Systèmes d'exploitation : Devoir 1

GROUPE 3
Thierno Sambegou DIALLO
Nathan LESNE

October 2019

1 Bilan

Nachos offre une version primitive des appels système d'entrées-sorties.
Mais le souci est que ces entrées-sorties fonctionnent de manière asynchrone. Ainsi le but de ce devoir est de procurer un équivalent synchrone à la Console de base ainsi que d'implémenter des appels système d'IO l'utilisant.

Nous avons donc implémenté et testé les appels suivants :
GetChar, PutChar, GetString ,PutString, GetInt, PutInt, ainsi qu'un équivalent à printf pour les programmes utilisateurs.

Pour le faire, nous avons mis en place une console synchronisée qui encapsule la console de Nachos en utilisant des Sémaphores.
Celle-ci a été implémentée dans la classe SynchConsole, dans userprog.

La classe contient les méthodes suivantes:

- void SynchPutChar(int ch)
- int SynchGetChar()
- void SynchPutString(const char str[])
- void SynchGetString(char *s, int n)
- void SynchPutInt(int n)
- int SynchGetInt(int *n)

Toutes les méthodes ont été testées et fonctionnent.
Elles peuvent cependant comporter des limitations détaillées en section 3.

Nous avons également ajouté la gestion de l'arrêt des programmes utilisateurs, ainsi ils ne sont pas obligés d'appeler Halt() en fin de programme. La valeur de retour de ces programmes est également gérée et affichée à la fin de ceux-ci.

La signature de l'appel GetInt est différente de celle proposée par le sujet, en effet elle contient maintenant une valeur de retour.
L'appel renvoie vrai si l'utilisateur a entré un nombre (avec/sans espaces), et faux si il a rentré autre chose. Cela annule la conversion implicite des caractères ASCII en entier sans vérification.

Enfin, nous avons fait le choix de placer les fonctions de transfert de chaînes du noyau vers l'espace utilisateur et inversement (copyStringFromMachine et copyStringToMachine) dans system.
En effet, au début nous pensions les mettre dans machine, car elles manipulent directement la mémoire, hors nous avons vite vu que cela est interdit.
Sachant que ces fonctions vont certes être utilisées dans l'espace utilisateur mais n'en font pas vraiment partie car trop bas niveau, nous avons fait le choix de les mettre dans system pour les utiliser de facon globale.

2 Points délicats

L'implémentation des appels système a été un point compliqué a comprendre à cause des multiples étapes, en effet pour chaque appel système il faut :

- Éditer le fichier syscall.h pour définir la fonction de l'appel système, par exemple *void PutChar(char c)*, ainsi que le code d'appel (SC_PutChar)
- Ajouter le stub de l'appel système dans start.S
- Coder l'appel système dans le switch de code d'appel dans exception.cc (en utilisant les méthodes de SynchConsole et les fonctions de gestion des registres).

Au début cela était un peu déroutant, néanmoins nous étions à l'aise avec la manipulation et l'implémentation de ces appels à la fin.

Le plus dur à faire était sans doute printf, notamment au niveau du Makefile, qui donnait des erreurs de compilation (stdarg non trouvé) à cause d'une installation différente de celle du CREMI.

Cette erreur à été réglée en linkant le dossier include dans le Makefile (ou en compilant au CREMI via ssh) en l'ajoutant INCDIR.
Nous avons ajouté un header à vsprintf.c, pour sélectionner seulement les fonctions utiles à avoir dans les programmes de test, que nous avons ajouté à tout les .o (comme syscall.h).
Nous n'avons pas eu besoin de filter-out des PROGS vsprintf, car nous l'avons placé dans un sous dossier lib/ ce qui nous semblais plus propre mais en même temps ne le compile pas de base.
A la place nous avons crée une cible pour compiler vsprintf, sur le même modèle que les .o et l'avons ensuite ajouté à la cible %.coff.

Un autre problème délicat de printf était le fait que la chaîne était limitée et que un flag faisait planter le programme. Malheureusement, le bug du flag n'a pas été résolu. Le problème du tronquage de printf est détaillé dans les Limitations.

3 Limitations

Il existe plusieurs limitations dans notre implémentation, il est notamment indispensable de limiter les buffers alloués lors des appels systèmes GetString, GetInt, PutString, PutInt. Nous avons donc défini 2 constantes:

- MAX_STRING_SIZE, fixé actuellement a 128
- MAX_INT_LENGTH, fixé à 3 * sizeof(int)+2. (6 ou 10 pour l'entier, 1 pour le signe et 1 pour le caractère de fin de chaîne.

MAX_STRING_SIZE est indispensable pour garantir qu'il n'y ait pas de débordements en mémoire en utilisant les appels GetString et PutString.
Pour PutString, la chaine affichée est tronquée.
Pour GetString, le nombre de caractères lu ne peut dépasser la constante. Cela peut être résolu en bufferisant la lecture et lire au fur et à mesure, idée que nous n'avons pas implémentée.

MAX_INT_LENGTH une constante qui change en fonction de si l'architecture est 16bits ou 32/64bits qui sert a avoir la taille d'un entier signé dans une chaîne de caractères (avec le caractère de fin de chaîne).
Elle est utilisée pour allouer les buffers lors des appels SynchPutInt et SynchGetInt.
La limitation inévitable engendrée est que lors d'un appel a PutInt ou GetInt, si l'entier ne rentre pas dans cette taille la variable overflow et donne un résultat inattendu.

Une dernière limitation est printf, la limite de la taille de la chaîne finale est fixée a celle de PutString.
Une solution que nous n'avons pas implémentée car complexe est de ne pas passer par l'intermédiaire de vsprintf mais d'afficher les parties de la chaîne au fur et à mesure du décodage.
La limite devient alors la taille maximum qui peut être allouée dans le programme utilisateur pour la taille de la chaine format.

4 Tests

Pour tester notre implémentation des entrées/sorties en Nachos, nous avons crée pour chaque appel système un fichier portant son nom.
Voici un récapitulatif des différents tests:

- GetChar: le programme fait juste un appel a GetChar, nous avons testé son comportement pour des caractères standards, la console Nachos gérant déjà les cas possibles.
- PutChar: nous n'avons pas touché au programme de base, n'ayant jamais rencontrés de bug avec cet appel.
- GetString: Nous faisons un appel a GetString avec une taille plus grande que MAX_STRING_SIZE, cela affiche un avertissement, et change la taille maximum lue pour la constante.
  Puis nous faisons un PutString pour regarder le résultat.
  Nous avons testé avec une taille négative, 0 ou très grande, sans soucis.
- GetInt: Testé avec valeurs en dehors des limites d'un entier, il y a un overflow inévitable. Avec des caractères autre que des nombres ou espaces, l'appel système ne fait que renvoyer faux.
- PutInt: Testé avec des paramètres entiers dépassant les limites. Overflow inévitable comme GetInt.
- printf: Testé avec tout les flags, tous sauf %n (erreur qui cause un crash) passent sans problème, avec une chaîne trop longue la chaine est tronquée, ce qui est le comportement attendu. L'erreur du flag %n est peut être du a la vieille implémentation de vsprintf, (kernel linux 2.2.0) mais nous ne l'avons pas résolue.