TP2 :
Appel Système : Entrées/Sorties Console

Polytech'ricm2

Janvier 2006

Note *Attention : ce sujet demande beaucoup de méthode de votre part. Réfléchissez avant de coder, sinon ça sera encore plus dur !*
D'autre part, toutes vos modifications doivent être encadrées avec

```
#ifdef CHANGED
  ...
#endif // CHANGED
```

pour pouvoir être réversibles. Par défaut, vous compilez avec `-D CHANGED`*. Les modifications* non signalées *sont rigoureusement interdites.*

Objectif *L'objectif de ce devoir est de mettre en place en Nachos quelques appels systèmes de base.*
L'énoncé est volontairement laissé flou sur plusieurs points. Certains choix de conception non triviaux *sont donc laissés à votre appréciation.*
L'objectif de ce devoir encadré est de mettre en place sous Nachos un mécanisme d'entrée-sortie minimal, permettant d'exécuter le petit programme `putchar.c` *suivant (quelle est la sortie raisonnablement attendue ?)*

```
#include "syscall.h"

void print(char c, int n)
{
  int i;
  for (i = 0; i < n; i++) {
    PutChar(c+i);
  }
  PutChar('\n');
}

int
main()
{
  print('a',4);
  Halt();
}
```

Il faut placer ce programme `putchar.c` *sous le répertoire* `test` *et adapter si nécessaire le* `test/Makefile`*.*

Partie I. Entrées-sorties Console

On trouve en Nachos :

Matériel une simulation fonctionnelle de 2 contrôleurs série d'entrée et sortie de caractères. On trouve dans `machine/console.h` et `machine/console.cc` l'implantation simulée des classes corrspondantes `ConsoleInput` et `ConsoleOutput`. On instancie un objet console d'entrée ou console de sortie à partir :

– soit d'un fichier via son nom

– soit de l'entrée standard (`std::cin` en C++ ou file descripteur 0 en C) ou de la sortie standard (`std::cout` en C++ ou file descripteur 1 en C). C'est l'option par défaut. Les entrées et sorties ont lieu sur le xterm à partir duquel nachos est lançé.

Dans tout les cas, on ne doit pas instancier plusieurs objets console sur le même fichier.

Les entrées-sorties fonctionnent de manière asynchrone, par interruption.

– Pour écrire un caractère, on émet une requête d'écriture de caractère grâce à `ConsoleOutput::PutChar(char ch)`. On retourne immédiatement. On sera averti de la terminaison de la requête par une interruption de fin de sortie provoquant l'exécution du traitant d'interruption(cf classe `CallBack`) associé initialement à la console de sortie (classe `ConsoleOutput`).

– Symétriquement, pour lire via `ConsoleOutput::GetChar(char ch)`, on doit être averti qu'il y a quelque chose à lire par l'exécution du handler (objet `CallBack`) associé à l'objet `ConsoleInput`.

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.
Expliquez pourquoi.
Pour faire effectivement des E/S console, il est donc nécessaire de garantir que les ordres d'E/S effectif sont émis au *bon moment* c'est à dire quand l'appareil est prêt à les faire. Notez aussi qu'il n'y a aucune raison de ne pas faire des choses utiles entre le moment où l'on "poste" la requête d'écriture et le moment où l'on est averti de sa terminaison ou pendant le temps d'attente de l'arrivée d'un caractère. On peut donc*recouvrir* les communications par des calculs !

E/S synchrone Nachos offre donc un ensemble d'opérateurs permettant à un thread noyau de faire des E/S console **synchrones** qui garantissent qu'on lit sur la console seulement quand un caractère est disponible et qu'on écrit seulement quand l'écriture est possible. Quand une opération n'est pas possible, le thread est bloqué et libère le processeur pour un autre thread. Les deux classes `SyncConsoleInput` et `SyncConsoleOutput` (fichiers `userprog/synchconsole.h` et `userprog/synchconsole.cc`)

