Systèmes d'Exploitation

Fiche 0 : Premiers pas avec le simulateur NACHOS

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/se/

Résumé

Nachos est un mini-système simulé assez basique dans lequel vous allez implémenter quelques briques d'un vrai systèmes. Il est implémenté en C++, dont la syntaxe orientée-objet utilisée dans Nachos est très similaire au Java. Le travail se fera par binômes (éventuellement un trinôme s'il y a un nombre impair d'étudiants dans le groupe)

Les objectifs des premières séances sont :

- la mise en place d'une archive svn pour pouvoir travailler à plusieurs sur le même code ;
- 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.

Il ne s'agit donc pas d'implémenter quoi que ce soit, mais surtout de passer un peu de temps à explorer Nachos. Un certain nombre de resources complémentaires sont disponibles ici : http://dept-info.labri.fr/~guermouc/SE/

1 Installation de NACHOS

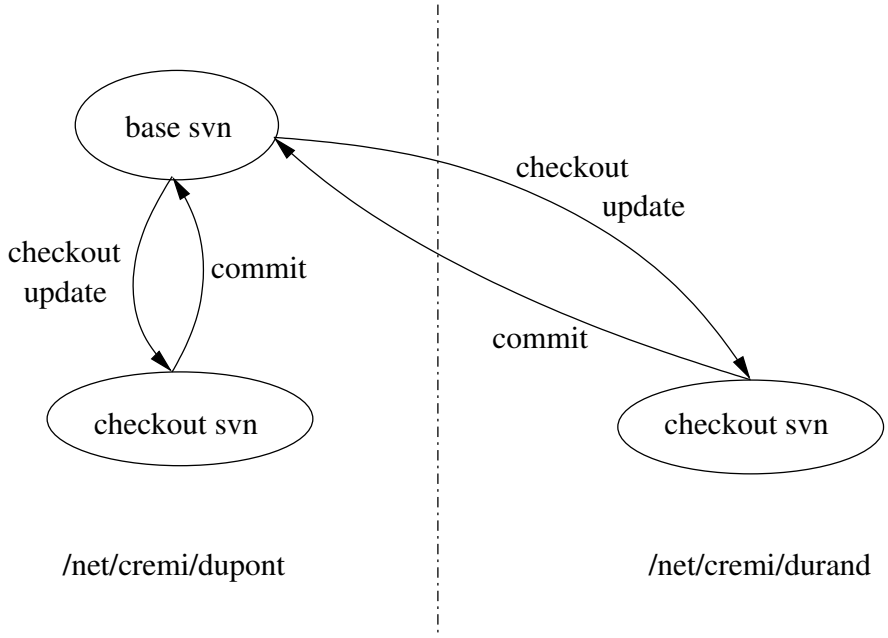
Pour travailler dans de bonnes conditions, en binôme de surcroît, vous allez utiliser un outil permettant de gérer facilement le travail en équipe (partage de fichiers sources) et l'archivage de versions multiples. Il s'agit de svn (*subversion*), qui vous permettra en effet de travailler simultanément avec votre binôme sur les mêmes fichiers sources (y compris en édition), mais aussi de gérer facilement les versions successives des fichiers afin de pouvoir revenir en arrière, inspecter les dernières modifications, etc.

Pour vous faciliter la tâche, nous vous distribuons le simulateur NACHOS sous la forme d'une "base svn" prête à l'emploi.

1.1 Installation de la base svn

Une base svn est une arborescence de fichiers dans laquelle l'outil svn va mémoriser l'intégralité de votre projet (fichiers sources) sous une forme qui lui permettra de restituer l'historique complet des modifications, n'importe quelle version intermédiaire des fichiers, ou encore une synthèse des différences entre deux versions d'un même fichier. Le principe général est que chaque utilisateur travaille sur une copie locale (un *checkout*) des fichiers gérés par svn. Les synchronisations vers/depuis la base svn sont effectuées explicitement à l'aide de la commande `svn`, comme nous le verrons ci-après.

Dans le cadre des TP de système, il suffira d'utiliser une seule base svn par binôme, hebergée dans le compte d'une seule des personnes. Ensuite, chaque étudiant (y compris celui qui héberge la base) devra travailler sur un *checkout* des fichiers, en utilisant la commande *svn*



commit pour mettre à jour la base svn, et la commande *svn update* pour récupérer les mises à jour de l'autre étudiant (nous verrons cela ci-après). Cela est illustré par le dessin ci-dessous :

Voici une illustration très concrète. Nous supposerons ici que vous vous appelez Dupont et Durand, et que c'est Dupont qui hébergera la base svn.

