Rapport sur le projet du système d'exploitation nachOS
Année 2013-2014

par Jerôme Barbier, Augustin Husson et David Levayer

8 avril 2014

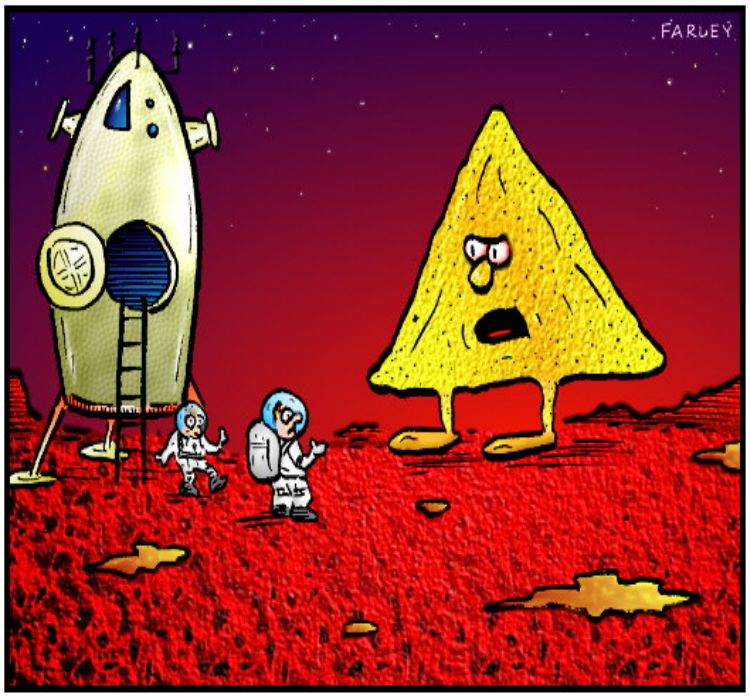
Table des matières

Première partie

Gestion des entrées/sorties

0.1 Introduction

NachOS est un mini-système d'exploitation à but éducatif. Simulé par un processeur MIPS, il permet d'appréhender les problématiques rencontrées dans tout système d'exploitation. Au cours de ce projet, nous allons faire évoluer une version basique de NachOS, en ajoutant des fonctionnalités clés telles que les entrée-sorties ou les threads. Cette approche permet d'expérimenter concretement les notions de système acquises au cours des deux dernières années. Chaque étape représente un concept précis. Dans un premier temps, nous verrons notre implémentation d'une console et du concept d'entrée-sortie. L'étape suivante nous menera à l'implémentation de threads utilisateurs, avec les différentes contraintes qui en découlent. Enfin, nous verrons dans une dernière partie comment généraliser le concept de thread pour l'étendre et permettre une exécution multi-processus



"This is the planet where nachos rule."

0.2 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;
}
```

0.3 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.

0.4 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.

0.5 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;
}
```

0.6 Des caractères aux chaînes

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

0.6.1 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.

0.6.2 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++;
        }
}
```

0.6.3 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.

0.7 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.

0.7.1 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.

0.7.2 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.

0.7.3 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;
          }
```

0.7.4 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;
          }
```

Deuxième partie

Multi-threading

0.8 Mise en place des threads utilisateurs

On va tâcher de permettre à l'utilisateur de créer des threads. Pour cela on met en place les appels systèmes suivants :

- int UserThreadCreate(void f(void *arg), void *arg) : cette fonction permettra de créer un thread et d'exécuter la fonction f dans ce thread Retourne -1 en cas d'erreur de création de thread.
- void UserThreadExit() : Cette procédure permettra de détruire le thread qu'on a créé précédement

A l'appel système UserThreadCreate, on récupère classiquement les paramètres dans les registres 4 et 5, on appel ensuite la fonction *do_UserThreadCreate*. Cette fonction est disponible dans *userthread.cc*. On va maintenant expliciter ce que contient ce fichier.

0.8.1 Fichier userthread.cc

Dans cette partie ce fichier comporte 3 fonctions :

- extern int do_UserThreadCreate(int f, int arg) : cette fonction est appelée par l'appel système *UserThreadCreate*
- static void StartUserThread(int f) : cette procédure est appelée par la fonction mère do_UserThreadCreate
- void do_UserThreadExit() cette procédure est appelée par l'appel système *UserThreadExit*

do_UserThreadCreate

Comme on l'a dit, cette fonction est appelée après un appel système. Dans un premier temps, elle va créer un thread :

