Systèmes d'Exploitation Avancés

Pablo Oliveira [pablo.oliveira@uvsq.fr]

ISTY

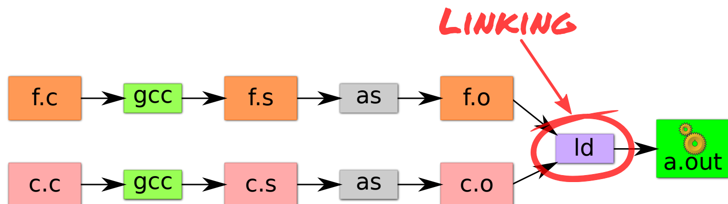
Programme vs. Processus

- Programme : entité passive
  - Liste d'instructions + données statiques
- Processus : instance d'un programme en exécution
  - Liste d'instructions
  - Contexte d'exécution

Anatomie d'un Programme

- Plusieurs formats de programmes (ELF, a.out)
- Le plus utilisé sous Linux est ELF Executable and Linkable Format
- ELF permet de définir différents segments :
  - Segment programme (lecture seule)
  - Segment données (lecture/écriture)
- ELF permet de définir l'adresse où les segments seront chargés

Édition de liens



- Comment nommer et trouver des objets qui n'existent pas encore
- Comment combiner plusieurs espaces de noms
- Plus d'information
  - Le standard ELF
  - Exécuter "`nm`," "`objdump`," et "`readelf`" sur des fichiers .o et a.out.

Système de nommage

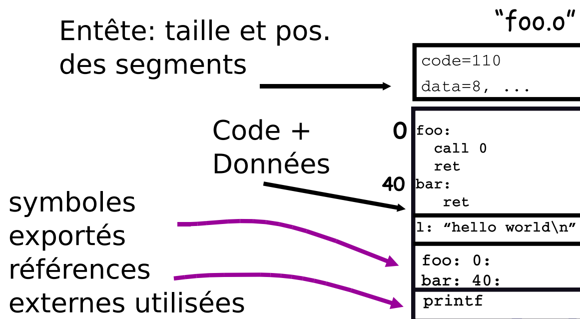
- Le nommage est un problème récurrent en informatique.
- Associer des clés à des valeurs
- Exemples :
  - Linking : Où est `printf` ? Comment s'y référer ? Que faire en cas d'homonyme ? S'il est absent ?
  - Adresse virtuelle (clé) traduite en adresse physique (valeur)
  - Système Fichiers : traduire un chemin en position sur le disque ...

Nommer des objets en mémoire

- Vue du programmeur : `x += 1;` `add $1, %eax`
  - Instructions : opérations à effectuer
  - Variables : opérandes qui changent au cours du temps
  - Constantes : opérandes qui ne changent pas
- Vue de la machine :
  - exécutable : code, d'habitude en lecture seule
  - read only : constantes (copie peut-être partagée)
  - read/write : variables (copie par processus)
- Besoin *d'adresses* pour accéder aux données :
  - Adresses localisent les objets. Elles changent si les objets sont déplacés.
- Édition de liens : Quand est ce qu'une adresse est résolue ?
  - à la compilation ? au link ? au chargement ? durant l'exécution ?

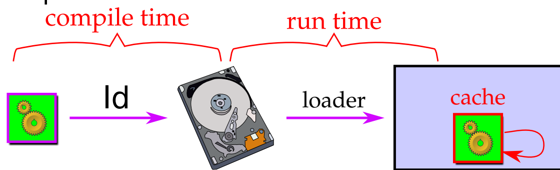
Composition d'un exécutable

- Exécutable : interface linker/SE
  - C'est quoi du Code ? des Données ?
  - Où faut-il les placer ?
- Linker fabrique des exe. à partir de .o



Exécution d'un programme ?