Dupont (et lui seul, donc) choisit d'installer la base svn sous son répertoire racine `$HOME` (`/net/cremi/dupont/`, pour vous il faut bien sûr remplacer `dupont` par votre *login*). Il extrait une copie de la base "prête à l'emploi" fournie par nos soin de cette façon :

```
cd
tar xvzf ~sathibau/distrib-nachos-svn.tar.gz
```

Le répertoire de la base svn ainsi installée est `/net/cremi/dupont/svnnachos`.

À présent, il faut s'assurer que les partenaires peuvent écrire dans ce répertoire et y créer des fichiers :

```
chmod +x /net/cremi/dupont
chmod -R a+rwX /net/cremi/dupont/svnnachos
```

L'inconvénient est que tous les membres du même groupe Unix peuvent aussi y accéder... Une méthode peut être d'utiliser un répertoire (`~/hidden` par exemple) doté seulement du droit en exécution (i.e. traversée) mais pas en lecture et d'y créer un répertoire avec un nom secret (ici `JK5VTqX` par exemple, mais utilisez autre chose, sinon ce ne serait pas secret !) que vous ne communiquerez qu'à votre binôme en secret.

```
umask 066
mkdir ~/hidden
umask 022
mkdir ~/hidden/JK5VTqX
mv svnnachos ~/hidden/JK5VTqX/
```

Maintenant, tout est prêt pour commencer à travailler en binôme.

1.2 Utilisation de svn

ATTENTION : contrairement aux instructions précédentes, ici Dupont et Durand doivent exécuter, chacun de leur côté, les instructions suivantes (qui sont illustrées uniquement pour Durand).

Il faut d'abord ajouter à son fichier `.bashrc` la ligne suivante.

```
export EDITOR=nano # Par exemple... Évitez VSCode qui ne se comportera pas bien
```

Il faut ensuite exécuter votre `.bashrc` pour bien positionner la variable d'environnement.

```
source ~/.bashrc
```

Lors de votre prochaine entrée sous le système, ce sera fait automatiquement à l'initialisation.

Durand a décidé ici de travailler dans un répertoire `Systeme/Nachos`.

```
mkdir ~/Systeme
cd ~/Systeme/
mkdir Nachos
cd Nachos
```

Si vous avez protégé votre base svn en la cachant, il faut interdire aux autres personnes de lire ce répertoire (car il contiendra le nom du répertoire de la base dans certains fichiers).

```
chmod og-rwx $HOME/Systeme/Nachos
```

Récupérez depuis la base svn une copie de travail de la version courante du projet.

```
svn checkout file:///net/cremi/dupont/hidden/JK5VTqX/svnnachos/ nachos
```

Cela reconstruit au point d'exportation `nachos` toute la hiérarchie des fichiers. Il n'est nécessaire de faire le `checkout` qu'une seule fois : pour mettre à jour par la suite, il suffira d'appeler `svn update`.

Promenez-vous dans l'arborescence pour aller modifier un fichier. Par exemple

```
cd nachos/code/test
```

et modifiez le fichier `halt.c`.

```
echo "/* This is a strange comment... */" >> halt.c
```

Vérifiez avec `svn status` que la modification a bien été vue par svn (*locally modified*), et `svn diff` les modifications. Par contre, votre binôme ne la voit pas encore dans sa propre copie de travail.

Maintenant, *commitez* cette modification :

```
svn commit
```

L'éditeur s'ouvre pour que vous puissiez documenter le changement que vous venez de faire, ici un simple test. Il est obligatoire de mettre un commentaire, à mettre en tête, **avant** la ligne `--Cette ligne, et les suivantes ci-dessous, seront ignorées--`. Vous en êtes à la révision `2`... Si votre binôme fait `svn checkout` ou `svn update` chez lui, il verra la nouvelle version. Faites `svn update` chez vous aussi pour mettre à jour le svn log.

De plus, si par hasard il était lui aussi en train de travailler sur `halt.c`, svn essaie de fusionner sa version et votre version de manière "raisonnable". En général, ce n'est pas trop mal. En cas de conflit insoluble, svn vous met un message et annote le fichier. Faites un essai !

