L3 – Cours de Système – Devoir dirigé 1
Nachos : Entrées-sorties synchrones

Emmanuel Agullo, Eddy Caron, Nicolas Veyrat Charvillon

Vendredi 17 février 2006

Note *Attention : ce sujet demande beaucoup de méthode de votre part. Réfléchissez avant de coder, sinon vous êtes perdus !*

D'autre part, toutes vos modifications doivent être encadrées avec

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

pour pouvoir être réversibles. Par défaut, vous compilez avec `-DCHANGED`*. Les modifications sauvages sont rigoureusement interdites. Pour votre bien !*

Par ailleurs, la compilation par `gcc -Wall` *devra continuer à se faire sans aucun avertissement (sauf ceux déjà présents dans la version originale de Nachos...).*

L'objectif de ce devoir est de mettre en place en Nachos quelques appels systèmes de base.

Le devoir est à faire en binôme. Le temps estimé est de 10h de travail par personne. Il est à rendre pour le

samedi 11 mars 2006.

Vous aurez à présenter votre démonstration au TD suivant (15 mn par binôme).

Il vous est demandé de rendre un rapport mis en forme, regroupant les pièces suivantes, soit en Postscript par mail soit imprimé sur papier.

- L'ensemble des lignes de code significatives, indentées et commentées.
- Une description de la stratégie d'implantation utilisée, et une discussion des choix que vous avez faits. La structure et les différentes questions de l'énoncé sont là pour vous guider, et ne devront pas forcément être reproduits dans votre rapport.
- Une série d'exemples de test représentatif, présentant les qualités de votre implantation et ses limites. Chaque test doit être accompagné d'un court commentaire (5–10 lignes) expliquant son intérêt.

Partie I. Quel est le but ?

L'objectif de ce devoir encadré est de mettre en place sous Nachos un mécanisme d'entrées-sorties 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 donc placer ce programme `test/putchar.c` sous le répertoire `test` et adapter si nécessaire le `test/Makefile`.

Essayer de faire `make putchar` sous `test`... Comme vous le voyez, il y a encore du travail... Allons-y !

Partie II. Entrées-sorties asynchrones

Nachos offre une version primitive d'entrées-sorties par la classe `Console` qui se trouve définie sous `machine/console.cc`. Lire attentivement les commentaires. Les entrées-sorties fonctionnent de manière asynchrone, par interruption.

- Pour écrire un caractère, on "poste" une requête d'écriture grâce à `Console::PutChar(char ch)`, puis on attend d'être averti de la terminaison de la requête par l'exécution du handler `writeDone`.
- Pour lire, on attend d'être averti qu'il y a quelque chose à lire par l'exécution du handler `readAvail`, puis on réalise la lecture effectivement par la fonction `Console::GetChar()`.

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.

Notez que les handlers sont des fonctions C, pas C++, car elles sont partagées par la console et les classes qui l'utilisent.

Notez aussi qu'il n'y a aucune raison de ne pas faire des choses utiles entre le moment où on "poste" la requête et le moment où on est averti de sa terminaison. On peut tout à fait *recouvrir* les communications par des calculs ! Pour plus de détails, voir le cours de Parallélisme.

Regardez maintenant la mise en oeuvre sous `userprog/progtest.cc`. On se place d'abord dans un cas simple où on se bloque sur l'attente de terminaison grâce à des sémaphores.

```
static Console *console;
static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvail(int arg) { readAvail->V(); }
static void WriteDone(int arg) { writeDone->V(); }
```

Pour attendre, on prend le sémaphore. Les handlers de notification les libèrent. En conséquence, si le caractère est déjà présent lors d'une demande de lecture, on est immédiatement servi !

```
readAvail->P();         // wait for character to arrive
ch = console->GetChar();
```

Action II.1. *Examinez le programme* `userprog/progtest.cc`*. Lancer* `./nachos -c` *qui exécute la procédure* `consoleTest` *(voir* `threads/main.cc`*). Bien comprendre ce qui se passe.*

Action II.2. *Modifiez* `userprog/progtest.cc` *pour prendre en compte la terminaison de l'entrée correctement : fin de fichier ou, sur un tty,* `^D` *en début de ligne.*

Action II.3. *Modifiez* `userprog/progtest.cc` *pour faire écrire* `<c>` *au lieu de* `c` *dans le corps de la boucle.*

Action II.4. *Essayez de faire cela avec un fichier d'entrée et un de sortie. Par exemple,* `nachos -c in out` *(voir* `threads/main.cc`*).*

Partie III. Entrées-sorties synchrones

L'objectif est d'implanter au-dessus de la couche `Console` une couche d'entrées-sorties *synchrones* `SynchConsole`. L'idée est qu'une *console synchrone* doit encapsuler tout le mécanisme des sémaphores pour ne fournir que deux fonctions. Ceci est implanté juste à côté de la classe `Console`.

