Compte Rendu
NACHOS

BARTHELEMY Romain, EUDES Robin, MORISON Jack, ROSSI Ombeline

4 avril 2015

Table des matières

1 Introduction

Ce projet a été réalisé dans le cadre de notre 4ème année d'étude à Polytech, avec la spécialisation "systèmes et réseaux". En réalisant ce projet, nous avons pu mettre en pratique l'ensemble des connaissances engrangées au cours de nos parcours et ainsi comprendre les concepts entrant en jeu lors de la réalisation d'un système d'exploitation.

Dans un premier temps, nous allons nous intéresser au fonctionnement d'un appel système, comment le système les détecte, les gère. Dans une seconde partie, nous étudierons le multithreading, et de façon plus générale la gestion des threads par NachOS. Enfin, dans un troisième chapitre, nous chercherons à mettre en place un système de gestion de mémoire par la pagination.

2 Étape 2 : Etude d'un syscall

2.1 Entrées-sorties asynchrones

Une version élémentaire de gestion des entrées-sorties nous est fournie par NachOS, au travers de la classe *Console*. Le code fourni effectue une gestion asynchrone des entrées-sorties. Nous devons donc gérer la synchronisation grâce à deux sémaphores (pour gérer l'écriture et la lecture) ainsi que deux handlers. Ceux-ci libèreront le sémaphore et nous informeront de la fin de l'opération de lecture/écriture. Ainsi, la synchronisation est assurée par ce mécanisme.

Voici un extrait de code permettant cette gestion des entrées/sorties. Si le caractère est EOF, la machine s'arrête.

```
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 ();

          #ifdef CHANGED
          if(ch!='\n' && ch!=EOF){
                  console->PutChar ('<');
                  writeDone->P ();
          }
          #endif

          // Original code
          #ifndef CHANGED
          console->PutChar (ch);
          writeDone->P (); // wait for write to finish
          if (ch == 'q')
                  return;
          #else

          // Now, we prefer to exit on EOF,
          // only if it's at the beginning of a new line.
          if(ch!=EOF){
                  console->PutChar (ch);
                  writeDone->P ();
                  if(ch!='\n'){
                          console->PutChar ('>');
                          writeDone->P ();
                  }
          }
          else{
                return;
          }
          if (ch=='\0'){ // EOT
                  return;
          }
      #endif
      }
}
```

2.2 Entrées-sorties synchrones

Nous devons maintenant créer une classe *SynchConsole* afin de réaliser les opérations de synchronisation d'entrées/sorties automatiquement :

```
static Semaphore *readAvail;
static Semaphore *writeDone;
static Semaphore *SemPut;
static Semaphore *SemGetChar;
static Semaphore *SemGetString;

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);

        SemPut = new Semaphore("Put", 1);
        SemGetChar = new Semaphore("GetChar", 1);
        SemGetString = new Semaphore("GetString", 1);
}

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

void SynchConsole::SynchPutChar(const char ch)
{
        SemPut->P();
        console->PutChar (ch);
        writeDone->P ();        // wait for write to finish
        SemPut->V();
}

char SynchConsole::SynchGetChar()
{
        SemGetChar->P();
        char ch;
        readAvail->P ();        // wait for character to arrive
        ch = console->GetChar ();
        SemGetChar->V();
        return ch;
}
```

Le test de ces méthodes est réalisé dans *progtest.cc* par la fonction *SynchConsoleTest*.

```
 void SynchConsoleTest (char *in, char *out){
        char ch;
        SynchConsole *synchconsoletest = new SynchConsole(in, out);

        while ((ch = synchconsoletest->SynchGetChar()) != EOF){
                if(ch!='\n'){
                        synchconsoletest->SynchPutChar('<');
                        synchconsoletest->SynchPutChar(ch);
                        synchconsoletest->SynchPutChar('>');
                }
                else{
                        synchconsoletest->SynchPutChar(ch);
                }
        }
        fprintf(stderr, "Solaris: EOF detected in SynchConsole!\n");
}
```

Note : Chaque caractère est par ailleurs encadré par < >

Le fichier *main.cc* est modifié pour prendre en compte l'option -sc qui permettra l'éxecution de la console synchrone. Initialement, la création de la console était effectuée dans *system.cc , fonction Initialize*, mais suite à des erreurs rencontrées dans les phases de test, l'instanciation de SynchConsole a été déplacée dans le main.

