

# Compte Rendu NACHOS

BARTHELEMY Romain, EUDES Robin, MORISON Jack, ROSSI Ombeline

18 mars 2015

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Étape 2 : Etude d'un syscall</b>	<b>3</b>
2.1	Entrées-sorties asynchrones . . . . .	3
2.2	Entrées-sorties synchrones . . . . .	4
2.3	Création du syscall Puchar . . . . .	6
2.4	La manipulation de chaine de caractères . . . . .	7
2.5	Bilan . . . . .	10
<b>3</b>	<b>Étape 3 : Multithreading</b>	<b>11</b>
3.1	Thread Nachos . . . . .	11
3.2	Thread Utilisateur . . . . .	13
3.2.1	Syscall UserThreadCreate . . . . .	13
3.2.2	Lancement du thread . . . . .	14
3.2.3	Syscall UserThreadExit . . . . .	15
3.2.4	Syscall UserThreadJoin . . . . .	15
3.3	Tests . . . . .	16

# 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.

## 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 libéreront 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 *SemPutChar;
static Semaphore *SemGetChar;
static Semaphore *SemPutString;
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);

    SemPutChar = new Semaphore("PutChar", 1);
    SemGetChar = new Semaphore("GetChar", 1);
    SemPutString = new Semaphore("PutString", 1);
    SemGetString = new Semaphore("GetString", 1);
}

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

void SynchConsole::SynchPutChar(const char ch){
    SemPutChar->P();
    console->PutChar (ch);
    writeDone->P ();
    SemPutChar->V();
}

