Machine virtuelle

Jump to bottom

VBrazhnik a modifié cette page le 3 janvier 2019 · 1 révision

Après avoir reçu les fichiers avec le bytecode champion, il est temps que la machine virtuelle s'exécute.

Fonctionnement

Lors du lancement du programme `corewar`, nous spécifions les `.cor` fichiers champions qui participeront à la bataille comme arguments :

```
$ ./corewar batman.cor ant-man.cor iron_man.cor
```

Au fait, il n'est pas nécessaire que ce soit trois fichiers différents avec trois champions différents. Nous pourrions démarrer la machine virtuelle en spécifiant trois fois le même fichier dans les paramètres. Par ex `batman.cor`.

Le programme ne fait aucune différence. Elle distingue les joueurs non pas par leurs noms, mais par les numéros uniques qu'elle attribue à chacun d'eux.

Pour elle, trois Batman sont `Player 1`, `Player 2` et `Player 3`.

Une constante garde une trace du nombre maximum de champions qui peuvent combattre simultanément en mémoire `MAX_PLAYERS`. Dans le fichier d'exemple, il est initialisé avec une valeur `4`.

Autrement dit, plus de 4 joueurs ne peuvent pas être chargés dans une machine virtuelle à la fois.

Drapeau `-n`

L'ordre des joueurs, ou plutôt l'ordre d'attribution des numéros d'identification, peut être modifié à l'aide du drapeau `-n`.

Dans la machine virtuelle d'origine, la `corewar` prise en charge d'un tel drapeau n'est pas implémentée, mais selon le texte de la tâche, elle doit être présente dans notre programme.

Ce drapeau attribue un numéro d'identification au joueur champion spécifié. Et les joueurs qui n'ont pas reçu un tel numéro en utilisant le drapeau se verront attribuer le premier des joueurs inoccupés:

> `-n number` définit le numéro du joueur suivant. S'il n'existe pas, le joueur aura le prochain numéro disponible dans l'ordre des paramètres.

Autrement dit, si vous exécutez cette commande ...:

```
$ ./corewar batman.cor -n 1 ant-man.cor iron_man.cor
```

... Ant-Man devient `Player 1`, Batman devient `Player 2` et Iron Man devient `Player 3`.

La seule chose à suivre dans ce cas est le numéro indiqué après `-n`. Il doit être supérieur ou égal au `1`nombre total de joueurs participant à la bataille, sans toutefois le dépasser.

De plus, ce numéro doit être unique dans la bataille, donc plusieurs joueurs ne peuvent pas avoir le même numéro.

Validation

La machine virtuelle lit les fichiers reçus avec le bytecode champion et les vérifie pour la présence d'un en-tête magique, si la taille de code spécifiée correspond à la taille réelle, et ainsi de suite.

À propos, si le programme de traduction n'a pas attaché d'importance à la taille maximale du code exécutable, alors pour la machine virtuelle, ce paramètre a une valeur et ne peut pas dépasser 682 octets:

```
# define MEM_SIZE           (4 * 1024)
// ...
# define CHAMP_MAX_SIZE     (MEM_SIZE / 6)
```

Il n'y a pas de limite minimale sur la taille du code exécutable pour une machine virtuelle, il peut donc être complètement absent.

Après avoir traité les données reçues sur chacun des joueurs, la machine virtuelle doit savoir ce qui suit:

- numéro d'identification unique
- nom du champion
- commentaire de champion
- taille du code exécutable en octets
- code exécutable

Initialisation de l'arène

Une fois que le programme s'est assuré que tout va bien avec les fichiers de lecteur fournis, vous devez initialiser l'arène où les codes exécutables du champion seront placés.

La taille de l'arène en octets est déterminée par une constante `MEM_SIZE`dont la valeur dans l'exemple est `4096`.

Pour comprendre où sera situé le code exécutable d'un joueur particulier, il est nécessaire de diviser la taille totale de la mémoire par le nombre de participants à la bataille.

Le résultat sera égal à la quantité de blocs de mémoire en octets, au début de chacun desquels se trouvera le code exécutable du champion correspondant:

```
4096 / 3 = 1365
```

Autrement dit, le code du premier joueur sera localisé à partir de la cellule de mémoire zéro et plus loin. Le deuxième code provient de la 1365e cellule. Et le troisième - à partir du 2730.

Organisation de la mémoire

La mémoire d'une machine virtuelle est organisée en anneau ou en boucle. Autrement dit, la dernière cellule est suivie de la première.

Et si la capacité de mémoire totale est de 4096 cellules, le nombre de la première cellule sera égal `0` et le dernier - `4095`.

