TP1
Premiers pas en Nachos

Polytech'ricm2

Janvier 2006

Résumé

Les objectifs de ces premières séances sont :
- installation de NACHOS
- la découverte du simulateur Nachos à travers la lecture du code source, le tracage d'éxécution grâce aux options de débogage et à gdb. En plus du démarrage du simulateur, on observe l'execution d'un programme utilisateur qui donnera un exemple d'appel systeme, puis la commutation et l'ordonnancement des threads noyaux.

Le travail se fait sur la machine **mandelbrot**.

1 Installation de NACHOS

L'installation de Nachos consiste à installer le répertoire nachos dans son environnement de travail. Cette installation peut se faire directement depuis un tar compressé l'arborescence nachos ou indirectement depuis une base CVS.

1.1 Choisir son répertoire de travail

L'environnement de travail est un sous répertoire du répertoire utilisateur défini à votre convenance. Il sera repéré par la variable d'environnement ASWD. Le répartoire Nachos le sera par la variable NOS. Pour cela, rajouter à votre fichier .bashrc les lignes suivantes[1] :

```
export ASWD=$HOME/votre chemin d'accès/
export NOS=$ASWD/NachOS-4.1
```

La prise en compte de cette définition sera automatique au prochain login ou au prochain lancement de xterm sinon on la force par

```
source ~/.bashrc
```

1.2 Installation directe

On copie dans son environnement de travail la distribution Nachos. Vous pouvez alors récupérer la distribution et l'installer par

[1]Si vous utiliser tcvsh, il vous faut faire une modification similaire à votre fichier .cshrc

```
cd  $ASWD
cp /opt/NACHOS/NachOS-4.1.tar.gz .
tar xvfz NachOS-4.1.tar.gz
```

Le répertoire NachOS-4.1 est alors installé. Vous pourrez éliminer la copie de l'archive.

1.3 Installation depuis CVS

CVS est un outil de gestion des différentes versions d'un programme qui est modifié fréquemment par un ou plusieurs utilisateurs. Il permet de détecter des conflits en cas de modifications concurrentes et de revenir en arrière dans les versions successives d'un programme.

Installer la base CVS de Nachos On initialise ASWD et NOS comme précédemment(cf Choisir son répertoire de travail) et on copie dans son environnement de travail la distribution de la base CVS de Nachos

```
cp /opt/NACHOS/NachOS-4.1-cvs.tar.gz ${ASWD}
tar xvfz NachOS-4.1-cvs.tar.gz
```

Le répertoire `$ASWD/CVSNACHOS` est la racine de la base.

Configurer cvs Il vous faut configurer cvs pour l'utilisation de cette base. Pour cela rajouter à votre .bashrc les lignes suivantes

```
export CVSROOT=$ASWD/CVSNACHOS
export CVSUMASK=000
# la commande suivante est optionnelle et permet
# d'utiliser son éditeur favori pour les commentaires
# à attacher aux versions
# par defaut, l'editeur est vi
export CVSEDITOR=votre editeur favori
```

La prise en compte de cette définition sera automatique au prochain login sinon on la force par

```
source ~/.bashrc
```

Installer nachos Il suffit d'extraire la version courante dans le répertoire de travail fixé.

```
cd  ${ASWD}
cvs co NachOS-4.1
```

Utiliser cvs Pour connaitre la version courante , on se place dans Nachos-4.1 :

```
cd ${NOS}
cvs status
```

Vous avez modifier Nachos. Par exemple faites :

```
cd ${NOS}/
cat code/test/halt.c
/* ce commentaire est rajouter à halt.c*/
^D # ceci est un control D
```

verifier avec que la modification de halt.c est vue par cvs (*locally modified*).

```
cd $NOS
# etat de tous les fichiers
cvs status
# etat de halt.c
cvs status | grep halt.c
```

On peut enregistrer la modification dans la base par :

```
cd ${NOS}
cvs commit
```

C'est à ce moment où cvs vous demande de commenter cette nouvelle version qui devient automatiquement la version courante.