char SynchConsole::SynchGetChar(){
    SemGetChar->P();
    char ch;
    readAvail->P ();
    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'exécution 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 chaîne 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*, elle seront appelé par le handler associé.

```
void SynchConsole::SynchPutString(const char s[]){
    SemPutString->P();
    int i;
    for (i=0;i<MAX_STRING_SIZE && s[i]!='\0';i++){
        if (s[i]=='\0')
            return;
        synchconsole->SynchPutChar ((char)s[i]);
    }
    SemPutString->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 Bilan

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

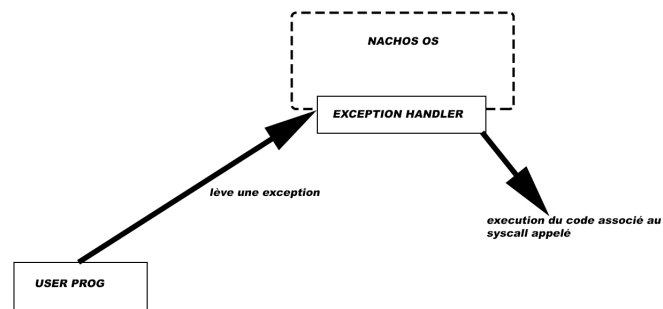


FIGURE 1 – Mécanisme simplifiée d'un syscall

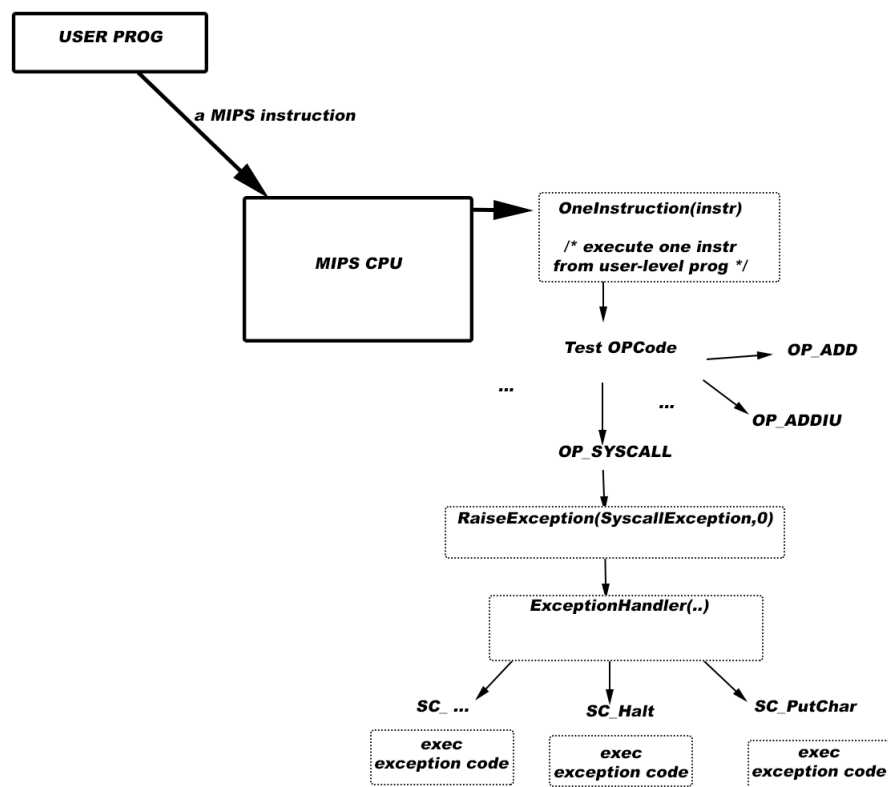


FIGURE 2 – Mécanisme détaillée d'un syscall

On retrouve par ailleurs les mécaniques d'interruptions 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 intéressons désormais aux threads. Nous souhaitons au terme de cette étape pouvoir exécuter 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 pour l'instant 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 (statut = *READY*).

Pour rendre ce 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). L'espace mémoire du thread pointe vers celui du thread courant, son statut est mis à jour ( *READY* ). Enfin, il est placé dans la ReadyQueue.

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*.

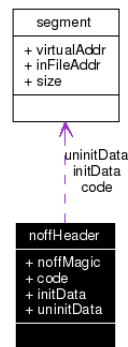


FIGURE 3 – Structure NoffHeader

NoffMagic est un entier identifiant l'objet contenant le code à exécuter comme étant de type NachOS. La documentation nous apprend par ailleurs que le format d'objet NachOS est une simplification du format d'objet UNIX.

Cet objet contient les segments code ( code exécutable ) et initdata (données initiales), ainsi qu'un segment "uninitdata". Ces données sont contenues dans un espace d'adressage dont la taille maximale est de 4\*1024 bits (StackSize défini dans *thread.h*).

Le système divise cet espace en pages, puis il effectue la correspondance des adresses physiques en adresses virtuelles.

En regardant le fichier *userprog/addrspace.cc*, nous pouvons comprendre le fonctionnement des fonctions *saveUserState* et *restoreUserState*. La fonction *saveUserState* sauvegarde les registres qui sont utilisés par le thread courant. 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 courant.

Les données sont ensuite chargées en mémoire, en utilisant la table des pages. Ces dernières étapes sont réalisées dans le constructeur d'AddrSpace (fichier *addrspace.cc*).

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

Les registres sont initialisés(PCReg, NextPCReg, StackReg, et les autres registres de NachOS). Enfin, le programme est lancé, grâce à la méthode *Run()*.

## 3.2 Thread Utilisateur

### 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 les mêmes mécaniques que précédemment.

Lors de l'appel système UserThreadCreate, nous appelons la fonction `do_UserThreadCreate`. Cette première fonction de manipulation des threads utilisateurs est dans le fichier *userthread.cc*.

```
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();

// struct with current f (first instruction)
// and args ( la suite) .
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();

// Si un thread X fait un join vers notre thread, il devra attendre
// que le semaphore soit libere par notre thread lors de l'appel a
// do_UserThreadExit()
// pour etre debloque : on prend ce semaphore ds UserThreadJoin(int <thread
// cible> ).
// Ainsi, on force l'ordre d'exec, en forçant une attente du thread X sur ce
// semaphore.

currentThread->space->TabSemJoin[t->initStackReg]->P();

// set id thread
machine->WriteRegister(2, t->initStackReg);

// On a termine d'init notre thread
CheckThreadExistence->V();

// Go to readyqueue
currentThread->Yield();

return 0;
}
```

Note : la fonction a été modifiée pour prendre en compte le multithreading au niveau user, d'où la présence de mutex "CheckThreadExistence", pour assurer la cohérence de la page table (bitmap) en cas de commutation de contexte.

Une fois le thread user créé, nous devons le lancer. Cette opération est réalisée par la fonction Star-

tUserThread. 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;
```

### 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 bitmap , retourne id ds bitmap
int alloc = currentThread->space->AllocStack();
// maj sommet pile
machine->WriteRegister (StackReg, currentThread->space->StackValue(alloc));
//save bitmap 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();
}
```

### 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 (*userthread.cc*) :

```
int do_UserThreadExit(){  
  
    // free bitmap  
    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).

```
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 Tests

```
#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();
}

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