My Linker

Version 2
05 décembre 2011

Laboratoire Sécurité/Système d'EPITA 2013 <contact@lse.epita.fr>

Copyright

Ce document est destiné à une utilisation interne à EPITA < http://www.epita.fr/>.

Table des matières

Obligations

- ▷ Lire le sujet en *entier*
- ▷ Respecter les règles
- ▷ Respecter l'heure de rendu

Rendu

Responsables du projet	Pierre Bourdon < delroth@lse.epita.fr> Pierre-Marie de Rodat < pmderodat@lse.epita.fr> Nicolas Hureau < nicolas@lse.epita.fr>
Balise du projet :	**[LD]**
Développeurs par équipe:	1
Procédure de rendu :	Git ACU
Rendu :	16/12/2011 à 11h42
Durée du projet :	2 semaines
Groupe de discussion :	iit.labos.lse.recrutement
Architecture/OS :	x86_64 Fedora
Langage(s) :	C++
Compilateur :	**g++**
Options du compilateur :	**-Wall -Wextra -Werror** (au minimum, C++0x est autorisé, de même pour boost)
Includes autorisés :	Tous

Consignes

Les informations suivantes sont très importantes :

Le non-respect d'une des consignes suivantes entraînera des sanctions pouvant aller jusqu'à la multiplication de la note finale par 0.

Ces consignes sont claires, non-ambiguës, et ont un objectif précis. En outre, elles ne sont pas négociables.

N'hésitez pas à demander si vous ne comprenez pas une des règles.

- **Consigne 0:** Le fichier `AUTHORS` doit être présent à la racine de votre rendu. Il comprend une étoile **`*`**, une espace, votre login (ex : login_x) et un retour à la ligne.

  Exemple :

  ```
  42sh$ cat -e AUTHORS
  * login_x$
  42sh$
  ```

  L'absence du fichier `AUTHORS` entrainera un 0 non négociable.
- **Consigne 1:** Lorsque des formats de sortie vous sont donnés en exemple, vous devez les respecter scrupuleusement. Les exemples sont indentés : **`42sh$`** représente le prompt, utilisez le comme repère.
- **Consigne 2:** Le respect de la "coding style" est obligatoire si elle existe dans le langage demandé. Dans tous les cas, le code rendu doit être propre, lisible et ne doit pas dépasser 80 colonnes.
- **Consigne 3:** Les droits sur les fichiers et les dossiers sont obligatoires pour tout le projet : répertoire principal, fichiers, sous répertoires ...
- **Consigne 4:** Si un exécutable est demandé, vous devez uniquement fournir ses sources dans le répertoire src, sauf mention contraire. Ils seront compilés par nos soins.
- **Consigne 5:** Votre rendu doit respecter l'aborescence suivante, où **`login_x`** doit être remplacé par votre login ou celui de votre chef de groupe:

  `login_x-my_ld/AUTHORS *`
  `login_x-my_ld/README *`
  `login_x-my_ld/TODO *`
  `login_x-my_ld/src/ *`

  Les fichiers suivants sont requis :

`TODO`	décrit les tâches à accomplir. Il doit être régulièrement mis à jour.
`AUTHORS`	contient les auteurs du projet. Doit respecter la norme EPITA (spécifiée plus haut).
`README`	décrit le projet et les algorithmes utilisés dans un anglais correct. Explique aussi comment utiliser votre projet.
- **Consigne 6:** Une archive non propre est une archive qui contient des fichiers interdits (**`*~`**, **`*.o`**, **`*.a`**, **`*.so`**, **`*#*`**, **`*core`**, **`*.log`**, binaires, etc.).

  Une archive non propre est aussi une archive dont le contenu n'a pas les bons droits.

  Une archive non propre entraîne automatiquement la perte de deux points sur votre travail.
- **Consigne 7:** Vous devez rendre à l'heure. Tout retard, même d'une seconde, entraînera au mieux la note de 0 non négociable.

- ▷ **Consigne 8:** Toutes les fonctions et commandes qui ne sont pas explicitement autorisées sont interdites. Les abus peuvent entraîner jusqu'à l'obtention de la note, non négociable, de -21.
- ▷ **Consigne 9:** La triche, l'échange de code source, de tests, d'outils de tests et de correction de norme, sont pénalisés par une note, non négociable, de -42.
- ▷ **Conseil:** N'attendez pas la dernière minute pour commencer le projet.