Si vous modifier votre version locale par erreur (effacer par exemple le fichier halt.c ou tout autre fichier), vous pouvez remettre à jour par :

```
cd ${NOS}
cvs update
```

Dans le cas où plusieurs manipulations sont en cours, cvs détectera lors du commit les versions incompatibles. On se reportera au manuel cvs pour plus de détails.

2 Génération NACHOS

2.1 Génération Coff2Noff

Il faut générer les outils permettant de passer des fichiers programme MIPS .coff à des fichiers binaires chargeables par le système NachOS

```
cd $NOS/coff2noff
make clean      # Pour se remettre dans un état standard (prudent!)
make            # Pour produire les fichiers de dépendances
                # et lancer la compilation
```

Normalement, tout doit bien se passer...

Ceci n'a besoin d'être fait qu'une fois.

2.2 Génération Nachos

On peut alors générer le simulateur MIPS et le système NachOS incomplet fourni initialement.

Positionnez-vous dans le répertoire de construction build.<votre système> et construisez NachOS :

```
cd $NOS/code/build.linux
make clean      # Pour se remettre dans un état standard (prudent!)
make depend     # Pour produire les fichiers de dépendances
                # chaque fois que le Makefile est changé
make             # et lancer la compilation
```

Normalement, tout doit bien se passer... `make clean ; make` doivent être fait chaque fois qu'un fichier des répertoires **lib , network , threads , filesys et userprog** sera modifié. `make depend` doit être fait chaque fois que le fichier **Makefile** est modifié. Ce sra le cas si vous rajoutez un nouveau répertoire et/ou de nouveaux fichiers

2.3 Exécution Nachos

Vous remarquerez que les fichiers binaires .o sont regroupés dans le répertoire de construction ainsi que l'exécutable **nachos**.

Pour pouvoir appeler nachos de n'importe quel environnement, vous devez rajouter sa localisation à la variable d'environnement PATH. Rajouter au fichier .bashrc :

```
export PATH=$PATH:$NOS/code/build.linux
```

La prise en compte de cette définition sera automatique au prochain login ou au prochain lancement de xterm sinon on la force par

```
source ~/.bashrc
```

2.4 Génération Test

Positionnez-vous dans le répertoire de test et génerer les tests :

```
cd $NOS/code/test
make clean      # Pour se remettre dans un état standard (prudent!)
make            # Pour produire les fichiers de dépendances
                # et lancer la compilation
```

Normalement, tout doit bien se passer...

Il vous suffit maintenant de lancer les tests. Nous reviendrons sur la signification de ces tests plus tard, rassurez-vous. Il s'agit juste de voir si tout est en ordre...

```
nachos
```

pour le premier, et pour le second

```
nachos -x  $NOS/code/test/halt
```

Si les affichages produits par ces deux tests ne comportent rien de suspect, alors tout devrait être valide.

3 Documentation

On peut naviger dans la documentation et dans le code via un navigateur Web. Le document de référence sur Nachos et le code source ont été mis sous une forme html et se trouve dans le répertoire `/opt/NACHOS/achOS-4.1_doc/` qui contient :

– NACHOSroadmap : Le document introductif à NachOS de T. Narten
– MACHINEhtml : Le code indexé du simulateur d'architecture support de NachOS et l'émulateur MIPS
– NACHOShtml : Le code du squelette NachOS initial

Les deux dernières documentation html ont été produites avec **Doxygen**.

4 Lecture du code source de Nachos

4.1 Principes de la simulation

Dans un système réel monoprocesseur, il existe au moins deux modes d'utilisation du processeur :

- le **mode utilisateur** (*user mode*) dans lequel il exécute les instructions d'un programme utilisateur ;
- le **mode système** (*system mode*) dans lequel il organise les différentes tâches systèmes qui lui sont demandées : communication avec les périphériques (disque-dur, clavier, carte graphique ...), gestion des ressources (mémoire (swap, allocation), processeur (ordonnancement des processus)), ...