- Unix : exe. lu par le "loader"



  - Charge les segments de code/données dans le cache disque (buffer cache)
  - Projette les segments de code en lecture seule dans l'espace d'adressage
  - Projette les segments de données en lecture/écriture dans l'espace d'adressage

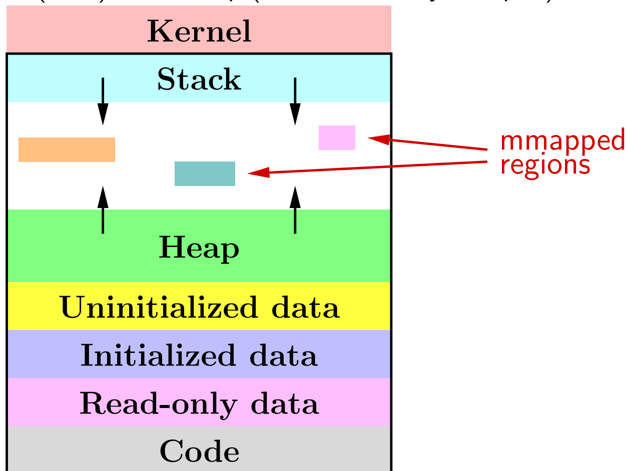
- Nombreuses optimisation
  - Segments initialisées à zéro ne sont pas alloués dans l'exe
  - Chargement à la demande : pages marquées comme invalides
  - Plusieurs copies du même processus : segments de code partagés

Bibliothèque Statiques ou Dynamiques

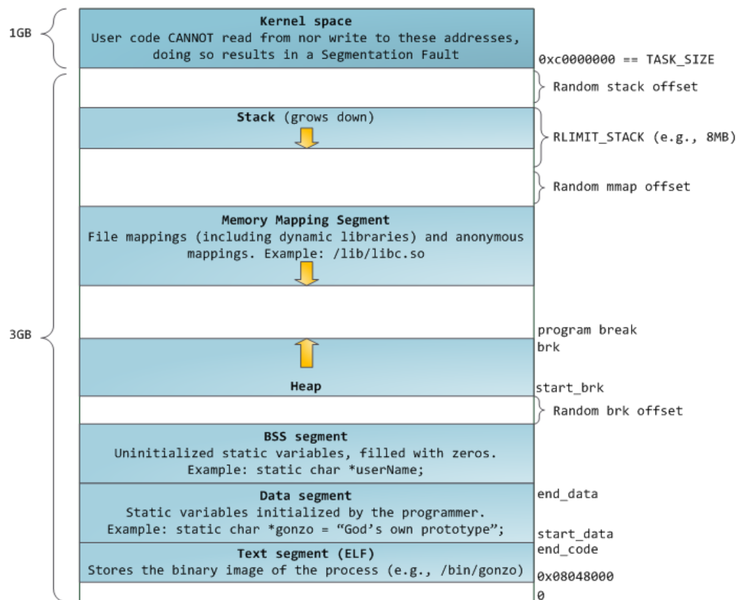
- Lien *statique* : le code de la bibliothèque est compilé dans l'exécutable même
  - pas de dépendances, le programme est autonome
- Lien *dynamique* : le code de la bibliothèque est chargé à l'exécution
  - plusieurs programmes partagent le même code bibliothèque
  - on peut corriger un bug dans une bibliothèque sans recompiler tout le système

Anatomie d'un processus (Unix)

- Espace d'adressage divisé en segments
  - text (code), data, heap (tas : données dynamiques), et stack (pile)



Anatomie d'un Processus (unix)



Gustavo Duarte

Segments

- Text : contient les instructions du programme
- Data + BSS : contient la mémoire allouée statiquement
- Pile : contient les variables locales, les adresses de retour et certains tableaux (cf. `alloca`).
- Tas : contient la mémoire allouée dynamiquement (cf. `malloc`).
- Mmap : mémoire allouée avec `mmap`, mapping de fichier vers la mémoire

Mémoire virtuelle

- Chaque processus a l'impression de pouvoir adresser toute la mémoire :
  - Simplifie l'adressage
  - Pas de collision avec la mémoire d'autres processus
- Mémoire virtuelle :
  - Pour chaque processus, le SE conserve une table qui traduit les adresses virtuelles (vues par le processus) en adresses physiques (vues par la mémoire).
- Plus de détails dans le cours sur la gestion de la mémoire

Qui alloue et positionne quoi en mémoire ?

