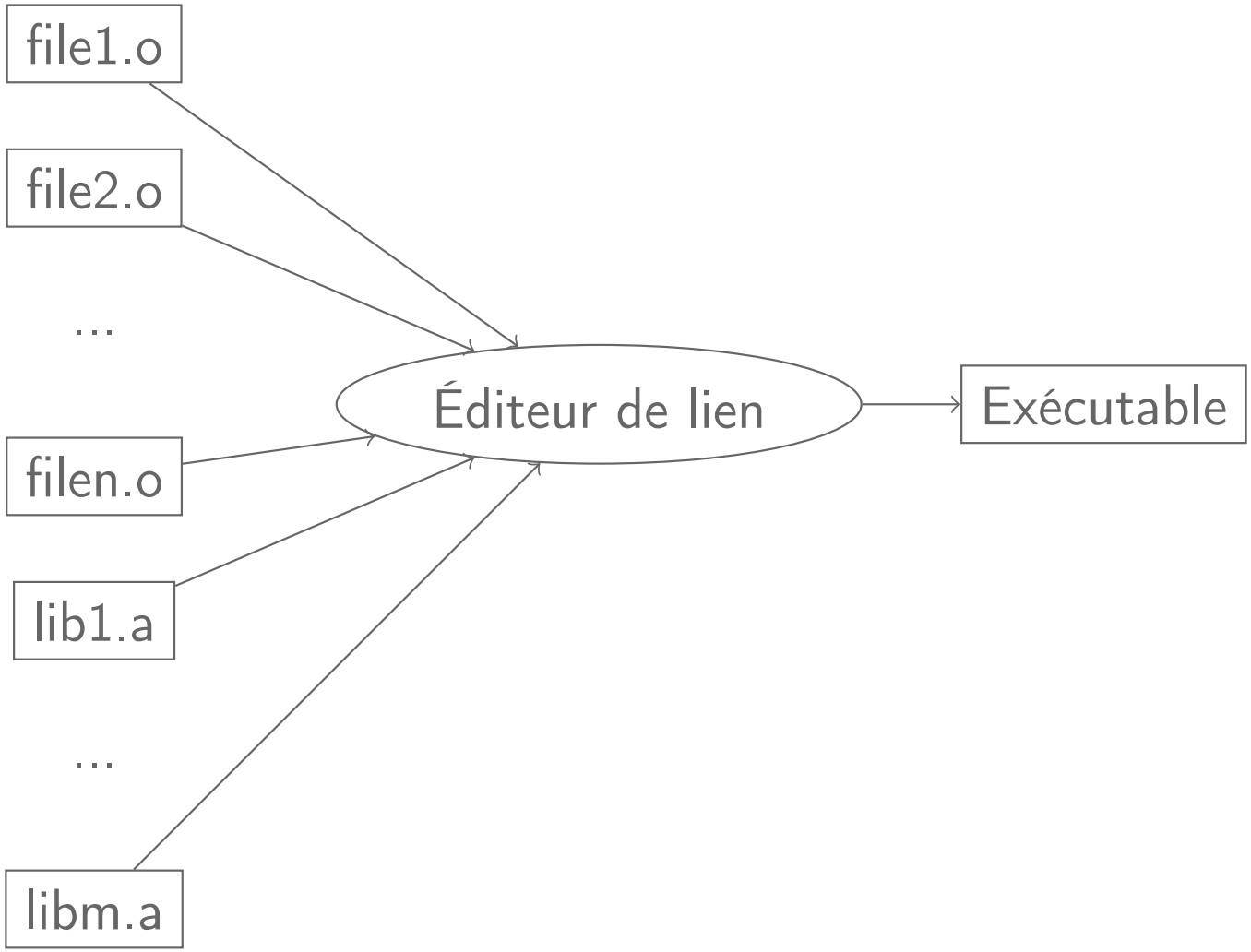
1. lit les en-têtes programme du fichier exécutable
2. copie des segments de programme initialisés à leur adresse
3. met à 0 des données globales non-initialisées
4. initialise la pile et le pointeur SP
5. lance le programme (PC $\leftarrow$ adresse de démarrage).

Le chargeur peut être :

- le système d'exploitation
- un installeur de logiciel sur une plate-forme embarquée (JTAG, ...)
- un simulateur de microprocesseur (QEmu)

Plan

Objectif


file1.o
file2.o
...
filen.o
lib1.a
...
libm.a
Éditeur de lien
Exécutable

Les sections

Caractéristiques :

- adresse dans l'image
- position dans le fichier
- taille, alignement en mémoire
- droits d'accès

Sections classiques :