Introduction

Ce projet est l'occasion de montrer que vous êtes aptes à travailler sur des projets internes du LSE. Vous allez aborder de multiples notions qui ont toutes trait à la manipulation de binaires produits par votre compilateur, qu'ils soient statiques ou dynamiques. Vous allez avoir à développer un linker, à savoir, le programme qui intervient dans la dernière phase du processus de compilation pour lier les objets binaires entre eux.

Amusez-vous bien et impressionnez-nous !

Note

L'idéal serait de créer un linker fonctionnant à la fois avec les ELF 32 bits et 64 bits. Mettre en place cette généricité est cependant ardu : chaque chose en son temps ! Vous pourriez commencer par gérer les ELF 64 bits puisque vos racks fonctionnent en 64 bits. Cependant certains d'entre vous vont peut-être utiliser des machines 32 bits pendant le développement, nous vous conseillons donc de commencer par gérer les ELF 32 bits. Pour générer des ELF 32 bits avec `gcc` et `ld`, ajoutez simplement `-m32` à vos lignes de commande, à la manières des exemples dans les prochaines sections.

1 Étape 1 : création d'un ELF et des segments

Le but principal d'un linker statique est de rassembler le contenu de *sections* qui proviennent des fichiers objets en *segments*. Ces segments doivent être placés dans le fichier exécutable final : ils représentent les zones mémoires contenant les données de votre exécutable. Quand vous exécutez votre programme, le noyau charge les segments en mémoire et commence l'exécution du programme.

Pour cette première étape, nous passerons une liste de fichiers objets (`*.o`) en arguments de votre linker. Vous devrez tout d'abord classer les sections par type de permission : chaque type de permission rassemble plusieurs sections au sein d'un même segment. Par exemple, le segment de lecture-écriture rassemble généralement toutes les sections contenant du code exécutable.

Vous devrez ensuite allouer des espaces au sein du fichier de sortie et au sein de l'espace d'adressage. Pour le fichier de sortie, cette étape facilitera l'écriture des différents *program headers* ; pour l'espace d'adressage, il s'agit simplement de spécifier au noyau à quelles adresses en mémoire il faut charger les segments (attention : sous Linux, utiliser la première page de l'espace d'adressage vous posera des problèmes).

Comme on ne peut pas encore savoir à quelle adresse se trouve le symbole `_start` qui symbolise l'*entry point* du programme (la gestion des symboles sera bien sûr ajoutée par la suite), vous pouvez considérer que l'entry point se trouve au tout début du segment contenant le code exécutable.

Si votre code fonctionne correctement, votre linker sera capable à ce point de produire un exécutable fonctionnel à partir d'un objet qui ne référence pas de symboles et dont l'entry point est au début d'une section exécutable. Exemple : le résultat de la compilation de ce code assembleur :

```
$ cat example1.s

.text

# _exit(0)
movl $1, %eax
movl $0, %ebx
int $0x80

$ gcc -m32 -c example1.s
$ ./my_ld -o example1 example1.o
$ readelf -a example1
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - Linux
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1000
  Start of program headers:          52 (bytes into file)
```

```
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00001000 0x00001000 0x0000c 0x0000c R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x00000 0x00000 RW  0x1000

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

No version information found in this file.
```

Évidemment, rien ne vous force à avoir exactement le même fichier ELF que nous, mais vous pouvez vous inspirer de cette sortie de `readelf(1)` en cas de problème.

Conseils

- Les sections à charger en mémoire sont de type `PROGBITS` ou `NOBITS`.
- Plusieurs sections peuvent tout à fait être placées dans le même segment, du moment qu'elles partagent les mêmes permissions (lecture, écriture et exécution).
- Faites bien la distinction entre les contraintes sur l'alignement des segments en mémoire et celles relatives à l'alignement des sections au sein des segments.
- Usez et abusez de `readelf(1)` pour analyser l'ELF que vous produisez et pour le comparer à un ELF généré par `ld(1)` : vous aurez peu d'indications de la part du noyau sur les potentielles erreurs de vos productions lors de vos tentatives d'exécution du binaire.

