NACHOS : Entrées - Sorties

Manon Philippot / Anthony Delasalle

Lundi 16 Octobre 2017

1 Bilan

1.1 Objectif

L'objectif de ce devoir encadré était de mettre en place sous NachOS un mécanisme d'entrée-sortie minimal permettant d'exécuter des petits programmes simples tels que putchar.c fourni dans le TD. Nous avons ainsi implémenté une console synchrone au dessus de la console asynchrone, puis l'ensemble des appels systèmes nécessaires, et enfin intégré une fonction printf à l'interface utilisateur.

1.2 Entrées-Sorties Asynchrone

Dans un premier temps nous nous sommes contentés d'appliquer des modifications mineures sur la console test *ConsoleTest* du mécanisme d'entrées-sorties asynchrones *Console* déjà proposé par NachOS afin de mieux l'appréhender. Cette console test, initialement configurée pour se comporter de la même façon que *echo*, est maintenant configurée pour renvoyer chaque caractère saisi dans le terminal entre chevrons (<>), et pour afficher "Au revoir" sur fin de fichier (EOF), en plus du caractère 'q'.

1.3 Entrées-Sorties Synchrone

Nous avons ensuite implémenté au-dessus de la couche asynchrone un mécanisme d'entrées-sorties synchrones devant encapsuler tout le mécanisme des sémaphores. Cette surcouche, appelée *SynchConsole*, est implémentée dans les fichiers *synchconsole.cc* et *synchconsole.h*. Le corps de ces fichiers étant déjà fourni, nous avons implémenté les fonctions suivantes, qui sont toutes fonctionnelles :

- **void SynchPutChar(int c)** : écrit le caractère *c* dans le terminal.
  Implémentation : on poste une requête d'écriture du caractère ASCII *c* grâce à *PutChar(char c)* puis on attend d'être averti de la terminaison de la requête grâce au sémaphore *WriteDone* qui exécute le traitant d'interruption *WriteDoneHandler*.
- **int SynchGetChar()** : retourne le caractère saisi dans le terminal.
  Implémentation : On attend d'être averti qu'un caractère est disponible grâce au sémaphore *readMutex* qui exécute le traitant d'interruption *ReadAvailHandler*, puis on réalise la lecture du caractère ASCII avec la fonction *Console::GetChar()*.

- **void SynchPutString(const char s[])** : écrit la chaîne de caractère s dans le terminal.
  Implémentation : on parcourt la chaîne de caractère *s* et pour chaque caractère on fait appel à la fonction *SynchPutChar*.

- **void SynchGetString(char *s, int n)** : récupère dans la chaîne de caractère *s* jusqu'à *n*-1 caractères saisis dans le terminal.
  Implémentation : on récupère un à un les caractères saisis dans le terminal à l'aide de SynchGetChar. Tant que *n*-1 caractères n'ont pas été lus, que la fin de fichier n'a pas été atteinte ou que le retour à la ligne n'a pas été rencontré, on copie le caractère en question dans la chaîne de caractère *s*. On force la copie du caractère '\0' en dernière position de la chaîne s pour garantir la sécurité du système.

- **int copyStringFromMachine (int from, char *to, unsigned size)** : copie une chaîne caractère par caractère d'au plus *size*-1, du monde utilisateur (MIPS) en partant de l'adresse *from* vers le monde noyau en partant de l'adresse *to*.
  Implémentation : tant que le nombre de caractère lus est inférieur à *size*, on lit un caractère de la source *from* grâce à *ReadMem*. Si ce caractère n'est pas égal au caractère de fin de chaîne '\0', on le copie à l'adresse *to* incrémentée du nombre de caractères déjà copiés. On passe à la lecture du caractère suivant. On force la copie du caractère '\0' en dernière position de la chaîne s pour garantir la sécurité du système.

- **void copyStringToMachine (int to, char *from, unsigned size)** : copie une chaîne caractère par caractère d'au plus *size*-1, du monde noyau en partant de l'adresse *from* vers le monde utilisateur (MIPS) en partant de l'adresse *to*.
  Implémentation : tant que le nombre de caractère lus est inférieur à *size*, on lit un caractère de la source *from*. Si ce caractère n'est pas égal au caractère de fin de chaîne '\0', on le copie à l'adresse *to* incrémentée du nombre de caractères déjà copiés grâce à la fonction *WriteMem*. On passe à la lecture du caractère suivant. On force la copie du caractère '\0' en dernière position de la chaîne s pour garantir la sécurité du système.

- **void SynchPutInt(int n)** : écrit l'entier *n* dans le terminal. L'entier doit être compris entre -2147483648 et 2147483647.
  Implémentation : convertit l'entier en chaîne de caractère grâce à la fonction *snprintf*, puis l'affiche sur le terminal grâce à la fonction *SynchPutString*.