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

En conséquence, la fonction *Cleanup() (system.cc)* est modifiée, pour supprimer cette nouvelle console lors de l'arrêt de NachOS.

```
#ifdef CHANGED
    delete synchconsole;
#endif //CHANGED
```

2.3 Création du syscall Putchar

Pour réaliser cet appel système, nous modifions *syscall.h*, afin d'y ajouter une constante associée au syscall putchar. Cette constante indiquera au handler la nature de l'exception (*exception.cc, fonction ExceptionHandler*).

```
#define SC_PutChar              11
```

Le syscall est ensuite défini dans *start.S* (en assembleur), en nous inspirant des syscall existants.

```
        .globl PutChar
        .ent    PutChar
PutChar:
        addiu $2,$0,SC_PutChar
        syscall
        j       $31
        .end PutChar
```

Le syscall *PutChar* défini, il nous reste à mettre en place le handler qui se chargera de la gestion des exceptions relatives à PutChar (*exception.cc, fonction ExceptionHandler*) :

```
if (which == SyscallException){
        switch(type){
                case SC_Halt:{
                        DEBUG ('a', "Shutdown, initiated by user program.\n");
                        interrupt->Halt ();
                        break;
                }
                case SC_PutChar:{
                        int c = machine->ReadRegister (4);
                        synchconsole->SynchPutChar((char)c);
                        break;
                }
                default:{
                        printf ("Unexpected user mode exception %d %d\n", which,
                             type);
                        ASSERT (FALSE);
                }
        }
}
```

2.4 La manipulation de chaine de caractères

La manipulation des string nous permet d'étudier les spécificités de la simulation d'un système d'exploitation par NachOS. En effet, nous devons jongler entre 2 espaces mémoire : l'espace MIPS (NachOS) et l'espace Linux.

```
// Used for SynchPutString
// get string from mips memory space, put it in linux memory space
void copyStringFromMachine( int from, char *to, unsigned size){
        unsigned int i;
        int tmp;
        for(i=0;i<size;i++){
                if(machine->ReadMem(from+i,1,&tmp))
                to[i]=tmp;
        }
        if(tmp!='\0'){
                to[size-1]='\0';
        }
}
// Used for SynchGetString
// get string from linux memory space, put it to mips memory space
void copyStringToMachine( char *from, int to, unsigned size){
        unsigned int i;
        int tmp;
        for(i=0;i<size-1;i++){
                tmp=from[i];
                machine->WriteMem(to+i,1,tmp);
        }
        tmp='\0';
        machine->WriteMem(to+i,1,tmp);
}
```

Nous devons ensuite ajouter les syscall associés SynchPutString et SynchGetString (*start.S*) :

```
 SynchPutString:
        addiu $2,$0,SC_SynchPutString
        syscall
        j       $31
        .end SynchPutString

        .globl SynchGetChar
        .ent    SynchGetChar
SynchGetString:
        addiu $2,$0,SC_SynchGetString
        syscall
        j       $31
        .end SynchGetString

        .globl SynchPutInt
        .ent    SynchPutInt
```

Et mettre en place les handlers associés, comme pour les précédents appels système. (*exception.cc, fonction ExceptionHandler*) :

```
case SC_SynchPutString:{
        char *buffer=new char[MAX_STRING_SIZE];
        int s = machine->ReadRegister (4);
        copyStringFromMachine(s, buffer, MAX_STRING_SIZE);
        synchconsole->SynchPutString(buffer);
        delete buffer;
        break;
}
case SC_SynchGetString:{
        char *buffer=new char[MAX_STRING_SIZE];
        int s = machine->ReadRegister (4);
        int size = machine->ReadRegister (5);
        synchconsole->SynchGetString(buffer,size);
        copyStringToMachine(buffer, s, size);
        delete buffer;
        break;
}
```

Note : MAX_STRING_SIZE , SC_SynchPutString, SC_SynchGetString sont définis dans system.h

Enfin, les fonctions SynchPutString et SynchGetString sont définies dans *SynchConsole.cc*, elles seront appelé par le handler associé.

```
void SynchConsole::SynchPutString(const char s[])
{
        SemPut->P();
        int i;
        for (i=0;i<MAX_STRING_SIZE && s[i]!='\0';i++){
                if (s[i]=='\0')
                        return;         // if end of string, quit
                        console->PutChar ((char)s[i]);
                        writeDone->P ();        // wait for write to finish
        }
        SemPut->V();
}

void SynchConsole::SynchGetString(char *s, int n)
{
        SemGetString->P();
        char c;
        int i;

        c = synchconsole->SynchGetChar ();
        if(c==EOF || c=='\n'){
                s[0]='\0';
                SemGetString->V();
                return;
        }
        else
                s[0] = c;
        for (i=1;i<n;i++){
                c = synchconsole->SynchGetChar ();
                if(c==EOF && s[i-1]=='\n')
                        break;
                else{
                        if(c==EOF)
                                i--;
                        else
                                s[i] = c;
                }
        }
        s[i]='\0';
        SemGetString->V();
}
```

La méthode SynchGetString est un peu plus complexe que SynchPutString car nous devons maintenir un comportement : Si EOF est vu en début de ligne, on termine la console, sinon ce dernier est ignoré (comme dans un système Linux).

2.5 Déroulement global d'un syscall

Par cet exercice autour d'une console synchrone, nous comprenons désormais le mécanisme d'appel système.