En cas d'adressage d'une cellule avec un nombre supérieur à `4095`, le reste de la division modulo sera considéré comme un nombre valide `4096`. Ainsi, le dépassement de la mémoire est exclu.

Paramètres du jeu

De plus, avant de commencer le travail, vous devez définir la valeur des variables suivantes:

- joueur dont on a dit qu'il était en vie pour la dernière fois

C'est avec cette variable que le vainqueur de la bataille sera déclaré. Cette variable est initialisée avec le pointeur vers le joueur avec l'ID le plus élevé et mise à jour dans l'opération `live`.

- nombre de cycles passés depuis le début du jeu
- le nombre d'opérations effectuées `live` au cours de la dernière période de `cycles_to_die`
- `cycles_to_die` - la durée du délai avant le contrôle

Initialisé par une valeur constante `CYCLES_TO_DIE` qui est `1536`.

- nombre de contrôles effectués

Qu'est-ce que la vérification et quel est son rôle dans le fonctionnement d'une machine virtuelle sera discuté plus tard.

Initialisation des chariots

Une fois que les codes exécutables du champion ont été placés dans l'arène, un chariot est installé au début de chacun d'eux.

Il contient les données suivantes:

- numéro de chariot unique
- `carry`

Un drapeau que certaines opérations peuvent changer. Au départ, sa valeur est `false`.

- le code de l'opération sur laquelle se trouve le chariot.

Cette variable n'a pas été définie avant le début de la bataille.

- le cycle dans lequel l'opération a été effectuée pour la dernière fois `live`
- le nombre de cycles restant jusqu'à l'exécution de l'opération sur laquelle repose le chariot
- position actuelle du chariot
- le nombre d'octets qui devront être "enjambés" pour être sur la prochaine opération
- registres dont le nombre est spécifié dans une constante `REG_NUMBER`

> Lorsque le chariot est initialisé au début du jeu, les valeurs de ses registres sont définies. Dans `r1` sera enregistré le numéro d'identification du joueur, sur le code duquel se trouve la voiture. Seulement avec un signe moins. Et tous les autres registres seront initialisés avec des zéros.

Il est important de comprendre que le chariot ne fonctionne pas pour un joueur en particulier.

Elle exécute simplement le code sur lequel elle se trouve. Indépendamment de qui en est propriétaire.

Du point de vue du moteur de rendu fourni, le curseur sait de quel joueur il provient. Par conséquent, si au début du jeu il se trouvait sur le code exécutable du lecteur "vert", ou s'il a été généré par un tel chariot, alors l'écriture en mémoire à l'aide d'opérations `st` / `sti` sera également verte. Quelle que soit la couleur du code pour le moment.

Mais du point de vue de la machine virtuelle, le seul moment où le chariot et le joueur sont connectés de quelque manière que ce soit est le début du jeu. À ce moment, le numéro d'identification du joueur est inscrit dans le premier registre de chariot, uniquement avec un signe moins.

Plus tard, lors de l'exécution des opérations `fork` ou des `lfork` valeurs de tous les registres, le chariot "nouveau-né" recevra du chariot parent, et non du joueur.

Liste des voitures

Tous les carets forment une liste. Et c'est dans l'ordre dans lequel ils apparaissent dans cette liste qu'ils seront exécutés.

L'ajout de nouveaux éléments à la liste se fait par insertion au début. Par conséquent, avant le début de la bataille, il y aura une calèche en haut, qui se trouve sur le code du dernier joueur (le joueur avec le numéro d'identification le plus élevé):

```
Cursor 3 (Player 3)
        |
        V
Cursor 2 (Player 2)
        |
```

```
        V
Cursor 1 (Player 1)
```

À ce stade, tous les travaux préparatoires peuvent être considérés comme terminés.

Présentation du joueur

Avant que la bataille des joueurs champions reçus ne commence, vous devez déclarer:

```
Introducing contestants...
* Player 1, weighing 22 bytes, "Batman" ("This city needs me") !
* Player 2, weighing 12 bytes, "Ant-Man" ("So small") !
* Player 3, weighing 28 bytes, "Iron Man" ("Jarvis, check all systems!") !
```

Bataille

Règles de combat

L'une des variables les plus importantes dans une bataille est le nombre de cycles qui se sont écoulés depuis le début.

Au cours de chaque cycle, la liste complète des chariots est scannée. Et en fonction de l'état de chacun d'eux, certaines actions sont effectuées - un nouveau code d'opération est défini, le nombre de cycles avant l'exécution est réduit ou l'opération elle-même, sur laquelle se trouve le chariot, est effectuée.

La bataille continue tant qu'il y a au moins une voiture vivante.