Dans la réalité le système exécute alternativement le mode système et le mode utilisateur de la même façon sur un processeur.

Dans le cas du simulateur Nachos le fonctionnement est différent : il vous faut bien distinguer ce qui est simulé de ce qui est réellement exécuté. Compilé en assembleur x86, le mode système est exécuté sur le processeur réel (i386 et affidés). En revanche, les programmes de niveau utilisateur sont compilés en assembleur MIPS et émulés par un simulateur de processeur MIPS. Ce simulateur étant un programme comme un autre, il est donc exécuté sur le processeur réel (i386) de la machine sur laquelle vous travaillez.

Le programme nachos est un exécutable comme les autres sur votre machine réelle qui dispose de temps à autre du processeur réel (i386) pour exécuter des instructions en mode système ou bien pour simuler divers éléments matériels comme un processeur MIPS (pour l'exécution des programmes utilisateur en mode utilisateur), une console pour les entrée/sorties clavier ou écran, un disque-dur ...

Pour les distinguer rapidement on appelle *mémoire réelle* la mémoire du processus (espace d'adressage) dans lequel est exécuté Nachos et *mémoire MIPS* la mémoire simulée associée au processeur MIPS simulé. On fera la même chose pour le processeur réel et le processeur MIPS qui est simulé.

Pour rattacher ce discours général à la réalité on se propose de lire dans le code source, l'exécution de la commande

```
cd $NOS/code/test
nachos -x halt
```

Le comportement de cette commande est d'initialiser le système `nachos` et d'exécuter le programme utilisateur MIPS `halt` qui demande l'arrêt du système.

NOTE : Vous pouvez lire le code via la documentation html. Dès qu'il s'agira de le modifier, vous devrez passer via votre éditeur favori. Celui-ci peut avoir une capacité de navigation via le mécanisme de **tag**. A ce sujet, vous pouvez (devez) consulter l'annexe sur l'utilisation des tags. Cela peut vous faciliter grandement la vie.

4.2 Initialisation du système

Comme tout processus, l'exécutable nachos dispose de mémoire réelle (espace d'adressage) subdivisée en zones de code, tas et pile. Le point d'entrée de ce programme est naturellement la fonction `main` du fichier `threads/main.cc`. Le premier objectif de cette partie est d'observer commment un exécutable usuel se "transforme " en un système d'exploitation sur lequel il existe un unique thread noyau. Le second objectif (plus délicat) est de distinguer ce qui correspond à une exécution réelle en mode système de ce qui correspond à une simulation de matériel (le processeur MIPS, le disque dur ...).

Allocation du simulateur Le schéma 1 représente les ressources du processus nachos et les éléments du système qui sont initialisés à la fin de la fonction `Kernel::Initialize()`. Ces éléments sont créés par l'opérateur C++ `new` qui est l'"équivalent" de `malloc` en C.

Indiquez sur le schéma le nom des classes C++ qui codent ces éléments. Précisez dans quelle mesure ces éléments appartiennent au mode système ou à la simulation de matériel.

Le premier thread noyau On s'intéresse à la création du premier thread noyau. Pour répondre aux questions on conseille aussi la lecture des fichiers `threads/thread.h` et `threads/thread.cc`. Comment est créé ce premier thread noyau ? C'est à dire d'où viennent sa pile et ses registres ? Quel est le rôle (futur) de la structure de données allouée par

```
currentThread = new Thread("main");
```

Les prochains threads noyaux peuvent ils être créés de la même façon ?

4.3 Exécution d'un programme utilisateur : simulateur MIPS

Le processeur MIPS Repérez dans le code de la fonction `Kernel::Initialize()` l'allocation du processeur MIPS et lisez le code d'initialisation de cet objet pour répondre aux questions suivantes : Comment sont initialisés les registres de ce processeur ? Quelle variable représente la mémoire MIPS ?