Bonus À l'image des sorciers du format QMAGIC, tentez de produire l'ELF le plus petit possible en jouant avec l'alignement des segments/sections et le chevauchement des différentes parties de l'ELF.

2 Étape 2 : table des sections

La table des sections n'est pas nécessaire pour un binaire autonome (c'est-à-dire qui n'utilise pas de bibliothèque dynamique). En effet, il suffit simplement de copier des données en mémoire (le contenu des segments) puis de sauter au point d'entrée spécifié par l'ELF. Cependant, un tel binaire ne profite pas des outils d'analyse des ELF reposant sur des sections, or ceux-ci sont très utiles dans le développement d'un linker.

Le but de cette étape est simple : récupérer les noms des sections provenant des fichiers objets en entrée et les lister dans la table des sections de votre exécutable en sortie. Cela peut paraître superflu mais de nombreux outils manipulant les ELF (`objdump`, `nm`, `strip`, ...) se basent sur cette table des sections pour réaliser leurs opérations. Ce sont autant d'outils dont vous pourrez vous servir pour déboguer vos productions !

Un test très simple consiste à lancer `objdump -d` sur le binaire de sortie de votre linker. Un exécutable linké par l'étape 1 ne produira aucune sortie, alors qu'après l'étape 2 le code assembleur de l'exécutable devrait être affiché correctement :

```
example2:     file format elf32-i386


Disassembly of section .text:

 1000:        b8 01 00 00 00              mov    $0x1,%eax
 1005:        bb 02 00 00 00              mov    $0x2,%ebx
 100a:        cd 80                       int    $0x80
```

3 Étape 3 : résolution des symboles

Un des objectifs majeurs du linker, en plus de son rôle de création des segments, est de résoudre les symboles référencés par les fichiers objets d'entrée.

Chaque fichier objet en entrée contient :
- Des symboles définis : ceux-ci référencent des emplacements au sein des sections qui seront chargées en mémoire. Par exemple : le symbole ayant pour nom `_start` représente l'adresse de l'octet 42 de la section `.text`. C'est ce type de symbole qui est généré par votre compilateur C quand vous implémentez une fonction ou une variable globale.
- Des symboles non définis : ceux-ci référencent un emplacement mémoire dont on ne connaît pas la localisation, ni rien d'autre que le nom. Votre compilateur C génère ce type de symbole quand vous utilisez une fonction ou une variable globale dans un module alors qu'il est défini dans un autre module.
- Des symboles locaux : en apparence, c'est inutile. En effet : pourquoi définir un symbole qui ne pourra pas être utilisé par les autres modules ? C'est simple : l'utilisation une variable globale non exportée (`static`) est dépendante de l'emplacement de la-dite variable en mémoire. Le compilateur ne sait pas à quelle adresse aller chercher la valeur de cette variable. Une référence à ce symbole local permet au compilateur de déléguer au linker la tâche de fournir cette adresse !

Résoudre signifie trouver l'adresse mémoire finale du symbole afin d'écrire cette valeur à tous les endroits référençant ce symbole. Le travail d'écriture de la valeur sera effectué lors de la prochaine étape. Ici, on vous demande les choses suivantes :
- Déterminer la valeur finale de tous les symboles se trouvant dans les fichiers objets en entrée. Déterminer l'adresse finale d'un symbole en mémoire est relativement simple : pour un fichier objet, un symbole est une référence à une section associée à un offset au sein du contenu de cette section. La première étape vous a permis de donner une adresse à laquelle charger cette section, il suffit donc d'ajouter cette adresse à l'offset indiqué.
- Signaler une erreur dans le cas où un symbole utilisé n'est défini nulle part.
- De la même manière, signaler les symboles dupliqués (présents dans deux fichiers différents avec le même nom) ;
- Utiliser la valeur du symbole `_start` en tant qu'entry point du programme généré.
- Générer une table des symboles dans votre ELF : ce n'est bien sûr pas nécessaire pour que votre programme fonctionne, mais cela facilitera votre débogage.

