NachOS : La pagination
Année 2013-2014

par Jerôme Barbier, Augustin Husson et David Levayer

7 avril 2014

Rapport généré avec LaTeX

Table des matières

1 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).

```
/*rcuprer un cadre libre et initialis  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();
```

2 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 sytme
           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*

2.1 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 srialisation
        currentThread->space = restor->space; // on affecte le nouvel espace mmoir  notre nouveau
            processus
        currentThread->space->InitRegisters (); // on rinitialise 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); // cration 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 mmoir

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

        newThread->Fork(StartProcess,(int)save); // on fork le processus pre
        machine->SetNbProcess(machine->GetNbProcess()+1); // on incrmente de 1 le nbre de processus
             cr
        delete executable;
        currentThread->Yield(); // le processus pre 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*.

2.2 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
      bloquer
    currentThread->space->LockEndMain();

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