Revenez à la fonction `main()` . Survolez le code de cette fonction pour y reconnaitre le chargement du programme en mémoire (simulée ou réelle ?), l'initialisation des registres du processeur MIPS et surtout le lancement de l'exécution du processeur MIPS qui est effectivement fait par la fonction `Machine::Run`.

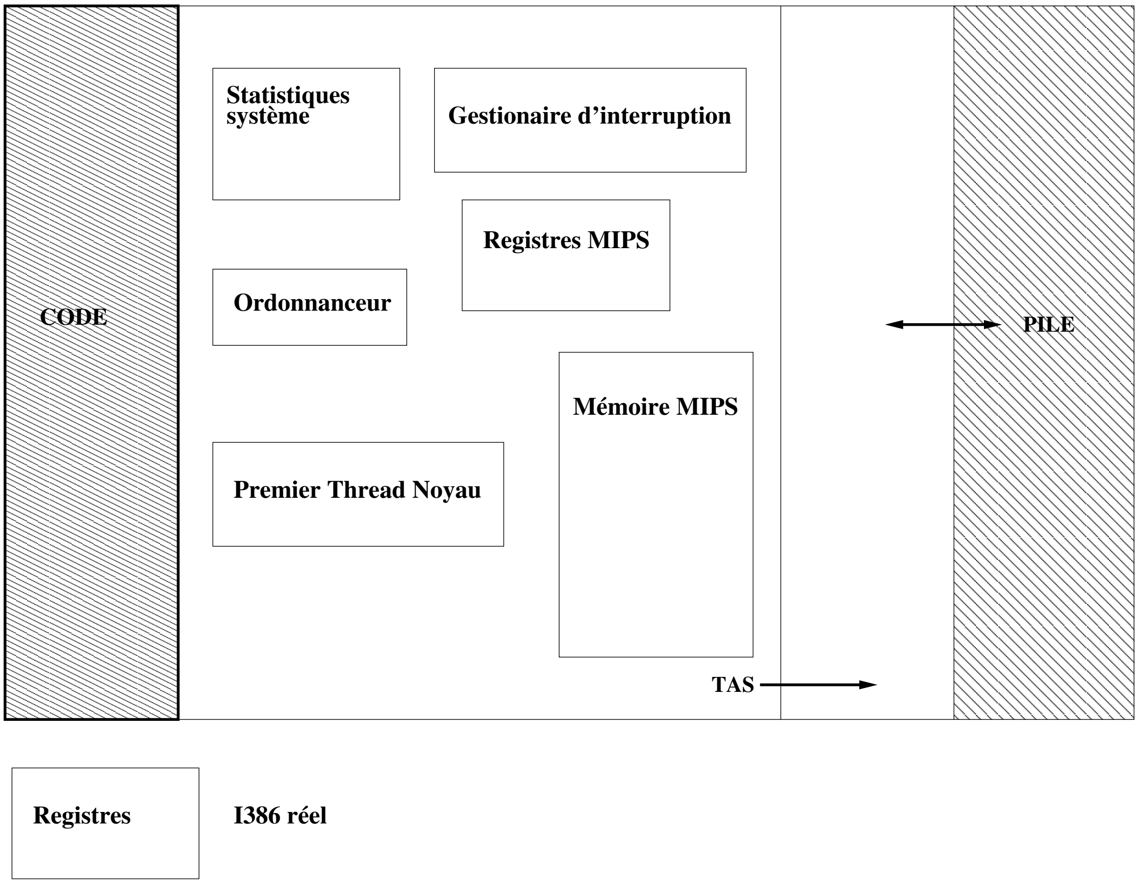

FIG. 1 – Organisation de la mémoire du process nachos

Lisez le code de la fonction `Machine::Run`, repérez la fonction qui exécute une instruction MIPS. L'observation de cette fonction vous permet de connaitre le nom de l'exception levée lorsqu'une addition (instruction assembleur `OP_ADD`) déborde (même si cela ne s'avère pas crucial pour la suite). Observez la fin de cette fonction pour trouver le registre contenant le compteur de programme.