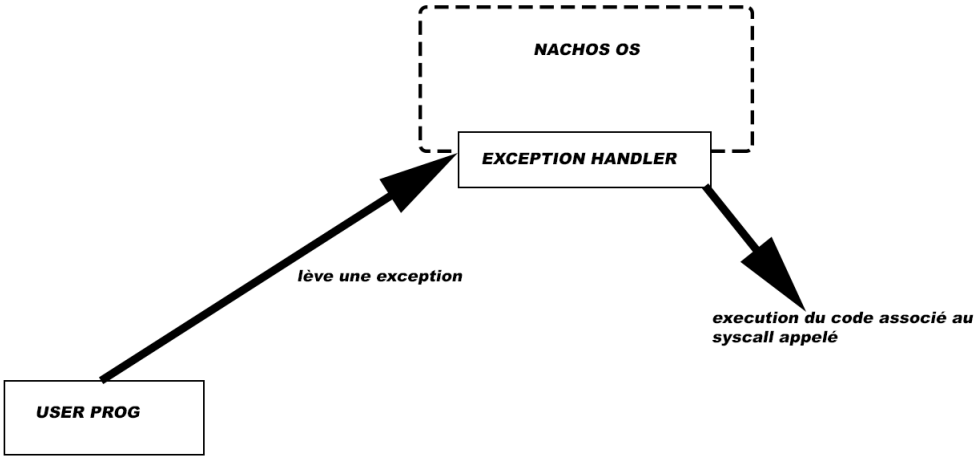

FIGURE 1 – Mécanique simplifiée d'un syscall

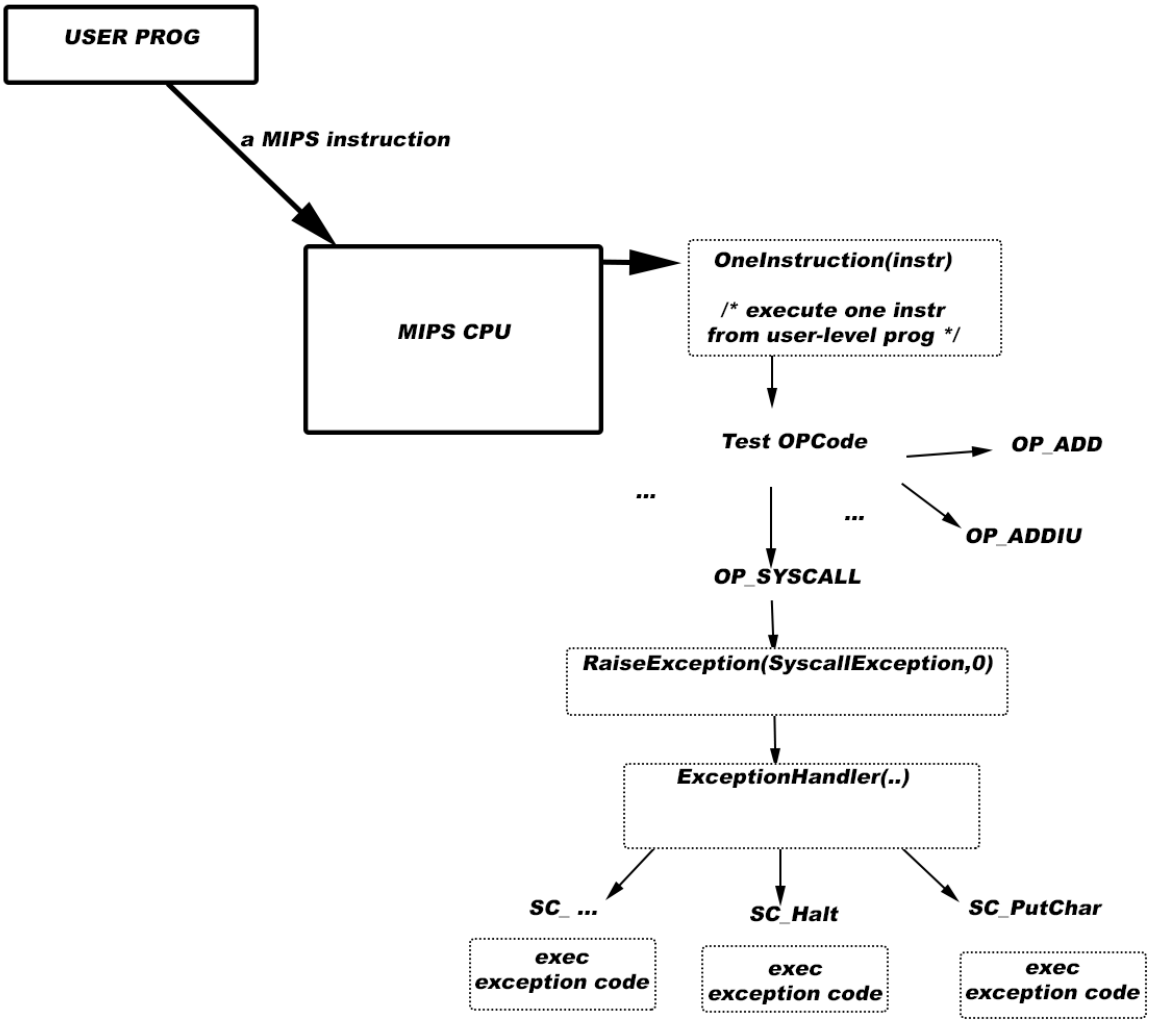

FIGURE 2 – Mécanique détaillée d'un syscall

On retrouve par ailleurs les mécaniques d'intéruptions vu en RICM3, dans le code de la fonction RaiseException (machine.cc). On passe en Kernel Mode pour la gestion de l'exception, puis on revient en User Mode.