- Tas : alloué et positionné par malloc à l'exécution
  - Compilateur/Linkeur : non concernés, mais réservent une zone
  - Adressage dynamique et géré par le programmeur (manipulation de pointeurs)
- Pile : alloué à l'exécution (appel de fct. / alloca), positionné par le compilateur
  - Adresses relatives au pointeur de pile/frame
  - Géré par du code généré par le compilateur
  - Linker non concerné : espace local à une fonction
- Code/Données statiques globales : allouées par le compilateur, positionnées par le linker
  - Compilateur crée des zones et leur donne des noms symboliques
  - Le linker décide où placer les zones et remplace les réf. symboliques par des adresses

Exemple

- Programme : “`printf ("hello world\n");`”
- Compilé avec : `gcc -m32 -fno-builtin -S hello.c`
  - `-S` génère du code assembleur (`-m32` 32-bit x86)
- Sortie dans `hello.s` avec réf. symbolique à `printf`

```
        .section        .rodata
.LC0:   .string "hello world\n"
        .text
.globl main
main:   ...
        subl    $4, %esp
        movl    $.LC0, (%esp)
        call    printf
```

- Désassembler un fichier `.o` avec `objdump -d` :

```
18: e8 fc ff ff ff      call 19 <main+0x19>
```

  - Saute à PC - 4 = au milieu de l'instruction courante ? ? ?

Linker (édition de liens)

- Unix : ld
  - Appelé automatiquement par le compilateur
- ld a trois responsabilités :
  - Rassembler en un seul exécutable toutes les pièces d'un programme
  - Fusionner les segments de même type
  - Remplacer les adresses symboliques par des vraies adresses
- Résultat : un programme exécutable
- Pourquoi le compilateur ne le fait pas directement ?
  - Vue limitée à un seul fichier (pas de connaissance des symboles des autres fichiers)
- Certains linkers peuvent réorganiser l'ordre des segments
  - E.g., réordonner les segments pour avoir moins de miss
    enlever des morceaux de code morts jamais appelés

Linker simple : deux passes

- Passe 1 :
  - Fusionner les segments de même type. Deux objets ne peuvent pas être au même endroit en mémoire.
  - Lecture des tables de symboles. Construction d'une table globale.
  - Calcul des débuts d'adresses virtuelles de chaque segment.
- Passe 2 :
  - Patche les références symboliques à l'aide la table globale.
  - Création du .exe final
- Table des symboles
  - Segments : nom, taille, position
  - Symboles : nom, segment, position à l'intérieur du segment

Comment les objets sont ils crées ?

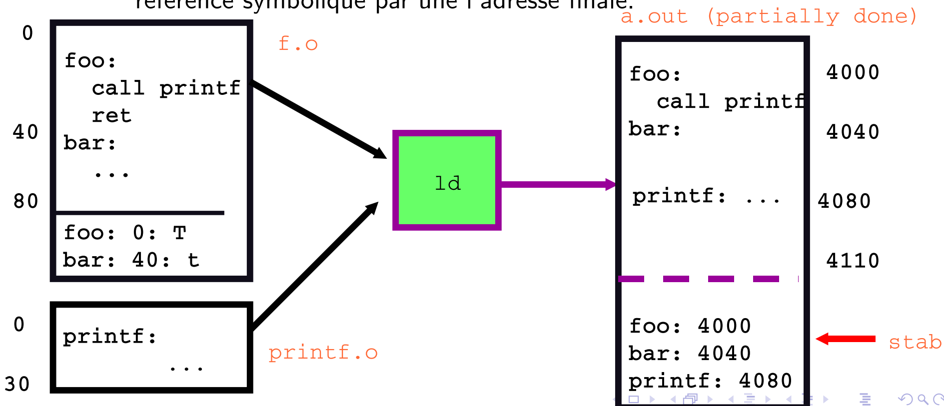
- Assembleur :
  - Ne sait pas encore où placer les objets dans l'espace d'adressage global
  - Suppose que l'objet courant commence à l'adresse zéro
  - Crée une table de symboles qui contient un nom et la position de chaque object
  - Certains symboles sont étiquetés comme définitions globales

```
0  foo:
      call printf
      ret
40 bar:
      ...
      ret
```

foo : 0 : T
bar : 40 : t

Où placer les objets crées en mémoire ?

- Lors de la phase d'édition de liens
  - Détermine l'ordre des segments
  - Détermine la taille de chaque segment et l'adresse de chaque objet
  - Crée une table globale des symboles qui permet de remplacer chaque référence symbolique par une l'adresse finale.



