

Projet NACHOS : Entrées/Sorties

Emmanoe Delar, Abdoul Diallo, Eudes Martinez

Mardi 16 Octobre 2018



Github: [dicowenza/NACHOS](https://github.com/dicowenza/NACHOS)

Table des matières

1	Bilan	2
2	Points délicats	2
2.1	Synchronicité	2
2.2	Appels systèmes	3
2.3	Chaînes de caractères	3
2.4	Arrêt (Halt & Exit)	4
3	Limitations	5
4	Tests	6
5	Annexes	7
5.1	Sémaphores	7
5.2	Exécution d'un programme utilisateur	7

1 Bilan

L'objectif de ce devoir consistait à mettre en place sous NACHOS quelques appels systèmes de base concernant la gestion des entrées/sorties.

Nous avons **travaillé sur la problématique de synchronicité**. À partir de la `Console` asynchrone présente initialement dans NACHOS, nous avons développé une couche d'abstraction qui gère la synchronicité (`SynchConsole`).

Nous avons **défini un ensemble d'appels systèmes** en spécifiant leurs comportements dans le traitant standard d'interruption (`ExceptionHandler`). Cette partie nous a également permis de mieux comprendre la frontière entre monde noyau et monde utilisateur (gestion des registres, accès mémoire, etc.).

Au travers de notre travail (observations & développement), nous nous sommes familiarisé avec **l'architecture de NACHOS** (gestion des entrées/sorties, des appels systèmes, exécution des programmes utilisateurs, etc.).

2 Points délicats

2.1 Synchronicité

Dans la partie II du sujet, nous avons manipulé la classe `Console` et effectué quelques tests qui nous ont permis de comprendre l'importance de contrôler les opérations de lecture et d'écriture.

En effet, sans indiquer précisément le moment où la lecture est possible, l'appel système `GetChar` — dans sa version asynchrone — va potentiellement lire dans des zones mémoires incorrectes du buffer, ce qui peut amener des incohérences voire la génération de signaux `"Segmentation Fault"`. Respectivement, sans indiquer le moment précis où l'écriture est terminée, l'appel système `PutChar` — dans sa version asynchrone — sera potentiellement stoppé avant l'écriture effective du caractère demandé.

Dans l'optique de résoudre cette problématique, nous utilisons deux sémaphores (`readAvail` / `writeDone`). Un premier nous permet de prévenir que la lecture est possible et le second nous permet de prévenir que l'écriture est terminée.

Couche d'abstraction (`SynchConsole`)

La partie III du projet consistait quant à elle à englober le travail effectué dans la partie II dans une nouvelle classe `SynchConsole` gérant en interne la synchronicité. Cette classe offre alors une couche d'abstraction et simplifie l'utilisation des méthodes `PutChar` & `GetChar`.

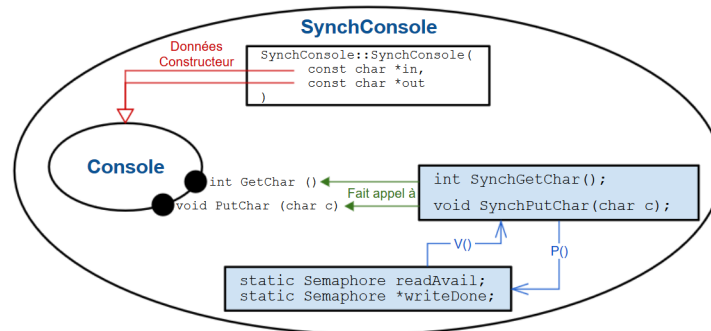


FIGURE 1 – Architecture de la classe SynchConsole.

A noter que nous avons choisi de ne pas faire de SynchConsole une classe fille de Console. Nous considérons un objet Console comme étant une propriété de cette nouvelle classe.

2.2 Appels systèmes

Dans la partie IV, nous devions mettre en oeuvre notre premier appel système. Pour ce faire, nous avons parcouru différentes étapes, en partant de la définition d'une constante et d'une signature permettant d'identifier l'appel système (syscall.h) jusqu'à la capture du signal d'interruption (dans le traitant standard d'interruption).

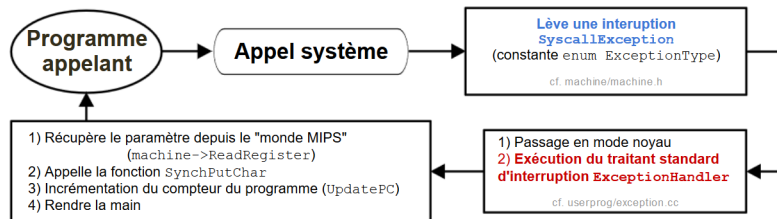


FIGURE 2 – Schéma de déroulement d'un appel système.

2.3 Chaînes de caractères

La partie V traitait la prise en charge des chaînes de caractères. Nous avons donc débuter cette partie en proposant une implémentation de la méthode SynchPutString qui fait appel au sein d'une boucle while à la méthode SynchPutChar jusqu'à la rencontre du caractère de fin de chaîne.

