TD 1 nachos

Master 1 informatique université de Bordeaux 2019-2020

par Julien Dauliac, Thomas Veloso

I. Bilan

Durant ce premier devoir, nous avons appris à nous familiariser avec le système Nachos, permettant plusieurs fonctionnalités, comme un mode console asynchrone, mais aussi l'exécution de test avec des appels systèmes.

I.1 Consoles Asynchrones

Avant de commencer à mettre en place les premiers appels systèmes, nous avons manipulé les consoles asynchrones, par le biais du programme `ConsoleTest`.
Dans ce programme, nous avons modifié plusieurs choses, comme l'ajout de chevrons autour des caractères que l'on saisit, ou encore la fermeture du programme en appuyant sur la touche « q ».

Ensuite, nous avons mis en place les mécanismes d'entrées-sorties synchrones, dans la classe `SynchConsole.cc`.

Les méthodes complétées à ce stade sont `SynchGetChar` et `SynchPutChar`, le but étant d'avoir le même comportement que les méthodes de la classe Console, GetChar et PutChar. Les différents tests que nous avons effectué sont fonctionnels.

I.2 Appels systèmes

I.2.1 PutChar

On a mis en place différents appels systèmes, à commencer par l'appel système `PutChar`. Pour ce faire, on modifie d'abord le fichier syscall.h pour y ajouter la définition de l'appel système et de la fonction C `PutChar(char c)`.

Ensuite on modifie le fichier `start.S` pour y ajouter le code en assembleur, en s'inspirant de ce qui a été fait pour implémenter l'appel système `Halt`.
Ce code en assembleur permet de récupérer la valeur de l'appel système dans le registre 2, et aussi de passer en mode noyau, qui nous permet de gérer les différents appels systèmes dans le fichier `exception.cc`.

Dans le fichier start.S, à l'initialisation du programme, la valeur du paramètre de la fonction `PutChar(char c)`, est transmise dans le registre 4. On va donc pouvoir se servir du registre 2 pour savoir quel appel système a été lancé, et du registre 4 pour récupérer le paramètre « c » de la fonction `PutChar(char c).`

Ensuite il nous reste juste à placer l'exception dans `exception.cc` en rajoutant un cas pour l'appel système `PutChar`.
Dans ce cas, nous récupérons la valeur du registre 4, que nous passons ensuite en paramètre à la méthode `SynchPutChar` qui prend en paramètre un caractère.
Enfin, afin de pouvoir exécuter et tester notre premier appel système, on déclare en global la variable `synchConsole` de type `SynchConsole` dans le fichier `system.cc`. On initialise cette variable dans la méthode `Initialize()`, puis on l'a détruit dans la méthode `Cleanup()`.

I.2.2 PutString

Maintenant que l'appel système `PutString` fonctionne, il nous est demandé de mettre en place l'appel système `PutString`. Après avoir complété la méthode `SynchPutString`, qui boucle simplement sur la méthode `\0` jusqu'à recevoir le caractère de fin de chaîne `\0`, nous mettons en place la méthode `copyStringFromMachine`, méthode que nous choisissons de placer dans la classe `SynchConsole`.

Cette méthode lit tous les caractères de la chaîne mips à l'aide de la méthode `ReadMem` de la classe `Machine`, en faisait un ajoutant `+1` à l'adresse lue à chaque tour de boucle. Ainsi, on remplit progressivement le tableau de caractères passé en paramètre, et on retourne la taille de la chaîne écrite à la fin de `copyStringFromMachine`.

Après avoir ajouté, comme pour l'appel système `PutChar`, le code assembleur dans le fichier `start.s` et l'appel système dans le fichier `syscall.h`, on compléte le fichier exception.cc afin de gérer le cas `SC_PutString`. On récupère la valeur du registre 4, puis on alloue un buffer de taille `MAX_STRING_SIZE`. Ensuite on appelle les méthodes copyStringFromMachine et `SynchPutString`, et on désalloue le buffer pour finir.

L'implémentation actuelle est correcte, elle est par contre limitée par la valeur de la constante `MAX_STRING_SIZE`. C'est pourquoi nous avons modifié la mécanique dans le fichier `exception.cc`, afin de gérer le cas où `MAX_STRING_SIZE` est inférieure à la chaîne de caractères que l'on passe en paramètre. Au lieu d'appeler une seule fois les méthodes `copyStringFromMachine` et SynchPutString, nous les appelons autant de fois que cela est nécessaire pour afficher la totalité de la chaîne de caractères. Nous utilisons la valeur retournée par la méthode `copyStringFromMachine` au sein d'une boucle, tant que cette valeur est égale à `MAX_STRING_SIZE`, on sait que l'on a pas encore récupéré la totalité de la chaîne.

L'utilisation d'un buffer instancié par la variable `from` et `to` permet de **ne pas**

charger l'entièreté de la string en mémoire mais de la lire **grâce à un tuyau**. Ceci évite de remplir la ram si un utilisateur saisit une string de 2 tera octet par exemple.

L'appel système `PutString` étant maintenant fonctionnel, on a aussi modifié la mécanique de finition du programme. Pour ce faire on a modifié `Start.S` dans la clause `__start`, où on déplace la valeur du registre 2 dans le registre 4, alors qu'auparavant la valeur du registre 0 était déplacer dans le registre 4.
On peut tester la valeur de retour qui est dans notre fichier de test `putstring.c`.