```
Thread* newThread = new Thread("threadUser");
```

Afin de pouvoir lui allouer le même espace d'adressage que le processus père, il va falloir faire un fork du programme principal. Le problème est que la méthode *Fork* de la classe *Thread*, ne permet pas de passer en paramètre les arguments de la fonction f. (On rappel que f est la fonction que souhaite exécuter l'utilisateur dans un thread). C'est pourquoi on met en place la structure suivante :

```
struct Serialisation{
  int function; // adresse du pointeur de fonction
  int arg; // adresse du pointeur des arguments
};
```

Cette structure est mise dans le fichier *userthread.cc* afin qu'elle reste locale au thread utilisateur.

On crée donc cette structure et on l'initialise avec l'adresse de la fonction *f*, et avec l'adresse des arguments *arg*.

```
Serialisation* save = new Serialisation;
save->function = f;
save->arg = arg;
```

On finit par faire l'appel à la méthode *Fork*

```
newThread->Fork(StartUserThread,(int)save);
```

Le fork fait un appel à la procédure StartUserThread. On va donc maintenant parler de cette procédure

StartUserThread

Cette procédure permet d'initialiser correctement le thread qu'on a créé dans la fonction précédente. C'est à dire entre autre lui allouer une pile différente de celle du processus père.

Dans un premier temps, on s'attache à récupérer notre fonction *f* et ses paramètres *args*. Ce qui se fait avec la ligne ci-dessous :

```
Serialisation* restor = (Serialisation*) f;
```

On initialise ensuite les registres du thread, ce qui se fait classiquement en mettant des 0 partouts :

```
for(int i=0;i<NumTotalRegs;i++)
 {
        machine->WriteRegister(i,0);
 }
```

On va maintenant positionner le pc sur l'adresse de la fonction *f*. Ceci est fait afin qu'au lancement du thread, la première ligne exécutée soit celle correspondante à *f*.

```
machine->WriteRegister(PCReg,restor->function);
```

Bien évidemment pour que f s'exécute correctement, il est nécessaire de lui fournir ses paramètres qu'elle devra retrouver dans le registre 4.

```
machine->WriteRegister(4,restor->arg);
```

la machine nachOS possède un pointeur *pcNext* qui est placé à l'instruction qui suit celle pointée par *pc*. La prochaine instruction se trouve classiquement 4 octets après pc. D'où :

```
machine->WriteRegister(NextPCReg,restor->function+4);
```

Il reste maintenant à positionner le pointeur de pile :

```
machine->WriteRegister(StackReg,currentThread->space->BeginPointStack());
```

Bien évidemment on expliquera par la suite (i.e dans le fichier *addrspace.cc* comment fonctionne la méthode *BeginPointStack*

Maintenant que tous les registres ont été correctement initialisés, il est temps de lancer le programme!!