Nous avons ensuite défini la fonction copyStringFromMachine dans le fichier system.cc, car c'est un élément important et bas niveau du système.

Elle permet de copier une chaîne de caractère d'une adresse du monde utilisateur MIPS vers une adresse du monde noyau. Pour garantir la sécurité du système, on place en dernière position le caractère de fin de chaîne.

La fonction `copyStringFromMachine` utilise la fonction `ReadMem`. Il est important de noter la conversion `int` vers `char` qui correspond au fait que bien que stocker sur un entier (4 octets), les données utiles se trouvent sur un unique octet (un `char`). C'est la raison pour laquelle on ne peut pas simplement passer un pointeur pointant à l'intérieur du tampon `to`.

Il n'est pas raisonnable d'allouer un buffer de la même taille que la chaîne MIPS car nous ne pouvons pas connaître au préalable la taille qui sera demandée et cette dernière peut être importante. C'est pourquoi nous utilisons une constante `MAX_STRING_SIZE`. Évidemment, sans gestion cohérente de cette limite (cf. illustration), l'appel système `PutString` ne pourra gérer que `MAX_STRING_SIZE` caractères.

Pour assurer le bon fonctionnement de `PutString` et `GetString`, nous devons donc boucler sur la lecture / écriture en mémoire tout en garantissant le fait de ne pas réécrire sur une zone mémoire déjà utilisée :

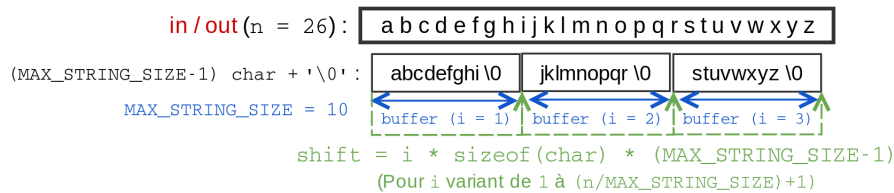


FIGURE 3 – Prise en charge d'un buffer de taille fixe.

2.4 Arrêt (Halt & Exit)

En retirant l'appel à la fonction `Halt` dans notre programme `putchar.c`, nous obtenons l'erreur suivante :

```
1 Unimplemented system call 1
2 Assertion failed: line 117, file "exception.cc"
3 Aborted
```

Cet assert — positionné dans le traitant standard d'interruption — nous signale qu'un appel système non implémenté a été déclenché (la constante représentative de l'appel étant 1). En allant chercher dans le fichier `syscall.h`, on observe que la constante de valeur 1 correspond à l'intitulé `SC_Exit`.

Notons que cet appel est visible dans le fichier `start.s` après l'exécution de la fonction `main`.

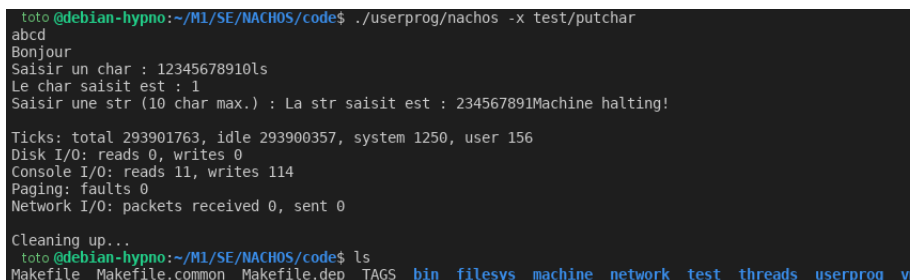
Afin de régler ce problème, il nous faut prendre en charge l'appel système `Exit` dans le traitant standard d'interruption. Nous récupérons tout d'abord la valeur `status` indiquée lors d'un appel système `Exit` en lisant le registre `r4`. Ensuite, nous déclenchons une interruption de type `Halt` afin de stopper `NACHOS`. Nous pouvons désormais utiliser ce nouvel appel système (donc supprimer les précédents appels à `Halt`) dans nos programmes utilisateur.

Pour prendre en compte la valeur de retour (`return n`) de la fonction `main`, nous avons dû modifier le fichier `start.s`. Le saut vers `Exit` n'ayant pas d'argument, le registre 4 est libre et on peut donc y stocker la valeur de retour de la fonction `main`. On stockera ensuite cette valeur dans le registre 2 après la fonction `main`.

