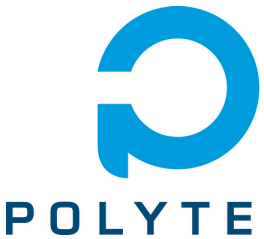

Polytech' Grenoble

RICM 4ème année

NachOS
Etape 4: Mémoire virtuelle

Étudiants:
Elizabeth Paz
Salem Harrache

Enseignant:
Vania Marangozova

Mars 2012

1 Git

Pour voir les différences entre l'étape 3 et l'étape 4 vous pouvez lancer un diff avec le tag step3 :

```
git diff step3
```

ou alors directement avec le commit 78799c....

```
git diff 78799ce7a7b788d0f3b501f0d85bab2f21c7190b
```

2 Test de l'étape 4

Le premier test consite à créer plusieurs processus simples (putstring). Dans le second, on fork trois processus, qui lancent chacuns deux threads qui vont écrire leurs noms trois fois.

```
$ ./runtest.sh
Compilation en cours...
Lancement du Test 1 : ForkExec PutString
_____________________________________
./build-origin/nachos-userprog -rs 1 -x ./build/forkprocess
Debut du pere
Commit d./build/putstring : Insufficient memory to start the process.
u./build/putstring : Insufficient memory to start the process.
./build/putstring : Insufficient memory to start the process.
 soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Commit du soir, espoir. Build du matin, chagrin
Fin du pere
Machine halting!

Ticks: total 83124, idle 58420, system 24120, user 584
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 605
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...


Lancement du Test 2 : ForkExec With Thread
_____________________________________
./build-origin/nachos-userprog -rs 1 -x ./build/forkthreadedprocess
Debut du pere
...Fin du pere
D but du fils 0 : lancement des deux threads a et z
D but du fils 1 : lancement des deux threads b et y
D but du fils 2 : lancement des deux threads c et x
zaybcxzaybcxza
Fin du thread main du fils 0
by
Fin du thread main du fils 1
cx
Fin du thread main du fils 2
Machine halting!

Ticks: total 651494, idle 8500, system 75110, user 567884
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 296
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

3 Lecture dans la mémoire virtuelle

Jusqu'a à présent, le (seul) procesus avec son espace d'adresse à l'adresse zero, lit et ecrit directement dans la mémoire physique au lieu de la mémoire virtuelle.

Pour y remédier, il faut que le processus, au moment de la lecture, fait un changement de contexte et utilise sa table de page pour charger l'executable dans sa mémoire virtuelle. C'est le role de la fonction **ReadAtVirtual**

Listing 1: *code/userprog/addrspace.cc*

```
void ReadAtVirtual( OpenFile *executable, int virtualaddr, int numBytes,
                    int position, TranslationEntry *pageTable,
                    unsigned int numPages ) {
    /* Ecriture dans la memoire virtuelle
    * On commence par initialise les table de pages dans la machine
    * Ensuite on lit a partir de memoire physique pour recopier octet
    * par octet dans la memoire virtuelle (avec un buffer par s curit )
    */
    TranslationEntry * old_pageTable = machine->pageTable;
    unsigned int old_numPages = machine->pageTableSize;
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
    //buffer to read the specified portion of executable
    char buffer[numBytes];
    //char * buffer = new char[numBytes];
    // On lit au plus numBytes octets
    int nb_read = executable->ReadAt(buffer, numBytes, position);
    // On ecrit dans la m moire virtuelle
    for (int i = 0; i < nb_read; i++)
        machine->WriteMem(virtualaddr+i, 1, buffer[i]);
    // On restore le context
    machine->pageTable = old_pageTable;
    machine->pageTableSize = old_numPages;
}
```

"'

4 Allocation des cadres de pages

Nous avons légèrement modifié la classe *frameprovider*. En effet, celle-ci alloue un certain nombre de cadres de façon atomique. Un processus à besoin de N cadres ou ne lance pas.

Listing 2: *code/userprog/frameprovider.cc*