Maintenant, détruisez (par erreur !) le fichier `halt.c`. Vous pouvez remettre à jour votre copie locale à partir de la base svn en tapant tout simplement :

```
svn update
```

Il peut arriver que `svn update` apporte des conflits, si plusieurs personnes ont modifié les mêmes parties du code. Svn vous propose alors de modifier le fichier vous-même pour résoudre le conflit : dans votre éditeur, chercher les sections encadrées par des `<<<<`, `====` et `>>>>`

De manière plus générale, à chaque fois que vous commencez à travailler, n'oubliez pas de faire `svn update` afin de mettre à jour votre copie locale à partir de la dernière version déposée dans la base. N'utilisez `svn checkout` que pour créer une nouvelle copie.

Vous pouvez également utiliser `svn log` et `svn diff` pour examiner l'historique des changements, par exemple :

```
$ svn log
------------------------------------------------------------------------
r2 | dupont | 2009-12-28 13:55:57 +0100 | 1 ligne

ajout test
...
$ svn diff -c 2
Index: test/halt.c
===================================================================
...
$ svn diff -r 1
...
```

L'option `-c` de `svn diff` permet d'examiner les changements apportés par un commit donné, tandis que l'option `-r` permet d'examiner tous les changements depuis un commit donné. `svn diff` seul montre les modifications locales non encore commitées (i.e. implicitement tous les changements depuis le dernier commit). Il est recommandé de l'utiliser avant de commiter, pour bien vérifier tout ce que l'on est sur le point de commiter.

On peut revenir à une version précédente en donnant un paramètre `-r` à `svn up` :

```
$ svn up -r 1
U test/halt.c
Actualisé à la révision 1.
```

C'est cependant juste pour regarder, on ne peut pas committer à partir de là ; pour faire revenir la base svn à une ancienne révision, il faut "défaire" les commits dont on ne veut pas :

```
$ svn up
U test/halt.c
Actualisé à la révision 2.
$ svn diff -r 1 > monpatch
$ patch -R < monpatch
$ svn commit
```

La commande `patch`, quand on lui donne l'option `-R`, applique les modifications à l'envers, ce qui est bien ce que l'on veut.

Notez enfin qu'il n'y a bien sûr aucune raison de travailler dans le même chemin de répertoires dans vos *home*s respectifs.

1.3 Test de NACHOS

Positionnez-vous sous `nachos/code` dans la copie que vous venez d'extraire, et construisez Nachos :

```
cd $HOME/Systeme/Nachos/nachos
cd code
make clean      # Pour se remettre dans un état standard (prudent!)
make            # Pour produire les fichiers de dépendances
                # et lancer la compilation (cela peut être long, ajoutez
                # éventuellement l'option -j8 pour profiter du
                # parallélisme de la machine...)
```

Normalement, tout doit bien se passer... Pendant cette compilation vous pouvez lire les conseils ci-dessous et commencer la partie suivante qui debute par une lecture du code.

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...

```
cd threads
./nachos
```

pour le premier, et pour le second

```
cd ..
cd userprog
./nachos -x ../test/halt
```

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

1.4 Quelques conseils...

Ne faites des `svn commit` que lorsque votre partie est suffisamment testée. Sinon, votre binôme sera complètement perdu ! L'invariant doit être : la version disponible sous svn fonctionne. Un avantage, c'est qu'ainsi il suffit d'utiliser `svn diff` pour regarder les modifications que vous avez apportées, et ainsi détecter facilement des choses qui ont été ajoutées par erreur (ou oubliées).

Prenez le temps de mettre des commentaires significatifs dans le *log*. Cela vous aidera beaucoup dans les phases d'intégration.

Placez votre `svnnachos` dans une autre partie de votre arborescence que la partie dans laquelle vous travaillez habituellement. Prenez aussi le temps de faire des sauvegardes de temps en temps, on ne sait jamais...

```
mkdir ~/Sauvegardes

tar czvf ~/Sauvegardes/nachos.tar.gz \
        /net/cremi/dupont/hidden/JK5VTqX/svnnachos
```

1.5 Travailler depuis chez soi avec svn

Vous pouvez aussi utiliser svn pour synchroniser votre travail réalisé en dehors du CREMI avec la base svn que vous venez d'installer. Pour cela il vous suffit de créer une copie locale sur votre machine personnelle en précisant que la base utilisée est au CREMI :

```
svn checkout svn+ssh://dupont@jaguar.emi.u-bordeaux.fr/net/cremi/dupont/hidden/JK5VTqX
```

Et toutes les autres commandes svn fonctionnent elles aussi comme précédement. N'oubliez pas d'utiliser `svn commit` lorsque vous avez fini de travailler en dehors du CREMI, et `svn update` pour mettre à jour la première copie locale au CREMI.

