Devoir: Éditeur de liens et chargeur

Automne 2023

1 Présentation

Le but de ce devoir/TP est d'écrire des fonctions réalisant les tâches d'édition de liens et de chargement. Prenez le temps de lire tout le sujet avant de commencer. Le fonctionnement de la mémoire est virtualisé. L'idée générale est de prendre une liste de modules compilés, chacun décrit par une séquence d'instructions et un morceau de mémoire, et de charger ces instructions en mémoire centrale.

Le code fourni contient les modules/fichiers suivants :

- `types.h` et `types.c` contient la définition des principaux types utilisés : les modules, les instructions, et arguments d'instruction ainsi que des constantes ;
- `utils.h` et `utils.c` contient des fonctions d'agrément comme la lecture de la description d'un module dans un fichier, l'affichage de la représentation d'une suite d'instructions ou d'une zone mémoire, ainsi que des primitives de création d'instruction ou de module ;
- `linker.h` et `linker.c` contiendra le code que vous écrirez pour la fonction linker ;
- `loader.h` et `loader.c` contiendra le code que vous écrirez pour la fonction loader ;
- `main.c` contient le main où l'on retrouve l'appel des fonctions linker et loader ;
- `module0.modu` et `module1.modu` contiennent les descriptions des 2 modules utilisés comme exemples ;
- `output_0.txt`, `output_1.txt` et `output_2.txt` qui contiennent ce qui doit être affiché à chacune des étapes ; et
- `makefile` contient les ordres de construction et de test.

2 Travail à effectuer

Il est à réaliser en binôme, sans restriction de groupe.

1. Écrire la fonction linker dans `linker.c`.
2. Écrire la fonction loader dans `loader.c`.
3. Déposer sur celene une archive `nom1_prenom1-nom2_prenom2.tgz` contenant ces deux fichiers (rien d'autre) *avant le 2 octobre 22h.*

3 Modules et affichage

Les modules sont décrits dans des fichiers `.modu` (*cf* exemples). Ils se composent d'un segment de code et d'un segment de données. Chaque instruction dans le segment de code et chaque entrée dans le segment de données occupent un espace mémoire. Le segment de code est représenté par une longueur et une séquence d'instructions, le segment de données par une taille et le contenu de la mémoire (*cf* `types.h`).

Les instructions considérées sont de type langage d'assemblage, avec un nom d'instruction et 1 ou 2 arguments selon l'instruction précédé d'une éventuelle définition de symbole. La syntaxe est définie dans l'Annexe A.

On pourra commencer par regarder `types.h` pour comprendre la représentation du code.

Pour simplifier la compilation et le test, un `makefile`[1] est fourni. La commande

```
make run
```

produit un exécutable et le démarre. Cet exécutable (`main`) affiche la représentation des deux modules donnés en exemple puis le module produit par l'édition de liens et la table des symbole et finalement la mémoire après le chargement. L'affichage d'une suite d'instructions produit deux colonnes représentant chacune la séquence. La première colonne contient les instructions avec le format de ce TP, chaque argument étant représenté par son type et sa valeur. La seconde colonne contient les mêmes instructions traduites dans un pseudo-langage d'assemblage. Les tables de l'annexe B représentent les modules dans leur état initial (l'affichage obtenu sur le terminal).

Les options de compilation sont prévues pour lever le maximum d'erreurs et d'alertes. *Il ne doit pas y en avoir à la compilation de la version finale.*

Il est également possible de tester la sortie par rapport à la sortie attendue avec :

```
make test
```

qui n'affiche que les lignes différentes.

Il est également possible de tester la gestion de la mémoire (`malloc`, `free`...) avec l'outil `valgrind` :

```
make vg
```

Il ne doit y avoir aucun problème de mémoire dans la version finale.

Il est encore possible de lancer le débugger `nemiver` avec :

```
make nmv
```

On peut également restreindre le fonctionnement pour ne faire que charger (step 0) ou charger, éditer les liens et afficher la table des symboles (step 1) ou tout (step 2). Il suffit d'ajouter un argument à `make` quand il est lancé comme ceci :

```
make STEP=1 vg
```

Ceci vérifie la gestion de la mémoire avec l'édition de lien sans faire le chargement en mémoire.

Finalement, en modifiant le makefile (variable TGZ_NAME), il est possible d'engendrer le fichier à rendre par :

```
make tgz
```

4 Mise en jambes

Ceci est optionnel et sert à mieux comprendre le langage d'assemblage. Mais il n'est pas nécessaire de le faire pour la suite.