Action III.1. *Créer à partir du fichier* `machine/console.h` *le fichier* `userprog/synchconsole.h` *comme suit. Remarquer que l'inclusion de* `"console.h"` *fonctionne correctement grâce au chemin de recherche spécifié dans l'appel au compilateur.*

```
#ifdef CHANGED

#ifndef SYNCHCONSOLE_H
#define SYNCHCONSOLE_H

#include "copyright.h"
#include "utility.h"
#include "console.h"

class SynchConsole {
  public:
    SynchConsole(char *readFile, char *writeFile);
                                // initialize the hardware console device
    ~SynchConsole();                    // clean up console emulation

    void SynchPutChar(const char ch);   // Unix putchar(3S)
    char SynchGetChar();                // Unix getchar(3S)

    void SynchPutString(const char *s); // Unix puts(3S)
    void SynchGetString(char *s, int n);       // Unix fgets(3S)
  private:
    Console *console;
};

#endif // SYNCHCONSOLE_H
```

```
#endif // CHANGED
```

Notez que les sémaphores doivent être partagés entre les objets de classe `SynchConsole` et ceux de classe `Console`. Ils doivent donc être des fonctions C et non C++, à moins d'utiliser des fonctionnalités évoluées de C++ (`SynchConsole` devrait en fait être une classe fille de `Console`). Le fichier `userprog/synchconsole.cc` doit donc avoir la structure suivante :

```
#ifdef CHANGED

#include "copyright.h"
#include "system.h"
#include "synchconsole.h"
#include "synch.h"

static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvailHandler(int dummy) { readAvail->V(); }
static void WriteDoneHandler(int dummy) { writeDone->V(); }

SynchConsole::SynchConsole(char *readFile, char *writeFile)
{
  readAvail = new Semaphore("read avail", 0);
  writeDone = new Semaphore("write done", 0);
  console = ...
}

SynchConsole::~SynchConsole()
{
  delete console;
  delete writeDone;
  delete readAvail;
}

void SynchConsole::SynchPutChar(const char ch)
{
  // ...
}

char SynchConsole::SynchGetChar()
{
  // ...
}

void SynchConsole::SynchPutString(const char *s)
{
  // ...
}
```

```
void SynchConsole::SynchGetString(char *s, int n)
{
  // ...
}

#endif // CHANGED
```

Action III.2. *Complétez `synchconsole.cc` en ce qui concerne les opérations sur les caractères.*

Normalement, les fichiers `userprog/Makefile` et `Makefile.common` devraient automatiquement prendre en compte tous les fichiers que vous ajouterez dans le répertoire `userprog`. Par contre, `make depend` devrait être nécessaire.

Notez que lorsque vous compilez dans `userprog`, le `Makefile` cherche les fichiers de headers dans `userprog` et dans `threads`. Ainsi, l'inclusion de ces fichiers est facile. Mais l'inclusion de headers de `userprog` dans un autre répertoire devra être protégée par `#ifdef USER_PROGRAM`.

Action III.3. *Modifiez `threads/main.cc` pour ajouter une option `-sc` de test de la console synchrone qui lance la fonction `SynchConsoleTest`.*

Action III.4. *Ajouter à la fin de `progtest.cc` la définition de cette fonction. Par exemple :*

```
#ifdef CHANGED

void
SynchConsoleTest (char *in, char *out)
{
  char ch;

  SynchConsole *synchconsole = new SynchConsole(in, out);

  while ((ch = synchconsole->SynchGetChar()) != EOF)
      synchconsole->SynchPutChar(ch);
  fprintf(stderr, "Linux: EOF detected in SynchConsole!\n");

  delete synchconsole;
}

#endif //CHANGED
```

Notez que le `fprintf` est effectué par Linux, pas par Nachos !

Action III.5. *Faites mijoter à feu doux et lancez une fois que vous avez bien compris ce qui devrait se passer. Observez ce qui se passe... et itérez jusqu'au succès !*

Action III.6. *Agrémenter la fonction `SynchConsoleTest` comme à la partie précédente.*

Partie IV. 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 `SynchPutChar`, 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 moins bien coûteux que `write(2)` sur chaque caractère, puisqu'il y a un appel système à chaque *ligne*, et non à chaque *caractère* !

De cette manière, le programme *utilisateur* Nachos `putchar` ci-dessus devrait fonctionner !

La première tâche est de mettre en place l'appel système.

Action IV.1. *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 putchar` pour vérifier !).*

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

Action IV.2. *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 !*

Il faut maintenant mettre en place le handler qui est activé par l'interruption `syscall`.