Note : pour établir la connexion ssh, cf le guide https://gforgeron.gitlab.io/ssh/guide.pdf

Remarque : vous aurez aussi besoin du compilateur MIPS `xgcc` comme expliqué plus loin. Pour l'installer, le plus simple est de récupérer la version déjà compilée depuis

`http://dept-info.labri.fr/~guermouc/SE/`

et la dépaqueter dans `/opt` (si vous avez un système 64bit, il faut penser à installer une libc 32bit, typiquement appelée libc6-i386) mais les warriors peuvent s'amuser à le compiler eux-mêmes.

2 Lecture du code source de Nachos

Important : Avant de commencer ce chapitre, lisez les annexes à propos des outils pour la lecture de code, à la fin du poly.

2.1 Principes de la simulation

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

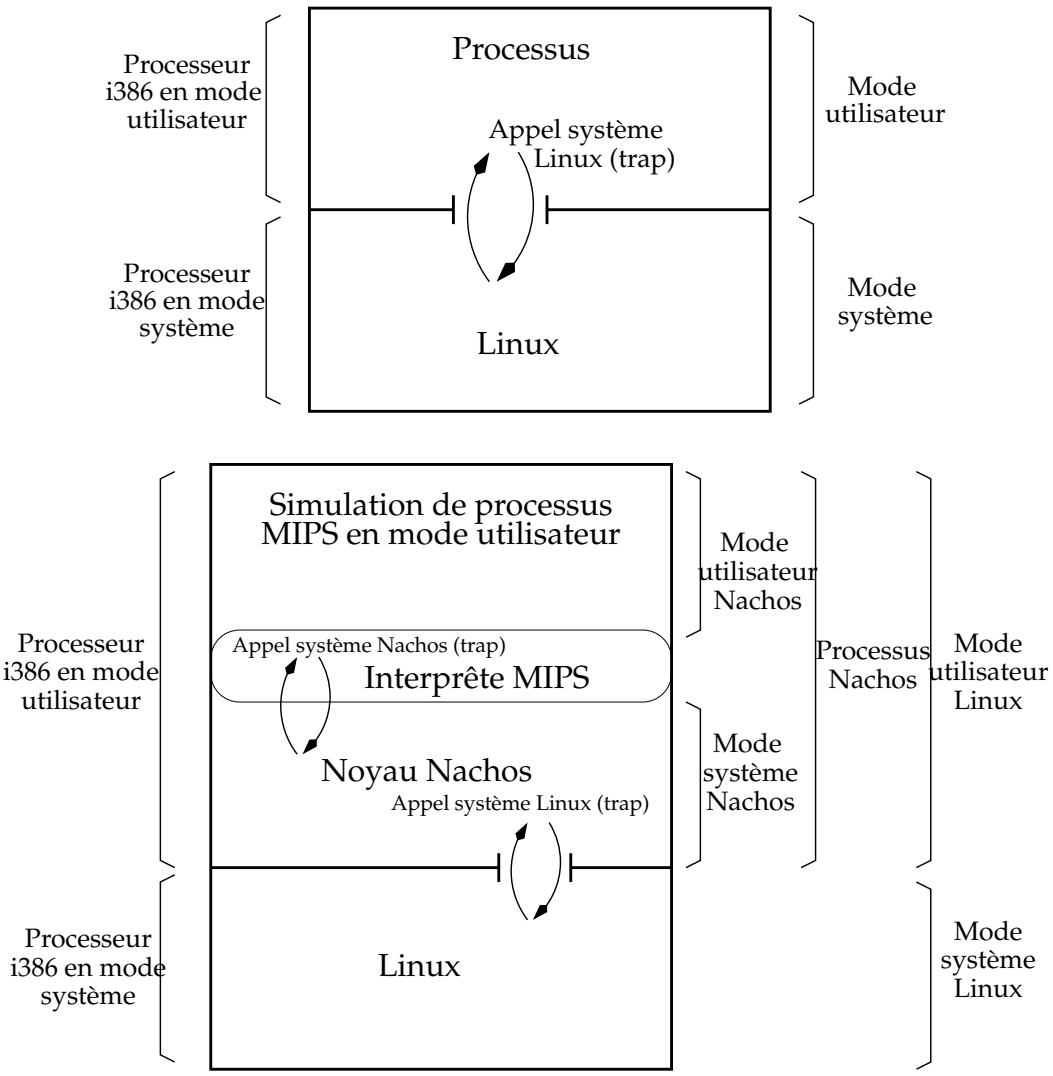
- le **mode utilisateur** (*user mode*) dans lequel il exécute les instructions d'un programme utilisateur ;
- le **mode système** (*system mode*) dans lequel il organise les différentes tâches systèmes qui lui sont demandées : communication avec les périphériques (disque-dur, clavier, carte graphique ...), 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, comme sur le schéma suivant.

