Devoir : Éditeur de liens et chargeur

novembre 2020

1 Présentation

Le but de ce 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 liste d'instructions, et de charger ces instructions en mémoire centrale. Le code fourni contient les fichiers suivants :

- `types.h` contient la définition des principaux types utilisés : les modules, les instructions, et arguments d'instruction.
- `listeschainees.h` et `listeschainees.c` donnent des outils pour gérer des listes chaînées, utilisées en particulier pour les listes de références et de définitions externes des modules.
- `utils.c` contient des fonctions d'agrément comme l'affichage de la représentation d'une liste d'instructions, ainsi que des primitives de création d'instruction ou de module.
- `linker.c` et `loader.c` contiendront le code que vous écrirez pour la fonction linker et la fonction loader.
- `modules.c` contient le main avec la construction d'un exemple et l'appel des fonctions linker et loader.

Les instructions considérées sont de type langage d'assemblage, avec un nom d'instruction et 1 ou 2 arguments selon l'instruction. On pourra commencer par regarder `types.h` pour comprendre le schéma de typage. La syntaxe est définie en annexe B. Chaque adresse est un emplacement pour une instruction exactement. La commande

```
gcc listeschainees.c linker.c loader.c utils.c modules.c -o exec
```

produit un exécutable qui affiche la représentation des deux modules donnés en exemple, ainsi que celle de la mémoire centrale (avant chargement donc vide).

L'affichage d'une liste d'instructions produit deux colonnes représentant chacune la liste. La première colonne contient la liste d'instructions avec le format de ce TP, chaque argument étant représenté par son type et sa valeur (on oublie ici le champ `intorname`). La seconde colonne contient les mêmes instructions traduites dans un pseudo-langage d'assemblage. La table 1 représente les modules dans leur état initial (l'affichage obtenu sur votre terminal).

Les langages d'assemblage utilisés sont décrits en annexe.

2 Mise en jambes

1. Sachant que la première instruction est celle marquée `deb`, que fait le code présenté ?
2. Dans un fichier `jambes.c`, créez un module contenant une liste 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).
3. Dans un fichier `modulesbis.c`, réécrivez les deux modules décrits dans `modules.c` en utilisant l'équivalence pour les appels `call` et `ret` donnée en annexe B.

9		prin	(1, 2)			9		prin	R2	
8		add	(1, 2)	(1, 3)		8		add	R2	R3
7		pop	(1, 3)			7		pop	R3	
6		pop	(1, 2)			6		pop	R2	
5		call	(3,fac)	(1, 3)		5		call	fac	R3
4		call	(3,fac)	(1, 2)		4		call	fac	R2
3		pop	(1, 3)			3		pop	R3	
2		push	(0, 6)			2		push	6	
1		pop	(1, 2)			1		pop	R2	
0	deb	push	(0, 3)			0	deb	push	3	

9	fin	ret	(1, 0)			9	fin	ret	R0	
8		mul	(1, 0)	(1, 1)		8		mul	R0	R1
7		pop	(1, 0)			7		pop	R0	
6		pop	(1, 1)			6		pop	R1	
5		call	(3,fac)	(1, 0)		5		call	fac	R0
4		dec	(1, 0)			4		dec	R0	
3		push	(1, 0)			3		push	R0	
2		jeq	(3,fin)			2		jeq	fin	
1		cmp	(1, 0)	(0, 1)		1		cmp	R0	1
0	fac	pop	(1, 0)			0	fac	pop	R0	

TABLE 1 – Modules avant édition de liens

3 Édition de liens

La première tâche consiste à écrire la fonction `linker` dont le type est donné. Ses arguments sont la liste des modules avec leur nombre et une table des symboles initialement vide sous forme de liste chaînée.

Cette fonction doit :

1. Phase 1 unifier l'adressage des modules de la liste, c'est à dire produire un unique module en décalant les instructions des modules et les adresses internes aux modules (pour les définitions notamment).
2. Phase 2 remplir la table des symboles en ajoutant tous les symboles des références (sans adresse) et des définitions (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, et ajouter le symbole s'il n'est pas présent.

   Rmq : On pourra ici écrire des fonctions dans `listeschainees.c` pour la manipulation de la table. La table, elle, est déjà créée et passée comme argument de `linker`.
3. Phase 3 parcourir les instructions des modules et remplacer les références à des objets externes par les adresses de la table.

La table 2 représente les instructions des modules telles qu'elles doivent être après l'édition de liens. Observez par exemple les modifications des lignes 4 et 5 du module 1.