Autrement dit, le chariot peut mourir ou quelqu'un peut le tuer?

Oui, la voiture peut mourir. Cela se produit lors d'un événement tel qu'un **contrôle** .

Vérifier

La vérification a lieu à chaque `cycles_to_die` cycle lorsque la valeur est `cycles_to_die` supérieure à zéro. Et une fois que sa valeur devient inférieure ou égale à zéro, le contrôle commence après chaque cycle.

Lors de cette vérification, les chariots morts sont supprimés de la liste.

Comment savoir si la voiture est morte?

Un chariot est considéré comme mort s'il a effectué des `live` `cycles_to_die` cycles en arrière ou plus.

Tout chariot est également considéré comme mort si `cycles_to_die <= 0`.

En plus de supprimer les chariots, la valeur est modifiée lors du contrôle `cycles_to_die`.

Si le nombre d' `cycles_to_die` opérations effectuées pendant la période est `live` **supérieur ou égal à** `NBR_LIVE` , la valeur est `cycles_to_die` diminuée de `CYCLE_DELTA`.

La valeur de la constante `NBR_LIVE` dans le fichier fourni est `21` et la valeur `CYCLE_DELTA` est `50`.

Si le nombre d'opérations effectuées est `live` inférieur à la limite définie, la machine virtuelle se souvient simplement qu'une vérification a été effectuée.

Si, `MAX_CHECKS` après vérification, la valeur `cycles_to_die` ne change pas, elle sera alors forcément diminuée de la valeur `CYCLE_DELTA`.

La valeur `MAX_CHECKS` du fichier d'exemple `op.h` est `10`.

Pour mieux comprendre quand la vérification a lieu et ce qu'elle change, prenons un exemple:

Cycles	Nombre d'opérations `live`	`cycles_to_die`	Nombre actuel de chèques
1536	5	1536	une
3072	193	1536 -> 1486	2
4558	537	1486 -> 1436	une
5994	1277	1436 -> 1386	une
7380	2314	1386 -> 1336	une
8716	3395	1336 -> 1286	une
...	...	...	...

> Le nombre d'opérations `live` est remis à zéro après chaque contrôle, quels que soient ses résultats.

> Le "nombre actuel de chèques" comprend le chèque en cours.

Il sera également utile d'envisager un autre scénario possible pour le développement de la bataille. Seulement dans ce cas, nous ne sommes intéressés que par les derniers cycles de bataille:

Cycles	Nombre d'opérations `live`	`cycles_to_die`	Nombre actuel de chèques
...	...	...	...
24244	41437	136 -> 86	une
24330	25843	86 -> 36	une
24366	10846	36 -> -14	une
24367	186	-14 -> -64	une

> Ces données peuvent être obtenues lors de l'exécution du champion Jinx, qui se trouve dans les archives en `vm_champs.tar` cours de route `champs/championships/2014/rabid-on/`.

À l'intérieur de la boucle

Regardons maintenant de plus près ce qui se passe à l'intérieur de la boucle.

Dans chaque cycle, la machine virtuelle parcourt la liste des chariots et effectue les actions nécessaires sur chacun d'entre eux:

Définit l'opcode

Si au cours du dernier cycle le chariot s'est déplacé, alors il faut établir sur le code de quelle opération il s'agit maintenant.

> A propos, une variable mémorisant le nombre de cycles avant l'exécution de l'opération peut indiquer que le chariot s'est déplacé dans le cycle d'exécution précédent. Si sa valeur est zéro, alors le mouvement a eu lieu et le chariot doit définir le code d'opération sur lequel il se trouve maintenant.

> Lors du tout premier cycle d'exécution, tous les chariots sans exception recevront les valeurs de l'opcode. Puisqu'il n'est pas installé lors de l'initialisation.

Pour connaître l'opcode, vous devez lire l'octet sur lequel se trouve le chariot.

Si le numéro reçu correspond au code de l'opération réelle, il doit être mémorisé.

Vous devez également définir la valeur d'une variable qui stocke le nombre de cycles avant l'exécution de l'opération.

Vous n'avez pas besoin de lire et de mémoriser d'autres données sur l'opération (code des types d'arguments, arguments). Ils doivent être reçus au moment de l'opération.

Si vous vous en souvenez auparavant, la machine virtuelle ne fonctionnera pas correctement. En effet, pendant le temps d'attente, les cellules correspondantes de la mémoire peuvent être écrasées par de nouvelles valeurs.

Mais si le nombre lu ne tombe pas dans la plage [`0x01` ; `0x10`], c'est-à-dire que le code reçu indique une opération inexistante? Comment être dans une telle situation?