L'appel système Halt (facultatif) Une fois dans la fonction `Machine::Run` la simulation d'un programme utilisateur ne peut être interrompue que de deux façons : soit une interruption est déclenchée (cf la fonction `Interrupt::OneTick()`), soit le programme utilisateur fait un appel système.

On se propose d'observer la fin du déroulement de l'appel système `Halt()` présent à la fin du programme de test `halt.c`. L'instruction assembleur codant un appel système dans `OneInstruction()` est `OP_SYSCALL`. Observer le traitement de cette instruction, notamment le moment où le système reprend la main et le passage du numero de l'appel système (ici `SC_Halt`) par un registre (lequel ?) du processeur MIPS. Suivre le code jusqu'à l'exécution de la fonction `CleanUp()` dont le rôle est de désallouer tout le simulateur.

5 Utilisation du système Nachos

5.1 Observation de l'exécution d'un programme utilisateur

Déplacez-vous sous le répertoire `$NOS/code/test`, et regardez le programme `halt.c`. Faire `make halt` pour le compiler.

Lancez

```
nachos -x halt
```

Les programmes utilisateurs sont écrits en `C`, compilés en binaires MIPS qui sont chargés et exécutés par la machine Nachos, instruction par instruction.

Tracer pour comprendre Essayez de tracer l'exécution du programme `halt` :

```
nachos -d m -x halt

nachos -rs 0 -d m -x halt
```

Vous pouvez en plus exécuter le simulateur MIPS pas à pas :

```
nachos -s -d m -x halt
```

Modifiez le programme `halt.c` pour y introduire un peu de calcul, par exemple en faisant quelques opérations sur une variable entière. Tracez pas à pas pour bien voir que cela change quelque chose...

N'oubliez pas que dans ce monde MIPS, vous n'avez à votre disposition que les fonctions du langage `C` et les appels systèmes de Nachos. Aucune fonction de bibliothèque (`printf`, ...) n'est disponible.

Il est aussi possible de générer facilement la version assembleur d'un programme MIPS. Commencez par détruire le fichiers `halt` et faites `make halt`. Récupérez la ligne de commande produite par copier-coller, qui doit ressembler à :

```
/opt/NACHOS/nachos_gcc/cross/decstation-ultrix/bin/gcc -G 0 -O3 \
  -ggdb -c -I../userprog -I../lib -c halt.c -o halt.o
```

Changez le `-c` final en `-S`. Cela va produire le code assembleur MIPS de `halt` dans le fichier `halt.s`. Exécutez de nouveau `halt` pas à pas en suivant maintenant les instructions code assembleur !

Repérez les instructions assembleur codant les calculs et l'appel à la fonction `Halt`. On peut trouver le code de cette fonction `Halt` dans le fichier `test/start.s` ce qui permet de faire le lien avec l'étude de la fin de l'appel système dans le partie précédente.

(Question facultative) Pourquoi la premiere instruction MIPS est elle éxécutée au dixième *tick* d'horloge ? (cf la fonction `Interrupt::OneTick()` et l'initialisation du système)

Plus sur les traces Le simulateur et le système Nachos sortent les traces sur la sortie d'erreur standard (`std::cerr` en C++, file descriptor 2 en C). La sortie standard (`std::cout` en C++, file descriptor 1 en C)est réservée à la sortie console (`ConsoleOutput` des fichiers `machine/console.h` et `machine/console.c`. Ces 2 sorties standards peuvent être redirigée dans des fichiers. Par exemple, si on exécute dans un xterm

```
rm -f TRACE_NACHOS;  nachos -K -d +  2> TRACE_NACHOS
```

récupère dans le fichier `TRACE_NACHOS` toutes les traces. On n'aura dans la fenêtre du xterm les seules E/S console Nachos.

```
rm -f TRACE_NACHOS;
(xterm -hold -e tail -f TRACE_NACHOS &);nachos -C -d + 2> TRACE_NACHOS
```

permet de visualiser la trace dans un xterm secondaire et de garder le xterm maître pour le test de la console Nachos.