4 Chargement

La seconde partie consiste à écrire la fonction `loader` dont le type est donné. Ses arguments sont la liste des modules avec leur nombre, une adresse de début dans la mémoire centrale et la

19		prin	(1,	2)			19		prin	R2	
18		add	(1,	2)	(1,	3)	18		add	R2	R3
17		pop	(1,	3)			17		pop	R3	
16		pop	(1,	2)			16		pop	R2	
15		call	(3,	0)	(1,	3)	15		call	0	R3
14		call	(3,	0)	(1,	2)	14		call	0	R2
13		pop	(1,	3)			13		pop	R3	
12		push	(0,	6)			12		push	6	
11		pop	(1,	2)			11		pop	R2	
10	deb	push	(0,	3)			10	deb	push	3	
9	fin	ret	(1,	0)			9	fin	ret	R0	
8		mul	(1,	0)	(1,	1)	8		mul	R0	R1
7		pop	(1,	0)			7		pop	R0	
6		pop	(1,	1)			6		pop	R1	
5		call	(3,	0)	(1,	0)	5		call	0	R0
4		dec	(1,	0)			4		dec	R0	
3		push	(1,	0)			3		push	R0	
2		jeq	(3,	9)			2		jeq	9	
1		cmp	(1,	0)	(0,	1)	1		cmp	R0	1
0	fac	pop	(1,	0)			0	fac	pop	R0	

TABLE 2 – Modules après l'édition de liens.

représentation de la mémoire elle même.

Cette fonction doit :

1. charger le 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 du `main`.
2. trouver l'emplacement correspondant à la première instruction du `main` et ajouter l'instruction initiale.

La table 3 représente la mémoire que vous devez obtenir après chargement en mémoire centrale à partir de la position 4. Lorsqu'une case mémoire est inoccupée, elle est marquée `empty`.

29		empty						29		empty				
28		empty						28		empty				
27		empty						27		empty				
26		empty						26		empty				
25		empty						25		empty				
24		prin	(1,	2)				24		prin	R2			
23		add	(1,	2)	(1,	3)		23		add	R2	R3		
22		pop	(1,	3)				22		pop	R3			
21		pop	(1,	2)				21		pop	R2			
20		call	(2,	5)	(1,	3)		20		call	5	R3		
19		call	(2,	5)	(1,	2)		19		call	5	R2		
18		pop	(1,	3)				18		pop	R3			
17		push	(0,	6)				17		push	6			
16		pop	(1,	2)				16		pop	R2			
15	deb	push	(0,	3)				15	deb	push	3			
14	fin	ret	(1,	0)				14	fin	ret	R0			
13		mul	(1,	0)	(1,	1)		13		mul	R0	R1		
12		pop	(1,	0)				12		pop	R0			
11		pop	(1,	1)				11		pop	R1			
10		call	(2,	5)	(1,	0)		10		call	5	R0		
9		dec	(1,	0)				9		dec	R0			
8		push	(1,	0)				8		push	R0			
7		jeq	(2,	14)				7		jeq	14			
6		cmp	(1,	0)	(0,	1)		6		cmp	R0	1		
5	fac	pop	(1,	0)				5	fac	pop	R0			
4		call	(2,	15)				4		call	15			
3		empty						3		empty				
2		empty						2		empty				
1		empty						1		empty				
0		empty						0		empty				

TABLE 3 – Mémoire après chargement

A Langage d'assemblage : les arguments

Le type `arginstr` représentant les arguments des instructions contient :

- 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 (dans le module courant)
- `intorname` permettant de distinguer les adresses relatives qui sont des entiers (`intorname = 0`) de celles qui sont des références (`intorname = 1`)
- une valeur dépendant des 2 premiers arguments et qui est soit un entier, soit une chaîne de caractères dans le cas d'une référence.

Par exemple :

- `{0,0,5}` représente la constante 5
- `{3,1,"fac"}` représente la référence "fac"
- `{1,0,1}` représente le registre numéro 1
- `{2,0,7}` représente l'adresse absolue 7.

B Langage d'assemblage : les instructions

Chaque instruction de type `instr` décrit une instruction préfixée par une *définition* (chaîne vide par défaut), qui peut servir de référence dans d'autres parties du programme. La définition `deb` (éventuellement non présente) doit être unique dans un module et correspond à l'emplacement d'appel de la fonction `main` si elle existe.

Les arguments utilisés par les instructions sont fournies au format défini plus haut A.

Le langage d'assemblage utilisé ici contient les instructions :

— `push x` : empile la valeur x ou la valeur du registre x
— `pop r` : dépile et met la valeur dans le registre r
— `cmp r y` : compare la valeur du registre r à y
— `jeq x` : si égalité, saute à l'adresse x
— `dec r` : décrémente la valeur du registre r
— `call x r` : appelle la procédure d'adresse relative x avec comme argument la valeur du registre r
— `mul r s` : multiplie les valeurs des registres r et s, place le résultat dans r
— `add r s` : ajoute les valeurs des registres r et s, place le résultat dans r
— `prin r` : affiche le contenu du registre r
— `ret r` : revient à la procédure parent avec comme valeur de retour celle contenue dans r

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

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

Et l'instruction `ret r` est équivalente à :

```
(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`.