COREWAR

1 - Introduction

Corewar -> arene virtuel où des champion en pseudo asm s'affrontent jusqu'au dernier

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 entends exécuter une instruction spéciale (live) qui veut dire « je suis en vie ». Ces programmes s'éxecutent 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 ».

Le projet va donc se decouper en 3 partie :

- L'asm : il va permetre de convertir les champions en langage capable d'etre lu par la vm
- La Vm : Elle va heberger les binaires des champ et leurs fournir un environement d'execution
- Les champions

2 - Programme :

a) ASM

La partie ASM à pour but de convertire le fichier asm (.s) en fichier executable par le programme corewar (.core).

Usage :

./asm [-BdDehHopSstu] [file.s]

-B : print the binary version of the source code.

-d : show which of the opcodes needed a description oct.

-D : print the decimal version of the source code.

-e : make the compiler sensible for errors.

-h : show this message.

-H : print the hexa-decimal version of the source code.

-o : show description of op-codes.

-p : Print a detailed description of the source file.

-s : print symboles table.

-S : show label size of op-code.

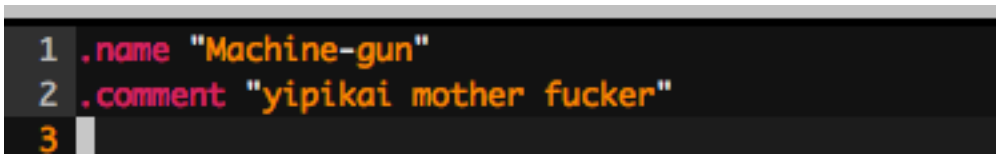
-t : show informations about type of each op-code.

-u : print usage.

B) Champions

- Les champs sont ecrits en pseudo asm dans des fichier .s

Leurs structure header doit etre la suivante :



Nous detailerons plus tard le fonctionement exact des champions

C) COREWAR

L'executable CORWAR lance la vm avec jusqu'a 4 champ en parametre

Usage :

./corewar [champ.s] {[-v] [-h] [-dump [int]] [[-n number_champ] champion_N.cor]

-v : visu

-h : help

-dump : exit le programme et renvois le dernier print de la vm apres x boucle du programme (ex : ./corware -dump 0 champ.core -> renvois le visu de la vm apres 0 tours de boucle) -> interet ? le debug (plus d info : https://cdiese.fr/les-dumps-memoire-en-5-min/)

-n : pour definir un id de depart au champ (optionel)

Comment jouer ?

Nous utiliseront 17 instructions assembleur reconnues par la machine. Un code machine est composé d'un mnémonique (ou instruction / opcode) et de un ou deux arguments représentant des adresses mémoires. Une ligne de code source (qui une fois compilée occupe une case mémoire) est donc composée ainsi :

<instruction> <type argument 1> <argument 1> <type argument 2> <argument 2>

Fonctionement :

La machine virtuelle est constituée d'une mémoire et d'une structure contenant l'état de chaque joueur et de chaque processus. La mémoire est circulaire, c'est-à-dire que les adresses mémoire sont toujours calculées modulo M. Plus généralement, toute l'arithmétique de la machine virtuelle est réalisée modulo M.

- Les structures :

```
typedef struct      s_champ
{
    int             num;
    int             id;
    char            *name;
    char            *comment;
    int             size;
    t_vm_inst       *src;
    struct s_champ  *next;
}                   t_champ;

typedef struct      s_url_file
{
    int             num;
    char            *url;
    struct s_url_file   *next;
}                   t_url_file;
```

Int num	Numero du champ
Id	Iddentifier le process pour la couleur
Name	Nom du champ
Comment	Commentaire du fichier .s
Size	Taille en octet du champ. Doit etre < a CHAMP_MAX_SIZE
Next	Les champ sont stock ds une lst chaine

Nbr champ	Comme son nom l'indique

```
typedef struct      s_vm
{
    int             nbr_champ;
    int             dump;
    int             mem[MEM_SIZE][2];
    t_champ         *champs;
    int             live[MAX_PLAYERS];//
    int             id_last_a_live;///
}                   t_vm;
```

Dump	Nb de tour avant exit(0); -->defini par l'option –dump
Mem**	Tableau a double entrée contenant les op code en casse 0 et l'idd du champ corespondant en casse 1 C'est depuis ce tableau qu'on gere l affichage
Champs	Pt sur la lst chane champ
Live	Vie par joueur
Id_last_alive	Id du dernier champ a donner signe de vie

