NachOS : Entrées / Sorties console
Année 2013-2014

par Jerôme Barbier, Augustin Husson et David Levayer

7 avril 2014

Rapport généré avec LaTeX

Table des matières

1 But

Dans cette partie, le sujet ne demande que l'ajout d'un fichier putchar.c dans le dossier Test. Le contenu de ce fichier est le suivant (il servira par la suite pour tester les entrées/sorties) :

```
#include "syscall.h"

int
main ()
{
    PutChar ('c');

    /* not reached */
    return 0;
}
```

2 consoles asynchrones

On exécute le programme ConsoleTest via la commande *./nachos-userprog -c* (dans le répertoire build). La console se lance et se contente d'imprimer sur l'écran le caractère que l'on a saisi. L'appui sur le caractère *'q'* ferme la console. Les modifications apportées à ce programme sont les suivantes :

- modification du mode de terminaison de la console (fin de fichier ou appui sur *Ctrl+D*)
- ajout de chevrons ('<' et '>') autour des caractères saisis
- possibilité d'utiliser des fichiers en entrée et en sortie

On modifie le fichier userprog/progtest.cc(et notamment la fonction ConsoleTest) :

```
void
ConsoleTest (char *in, char *out)
{
    char ch;
    console = new Console (in, out, ReadAvail, WriteDone, 0);
    readAvail = new Semaphore ("read avail", 0);
    writeDone = new Semaphore ("write done", 0);

    for (;;)
      {

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

      if(ch == EOF){
        return;
      }
      if(ch != '\n')
      {
        console->PutChar ('<');
        writeDone->P (); // wait for write to finish
      }

          console->PutChar (ch); // echo it!
      writeDone->P (); // wait for write to finish

      if(ch != '\n')
      {
        console->PutChar ('>');
           writeDone->P (); // wait for write to finish }}}
```

Petite explication sur les ajouts effectués

- on boucle pour lire les caractères
- on quitte la boucle lors de la réception du caractère EOF.
- lorsque l'on écrit un caractère, on écrit également les chevrons (avant et après)
- on utilise deux sémaphores pour synchroniser la lecture et l'écriture (afin d'attendre l'arrivée d'un caractère à lire et d'attendre la fin de l'écriture courante).

La console se comporte désormais de la façon décrite par l'énoncé.

3 Entrées-sorties synchrones

Dans cette partie on met en place une surcouche qui encapsulera la console et le mécanisme des sémaphores. L'idée étant de rendre transparent l'utilisation des sémaphores.

Pour cela, on créé la classe *SynchConsole* à l'aide des fichiers *synchconsole.cc* et *synchconsole.h*. Pour le moment on complète ces fichiers avec les éléments suivants :

- On a pour seul attribut : la console qu'on mettra en privé
- Le constructeur *SynchConsole(char *readFile, char *writeFile)* qui prend comme paramètre un nom de fichier d'entrée et un nom de fichier de sortie. Il permet d'initialiser notre console et deux sémaphores qu'on a préalablement déclaré *static*
- Le desctructeur *SynchConsole()* qui pour l'instant supprime simplement la console et les deux sémaphores.
- La procédure *SynchPutChar(const char ch)* qui permet d'écrire le caractère passé en paramètre dans la console
- La fonction *SynchGetChar()* qui permet de lire un caractère en entrée de la console.

Code du fichier synchconsole.cc

```
 #ifdef CHANGED
#include "copyright.h"
#include "console.h"
#include "system.h"
#include "synch.h"
#include "synchconsole.h"

#define NBREMAXCARACTENTIER 12 //entier sign --> -2147483648  2147483647 soit 11 caractres max
    avec le "-"

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

SynchConsole::SynchConsole(char *readFile, char *writeFile)
{
        readAvail = new Semaphore("read avail", 0);
        writeDone = new Semaphore("write done", 0);
        console = new Console (readFile, writeFile, ReadAvail, WriteDone, 0);
}
SynchConsole::~SynchConsole()
{
        delete console;
        delete writeDone;
        delete readAvail;
}
```

```
void SynchConsole::SynchPutChar(const char ch)
{
        console->PutChar(ch);
        writeDone->P();
}
char SynchConsole::SynchGetChar()
{
        readAvail->P();
        return console->GetChar();
}
#endif // CHANGED
```

On modifie ensuite le fichier *thread/main.cc* afin de pouvoir utiliser explicitement la nouvelle console que l'on vient d'implémenter en tapant la ligne de commande *./nachos-userprog -sc*

Modification de main.cc

```
 #ifdef CHANGED
          else if (!strcmp (*argv, "-sc"))
            {   // test the synchronous console
                if (argc == 1)
                    SynchConsoleTest (NULL, NULL);
                else
                  {
                      ASSERT (argc > 2);
                      SynchConsoleTest (*(argv + 1), *(argv + 2));
                      argCount = 3;
                  }
                interrupt->Halt ();
            }
#endif // CHANGED
```

La procédure *SynchConsoleTest* est une procédure qui est implémentée dans le fichier *progtest.cc* qui permet d'utiliser explicitement la nouvelle console.