`.text` code

`.data` variables globales initialisées

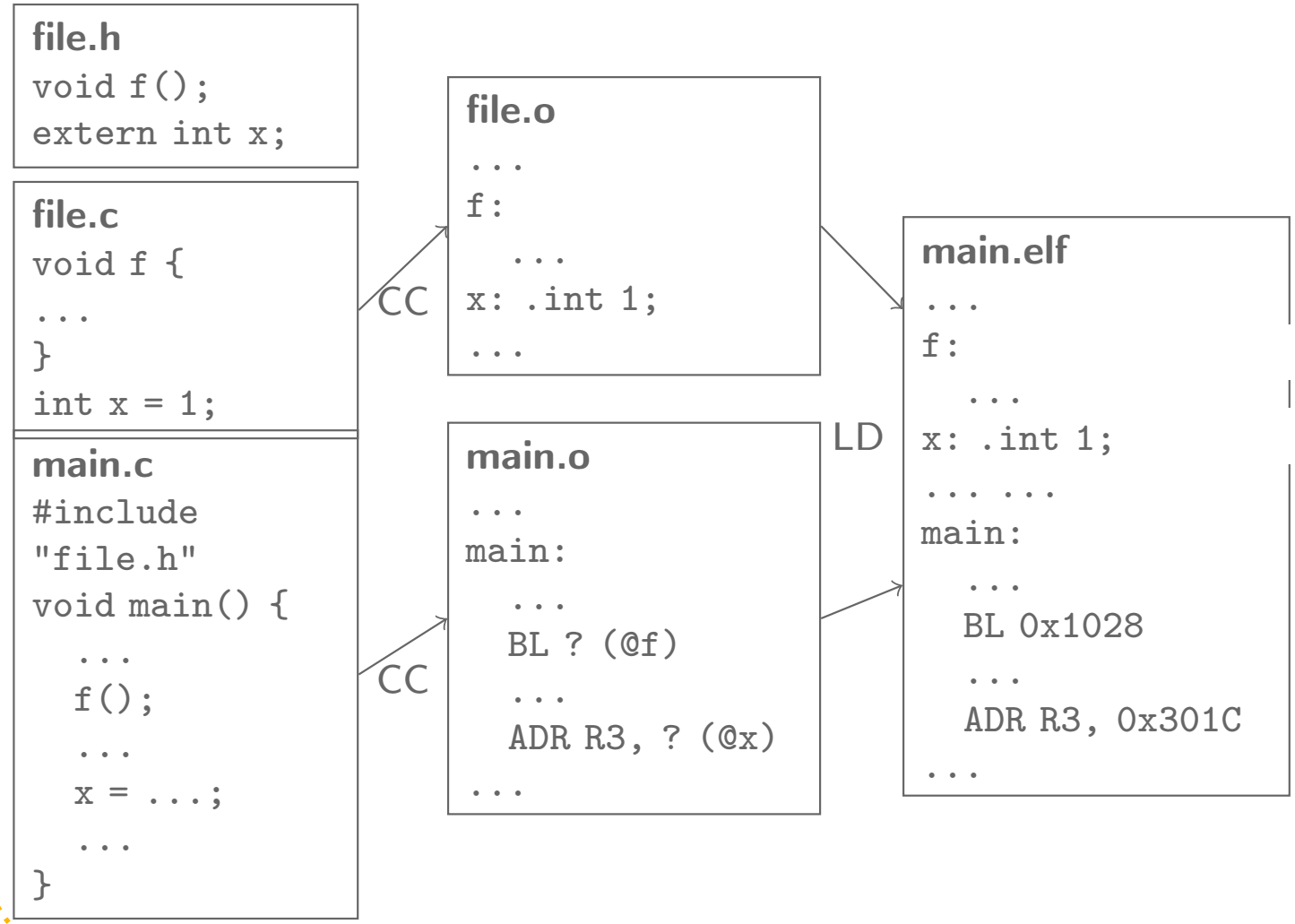
`.rodata` constantes

`.bss` variables globales non-initialisées

`.debug_line` table de correspondance ligne/source $\Leftrightarrow$ adresse

`.debug_XXX` information sur les entités du programme

Problème de l'édition des liens



Sections d'édition de lien

`.symtab` (bibliothèque) – liste des symboles exportés

- nom du symbole
- section contenant le symbole
- position dans la section
- taille du symbole
- nature du symbole (étiquette, fonction, donnée constante)

`.rel.XXX` ou `.rela.XXX` (exécutable) – opération de relocation

- `XXX` = section relogée
- symbole adressé
- position dans la section
- calcul de la relocation (dépendant de l'architecture)

Processus d'édition des liens

1. construction de l'ensemble des objets nécessaires (programme + objets nécessaires des bibliothèques)
2. agrégation des sections de même type
3. détermination des adresses des sections
4. relocation des sections

Construction de l'ensemble des objets

$let\ Libs = \{set\ of\ libraries\}$
$let\ Objs = \{objets\ du\ programme\}$
$let\ Imps = \bigcup_{obj \in Objs} imports(obj)$
$let\ Exps = \bigcup_{obj \in Objs} exports(obj)$
while $Imps \neq \emptyset$ **do**
    $let\ s \in Imps$
    $Imps \leftarrow Imps \setminus \{s\}$
    if $s \notin Exps$ **then**
        if $\exists obj \in Libs \wedge s \in exports(obj)$ **then**
            $Objs \leftarrow Objs \cup \{obj\}$
            $Exps \leftarrow Exps \cup exports(obj)$
            $Imps \leftarrow Imps \cup imports(obj)$
        else
            $error\ "symbole\ manquant"$
        end if
    end if
end while

Exemple (a)

`file.o`

- *imports* $= \{$`printf`$\}$
- *exports* $= \{$`f`, `x`$\}$

`main.o`

- *imports* $= \{$`f`, `x`$\}$
- *exports* $= \{$`main`$\}$

`crt0.o`

- *imports* $= \{$`main`$\}$
- *exports* $= \{$`_start`$\}$

`libc.a` / `printf.o`

- *imports* $= \{...\}$
- *exports* $=$ $\{..., $`printf`$, ...\}$

1. *Objs* $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`main`, `f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$