```
machine->Run();
```

0.8.2 Fichier addrspace.cc

Afin de positionner le pointeur de pile du nouveau thread, il a fallu modifier le fichier *addrspace.cc et addrspace.h* qui permettent de gérer l'espace mémoir des threads/processus. La principale modification est l'ajout d'un attribut de type *BitMap* ainsi que l'ajout d'une méthode : *BeginPointStack*

L'attribut BitMap

Une bitMap est un tableau de bit. Il permet d'avoir une représentation de l'état de la mémoire alouée. C'est à dire qu'à chaque création de thread, on fera une demande à la bitMap afin de savoir s'il reste des pages mémoires qui peuvent être alouées au thread. Chaque bit de la bitMap correspond à un certain nombre de page. Ce nombre est déterminé dans le fichier *addrspace.h*par :

```
#define PagePerThread 2
```

La méthode BeginPointStack

Tout d'abord, voici le code de cette méthode :

```
int AddrSpace::BeginPointStack(){
  int find = bitmapThreadStack->Find();

  ASSERT(find != -1 );
  currentThread->SetIdThread(find);
  return numPages*PageSize - find*PagePerThread*PageSize;
}
```

Le but de cette méthode est de déterminé l'adresse de début du pointeur de pile du nouveau thread. Il faut donc éviter que les piles des threads d'un même processus ne se chevauchent pas.

Pour cela, on cherche dans un premier temps le 1er bit à 0 dans la bitMap qui permet donc de déterminer s'il reste de la place dans la mémoire pour un nouveau thread. Pour permettre de faire la suite du sujet, on prévoit de mettre un id par thread. Par convention, on choisit de dire que l'id du thread correspond à l'indice du tableau de la bitMap.

Le calcul retourné est de la pure logique.

0.9 Plusieurs threads par processus

Cette section est là pour permettre de mettre en place des sémaphores afin de garentir une sécurité à l'accès de zone critique (tel que la console), ou pour déterminer quand doit terminer le programme principal.

0.9.1 Protection de la console

Si plusieurs threads accèdent en même temps à la console, une erreur est générée. On doit donc mettre en place des sémaphores qui garentiront son accès.

Pour cela il faut mettre en place 4 sémaphores dans le fichier *synchconsole.cc*.Vu qu'on souhaite qu'il n'y ait qu'un thread qui est accès à la console en même temps, les sémaphores sont donc initialisés à 1 :

```
writeChar = new Semaphore("write char",1);
readChar = new Semaphore("read char",1);

writeString = new Semaphore("write string",1);
readString = new Semaphore("read string",1);
```

Comme leurs noms l'indiquent, on a deux sémaphores lors de la lecture/écriture d'un caractère, et deux sémaphores pour la lecture/écriture d'un string. Si on avait protégé l'accès que lors de la lecture/écriture d'un caractère, on n'aurait pas pu lire une chaîne de caractère. En effet, on aurait juste récupéré des morceaux de la chaîne de caractère dans un thread, les autres morceaux auraient été dans d'autres threads.

La protection de la lecture/écriture d'un entier est inutile vu qu'en fait les entiers sont traduits comme des chaînes de caractères qui elles sont protégées comme on l'a vu plus haut.

0.9.2 Fin du programme principal

On souhaite que le programme principal s'arrête uniquement quand tous les threads secondaires ont terminés leur exécution. On va donc mettre en place un thread dans le fichier *addrspace*. Ce sémaphore est mit dans ce fichier afin qu'il soit commun à tous les threads d'un même processus. On veut que les threads secondaires puissent libérer le thread principal quand ils ont finis leur exécution. D'où la nécessité d'avoir un sémaphore commun.

Ce nouveau sémaphore est bien sûr initialisé à 0 :

```
lockEndMain = new Semaphore("lock at the end",0);
```

On met ensuite en place deux méthodes de classe permettant de décrémenter ou d'incrémenter la ressource du sémaphore :

```
void AddrSpace::LockEndMain(){
  lockEndMain->P();
 }

 void AddrSpace::FreeEndMain(){
  lockEndMain->V();
 }
```

Ensuite lorsque le programme principal est sur le point de faire un *halt()*, il va demander à la bitMap de lui dire s'il y a plus de 1 bit à 1. Si c'est le cas, alors il se bloque en attendant que tous les bits (à part le sien) soit à 0. Cette partie du code se trouve dans *exception.cc* :