Procédure SynchConsoleTest

```
 void
SynchConsoleTest (char *in, char *out)
{
    char ch = 'c';
    synchconsole = new SynchConsole(in, out);
    while ((ch = synchconsole->SynchGetChar()) != EOF)
    synchconsole->SynchPutChar(ch);
    fprintf(stderr, "Solaris: EOF detected in SynchConsole!\n");
    delete synchconsole;
}
```

C'est une procédure assez classique pour tester la synchconsole. On créé la synchconsole grâce au constructeur. On prend une variable *ch* permettant de récupérer les caractères tapés sur la console et également de tester si le caractère correspond au caractère EOF. On n'ajoute pas les chevrons car cela ne rajoute pas grand chose au programme de test.

Une fois que le programme est terminé, on supprime la synchconsole.

4 Appel système PutChar

On va mettre en place pour la première fois un appel système. Il permettra de passer en mode noyau pour pouvoir afficher un caractère en mode user comme le ferait la fonction *printf*.

Pour cela il y a des 4 étapes à respecter :

1. Tout d'abord on édite le fichier *syscall.h* . On y place la signature de la procédure *PutChar* (qui est donc notre appel système). On y ajoute également une constante qui permettra lorsqu'on est en mode noyau de savoir quel appel système a été demandé. Le nommage de cette constante se fait selon le principe d'écriture suivant :

$$SC\_ + nom\_de\_l'appel\_system$$

Ainsi pour le cas présent on a rajouté les lignes suivantes :

```
#define SC_PutChar 11
[...]
void PutChar(char c);
```

2. Ensuite il faut implémenter la signature de la procédure précédente. Ceci est fait dans le fichier *start.S* et cela donne donc :

```
        .globl PutChar
        .ent  PutChar

PutChar:
        addiu $2,$0,SC_PutChar /*on place la valeur de l appel systeme dans le registre r2. Pour cela
             on fait une addition avec le registre r0 qui contient la valeur 0*/
        syscall
        j $31
        .end PutChar
```

la ligne : *syscall* permet de faire l'appel système et de passer en mode noyau. La gestion des appels systèmes est alors fait dans le fichier *exception.cc*.

3. Comme Il y aura de nombreux appels systèmes à mettre en place par la suite, on met en place switch afin de pouvoir les gérer plus facilement. On notera au passsage que le type d'exception est récupéré dans le registre deux.

```
int type = machine->ReadRegister (2);
[..]
switch(type){
    default :{
      printf("Unexpected user mode exception %d %d\n", which, type);
      ASSERT(FALSE);
      break;
    }
}
```

Il ne reste plus qu'à placer notre exception :

```
          case SC_PutChar:{
            break;
          }
```

Il faut ensuite complèter ce cas là. Pour cela on lit dans le registre 4 le paramètre de la fonction utilisateur *PutChar*. Et enfin faire l'appel à la méthode *SynchPutChar* écrite dans la classe *SynchConsole*. Ce qui donne au finale :

```
case SC_PutChar:{
  int c = machine->ReadRegister(4); // registre contenant le parametre de la fonction
      appele
  DEBUG('a',"appel de la fonction SynchPutChar\n");
  synchconsole->SynchPutChar((char)c);
  DEBUG('a',"\nfin d'appel en mode kernel\n");
  break;
}
```

4. Enfin il mettre en place si besoin des fonctions supplémentaires permettant le bon fonctionnement de l'appel système. Dans le cas présent, la méthode *SynchPutChar* ayant déjà été implémentée dans la partie précédente, il n'y a pas besoin d'en rajouter.

Pour finir on met en place un petit programme de test qu'on appelera *putchar.c* :

```
#include "syscall.h"

int
main ()
{
     PutChar ('c');

    /* not reached */
    return 0;
}
```

Afin de pouvoir exécuter ce programme utilisateur, il faut également déclarer en global la *synchconsole* dans le fichier *threads/system.cc*. Il faut également l'initialiser et la détruire à la fin de l'exécution. Ce qui se fait respectivement dans les procédures *Initialize* et *Cleanup*.

