Programmation élémentaire

Corewar

Responsable du module
younes2.serraj@epitech.eu
Dernière modification
23/03/2016_13h53

Table des matières

Détails administratifs

- Nom du répertoire de rendu : CPE_année_corewar
  `Exemple pour l'année 2015-2016 : CPE_2015_corewar`
- Les binaires s'appelleront `asm` et `corewar` et seront dans leurs dossiers respectifs éponymes
- Le Makefile devra être à la racine du dépôt et compiler les deux binaires.
- Si vous souhaitez implémenter des bonus, ils devront se trouver dans un sous-dossier "bonus". Attention, lorsque nous compilerons votre programme dans sa version basique (cad celle qui répond scrupuleusement au sujet ci-dessous), nous supprimerons le sous-dossier "bonus". Assurez vous que votre programme compile et s'exécute correctement dans cette configuration.

`Attention : la norme sera vérifiée sur tous les fichiers que vous rendrez, op.c et op.h compris`

Infos

- Le championnat vous a permis de tout comprendre sur le Corewar.
- Le sujet est désormais très simple : à vous de développer votre assembleur et votre machine virtuelle.
- Des binaires de référence sont disponibles sur l'intra.
- Détails de la machine virtuelle graphique version 4.2
  C'est la machine utilisée pour le championnat, elle a certaines spécificités :
  - #define CYCLE_TO_DIE 1536
  - #define CYCLE_DELTA 4
  - #define NBR_LIVE 2048
  - des nombres de cycles parfois différents pour certaines instructions
  - on peut mettre de 2 à 4 joueurs
  - diverses options (lancer le binaire sans paramètre)
  - un système de blocage des processus dans leur espace mémoire initial pour les N premiers cycles (N paramétrable : option "-ctmo")
  - une instruction "gtmd" qui prend un octet de codage des param. Seul paramètre : un numéro de registre. Action : met dans le registre désigné le nombre de cycles à attendre avant l'"ouverture" de la ram à tout le monde
  - Vous pouvez implémenter si vous le souhaitez le système de blocage des processus -ctmo, ainsi que l'instruction supplémentaire gtmd (ceci est optionnel).

Introduction

Qu'est-ce que c'est que ce truc ?

- Le corewar est un jeu, un jeu très particulier. Il consiste à opposer de petits programmes dans une machine virtuelle.
- Le but du jeu est d'empêcher les autres programmes de fonctionner correctement, et ce par tous les moyens disponibles.
- Le jeu va donc créer une machine virtuelle dans laquelle les programmes (écrits pas les joueurs) s'affrontent. L'objectif de chaque programme est de "survivre". Par "survivre" on entend exécuter une instruction spéciale (live) qui veut dire "je suis en vie". Ces programmes s'exécutent simultanément dans la machine virtuelle et ce dans un même espace mémoire, et peuvent donc écrire les uns sur les autres. Le gagnant du jeu est le dernier a avoir exécuté l'instruction "live".

Comment ça marche ?

- Le projet va se découper en trois parties :
  - **L'assembleur** : Il va permettre d'écrire des programmes destinés à se battre. Il devra donc comprendre le langage assembleur et générer des programmes en binaire compréhensibles par la machine virtuelle.
  - **La machine virtuelle** : Elle va héberger les programmes binaires que sont les champions et leur fournir un environnement d'exécution standard. Elle offre tout un tas de fonctionnalités utiles aux combats des champions. Il va de soi qu'elle doit pouvoir exécuter plusieurs programmes à la fois (sinon, les combats ne vont pas être passionnants ...)
  - **Le champion** : C'est votre oeuvre personnelle. Il devra pouvoir se battre et sortir vainqueur de l'arène qu'est la machine virtuelle. Il sera donc écrit dans le langage assembleur propre à notre machine virtuelle (décrite plus loin dans le sujet).