Exemple (a)

`file.o`

- *imports* $= \{$`printf`$\}$
- *exports* $= \{$`f`, `x`$\}$

`main.o`

- *imports* $= \{$`f`, `x`$\}$
- *exports* $= \{$`main`$\}$

`crt0.o`

- *imports* $= \{$`main`$\}$
- *exports* $= \{$`_start`$\}$

`libc.a` / `printf.o`

- *imports* $= \{...\}$
- *exports* $=$ $\{..., $`printf`$, ...\}$

1. *Objs* $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`main`, `f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
2. `main` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$

Exemple (a)

`file.o`

- *imports* $= \{$`printf`$\}$
- *exports* $= \{$`f`, `x`$\}$

`main.o`

- *imports* $= \{$`f`, `x`$\}$
- *exports* $= \{$`main`$\}$

`crt0.o`

- *imports* $= \{$`main`$\}$
- *exports* $= \{$`_start`$\}$

`libc.a` / `printf.o`

- *imports* $= \{...\}$
- *exports* $=$ $\{..., $`printf`$, ...\}$

1. *Objs* $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`main`, `f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
2. `main` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
3. `f` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$

Exemple (a)

`file.o`

- *imports* $= \{$`printf`$\}$
- *exports* $= \{$`f`, `x`$\}$

`main.o`

- *imports* $= \{$`f`, `x`$\}$
- *exports* $= \{$`main`$\}$

`crt0.o`

- *imports* $= \{$`main`$\}$
- *exports* $= \{$`_start`$\}$

`libc.a` / `printf.o`

- *imports* $= \{\ldots\}$
- *exports* $=$ $\{\ldots,$ `printf`$, \ldots\}$

1. *Objs* $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`main`, `f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
2. `main` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`f`, `x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
3. `f` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`x`, `printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
4. `x` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$

Exemple (b)

`file.o`

- *imports* $= \{$`printf`$\}$
- *exports* $= \{$`f`, `x`$\}$

`main.o`

- *imports* $= \{$`f`, `x`$\}$
- *exports* $= \{$`main`$\}$

`crt0.o`

- *imports* $= \{$`main`$\}$
- *exports* $= \{$`_start`$\}$

`libc.a` / `printf.o`

- *imports* $= \{...\}$
- *exports* $=$ $\{..., $`printf`$, ...\}$

4. `x` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$

Exemple (b)

`file.o`

- *imports* $= \{$`printf`$\}$
- *exports* $= \{$`f`, `x`$\}$

`main.o`

- *imports* $= \{$`f`, `x`$\}$
- *exports* $= \{$`main`$\}$

`crt0.o`

- *imports* $= \{$`main`$\}$
- *exports* $= \{$`_start`$\}$

`libc.a` / `printf.o`

- *imports* $= \{...\}$
- *exports* $=$ $\{..., $`printf`$, ...\}$

4. `x` $\in$ *Exps*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`$\}$
   Imps $= \{$`printf`$\}$
   Exps $= \{$`main`, `f`, `x`, `_start`$\}$
5. `printf` $\notin$ *Exps* $\Longrightarrow$ `printf.o` *added*
   Objs $=$ $\{$`crt0.o`, `main.o`, `file.o`, `printf.o`$\}$
   Imps $= \{\}$
   Exps $=$ $\{$`main`, `f`, `x`, `_start`, `printf`$\}$

Le script d'édition des liens

Comportement par défaut :

- architecture, système, endianness – fichiers objets (identique pour tous)
- adresse de démarrage = `_start`
- section = concaténation des sections des objets liés
- affectation d'une adresse à chaque section (en suivant les contraintes d'alignement)
- relocation en utilisant les sections `.rel.XXX` et `.rela.XXX` (qui disparaissent)

Exemple

Objet	.text	.data	.bss
crt0.o	540	a0	120
main.o	f00	b00	1800
file.o	a00	100	100
printf.o	400	100	a0

Tailles en hexadécimal et en octet.

Contraintes d'alignement : 4 KO (0x1000) pour chaque section.

Adresse	section
0000 0000	crt0.o (.text)
0000 0540	main.o (.text)
0000 1440	file.o (.text)
0000 1e40	printf.o (.text)
0000 2240	fin de section .text
0000 3000	crt0.o (.data)
0000 30a0	main.o (.data)
0000 3ba0	file.o (.data)
0000 3ca0	printf.o (.data)
0000 3cb0	fin de section .data
0000 4000	crt0.o (.bss)
0000 4120	main.o (.bss)
0000 5920	file.o (.bss)
0000 5a20	printf.o (.bss)
0000 5ac0	fin de section .bss
0000 5ac0	(mémoire dynamique)

Utilisation d'un script

Édition de lien spécialisée :

```
SECTIONS {
  .text : { *(.text) }
  .data : { *(.data) }
  .bss  : { *(.bss) }
}
```

Syntaxe :

- *script* $\rightarrow$ `SECTIONS {`*section**`}`
- *section* $\rightarrow$ *name*`: {`*contenu**`}`
- *contenu* $\rightarrow$ $ER_{fichier}$(*section**)