Cette ajout dans le fichier *threads/system.cc* entraîne quelques modifications dans le fichier *progtest.cc* et ce parce qu'il ne peut exister plusieurs instances de la console :

1. Dans la procédure *ConsoleTest* on doit supprimer la *synchconsole* afin de pouvoir utiliser explicitement la console et d'avoir qu'une seule instance de celle-ci. Quand le programme quitte il faut reconstruire la *synchconsole* car elle sera de nouveau détruite dans la procédure *Cleanup*. On a procédé de cette manière afin de ne pas s'embourber dans des gestions de cas de destruction/construction dans le fichier *system.cc*
2. Dans la procédure *SynchConsoleTest*, qui prend en paramètre un fichier d'entré et de sortie, on supprime la *synchconsole* pour pouvoir la reconstruire avec les paramètres de la procédure. On aurait pu mettre en place un getter/setter.

Après avoir compilé et exécuté le programme de test via la commande *./nachos-userprog -x ./putchar*, une exception est levée et spécifie que le type *SC_EXIT* n'est pas pris en compte. C'est pourquoi on rajoute le cas *SC_EXIT* :

```
case SC_Exit:{ // cas appel lors d'un fin de programme sans appel  la fonction Halt()
  break;
}
```

5 Des caractères aux chaînes

Maintenant qu'on a vu comment afficher un caractère, on va maintenant voir comment afficher une chaîne de caractère.

5.1 Passer de mips à Linux

La première chose à faire est de passer d'un pointeur mips à un pointeur Linux. Pour cela on met en place la méthode *CopyStringFromMachine* que l'on place dans la classe *SynchConsole*. On a choisit de placer cette méthode dans cette classe car elle sera spécifiquement utiliser dans le cas de gestion des strings qui sont essentiellements gérés dans cette classe.

Code de la méthode CopyStringFromMachine

```
void SynchConsole::CopyStringFromMachine( int from, char *to, unsigned size)
{
        unsigned i = 0;
        int res;

        while((i<size)&&(machine->ReadMem(from+i,1,&res))){
                *(to+i)= (char)res;
                i++;
        }
        *(to+i)='\0';

}
```

Afin d'expliquer au mieux cette procédure, on va d'abord donner le détail des paramètres :
from : adresse virtuelle de la chaîne mips

to : pointeur Linux

size : taille max du nombre d'octet que l'on peut écrire

Il suffit maintenant de lire tous les caractères de la chaîne mips ce qui est fait avec *machine->ReadMem(from+i,1,&res))* qui lit un caractère à l'adresse *from+i* et met le résultat à l'adresse *&res*

le résultat est ensuite casté en *char* que l'on place à l'adresse *to+i*. Bien entendu on met le caractère de fin de chaîne à la fin.

5.2 La méthode SynchPutString

Cette partie est simple, il suffit de faire un appel à la méthode *SynchPutChar* pour chaque caractère contenu dans la chaîne passé en paramètre de la méthode.

Code de la méthode SynchPutString

```
void SynchConsole::SynchPutString(const char s[])
{
        int i = 0;

        while(*(s+i)!='\0'){
                SynchPutChar(*(s+i));
                i++;
        }
}
```

5.3 Fin de la mise en place de l'appel système

Maintenant que les briques sont en places pour que l'appel système se fasse bien, il n'y a plus qu'à les assembler.
Classiquement, on complète les fichiers *syscall.h* et *start.s* de la manière habituelle. On finit ensuite par complèter par le fichier *exception.cc*

Completion de exception.cc

```
case SC_SynchPutString:{
  int c = machine->ReadRegister (4); // recupration de la chaine de caractre
  char* to = new char[MAX_STRING_SIZE+1]; // buffer le +1 permet d'ajouter le caractere de fin de
      chaine
  synchconsole->CopyStringFromMachine(c, to, MAX_STRING_SIZE); // copie chaine mips vers
      chaine Linux
  DEBUG('a',"appel systme de la fonction SynchPutString\n");
  synchconsole->SynchPutString(to);
  delete [] to; //desallocation du buffer
  break;
}
```

On récupère la chaine mips dans le registre 4. On créé une chaîne Linux. On appel notre méthode qui copie une chaîne mips dans une chaîne linux et enfin, on appel notre méthode *SynchPutString*. On finit par supprimer notre chaîne Linux.

6 Fonctions de lecture

L'implémentation des fonctions de lecture est symétrique à l'implémentation des fonctions d'écriture. Pour chaque sous section qui suivra, on mettra en place un appel système correspondant de manière classique.

6.1 SynchGetChar

Vu que la méthode *SynchGetChar* est déjà implémentée dans la classe *SynchConsole*, il suffit de complèter le fichier *exception.cc*.

Completion de exception.cc

```
case SC_SynchGetChar:{
  char c = synchconsole->SynchGetChar();
  //printf("%c",c);
  machine->WriteRegister(2,(int)c); // ecriture dans le registre 2 du rsultat de la fonction
  break;
}
```

Afin de permettre à l'utilisateur de trouver le résultat renvoyé par *SynchGetChar* on l'écrit dans le registre 2.