La machine virtuelle

- La machine virtuelle est une machine multi-programmes. Chaque programme dispose de :
  - REG_NUMBER registres qui font REG_SIZE octets.
    Un registre est une petite mémoire qui ne contient qu'une seule valeur. Dans une vraie machine, elle est interne au processeur, et de ce fait elle est très rapide d'accès.
    REG_NUMBER et REG_SIZE sont des #define disponibles dans op.h.
  - d'un pc (compteur de programme)
    C'est un registre spécial qui contient juste l'adresse en mémoire (dans la machine virtuelle) de la prochaine instruction à décoder et exécuter. Très pratique si on veut savoir où l'on se trouve et pour écrire des choses en mémoire.
  - d'un flag nommé (bien mal) carry qui vaut un si la dernière opération a renvoyé zéro.
- Le rôle de la machine est d'exécuter les programmes qui lui sont donnés en paramètre.
- Elle doit vérifier que chaque processus appelle l'instruction "live" tous les CYCLE_TO_DIE cycles.
- Si après NBR_LIVE appels à la fonction "live" tous les processus en vie, le sont toujours, on décrémente CYCLE_TO_DIE de CYCLE_DELTA unités et on recommence jusqu'à ce qu'il n'y ait plus de processus en vie.
- C'est le dernier joueur valide à avoir dit "live" qui a gagné.
- La machine doit alors afficher : "Le joueur x(nom_du_joueur) a gagné." où x est le numéro et nom_du_joueur le nom.
- Exemple :

  "Le joueur 3(zork) a gagné."

  A chaque exécution de l'instruction "live" la machine doit afficher :

  "Le joueur x(nom_du_joueur) est en vie."

  Le numéro de joueur est généré par la machine et est fourni aux programmes dans le registre r1 au démarrage du processus (tous les autres seront mis à 0 sauf bien sûr le PC).

Syntaxe de la ligne de commande

- SYNOPSIS
  corewar [-dump nbr_cycle] [[-n prog_number] [-a load_address] prog_name] ...
- DESCRIPTION
  - -dump nbr_cycle
    Dump la mémoire après nbr_cycle d'exécution (si la partie n'est pas déjà finie)
    au format 32 octets par ligne au format xx code en hexadécimal : A0BCDEFE1DD3.....
    une fois la mémoire dumpée, on quitte la partie.
  - -n prog_number
    Fixe le numéro du prochain programme. Par défaut il choisit le premier numéro libre dans l'ordre des paramètres.
  - -a load_address
    Fixe l'adresse de chargement du prochain programme. Lorsque aucune adresse n'est précisée, on choisira les adresses de telle sorte que les programmes soient les plus éloignés. Les adresses sont modulo MEM_SIZE.
  - Dans tous les cas d'erreurs de syntaxe, envoyer un message d'erreur.

Code machine

- La machine doit reconnaître les instructions suivantes :

Mnémonique	Effets
0x01 (live)	Suivie de 4 octets qui représente le numéro du joueur. Cette instruction indique que ce joueur est en vie. Pas d'octet de codage des paramètres.
0x02 (ld)	Cette instruction prend 2 paramètres, le deuxième est forcément un registre (pas le PC). Elle load la valeur du premier paramètre dans le registre. Cette opération modifie le carry. ld 34,r3 charge les REG_SIZE octets à partir de l'adresse (PC + (34 % IDX_MOD)) dans le registre r3.
0x03 (st)	Cette instruction prend 2 paramètres. Elle store (REG_SIZE OCTET)la valeur du premier argument (toujours un registre) dans le second. st r4,34 store la valeur de r4 à l'adresse (PC + (34 % IDX_MOD)) st r3,r8 copie r3 dans r8
0x04 (add)	Cette instruction prend 3 registres en paramètre, additionne le contenu des 2 premiers et met le résultat dans le troisième. Cette opération modifie le carry. add r2,r3,r5 additionne r2 et r3 et met le résultat dans r5
0x05 (sub)	même que add mais soustrait
0x06 (and)	p1 & p2 -> p3. Le paramètre 3 est toujours un registre. Cette opération modifie le carry. and r2, %0,r3 met r2 & 0 dans r3
0x07 (or)	même que and mais avec le ou (| du c).
0x08 (xor)	même que and mais avec le ou exclusif (ôu c).
0x09 (zjmp)	Cette instruction n'est pas suivie d'octet pour décrire les paramètres. Elle prend toujours un index (IND_SIZE) et fait un saut à cet index si le carry est à un. Si le carry est nul, zjmp ne fait rien mais consomme le même temps. zjmp %23 met, si carry == 1, (PC + (23 % IDX_MOD)) dans le PC.
0x0a (ldi)	Cette opération modifie le carry. ldi 3,%4,r1 lit IND_SIZE octets a l'adresse : (PC + (3 % IDX_MOD)) ajoute 4 à cette valeur. On nommera S cette somme. On lit REG_SIZE octet à l'adresse (PC + (S % IDX_MOD)) que l'on copie dans r1. Les paramètres 1 et 2 sont des index.
0x0b (sti)	sti r2,%4,%5 sti copie REG_SIZE octet de r2 a l'adresse (4 + 5) Les paramètres 2 et 3 sont des index. Si les paramètres 2 ou 3 sont des registres, on utilisera leur contenu comme un index.