- **void SynchGetInt(int *n)** : récupère à l'adresse de *n* l'entier signé saisi dans le terminal. L'entier doit être compris entre -2147483648 et 2147483647.
  Implémentation : on récupère sous forme de chaîne de caractère l'entier signé grâce à *SynchGetString*, puis on le convertit en entier grâce à la fonction *sscanf*.

Pour que la classe SynchConsole soit prise en compte à la compilation, nous avons du effectuer des modifications sur le fichier *code/Makefile.common* : il suffisait de rajouter *synchconsole.o* dans *USERPROG_O*. Une console de test *SynchConsoleTest* ayant les mêmes fonctionnalités que *ConsoleTest*, a été mise en oeuvre dans *progtest.cc*. Nous avons modifié le fichier *main.cc* afin que l'option *-sc* de nachos lance cette *SynchConsoleTest* et que cette option apparaisse dans l'aide (option *-h*). Tout fonctionne.

1.4 Appels Systèmes

Afin de pouvoir utiliser les fonctions d'entrées-sorties de SynchConsole dans l'espace utilisateur, nous avons implémenté des appels systèmes. Un appel système est une fonction fournie par le noyau d'un système d'exploitation et utilisée par les programmes s'exécutant dans l'espace utilisateur.

Pour chaque appel système, nous avons ajouté l'appel système et la déclaration de la fonction utilisateur Nachos correspondante dans le header */userprog/syscall.h*. Nous avons édité le fichier *test/start.S* pour y rajouter la définition en assembleur de l'appel système, sur le modèle des autres fonctions déjà implémentées. Ensuite dans *userprog/exception.cc*, nous avons mis en place le traitant d'interruption qui est appelé par l'appel système en question :

- **void SC_PutChar(char c) :** écrit le caractère *c* dans le terminal.
  IMPLÉMENTATION : le caractère *c* se trouvant dans le registre 4, on fait appel à *SynchPutChar* en passant en argument la valeur du registre 4 lue grâce à la fonction *ReadRegister*.

- **char SC_GetChar() :** retourne le caractère saisi dans le terminal.
  IMPLÉMENTATION : récupère le caractère saisi dans le terminal grâce à la fonction *SynchGetChar* et l'écrit dans le registre 2 correspondant au registre de retour grâce à la fonction *WriteRegister*.

- **void SC_PutString(char *s) :** écrit la chaîne de caractère dans le terminal.
  IMPLÉMENTATION : on récupère la chaîne de caractère *s* avec un *ReadRegister*. Tant qu'il y a des caractères à lire dans cette chaîne (on le sait avec la valeur de retour de la fonction *copyStringFromMachine*), on fait appel à *SynchPutString* pour les afficher.

- **void SC_GetString(char *s, int n) :** écrit à l'adresse *s* de la chaîne de caractère au plus *n* caractère saisi dans le terminal.
  IMPLÉMENTATION : on récupère l'adresse *s* et l'entier *n* grâce à des *ReadRegister*. On récupère la chaîne de caractère d'au plus n caractères saisie dans le terminal avec *SynchGetString*, puis on copie cette chaîne à l'adresse *s*.

- **void SC_PutInt(int n) :** écrit l'entier signé *n* dans le terminal.
  IMPLÉMENTATION : l'entier *n* se trouvant dans le registre 4, on fait appel à *SynchGetInt* en passant en argument la valeur du registre 4 lue grâce à la fonction *ReadRegister*.

- **void SC_GetInt(int *n) :** écrit à l'adresse *n* l'entier signé saisi dans le terminal.
  IMPLÉMENTATION : récupère l'adresse *n* se trouvant dans le registre 4 grâce à *ReadRegister*, puis l'entier saisi dans le terminal grâce à la fonction *SynchGetInt* et l'écrit à l'adresse *n* grâce à *WriteMem*.

Les traitants dont l'implémentation n'est pas atomique sont encadrés par un sémaphore *mutex* pour prendre en compte les appels concurrents de threads. Ce sémaphore est déclaré et initialisé dans *system.cc*.

Mais un appel système ne marche que si la console synchrone existe déjà lorsque la requête est émise. Nous l'avons donc déclarée dans les fichiers *threads/system.cc* et

threads/system.h, initializée dans le fichier *threads/main.cc* et nous avons assuré sa destruction pendant la destruction de la machine lors de l'appel à la fonction *Cleanup()*.Tout marche désormais, l'utilisateur peut faire appel aux appels systèmes.

1.5 Arrêt et valeur de retour

Les petits programmes *.c* que nous exécutons avec nachos marchent désormais. Pour ne pas avoir à utiliser l'appel Système Halt à la fin de tous nos programmes *.c*, nous avons implémenté le traitant d'interruption manquant pour l'appel système *Exit(int status)* avec un appel à *Halt*. Pour prendre en compte la valeur de retour *return n* de la fonction *main* dans le cas où elle est déclarée à valeur entière, nous avons modifié le fichier *test\start.S* afin de mettre la valeur de retour dans le registre 4 en fin d'exécution de programme (*move $4, $2*). Nous avons ensuite récupéré cette valeur de retour dans une variable *ret* puis fait appel à *Exit(ret)* dans *Cleanup()*. Nous pouvons ainsi récupérer la valeur de retour.