6.2 SynchGetString

Afin de rendre symétrique l'utilisation de *SynchGetString* par rapport à *SynchPutString*, il est donc nécessaire de mettre en place une méthode qui permettra de transformer une chaîne Linux en une chaîne Mips. Ce qui se fera à l'aide la méthode *CopyMachineFromString*

CopyMachineFromString

```
void SynchConsole::CopyMachineFromString(char* from, int to, unsigned size){
        unsigned i = 0;
        int res;

        while((i<size)&&(*(from+i)!='\0')){
                res = *(from+i);
                machine->WriteMem(to+i,1,res);
                i++;
        }
        machine->WriteMem(to+i,1,'\0');

}
```

On commence par donner une rapide description des paramètres de la méthode :
from : pointeur de chaîne Linux

to : adresse virtuelle de chaîne mips

size : nombre max de caractère lu

À partir de là , l'implémentation est relativement simple. Il suffit de lire caractère par caractère la chaîne Linux et de les écrire à l'adresse mips. On veillera à mettre un marqueur de fin de chaîne.

Il est maintenant temps d'implémenter la méthode *SynchGetString* dans la classe *SynchConsole*

Implémentation de SynchGetString

```
void SynchConsole::SynchGetString(char *s, int n)
{
        int i = 0;
        char c;
        while((i<n)&&((c=SynchGetChar())!=EOF)&&(c!='\n')){
                *(s+i)=c;
                i++;
        }
        *(s+i) = '\0';
}
```

Cette méthode prend en paramètre un pointeur de chaîne Linux et la taille max du nombre de caractère pouvant être lu. Ensuite afin de respecter le cahier des charges, il faut que le programme s'arrête quand on lit le caractère de fin de chaîne ou un retour chariot.

Comme d'habitude on veillera qu'à la fin de la lecture, on place un marqueur de fin de chaîne.

Il faut maintenant complèter le fichier exception.cc afin de rassembler les briques précédement décrites.

Completion de exception.cc

```
case SC_SynchGetString:{
  int to = machine->ReadRegister(4);
  int taille = machine->ReadRegister(5); //recuperation du 2eme param de la fonction
      SynchGetString
  char* from = new char[taille];
  synchconsole->SynchGetString(from,taille-1);
  synchconsole->CopyMachineFromString(from,to,taille); //copie de chaine linux vers chaine
      mips
  delete [] from;
  break;
}
```

La fonction utilisateur SynchGetString possède comme son homologue noyau deux paramètres. Ceux-ci sont récupérés dans les registres 4 et 5. On lit les caractères tapés par l'utilisateur dans la console. Ces caractères sont placés dans une chaîne Linux. On copie ensuite la chaîne Linux en chaîne Mips.

NB : dans le cas où plusieurs thread accedent en même temps à la console, il y aura une erreur. Il faudra donc mettre en place un système de sémaphore plus poussé que celui actuel.

6.3 SynchPutInt

On va maintenant écrire des entiers signés. Pour cela on va utiliser la fonction *snprintf* de Linux qui consiste à écrire un entier dans une chaîne de caractère. Cette chaîne de caractère est ensuite écrite via la méthode *SynchPutString*.

Code de la methode SynchPutInt

```
void SynchConsole::SynchPutInt(int n){

        char* string = new char[NBREMAXCARACTENTIER];
        snprintf(string,NBREMAXCARACTENTIER,"%d",n); //ecrit n dans string
        SynchPutString(string);

        delete [] string;
}
```

Et on complète le fichier exception.cc :

Completion de exception.cc

```
          case SC_SynchPutInt:{
            int entier = machine->ReadRegister(4);
            synchconsole->SynchPutInt(entier);
            break;
          }
```

6.4 SynchGetInt

On va maintenant lire des entiers signés. Pour cela on va utiliser la fonction *sscanf*.

```
void SynchConsole::SynchGetInt( int *n){
        int* i = new int;
        char* string = new char[NBREMAXCARACTENTIER];
        SynchGetString(string,NBREMAXCARACTENTIER);
        sscanf(string,"%d",i);

        machine->WriteMem(*n,4,*i);
        delete [] string;
        delete i;
}
```

la fonction*sscanf* prend en paramètre un pointeur de chaîne Linux “string” et il va écrire l’entier contenu dans *string* dans le pointeur d’entier *i* également passer en paramètre.

La chaîne *string* est préalablement initialisée par la méthode *SynchGetString*.

Completion de exception.cc

Il ne reste plus qu’à complèter le code d’exception.cc qui se fait sans surprise.

```
          case SC_SynchGetInt:{
            int* n = new int;
            *n = machine->ReadRegister(4);
            synchconsole->SynchGetInt(n);
            delete n;
            break;
          }
```