Résoudre les symboles ne sert à rien si l'on n'est pas capable d'effectuer les relocations listées dans les fichiers objets. Cependant, il n'est pas possible de s'occuper de ces relocations avant l'étape de résolution des symboles : en effet toutes les relocations s'appuient sur un symbole.

Vous pouvez tester vos résultats de cette étape de plusieurs manières : un fichier objet d'entrée où le point d'entrée n'est pas au début de la section (vous permettant de tester la résolution de l'entry point), ou utiliser `nm` afin d'afficher la table des symboles de votre exécutable de sortie.

```
$ cat example2.s
```

```
    .text
    .globl _start
    .local exit_code

_start:
```

```
    movl    $1, %eax
    movl    exit_code, %ebx
    int     $0x80

    .section .rodata
exit_code:
    .long 3
```

```
$ gcc -m32 -c example1.s
$ ./my_ld -o example1 example1.o
$ nm example1
00001000 T _start
00002000 r exit_code
```

Attention Sans relocation, le code ci-dessus ne fonctionnera pas exactement comme on peut s'y attendre : à la place de `exit_code`, votre assembleur va sûrement placer des zéros en attendant que votre linker y place l'adresse associée à ce symbole.

Résultat : au lieu de charger l'entier 3 (pointé par `exit_code`) dans le registre EBX, le programme va tenter d'aller chercher l'entier à l'adresse 0...et va donc provoquer une erreur de segmentation. Patience, l'étape suivante va permettre de faire fonctionner ce bout de code !

Conseils

- Il sera nécessaire par la suite de pouvoir accéder à un symbole en ne disposant que de la section à laquelle il est associé et son numéro au sein de cette section. Conservez donc à la fois une table des symboles globale et une table locale à chaque section.

4 Étape 4 : gestion des relocations

Lorsque le compilateur produit du code, il ne sait pas à quelle adresse celui-ci va être chargé en mémoire, ni à quelles adresses seront chargées les données, etc. En effet, c'est le linker (vous-même !) qui se charge d'allouer l'espace en mémoire virtuelle.

Pour combler ce manque, le compilateur écrit à la place de l'adresse inconue 0 (ou parfois un offset, s'il connaît la section contenant le symbole visé), et laisse des instructions au linker pour que celui-ci « patche » le code des sections en conséquence. Ces indications sont contenues dans une section (une table des relocations) `rel.*` de type `REL` ou `RELA` ; elles contiennent :

- La position à laquelle s'effectue la substitution au sein de la section concernée.
- Le type de relocation (la manière de calculer l'adresse résultat, la manière de la stocker, ...) ; pour un linker statique, seuls 3 types de substitution sont à gérer, de nombreux autres concernent les linkers dynamiques.
- Le symbole utilisé pour la substitution.
- Un nombre optionnel à rajouter au résulat obtenu.

Pour cette étape, il suffit donc de parcourir les tables des relocations de tous les fichiers objets passés en paramètre et d'appliquer les modifications indiquées.

```
$ cat example3-1.s example3-2.s

# example3-1.s
    .text
    .globl _start
    .extern external_exit
    .globl external_exit_code

_start:
    movl    $1, %eax
    jmp     external_exit

    .section .rodata
external_exit_code:
    .long 4

# example3-2.s
    .text
    .globl external_exit
    .extern external_exit_code

external_exit:
    movl    external_exit_code, %ebx
    int     $0x80

$ gcc -m32 -c example3-1.s
$ gcc -m32 -c example3-2.s
$ ./my_ld -o example3 example3-1.o example3-2.o
$ nm ./example3
00001000 T _start
0000101e T external_exit
00001000 T external_exit_code
```

Un autre exemple, pour la route :

```
    .text
    .globl _start
```

```
    .local message

_start:
    call    init_data

    movl    $4, %eax
    movl    $1, %ebx
    movl    $message, %ecx
    movl    $13, %edx
    int     $0x80

    movl    $1, %eax
    movl    exit_code, %ebx
    int     $0x80

init_data:
    movl    $0, exit_code
    movl    $message, %eax
    movl    $0x6c6c6548, (%eax)
    addl    $4, %eax
    movl    $0x6f77206f, (%eax)
    addl    $4, %eax
    movl    $0x21646c72, (%eax)
    addl    $4, %eax
    movl    $0x0000000a, (%eax)
    ret

    .lcomm exit_code 4
    .lcomm message 16
```

5 Étape 5 : Gestion des bibliothèques statiques

Afin de ne pas rajouter autant de fichiers objets que nécessaire en argument des linkers, une pratique courante consiste à créer des bibliothèques statiques. Celle-ci sont de simples archives (crées avec `ar(1)`) qui contiennent des fichiers objets ainsi que quelques métadonnées optionelles (comme une table des symboles listant les symboles définis dans les fichiers objets contenus).