1.6 Fonction printf

Nous avons réussi à intégrer une fonction *printf* à l'espace utilisateur. Pour cela nous avons récupéré le fichier *vsprintf.c* des sources de Linux 2.2, implémenté les fonctions manquantes *isxdigit*, *isdigit*, *islower*, *toupper* et *strnlen* pour que la fonction *vsprintf* fonctionne sans les inclusions manquantes, puis implémenté un *printf* avec *vsprintf*. Après modification de *test/Makefile* pour inclure automatiquement vsprintf.o à la liaison de nos programmes MIPS, nous avons obtenu un *printf* fonctionnel.

2 Points délicats

L'aspect qui nous a demandé le plus de travail est sûrement l'implémentation du traitant d'interruption pour l'appel système *SC_PutString*, ainsi que les fonctions *copyStringFromMachine* et *copyStringToMachine*. Concernant *SC_Putstring*, la première version que nous avions implémentée ne nous permettait pas d'afficher sur le terminal un texte dont la longueur était supérieure à la taille du buffer prédéfinie. Nous avons résolu ce problème par l'intermédiaire d'une boucle while qui charge le buffer avec le texte à afficher puis l'affiche avec *PutString*, tant qu'il y en a. Concernant les deux fonctions de copie, nous nous sommes interrogés sur comment les implémenter tout en prenant en compte la copie forcée du '\0' en dernière position pour garantir la sécurité. Devions-nous copier *size* caractères de la source vers la destination, puis rajouter le caractère de sécurité à la fin, et donc écrire dans la destination *size*+1 caractères, ou bien copier *size*-1 caractères de la source vers la destination, puis rajouter le caractère de sécurité à la fin pour finalement écrire dans la destination *size* caractères. L'énoncé indiquait qu'au plus *size* caractères devaient être écrits, nous avons donc opté pour la deuxième solution. Il fallait également faire attention à bien utiliser *ReadMem/WriteMem* pour les accès à la mémoire du monde MIPS et à convertir les valeur récupérés de la mémoire source en int ou char selon le type de mémoire de destination avant la copie.

La prise en compte de la valeur de retour *return n* de la fonction *main* et la mise en place des sémaphores pour prendre en compte les appels concurrents de threads étaient

également des points délicats.

3 Limitations

Notre implémentation de la classe SynchConsole, des appels systèmes ainsi que des fonctions associées est fonctionnelle. Cependant on remarque que pour les appels système *SC_PutInt* et *SC_GetInt*, lorsque l'entier en question est trop grand, il y a débordement et ce n'est pas traîté : cela peut être source d'erreurs. Il serait préférable de gérer ce cas ou bien d'en avertir l'utilisateur de ces appels systèmes.

4 Tests

Nous avons implémenté 4 tests afin de tester notre implémentation :

- **testChar.c :** teste d'abord la récupération d'un seul caractère avec *GetChar*, puis l'écriture de ce même caractère avec *PutChar*. Teste ensuite, jusqu'a ce que l'utilisateur saisisse le caractère '5', la récupération d'un ou plusieurs caractères avec *GetChar* et l'écriture de ce(s) même(s) caractère(s) avec *PutChar*.

- **testString.c :** teste d'abord le comportement de la fonction *PutString* lorsque celle-ci doit écrire une chaîne de caractère dont la longueur est supérieure à celle du buffer. Teste ensuite la fonction *GetString* avec un paramètre de taille 5, et enfin, jusqu'à ce que l'utilisateur saisisse la chaîne de caractère 'quit', la récupération d'une chaîne de caractère avec *GetString* et l'écriture de cette même chaîne de caractère avec *PutString*.
  REMARQUE : L'utilisation des "ou" et "et" semble être inversé dans l'interface utilisateur. En effet "ou" a l'effet de "et" et inversement.

- **testInt.c :** teste de dépassement qui montre le comportement de PutInt lorsque l'entier n'est pas compris dans l'interval des int, teste ensuite, jusqu'à ce que l'utilisateur saisisse l'entier 0, la récupération d'un entier compris entre -2147483648 et 2147483647 avec *GetInt* et l'écriture de ce même entier avec *PutInt*.

- **testPrintf.c :** teste la fonction *printf* avec la majorité des formats : %i, %o, %x, %u, %c, %s...

L'intégralité des tests peut être lancé grâce au script *tests.sh* se trouvant dans le dossier *test*. Une fois placé dans le dossier *test* avec le terminal, cela revient à saisir la commande *./tests.sh*.