



POLYTECH' GRENOBLE

RICM 4ÈME ANNÉE

NachOS

Etape 2: Entrées/Sorties

Étudiants:
Elizabeth PAZ
Salem HARRACHE

Professeur:
Vania MARANGOZOVA

février 2012

1 Entrées/Sorties asynchrones

l'objet `Console` est asynchrones, dans le programme de test (`ConsoleTest`) on gère manuellement la synchronisation avec deux sémaphores (un pour l'écriture et un autre pour la lecture) et des handlers qui seront appelés par le traitant une fois la tâche de lecture/écriture effectuée. Dans cette partie, on ajoute la prise en compte du caractère de fin de fichier (EOF).

code/usrprog/progtest.cc

```
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 or ch == 'q')
            return;
        console->PutChar('<');
        writeDone->P ();
        console->PutChar(ch); // We echo it
        writeDone->P ();
        console->PutChar('>');
        writeDone->P (); // wait for write to finish
    }
}
```

Comme on le voit, à chaque fois qu'on écrit ou lit un caractère il faut utiliser le sémaphore explicitement pour obliger le programme à se bloquer et reprendre qu'une fois le caractère lu/écrit

2 Entrées/Sorties synchrones

On va maintenant gérer la synchronisation directement dans la `Console`. Pour ça on va implementer la classe `SynchConsole` qui aura utilisé de façon transparente les sémaphores. On ajoute les deux fichiers *code/userprog/synchconsole.h* et *code/userprog/synchconsole.cc*

code/userprog/synchconsole.h

```
#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
```

code/userprog/synchconsole.cc

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

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) {
    /* Les s maphores sont initialis 0 car par d faut il n'y a rien lire
    * et on n'a rien crite
    */
    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;
    delete mutex;
}

void SynchConsole::SynchPutChar(const char ch) {
    /* On crit un char et on se bloque en attendant que le traitant appelle
    * (WriteDone>V())
    */
    console->PutChar(ch);
    writeDone->P();
}

char SynchConsole::SynchGetChar() {
    /* Lorsqu'il y a rien lire, on se bloque, et d s qu'il y a quelque chose
    * lire, on sait qu'on sera d bloqu (ReadAvail->V())
    */
    readAvail->P();
    return console->GetChar();
}

void SynchConsole::SynchPutString(const char string[]) {
}

void SynchConsole::SynchGetString(char *buffer, int n) {
}

```

On peut tester la synchconsole depuis le fichier *code/userprog/synchconsole.h*

```

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

```

3 Appels systèmes

Dans cette partie on va implémenter les appels systeme qui utiliseront notre synconsole.

3.1 PutChar

3.1.1 Mise en place

On commence par déclarer la fonction d'appel système *void PutChar(char c)* et son numéro *code/userprog/syscall.h*

```

#define SC_PutChar 11
// ...
void PutChar(char c);

```

Un appel système entraîne un changement d'environnement, du mode au mode noyau. Il faut donc écrire (en assembleur) le code qui permet de faire cette interruption pour basculer en mode noyau et prévoir le retour au programme utilisateur : *code/test/start.S*

```
.globl PutChar
.ent PutChar
PutChar:
/* On place le signal dans le registre R2
   Il va servir au handler d'exceptions pour qu'il puisse
   savoir qui sera le traitant de cette exception.
*/
addiu $2,$0,SC_PutChar
/* syscall provoque un d routement et place le compteur de
   programme (PC) la premi re instruction du traitant :
   ExceptionHandler
*/
syscall
/* Maintenant on revient au programme appelant
   Le registre R31 sauvegarde l'adresse de retour de la
   fonction appelante
*/
j $31
.end PutChar
```

Le traitement sera fait dans *code/userprog/exception.cc*. Pour faciliter l'ajout de nouveaux appels systèmes on utilisera un switch/case qui associe un traitement a chaque numéro d'appel système. *code/userprog/exception.cc*

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

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

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

            case SC_PutChar: {
                DEBUG('a', "PutChar, initiated by user program.\n");
                // On recupere le premier parametre
                char c = (char)(machine->ReadRegister(4));
                // on l'affiche dans gr ce a synchconsole
                synchconsole->SynchPutChar(c);
                break;
            }

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

    // LB: Do not forget to increment the pc before returning!
    UpdatePC ();
    // End of addition
}
```

L'objet synchconsole appartient au noyau, il faut donc l'initialiser dans le fichier *code/threads/system.cc* comme suit

```
#ifdef USER_PROGRAM // requires either FILESYS or FILESYS_STUB
Machine *machine; // user program memory and registers
SynchConsole *synchconsole; // On declare la synchconsole
#endif
// [...]
void Initialize (int argc, char **argv)
{
    // ...
#ifdef USER_PROGRAM
    machine = new Machine (debugUserProg); // this must come first
    synchconsole = new SynchConsole(NULL, NULL); // On declare la synchconsole sans
    aucun in/out
```

```
#endif
// ..
}

void Cleanup ()
{
// ..
#ifdef USER_PROGRAM
delete machine;
delete synchconsole;
#endif
// ..
}
```

code/threads/system.h

```
#ifdef USER_PROGRAM
#include "machine.h"
#include "../userprog/synchconsole.h"
extern Machine *machine; // user program memory and registers
extern SynchConsole *synchconsole;
#endif
```

Maintenant on peut utiliser `synchconsole` en important *system.h*.