Résolution des références symboliques

- Comment remplacer par les vraies adresses ?
  - E.g., un appel à `printf` nécessite l'adresse destination
  - Assembleur utilise une adresse factice
  - Crée une référence symbolique qui permet au linker de patcher le saut plus tard

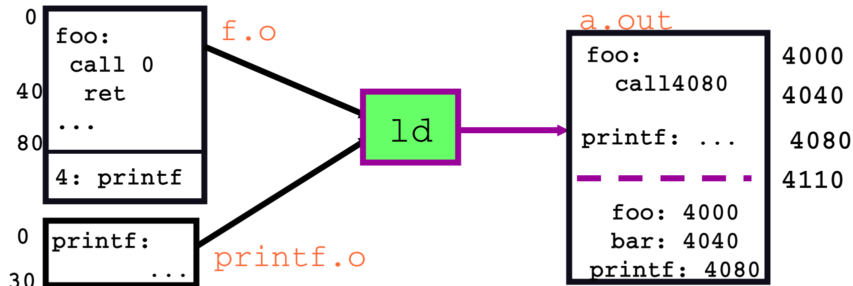
```
0   foo:
      pushl $.LC0
4     call -4
      ret
40  bar:
      ...
      ret
```

foo : 0 : T
bar : 40 : t
printf : 4

- Lors de l'édition de liens, chaque référence est patchée

Linker : Résolution des liens

- Le linker
  - Vérifie que chaque symbole est défini exactement une fois
  - Cherche chaque référence et remplace les adresses factices par les vraies adresses qu'il vient de calculer.



Exemple : 2 modules et la bibliothèque C

main.c

```
extern float sin();
extern int printf(), scanf();
float val = 0.0;
int main() {
  static float x = 0.0;
  printf("enter␣number:␣");
  scanf("%f", &x);
  printf("Sine␣is␣%f\n", val);
}
```

math.c

```
float sin(float x) {
  float tmp1, tmp2;
  static float res = 0.0;
  static float lastx = 0.0;
  if (x != lastx) {
    lastx = x;
    /* compute sin(x) */
  }
  return res;
}
```

libc

```
int scanf(char *fmt, ...) { /* ... */ }
int printf(char *fmt, ...) { /* ... */ }
```

Fichiers objets

Main.o:

```
def: val @ 0:D        symbols
def: main @ 0:T
def: x @ 4:d
```

```
                     relocation
ref: printf @ 8:T,12:T
ref: scanf @ 4:T
ref: x @ 4:T, 8:T
ref: sin @ ?:T
ref: val @ ?:T, ?:T
```

```
0   x:
4   val:        data
```

```
0   call printf
4   call scanf(&x)
8   val = call sin(x)     text
12  call printf(val)
```

Math.o:

```
                symbols
def: sin @0:T
def: res @ 0:d
def: lastx @4:d
```

```
              relocation
ref: lastx@0:T,4:T
ref res @24:T
```

```
0   res:              data
4   lastx:
```

```
0   if(x != lastx)
4       lastx = x;    text
…       … compute sin(x)…
24  return res;
```

Passe 1 : Réorganisation des segments

a.out:

	symbol table
0	val:
4	x:
8	res:
12	lastx:
16	main:
…	…
26	call printf(val)
30	sin:
…	…
50	return res; text
64	printf: …
80	scanf: …

Starting virtual addr: 4000

```
Symbol table:
    data starts @ 0
    text starts @ 16
    def: val @ 0
    def: x @ 4
    def: res @ 8
    def: main @ 16
    …
    ref: printf @ 26
    ref: res @ 50
    …
```

(what are some other refs?)

Passe 2 : Résolution des adresses

“final” a.out:

Starting virtual addr: 4000

	symbol table	
0	val:	4000
4	x:	4004
8	res:	4008
12	lastx: data	4012
16	main:	4016
		…
26	call ??(??)//printf(val)	4026
30	sin: text	4030
		…
50	return load ??;// res	4050
64	printf: …	4064
80	scanf: …	4080

Symbol table:

```
data starts 4000
text starts 4016
def: val @ 0
def: x @ 4
def: res @ 8
def: main @ 14
def: sin @ 30
def: printf @ 64
def: scanf @80
…
```

(usually don’t keep refs, since won’t relink. Defs are for debugger: can be stripped out)