Exercice 1

Objet	.text	.data	.bss
crt0.o	200	20	f0
main.o	b00	400	1000
file1.o	200	500	2000
file2.o	200	500	a000

Calculez l'adresse des sections pour le script ci-dessous.

```
SECTIONS {
   .text : { *(.text) }
   .data : { *(.data .bss) }
}
```

Au niveau compilateur

⇒ remplir les sections

- fonctions ⇒ `.text`
- données constantes / chaînes de caractère ⇒ `.rodata`
- variables locales initialisées ⇒ `.data`
- variables locales non-initialisées ⇒ `.bss`

⇒ génération de la table des symboles (`.symtab`) :

- identificateur fonction / variable
- section
- déplacement

⇒ variables locales (pile)

- taille de bloc d'activation
- déplacement dans le bloc (locale)
- déplacement avant le bloc (paramètres)

Exercice 2

Nous voulons écrire un script pour lier des fichiers `.o` composés des sections `.text`, `.rodata`, `.data` et `.bss`.

1. Écrire le script pour que chacune des sections soient liées dans une section de l'exécutable correspondant : les `.text` dans `.text`, etc.
2. Modifiez le script pour que `.text` et `.rodata` soient dans la même section `.text` (l'un juste après l'autre).
3. Modifiez le script pour créer une section `.start` contenant le `.text` et le `.rodata` des fichiers `.o` préfixés par `start`.

Assembleur et édition des liens

Choisir une section :

`.text` – constant + exécutable

`.data` – variable

`.lcomm` *nom*, *longueur* – dans la `.bss`

`.section` *nom*[, "*options*"]

Travailler avec les symboles :

`.global` *noms* – symboles visibles à l'extérieur

`.extern` *noms* – symboles définis dans un autre fichier

`.size` *nom*, *taille* – définition de taille de symbole

Autres commandes :

`.align` *alignement*

`.file` *fichier*, *ligne*

Compilateur et édition des liens

`extern`, `static` – propriétés de symbole

Attributes de GCC : `__attribute__` (*attribut*)

- `aligned` (*alignement*)
- `section` ("*name*")
  - `weak` – accepte des doublons

Commandes

Compilation et édition des liens :

`cc -o` *excutable* *objets* `-l`lib_1 `-l`lib_2 `...`

Bibliothèque statique = collection d'objets

- Unix – `lib`*XXX*`.a`
  `ar rcs lib`*XXX*`.a` *objets*
- Windows – *XXX*`.lib`
  `lib -out:`*XXX*`.lib` *objets*

Options de `gcc`

-T *script* – script d'édition ds liens

-Xlinker *option* – passage d'option à l'éditeur de lien

L'éditeur de lien `ld`

Options :

- `-e` *nom* – symbole de démarrage
- `-l` *bibliotheque*
- `-L` *chemin* pour retrouver les bibliothèques
- `-T` *script*
- `-T` *section=adresse* – fixe l'adresse d'une section (`.text`, `data` ou `.bss`)

Plan

La MMU

Isolation :

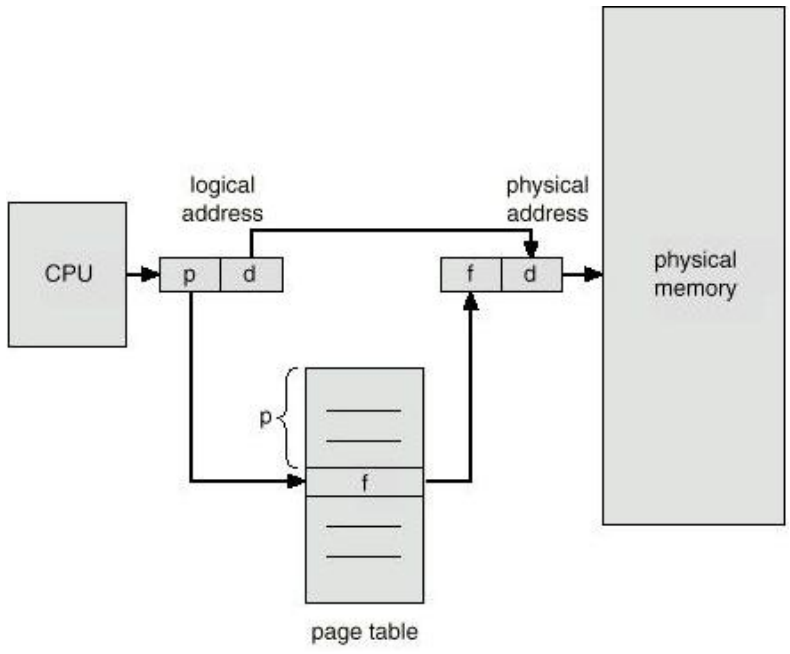
- 1 processus = 1 espace d'adressage virtuel
- protection des erreurs des autres processus

Mémoire virtuelle :