```
case SC_Halt:{
      DEBUG('a', "Shutdown, initiated by user program.\n");
      while(currentThread->space->NbreThread()>1) // tant qu'il y a plus que un thread on reste
          bloquer
        currentThread->space->LockEndMain();

      interrupt->Halt();
      break;
    }
```

Enfin lorsqu'un thread termine son exécution, c'est à dire lorsqu'il appel la procédure *do_ UserThreadExit* il va mettre à 0 le bit qui lui correspond dans la bitMap et libérer une ressource dans le sémaphore *lockEndMain* :

```
void do_UserThreadExit(){
    // on signal au main qu'on a fini l'excution du thread
    currentThread->space->FreeEndMain();
    //fin du thread
    currentThread->space->DealloateMapStack();
    currentThread->Finish ();
}
```

0.9.3 Appel système UserThreadJoin

Ici on met en place un mécanisme permettant à un thread d'utilisateur d'attendre la terminaison d'un autre thread utilisateur. On va donc de nouveau mettre en place des sémaphores. Pour cela, on créé un sémaphore pour chaque bit de la bitMap. Le nombre de bit correspond au nombre de thread qu'il est possible de créer. Comme pour le sémaphore permettant de déterminer si oui ou non le thread principal peut s'arrêter, on va mettre ses threads dans le fichier *addrspace.cc* afin qu'ils soient global à tous les threads d'un processus donné :

```
AddrSpace::AddrSpace (OpenFile * executable)
{
    [...]
     int lengthBitMap = (int)(UserStackSize/(PagePerThread*PageSize));
     int j;

     for(j = 0; j<lengthBitMap; j++){
       waitOtherThread[j] = new Semaphore("wait executing other thread",0);
     }
}
```

On met ensuite en place deux méthodes permettant de libérer ou de prendre une ressource pour un id de thread donné :

```
void AddrSpace::LockIdThread(int id){
  waitOtherThread[id]->P();
}

void AddrSpace::FreeIdThread(int id){
  waitOtherThread[id]->V();
}
```

L'appel système utilisateur *UserThreadJoin* va appeler la procédure *(*do_UserThreadJoin) qui est dans le fichier *userthread.cc* :

```
void do_UserThreadJoin(int idThread){
    ASSERT(idThread!=0)// un thread ne doit jamais pouvoir attendre la fin du main avant de s'executer
    currentThread->space->LockIdThread(idThread);
}
```

Comme on le voit, lors de cet appel système, on attend que le thread d'id *idThread* ait finit son exécution. Il peut y avoir un problème si on attend la fin d'exécution d'un thread qui n'existe pas.

La libération de cette ressource se fait dans la procédure *do_ UserThreadExit*. On rajoute cette ligne de code dans cette procédure :

```
currentThread->space->FreeIdThread(currentThread->GetIdThread());
```

Troisième partie

La Pagination

0.10 Adressage virtuelle par une table des pages

En regardant le fichier addrspace.cc, on remarque un appel à readAt. La fonction readAt, présente dans openfile.cc, contient notamment le code suivant :

```
// read in all the full and partial sectors that we need
buf = new char[numSectors * SectorSize];
for (i = firstSector; i <= lastSector; i++)
  synchDisk->ReadSector(hdr->ByteToSector(i * SectorSize),
&buf[(i - firstSector) * SectorSize]);

// copy the part we want
bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
```

ReadAt utilise directement la mémoire (et les secteurs) physique. Pour gérer plusieurs processus en même temps, il va falloir changer ça. D'où l'intérêt de passer par des adresses virtuelles, notamment pour ̈compartimenter c̈haque processus (et garantir un accès exclusif à cette portion de mémoire).

L'écriture de ReadAtVirtual passe par plusieurs étapes clés. On commence par faire un appel à readAt (cet appel est désormais masqué du point de vue de addrspace). Le résultat est stocké dans un buffer créé pour l'occasion.