Dans ce cas, vous devez vous souvenir du code lu et laisser la valeur de la variable stockant le nombre de cycles jusqu'à ce que l'exécution soit laissée égale à zéro.

Réduisez le nombre de cycles avant l'exécution

Si le nombre de cycles avant exécution, qui est stocké par la variable correspondante dans le chariot, est supérieur à zéro, il faut le diminuer de 1.

> Il est important que pendant le cycle, toutes les vérifications et actions décrites soient exécutées strictement dans l'ordre spécifié.
>
> Parce que dans un cycle, le même chariot peut recevoir un nouvel opcode et définir le nombre de cycles avant son exécution. Et réduisez également ce montant de un.
>
> S'il y avait une opération avec un seul cycle d'attente, alors elle serait également exécutée pendant ce cycle.

Réaliser une opération

Si le nombre de cycles avant exécution est nul, alors il est temps d'effectuer l'opération dont le code est stocké par le chariot.

Si le code stocké correspond à une opération existante, vous devez alors vérifier la validité du code contenant les types d'arguments.

Si ce code est correct et indique qu'il existe un registre parmi les arguments d'opération, vous devez également vous assurer que le numéro de registre est correct.

Si toutes les vérifications nécessaires ont été réussies, vous devez effectuer l'opération et déplacer le chariot vers la position suivante.

Si l'opcode est incorrect, il vous suffit de déplacer le chariot vers l'octet suivant.

Si tout va bien avec le code lui-même, mais que le code des types d'argument ou le numéro de registre est faux, vous devez ignorer cette opération avec le code des types d'argument et les arguments eux-mêmes.

Mais si tout est facile avec l'omission d'un opcode incorrect, il vous suffit de passer à l'octet suivant. Avec le mouvement après l'opération, tout est encore transparent - les types d'arguments, ainsi que leurs tailles, sont connus. Ensuite, dans le cas d'un code de type d'argument incorrect, une question importante se pose: "Comment ignorer les arguments d'opération si le code de type d'argument est incorrect?"

Vous devez ignorer l'opcode, le code de type des arguments et les arguments spécifiés dans le code de type.

> C'est pourquoi les tailles des arguments ont été spécifiées dans le tableau des opérations `T_DIR` même pour les opérations qui n'acceptent pas de tels arguments.

Disons que le code de nos opérations est `0x04` :

Code d'opération	Nom de l'opération	Argument n ° 1	Argument n ° 2	Argument n ° 3
4	`add`	`T_REG`	`T_REG`	`T_REG`

Mais la valeur de l'octet contenant les types d'argument est `0xb6` :

Argument n ° 1	Argument n ° 2	Argument n ° 3	-
`10`	`11`	`01`	`10`
`T_DIR`	`T_IND`	`T_REG`	`T_DIR`

Puisque cette opération prend trois arguments, nous ne considérons que les valeurs des trois premières paires de bits dans le code de type. Nous ne sommes pas intéressés par les valeurs des autres paires.

Après un court calcul, nous apprenons que dans cette situation, nous devons sauter les 9 octets suivants: opcode (1 octet) + octet avec types (1 octet) + argument de type `T_DIR` (4 octets) + argument de type `T_IND` (2 octets) + argument de type `T_REG` (1 octet)).

Drapeau `-dump`

En général, c'est tout ce qu'il y a à savoir sur le fonctionnement d'une machine virtuelle. Mais il existe un autre indicateur important qui peut arrêter le processus d'exécution. Ceci est un drapeau `-dump` .

L'essence de son travail est décrite dans le texte de la mission dans les lignes suivantes:

> `-dump nbr_cycles`
>
> à la fin des `nbr_cycles` exécutions, videz la mémoire sur la sortie standard et quittez le jeu. La mémoire doit être vidée au format hexadécimal avec 32 octets par ligne.

Un analogue de cet indicateur est également présent dans la machine virtuelle d'origine. Seulement sous un nom différent - `-d` .

Les deux drapeaux reçoivent le numéro de cycle, après quoi il est nécessaire d'afficher l'état de la mémoire à l'écran et de terminer le programme `corewar` . Mais les modes d'affichage du contenu de l'arène pour ces deux drapeaux sont légèrement différents.

L'indicateur `-d` affiche des valeurs de 64 octets d'affilée. Et le drapeau `-dump` ne fait que 32 octets.

▸ **Des pages** neuf

1. introduction
2. Fichiers fournis
3. Assembleur
4. Analyse lexicale
5. Assemblage en Bytecode

6. Démontage
7. Machine virtuelle
8. Visualisation

Cloner ce wiki localement

```
https://github.com/VBrazhnik/Corewar.wiki.git
```