- moins de contrainte pour le programme
- stockage sur le disque dur

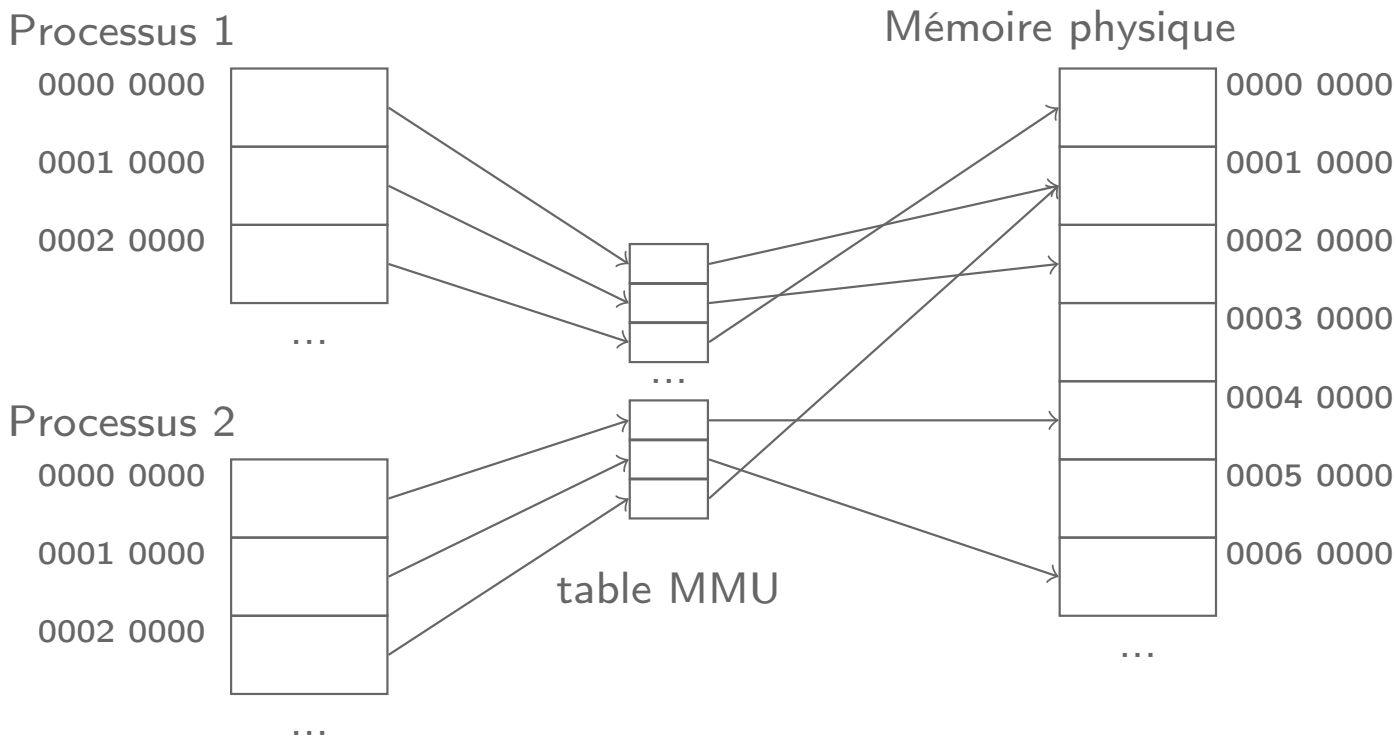
Mémoire physique :

@ physique $\neq$ @ virtuelle



La MMU

MMU = *Memory Management Unit*



MMU et système d'exploitation

Interruptions lors de l'accès à une page :

- page non-exécutable $\Rightarrow$ arrêt
- page non-modifiable
  - résultat d'un `fork` $\Rightarrow$ duplication (Unix)
  - sinon arrêt
- page indisponible
  - zone privée $\Rightarrow$ arrêt
  - mémoire dynamique $\Rightarrow$ allocation réelle d'une page physique
  - page sauvegardée sur le disque $\Rightarrow$ allocation page physique + rechargement $\Rightarrow$ déchargement d'une autre page
  - page correspondant à un fichier $\Rightarrow$ allocation page physique + chargement $\Rightarrow$ déchargement d'une autre page

Processus de chargement avec MMU

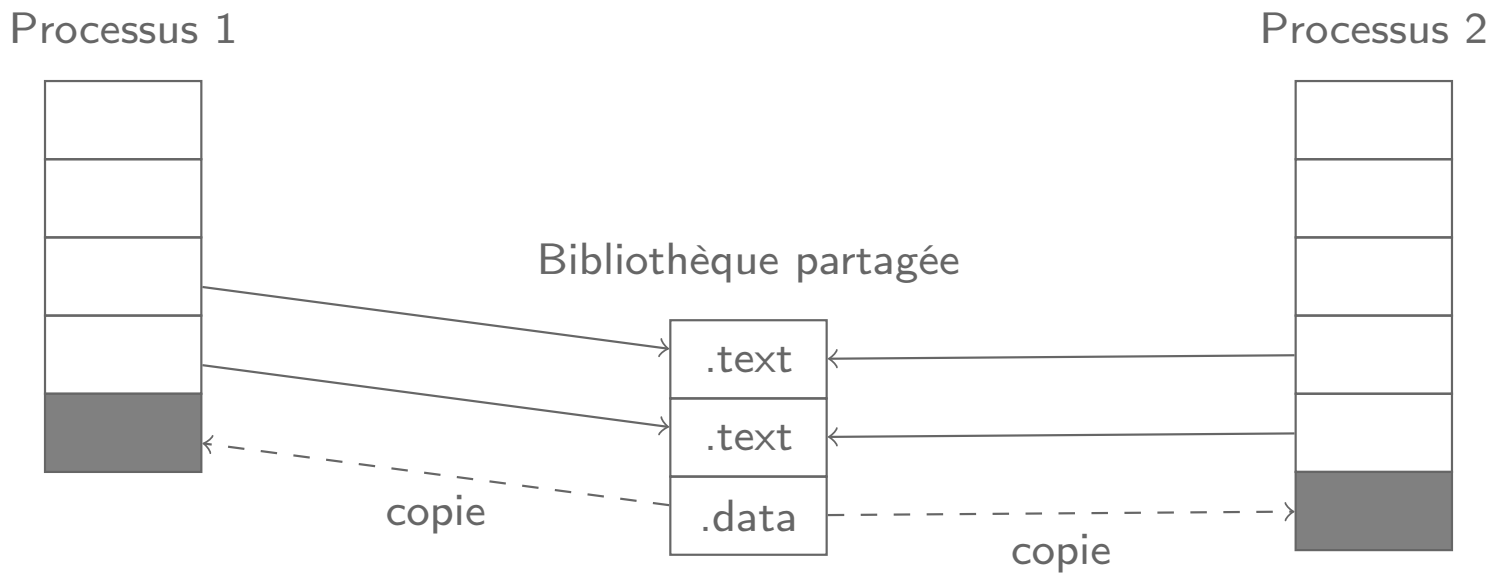
Le système d'exploitation :