Écriture finale

a.out:

	symbol table	
16	main: Text segment	4016
26	call 4064(4000)	4026
30	sin:	4030
50	return load 4008;	4050
64	printf:	4064
80	scanf:	4080
1000	Data segment	5000
	val: 0.0	
	x: 0.0	
	...	

virtual addr: 4016

Symbol table:

```
initialized data = 4000
uninitialized data = 4000
text = 4016
def: val @ 1000
def: x @ 1004
def: res @ 1008
def: main @ 14
def: sin @ 30
def: printf @ 64
def: scanf @ 80
```

Examiner les programmes avec nm

```
int uninitialized;
int initialized = 1;
const int constant = 2;
int main ()
{
  return 0;
}
```

```
$ nm a.out
...
0400400 T _start
04005bc R constant
0601008 W data_start
0601020 D initialized
04004b8 T main
0601028 B uninitialized
```

VA — symbol type

- const variables R en lecture seule
  - Choix d'une adresse constant sur la même page que main
  - Partage de pages en lecture seule
- Données à zéro dans le segment "BSS", B
  - Pas de place réservée dans le .exe
  - Des pages remplies de 0 sont allouées par le SE à la demande

Examiner des programmes avec objdump

LMA. et File off ont
le même aligne :ment de page

```
$ objdump -h a.out
a.out:     file format elf64-x86-64
Sections:
Idx Name     Size      VMA       LMA       File off  Algn
...
 12 .text    000001a8  00400400  00400400  00000400  2**4
             CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 14 .rodata  00000008  004005b8  004005b8  000005b8  2**2
             CONTENTS, ALLOC, LOAD, READONLY, DATA
...
 17 .ctors   00000010  00600e18  00600e18  00000e18  2**3
             CONTENTS, ALLOC, LOAD, DATA
...
 23 .data    0000001c  00601008  00601008  00001008  2**3
             CONTENTS, ALLOC, LOAD, DATA
...
 24 .bss     0000000c  00601024  00601024  00001024  2**2
             ALLOC
...
```

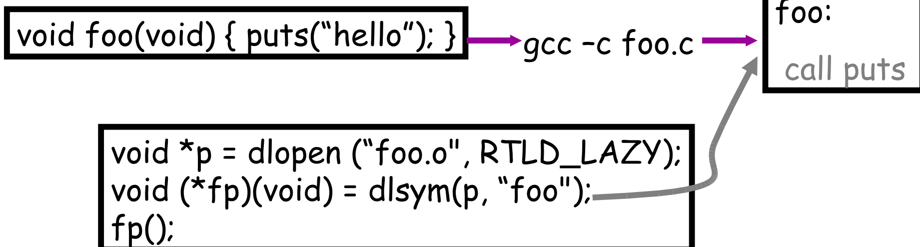
N'occupe pas de place dans le bin

Types de résolution de liens

- Patche avec une adresse absolue
  - Exemple : `int y, *x = &y;`
  - `call printf` devient
    `8048248: e8 e3 09 00 00  call 8048c30 <printf>`
  - L'encodage binaire correspond à l'adresse virtuelle de `printf` (Attention : encodage de l'argument de `call` est relatif au PC)
- Patche avec un offset
  - Utilisé pour les structures
  - Exemple : `struct queue { int type; void *head; } q;`
    `q.head = NULL` → `movl $0, q+4` → `movl $1, 0x804a01c`
- Ajouter la différence entre le segment original (dans le .o) et final.

Variation 0 : Linker dynamique

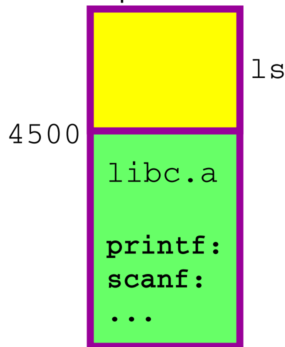
- Pourquoi ne pas linker à l'exécution ?
  - Si un code n'est pas trouvé, on va le chercher
  - Chargement de code *à la demande*

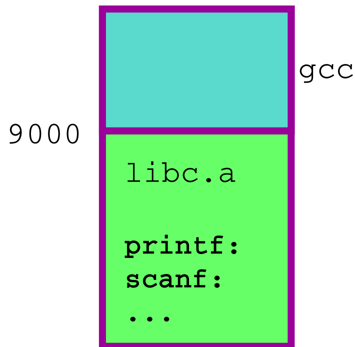


  - Problèmes : Différences par rapport au Linker statique ? Où aller chercher le code manquant ? Que faire si résolution impossible ?