3.1.2 Test de PutChar()

code/userprog/putchar.c

```
#include "syscall.h"

int main() {
    PutChar('a');
    PutChar('\n');
    Halt();
}
```

```
$ ./build-origin/nachos-userprog -x ./build/putchar
a
Machine halting!

Ticks: total 253, idle 200, system 30, user 23
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 2
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

3.1.3 Terminaison

Un programme est obligé d'appeler `Halt()` pour dire qu'il s'est terminé, ce qui n'est pas pratique en temps normal. Un programme est lancé par la méthode *Machine::Run()* qui ne termine pas :

code/machine/mipsim.cc

```
void Machine::Run()
{
    // [...]
    interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        interrupt->OneTick();
        if (singleStep && (runUntilTime <= stats->totalTicks))
            Debugger();
    }
}
```

L'absence de *Halt()* provoque une interruption :

```
./build-origin/nachos-userprog -x ./build/putchar
a
Unexpected user mode exception 1 1
```

L'interruption 1 correspond à l'appel système *Exit()*, il faut donc implementer cet appel système, qui se contentera, dans un premier temps, d'éteindre explicitement la machine:

code/userprog/exception.cc

```
case SC_Exit: {
    DEBUG('a', "Exit, initiated by user program.\n");
    interrupt->Halt();
    break;
}
```

```
$ ./build-origin/nachos-userprog -x ./build/putchar
a
Machine halting!

Ticks: total 260, idle 200, system 30, user 30
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 2
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

3.2 PutString

La difference entre PutChar et PutString c'est que PutString prend une pointeur sur une chaine de caractères en mémoire **user**. On va donc devoir par précaution, préalablement la copier dans un buffer en mémoire **noyau**.

code/userprog/exception.cc

```
char * ReadStringFromMachine(int from, unsigned max_size) {
    /* On copie octet par octet, de la mmoire user vers la mmoire noyau (buffer)
     * en faisant attention bien convertir explicitement en char
     */
    int byte;
    unsigned int i;
    char * buffer = new char[max_size];
    for(i = 0; i < max_size-1; i++) {
        machine->ReadMem(from+i, 1, &byte);
        if((char)byte=='\0')
            break;
        buffer[i] = (char) byte;
    }
    buffer[i] = '\0';
    return buffer;
}
```

La mie en place de l'appel systeme est similaire à *PutChar()*. Il faut simplement spécifier la taille du buffer de copie.

code/userprog/exception.cc

```
case SC_PutString: {
    DEBUG('a', "PutString, initiated by user program.\n");
    // Le premier argument (registre R4) c'est l'adresse de la chaine de
caractere
    // Que l'ont recopie dans le monde linux (noyau)
    // R4 >> pointeur vers la mmoire MIPS
    // MAX_STRING_SIZE est defni prealablement dans code/threads/system.h
    char *buffer = ReadStringFromMachine(machine->ReadRegister(4),
MAX_STRING_SIZE);
    synchconsole->SynchPutString(buffer);
    delete [] buffer;
    break;
}
```

Dans la SynchConsole on implemente *SynchConsole::SynchPutString* :

code/userprog/synchconsole.cc

```

void SynchConsole::SynchPutString(const char string[]) {
    /* On utilise un mutex pour que les appels SynchPutString soient atomiques
     * C'est dire que deux appels SynchPutString() affichent correctement
     * les chaines de caract res...
     * */
    mutex->P();
    for(int i=0; i<MAX_STRING_SIZE-1;i++) {
        if(string[i] == '\0')
            break;
        this->SynchPutChar(string[i]);
    }
    mutex->V();
}

```

On utilise un mutex (Sémaphore initialisé à 1) pour assuré l'atomicité de PutString. En effet, on souhaite avoir tous les caractères dans le bon ordre, et deux appels a PutString doivent se faire l'un après l'autre.