3 Étape 3 : Multithreading

Nous nous intéréssons désormais aux threads. Nous souhaitons au terme de cette étape pouvoir éxecuter des programmes utilisateurs multi-thread sur NachOS. Nous allons d'abord comprendre le fonctionnement d'un thread NachOS, afin de pouvoir ensuite utiliser cette mécanique pour nos threads utilisateur.

3.1 Thread Nachos

Les threads NachOS sont créés et initialisés dans la méthode *Thread()* dans le fichier *thread.cc*.

```
Thread::Thread (const char *threadName){
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;

#ifdef USER_PROGRAM
#ifdef CHANGED
    dependance=-1;
#endif //CHANGED

    space = NULL;

    for (int r=NumGPRegs; r<NumTotalRegs; r++)
      userRegisters[r] = 0;
#endif
}
```

La pile est initialisée. On positionne un entier "dépendance" à -1 (aucune dépendance vers un autre thread). Cette variable ajoutée nous sera utile par la suite, pour ajouter une dépendance du thread courant vers un autre thread. Ensuite, l'espace mémoire du thread, (code exécuté par le thread) est initialisé à NULL. Enfin, les registres sont initialisés. À ce niveau, le thread a le statut *JUST_CREATED*, il n'est pas encore prêt à être lancé.

Pour rendre un thread exécutable, un appel à la méthode *Fork* doit être effectué. Cette méthode prend en paramètre un pointeur vers le programme à charger en mémoire, et les paramètres de la fonction (pointeur vers une structure contenant les arguments). Le thread "forké" partage le même espace mémoire que les autres thread du processus. Son statut est mis à jour (*READY*). Enfin, il est placé dans la ReadyQueue , le thread a un programme à exécuter.

Lors de la mise en place de l'espace d'adressage, le code qui sera exécuté par le thread est placé dans un objet *NoffHeader*.

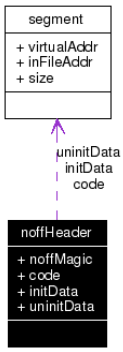

FIGURE 3 – Structure NoffHeader

NoffMagic est un entier identifiant l'objet contenant le code à éxecuter comme étant de type NachOS. La documentation nous apprend par ailleurs que le format d'object NachOS est une simplification du format d'oject UNIX. Cet objet contient les segments code (code exécutable) et initdata (données initiales), entre autres. Ces données sont contenues dans un espace d'adressage propre au thread dont la taille est de 4*1024 bits (StackSize défini dans *thread.h*). Par ailleurs, en lisant les commentaires, nous apprenons que nous allouons, dans cette simulation, un même espace (thread's private execution stack), indépendament du code exécuté par le thread. StackSize doit donc être de taille suffisante, pour nous éviter des erreurs diverses...

Le système divise cet espace en pages, puis il effectue la correspondance entre les adresses physiques et les adresses virtuelles. En observant le code fourni dans *addrspace.cc*, nous pouvons comprendre le fonctionnement des fonctions *saveUserState* et *restoreUserState*. La fonction *saveUserState* sauvegarde les registres qui sont utilisés par le thread. Ces données sont sauvegardées dans le tableau userRegisters, propre à chaque thread. La fonction *restoreUserState* effectue l'opération inverse, en restaurant les registres du thread. Les données sont ensuite chargé en mémoire, en utilisant la table des pages créée précédement. Ces dernières étapes sont réalisées dans le constructeur d'AddrSpace.

À ce niveau, les pages virtuelles sont des pages physiques, l'adressage virtuel sera réalisé dans l'étape 4.

3.2 Thread Utilisateur

Cette mécanique autour des threads est utilisée afin de supporter chaque thread utilisateur par un thread NachOS. Nous étudions maintenant *StartProcess(..) , progtest.cc* . Après avoir chargé le programme à exécuter en mémoire, les registres sont initialisés(*PCReg, NextPCReg, StackReg*, et les autres registres de NachOS) puis le programme est lancé, grâce à la méthode *Run()*. Une fois le programme lancé, chaque thread sera "lancé" par un appel à la fonction *Fork*. Cette dernière fait appel à *StackAllocate*, qui se chargera de l'allocation dans la pile pour le thread à créer. Les paramètres donnés à la fonction *Fork* (et donc *StackAllocate*) seront explicités par la suite.

3.2.1 Syscall UserThreadCreate

On souhaite maintenant qu'un programme utilisateur puisse créer des threads au niveau utilisateur, c'est-à-dire effectuer un appel système *int UserThreadCreate(void f(void *arg), void *arg)*. Nous réutilisons donc les mécaniques de l'étape précédante, pour mettre en place le syscall voulu.