5.2 Observation des threads noyau

Nous allons maintenant tester les threads systèmes Nachos, c'est-à-dire les threads de niveau *noyau*. Nous allons utiliser le système NACHOS "seul", c'est-à-dire en configuration d'auto-test. Lancez `nachos -u` pour connaitre les différentes options de lancement de nachos.

```
Partial usage: nachos [-z -d debugFlags]
Partial usage: nachos [-x programName]
Partial usage: nachos [-K] [-C] [-N]
Partial usage: nachos [-rs randomSeed]
Partial usage: nachos [-s]
Partial usage: nachos [-ci consoleIn] [-co consoleOut]
Partial usage: nachos [-n #] [-m #]
```

L'option d'exécution `K` fait que Nachos lance la fonction `Kernel::ThreadSelfTest` dans `main.cc` (allez jeter un œil !) Cette fonction est définie dans `thread.cc` et teste différentes parties de l'implantation des threads. Lesquelles ? Examinez attentivement la définition de `Thread::SelfTest`. Ce test lance deux threads internes au noyau qui affichent tour à tour une ligne à l'écran pendant 5 itérations.

Nachos dispose d'options de déboggage. Essayez par exemple

```
nachos -K -d +   # + = all possible options
```

Vous pouvez voir les *ticks* de l'horloge interne de Nachos, les commutation entre threads, la gestion des interruptions, etc.

Il est possible de n'afficher qu'une partie de ces informations. Au lieu du `+`, on peut mettre :

t pour ce qui a trait aux threads systeme Nachos,

a pour ce qui a trait à la mémoire MIPS,

m pour ce qui a trait à la machine Nachos,

t pour ce qui a trait à la gestion des threads,

i pour ce qui a trait aux interruptions,

n pour ce qui a trait au réseau Nachos,

f pour ce qui a trait au filesystem Nachos,

d pour ce qui a trait aux disques Nachos.

Certaines parties de Nachos (disques, ...) ne sont pas manipulées pour l'instant. Le flag correspondant n'affichera alors aucun message supplémentaire.

Allons maintenant éditer le fichier `thread.cc`. Ajoutez le lancement de 2 threads supplémentaires dans la fonction `Thread::SelfTest()`. Ça marche toujours ?

La sémantique de la méthode `Fork` de l'objet `Thread` n'a rien à voir avec celle de la fonction Unix `fork`. Que fait la méthode `Thread::Fork` dans Nachos ? À quel moment les threads nachos sont-ils créés (mémoire allouée, structures initialisées, ...) ?

Maintenant, commentez la ligne suivante :

```
kernel->currentThread->Yield();
```

Recompilez (`make`) et examinez ce qui se passe. Qu'en déduisez-vous pour la préemption des threads systèmes par défaut ?

Restaurez cette ligne. On peut lancer Nachos en forçant une commutation pseudo aléatoire par l'option `-rs <semence>`. La semence passée en paramètre rend aléatoire (mais *reproductible* !) l'entrelacement des threads (le nombre `semence` n'a pas de signification particulière, autre qu'être celui servant à initialiser le générateur aléatoire).

```
nachos -K -rs 0
nachos -K -rs 1
nachos -K -rs 7
```

Que se passe-t-il ? Couplez cela avec l'option `-d +`. Combien de ticks d'horloge maintenant ?

Ce point est assez difficile à comprendre. Vérifiez votre intuition en commentant la ligne :

```
kernel->currentThread->Yield();
```

Vos conclusions ?

5.3 Découverte de l'ordonnanceur

L'objectif est maintenant de comprendre une partie du fonctionnement de l'ordonnanceur en s'appuyant sur l'expérience précédente.