Action I.1. Test *On peut tester le fonctionnement de la console* synchrone *en lançant nachos avec test de console(`Kernel::ConsoleTest` du fichier `kernel.cc`).*

```
nachos -C
```

ou

```
nachos -C 2>> <fichier_de_trace>
```

si l'on veut séparer les traces des E/S sur la console xterm

Action I.2. Inspection et trace *Regardez maintenant la mise en oeuvre de`SyncConsoleInput` et `SyncConsoleOutput`. On se place d'abord dans un cas simple où l'on se bloque sur l'attente d'arrivée d'un caractère ou de fin d'impression de caractère via des sémaphores. Pour attendre, on prend le sémaphore correspondant. Les handlers de notification les libèrent. En conséquence, si le caractère est déjà présent ou a été sortie , on peut lire ou écrire immédiatement ! Relancer le test en traçant les commutations de threads.*

```
nachos -C -d t
```

Commentez la trace obtenu pour expliquer brièvement ce qui se passe.

Action I.3. Multiprogrammation *On veut vérifier qu'il y bien possible de recupérer du temps d'inactivité d'un thread pour faire du calcul. Modifiez `Kernel::ConsoleTest` pour quelques threads exécutant `SimpleThread`. Vérifier que ces threads s'exécutent pendant les temps d'attente sur la console.*

Partie II. Appel système `PutChar`

L'objectif est maintenant de mettre en place un appel système `PutChar(char c)` qui prend en argument un caractère `c` en mode utilisateur puis lève une interruption `SyscallException`. Celle-ci provoque le passage en mode noyau et l'exécution du handler standard. Celui-ci doit appeler la fonction noyau `PutChar`, puis rendre la main au programme appelant, en ayant soin d'incrémenter le compteur de programme ! Vous comprenez maintenant pourquoi un appel système est si coûteux. C'est pourquoi les entrées-sorties Unix sont *bufferisées* : `fprintf(3)` est bien moins coûteux que `write(2)` sur chaque caractère, puisqu'il y a un appel système à chaque *ligne*, et non à chaque *caractère* !

Après cette mise en place, le programme *utilisateur* Nachos `putchar` ci-dessus devrait fonctionner !

Action II.1. Déclaration PutChar coté utilisateur *La première tâche est de mettre en place l'appel système.*

Éditez le fichier `userprog/syscall.h` pour y rajouter un appel système `#define SC_PutChar ...` et la fonction `void PutChar(char c)` correspondante. Il s'agit ici de la fonction utilisateur *Nachos : en terme Unix, `putchar(3)`. (Faire `man 3 putchar` pour vérifier !)*

Action II.2. Définition PutChar coté utilisateur *Il faut maintenant définir le code de la fonction utilisateur`PutChar(char c)`. Comme celle-ci doit provoquer un déroutement (*trap*), ce code doit être écrit en assembleur.*

Éditez le fichier `test/start.s` pour y rajouter la définition en assembleur de `PutChar`. Vous pouvez copier celle de `Halt`. Notez que l'on place le numéro de l'appel système dans le registre `r2` avant d'appeler l'instruction "magique" `syscall`. Le compilateur place le premier argument `char c` dans registre `r4`. Ce registre est un registre entier 32 bits : le caractère est donc implicitement converti : `r4 = (int)c`. Il faudra penser à faire la conversion inverse à son extraction !

Action II.3. Définition PutChar coté système *Il faut maintenant déclarer et définir la fonction noyau `SysPutChar(char)` dans le fichier `userprog/ksyscall.h`.*

Action II.4. Récupération exception*Il faut maintenant mettre en place le handler qui est activé par l'interruption `syscall`. Pour cela modifier le fichier `userprog/exception.cc`. Transformez la fonction*

`ExceptionHandler (ExceptionType which)`

pour traiter le cas du `SC_PutChar`.

```
 #ifdef CHANGED
case SC_PutChar: {
        .......
        SysPutChar (.....);
....
      }

#end // CHANGED
```

Inspirez vous des appels systèmes existant pour récupérer l'argument de `SysPutChar`. Attention, bien penser à incrémenter le compteur d'instruction : par défaut, on réactive l'instruction courante au retour d'une interruption (pensez aux défauts de page !).

Action II.5. *Et maintenant tester ! Il faut recompiler nachos et les tests.*

```
cd $NOS/build.linux
make
cd $NOS/build.linux
make
```

puis quand tout est OK : `nachos -x putchar`.