1. Sachant que la première instruction est celle marquée `deb`, que fait le code présenté ?
2. Dans un fichier `jambes.modu` (qui n'est pas à rendre), créez un module contenant une séquence d'instructions qui réalise l'affichage (`print`) successif des entiers 1 à 100 (vous ne disposez que d'un faible espace mémoire correspondant à 20 instructions).

1. Il marche normalement sur linux. Ailleurs, c'est en fonction de la configuration.

5 Édition de liens

La première étape consiste à écrire la fonction `linker` dont le type est donné. Ses arguments sont le taille et un tableau des modules.

Cette fonction doit :

1. Phase 1 unifier l'adressage des modules, c'est-à-dire produire un unique module en concaténant les instructions des modules et en décalant les adresses internes aux modules. Le segment de données est la concaténation des segments de données.
2. Phase 2 créer et remplir la table des symboles en ajoutant tous les symboles des références (sans adresse) et des symboles (avec l'adresse dans le module unifié). Pour chaque symbole, on parcourt la table pour le chercher, ajouter l'adresse si on a une définition de symbole, et ajouter le symbole s'il n'est pas présent. Le codage de cette liste est laissé à votre discrétion (on peut utiliser des listes chaînées ou un tableau si on connaît la taille à allouer à l'avance).
3. Phase 3 parcourir les instructions des modules et remplacer les références par les adresses de la table.

Attention, il peut y avoir des adressages par des entiers dans le code comme dans les données.
La table de l'annexe D représente le module tel qu'il doit être après l'édition de liens.

6 Chargement

La seconde étape consiste à écrire la fonction loader dont le type est donné. Ses arguments sont une adresse de début dans la mémoire centrale (debut), le module à charger et la représentation de la ram elle-même. Cette fonction doit :

1. charger les deux segments du module en mémoire centrale à partir de l'adresse debut+1, en réservant l'adresse debut pour ajouter une instruction d'appel vers l'emplacement correspondant à la première instruction à exécuter.
2. trouver l'emplacement correspondant à la première instruction à exécuter (référence deb) et ajouter l'instruction initiale.

La Table 5 représente la mémoire obtenue après chargement en mémoire centrale à partir de la position 4. Lorsqu'une case mémoire est inoccupée, la ligne est laissée vide.

A Langage d'assemblage

A.1 Arguments

Le type `targ` représentant les arguments des instructions peut être :

- un `type` d'argument, qui vaut :
  — 0 si l'argument est un entier,
  — 1 si l'argument est un numéro de registre,
  — 2 si l'argument est une adresse absolue (en mémoire centrale),
  — 3 si l'argument est une adresse relative donnée par référence, et
  — 4 si l'argument est une adresse relative entière dans le module courant.
- une valeur dépendant du type et qui est soit un entier, soit une chaîne de caractères dans le cas d'une référence.

Par exemple :

- `{0,5}` représente la constante 5 (notée `5` dans le code) ;

- `{1,1}` représente le registre numéro 1 et est noté `r1` ou `R1` dans le code ;
- `{2,7}` représente l'adresse absolue 7 et est notée `@7` dans le code ;
- `{3,"fac"}` représente la référence `fac` ; et
- `{4,1}` représente l'adresse 1 du module courant (code ou données) et est notée `m5` ou `M5` dans le code.

A.2 Instructions et données en mémoire

Chaque case mémoire (type `casemem`) contient une instruction ou une donnée et est préfixée par une potentielle *définition de symbole* (le souligné veut dire qu'il n'y a pas de définition), qui peut servir de référence dans d'autres parties du programme. Les symboles sont composés de 3 caractères. Aucun symbole ne doit apparaître plusieurs fois dans l'ensemble des modules. Le symbole `deb` doit être défini pour pouvoir faire le chargement.

Les cases mémoire contiennent ensuite un champ `name` qui vaut `mem` pour une donnée et le nom de l'instruction sinon qui sera suivi du nombre d'arguments puis du ou des arguments (aucun argument pour une donnée).

Les arguments utilisés par les instructions sont fournies au format défini plus haut (sous-section A.1).

Le langage d'assemblage utilisé ici contient les instructions :

— `push v` : empile une valeur qui peut être une constante, celle d'un registre ou d'une adresse mémoire ;
— `pop r` : dépile et met la valeur dans le registre `r` ;
— `rdmem r s` : lit dans la mémoire à l'emplacement contenu dans `r` et écrit la valeur dans `s` ;
— `call a r` : appelle la procédure d'adresse relative `a` avec le contenu du registre `r` comme argument ;
— `cmp r v` : compare la valeur du registre `r` à `v` (valeur ou contenu d'un registre ou d'une adresse) ;
— `add r s` : ajoute les valeurs des registres `r` et `s`, place le résultat dans `r` ;
— `jeq a` : si égalité (lors du dernier teste), saute à l'adresse `a` ;
— `jse a` : si supérieur ou égal (lors du dernier teste), saute à l'adresse `a` ;
— `jmp a` : saute à l'adresse `a` ;
— `print r` : affiche le contenu du registre `r` ;
— `inc r` : incrémente la valeur du registre `r` ;
— `dec r` : décrémente la valeur du registre `r` ; et
— `ret r` : revient à la procédure parent avec comme valeur de retour celle contenue dans `r`.

