Fichiers de commandes

Mirjana Stojilovic & Jean-Cédric Chappelier 2019

Cette semaine vous allez devoir mettre en place les composants permettant la simulation de « programmes » c.-à-d. de séquences d'instructions que le processeur devrait exécuter. Cela nous permettra de simuler des scénarios d'accès à la mémoire. Un « programme » sera simplement un tableau de commandes ; cette semaine, ce sera un tableau statique ; il sera transformé en tableau dynamique dès la semaine prochaine.

Ce que vous avez concrètement à faire cette semaine est de :

1. définir 3 types dans le fichier `commands.h` fourni ;
2. créer le fichier `commands.c` et y définir les 5 fonctions nécessaires (mais c'est le gros du travail ; à se répartir entre les membres du groupe) ;
3. [optionnel mais conseillé] créer un fichier de tests unitaires.

I. Description générale d'un « programme »

En guise d'illustration, commencez par lire/ouvrir le fichier fourni `tests/files/commands01.txt`.

Les « programmes » seront des séquences de commandes. Les commandes se composent (dans cet ordre) :

- d'une indication d'accès en lecture (**R**ead) ou en écriture (**W**rite) ;
- du type d'accès désiré : vers des données (**D**) ou des instructions (**I**) ;
- si l'on accède à des données, de la nature de ces données : mot (**W**ord) ou octet (**B**yte) ;
- si l'accès est en écriture (uniquement pour des données), de la valeur à écrire ;
- de l'adresse virtuelle où accéder.

II. Types

Dans le fichier `commands.h`, commencez par définir `command_word_t` comme (alias vers) un type énuméré représentant les sortes d'accès possibles à la mémoire : `READ` (accès en lecture) ou `WRITE` (accès en écriture).

Rappel de la note de la semaine passée : nous prenons la convention de nommer tous les types suivant le schéma `X_t`. Pour les types comme des structures ou des types énumérés, il s'agit donc d'alias (vers la `struct` ou l'`enum`). Par exemple le type `virt_addr_t` est en fait un alias vers le `struct` « *bitfield* » correspondant.

Définissez ensuite le type `command_t` comme (un alias de) structure contenant les champs suivants :

- `order` de type `command_word_t` ; il indiquera le mode d'accès désiré à la mémoire (lecture ou écriture) ;
- `type` de type `mem_access_t` (défini dans `mem_access.h`) ; il indiquera the type d'information recherchée (instruction ou donnée) ;
- `data_size` de type `size_t` ; il indiquera la taille, en octets, des données manipulées (mot ou octet) ;
- `write_data` de type `word_t` (défini dans `addr.h`) ; il contiendra, lorsque nécessaire, la valeur à écrire ;
- `vaddr` de type `virt_addr_t` ; il indiquera l'adresse virtuelle où accéder.

Définissez enfin le type `program_t` comme (un alias de) structure contenant les champs suivants :

- `listing` comme un tableau *statique* de 100 `command_t` ; il sera transformé la semaine prochaine en un pointeur pour une allocation dynamique ;
- `nb_lines` de type `size_t` représentant le nombre de lignes du programme, c.-à-d., le nombre de commandes effectivement utilisées dans `listing` ;
- `allocated` de type `size_t`, inutile pour le moment mais qui représentera dès la semaine prochaine la taille allouée à `listing` (qui pourra être plus grande que la taille strictement nécessaire à utiliser).

III. Fonctions

Commencez par regarder la suite du fichier `commands.h` pour déjà avoir une première idée du travail à faire. Le plus gros du travail est `program_read()` et une répartition possible du travail serait de que l'un des deux s'occupe de cette fonction et l'autre de tout le reste (y compris les types ci-dessus). Mais vous êtes bien sûr libres de vous organiser comme bon vous semble.

III.1 `program_init()`

Le fonction `program_init()` doit simplement mettre à 0 tout le champs `listing` ainsi que le champ `nb_lines` et mettre le champ `allocated` à la taille mémoire de `listing` (simplement : `sizeof(pgm_p->listing)`).

III.2 `program_print()`

La fonction `program_print()` a pour but d'écrire un `program_t` dans le flot passé en premier paramètre.

Si par exemple vous deviez écrire le contenu du fichier fourni `tests/files/commands01.txt`, cela donnerait (une commande par ligne) :

```
R I @0x0000000000000000
R DW @0x0000000040200000
R DB @0x0000000040200002
W DB 0xAA @0x0000000040000005
W DW 0x0000BEEF @0x0000000040000010
```

(Il y a bien un saut de ligne après chaque commande, y compris la dernière.)

Il pourrait être utile de modulariser cette fonction...