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 Nachos est exécuté sur le processeur réel (i386 et affiliés), ce qui nous permettra de le débugguer avec gdb, valgrind, etc. En revanche, les programmes de niveau utilisateur nachos sont compilés pour processeur MIPS (un autre type de processeur que l'x86 d'Intel) et émulés par un simulateur de processeur MIPS (aussi appelé "interprète MIPS"), ce qui nous permet de contrôler la progression des programmes. 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 ...

Tout ceci est résumé sur le schéma suivant.



Donner un exemple de fichier source d'un programme utilisateur MIPS, et un exemple de fichier source du noyau Nachos. Quel est le langage de programmation utilisé à chaque fois ?

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 suivante depuis `userprog` :

```
./nachos -x ../test/halt
```

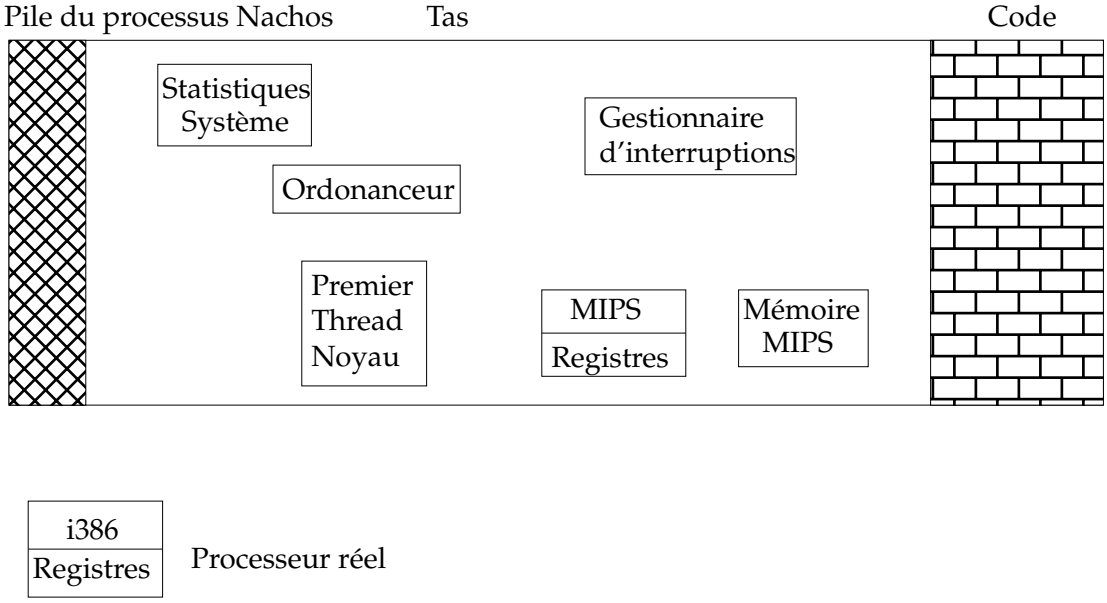
`nachos` ayant été compilé avec le seul "drapeau" (flag) `USER_PROGRAM`. Le comportement de cette commande est d'initialiser le système `nachos` et d'exécuter le programme utilisateur MIPS `../test/halt` qui demande l'arrêt du système.