```
// On appelle ReadAt et on stocke le rsultat (donnes lues) dans un buffer local
char buffer[numBytes];
int size = executable->ReadAt(buffer, numBytes, position);
```

On sauvegarde ensuite la table des pages courante (pour pouvoir la restaurer) avant de charger la table des pages passée en paramètre. On peut désormais recopier le buffer sur les pages virtuelles (avec la fonction writeMem). Le processus pourra manipuler ces pages virtuelles, sans avoir conscience de ne pas manipuler directement les frames physiques.

```
// Au pralable, on sauvegarde la page des tables et on charge celle fournie lors de l'appel
int i;
for(i=0;i<size;i++) {
  machine->WriteMem(virtualaddr+i, 1, *(buffer+i));
}
// Ici, on restaure la table sauvegarde
```

Reste maintenant à définir la méthode de translation pages virtuelles -> pages physiques. Dans un premier temps, on se contente d'une simple incrémentation (dans addrspace.cc) :

```
pageTable[i].virtualPage = i;
pageTable[i].physicalPage = i + 1;
```

Après ces modifications, nos programmes fonctionnent toujours. C'est plutôt bon signe !

Pour encapsuler les pages physiques dans des pages virtuelles, on va maintenant utiliser un frameProvider. Ce dernier aura pour rôle de recenser les pages physiques disponibles (via une bitmap) et de fournir (sur demande)des pages physiques libres et vierges (fonction bzero).

```
/*recuprer un cadre libre et initialise  0 par la fonction bzero*/
int FrameProvider::GetEmptyFrame(){
  int frame = myFrame->Find();
  ASSERT(frame!=-1)
  //on doit utiliser le symbole '&' car bzero  besoin d'une adresse
  bzero(&machine->mainMemory[frame*PageSize], PageSize);
  return frame;
}
```

Déclaration du frame dans machine.h et machine.cc :

```
FrameProvider* myFrameProvider;
myFrameProvider = new FrameProvider((int)(MemorySize/PageSize));
```

Puis on adapte addrspace.cc :

```
pageTable[i].physicalPage = machine->myFrameProvider->GetEmptyFrame();
```

0.11 Exécuter plusieurs programmes en même temps

On met en place de manière classique l'appel système :

```
int ForkExec( char *s)
```

Pour rappel, on met en place un appel système en faisant les étapes suivantes :

1. Dans *syscall.h*, on écrit le prototype de notre appel système et on ajoute également une constante suivant le principe de nommage suivant :

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

   Dans le cas présent on a rajouté ceci :

   ```
   #define SC_ForkExec 20
   [...]
   int ForkExec( char *s);
   ```

2. On complète ensuite le fichier *Start.s* qui implémente le fichier *syscall.h* en assembleur. Ce qui donne ici :

   ```
           .globl ForkExec
           .ent ForkExec

   ForkExec: // nom de l'appel systme
           addiu $2,$0,SC_ForkExec // ajout dans le registre 2 la valeur correspondant  l'appel syteme
           syscall // appel systme
           j $31
           .end ForkExec
   ```

3. Puis on s'efforce de complèter le fichier *exception.cc* qui permet de gérer l'aiguillage vers les différents appels systèmes mise en place tout au long du projet. Les différents cas étant gérer dans un switch, il suffit donc de rajouter notre "cas" comme ceci :

   ```
   case SC_ForkExec:{
     break;
   }
   ```

   Bien entendu on tâchera de complèter ce cas là afin de répondre aux exigences du cahier des charges. On expliquera par la suite ce qu'il faut ajouter ici.

4. Enfin il est nécessaire de créer un fichier de test qui permettra de tester cet appel sytème. On pensera à compiler à cet instant afin de bien vérifier qu'aucun oublie dans ces étapes n'a été fait avant de complèter le code de *exception.cc*

0.11.1 Un processus : un thread de plus haut niveau

Comme l'indique le sujet de la sous-section, on va créer un processus en suivant le même procéder que lorsqu'on a créé des threads dans la partie III. On commence donc par créer deux fichiers : *fork.cc* et *fork.h*. Le second fichier ne présente guère de difficulter. Il contient simplement les signatures des fonctions qu'on a besoin. En l'occurance, nous avons besoin que d'une seule fonction publique :