Lors de l'appel système UserThreadCreate, nous appelons *do_UserThreadCreate*. Cette première fonction de manipulation des threads utilisateurs est définie dans *userthread.cc*. Pour l'instant, nous nous intéréssons uniquement aux opérations liéé à la création d'un thread utilisateur. Certaines mécaniques ont du être ajouté pour assurer la cohérence de la page table et le bon fonctionnement d'une dépendance entre deux threads.

```
int do_UserThreadCreate(int f, int arg){

if(!currentThread->space->CheckFreeStack()){
  printf("Error: Stack already full\n");
  return -1;
}

Thread *t = new Thread("UserThread");

if(t==NULL){
  printf("Error: Thread non created\n");
  return -1;
}

// En cas de user_join, mutex autour de son init
//  -> eviter une dep vers un thread pas encore init.
CheckThreadExistence->P();

argThread *argt = new argThread;
argt->func = f;
argt->argv = arg;

t->Fork(StartUserThread,(int)argt);

// Afin que l'alloc dans la stack soit secure
await->P();

// Prend le sem, si un autre t a une dep sur lui -> en attente sur sem.
currentThread->space->TabSemJoin[t->initStackReg]->P();

// set id thread into result reg, useful for join fct...
machine->WriteRegister(2, t->initStackReg);

CheckThreadExistence->V();

currentThread->Yield();

return 0;
}
```

Le thread user créé, nous devons le lancer. Cette opération est réalisée par la fonction *StartUserThread*. Le prototype de la fonction nous impose un unique paramètre. Nous devons passer les paramètres suivant : le poiteur vers la fonction et un pointeur vers les paramètres de la fonction. Nous avons créé une structure regroupant ces paramètres. Nous passerons un pointeur vers cette structure en paramètre à *StartUserThread*.

```
typedef struct{
                int func;
                int argv;
        }argThread;
```

Toute les instructions exécutées après le Fork sont réalisées par le "père" et ont pour but d'assurer le bon fonctionnement d'une dépendance vers le thread créé par le Fork.

3.2.2 Lancement du thread

```
 static void StartUserThread(int f){
argThread *argt = (argThread *) f;

//Save old registers
currentThread->space->SaveState();
//Clean registers
currentThread->space->InitRegisters();

// On place PC sur notre fonction
machine->WriteRegister (PCReg,argt->func);
machine->WriteRegister (NextPCReg,(argt->func)+4);
// argument de notre fct
machine->WriteRegister (4,argt->argv);
// init page , retourne id ds bitmap
int alloc = currentThread->space->AllocStack();
// maj sommet pile
machine->WriteRegister (StackReg,currentThread->space->StackValue(alloc));
//save bitmap page id thread (necesaire pour liberer l'espace ensuite )
currentThread->initStackReg=alloc;

// Alloc dans la stack terminee
//(  important pour que join fonctionne bien)
// on controle pas quand l'OS commute...
await->V();

machine->Run();
}
```

Voici comment est réalisé l'allocation de la mémoire dans la pile pour notre thread :

```
int
AddrSpace::AllocStack ()
{
        SemThread->P();
        //On veut allouer 4 pages pour chaque thread
        if((stack->NumClear())<=(NbPagesThread-1)){
                printf("Stack overflow\n");
                return -1;
        }
        int tmp = stack->Find();
        for(int i=1;i<NbPagesThread;i++){
                if(stack->Test(tmp+i)){
                        printf("Pages block(%d) is not available\n",
                            NbPagesThread);
                        return -1;
                }
                stack->Mark(tmp+i);
        }
```

```
        nbThreads++;
        SemThread->V();
        return tmp;
}
```

On recherche la première page libre, on la marque comme “occupée”. Un compteur du nombre de thread est incrémenté. Ce dernier empeche le syscall Halt d’arrêter NachOS tant qu’un thread utilisateur existe (et n’a pas terminé).

3.2.3 Syscall UserThreadExit

Le thread utilisateur lancé, nous devons maintenant nous intéresser à son arrêt. Un nouvel appel système *UserThreadExit* est créé, ce dernier fait appel à la fonction suivante :

```
int do_UserThreadExit(){

// free bitmap page
currentThread->space->FreeStack(currentThread->initStackReg);

currentThread->space->TabSemJoin[currentThread->initStackReg]->V();

if(currentThread->dependance!=-1)
 currentThread->space->TabSemJoin[currentThread->dependance]->V();

currentThread->Finish();

return 0;
}
```

3.2.4 Syscall UserThreadJoin

Enfin, nous réalisons un dernier appel système, UserThreadJoin afin de forcer un thread à attendre la terminaison d'un thread (autre que lui même ou le thread principal). Pour mettre en place ce fonction, nous avons du mettre en place un système de sémaphore dans les précédantes fonctions.

Lors de la création du thread (*do_UserThreadCreate(...)*) , nous effectuons plusieurs manipulations après le Fork. Nous prenons d'abord un sémaphore *await*. Il sera libéré à la fin de *StartUserThread* juste avant l'appel à *Run()*. Ce sémaphore assure la cohérence de la bitmap. En effet, si un autre thread veut effectuer un join vers un thread créé plus tôt, ce dernier doit avoir terminé son allocation dans la pile. Si le thread principal veut créer un autre fils, il devra donc attendre que l'allocation du premier soit terminé. Nous prenons ensuite un second sémaphore *TabSemJoin[t->initStackReg]*. Si un thread X fait un join vers ce thread, il devra attendre que le semaphore soit libéré par notre thread lors de l'appel à do_UserThreadExit() pour prendre le sémaphore (on prend ce semaphore dans UserThreadJoin(..)). Ainsi, on force l'ordre d'exécution (la dépendance voulue !) , en forcant une attente du thread X sur le semaphore corespondant au thread dont on veut dépendre.