Avant de lire le code vous pouvez (devez) consulter l'annexe (à la fin de ce document) sur l'utilisation des tags qui peut vous faciliter grandement la vie.

2.2 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 comment 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 suivant représente les ressources du processus nachos et les éléments du système qui sont initialisés à la fin de la fonction `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. Notez bien que lors des prochains TPs il ne faudra pas modifier les fichiers concernant la simulation de matériel (c'est un cours de système, pas d'architecture !). On pourra par contre acheter des barrettes mémoire à prix très compétitif en augmentant la macro `NumPhysPages`.

Le premier thread noyau On s'intéresse à la création du premier thread noyau. Pour répondre aux questions on conseille aussi de parcourir brièvement les 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");
```

Pourquoi pour les prochains threads noyaux faudra-t-il en plus appeler la méthode `Start` ?

2.3 Exécution d'un programme utilisateur

Le processeur MIPS Repérez dans le code de la fonction `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()` et vérifiez que la fonction `StartProcess()` est appelée avec le nom du fichier `../test/halt`. 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 par la fonction `Machine::Run`.

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 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 `../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.

3 Exécution de NACHOS pas à pas

En cas de bug, il vous sera toujours possible de suivre l'exécution de NACHOS, en utilisant un deboggeur tel que `gdb`. Par défaut, NACHOS est d'ailleurs compilé avec l'option `-g` (voir les `Makefile`). On pourra éventuellement préférer un débuggueur plus graphique, que ce soit `gdb -tui`, `ddd`, `tdb`, ...

Retournez dans le répertoire `threads` et lancer Nachos dans le deboggeur `gdb`. Prenez l'habitude d'utiliser l'option `--args` de `gdb`, pour que lorsque vous ajoutez des options après `./nachos`, elles sont passées à Nachos, pas à `gdb`.

```
gdb --args ./nachos
[...]
(gdb) break main
Breakpoint 1 at 0x804d6fa: file main.cc, line 84.
(gdb) run
Starting program: nachos
[...]
Breakpoint 1, main (argc=1, argv=0x8046db0) at main.cc:84
84          DEBUG('t', "Entering main");
(gdb)
```

Vous pouvez progresser avec les commandes `s` (atomic *step*), `n` (*next* instruction in the current fonction), `b` (*breakpoint* pour définir un point d'arrêt), `c` (*continue* jusqu'au prochain *breakpoint*),

`r` (*run* : (re)démarre l'exécution du progamme), etc. Vous pouvez également utiliser `p` (*print*) pour afficher les valeurs des variables.

Essayez par exemple :

```
(gdb) break ThreadTest
[...]
(gdb) cont
[...]
```

Ceci place un point d'arrêt sur la fonction `ThreadTest`, puis dit à Nachos de continuer son exécution jusqu'à ce qu'il arrive sur un point d'arrêt. Remarque : pour positionner un point d'arrêt sur une méthode de classe, il faut préciser le nom de la classe, par exemple : `break Thread::Start` On peut également spécifier une ligne précise, avec `break main.cc:100` par exemple.

Notez par ailleurs que si une exécution de nachos se termine par une erreur d'assertion, il suffit de le lancer dans `gdb` pour savoir où elle est et qu'enfilade des appels. Essayez d'ajouter `ASSERT(FALSE);` au début de `SimpleThread`. On constate

```
$ ./nachos
Assertion failed: line 30, file "threadtest.cc"
zsh: abort      ./nachos
```

Et donc on lance :

```
$ gdb --args ./nachos
[...]
(gdb) run
Starting program: /home/samy/enseignement/SE/nachos/code/threads/nachos
Assertion failed: line 30, file "threadtest.cc"