1. crée un descripteur de processus contenant la correspondance entre pages virtuelles/physiques
2. alloue des pages pour stocker les segments
3. charge les segments du disque dans les pages mémoire
4. met à 0 les données non-initialisées
5. alloue des pages mémoire pour stocker la pile
6. initialise SP et la pile avec des informations systèmes (arguments de commande)
7. lance le programme à l'adresse de démarrage

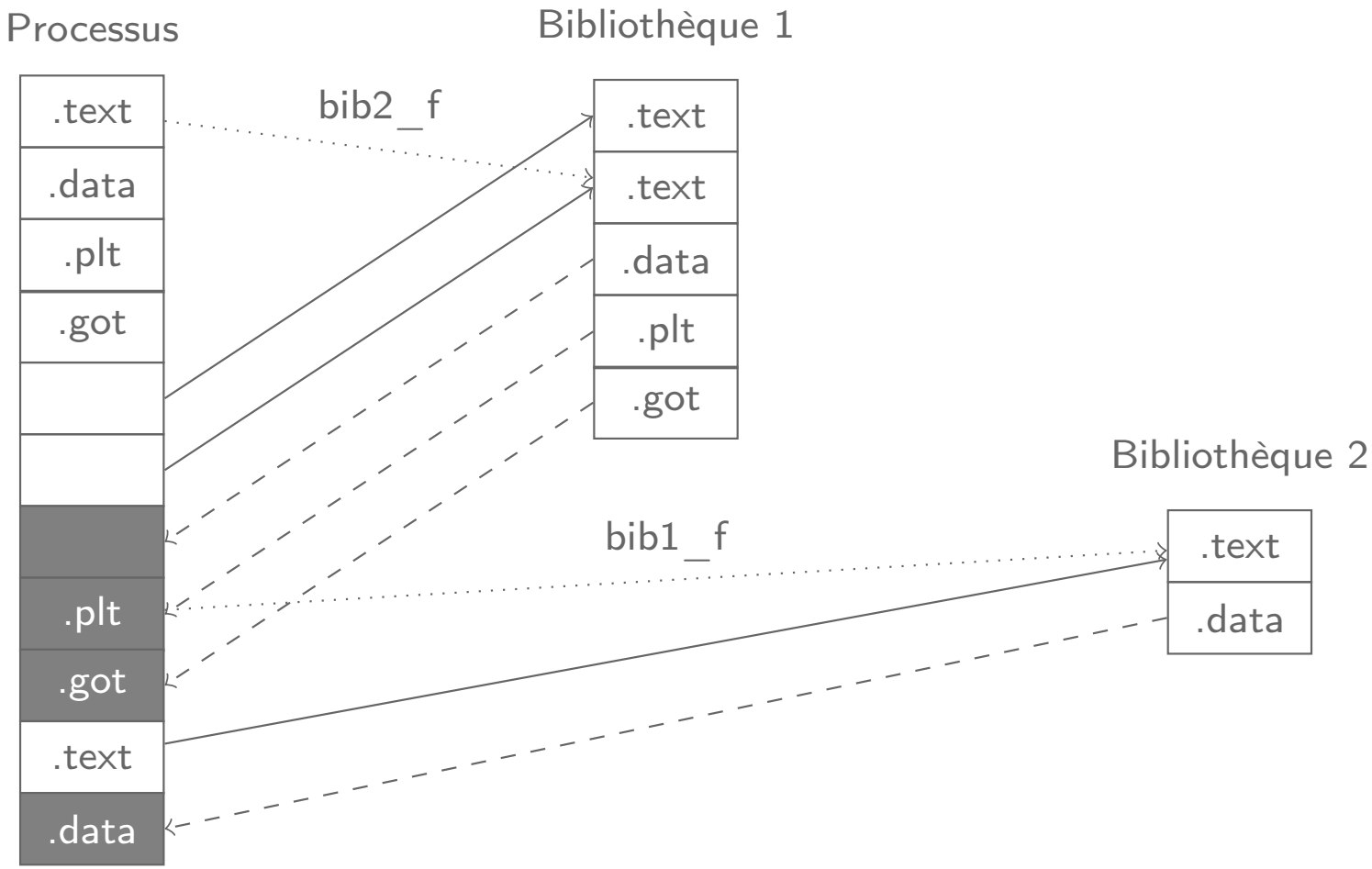
Principe

- édition des liens au chargement du programme
- déjà en mémoire $\Longrightarrow$ partage des pages $\Longrightarrow$ réduction de la place mémoire
- pas en mémoire $\Longrightarrow$ recherche et chargement `/lib`, `usr/lib` ou dans `$LD_LIBRARY_PATH`
- exécutable
  - `.dynamic` – information pour l'édition des liens
  - édition des liens avec `-lXXX`
- bibliothèque partagée (`lib`*XXX*`.so`)
  - compilée avec `-shared`
  - compilée avec `-fPIC` (*Position Independent Code*)
  - symboles dans `.symtab`

Partage des pages



Relocation dynamique



PLT = *Procedure Linkage Table*

GOT = *Global Offset Table*

Programmer la MMU

Bibliothèque standard :

- `mmap` – allocation de page
- `munmap` – désallocation de page

Bibliothèque `dl` (*dynamic linkage*) :

- `dlopen` (*chemin*, *options*)
- `dlclose` (*handle*)
- `dlsym` (*handle*, *nom*)
- `dlinfo` (*handle*, *ID*, *pointer*)

Options de `dlopen` :

- `LAZY` édition de lien paresseuse
- `NOW` édition de lien immédiate
- `GLOBAL` symboles de la biblothèque publics
- `LOCAL` symboles de la bibliothèque privés

Commandes Unix

- Compilation d'une bibliothèque dynamique :
  `cc -shared -o lib`*XXX*`.so` *objets*
- Objets compilés avec `-fpic` $\Rightarrow$ relocation dynamique
- `ld.config` – chargement de programme et édition des liens dynamique (Unix)
- `ldd [-r -d]` *executable* – vérification d'édition des liens
- `objdump -T` *executable* – liste des symboles dynamiques
- `readelf -d` *executable* – information d'édition des liens dynamiques

Bilan

- nécessaire pour économiser de la mémoire
- coût
  - code un peu moins efficace
  - table `.plt` et `.got`
  - édition de lien à l'exécution
    $\Longrightarrow$ liaison paresseuse
- autres gains
  - taille des fichiers exécutables
  - mise à jour des bibliothèques indépendantes
