Descriptif du projet

Sommaire

Autres fichiers

Tous les fichiers de documentation sont présents dans le répertoire doc.

Explication_elf.png :

Schéma explicatif de la première phase, afin d'aider à la compréhension de l'organisation du contenu d'un fichier ELF.

journal.pdf :

Journal de bord mis à jour quotidiennement.

Mode d’emploi

Compilation

Nous avions des difficultés à comprendre le fonctionnement des outils automake/autoconf. C’est pourquoi nous avons préféré écrire notre propre Makefile. Chaque programme est compilé de façon à pouvoir être débuggé.

Chaque programme se présentant sous la forme read-*.c est compilable selon la même structure

```
make bin/dobby-read-*
```

Concernant load-elf-file.c, il faut exécuter

```
make bin/dobby-load-elf-file.c
```

Exécution

Chaque exécutable se présentant sous la forme read-* s’exécute via la commande

```
./bin/dobby-read-* < arguments >
```

Pour la plupart des exécutables, l’argument demandé est un fichier elf, se trouvant dans le répertoire tests/ARM_SOURCE/.

Le programme read-section-content est une exception. Plusieurs arguments sont demandés : un fichier elf, -s | -i selon si la section est indiquée par son nom ou par un numéro (s pour le nom et i pour le numéro), et le nom ou le numéro de la section souhaitée. La commande se présente alors sous la forme

```
./bin/dobby-read-section-content < fichier elf > < -s | -i >
< section souhaitée >
```

ou

```
./bin/dobby-read-section-content < -s | -i > < section souhaitée >
< fichier elf >
```

Concernant load-elf-file, il faut exécuter

```
./bin/dobby-load-elf file < fichier > < options >
```

Avec comme options

```
-o --output < chemin/du/fichier/de/sortie >
-s --section start < nom de section >=< adresse >
```

S’il n’y a pas d’option -o, le nom de l’exécutable créé sera le nom du fichier source sans extension.

S’il n’y a pas d’option -s, il n’y aura pas d’adressage particulier.

Concernant tous les exécutables, un message indiquant les arguments nécessaires s'affiche lorsqu'il n'y a aucun argument lors de l'exécution.

Tests

Tests fonctionnels

Les fichiers .s et .c se trouvent dans tests/ARM_SOURCE/. Afin de générer tous les fichiers objets et les exécutables, il faut lancer le script generate_test_file.sh se trouvant dans le répertoire tests/. Ainsi, deux répertoires sont créés : ARM_BIN contenant tous les exécutables, et ARM_OBJ contenant les fichiers objets.

Les tests peuvent tous être exécutés d'un coup sur tous les exécutables, pour cela, il faut se placer dans le répertoire tests/ et lancer

```
./launch_all_test.sh
```

Chaque test peut être lancé individuellement, en se plaçant dans /test/launch. La commande à effectuer est alors

```
./test/launch_test_*.sh  < programme à tester >  < fichier de test >
```

Sachant que les programmes à tester se trouvent (depuis le répertoire test/) dans ../bin/ et que les exemples se trouvent dans ARM_BIN ou ARM_OBJ.
Si la commande echo "$?" renvoi 0, alors il n'y a pas d'erreur.

Tests de fuite mémoire

Pour lancer les tests de fuite mémoire, il faut se placer dans /tests/ et lancer la commande

```
./dobby_valgrind.sh
```

Descriptif de la structure du code développé

elf_types.h

Les structures nécessaires à la définition du type ELF se trouvent chacune dans les fichiers elf_types_*.h, et sont la retranscription des types définis dans la documentation spécifiant le format ELF.

Le fichier elf_file.h quant à lui défini une structure permettant d'enregistrer en mémoire toutes ces informations.

elf_io.c

Le fichier elf_io.c contient les différentes fonctions permettant de lire et écrire octet par octet dans un fichier ou dans une chaîne de caractères. Ces fonctions sont définies dans elf_io.h et nécessitent la fonction is_big_endian() définie dans utils.h.

Lecture dans un fichier

```
size_t fread_*bits(void *ptr, size_t nmemb, FILE *stream)
```

Ecriture dans un fichier

```
size_t fwrite_*bits(void *ptr, size_t nmemb, FILE *stream)
```

Lecture dans une chaîne de caractères

```
size_t fread_*bits(void *ptr, size_t nmemb, FILE *stream)
```

Ecriture dans une chaîne de caractères

```
size_t fwrite_*bits(void *ptr, size_t nmemb, FILE *stream)
```

elf_header.h

Définition du type Err_ELF_Header, qui sert à vérifier si les données lues sont bien correctes, et si non, indique quelle partie du header a été mal lue. Le fichier elf_header.h contient également la déclaration des fonctions écrites dans elf_header.c, et servent à la lecture / écriture du header d'un fichier au format ELF.

Lecture du header d'un fichier ELF dans une variable de type Elf32_Ehdr

```
Err_ELF_Header read_elf_header(FILE *f, Elf32_Ehdr *hdr)
```

Ecriture du header depuis une variable de type Elf32_Ehdr dans un fichier

```
Void write_elf_header(FILE *f, Elf32_Ehdr hdr)
```

Affichage du header

```
void display_header(Elf32_Ehdr *hdr)
```

La gestion des erreur aura été particulièrement utile pour détecter les diverses erreurs liées notamment à la configuration en Big Endian / Little Endian du fichier ELF.

elf_section_table.h

Définition des fonctions écrites dans elf_section_table.c, qui lisent / écrivent la section table d'un fichier au format ELF.