En particulier l'instruction `call a r` en position `i` est équivalente à :

```
(i+2).  jmp a
(i+1).  pushr r
i.      push (i+3)
```

C'est-à-dire que l'adresse de retour est placée sur la pile avant d'aller à la ligne `a`. On peut si nécessaire empiler ensuite des arguments à passer à la fonction appelée.

Et l'instruction `ret r` est équivalente à (ici `R4` est différent de `r`) :

```
(i+2).  jmp R4
(i+1).  push r
i.      pop R4
```

où le premier `pop` sert à récupérer l'adresse de retour de la procédure empilée lors du `call`.

B Modules avant édition de liens

```
################ Segment de code ################
-------------------------------------------------------------------------------
 15 |      _   push ( 4 ,  17 )              |  15 |      _   push M17         |
 14 |    fin    ret ( 1 ,   0 )              |  14 |    fin    ret R0          |
 13 |      _    jmp ( 3 , tst )              |  13 |      _    jmp tst         |
 12 |      _    dec ( 1 ,   2 )              |  12 |      _    dec R2          |
 11 |    elp    inc ( 1 ,   1 )              |  11 |    elp    inc R1          |
 10 |      _    pop ( 1 ,   0 )              |  10 |      _    pop R0          |
  9 |      _   push ( 1 ,   3 )              |   9 |      _   push R3          |
  8 |      _    jse ( 3 , elp )              |   8 |      _    jse elp         |
  7 |      _    cmp ( 1 ,   0 ) ( 1 ,   3 )  |   7 |      _    cmp R0   R3     |
  6 |    blp  rdmem ( 1 ,   1 ) ( 1 ,   3 )  |   6 |    blp  rdmem R1   R3     |
  5 |      _    jeq ( 3 , fin )              |   5 |      _    jeq fin         |
  4 |    tst    cmp ( 1 ,   2 ) ( 0 ,   0 )  |   4 |    tst    cmp R2   0      |
  3 |      _  rdmem ( 1 ,   1 ) ( 1 ,   2 )  |   3 |      _  rdmem R1   R2     |
  2 |      _    pop ( 1 ,   1 )              |   2 |      _    pop R1          |
  1 |      _    pop ( 1 ,   0 )              |   1 |      _    pop R0          |
  0 |    max   push ( 0 ,   0 )              |   0 |    max   push 0           |
-------------------------------------------------------------------------------

################ Segment de données ################
-------------------------------------------------------------------------------
 18 |      _     3   |
 17 |      _     0   |
 16 |      _     0   |
-------------------------------------------------------------------------------
```

TABLE 1 – Module 1 avant édition de liens

```
################ Segment de code ################
-------------------------------------------------------------------------------
  7 |      _  print ( 1 ,   0 )              |   7 |      _  print R0          |
  6 |      _    add ( 1 ,   0 ) ( 1 ,   1 )  |   6 |      _    add R0   R1     |
  5 |      _    pop ( 1 ,   1 )              |   5 |      _    pop R1          |
  4 |      _    pop ( 1 ,   0 )              |   4 |      _    pop R0          |
  3 |      _   call ( 3 , max )              |   3 |      _   call max         |
  2 |      _   push ( 3 , ta2 )              |   2 |      _   push ta2         |
  1 |      _   push ( 3 , ta1 )              |   1 |      _   push ta1         |
  0 |    deb    jmp ( 4 ,   1 )              |   0 |    deb    jmp M1          |
-------------------------------------------------------------------------------

################ Segment de données ################
-------------------------------------------------------------------------------
 18 |      _     4   |
 17 |      _   234   |
 16 |      _     0   |
 15 |      _    10   |
 14 |    ta2     4   |
 13 |      _     4   |
 12 |      _     5   |
 11 |      _     2   |
 10 |      _     5   |
  9 |      _     1   |
  8 |    ta1     5   |
-------------------------------------------------------------------------------
```

TABLE 2 – Module 2 avant édition de liens

C Table des symboles

```
    ################ Table des symboles ################
--------------------------------------------------------------------------------
    max  =>     0
    tst  =>     4
    blp  =>     6
    elp  =>    11
    fin  =>    14
    deb  =>    16
    ta1  =>    27
    ta2  =>    33
--------------------------------------------------------------------------------
```

TABLE 3 – Table des symboles.