Lors de la terminaison du thread *do_UserThreadExit()*, le thread libère le sémaphore associé à son thread. Ainsi, il débloque d'éventuels thread en attente sur ce sémaphore, des thread qui avaient donc une dépendance vers lui. Si il avait lui même une dépendance, et qu'il est dans cette méthode, alors il doit libérer le sémaphore vers cette dépendance, permettant donc à d'autres threads en attente sur cette dernière de s'exécuter. Par cette mécanique de sémaphore, nous nous assurons d'avoir autant de P() que de V() sur les différents sémaphores, nous évitons donc tout deadlock.

```
 int UserThreadJoin(int t){
if(currentThread->dependance!=-1){
  printf("Le␣thread␣possede␣deja␣une␣dependance\n");
  return -1;
}
if(currentThread->initStackReg==t || t==0){
  printf("Tentative␣de␣dependance␣vers␣un␣thread␣invalide\n");
  return -1;
}
CheckThreadExistence->P();

if(!currentThread->space->Test(t)){
  printf("Tentative␣de␣dependance␣vers␣un␣thread␣non␣existant\n");
  CheckThreadExistence->V();
  return -1;
}

CheckThreadExistence->V();

currentThread->dependance=t;
currentThread->space->TabSemJoin[t]->P();
return 0;
}
```

3.3 Test du multithreading utilisateur

```
 #include "syscall.h"

void thread(int *i){
        if(*i!=-1){
                UserThreadJoin(*i);
                SynchPutString("Thread␣");
                SynchPutInt(*i);
                PutChar('\n');
        }
        else{
                SynchPutString("Thread␣initial\n");
                int a=1001;
                int j;
                //Calcul sale pour "ralentir" le premier thread
                for(j=0;j<1000;j++){
                        if(a%2){
                                a=a*2;
                        }
                        else{
                                a=a/2;
                        }
                }
        }
        UserThreadExit();
}

int main(){
        int param=-1;
        int t1 = UserThreadCreate((void (*)(void *))thread,(void *)(&param));
        int t2 = UserThreadCreate((void (*)(void *))thread,(void *)(&t1));
        UserThreadJoin(t2);
        SynchPutString("Main␣program␣terminated\n");
        Halt();
}

// Trace d'execution
./nachos-step2 -x makemultithreads
Thread initial
Thread 1
Main program terminated
Machine halting!
```