0x0c (fork)	Cette instruction n'est pas suivie d'octet pour décrire les paramètres. Elle prend toujours un index et crée un nouveau programme qui s'exécute à partir de l'adresse : (PC + (premier paramètre % IDX_MOD)). Fork %34 crée un nouveau programme. Le nouveau programme hérite des différents états du père.
0x0d (lld)	Comme ld sans le %IDX_MOD Cette opération modifie le carry.
0x0e (lldi)	Comme ldi sans le %IDX_MOD Cette opération modifie le carry.
0x0f (lfork)	Comme fork sans le %IDX_MOD Cette opération modifie le carry.
0x10 (aff)	Cette instruction est suivie d'un octet de paramétrage pour décrire les paramètres. Elle prend en paramètre un registre et affiche le caractère dont le code ascii est présent dans ce registre (base 10). (un modulo 256 est appliqué au code ascii, le caractère est affiché sur la sortie standard) Ex : ld %42,r3 aff r3 affiche '*' sur la sortie standard

- Tous les adressages sont relatifs au PC, IDX_MOD sauf "lld", "lldi" et "lfork".
- En tout état de cause, la mémoire est circulaire et fait MEM_SIZE octets.
- Le nombre de cycles de chaque instruction, leur représentation mnémonique, leur nombre de paramètres et les types de paramètres possibles sont décrits dans le tableau op_tab déclaré dans op.c.
- Tous les autres codes n'ont aucune action à part passer au suivant et perdre un cycle.

Scheduling

- La machine virtuelle est supposée émuler une machine parfaitement parallèle.
- Mais pour des raisons d'implémentation, on supposera que chaque instruction s'exécute entièrement à la fin de son dernier cycle et attend durant toute sa durée. Les instructions commençant à un même cycle s'exécutent dans l'ordre croissant des numéros de programme. (voir figure)
- Exemple :
  Considérons trois programmes (P1, P2, P3) respectivement constitués des instructions 1.1 1.2 .. 1.7 , 2.1 .. 2.7, 3.1 .. 3.7 . Les timings de chaque instruction étant donnés dans le tableau suivant :

P1 :	1.1(4 cycles)	1.2(5)	1.3(8)	1.4(2)	1.5(1)	1.6(3)	1.7(2)
P2 :	2.1(2 cycles)	2.2(7)	2.3(9)	2.4(2)	2.5(1)	2.6(1)	2.7(3)
P3 :	3.1(2 cycles)	3.2(9)	3.3(7)	3.4(1)	3.5(1)	3.6(4)	3.7(9)

- La machine virtuelle exécutera les instructions dans l'ordre suivant :

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13
Instructions	1,1				1,2					1,3			
Instructions	2,1		2,2							2,3			
Instructions	3,1		3,2									3,3	

Cycles	14	15	16	17	18	19	20	21	22	23	24	25
Instructions					1,4		1,5	1,6			1,7	
Instructions						2,4		2,5	2,6	2,7		
Instructions						3,4	3,5	3,6				3,7

Indices `Au cycle 21 la machine exécute 1.6 (l'instruction 6 du prog 1) puis l'instruction 2.5 (l'instruction 5 du prog 2) puis 3.6.`

L'assembleur

La machine virtuelle exécute du code machine. Mais pour écrire les programmes, on utilisera un langage simple nommé assembleur. Il est composé d'une instruction par ligne. Les instructions sont composées de trois éléments :

- Un label optionnel suivi du caractère LABEL_CHAR (ici " :") déclaré dans op.h.
- Les labels peuvent être n'importe quelle chaîne de caractères composée des éléments de la chaîne LABEL_CHARS déclarée dans op.h.
- Un code d'instruction (opcode). Les instructions que la machine connaît sont définies dans le tableau op_tab déclaré dans op.c.
- Les paramètres de l'instruction.

