Rapport TD1 OS

Valentin Jonquière, Mathilde Chollon

6 octobre 2024

Table des matières

1 Bilan

Pour ce premier projet, nous avons décidé de tout faire les 4 premières parties ensemble afin de s'habituer à utiliser NACHOS et mieux se répartir le travail sur les Parties plus chronophages. Ces parties ne posant pas de problèmes, car elles étaient détaillées pas à pas dans le sujet, elles n'ont donc pas posé de problèmes particuliers et sont implémentés sans bug connu de notre part. Nous avons donc choisi d'axer ce rapport sur les parties V, VI, VII et VIII qui nous ont posé plusieurs problèmes et demandé des choix d'implémentation non triviaux.

— Dans la partie V, nous avons remarqué que notre implémentation ne prend pas en compte les caractères spéciaux dans la fonction *PutString* si ceux-ci sont entrés 'en dur' dans chaîne de caractères.
— Lors de l'implémentation de la partie VIII, tous les tests que nous avons mis en place fournissent les résultats attendus. Cependant, il y a de forte chances que nous ayons omis des cas qui pourraient casser notre code.
— Notre fonction *printf* ne gère pas non plus les caractères spéciaux.

2 Points Délicats

2.1 Partie II

C'est une erreur de chercher à lire un caractère avant d'être averti pour plusieurs raisons. Il n'y a peut-être pas de caractère tapé, et on ne peut pas lire un caractère nul puisqu'il n'existe pas. L'écriture du caractère précédant pourrait ne pas être finie, et dans ce cas-là, on pourrait avoir la lecture d'un caractère qui n'est pas le nôtre. De plus, on peut faire des combos de touches pour taper un caractère. Si on lit avant que l'on ait fini, on se retrouvera avec le mauvais caractère.

Lorsque nous avons implémenté le *ConsoleTest*, nous avons rencontré un problème. Nous avons eu des <>en trop. En réfléchissant un peu mieux, nous nous sommes rendu compte que cela venait du retour charriot (\n). Nous avons donc rajouté une condition : si l'utilisateur tape un retour à la ligne, nous avons décidé de le faire sans mettre de chevrons pour avoir un affichage dans la console plus lisible.

2.2 Partie III

Cette partie n'a pas particulièrement posé de problème. Ayant analysé *ConsoleTest* dans la partie précédante, nous avons rapidement implémenté *PutChar* et *GetChar*. En ce qui concerne les chevrons nous avons adopté la même technique que pour *ConsoleTest* pour *ConsoleDriverTest* afin d'avoir un résultat plus lisible dans la console.

2.3 Partie V

Nous avons passé du temps à réfléchir à la fonction *copyStringFromMachine*. Nous avons surtout réfléchi à l'endroit où placer cette fonction. Cela n'avait pas de sens de le mettre dans *test* car cela correspond au monde utilisateur et le mettre dans la machine ne nous plaisait pas non plus étant donné que nous ne devons pas modifier les fichiers à l'intérieur. Comme c'est une fonction qui doit accéder au monde noyau, nous avons donc décidé de la mettre dans le dossier *userprog*. Nous pensons que cette fonction peut encore nous être utile dans la suite du projet, nous avons donc décidé de créer un fichier *utils* pour pouvoir la retrouver facilement.

Lors de la création de l'appel système *PutString*, nous nous sommes demandés quelle taille devait faire le buffer que nous utilisons pour copier la chaine de caractères. Nous avons choisi 8 pour pouvoir facilement se rendre compte s'il y a un problème lorsque l'on copie une chaine de caractères plus longue que le buffer. Si nous n'avions pas fixé de taille pour le buffer et fait en fonction de la taille de la chaine, nous aurions pu avoir plus de fragmentation de mémoire (même si nous ne nous en soucions pas encore) et cela aurait surtout rendu le code plus compliqué et la découverte d'erreurs et de dépassement de mémoire vraiment difficile.

2.4 Partie VII

Nous n'avons pas eu de problème particulier en implémentant la fonction *getChar*. Cependant, nous avons remarqué que si le caractère récupéré était *EOF* cela pouvait créer un problème d'affichage, car il correspond à l'entier -1 qui affiche *'ÿ'*. Nous avons donc décidé que si le caractère récupéré était un *EOF*, nous ne retournerions pas -1, mais 0 (qui correspond au caractère *NULL* et corrige l'affichage).