D Module après édition de liens

```
################# Segment de code #################
-------------------------------------------------------------------------------
 23 |      _  print ( 1 ,   0 )               |  23 |     _  print R0         |
 22 |      _    add ( 1 ,   0 ) ( 1 ,   1 )   |  22 |     _    add R0   R1   |
 21 |      _    pop ( 1 ,   1 )               |  21 |     _    pop R1         |
 20 |      _    pop ( 1 ,   0 )               |  20 |     _    pop R0         |
 19 |      _   call ( 4 ,   0 )               |  19 |     _   call M0         |
 18 |      _   push ( 4 ,  33 )               |  18 |     _   push M33        |
 17 |      _   push ( 4 ,  27 )               |  17 |     _   push M27        |
 16 |    deb    jmp ( 4 ,  17 )               |  16 |   deb    jmp M17        |
 15 |      _   push ( 4 ,  25 )               |  15 |     _   push M25        |
 14 |    fin    ret ( 1 ,   0 )               |  14 |   fin    ret R0         |
 13 |      _    jmp ( 4 ,   4 )               |  13 |     _    jmp M4         |
 12 |      _    dec ( 1 ,   2 )               |  12 |     _    dec R2         |
 11 |    elp    inc ( 1 ,   1 )               |  11 |   elp    inc R1         |
 10 |      _    pop ( 1 ,   0 )               |  10 |     _    pop R0         |
  9 |      _   push ( 1 ,   3 )               |   9 |     _   push R3         |
  8 |      _    jse ( 4 ,  11 )               |   8 |     _    jse M11        |
  7 |      _    cmp ( 1 ,   0 ) ( 1 ,   3 )   |   7 |     _    cmp R0   R3   |
  6 |    blp  rdmem ( 1 ,   1 ) ( 1 ,   3 )   |   6 |   blp  rdmem R1   R3   |
  5 |      _    jeq ( 4 ,  14 )               |   5 |     _    jeq M14        |
  4 |    tst    cmp ( 1 ,   2 ) ( 0 ,   0 )   |   4 |   tst    cmp R2   0    |
  3 |      _  rdmem ( 1 ,   1 ) ( 1 ,   2 )   |   3 |     _  rdmem R1   R2   |
  2 |      _    pop ( 1 ,   1 )               |   2 |     _    pop R1         |
  1 |      _    pop ( 1 ,   0 )               |   1 |     _    pop R0         |
  0 |    max   push ( 0 ,   0 )               |   0 |   max   push 0          |
-------------------------------------------------------------------------------

################# Segment de données #################
-------------------------------------------------------------------------------
 37 |       _     4   |
 36 |       _   234   |
 35 |       _     0   |
 34 |       _    10   |
 33 |     ta2     4   |
 32 |       _     4   |
 31 |       _     5   |
 30 |       _     2   |
 29 |       _     5   |
 28 |       _     1   |
 27 |     ta1     5   |
 26 |       _     3   |
 25 |       _     0   |
 24 |       _     0   |
-------------------------------------------------------------------------------
```

Table 4 – Module après l'édition de liens.

E Ram après chargement

```
    ################ Mémoire centrale ################
-------------------------------------------------------------------------------
 44 |                      |
 43 |                      |
 42 |          4           |
 41 |        234           |
 40 |          0           |
 39 |         10           |
 38 |          4           |
 37 |          4           |
 36 |          5           |
 35 |          2           |
 34 |          5           |
 33 |          1           |
 32 |          5           |
 31 |          3           |
 30 |          0           |
 29 |          0           |
 28 |   print  R0          |
 27 |     add  R0   R1     |
 26 |     pop  R1          |
 25 |     pop  R0          |
 24 |    call  M5          |
 23 |    push  M38         |
 22 |    push  M32         |
 21 |     jmp  M22         |
 20 |    push  M30         |
 19 |     ret  R0          |
 18 |     jmp  M9          |
 17 |     dec  R2          |
 16 |     inc  R1          |
 15 |     pop  R0          |
 14 |    push  R3          |
 13 |     jse  M16         |
 12 |     cmp  R0   R3     |
 11 |   rdmem  R1   R3     |
 10 |     jeq  M19         |
  9 |     cmp  R2   0      |
  8 |   rdmem  R1   R2     |
  7 |     pop  R1          |
  6 |     pop  R0          |
  5 |    push  0           |
  4 |     jmp  M21         |
  3 |                      |
  2 |                      |
  1 |                      |
  0 |                      |
-------------------------------------------------------------------------------
```

Table 5 – Mémoire centrale après chargement