Le changement de contexte explicite Que se passe-t-il exactement lors d'un appel à la fonction `Thread::Yield()` ? Allez inspecter le source de cette fonction dans le fichier `code/threads/thread.cc`. A quel moment un thread ressort-il de cette fonction ?

La classe Scheduler Examinez les méthodes de la classe `Scheduler` appelées par la fonction `Thread::Yield()`. Quels sont les rôles respectifs des fonctions `Scheduler::ReadyToRun()`, `Scheduler::FindNextToRun()` et `Scheduler::Run()` ?

Au cœur du changement de contexte Dans quelle fonction de la classe `Scheduler` trouve-t-on la véritable instruction provoquant un changement de contexte (i.e. une commutation) entre deux processus ? Trouvez le source de la fonction de bas niveau correspondante. Que fait-elle ?

5.4 Exécution de NACHOS pas à pas

Dans les situations extrêmes, il vous sera toujours possible de suivre l'exécution de NACHOS réellement pas à pas, en utilisant un deboggeur tel que `gdb`. Par défaut, NACHOS est d'ailleurs compilé avec l'option `-g` (voir les `Makefile`).

Retournez dans le répertoire `threads` et lancer Nachos dans le deboggeur `gdb` en plaçant un point d'arrêt sur `main` `Thread::SelfTestet` et `SimpleThread` après avoir passé l'argument `-K`

```
gdb nachos
...
(gdb) set args -K
(gdb) break main
Breakpoint 1 at 0x805704e: file ../threads/main.cc, line 182.
(gdb) break Thread::SelfTest
Breakpoint 2 at 0x805c72f: file ../threads/thread.cc, line 428.
(gdb) break SimpleThread
Breakpoint 3 at 0x805c6aa: file ../threads/thread.cc, line 413.
(gdb) run
Starting program: /home/nachos/NachOS-4.1/code/build.linux/nachos -K
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000

Breakpoint 1, main (argc=2, argv=0xbfbd4ac4) at ../threads/main.cc:182
```

```
182         char *debugArg = "";
(gdb) cont
Continuing.


tests summary: ok:0

Breakpoint 2, Thread::SelfTest (this=0x806a2a8) at ../threads/thread.cc:428
428         DEBUG(dbgThread, "Entering Thread::SelfTest");
(gdb) cont
Continuing.

Breakpoint 3, SimpleThread (which=1) at ../threads/thread.cc:413
413         for (num = 0; num < 5; num++) {
(gdb) cont
Continuing.
*** thread 1 looped 0 times

Breakpoint 3, SimpleThread (which=0) at ../threads/thread.cc:413
413         for (num = 0; num < 5; num++) {
(gdb) cont
Continuing.
*** thread 0 looped 0 times
*** thread 1 looped 1 times
*** thread 0 looped 1 times
*** thread 1 looped 2 times
*** thread 0 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
Machine halting!

Ticks: total 2250, idle 0, system 2250, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Program exited normally.
(gdb)
```

Modifiez la fonction `SimpleThread()`, recompilez, relancez, réfléchissez, etc.

Exécutez `SimpleThread()` pas à pas. Que se passe-t-il quand on appuie sur `n` (*next*) lorsque gdb est prêt à exécuter la méthode `Thread::Yield`? Comment peut-on placer un point d'arrêt pour reprendre la main dans gdb chaque fois qu'un changement de contexte est exécuté par Nachos ?

5.5 Exercice

Modifiez la méthode `Thread::Yield` pour qu'elle ne fasse de changement de contexte qu'une fois tous les deux appels.

Relancez le programmes `nachos` et bservez le déroulement (avec l'option `-K -d t` de Nachos et/ou avec `gdb`).

Annexe : Outils d'édition du code

Le code de NACHOS est réparti dans de nombreux fichiers et lors de sa lecture on est amené à sauter de fichier en fichier. Les *tags* ("étiquettes" en francais), permettent de trouver automatiquement la définition d'une fonction à partir d'un endroit où elle est appelée.