Une instruction peut avoir de 0 à MAX_ARGS_NUMBER paramètres séparés par des virgules. Chaque paramètre peut être de trois types :

- Registre : r1 <–> rx avec x = REG_NUMBER
  Exemple : ld r1,r2. (load r1 dans r2)
- Direct : Le caractère DIRECT_CHAR suivi d'une valeur ou d'un label (précédé de LABEL_CHAR), ce qui représente la valeur directe.
  Exemple : ld %4,r5 (load 4 dans r5)
  Exemple : ld % :label, r7 (load label dans r7)
- Indirect : Une valeur ou un label (précédé de LABEL_CHAR) qui représente la valeur qui se trouve à l'adresse du paramètre relativement au PC.
  Exemple : ld 4,r5 (load les 4 octets se trouvants à l'adresse (4+PC) dans r5).

Syntaxe de la ligne de commande

- SYNOPSIS
  asm file_name[.s] .....
- DESCRIPTION
  transforme file_name[.s] en file_name.cor (un exécutable de la machine virtuelle).

Codage

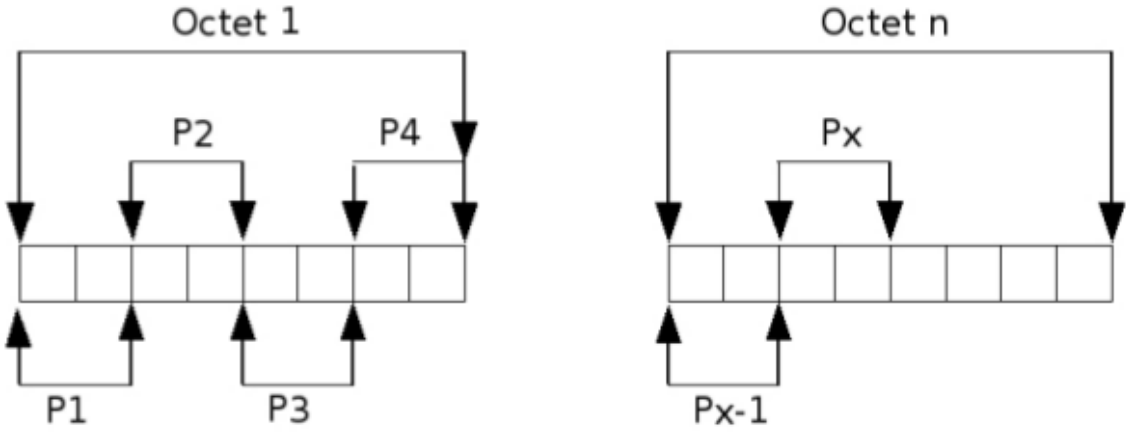
Le codage est plutôt simple :

Chaque instruction est codée par le code de l'instruction, la description du type des paramètres puis des paramètres.

- Le code de l'instruction (on le trouve dans op_tab qui est lui même dans op.h).
- La description du type de paramètres (voir figure). Pour les instructions live,zjmp, fork et lfork, elle n'est pas présente.
- Exemple d'octet de codage :

r2,23,%34	Donnera un octet de codage 01 11 10 00 soit 0x78
23,45,%34	Donnera un octet de codage 11 11 10 00 soit 0xF8
r1,r3,34	Donnera un octet de codage 01 01 11 00 soit 0x5c

Codages du type de paramètres :



01 Registre, Suivie d'un octet (le numéro de registre)
10 Direct, Suivie de DIR_SIZE octets (la valeur directement)
11 Indirect, Suivie de IND_SIZE octets (la valeur de l'indirection)

- Les paramètres.