L'action nous ayant pris le plus de temps en réflexion dans cette partie est surement la fonction *copyStringToMachine*. En effet, nous avons cherché un maximum de cas spécifique dans lesquels notre fonction pourrait écrire à des endroits indésirés. Cependant, nous avons tout de même eu des problèmes après implémentation, car *Valgrind* nous indiquait des écritures illégales. Nous avons donc rapidement vu que *copyStringToMachine* n'était pas la seule fonction qui pouvait créer des problèmes sur l'écriture, et que la méthode *GetString* pouvait, elle aussi, en être responsable (une simple erreur de boucle qui écrivait un caractère plus loin que le malloc).

Afin d'implémenter les appels systèmes *GetInt* et *PutInt* nous avons dû faire le choix de donner une taille maximum aux nombres pour les représenter dans des strings. Nous avons choisi qu'un nombre pourrait faire au maximum 12 caractères (variable *MAX_INT_SIZE* dans *consoleDriver.h*). En effet, l'entier maximum étant composé de 10 chiffres, nous avons conservé deux caractères supplémentaires afin de pouvoir stocker le signe *'-'* si celui est négatif ainsi que le caractère *'\0'* pour clôturer correctement la chaine. Lors de la lecture d'un entier avec *GetInt*, si aucun entier n'est reconnu par *sscanf* nous avons choisi

de garder sa valeur dans le buffer (0) et donc d'affecter la valeur 0 à l'adresse donnée par l'utilisateur.

2.5 Partie VIII

L'implémentation a été compliquée à adapter au monde utilisateur, car les types de données de la *libC* n'y sont pas disponibles. Il a donc fallu adapter notre code et certains choix provoqueront surement des bugs ou des problèmes d'implémentation dans le futur. L'indisponibilité de la gestion dynamique de la mémoire avec *malloc* a par exemple compliqué l'implémentation d'un buffer. Il a également fallu adapter d'autres types tels que *size_t* (mais facilement remplaçable par un entier).

3 Limitations

3.1 Partie V

Au niveau des fuites mémoires, nous n'avons pas trouvé de mémoire perdue. En revanche, nous nous sommes rendu compte qu'il y avait toujours de la mémoire accessible après la fin de l'exécution du programme. Nous avons donc testé les mêmes commandes de détection de fuite mémoire sur notre code actuel et sur le code qui nous a été fourni au début du projet. Sur les tests que nous n'avions pas implémentés, il y avait tout de même de la mémoire encore accessible. Nous en avons donc déduit que c'était dû au thread principal, que nous ne pouvons donc pas détruire étant donné qu'il est la source de la machine.

Une autre limitation pourrait être la taille du buffer utilisé. Nous avons choisi 8 pour que les erreurs d'implémentation soient plus faciles à détecter, mais cela reste un nombre très petit. Peut-être que si nous sommes amenés à utiliser *PutString* avec des chaînes de caractères très longues, il sera judicieux de modifier cette constante.

3.2 Partie VI

Lorsque nous avons implémenté l'appel système *SC_EXIT*, nous avons choisi d'uniquement afficher la valeur de retour dans la console. Il est donc affiché après l'exécution du programme, mais pas réellement utilisé par notre programme pour faire une action spécifique si la fonction *main* retourne une valeur spécifique (un code erreur par exmple)

3.3 Partie VII

Dans cette partie il a fallu faire plusieurs choix d'implémentation :

— Lors de l'appel de *getString* l'utilisateur doit fournir un pointeur où stocker sa chaine de caractères ainsi que la taille de cette chaine (*void GetString(char *s, int n)*). Au moment de l'implémentation, nous avons décidé que l'utilisateur devait lui-même penser que le caractère *'\0'*

occuperait une place dans la chaîne finale et qu'il devrait donc fournir uniquement $N - 1$ caractères.
- Il sera également du ressort de l'utilisateur de donner un pointeur assez grand pour contenir N caractères. Si N est plus grand que l'espace alloué à s alors la fonction *copyStringToMachine* pourra écrire en-dehors de l'espace prévu.
- Le choix d'avoir implémenté *GetInt* de manière que l'entier 0 soit stocké si aucun entier n'est reconnu peut poser problème suivant ce que l'on souhaite en faire. Cependant, nous ne pouvons pas lever d'erreur, c'est donc une bonne option pour que le programme continue à s'exécuter (même s'il pourrait y avoir une division par 0).