```
int * FrameProvider::GetEmptyFrames(int n) {
    RandomInit(0);
    this->semFrameBitMap->P();
    int * frames = NULL;
    if (n <= this->bitmap->NumClear()) {
        frames = new int[n];
        for(int i=0; i<n; i++) {
            int frame = Random()%NumPhysPages;
            // Recherche d'une page libre
            while(this->bitmap->Test(frame)) {
                frame = Random()%NumPhysPages;
            }
            this->bitmap->Mark(frame);
            bzero(&(machine->mainMemory[ PageSize * frame ]), PageSize );
            frames[i] = frame;
        }
    }
    this->semFrameBitMap->V();
    return frames;
}
```

L'utilisation dans **AddrSpace** est la suivante :

Listing 3: *code/userprog/addrspace.cc*

```
    int * frames = frameprovider->GetEmptyFrames((int) numPages);
    if (frames == NULL) {
        DEBUG ('p', "Pas suffisamment de memoire !\n");
        this->AvailFrames = false;
```

```
        return;
    } else {
        this->AvailFrames = true;
    }
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;
        // for now, virtual page # = phys page #
        pageTable[i].physicalPage = frames[i];
    [...]

    delete frames;
```

l'instruction **return** dans le constructeur avorte la construction de l'objet AddrSpace sans pour autant le détruire. Il faut donc utiliser une variable d'état (*AvailFrames*) pour s'assurer que la mémoire a correctement été alouée. On s'assure également que le destructeur libère les cadres un par un...

5 Création d'un nouveau processus

La création d'un nouveau processus se déroule en plusieurs étapes :

- Création d'un nouvel espace d'adressage
- Creation d'un nouveau Thread main auquel on associe cet espace d'adressage
- Appel de Fork de ce nouveau thread.

Listing 4: *code/userprog/forkprocess.cc*

```
void StartForkedProcess(int arg) {
    currentThread->space->RestoreState();
    currentThread->space->InitRegisters ();
    currentThread->space->InitMainThread();
    machine->Run();
}

int do_ForkExec (char *filename)
{
    OpenFile *executable = fileSystem->Open (filename);
    AddrSpace *space;

    if (executable == NULL) {
        printf ("Unable to open file %s\n", filename);
        delete [] filename;
        return -1;
    }
    // Creation d'un nouvel espace d'adressage
    space = new AddrSpace (executable);

    // Si c'est null ou qu'il n'y a pas assez de memoire on arrete la
    if (space == NULL || !space->AvailFrames) {
        printf("%s : Insufficient memory to start the process.\n",
                    filename);
        delete executable;
        delete [] filename;
        return -1;
    }
    delete executable;

    // Creation du nouveau thread main du nouveau processus
    Thread * mainThread = new Thread(filename);
    mainThread->space = space;
    machine->UpdateRunningProcess(1); // appel atomique
    mainThread->Fork (StartForkedProcess, 0);

    return 0;
}
```

L'appel Fork est légèrement modifié. En effet, par défaut Fork lance le thread dans l'espace d'adressage du thread appellant (*currentThread*), or pour la création de nouveau processus, il faut lancer la thread main du nouveau processus dans un nouvel espace d'adressage (qu'on associe au thread avant d'appeler Fork)

Listing 5: *code/threads/thread.cc*

```
void
Thread::Fork (VoidFunctionPtr func, int arg)
{
    [...]
    if (this->space == NULL) {
        this->space = currentThread->space;
    }
    [...]
}
```

6 Terminaison

Pour gérer correctement la terminaison, nous utilons un compteur de processus (dans machine) qui permet à n'importe quel processus de savoir s'il est le seul à s'executer sur la machine lors de l'appel à **Exit())**, auquel cas, on éteint la machine. Si non on termine simplement le processus. De même pour les thread utilisateur, soit il se terminent si d'autres sont en train de s'éxecuter, soit ils appellent **do_Exit()** qui termine le processus.

Listing 6: *code/userprog/forkprocess.cc*

```
void do_Exit() {
    DEBUG('p', "ExitProcess : %s", currentThread->getName());
    machine->UpdateRunningProcess(-1);
    if (machine->Alone()) {
        interrupt->Halt();
    }

    // /!\ Provoque une erreur
    // delete currentThread->space;
    // On libere les frames c'est la fin du processus
    currentThread->space->ReleaseFrames();
    currentThread->Finish();
}
```

Nous n'avons par contre par trouvé comment Nachos procède à la destruction d'un espace d'adressage. En effet on ne peut détruire celui ci dans **do_Exit()** puisque le thread est cours d'execution (même en train de terminé...). Autrement dit un processus ne peut pas se terminer lui même, mais un autre doit le faire. Nous nous contentons donc de libérer les frames pour le moment.