Cette struct sera utilisée tout au long de l'execution

de la Vm que nous detailerons plus bas

```
typedef struct      s_process
{
    int             id_parent;
    int             curent_pc;
    int             pc;
    BOOL            carry;
    int             reg[REG_NUMBER];
    int             time_to_exe;
    t_opr_exe       curent_instruction;
    int             nbr_live;
    int             color_start;
    BOOL            a_live; //tempt enle
    struct s_process    *next;
}                   t_process;
```

Id_parent	Pour l'affichage
Current_pc	Addresse de l'instruction courante
Carry	Flag -> si true la dernier opt a reussi sinon a 0
Reg[REG_NB]	
Time_to_exe	
Current_instruction	
Nbr_live	
Color_start	
a_live	
Next	
Pc	

Cette struct en list chaine permet de gerer les

Processes qui tourneront sur la vm. Chaque process corespondra à un node de la list.

Cette technique permet de simuler le multi-threading (plusieurs process en paralelle)

```
typedef struct      s_opr_exe
{
    int             id_opr;
    int             nbr_param;
    int             type_arg[3];
    int             size_arg[3];
    int             vale_arg[3];
}                   t_opr_exe;
```

→ Struct permettant de stocker le rsl des operation

-Le code…

##detailler la lecture du champ

Si le parsing du champ est ok on peut alors lancer la vm :

```
void	ft_fight(t_vm *vm, t_process *list_process)
{
	t_op	*op_tab[NBR_OP];
	int 	cycle_to_die;
	int	time_total;
	int 	check;
	int 	time;

	ft_get_op_tab(op_tab);
	cycle_to_die = CYCLE_TO_DIE;
	time_total = 0;
	check = 1;
	while (cycle_to_die > 0)
	{
		time = 0;
		while (time < cycle_to_die)
		{
			if (time_total + time++ == vm->dump)
				ft_dump(vm);
			run_cycle(vm, list_process, op_tab);
		}
		time_total += time;
		if (!ft_check_survivor(list_process, vm))
			ft_put_winer(vm);
		if (ft_get_total_live(list_process) >= NBR_LIVE)
			cycle_to_die =- CYCLE_DELTA;
		if (check == MAX_CHECKS)
		{
			cycle_to_die -= CYCLE_DELTA;
			check = 1;
		}
		else
			check++;
	}
	ft_free_optab(op_tab);
}
```

La vm c’est cette boucle while dans un autre boucle while.

L’idee c’est de boucler temps que la limite de temps n’est pas fini OU jusqu’a la mort des champions OU jusqu a un dump (arret vonlontaire du prgm pour voir les log)

Nous allons maintenant detailer ce qui ce passe lors d un cycle dans la vm :

Mais avant petit recapitulatif : a ce stade la du code on a :

- un double tableau d'int (**mem) contenant en [0] les op codes (stockés en base decimal dans le tableau mais converti en hexa lors de l'affichage.) et en [1] l'idd corespondant au champions pour gerer la couleur lors de l affichage.
- Les processus stockés dans la liste chainé t_process (1 process créé par champion passé en param).

Tres bien maintenant que va t il se passer ?

Voila un petit visual du rendu de la vm

Mem[0] : 0e ff 0a 5b 00 00 00 00 00 00 00 00

Mem[1] : 1 1 1 1 0 0 0 0

Les instruction en hexa sont applele op code ce sont des "move" en asm qui seront executé par les process lorsque qu'ils paseront dessus.

Un opcode est constitué de trois parties : l'opcode qui dessigne l'operation à effectuer, l'adresse source et l'adresse de destination

Pour ordonnancer l'exécution des programmes, La vm fait du simple time-sharing : si il y a deux programmes X et Y qui s'affrontent, la première instruction de X est exécutée puis la première de Y, puis la deuxième de X, et ainsi de suite... L'exécution de chaque instruction prend un temps égal.

Notre procesus va donc arriver sur la case 0e. Il va aller voir dans la table de corespondance si dessous à quel move correspond son op code. Ici ce sera un lldi.