Lecture de la section table d'un fichier ELF, qui est stockée dans la variable e_table de type Elf32_Shdr

```
void read_elf_section_table(FILE *f, Elf32_Ehdr *header,
Elf32_Shdr e_table[ ])
```

Ecriture de la section table d'un fichier au format ELF

```
void write_elf_section_table(FILE *f, Elf32_Ehdr header,
Elf32_Shdr e_table[ ])
```

Affichage de la section table d'un fichier au format ELF

```
void display_section_table(Elf32_Ehdr *header, Elf32_Shdr e_table[ ],
unsigned char *str_table)
```

elf_symbol.h

Définition des fonctions écrites dans elf_symbol.c, qui lisent / écrivent la table des symboles d'un fichier au format ELF.

Lecture de la table des symboles d'un fichier au format ELF

```
void read_elf_symbol_table(unsigned char* symtab, Elf32_Shdr*
symbol_header, Elf32_Sym sym_table[ ])
```

Affichage de la table des symboles d'un fichier au format ELF

```
void display_symbol_table(Elf32_Sym sym_table[ ],
unsigned int num_symbols, unsigned char *string_table,
unsigned char *section_name)
```

elf_section_content.h

Définition des fonctions écrites dans elf_section_content.c, qui servent à lire / écrire le contenu d'une section d'un fichier au format ELF.

Lecture d’une section donnée d’un fichier ELF

```
unsigned char * read_elf_section_content(FILE *f, Elf32_Shdr shdr)
```

Lecture de toutes les sections d’un fichier ELF (utilise alors la fonction read_elf_section_content)

```
int read_elf_all_section_content(FILE *f, Elf32_Ehdr hdr,
Elf32_Shdr shdr[ ], unsigned char *tab[ ])
```

Ecriture d’une section donnée d’un fichier ELF

```
void write_elf_section_content(FILE *f, Elf32_Shdr shdr,
unsigned char *content)
```

Ecriture de toutes les sections d’un fichier ELF (utilise la fonction write_elf_section_content)

```
void write_elf_all_section_content(FILE *f, Elf32_Half shnum,
Elf32_Shdr shdr[ ], unsigned char *content[ ])
```

Affichage de toutes les sections d’un fichier ELF

```
void display_elf_section_content(unsigned char *content,
Elf32_Word size)
```

elf_load_section.h

Modification du tableau des en-têtes de sections, c’est à dire suppression des sections de relocation, ainsi que des sections vides (exceptée la première). Le tableau des contenus de sections est donc modifié en conséquence, enfin, les indices de sections dans l’en-tête ainsi que dans l’en-tête de la table des symboles sont également corrigés

Renumérotation des sections

```
void renum_section_elf_file(Elf32_File *dest, Elf32_File src,
Elf32_Half correl_table[ ])
```

Modification des adresses de sections

```
int change_section_address(Elf32_File *dest,
const char* section_name, Elf32_Half addr)
```

elf_load_symbol.h

Correction des symboles.

Correction de la numérotation des symboles après renumérotation des sections

```
Elf32_Sym correct_symbol_section(Elf32_Sym sym,
Elf32_Half correl_table[ ])
```

Correction de la valeur des symboles en valeur absolue

```
Elf32_Sym correct_symbol_value(Elf32_Sym sym,
Elf32_Shdr section_table[ ], Elf32_Half sh_count)
```

Application des deux fonctions précédentes sur tous les symboles de la section donnée en argument (sh_num).

```
void correct_all_symbol(Elf32_File* ef, Elf32_Half sh_num, Elf32_Half
correl_table[ ])
```

elf_load_reloc.h

Application des relocations sur le fichier

```
void execute_relocation_section(Elf32_File *ef, Elf32_Shdr reloc_shdr,
unsigned char * reloc_content)
```

Fonctionnalités implémentées et manquantes

Fonctionnalités implémentées

Affichage de l’en-tête
Affichage de la table des sections
Affichage du contenu d’une section
Affichage de la table des symboles
Affichage de la table de réimplantation
Renumérotation des sections
Correction des symboles

Fonctionnalités manquantes

Réimplantations de type R_ARM_ABS*
Réimplantations de type R_ARM_JUMP24 et R_ARM_CALL
Interfaçage avec le simulateur ARM
Exécution avec arm-eabi-run

Bogues connus

Concernant l’exécution de dobby-read-elf-section-content, lors du passage en argument d’un fichier objet dont le passage en format ELF a été incomplet (c’est à dire un fichier dans lequel certaines sections n’auraient pas été écrites), il y a une erreur de segmentation.

Description des tests

Une description de comment lancer les commandes de test se trouve dans le Mode d’emploi.

Les exemples fournis auront été très utiles notamment lors de la phase 1.

Nous avons également de notre côté établis nos propres jeux de tests, notamment des fichiers .c (compilés en ARM) simples. L’un utilise des pointeurs, un autre contient des fonctions, afin d’observer les relocations effectuées lorsqu’on accède à des adresses mémoire. Il y a également un fichier.c qui contient des fonctions non appelées.

Les jeux de tests contiennent également un fichier.o incomplet dont le format ELF ne contient que un header (qui pointe vers des sections qui n’existent pas).