L'implémentation de cette mécanique se trouve dans le fichier `exception.cc`, dans le cas où l'appel système `SC_Exit` est appelé, on retourne un message d'erreur si la valeur du registre 4 est différente de 0, et ensuite mettre fin au programme. Ceci nous évite de faire appel directement à l'appel système halt dans notre de fichier de test 'putstring.c'.

I.2.3 GetChar

Ensuite nous avons mis en place l'appel système `getChar`. De la même manière que pour les autres appels systèmes, on modifie les fichiers `start.S` et `syscall.h`. La méthode `SynchGetChar` étant déjà existante, il nous reste juste à compléter le fichier exception.cc, pour le cas `SC_GetChar`, en appelant `SynchGetChar` et en écrivant dans le registre 2 le résultat obtenu.

I.2.4 GetString

De la même manière que pour l'appel système putstring, on met en place l'appel système getstring. Pour ce faire nous créons une méthode `copyStringToMachine`, permettant de passer une chaîne Linux en chaîne MIPS, et une méthode `SynchGetString` qui boucle simplement sur la méthode `SynchGetChar` tant que des caractères sont à lire.

De la même manière que pour `putString`, on a mis en place une boucle qui appel `SynchGetChar` et `copyStringToMachine` tant que `copyStringToMachine` renvoie la même taille que `MAX_STRING_SIZE`.

Dans le cas où plusieurs appels simultanés ont lieu sur le même appel système, il y aura une erreur. Cette gestion de plusieurs thread n'a pas été mis en place actuellement.

I.2.5 PutInt

Le dernier appel système mis en place est `PutInt`, pour celà on utilise la fonction `snprintf` qui écrit un entier dans un tableau de caractères.

II. Difficultés rencontrées

Lors de nos développements, nous avons eu de nombreux problèmes liés à des défauts de segmentation à cause d'une mauvaise allocation de la mémoire. Un en particulier nous a fait perdre du temps, on avait instancié par erreur deux fois un objet de type `synchConsole`.

Nous avons longuement discuté sur l'implémentation des boucles dans les différentes méthodes, notamment sur le choix d'opter pour une boucle `while` plutôt qu'une boucle `for`. Le choix s'est plus porté sur les boucles `while` par soucis de lisibilité et de compréhension du code.

L'utilisation de la constante `MAX_STRING_SIZE` à été sources de débats, nous ne savions pas vraiment si cela était préférable de l'utiliser au sein même des méthodes de la classe `SynchConsole` ou au sein appels de ces méthodes dans le fichier exception.cc.
Si l'on avait opté pour la première solution, donc directement utilisé dans la classe `SynchConsole`, cela aurait voulu dire que le 3 ème paramètre des méthodes copyStringFromMachine et `copyStringToMachine`, size, n'aurait servi à rien puisqu'il aurait été "surchargé" par `MAX_STRING_SIZE`. Nous avons donc opté pour une utilisation de `MAX_STRING_SIZE` dans les appels des méthodes dans le fichier exception.cc

III. Limitations

Si plusieurs appels au même appel système se font, il y aura une erreur. Cela est dû à une gestion des sémaphores et des threads pas assez poussé actuellement.

Pour l'appel système `getString`, dans le fichier getstring.c se trouvant dans le dossier `test`, on initialise un tableau de caractères de taille `n`. Selon que la valeur de `n` est plus ou moins grande, nous avons un comportement différent. Si la chaîne de caractères entrée est plus grande que `n`, nous obtenons une erreur, probablement dû à la boucle `while` que l'on fait dans le cas `SC_GetString` dans le

cas où la chaîne est plus grande que `MAX_STRING_SIZE`. Nous avons tenté de trouver la solution à ce problème sans pour parvenir à le résoudre actuellement. C'est le problème le plus gênant parmis toutes nos implémentations.

Le EOF Ne fonctionne pas et nous devons insérer un char q dans nous fichiers `in` `out`

IV. Tests

IV.1 Les chars

Pour les tests sur les chars, nous avons mis en place des tests sur des caractères normaux (a,b,c), ainsi que sur les caractère d'échappement comme le `\n` ou le `\0`. Nous mettrons ces testés de manière systématique pour `PutChar` et `GetChar`, dans le `progtest.cc` et les fichiers de testes en `C`.

Commandes:

Nous testerons avec le `progtest.cc` et les fichiers de tests. Les commandes sont:

- `./nachos -sc in out` Pour les entrées sortie par fichiers.
- `./nachos -sc` Pour les entrées sortie par clavier
- `./nachos -x test/putchar` ou `./nachos -x test/getchar`

IV.2 Les string

Dans le cas des string, la méthode sera la même que pour les chars.
Exception faite que nous testerons aussi des strings de différentes tailles:

- Supérieure à `MAX_STRING_SIZE`
- Inférieur à `MAX_STRING_SIZE`
- Égale à `MAX_STRING_SIZE`

Commande:

Les commandes de tests sont:

- `./nachos -x test/putstring` ou `./nachos -x test/getstring`

IV.3 Exit et return

Notre appel système `Exit` est capable de gérer les erreurs et les affiches si elles sont `!= 0`. Nous testerons donc les cas ou il n'y a pas de `return`, ou return renvoie 0 et ou return renvoie 1 par exemple.

Commandes:

Pour les tests nous aurons juste à exécuter les tests de putstring et getstring.