```
int do_UserFork(char * s);
```

Et maintenant voici le code que contient le fichier *fork.cc*. On va bien sûr ajouter quelques explications en plus des commentaires du code qui y sont déjà.

Le fichier fork.cc

```
#ifdef CHANGED
#include "fork.h"
#include "thread.h"
#include "addrspace.h"
#include "synch.h"
#include "system.h"
#include "console.h"

struct Serialisation{
        AddrSpace* space;
};

void StartProcess(int arg){
        Serialisation* restor = (Serialisation*) arg; // on restaure notre serialisation
        currentThread->space = restor->space; // on affecte le nouvel espace mmoir  notre nouveau
            processus
        currentThread->space->InitRegisters (); // on reinitialise les registres
        currentThread->space->RestoreState (); // on charge la table des pages des registres

        machine->Run ();  // on lance le processus
}

int do_UserFork(char *s){

        OpenFile *executable = fileSystem->Open (s);
        AddrSpace *space = new AddrSpace(executable); // creation du nouvel espace mmoir du processus
            que l'on va mettre en place

        Thread* newThread = new Thread("newProcess"); // un processus est juste un thread avec un nouvel
             espace memoir

        Serialisation* save = new Serialisation; // comme pour les threads, on serialise l'espace
            memoir qu'on souhaite affecter  notre processus
        save->space = space;

        newThread->Fork(StartProcess,(int)save); // on fork le processus pere
        machine->SetNbProcess(machine->GetNbProcess()+1); // on incremente de 1 le nbre de
            processus cree
        delete executable;
        currentThread->Yield(); // le processus pere est mis en attente
        return 0;
}

#endif
```

Les explications

Tout d'abord, il faut savoir que pour réaliser cette implémentation, il y avait deux solutions qui résident dans le fait d'utiliser ou non la méthode native *Fork* de la classe *Thread*. On rappel que la méthode Fork est destinée à créer un thread fils d'un processus père et non pas à créer un autre processus père.

Ainsi, détourner l'objectif premier de cette méthode peut peut-être rebiffer les puristes. Afin de satisfaire les puritains et de montrer qu'il y avait plusieurs solutions possibles, deux implémentations différentes sont mises en place.

- La première ne modifie pas la classe *Thread* et utilise les méthodes déjà en place. Ce qui permet donc de ne pas rajouter du code supplémentaire dans les classes du système. Cependant pour contourner les restrictions de la méthode *Fork*, une sérialisation a été mise en place. Elle permet de sauvegarder le nouvel espace mémoir qu'on souhaite alloué au processus créé.
- La seconde méthode consiste à faire une surcharge de la méthode *Fork* en ajoutant comme paramêtre l'espace mémoir que le nouveau processus doit occuper. Ensuite cette nouvelle méthode fait exactement la même chose que son original au détail près relatif à la mémoire allouée.

Dans le code ci-dessus, c'est la première méthode qui a été mise en place pour faire fonctionner l'appel système. Notez bien que ce choix est totalement arbitraire. Par ailleurs, si on souhaitait mettre en place la deuxième méthode, il suffirait de remplacer la ligne :

```
newThread->Fork(StartProcess,(int)save);
```

par :