3.4 Déroulement d'une exécution avec dépendance entre 2 threads

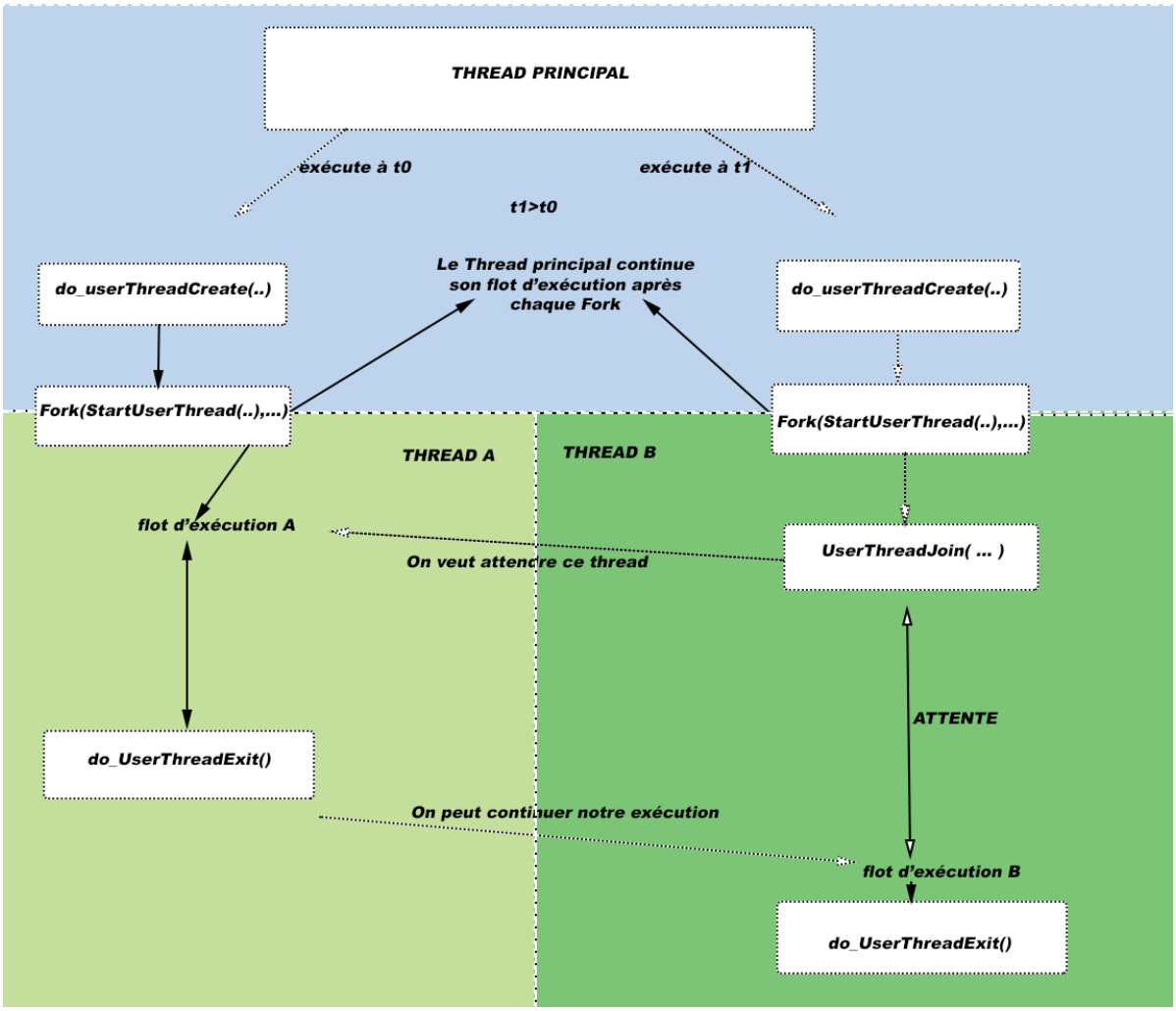

FIGURE 4 – Déroulemet de l'exécution

On peut par ailleurs noter que cette exécution est englobé dans un thread principal, comme l'a montré notre trace d'exécution.

4 Étape 4 : NachOS et la pagination

4.1 Adressage Virtuel par une table des pages

Initialement, un processus avait accès à la mémoire physique via la fonction *ReadAt*. Nous adressions donc directement la mémoire physique. Nous voulons maintenant accéder à la mémoire au travers d'un mécanisme de translation d'une adresse virtuelle vers une adresse physique. La translation vers l'adresse physique est réalisée par l'OS, les programmes utilisateurs fonctionnent quant à eux uniquement avec les adresses virtuelles. NachOS devra donc assurer la traduction des adresses virtuelles, si nous voulons pouvoir utiliser un mécanisme de pagination.

Pour rappel, dans un OS "complexe", une adresse virtuelle se compose d'une série d'offset qui indiqueront l'index à lire dans le Page Directory, le ou les Page Table, et un dernier offset qui indiquera quand commencent nos données dans la Frame (bloc mémoire).

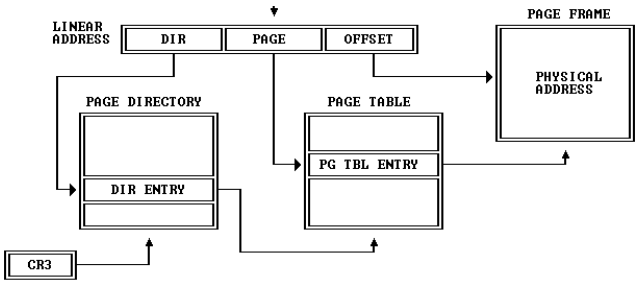

FIGURE 5 – Mécanisme d'adressage

source : http ://www.scs.stanford.edu/05au-cs240c/lab/i386/s05_03.htm

Dans NachOS, et pour l'instant, ce mécanisme est simplifié, nos adresses virtuelles indiquent seulement un numéro de page, et le déplacement dans la Frame. Nous avons donc implémentée la fonction *ReadAtVirtual*, qui à la différence de *ReadAt*, écrit dans l'espace d'adressage virtuel :

```
static void ReadAtVirtual(OpenFile *executable, int virtualaddr, int numBytes,
    int position, TranslationEntry *pageTable,unsigned numPages){
        if ((numBytes <= 0) ||  (virtualaddr < 0) || ((unsigned)virtualaddr >
            numPages*PageSize)){
                printf("Erreur␣ReadAtVirtual\n");
                return;
        }
        int numPages_old = machine->pageTableSize;
        TranslationEntry *page_old = machine->pageTable;
        char *into = new char[numBytes+(numBytes%4)];
        executable->ReadAt(into,numBytes,position);
        int i;

        machine->pageTable = pageTable ;
        machine->pageTableSize = numPages ;

        for(i=0;i<numBytes;i+=4){
                machine->WriteMem((int)(virtualaddr+i), 4, (int)(*(int *)(into+i
                    )));
        }

        machine->pageTable = page_old ;
        machine->pageTableSize = numPages_old ;

        delete [] into;
}
```

Nous vérifions dans un premier temps que l'adresse à traduire est "valide" : positive et inférieure à la valeur maximale adressée par la table. On remarque que l'on écrit en mémoire par bloc de 4 octets

(un word , 32 bits) et l'utilisation d'un buffer (into). On note par ailleurs une sauvegarde puis une restauration des informations de la page table.

