NachOS : Multithreading
Année 2013-2014

par Jerôme Barbier, Augustin Husson et David Levayer

31 mars 2014

Rapport généré avec LaTeX

Table des matières

1 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.

1.1 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();
```

1.2 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.

2 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.

2.1 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.

2.2 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 ();
}
```

2.3 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());
```

La partie Bonus sera peut être faite.