Variation 1 : Bibliothèque partagée statique

- Observation : la plus part des programmes utilisent la bibliothèque statique, libc.a.
  Plein de copies inutiles.

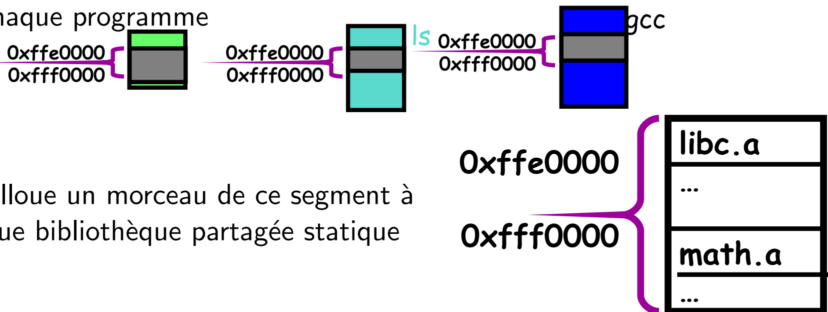




- Idée : une seule copie sur disque inclue dans chaque programme.

Bibliothèque partagée statique

- Définir un segment réservé à une adresse fixe dans l'espace d'adressage de chaque programme



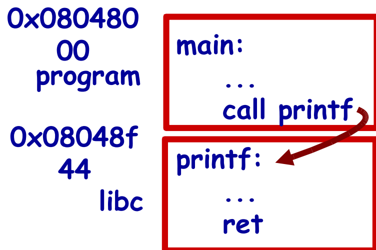
- On alloue un morceau de ce segment à chaque bibliothèque partagée statique
- Le loader marque la région comme invalide
- Si le processus saute dans la région, le SE déclenche un seg fault, qui est récupéré par le loader qui peut charger le code depuis une zone mémoire partagée.
- Plusieurs programmes se partagent le même code.

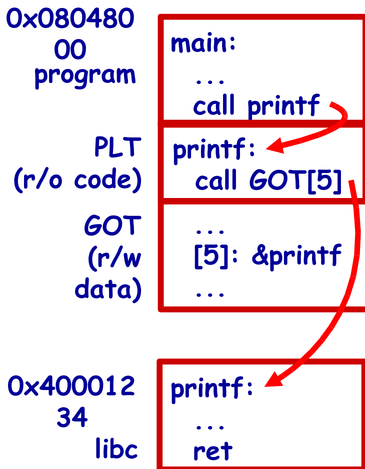
Variation 2 : Bibliothèques partagées dynamiques

- Inconvénient du statique : zone fixe pré-allouée dans tous les processus
  - Espace perdu
  - Et si la librairie dépasse la taille de la zone fixe ?
- Solution : Chargement dynamique de bibliothèques partagées
  - Chargement d'une librairie possible à n'importe quelle adresse
  - Problème : le linker ne sait pas où le code sera chargé ...

PIC : Code position-indépendant

- Le code de la bibliothèque doit pouvoir tourner indépendemment de l'adresse
- Ajouter un niveau d'indirection !





Chargement à la demande

0x08048000 program

```
main:
  ...
  call printf
```

PLT (r/o code)

```
printf:
  call GOT[5]
```

GOT (r/w data)

```
  ...
  [5]: dlfixup
  ...
```

0x40001234 libc

```
printf:
  ...
  ret
```

```
dlfixup:
  GOT[5] = &printf
  call printf
```

- Faire la résolution au chargement est coûteux
- Un programme n'utilise que quelques fonctions
- Résolution faite lors du premier appel

Résumé : Édition de Liens

- Compilateur : 1 fichier objet par fichier source
  - Problème : vue incomplète du monde
  - Solution : utiliser des références symboliques (“`printf`”)
- Linker : combine les .o dans un seul exécutable
  - Vue globale
  - Patche les références symboliques
  - Interface avec le SE, où est le code ? les données ? où commence le programme ?