Partie III. Des caractères aux chaînes

Pour le moment, nous ne pouvons écrire qu'un seul caractère à la fois. Écrire une chaîne se résume à faire une suite d'écritures de caractères, bien sûr ! On pourrait faire un procédure utilisateur `PutString(char *)` qui appellerait itératvement `PutChar`. Le surcoût système serait payé pour chaque caractère. C'est pourquoi, vous allez rajoutez un nouvel appel système `PutString(char *)`.

Action III.1. PutString coté utilisateur *Procéder de façon identique à `PutChar`.*

Action III.2. PutString coté noyau *Procéder de façon identique à `PutChar` en rajoutant `SyncPutString(char*)` à `userprog/ksyscall.h`.*

Action III.3. Récupération exception *Il faut maintenant mettre en place le handler qui est activé par l'interruption `syscall`. Pour cela modifier le fichier `userprog/exception.cc`. Transformez la fonction*

`ExceptionHandler (ExceptionType which)`

pour traiter le cas du `SC_PutString`.

```
 #ifdef CHANGED
case SC_PutString: {
        .......
        SysPutString (.....);
....
       }

#end // CHANGED
```

Ici l'argument en registre n'est plus une valeur mais une adresse (de chaîne). Le seul problème est que l'on ne dispose que d'un pointeur en espace utilisateur(MIPS) vers la chaîne c'est à dire en mémoire virtuelle, et non pas d'un pointeur en espace système (linux) c'est à dire en mémoire physique ... Une solution est de copier la chaîne de l'espace utilisateur à l'espace système.

On a besoin d'une procédure auxiliaire

`void copyStringFromMachine(int from, char *to, unsigned size)`

qui copie une chaîne du monde utilisateur vers le monde système. Au plus `size` caractères sont copiés. Un `'\0'` est forcé à la fin de la copie en dernière position pour indiquer la fin de chaîne(convention C) et garantir ainsi la sécurité du système. Utilisez les opérateurs matériels *(répertoire machine) d'accès à la mémoire `Machine::ReadMem()`, `Machine::WriteMem()` et les fonctions de conversion `WordToMachine()`, `WordToHost()` pour écrire cette procédure. On pourra utiliser un buffer local au traitement de l'exception de taille `MAX_STRING_SIZE`,* en déclarant cette *constante dans le fichier `syscall.h`. Veillez à libérer le buffer une fois l'appel système terminé !*

Action III.4. *Montrez sur quelques exemples le comportement de votre implémentation, notamment en cas de chaîne trop longue.*

Partie IV. Mais comment s'arrêter ?

Action IV.1. *Que se passe-t-il si vous enlevez l'appel à `Halt()` à la fin de la fonction `main` de `putchar.c` ? Décryptez le message d'erreur et expliquez. Comment faire pour ne pas appeler la fonction `Halt()` explicitement dans vos programmes ? Comment faire pour prendre en compte la valeur de retour `return n` de la fonction `main` si celle-ci est déclarée à valeur entière ?*

Partie V. Fonctions de lecture

Action V.1. *Rajoutez l'appel système* `GetChar`*. Le registre utilisé pour le retour d'une valeur à la fin d'une fonction est le registre* `2` *: c'est là qu'il faut placer la valeur lue à la console. Attention, un registre est un entier. Pensez aux conversions éventuelles. Que faites-vous en cas de fin de fichier ?*

Action V.2. *Faites de même pour l'appel système* `void GetString(char *s, int n)` *sur le modèle de* `fgets` *(lisez bien le manuel pur la gestion des caractères de fin de ligne et des débordements !). Attention : 1) Vous devez absolument garantir qu'il n'y a pas de débordement au niveau du noyau. 2) Vous devez désallouer toutes les structures temporaires allouées pour éviter les fuites mémoire. 3) Vous devez prendre en compte les appels concurrents : que se passe-t-il si plusieurs threads appellent en même temps cette fonction ?*

Action V.3. *Mettez en place un appel système* `void PutInt(int n)` *qui écrit un entier signé en utilisant la fonction* `snprintf` *pour en obtenir l'écriture externe décimale. Idem dans l'autre sens avec* `void GetInt(int *n)` *et la fonction* `sscanf`*.*