Rapport du TD01 de Système d'Exploitation

CAYROL Pierre
FURELAUD Thomas

Septembre-Octobre 2019

1 Bilan

Lors de cette première partie de projet, nous avons réussi à implémenter les fonctions principales demandées (GetString, PutString, PutChar, GetChar) ainsi que leurs appels systèmes. Nous avons aussi fait un des bonus, qui était d'implémenter *PutInt* et *GetInt*.
Les fonctions "secondaires" qui font que ces appels puissent s'exécuter sont aussi codé, telles que *copyStringFromMachine* ou encore *copyStringToMachine*.
De notre connaissance ainsi que de nos tests, tout fonctionne correctement mis à part *getString* et *getInt*. En effet, lorsque nous faisons un getString sur une chaîne de caractères qui est plus longue que la taille du tableau que l'on a alloué pour stocker le résultat, et que nous voulons appeler de nouveau *getString* alors celui-ci prend les caractères qui n'ont pas été récupérer lors du premier appel. La même chose se passe pour getInt. Nous n'avons pas trouvé de solution à ces problèmes.
Pour ce qui est de la partie tests, nous avons testé appel par appel dans le dossier test, en appelant l'appel correspondant dans le fichier en question (*GetString* dans getstring.c). Le point qui n'a pas été fait est la dernière instruction du point 6 (prendre compte de la valeur de retour de la fonction n).

2 Points délicats

Les points qui nous ont le plus posés problème sont les fonctions *copyStringToMachine* et *copyStringFromMachine* ainsi que le buffer. Il nous a été difficile de bien comprendre comment devaient fonctionner ces deux méthodes mais une fois compris, l'implémentation n'a pas été compliquée.
Le buffer dans exception.cc nous a posé quelques problèmes (pour la méthode *PutString*). Nous avons donc décidé d'allouer un tableau de caractère de la taille d'une constante définie (MAX_STRING_SIZE) dans systeme.h. Ensuite on essaie de lire MAX_STRING_SIZE caractères. Si la valeur renvoyée par *copyStringToMachine* est différente de MAX_STRING_SIZE alors on a fini de lire nos caractères. Alors on appelle *PutString* et on vide le buffer.

3 Limitations

Le principal avantage de notre implémentation est que l'on peut écrire un texte avec *putString* d'une longueur très grande sans avoir de soucis (du moins à notre connaissance). Un autre avantage se trouve dans le fichier exception.cc. Nous avons décidé d'implémenter les méthodes *copyStringToMachine* et *copyStringFromMachine* dans ce fichier car si plus tard, pour une raison quelconque nous voulons l'utilier avec une console asynchrone alors nous le pourrons.
Le plus gros inconvenient de notre implémentation est, comme dit dans la partie 1, l'appel à *getString* ainsi que *GetInt* et *PutInt*. En effet nous ne pouvons pas appeler plusieurs fois cette fonction à la suite si l'on dépasse la taille du tableau que l'on a alloué. Une des solutions serait de renvoyer une erreur en disant que la taille du tableau a été depassée, ou qu'un int est demandé pour *GetInt*. Un autre inconvénient est que l'on ne prend pas en compte la valeur de retour du main. Mais cet inconvénient nous semble moins gênant.

4 Tests

Pour ce qui est des tests, nous avons tout simplement créé un fichier par méthode ajoutée. Dans chaque fichier nous testons plusieurs fois notre fonction. Pour *PutString* nous l'appelons une fois avec une chaîne de caractères de taille inférieur à MAX_STRING_SIZE(taille du buffer) et une autre fois avec une chaîne plus grande que la taille du buffer. De même pour l'appel *GetString*.

5 Réponses des questions posées sur le sujet

5.1 Partie 1

La sortie raisonnablement attendue pour ce programme est "abcd". En effet, la fonction récupère le code ASCII du caractère donné (ici 'a') puis ajoute 1 à chaque tour de boucle (comme celui-ci est inférieur à 80). Puis à chaque tour de boucle il écrit le caractère. Le premier tour étant à 0, le code ASCII de 'a' + 0 est égal à 'a', puis le second étant 1, alors le code de ASCII de a + 1 est 'b' et ainsi de suite. Jusqu'à 'd' car le nombre de tours est le deuxième paramètre (soit 4) de la fonciton print.

5.2 Partie 2

C'est une erreur que de chercher à lire un caractère avant d'être averti qu'un caractère est disponible, ou de chercher à écrire avant d'être averti que l'écriture précédente est terminée car on risquerait d'écraser les données de l'ancien caractère. Si un caractère n'a pas encore renvoyé de signal alors celà veut dire que son écriture n'est pas fini. En voulant de nouveau écrire un caractère avant que celui-ci n'ait renvoyé un signal on risque alors d'écraser l'information du premier caractère.

La fonction ConsoleTest a pour but, comme son nom l'indique de tester la console asynchrone. Elle crée donc deux sémaphores ainsi qu'une console. Puis dans une boucle infinie elle attend que l'utilisateur écrive un caractère grâce à readAvail et récupère le caractère retourné dans une variable grâce à getChar. Ensuite, grâce à la fonction PutChar, elle écrit le caractère récupérer dans le terminal et attend que son écriture soit fini grâce à la fonction writeDone. readAvail et writeDone étant des sémaphores ils sont donc bloquant tant que la condition n'est pas remplie. Si le caracère relevé est q alors on sort de la boucle infinie, on libère les sémaphores et la console.

5.3 Partie 5

Il ne serait pas raisonnable d'allouer un buffer de la même taille que la chaîne MIPS car si la chaîne est vraiment très grande on allouerait trop d'espace mémoire pour seulement écrire du texte.

5.4 Partie 6

Si l'on enlève l'appel à la fonction Halt() à la fin de la fonction main dans putchar.c, une erreur apparaît ("Unimplement system call 1"). L'erreur, comme indiquée, se trouve dans le fichier exception.cc. En effet, dans la fonction ExceptionHandler, la fonction qui est donc appelée lorsqu'un programme côté utilisateur est exécuté, nous avons un switch case. A ce stade du projet nous avons 3 cas. Le SC_HALT (y été par défaut), et SC_PutChar et SC_PutString que nous avons rajouté. Le cas par défaut retourne, quant à lui, une erreur (grâce à "ASSERT(FALSE);") ainsi qu'un message d'erreur ("Unimplement system call" suivi du numéro de registre).
Pour éviter cette erreur nous pouvons tout simplement mettre le case de SC_Halt comme cas par défault. Ainsi ce cas sera appelé si aucun autre appel système n'est fait et donc nous n'aurons plus besoin d'appeler explicitement Halt().