3.4 Partie VIII

Nous avons dû faire le choix d'un buffer de taille fixe pour implémenter notre fonction *printf*. Cette implémentation n'est surement pas la plus robuste, car nous n'avons pas connaissance de ce qu'il se passerait si l'écriture dépassait la taille du buffer. Pour compenser ce point faible, nous avons mis un buffer de taille 200 afin qu'il n'y ait pas de problèmes dans la plupart des cas (un *printf* de taille >200 n'arrive pas dans notre cas).

4 Tests

Plusieurs des fonctions que nous avons testées le sont sur un grand nombre de valeurs. Nous avons donc créé des scripts bash pour exécuter avec des entrées différentes. C'est le cas pour les fonctions *GetChar*, *GetString* et *GetInt*. Pour les exécuter il faut se placer dans le dossier *code/* et exécuter l'une des commandes suivantes :
- *GetChar* : ./getCharTest.sh
- *GetString* : ./getStringTest.sh
- *GetInt* : ./getIntTest.sh

4.1 Partie V

Afin de tester notre appel système *PutString*, nous avons rajouté un programme utilisateur *putstring.c* dans le dossier *test*. Nous avons testé notre fonction avec différents cas qui pourraient souligner des problèmes d'implémentation. Pour cela, nous avons décidé d'appeler *PutString* sur des chaînes de différentes tailles : avec une taille inférieure, égale ou supérieure à celle de notre buffer utilisé dans l'implémentation de l'appel système. Nous avons aussi testé notre fonction avec des appels avec des chiffres, ce qui ne pose pas de problème. En revanche, l'utilisation d'accents reste problématique, car MIPS utilise l'encodage ASCII et non UTF-8. Nous n'avons pas cherché à corriger cela.

4.2 Partie VI

L'appel système *SC_Exit* est surement le plus simple à tester. Il suffit de regarder après l'exécution du programme la valeur affichée dans la console. Nous avons donc exécuté ce test avec plusieurs valeurs avant d'en laisser une définitive.

4.3 Partie VII

4.3.1 Syscall getChar

La fonction *getChar* dépend d'une entrée utilisateur, il était donc impossible de la tester avec uniquement un programme C. C'est pour cela que nous avons combiné notre test et la commande Linux *printf*. Celle-ci nous permet de passer n'importe quel caractère dans notre programme. Afin d'avoir un échantillon de test convaincant, nous avons cherché quels cas auraient pu mettre à mal notre implémentation. Nous avons donc pensé tout d'abord aux entrées de plus d'un caractère (puisque *getChar* doit lire uniquement un caractère), et notre implémentation semblait gérer ces cas. Nous avons ensuite pensé aux caractères spéciaux, par exemple les caractères tels que *'\n'*, *'é'* ou encore *'EOF'*. Le test affiche donc simplement le caractère écrit à l'aide de la fonction *putChar*

4.3.2 Syscall getString

Tout comme la fonction *getChar* la fonction *getString* dépend d'une entrée utilisateur. Nous avons utilisé la même stratégie de test afin de vérifier l'implémentation de notre fonction. Cependant, cette fonction possède plus de cas particuliers pouvant ne pas être gérés par notre implémentation. En effet, nos tests ont beaucoup porté sur la gestion de mémoire/copie de la chaîne de caractères entrée par l'utilisateur. Nous avons donc testé les limites évidentes (si on appelle avec des chaînes de tailles $N-1$, N ou $N+1$). Ce test nous a permis de détecter le bug vu plus tôt (cf 2.4).

4.3.3 Syscall getInt

Nous avons utilisé le même système de test que vu précédemment pour cet appel système. Il était nécessaire de tester plusieurs valeurs limites (*INT_MAX* et *INT_MIN*) pour savoir si notre programme pouvait reconnaitre n'importe quel entier dans cet intervalle. Nous avons également du tester des chaines particulières, par exemple un nombre suivi d'une chaîne de caractères (ex *'12473H'*)

4.4 Partie VIII

Afin d'avoir les tests les plus représentatifs possibles, nous avons essayé les formatages à notre disposition (*%d*, *%c*, ...). Les tests intègrent également des tests pour les fonctions nécessaires au fonctionnement de *printf* (*isLower*, *strlen*, ...)