Pour afficher un `word_t` en tant que mot (32 bits), utilisez `"0x%08" PRIX32`, p.ex. :

```
prinft("un mot : 0x%08" PRIX32, mot);
```

Pour l'afficher simplement en tant qu'octet (8 bits), utilisez `"0x%02" PRIX32`.

Et pour afficher un `uint64_t`, utilisez `"0x%016" PRIX64`.

III.3 `program_shrink()`

A part vérifier la validité de son argument, cette fonction ne doit encore rien faire cette semaine. Elle sera utile dès la semaine prochaine lorsque nous aurons l'allocation dynamique.

III.4 `program_add_command()`

Le but de la fonction `program_add_command()` est simplement d'ajouter une nouvelle commande au programme. Mais tout son intérêt consiste à vérifier la validité de la commande avant de l'ajouter et retourner un code d'erreur (sans l'ajouter) si elle n'est pas valide, p.ex. (non exhaustif) :

- la taille des informations accédées n'est pas correcte : pour des données, elle doit être 1 (octet) ou `sizeof(word_t)` (mot) ; pour des instructions, c'est forcément `sizeof(word_t)` (les instructions sont nécessairement des mots du processeur) ;
- on cherche à écrire une instruction (on ne peut que lire des instructions) ;
- l'adresse virtuelle n'est pas correcte (son offset doit être un multiple de la taille des données utilisées [octet ou mot]).

Par ailleurs, si le programme (alloué statiquement cette semaine) est déjà plein (100 commandes), alors il faut renvoyer le code d'erreur `ERR_MEM`.

III.5 `program_read()`

La description de la fonction `program_read()` est triviale, mais sa réalisation peut être complexe si l'on ne s'y prend pas correctement, avec ordre.

Le but de la fonction `program_read()` est simplement de transformer le contenu d'un fichier texte décrivant un « programme_» (tel que défini en section I et illustré par les fichiers `tests/files/commands01.txt` et `tests/files/commands02.txt` fournis) en une structure `program_t` correctement remplie. Cette fonction devra envisagé le plus de cas d'erreur possible et retourner un code d'erreur approprié le cas échéant.

Pour écrire cette fonction, nous vous conseillons vraiment de procéder par ordre, de *décomposer* la tâche et d'écrire des fonctions auxiliaires, comme, par exemple :

- lire une commande ;
- lire jusqu'au prochain caractère non blanc (`man isspace`) ;
- traiter un « read » ;
- traiter un « write » ;
- etc.

Pensez à chaque fois à tester (et reporter) les cas d'éventuelles erreurs.

IV. Tests et debuging

Pour vous aider dans vos recherches de bugs, nous vous conseillons en plus d'utiliser un débogueur, de compiler en mode « debug » (ajoutez `-DDEBUG` à vos commandes de compilation) et d'utiliser dans vos programme la « fonction » `debug_print()` fournie dans `error.h` (allez voir) pour afficher ce qui se passe dans vos programmes.

Comme indiquez au début de ce document, nous vous conseillons (comme toujours) d'écrire vos propres tests unitaire pour tester vos fonctions.

Nous avons de plus fourni un fichier `test-commands.c` qui ne fait simplement que lire un programme de commandes et le réécrire, mais cela devrait vous permettre de tester « de bout en bout » vos fonctions. Compilez-le et lancez-le par exemple comme suit :

```
./test-commands tests/files/commands01.txt
```

et comparez le résultat obtenu avec le contenu du fichier `tests/files/commands01.txt`. Vous pouvez même lancer le script de test `05.basic.sh` :

```
./tests/05.basic.sh
```

qui fait exactement cela (lancer `test-commands` et comparer) sur les deux fichiers `commands01.txt` et `commands02.txt`.

V. Conseil et rendu

ATTENTION : il y a *beaucoup* de sources d'erreurs possibles dans toutes ces fonctions. Il faut bien systématiquement vérifier tous les arguments, et aussi les valeurs de retour des fonction outils utilisées dans une fonction de plus haut niveau. Il faut imaginer que chaque fonction peut être testée séparément avec les entrées « *bizarres* » (ce que nous ne nous priverons pas de faire). Si vous avez des questions, n'hésiter pas à profiter des séances d'exercices pour demander.

Concernant le rendu, comme la semaine passée le travail de cette semaine ne sera pas évalué en tant de tel (c.-à-d. seul), mais devra faire partie du premier rendu intermédiaire du projet (délai : le dimanche 14 avril 23:59). Néanmoins, nous vous conseillons fortement de travailler régulièrement et faire des commits réguliers comme indiqué la semaine passée.