J1INAW11: NachOS project

Due on Monday, November 9, 2015

NAMYST Raymond

VER VALEM Willian , RAKOTONIERA Hoby

Contents

Introduction

Pour cette partie du projet, il était demandé de mettre en place un niveau utilisateur Multithreading avec NachOS.
Pour réaliser ce travail, on a une suivi une série d'étapes fournies par l'enseignant, en commençant tout d'abord avec la possibilité de créer un seul thread d'une façon rudimentaire, et en l'améliorant ensuite, jusqu'à arriver à la possibilité de faire fonctionner un nombre indéterminé de threads, et synchroniser la façon dont ils se terminent.
L'objectif était de comprendre comment les threads sont créés, gérés, chargés et terminés, c'est à dire le cycle de vie complet d'un thread; et ensuite de mettre en oeuvre ce cycle de vie dans NachOS.

Multithreading

Multithreading dans les programmes utilisateurs

ThreadCreat

La mise en oeuvre de l'appel système "*ThreadCreat*" a été réalisée en créant une fonction qui a la responsabilité de mémoriser le "pointeur" de la fonction et le pointeur des arguments, de créer un thread et de le démarrer en utilisant *StartUserThread* et en lui passant les arguments.

ThreadExit

Dans la première partie du projet, *ThreadExit* avait à effectuer une tâche simple, qui était d'appeler *cutrrentThread->Finish()*.

StartUserThread

StartUserThread était le point clé de la création d'un thread, puisqu'il est responsable de charger le thread en mémoire et de le faire tourner. Pour réaliser cela, une nouvelle fonction similaire à *InitRegisters* a été mise en oeuvre dans la classe *addrespace*, la fonction *InitUserRegisters* qui charge les registres et pointe vers eux (au sommet de la pile) - 256.
Voici le code snippet de la fonction :

```
void
AddrSpace::InitUserRegisters (int f,int arg)
{
    printf("inside\n");
    int threadNumber = currentThread->threadNumber;
    //int i;
    //for (i = 0; i < NumTotalRegs; i++)
    //machine->WriteRegister (i, 0);
    // Initial program counter -- must be location of "Start"
    machine->WriteRegister (PCReg, f);
    machine->WriteRegister(4, arg);
    machine->WriteRegister (NextPCReg, machine->ReadRegister(PCReg) + 4);
    machine->WriteRegister (StackReg, numPages * PageSize -( 256 * threadNumber)-16);
    DEBUG ('a', "Initializing USER stack register to 0x%x\n", numPages * PageSize - 256);
}
```

Il utilise une variable globale *threadNumber* pour allouer l'espace approprié sur la pile, et qui, à ce stade du projet, avait pour seule valeur "1".

Plusieurs threads par processus

ThreadCreat

Lorsque nous avons eu besoin de gérer plus d'un thread par processus, il a fallu changer cette fonction, car elle a créé des problèmes. Ces problèmes sont les suivants :

- ThreadCreate avait besoin de répondre s'il était possible de créer le thread
- il ne pouvait pas créer un thread s'il ne savait pas s'il y avait des slots libres
- une fois qu'un thread était créé, il devait occuper ce slot pour éviter que d'autres threads y accèdent.

Pour résoudre ces problèmes, l'approche choisie a été d'accéder au BitMap du processus, et de donner le nombre de threads avec cette fonction, et de cette manière, une fois qu'un thread est créé, il a déjà son numéro et le slot sur lequel il sera chargé ; cela évite ainsi que le thread principal se termine avant que le thread soit exécuté.

Comme cette méthode incrémente également le *threadCounter*, une fois qu'un *StartUserThread* est appelé, le thread a déjà son espace affecté, auquel il est alloué. Cette démarche a été réalisée comme suit :

```
int do_ThreadCreate(int f, int arg){
    int* temp = new int[2];
    temp[0] = f;
    temp[1] = arg;
    if(currentThread->space->threadBitMap->NumClear() > 0){
        Thread *t = new Thread ("new thread");
        t->threadNumber = currentThread->space->threadBitMap->Find();
        currentThread->space->threadCounter++;
        t->Start(StartUserThread,temp);
        return 0;
    }else{
        return -1;
    }
}
```

ThreadExit

Comme maintenant il a un nombre indéterminé de threads, ThreadExit doit aussi garder la trace des threads et des slots et pour effectuer ce comportement, il appelle *currentThread->Finish()*, libère le *BitMap* et réduit le *threadCounter*.
De cette manière, NachOS peut garder la trace des threads actifs et réutiliser un espace pour allouer à de nouveaux threads et dans la classe *exception*, le *sysxall SC_Exit* et le *SC_ThreadExit* ont été modifiés et font tous les deux la même chose.

```
    if(currentThread->space->threadCounter == 1){
        interrupt->Halt ();
    }else{
        do_ThreadExit();
    }
    break;
```

Ainsi, avec cette méthode, ce n'est que le dernier thread qui est capable d'appeler le *interrupt->Halt()*.

SynchConsole and Lock

Pour pouvoir utiliser *SynchConsole* avec les threads, un lock a été mis en place sur la classe *Synch*. Ce qu'il fait, essentiellement, est de créer un *semaphore* initialisé à un et arrêter le system interrupt à *aquire* et redémarrer le system interrupt à *release*
Ainsi, grâce qu fait que j'aie mis ce système en place, j'ai été capable de vérouiller l'appel de *SynchPutString* et *SynchGetString* et de cette manière, un thread est capable d'imprimer et utiliser ces appels sans interruptions, mais lorsque ces fonctions sont appelées par le syscall à *exception* il peut être interrompu par le système à la fin du buffer. Le meilleur moyen d'exécuter un résultat à ce niveau est de verrouiller cette fonction au niveau du syscall, à *exception*. A ce moment là, il n'est pas mis en oeuvre, donc quand des chaînes longues (plus longues que le buffer) dépendant de *-rs* passés à NachOS, la chaîne peut être coupée au milieu.

tests

Pour tester les threads, j'ai utilisé un simple bout de code, comme suit :

```
void printString(char* c, int x)
{
    c[6]=48 + counter++;
    PutString(c);
    ThreadExit();
}

int main()
{

    char* stt = "\n\n\nth=0\n\n\n";
    char* mai = "\n\n\nmain\n\n\n";
    int x;
    for(x=0;x<4;x++){
        ThreadCreate(printString,stt);
    }
    PutString(mai);
    for(x=0;x<100;x++){}
    for(x=0;x<4;x++){
        ThreadCreate(printString,stt);
    }
    return 0;
}
```

De cette façon, j'ai été capable de tester où le programme crée les threads, quand il les exécute et les libère correctement. Le problème que j'ai eu à résoudre dépend du *-rs* passé à NachOS. Le second cycle n'est pas exécuté tant que les 4 premiers threads ne sont pas finis, donc ça dépend comment le scheduler priorise l'exécution des threads.