3 Limitations

Nous avons identifié les limitations suivantes :

- Le nombre de caractères saisis sur l'entrée standard n'est pas contrôlé. Nous avons remarqué que lorsqu'un utilisateur saisit sur l'entrée standard un nombre de caractères plus important que la limite définie dans le programme utilisateur, des problèmes peuvent survenir (cf. illustration). Nous identifions également ce problème lorsqu'un appel à `GetString` est lancé à la suite d'un appel à `GetChar`. On saisit par exemple le caractère `'a'` puis on appuie sur Entrée. `GetChar` va prendre en compte le premier caractère, mais le caractère `'\n'` sera conservé dans le buffer du terminal et ensuite pris en compte par `GetString`.
- Les appels systèmes `PutInt` et `GetInt` dépendent des machines. En effet, nous supposons que la taille d'un entier est fixe : représenté sur 4 octets et donc définie par au maximum 12 char (10 char de chiffres + 1 char de signe + 1 char de fin de chaîne). Afin d'assurer une meilleure portabilité, il serait plus pertinent de calculer cette taille maximale (dans le constructeur de la `SynchConsole`) en nous basant sur l'appel à la fonction `sizeof(int)`.



```
toto@debian-hypno:~/M1/SE/NACHOS/code$ ./userprog/nachos -x test/putchar
abcd
Bonjour
Saisir un char : 123456789101s
Le char saisi est : 1
Saisir une str (10 char max.) : La str saisi est : 234567891Machine halting!

Ticks: total 293901763, idle 293900357, system 1250, user 156
Disk I/O: reads 0, writes 0
Console I/O: reads 11, writes 114
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
toto@debian-hypno:~/M1/SE/NACHOS/code$ ls
Makefile  Makefile.common  Makefile.dep  TAGS  bin  filesys  machine  network  test  threads  userprog  vm
```

FIGURE 4 – Illustration du débordement du buffer du terminal.

4 Tests

Nous avons testé en continu nos implémentations tout au long de notre développement.

Nous avons utilisé plusieurs fichiers de tests (`putchar.c`, `putstring.c`, `putint.c`) afin d'identifier les différents problèmes qui pouvaient survenir. Nos tests ont tous la même structure : nos programmes demandent à l'utilisateur de saisir un élément (`char`, `string`, `int`) depuis l'entrée standard, suite à quoi ce dernier est réécrit sur la sortie standard.

Nous avons donc assuré le bon fonctionnement des appels systèmes d'entrées/-sorties sur ces 3 types de données. Tant que l'utilisateur écrivant dans l'entrée standard respecte les limitations de taille sous-jacentes aux types de données :

- Pour les `char` : L'utilisateur doit saisir un unique caractère.
- Pour les `string` : L'utilisateur doit respecter la taille indiquée dans l'appel à la fonction `GetString`.
- Pour les `int` : L'utilisateur doit respecter les limites de ce qu'il est possible de stocker sur 4 octets (`[-2 147 483 648, 2 147 483 647]`).

... aucun problème ne survient.

Pour les `string`, on notera qu'aucune contrainte n'est engendrée par le fait que le buffer noyau soit de taille constante.

Profilage Afin de garantir l'absence de fuite mémoire dans notre code, nous avons utilisé l'outil de profilage Valgrind.

5 Annexes

Le rapport étant limité à 5 pages, nous plaçons dans cette section des idées et observations que nous avons eu lors de notre travail. Cette partie est facultative à la lecture et est présente afin de garder une trace écrite des différents points que nous avons traités mais qui ne font pas directement partie du projet.

5.1 Sémaphores

Les sémaphores sont des outils de haut niveau pour la synchronisation composés par :

- Une valeur entière k (initialement ≥ 0).
- Une liste de processus (initialement vide).
- Un couple d'opérations :
 - $P(s)$: Décrémenter k et si ($k < 0$), bloquer le processus dans la liste.
 - $V(s)$: Incrémenter k et si ($k \leq 0$), réveiller un processus de la liste.

Les cours magistraux nous ont permis de voir plus en profondeur la structure (**instruction `test_and_set`** du processeur) et les cas d'utilisations comme par exemple l'**exclusion mutuelle (MUTEX)**.

5.2 Exécution d'un programme utilisateur

Afin de comprendre plus en détails ce qu'il se passe lors de l'exécution d'un programme utilisateur, nous avons suivi les différentes étapes qui se produisent lors de l'exécution (depuis le répertoire `code`) de la commande :

```
> ./userprog/nachos -x test/putchar
```

1. Le fichier `start.s` fait appel à la fonction `main`.
2. La méthode `main` (du fichier `threads/main.cc`) est lancée. Cette fonction fait appel à la méthode `StartProcess` en donnant en argument le nom du fichier exécutable ("`test/putchar`").
3. La méthode `StartProcess` (du fichier `userprog/progtest.cc`) alloue un espace d'adressage virtuelle dans lequel il place le code de l'exécutable, puis fait appel à la méthode `Run` de la classe `Machine` (cf. fichier `machine/mipssim.cc`).
4. La méthode `Run` constitue le cœur de l'exécution d'un programme : elle exécute les instructions une à une et à chaque fin d'exécution d'une instruction lève une interruption de type `OneTick`.
5. Une fois le programme terminé, nous revenons dans le fichier `start.s` qui fait un `jump` vers `Exit`.