Operation mnemonique	Name	Nbr param	Cycle	param
0X01	live	1	10	000 000 100
0X02	ld	2	5	000 001 110
0X03	st	2	5	000 011 001
0X04	add	3	10	001 001 001
0X05	sub	3	10	001 001 001
0X06	and	3	6	001 111 111
0X07	or	3	6	001 111 111
0X08	xor	3	6	001 111 111
0X09	zjmp	1	20	000 000 100
0X0A	ldi	3	25	001 101 111
0X0B	sti	3	25	101 111 001
0X0C	fork	1	800	000 000 100

	0X0D		lld		2		10	000 001 110
	0X0E		lldi		3		50	001 101 111
	0X0F		lfork		1		1000	000 000 100
	0X10		aff		1		2	000 000 001

Le move s'effectura a la fin d'un timer comuns aux process qui tournent sur la map (je rappel qu'au debut de l'exectution il y a un process par champ). A noter que les processus sont iddependants du champion qui l'a crée. Il peut et va donc executer les moves de n importe quelle champion.

Donc si un processus tombe sur un op code a executer et que le timer est supperieur a 0, il devra attendre la fin du timer pour executer le code. Pendant ce temps les autres prosecus se deplacerons jusqua leurs destination suivante. Toutes les instruction serons donc execute Presque en meme temps.

A noter que qu'un opcode peut prendre plusieurs param. Le process englobera donc toute l'instruction comme sur le shema si dessous :

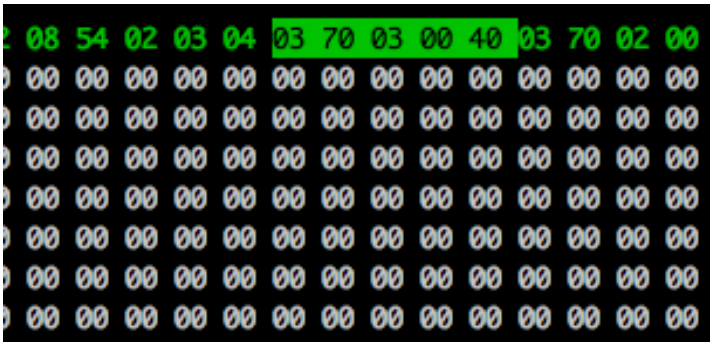
```
40 09 02 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

L'addressage memoire est relatif. Cela signifie qu'un programme ne peut pas calculer

operation table / descption param :

Operation mnemonique	Name	Nbr param	description
0X01	live	1	don't need the OCP
0X02	ld	2	need the OCP
0X03	st	2	need the OCP
0X04	add	3	need the OCP
0X05	sub	3	need the OCP
0X06	and	3	need the OCP
0X07	or	3	need the OCP
0X08	xor	3	need the OCP
0X09	zjmp	1	don't need the OCP
0X0A	ldi	3	need the OCP
0X0B	sti	3	need the OCP
0X0C	fork	1	don't need the OCP
0X0D	lld	2	need the OCP
0X0E	lldi	3	need the OCP
0X0F	lfork	1	don't need the OCP
0X10	aff	1	need the OCP

OCP -> octet de description



Exemple : ici on a comme opcode 03
On voit que dans notre table d'opration il correspond au move st qui implique 2 params et un OCP

Les arguments :

Une fois notre op code analysée par le processus il va aller recuperer les arguments (si bessoin)
On peut avoir jusqu a 3 parametres par moves. Chaque params peuvent prendre jusqu a 3 arg

- Les registres : codé sur 1 octet -> iddentifiant un registre
  Charge le contenu d'un registre et stock un val dedans (le registre doit exister et chaque processus a ses propes registres -> 16 par processus chez nous)

- Les index (indirect): codé sur 2 octets -> addresse d'un entier en ram
  Charge le contenu des registres et stock la valeur dans 4 octet en function du registre
- Les direct definie par une valeur et un modulo represente la valeur à l'index du registre correspondant

  Ex : st, R4 , -37 -> prend l'info dans le registre 4 et l'ecrit a l'addresse [pc – [37 % IDX_MOD] % MEM_SIZE]
  Ld %30, r8 -> prends la val 30 et la copie dans r8
  %4->direct
  4->indirect

Usefull link :

https://fr.wikipedia.org/wiki/Mode_d%27adressage#Adressage_relatif

http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Christophe.Filliatre/10-11/INF431/core_war/sujet.pdf

https://www.exploit-db.com/docs/french/13878-[french]-hzv-e-zine-2.pdf

http://nicolab.chez.com

A noter que la version du corewar demandé par 42 differe de la version originale