Program received signal SIGABRT, Aborted.
[...]
(gdb) bt
#0  0x00007ffff730d165 in *__GI_raise (sig=<value optimized out>)
    at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
#1  0x00007ffff730ff70 in *__GI_abort () at abort.c:92
#2  0x0000000000404697 in Abort () at ../machine/sysdep.cc:432
#3  0x00000000004032ee in SimpleThread (arg=0x0) at threadtest.cc:30
#4  0x00000000004033ae in ThreadTest () at threadtest.cc:53
#5  0x0000000000401203 in main (argc=1, argv=0x7fffffffdad8) at main.cc:104
```

On voit ici que c'est l'appel direct depuis ThreadTest qui a provoqué l'abandon, et non pas dans le thread créé. Utilisez `frame 3`, pour remonter dans la *backtrace* au niveau de `SimpleThread`. Constatez que vous pouvez alors utiliser par exemple `p arg` pour afficher la variable `arg`, etc. On peut aussi utiliser `up` et `down` pour remonter et redescendre dans la pile d'appels.

Enlevez l'ASSERT, et exécutez `SimpleThread()` pas à pas. Utilisez `display num` pour constater que gdb vous affiche l'évolution de la variable `num`. Que se passe-t-il quand on appuie sur `n` (*next*) lorsque gdb est prêt à exécuter la méthode `yield`?

Utilisez `watch currentThread` et relancez le programme. L'exécution s'arrête ainsi à chaque fois que la variable `currentThread` change de valeur, ce qui permet de savoir qui la modifie.

Pour chaque commande gdb, on peut utiliser `help lacommande` pour avoir un résumé de son utilisation.

4 Exécution de NACHOS dans valgrind

Il arrivera sans doute que vous vous trompiez et débordiez des espaces alloués, voire non alloués, auquel cas vous obtenez un segfault lors d'un malloc. Dans ce cas, utilisez `valgrind` :

```
valgrind ./nachos
```

Il vous indiquera par exemple

```
==7104== Invalid write of size 1
==7104==    at 0x400511: main (test.c:4)
==7104==  Address 0x517a04a is 0 bytes after a block of size 10 alloc'd
==7104==    at 0x4C221A7: malloc (vg_replace_malloc.c:195)
==7104==    by 0x400504: main (test.c:3)
```

Ici, on est sorti du tableau : on a écrit juste après. Ou encore

```
==7310== Invalid read of size 1
==7310==    at 0x4005BA: main (test.c:5)
==7310==  Address 0x517a045 is 5 bytes inside a block of size 10 free'd
==7310==    at 0x4C21DBC: free (vg_replace_malloc.c:325)
==7310==    by 0x4005B1: main (test.c:4)
```

Là, on accède à un tableau qui a été libéré.

5 Utilisation du système Nachos

5.1 Observation de l'exécution d'un programme utilisateur

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

Placez-vous ensuite sous `userprog`. Lancez

```
./nachos -x ../test/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 ../test/halt
```

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

```
./nachos -s -d m -x ../test/halt
```

Notez la présence de l'option `-h` qui documente brièvement les options.

Remarquez également la présence d'un fichier `memory.svg`, que vous pouvez ouvrir par exemple avec Firefox, il vous montre la mémoire lors du démarrage du programme, on en reverra les détails plus tard.

Modifiez le programme `halt.c` pour y introduire un peu de calcul, par exemple en faisant quelques opérations sur une variable entière **globale** (pour éviter que gcc se rende compte que c'est un calcul bidon et optimise). 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, en lançant par exemple `make 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 ? (mettez un *breakpoint* sur la fonction `Interrupt::OneTick()` pour savoir à quel moment elle est appelée en premier)

5.2 Observation des threads noyau

Nous allons tester le système NACHOS "seul", c'est-à-dire en configuration d'auto-test. En l'occurrence, ce test consiste à lancer deux threads internes au noyau qui affichent tour à tour une ligne à l'écran pendant 5 itérations. Placez-vous dans le répertoire `threads`. Lancez NACHOS :

```
./nachos
```

Quelles sont les options de compilation de ce répertoire ? Ces options de compilation font que Nachos lance la fonction `ThreadTest` dans `main.cc` (allez jeter un œil !) Cette fonction est définie dans `threadtest.cc`. Examinez attentivement sa définition.

Nachos dispose d'options de déboggage. Essayez par exemple

```
./nachos -d +   # + = all possible options
```

Vous pouvez voir les *ticks* de l'horloge interne de Nachos, les commutation entre threads, la gestion des interruptions, etc. Ce sont les appels à DEBUG, qui se comporte comme printf, mais seulement quand l'option `-d` est utilisée.

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,

`i` pour ce qui a trait aux interruptions,

`s` pour ce qui a trait aux appels système.

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 `threadtest.cc`. Compilez, relancez l'exécution, observez attentivement...

Ajoutez le lancement d'un thread supplémentaire dans la fonction `ThreadTest()`. Ça marche toujours ?

Que fait la méthode `Thread::Start()` dans Nachos ? À quel moment les threads nachos sont-ils créés (mémoire allouée, structures initialisées, ...) ?

Maintenant, commentez la ligne suivante :