3.3 GetChar GetString

Ces deux appels systèmes sont symétriques à PutChar et PutString. Dans le cas de GetChar, rien de plus simple, étant donné qu'il renvoi directement la valeur (C'est le registre 2 qui est utilisé pour les valeurs de retour). On écrit directement dans le registre la valeur de retour de SynchGetChar().

code/userprog/exception.cc

```

case SC_GetChar: {
    DEBUG('a', "GetChar, initiated by user program.\n");
    machine->WriteRegister(2, (int) synchconsole->SynchGetChar());
    break;
}

```

Pour GetString, on écrit dans un buffer intermédiaire, puis on copie ce buffer dans la mémoire user à l'adresse donnée à l'appel système.

```

case SC_GetString: {
    DEBUG('a', "GetString, initiated by user program.\n");

    // le premier argument est une adresse (char *)
    int to = machine->ReadRegister(4);
    // le second est un int >> la taille
    int size = machine->ReadRegister(5);
    // On donne pas acces la m moire directement, on écrit écrit dans un
buffer
    // Peut etre pas oblig , mais au cas ou on utilise un buffer..
    char * buffer = new char[MAX_STRING_SIZE];
    synchconsole->SynchGetString(buffer, size);
    WriteStringToMachine(buffer, to, size);
    delete [] buffer;
    break;
}

```

```

void WriteStringToMachine(char * string, int to, unsigned max_size) {
    /* On copie octet par octet, en faisant attention bien convertir
     * explicitement en char
     */
    char * bytes = (char *)&machine->mainMemory[to];
    for(unsigned int i = 0; i < max_size-1; i++) {
        bytes[i] = string[i];
        if(string[i]=='\0')
            break;
    }
}

```

code/userprog/synchonsole.cc

```

void SynchConsole::SynchGetString(char *buffer, int n) {
    /* On utilise un mutex pour que tous les appels SynchGetString soient
     * atomiques.
     * */
    int i;
    char c;

```

```
mutex->P();
for (i=0; i<n-1; i++) {
    c = this->SynchGetChar();
    // CTRL+D pour arrêter la saisie
    if(c == EOF)
        break;
    else
        buffer[i] = c;
}
buffer[i] = '\0';
mutex->V();
}
```

3.4 PutInt et GetInt

Pour PutInt et GetInt on va utiliser les fonction **sscanf** et **snprintf**. Pour faciliter la saisie, on ajoute la fonction *SynchConsole::SynchGetString(char *buffer, int n, char delim)* qui permet de lire un chaîne de caractère et de s'arrêter dès qu'on rencontre un délimiteur (delim). Dans notre cas, on va utiliser '\n' comme délimiteur lors de la saisie de nombre entier :

code/userprog/synchconsole.cc

```
void SynchConsole::SynchPutInt(int value) {
    char * buffer = new char[MAX_STRING_SIZE];
    // on écrit dans le buffer la valeur avec sprintf
    snprintf(buffer, MAX_STRING_SIZE, "%d", value);
    this->SynchPutString(buffer);
    delete [] buffer;
}

int SynchConsole::SynchGetInt() {
    int value;
    char * buffer = new char[MAX_STRING_SIZE];
    this->SynchGetString(buffer, MAX_STRING_SIZE, '\n');
    sscanf(buffer, "%d", &value);
    delete [] buffer;
    return value;
}
```

code/userprog/exception.cc

```
case SC_PutInt: {
    DEBUG('a', "PutInt, initiated by user program.\n");
    // le premier est la valeur int
    int value = machine->ReadRegister(4);
    synchconsole->SynchPutInt(value);
    break;
}

case SC_GetInt: {
    DEBUG('a', "GetInt, initiated by user program.\n");
    int value = synchconsole->SynchGetInt();
    machine->WriteRegister(2, value);
    break;
}
```

4 Détection de fin de fichier

flemme...

5 Test Nachos étape 2

Voici le programme de test :

```
#include "syscall.h"

int main() {
    PutString("Veuillez saisir un nombre : \n");
    int nombre = GetInt();
    PutString("Nombre +10 = "); PutInt(nombre+10); PutChar('\n');
```



```
PutString("Veuillez saisir une lettre : \n");
char c = GetChar();
PutString("Voici la lettre : ");
PutChar(c);
PutString("\nVeuillez saisir une phrase (max = 100) : ");
char buffer[100];
GetString(buffer,100);
PutString("\nVoici la phrase : ");
PutString(buffer);
PutChar('\n');
return 0;
}
```

```
$ ./build-origin/nachos-userprog -x ./build/etape2
Veuillez saisir un nombre :
5
Nombre +10 = 15
Veuillez saisir une lettre :
g
Voici la lettre : g
Veuillez saisir une phrase (max = 100) : le mot de la fin

Voici la phrase :
le mot de la fin

Machine halting!

Ticks: total 1136142117, idle 1136139819, system 2170, user 128
Disk I/O: reads 0, writes 0
Console I/O: reads 22, writes 174
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```