Après le type de paramètre, on met directement les paramètres :

- pour un registre on met son numéro sur un octet
- pour un direct la valeur sur DIR_SIZE octet(s)
- pour un indirect sur IND_SIZE.

R2,23,%34	Donnera 0x78 puis 0x02 0x00 0x17 0x00 0x00 0x00 0x22
23,45,%34	Donnera 0xF8 puis 0x00 0x17 0x00 0x2d 0x00 0x00 0x00 0x22
r1,r3,%34	Donnera 0x01 puis 0x03 0x00 0x22

Exemple :

```
#
# ex.s for corewar
#
# Alexandre David
#
# Sat Nov 10 22:24:30 2201
#
.name "zork"
.comment "just a basic living prog"

l2:
sti r1,%:live,%1
and r1,%0,r1
live: live %1
zjmp %:live

# Exécutable compilé :
#
# 0x0b,0x68,0x01,0x00,0x0f,0x00,0x01
# 0x06,0x64,0x01,0x00,0x00,0x00,0x00,0x01
# 0x01,0x00,0x00,0x00,0x01
# 0x09,0xff,0xfb
```

- Le programme doit, à partir du fichier assembleur qui lui est passé sur la ligne de commande, produire un exécutable pour la machine virtuelle. Cet exécutable commence par un header défini par le type header_t déclaré dans op.h.
- Note importante : la machine virtuelle est BIG ENDIAN (comme les Sun et non comme i386).

Modus operandi d'un champion

- Un objectif : Il ne peut en rester qu'un.
- Au démarrage du jeu et donc de la machine virtuelle, chaque champion va trouver dans son registre r1 perso le numéro qui lui est attribué. Il devra s'en servir pour ses instructions "live".
- Si un champion fait un live avec un numéro autre que le sien, pas de chance, au pire ce n'est pas perdu pour tout le monde.
- A ce sujet-là, jetez un oeil au code donné en exemple pour la partie codage des instructions, il est très utile.
- Toutes les instructions sont utiles, toutes les réactions de la machine qui sont décrites dans ce sujet sont exploitables pour donner de la vie à vos champions.
- Par exemple, quand la machine tombe sur un opcode inconnu, que fait-elle ? Comment alors en détourner l'utilisation ?
- Notez bien que la machine dispose d'une instruction "fork", elle est très utile pour submerger l'adversaire, mais elle prend du temps et peut devenir fatale si la fin du CYCLE_TO_DIE arrive sans qu'elle ait pu se terminer et permettre de faire un "live" derrière !

Les messages

Les messages ne sont pas tenus d'être respectés au caractère près, mais ils doivent être pertinents. L'idéal est de reproduire le comportement des binaires de référence.

Pour la machine

1. "file_name is not a corewar executable" (ou file_name est le nom d'un des arguments).
2. "prog number the_number already used" (ou the_number est le numéro demandé).
3. "le joueur x(nom_du_joueur) est en vie"
4. "le joueur x(nom_du_joueur) a gagne"

Les messages 1 et 2 provoquent l'arrêt du programme.
Pour l'assembleur

1. "Syntax error line x" (ou x le numéro de ligne(première line 1))
2. "Warning Indirection to far line x" (indirection > a IDX_MOD)
3. "label the_label undefine line x"
4. "no such register line x"
5. "Warning Direct too big line x"

Les messages 1, 3 et 4 provoquent l'arrêt du programme.
Pour les deux

1. "File file_name not accessible"
2. "Can't perform malloc"

Ces 2 messages provoquent l'arrêt du programme.

Si vous avez besoin d'autres messages, consultez le responsable du module pour avoir son aval. Toute notification du responsable du module dans le groupe CPE 2020 sur Yammer sera considérée comme faisant partie du sujet (vous devez donc en tenir compte).

Conclusion

- Pour le reste, réfléchissez, lisez op.h et op.c, et dans le cas où il y a une ambiguïté, demandez des instructions précises à vos assistants et/ou sur le groupe Yammer correspondant au module.
- Nous vous invitons à vérifier que vos programmes soient parfaitement conformes à la norme.
- Bon travail.

Fonctions autorisées

- open
- read
- write
- lseek
- close
- malloc
- realloc
- free
- exit