Lors de la résolution des symboles, le linker doit tout d'abord tenter de lier les fichiers objets listés seulement. Puis, s'il reste des symboles non définis, le linker doit inclure les fichiers objets contenus dans les archives (uniquement ceux qui définissent de nouveaux symboles).

```
$ gcc -m32 -c example3-1.s
$ gcc -m32 -c example3-2.s
$ ar cru libexample3.a example3-2.o
$ ./my_ld -o example3 example3-1.o libexample3.a
$ nm ./example3
00001000 T _start
0000101e T external_exit
00001000 T external_exit_code
$ gcc -m32 -c example1.s
$ ./my_ld -o example1 example1.o libexample3.a
$ nm ./example3
$
```

À partir de ce point, votre linker devrait être capable de linker correctement un programme avec la libc : félicitations !

...Mais attention, n'essayez pas avec la glibc, celle-ci demande plus de coopération au linker. Vous pouvez en revanche essayer avec une bibliothèque standard plus minimaliste, par exemple la `dietlibc` (`http://www.kernel.org/pub/linux/libs/dietlibc/`) :

```
$ cat test-libc.c
#include <stdio.h>
int main(void)
{
    puts("Hello world!");
    return 0;
}
$ gcc -c test-libc.c
$ wget http://www.kernel.org/pub/linux/libs/dietlibc/dietlibc-0.29.tar.gz
$ tar xf dietlibc-0.29.tar.gz
$ make -kCdietlibc-0.29
$ ./my_ld test-libc.o dietlibc-0.29/bin-i386/{start,std{err,in,out},\\
> fflush,environ,unified,lseek,atexit,puts,write,fwrite,errno_location,\\
> fputc_unlocked,errno}.o
$ ./a.out
Hello world!
```

6 Étape 6 : Be a man!

6.1 Linker dynamique

Le linking des programmes avec des bibliothèques dynamiques se fait en deux étapes. Après la compilation, le linker lie comme vu plus haut les fichiers objets entre eux, un traitement spécial étant fait pour les symboles et les relocations en rapport avec les symboles définis dans les bibliothèques dynamiques.

En effet, ces bibliothèques sont chargées en mémoire au démarrage ; les adresses de leurs segments ne sont donc connues qu'au lancement du programme. C'est le travail du *loader* de s'occuper de ces détails : il est exécuté juste avant le début de l'exécution du programme pour « patcher » la mémoire. Linker et loader doivent coopérer, utiliser les mêmes conventions pour arriver à un processus qui s'interface correctement avec les bibliothèques dynamiques.

6.2 Script de link

GNU ld est paramétré par un script de link. Celui permet de spécifier par exemple l'adresse virtuelle à laquelle placer telles sections, etc.

6.3 Gestion de plusieurs architectures

Le format ELF contient des informations concernant l'architecture de destination (principalement endianness et taille du mot processeur). Les linkers peuvent s'en servir pour gérer plusieurs architectures ! Le but est de garder le code le plus générique possible (pensez template ?). Certains types de relocations sont aussi plus spécifiques qu'en x86. Par exemple, en SPARC, pour charger une adresse en mémoire, il faut deux instructions : une pour la partie haute, une pour la partie basse de l'adresse.

6.4 Compactage de l'ELF

Générer des binaires les plus petits (et donc les plus efficaces à charger en mémoire) est aussi une préoccupation des linkers. Il est ainsi possible de jouer avec la disposition des segments dans l'ELF généré : il s'agit de jouer avec l'alignement des sections, qui n'est pas le même que celui des segments (et donc mapper certaines zones du fichier dans plusieurs segments à la fois).