```
newThread->ForkExec(StartProcess,NULL,(int)space);
```

Et bien entendu dans la procédure *StartProcess*, il faudra commenter les deux premières lignes qui ne sont plus utiles pour la 2^{eme} solution.

On remarquera que le 3^{eme} argument de la méthode *ForkExec* est de type *int* et pas de type *AddrSpace**. Cette spécificité est une astuce pour éviter des appels réccursifs à la bibliothèque *addrspace.h* dans la classe *Thread*.

La completion du fichier execption.cc

Maintenant que tout est en place pour faire enfin notre appel système, il est grand temps de faire le pont entre l'utilisateur et le système et donc de complêter le fichier *exception.cc*. Sans plus attendre voici ce qui est rajouté :

```
case SC_ForkExec:{
  int arg = machine->ReadRegister(4);
  char* to = new char[MAX_STRING_SIZE+1]; // buffer le +1 permet d'ajouter le caractere de fin de
      chaine
  synchconsole->CopyStringFromMachine(arg, to, MAX_STRING_SIZE);
  int res = do_UserFork(to);

  ASSERT(res==0);
  break;
}
```

Donc classiquement on retrouve les arguments de l'appel système dans le registre 4. Et comme toujours, on a juste l'adresse de l'argument qui est donc de type int. Hors, on sait que l'argument est une chaîne de caractère. Il faut donc refaire la même procédure que pour l'appel système *SynchPutString*. Et c'est donc ce qui est fait là avec la variable*char* to* et l'appel à la méthode *CopyStringFromMachine*.

0.11.2 Main ! ! Ne vas pas trop vite

Après avoir testé ce nouvel appel système, on se rend rapidement compte que le nouveau processus a à peine le temps de s'executer que la machine nachOS est arrêté. Il faut donc pouvoir empêcher que l'appel à la méthode *Halt* qui met stoppe nachOS de s'exécuter tant que tous les processus n'ont pas fini leur exécution.

Dans ce but, on met en place un compteur de processus qui se fait donc dans la classe *machine*. Ce compteur est l'attribut *nbProcess*. Cet attribut étant privé, un getter et setter (*GetNbProcess* et *SetNbProcess*) sont mis en place pour le modifier.

De cette façon à chaque création d'un processus (i.e à chaque appel système *ForkExec*) on incrémente ce compteur de 1. Et lorsqu'un processus est sur le point de s'arrêter :

- On teste tout d'abord combien il y a de processus en cours d'exécution. Le compteur étant initialement mis à 0, si le compteur est supérieur strict à 0, alors y a plus de deux processus qui existent.
- Dans le cas où il y a plus de deux processus, celui qui souhaite s'arrêter, décrémente alors de 1 le compteur et exécute la méthode *Finish()*
- Dans le cas où il y a un seul processus, on fait alors un appel à la méthode *Halt*.

Bien entendu ce code se met après celui qui vérifie que tous les threads d'un processus se sont arrêtés. Et pour conclure ces explications voici le code qui y correspond :

```
case SC_Halt:{
  DEBUG('a', "Shutdown, initiated by user program.\n");
  while(currentThread->space->NbreThread()>1) // tant qu'il y a plus que un thread on reste
      bloque
    currentThread->space->LockEndMain();

  if(machine->GetNbProcess() > 0){
    machine->SetNbProcess(machine->GetNbProcess()-1);
    currentThread->Finish();
  }
  interrupt->Halt();
  break;
}
```

0.12 Conclusion

Pour conclure, ce projet entre en adéquation avec les compétences acquises lors de notre parcour et plus particulièrement en rapport avec notre option : Architectures des Réseaux. Suite à nos cours d'architectures et de système d'exploitation, le principe de plusieurs concepts abordés dans ce projet ne nous était pas inconnu (tel que les semaphores ou encore les threads). Cependant, la mise en place des différents objectifs nous on demandé beaucoup de réflexion pour quelques lignes de code. C'est pourquoi ici la quantité ne reflète pas la qualité de la réflexion.

Le projet fut, dans sa globalité, très interressant et nous as permis de comprendre comment fonctionné les outils que nous utilisions déjà. Il est tout de même dommage de ne pas avoir eut le temps de finir tous les objectifs.