```
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 un certain degré de préemption par l'option `-rs <n>`. De plus, la semence passée en paramètre rend aléatoire (mais *reproductible* !) l'entrelacement des threads (le nombre `n` n'a pas de signification particulière, autre qu'être celui servant à initialiser le générateur aléatoire).

```
./nachos -rs 0
./nachos -rs 1
./nachos -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 :

```
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 `Yield()` ? Allez inspecter le source de cette fonction dans le fichier `code/thread/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 `Yield()`. Quels sont les rôles respectifs des fonctions `ReadyToRun(),FindNextToRun()` et `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 Exercice

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

Relancez le programmes Nachos dans le répertoire **`thread`** et observez le déroulement (avec l'option `-d` de Nachos et/ou avec `gdb`).

Annexe : Outils pour la lecture 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. Suivez l'une des sections suivantes selon l'éditeur que vous utiliser.

Mise en œuvre pratique pour l'éditeur `VSCode`

Commencez par faire ouvrir le répertoire `code/` par VSCode. Pas seulement un fichier, mais vraiment le répertoire lui-même, pour qu'il détermine que c'est la racine du projet.

Lorsque vous ouvrez un fichier C++, VSCode vous propose d'installer l'extension C/C++, installez-la.

Une fois cela fait, VSCode se met à analyser vos fichiers (on le voit consommer beaucoup de temps CPU). Après un certain nombre de secondes, il aura fini d'analyser tous les fichiers.

Vous pouvez alors utiliser control-clic pour atteindre la définition d'une fonction, ou F12 au clavier.

Pour revenir en arrière, utiliser Go->Back (ou mieux : définissez un raccourci clavier !)

Mise en œuvre pratique pour l'éditeur `emacs`

C'est le programme `etags` pour emacs qui construit le dictionnaire et le stocke dans le fichier `TAGS`.

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

Pour le C++, la commande `etags` sera utilisée. Pour créer le dictionnaire des fonctions de NACHOS taper :

```
cd Mon_Nachos/nachos/code
etags */*.cc */*.c */*.S */*.h
```

et vous devez voir apparaître un fichier `TAGS` dans le répertoire courant (ici `Mon_Nachos/nachos/code`).

Pour l'utiliser lancer `emacs threads/main.cc` par exemple, et pour chercher la définition d'une fonction, placer le curseur sur son nom et taper `Meta .` ("Méta point", la touche Meta est appelée "Alt" sur vos claviers, et le plus simple est d'utiliser le point du pavé numérique). emacs demande confirmation du nom de fonction à chercher, il suffit de valider, ou bien taper à la main le nom d'une autre fonction. Lors de la première recherche d'un tag, emacs demande ensuite l'emplacement du fichier dictionnaire TAGS, ici c'est `Mon_Nachos/nachos/code/`. Pour revenir à l'endroit d'où l'on venait, taper `Meta ,`

Note : il arrive qu'il existe plusieurs définitions pour un même nom. Pour atteindre la définition « suivante », utiliser `Control u Meta .`

Note 2 : Il peut être utile de préciser directement la classe à la main en tapant son nom complet après `Meta .` , par exemple `Meta . Thread::setStatus`

Note 3 : lorsque vous ajouterez vos propres fonctions il faudra relancer `etags`

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 Mon_Nachos/nachos/code
grep -r ``currentThread'' */*.cc */*.h */*.c
```

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

Mise en œuvre pratique pour l'éditeur `vim`

Pour ceux qui préfèrent `vim` à emacs, vous pouvez utiliser `ctags` qui crée le fichier `tags`)

Pour créer le dictionnaire des fonctions de NACHOS taper :

```
cd ~/Systeme/Nachos/nachos/code
ctags */*.cc */*.c */*.S */*.h
```

et vous devez voir apparaître un fichier `tags` dans le répertoire courant (ici `~/Systeme/Nachos/nachos/code`)

Commandes élémentaires

Commande	Résultat
`: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*
`:tn`	Va à la définition suivante de la dernière utilisation de `:ta`
`: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-]` (oui, il faut taper sur ctrl, altgr, et la touche `)`. De même,

`:pop`

admet `Ctrl-t` comme abréviation.