  - mise en œuvre de greffon
    bibliothèque `dl` : `dlopen()`, `dlsym()`

Plan

Caractéristiques

Espace d'adressage non-uniforme :

- RAM (*Random Access Memory*) – modifiable, perdu au redémarrage
  $\Longrightarrow$ variables globales + tas + pile
- ROM (*Read-Only Memory*) – difficilement modifiable mais survit à un redémarrage (FLASH)
  $\Longrightarrow$ code + données *read-only*
- IO (*Intput / Output*) – registres réservés pour les entrées / sorties

$\Longrightarrow$ très différent d'une machine à l'autre

NOTE : s'applique également au noyau d'un système d'exploitation

Description de la mémoire dans un script

```
MEMORY {
  ROM (RIX): ORIGIN = 0, LENGTH = 3M
  RAM (RW): ORIGIN = 0x10000000, LENGTH = 1M
}
```

Mémoire définie par :

- nom
- `ORIGIN` – adresse de base
- `LENGTH` – taille
- attributes (`R`, `W`, `X`, `A`, `I`)

Script avec désignation de la mémoire

```
SECTIONS {
  .text : { *(.text) } > ROM
  .data : { *(.data) } > RAM
  .bss : { *(.bss) } > RAM
}
```

Problème : au démarrage, la RAM est vide $\Rightarrow$ `.data` n'est pas initialisée.
Solution :

1. installer la `.data` dans la ROM (éditeur de lien)
2. copie la `.data` de la ROM dans la RAM (application embarquée)

Concept :

VMA (*Virtual Memory Address*) adresse virtuelle (adresse logique en fonctionnement)

LMA (*Load Memory Address*) adresse de chargement

Chargement de .data en ROM

```
SECTIONS {
  .text.start : {
    startup.o(.text )
  } > ROM
  .text : {
    *(.text)
    _TEXT_END_ = .;
  } > ROM
  .data : {
    _DATA_START_ = .;
    *(.data)
    _DATA_END_ = .;
  } > RAM AT(_TEXT_END_)

}
```

Manipulation des adresses

- *nom* = *addr* ; – définition d'une adresse nommée (+=, -=, etc)
- . – adresse courante
- constante – décimal, **0x** hexadécimal, suffixé par **K**, **M**,etc
- *nom* – référence à une autre adresse
- $addr_1$ *op* $addr_2$ avec *op* $\in \{+, -, *, /, \%, <<, >>, ...\}$
- **ADDR**(*section*) – adresse VMA d'une section
- **LOADADDR**(*section*) – adresse LMA d'une section
- **SIZEOF**(*section*) – taille d'une section
- **ALIGN**(*alignement*) – adresse courante alignée
- **NEXT**(*alignement*) – adresse suivante alignée et non-allouée

Les définitions d'adresse peuvent être réalisées n'importe où dans la définition de section !