On remplace ensuite les appels à ReadAt par ReadAtVirtual, dans le constructeur d'AddrSpace :

```
[...]
    if (noffH.code.size > 0)
      {
          DEBUG ('a', "Initializing␣code␣segment,␣at␣0x%x,␣size␣%d\n",
                 noffH.code.virtualAddr, noffH.code.size);
        #ifndef CHANGED
          executable->ReadAt (&(machine->mainMemory[noffH.code.virtualAddr]),
                              noffH.code.size, noffH.code.inFileAddr);
        #else
        ReadAtVirtual(executable, noffH.code.virtualAddr, noffH.code.size, noffH
            .code.inFileAddr, pageTable, numPages);
        #endif //CHANGED
      }
    if (noffH.initData.size > 0)
      {
          DEBUG ('a', "Initializing␣data␣segment,␣at␣0x%x,␣size␣%d\n",
                 noffH.initData.virtualAddr, noffH.initData.size);
        #ifndef CHANGED
          executable->ReadAt (&
                              (machine->mainMemory
                               [noffH.initData.virtualAddr]),
                              noffH.initData.size, noffH.initData.inFileAddr);
        #else
        ReadAtVirtual(executable, noffH.initData.virtualAddr, noffH.initData.
            size, noffH.initData.inFileAddr, pageTable, numPages);
        #endif //CHANGED
      }
```

En modifiant la page table pour que la page i soit mappé sur la page physique i+1, voici ce que nous observons au debugger :

```
  Avec l'adresse physique placee a i:

Reading VA 0x14, size 4
Translate 0x14, read: phys addr = 0x14 value read = 0000000c

Avec l'adresse physique placee a i+1:

Reading VA 0x14, size 4
Translate 0x14, read: phys addr = 0x94 value read = 0000000c
```

Conclusion : l'adresse physique est donc décalée de 0x80. Cela correspond en base 10 à 128, soit la taille d'une page.

4.2 Mise en place d'un Frame Provider

Plus généralement, il est utile d'encapsuler l'allocation des pages physiques aux pages virtuelles dans une classe spéciale FrameProvider. C'est alors à cette classe de remettre à zéro les pages fournies, et non plus au constructeur de AddrSpace. Nous allons donc nous atteler à la mise en place de cette classe. Cette dernière utilisera une bitmap pour savoir quelle frames sont disponibles. Elle se chargera de les initialiser, puis de les libérer le moment venu.

```
#include "frameprovider.h"
#include "synch.h"
#include "sysdep.h"

static Semaphore *semFrame = new Semaphore("semFrame",1);

 FrameProvider::FrameProvider(int NumPages)
{
        phyMemBitmap = new BitMap(NumPages);
        BitmapSize = NumPages;
}

FrameProvider::~FrameProvider(){
        delete phyMemBitmap;
}

int
FrameProvider::GetEmptyFrame()
{
        semFrame->P();
        if((phyMemBitmap->NumClear())<=0){
                printf("Stack overflow\n");
                return -1;
        }

        //Version non aleatoire - retourne le 1er
        int tmp = phyMemBitmap->Find();
        semFrame->V();
        return tmp;
}

void
FrameProvider::ReleaseFrame(int numFrame)
{
        if(!phyMemBitmap->Test(numFrame)){
                printf("Error numFrame %d\n",numFrame);
                return;
        }
        // liberation frame
        phyMemBitmap->Clear(numFrame);
}

int
FrameProvider::NumAvailFrame()
{
        // maj nb frame dispo
        return phyMemBitmap->NumClear();
}
```

La classe AddrSpace s'en retrouve donc modifiée, puisque ce n'est plus à elle de gérer l'allocation des Frames.

```
 pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++)
      {
          pageTable[i].virtualPage = i;
        #ifndef CHANGED
          pageTable[i].physicalPage = i;// virtual page # = phys page #
        #else
          pageTable[i].physicalPage = frameProvider->GetEmptyFrame();
        #endif //CHANGED
          pageTable[i].valid = TRUE;
          pageTable[i].use = FALSE;
          pageTable[i].dirty = FALSE;
          pageTable[i].readOnly = FALSE;
          // if the code segment was entirely on
          // a separate page, we could set its
          // pages to be read-only
      }
```

Le Frame Provider ne doit pas être lié à un processus, mais à l'OS. Aussi, nous l'instancions dans *Initialize (fichier system.cc)*

```
 #ifdef CHANGED
    frameProvider = new FrameProvider(NumPhysPages);
    nbProcess = 0;
#endif //CHANGED
```

Enfin, lors de la terminaison d'un processus et la supression de son espace mémoire associé, nous libérons donc les frames occupées par chaque page :

```
 AddrSpace::~AddrSpace ()
{
        #ifdef CHANGED
        unsigned i;
        for(i=0;i<divRoundUp(UserStackSize,PageSize);i++){
                printf("Thread: %d Sem: %d\n",i,this->TabSemJoin[i]->get());
                delete this->TabSemJoin[i];
        }
        for(i=0;i<numPages;i++){
                frameProvider->ReleaseFrame(pageTable[i].physicalPage);
        }
        delete stack;
        #endif //CHANGED
  delete [] pageTable;
}
```