Pour cela il faut dans un premier temps construire un dictionnaire référençant les définitions des fonctions. Ce dictionnaire diffère selon l'éditeur utilisé. Ce sont les programmes `etags` pour emacs et `ctags` pour vim qui construisent un tel dictionnaire et le stockent dans un fichier (`TAGS` pour emacs et `tags` pour vim).

Ensuite l'éditeur que vous utilisez va lire dans ce fichier (quand il l'a trouvé) la position de la définition d'une fonction et saura ouvrir le fichier correspondant.

Mise en œuvre pratique pour l'éditeur `vim`

Construction du dictionnaire

Les premières versions de `ctags` traitaient principalement les fichiers C. Comme NACHOS est écrit en C++ il faut une version relativement récente qui prenne en compte les classes.

Pour créer le dictionnaire des fonctions de NACHOS taper :

```
cd $NOS/code
ctags */*.cc */*.c
```

et vous devez voir apparaître un fichier `tags` dans le répertoire courant (ici `$NOS/code`)

Lire le code

On va utiliser ce dictionnaire pour lire le code de NACHOS à l'aide de l'éditeur `vim`. Pour éviter des écritures accidentelles on l'utilisera en lecture uniquement (cf l'option -R).

```
vim -R
```

Cet éditeur fonctionne selon deux modes : le mode insertion où l'on peut modifier le contenu du fichier et le mode commande où l'on réalise par exemple les ouvertures et fermetures de fichiers (á comparer á ce que l'on tape dans la barre de commande de `emacs`). La lecture des sources s'effectue dans le mode commande. Pour y basculer il suffit d'appuyer sur la touche **Echap**.

Commandes élémentaires

Commande	**Résultat**
`Echap`	Passe en mode commande (y reste si déjà en mode commande).
`:help` *sujet*	Donne l'aide sur le *sujet* (exemple `:help tags`).
`:q!`	Force la sortie de l'éditeur sans sauvegarde.
`:new` *un_fichier*	Ouvre *un_fichier*
`:ts` *une_fonction*	Propose la liste des fonctions de nom *une_fonction* et va à la définition choisie.
`:ta` *une_fonction*	Va à la définition de la première fonction dans le dictionnaire de noms *une_fonction*
`:tags`	Affiche la pile des tags courants (similaire à la pile des appels de fonctions).
`:pop`	Dépile le premier tag de la pile.

Par défaut `vim` cherche le dictionnaire dans le répertoire courant. C'est pourquoi il est conseillé de lancer `vim` dans le répertoire où se trouve le dictionnaire. Sinon on peut donner le nom du dictionnaire à lire par la commande

`:set tags=`*`repertoire/le_fichier_dictionnaire`*

Comme `emacs`, `vim` propose des raccourcis claviers pour les commandes les plus usuelles. Ainsi au lieu de la commande

`:ta` *`une_fonction`*

on pourra préférer positionner le curseur sur l'occurrence de *une_fonction* dans le code et taper `Ctrl-]`. De même,

`:pop`

admet `Ctrl-t` comme abréviation.

Mise en œuvre pratique pour l'éditeur `emacs`

Pour contruire le dictionnaire taper :

```
cd $NOS/code
etags */*.cc */*.c
```

Pour l'utiliser lancer `emacs` dans le même répertoire.

Pour chercher la définition d'une fonction, placer le curseur sur son nom et taper `Meta-.` ("Méta - point"). Pour revenir à la définition précédente taper `Meta-*`.

Lors de la première recherche d'un tag, emacs demande le nom du fichier du dictionnaire, ici c'est `$ASWD/nachos/code/TAGS`.

La commande `grep`

Si on cherche autre chose que la définition d'une fonction (par exemple un appel de cette fonction ou un nom de variable) la commande `grep` est utile.

```
cd $NOS/code
grep -r ``currentThread'' */*.cc */*.h */*.c
```

Cela affiche les lignes contenant la variable `currentThread` dans tous les fichiers sources.