Construire l'image au démarrage

Problème : les variables globales ne sont pas initialisées !
Solution :

- copier `.data` de la ROM à la RAM au démarrage
- adresse de début dans la RAM – `_DATA_START_`
- adresse de début dans la ROM – `_TEXT_END_`

```
extern char _DATA_START_, _DATA_END_;

void init() {
  memcpy(
    &_DATA_START_,
    &_TEXT_END_,
    &_DATA_END_ - &_DATA_START_);
}
```

Exercice 3

1. Modifiez le script initial pour prendre en compte les sections `.bss`.
2. Ecrivez le code qui permet de mettre à 0 le code de la section `.bss`.

Exercice 4

Dans certains contextes de fonctionnement (spatial, avionique), un programme peut être altéré par des perturbations extérieures (SEU – *Single Event Upset* et il est dangereux de laisser un tel programme fonctionner. Pour détecter les erreurs, on réalise une somme de contrôle (la somme des mots composant le programme) et re-calculons cette somme régulièrement. Si la somme change, le programme est altéré et on l'arrête proprement.
Fournir :

1. un script d'édition des liens
2. la fonction de calcul de somme de contrôle, compute_sum()

pour prendre en compte ce problème.

Générer du contenu

Dans le contenu d'une section :

- `BYTE` (*entier*8 – *bit*)
- `SHORT` (*entier*16 – *bit*)
- `LONG` (*entier*32 – *bit*)
- `QUAD` (*entier*64 – *bit*)
- `FILL` (*octets*) – remplissage des parties non-spécifiées

Exemple 1 : vecteurs d'initialisation

Table de vecteurs d'initialisation :

```
SECTIONS {
  .init : {

    init_size = .;
    LONG(
      (_INIT_END_-_INIT_BEGIN_)/4);

    init_tab = .;
    _INIT_BEGIN_ = .;
    *(.init)
    _INIT_END_ = .;
    LONG(0)
  }
  ...
}
```

Exploitation du tableau :

Ou alors :

Exemple 1 : vecteurs d'initialisation

Table de vecteurs d'initialisation :

```
SECTIONS {
  .init : {

    init_size = .;
    LONG(
      (_INIT_END_-_INIT_BEGIN_)/4);

    init_tab = .;
    _INIT_BEGIN_ = .;
    *(.init)
    _INIT_END_ = .;
    LONG(0)
  }
  ...
}
```

Exploitation du tableau :

```
typdef void (*init_t)();
extern int init__size;
extern init_t init_tab);

for(i = 0; i < init_size;
    i++)
  init_tab[i]();
```

Ou alors :

Exemple 1 : vecteurs d'initialisation

Table de vecteurs d'initialisation :

```
SECTIONS {
  .init : {

    init_size = .;
    LONG(
      (_INIT_END_-_INIT_BEGIN_)/4);

    init_tab = .;
    _INIT_BEGIN_ = .;
    *(.init)
    _INIT_END_ = .;
    LONG(0)
  }
  ...
}
```

Exploitation du tableau :

```
typdef void (*init_t)();
extern int init__size;
extern init_t init_tab);

for(i = 0; i < init_size;
i++)
  init_tab[i]();
```

Ou alors :

```
for(i = 0; init_tab[i] != NULL;
i++)
  init_tab[i]();
```

Exemple 2 : objets globaux en C++

```
class MyClass {
   ...
   MyClass(int x)
   ...
}

MyClass glob(1024);
```

section .ctor :

```
   adr R0, glob
   mov R1, #1024
   bl  MyClass_MyClass_int
```

section .dtor :

```
   adr R0, glob
   bl  MyClass_~MyClass_int
```

Exemple 2 : objets globaux en C++

```
class MyClass {
   ...
   MyClass(int x)
   ...
}

MyClass glob(1024);
```

section .ctor :

```
   adr R0, glob
   mov R1, #1024
   bl  MyClass_MyClass_int
```

section .dtor :

```
   adr R0, glob
   bl  MyClass_~MyClass_int
```

En C++, code de construction et de destruction des objets globaux :

```
SECTIONS {
   .ctors: {
      _INIT_ = .;
      *(.ctors)
      LONG(0xE12FFF1E);
   }

   .dtors: {
      _FINI_ = .;
      *(.dtors)
      LONG(0xE12FFF1E);
   }
```

Exercice 5

Nous voulons réaliser un système embarqué composé de modules. Selon les cibles du système embarqués, certains modules seront utilisés et d'autres pas. Chaque module est identifié par la structure de donnée, ci-dessous, contenue dans les sources du module :

```
typedef struct module_t {
  const char *name;
  uint32_t version;
  void (*init)();
  void (*fini)();
} module_t;
```

Afin de gérer les modules de manière flexible, on va utiliser l'éditeur de lien pour générer une table des modules se terminant par une structure dont l'adresse du nom est 0. Indiquez :

1. comment déclarer les modules dans les sources,
2. le script d'édition de lien,
3. la fonction d'initialisation des modules.

Plan

Conclusion

L'édition des liens :

- assemble les objets constituant un programme
- construit l'image mémoire du programme
- libère le compilateur de ces questions

La tâche du compilateur devient :

- la génération du code et des données
- le remplissage des sections correspondant aux objets compilés