Action IV.3. *Éditez le fichier `userprog/exception.cc`. Transformez la fonction*

`ExceptionHandler (ExceptionType which)`

en un `switch` C/C++, car il y aura de nombreuses exceptions possibles, bien sûr ! 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 !).

```
void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

#ifndef CHANGED // Noter le if*n*def

    if ((which == SyscallException) && (type == SC_Halt)) {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }

    UpdatePC();

#else // CHANGED

    if (which == SyscallException) {
      switch (type) {

      case SC_Halt: {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
```

```
          break;
        }

        case SC_PutChar: {
          ...
        }

        default: {
          printf("Unexpected user mode exception %d %d\n", which, type);
          ASSERT(FALSE);
        }

        }

        UpdatePC();

      }

#endif // CHANGED

}
```

Bien... Mais tout ceci ne marche que si la console synchrone existe déjà lorsque la requête est émise. Il faut donc la créer à l'initialisation du système.

Action IV.4. *Éditez le fichier `threads/system.cc`. Ajoutez une déclaration globale.*

```
#ifdef CHANGED
#ifdef USER_PROGRAM /* nécessaire car synchconsole ne peut pas
                       forcément être inclu si on compile ailleurs. */
#include "synchconsole.h"
SynchConsole *synchconsole;
#endif
#endif
```

Cependant, il ne faut pas créer de conflit entre cette synchconsole et la console créée lorsqu'on lance nachos avec l'option `-c` ou `-sc`. Il faudra donc ici ne la créer que lorsque l'option `-x` est passée. Il faudra bien entendu veiller à la détruire proprement lors de l'extinction de la machine.

Notez le `#define USER_PROGRAM` : cette modification n'est faite que lorsque l'on souhaite exécuter un programme utilisateur, c'est-à-dire que l'on compile depuis `userprog`.

Les différentes instances de la classe `Console` ne partagent que les entrées et sorties standard. Par contre, la classe `SynchConsole` utilise des variables globales qui seront donc partagées entre ses différentes instances. Vous trouvez cela mal ? Qu'à cela ne tienne...

Note *Idée bonus : Intégrer toutes ces variables dans la classe `SynchConsole` en conservant les handlers en C. On veillera à ne pas détruire le prototype du constructeur de la classe Console.*

Bon, tout (ou presque !) est en place maintenant. Allez sous `test`, faites `make`, et lancez `putchar`... Que se passe-t-il ? Ne désespérez pas : ça va bien finir par marcher !

Partie V. 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 d'un seul caractère, bien sûr ! Le seul problème est que l'on ne dispose que d'un pointeur MIPS vers la chaîne, et non pas d'un pointeur Linux...

Action V.1. *Écrivez une procédure*

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

qui copie une chaîne du monde MIPS vers le monde Linux. Au plus `size` *caractères sont copiés. Un* `'\0'` *est forcé à la fin de la copie en dernière position pour garantir la sécurité du système. On utilisera la fonction* `machine->ReadMem` *définie dans* `machine/machine.h` *pour accéder à la mémoire MIPS. On veuillera à éviter les débordement dus à l'utilisation d'un pointeur de type entier dans son prototype.*

Action V.2. *Complétez l'appel système* `PutString`*. On pourra éventuellement utiliser un buffer local de taille* `MAX_STRING_SIZE`*, en déclarant cette constante dans le fichier* `threads/system.h`*. Soyez vigilant à libérer le buffer une fois l'appel système terminé !*

Action V.3. *Montrer sur quelques exemples le comportement de votre implémentation, notamment en cas de chaîne trop longue.*

Partie VI. Mais comment s'arrêter ?

Action VI.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 ?*

Plus généralement, il faut mettre en place toute la mécanique nécessaire pour récupérer proprement les ressources utilisées par un programme et remettre le système d'exploitation en position de charger un autre programme. Mais ceci est une autre histoire...

Partie VII. Fonctions de lecture

Action VII.1. *Complétez 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 VII.2. *Faites de même pour l'appel système* `void GetString(char *s, int n)` *sur le modèle de* `fgets` *(lisez bien le manuel pour 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.*

Action VII.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* `int GetInt()` *et la fonction* `sscanf`*. Comment gérer proprement la fin de la ligne entrée ? Vaut-il mieux (au niveau sémantique ? en terme de facilité d'implémentation ?) tout lire et ne garder que le nombre au début, ou bien ne lire uniquement que le nombre et laisser les autres caractères en attente ?*

Partie VIII. Détection de fin de fichier

Action VIII.1. *Dans toutes les fonctions ci-dessus, la lecture du caratère* `'\127'` *(y trémas) est confondue avec la détection d'une fin de fichier. Une solution est d'ajouter une fonction du genre de* `feof(3)` *pour détecter spécifiquement la fin de fichier. Ajoutez cette fonction (Vous aurez éventuellement besoin de modifier la classe* `Console`*).*

Action VIII.2. *Une autre manière est que* `SynchGetChar` *renvoie un* `int` *et non un* `char`*, comme* `getchar(3)`*. Implantez cette solution.*