# Nachos 5.0j Tutorial

Rong Zheng*, Ala Shaabana and Qiang Xu
Dept. of Computing and Software
McMaster University
Hamilton, ON, Canada
{*rzheng,shaabaa,xuq22*}@mcmaster.ca

March 3, 2015

## Contents



*Contact author

## Disclaimer

This document synthesizes and extends relevant materials from the web on Nachos 5.0j, in particular, Narten's "A roadmap through nachos, Hettena and Cox's guide to Nachos 5.0j and Nachos C++ roadmaps by Qiao. The goal is to provide a one-stop place for Nacho 5.0j for students and instructors. In addition to discussing the internals of Nachos, we also provide code tracing examples and exercises.

## How to Use this Document

Full understanding of this document requires knowledge in Operating Systems. Therefore, we suggest you read relevant sections of this document as you progress along with the course materials and project assignments. Suggested readings will be specified in class lectures and individual project description.

# 1 Introduction

Nachos is an instructional software that allows students to study and modify a real operating system. It was originally developed in C++ by researchers at the University of California, Berkeley and was later port to Java.

Nachos simulates a machine that roughly approximates the MIPS architecture with registers, memory and a CPU. It also simulates the general low-level facilities of typical machines, including interrupts, virtual memory and interrupt-driven device I/O. Similar to a real OS, Nachos supports two types of processes, namely, kernel processes and user level processes.

To use Nachos 5.0j, one is expected to be proficient with Java programming. Knowledge in generic types, exception handling, abstract class/interface will be helpful in understanding and implementing new modules in Nachos. Interested users can refer to online Java tutorials and Java API documentation.

# 2 Installation and Execution of Nachos 5.0j

We will now cover the Nachos installation and execution procedure. For complete installation, one needs to install both Nachos and a suitable cross-compiler on the target platform. Note that if one does not wish to go through the installation procedure, a Virtual Machine containing a 32-bit version of Kubuntu with Nachos and the MIPS cross-compiler installed can be downloaded here (username/passwd as 3sh3/3sh3 in lower case). In this case, you will need a virtual machine manager (VMM), also known as a hypervisor. Virtualbox is a free VMM one can use to run the virtual machine. For more information about Virtual Machines can be found here.

If one opts to install Nachos by oneself, keep in mind that the MIPS cross-compiler does not work properly on 64-bit Linux systems and Mac OSX based on our experience. MIPS cross compilers for 32-bit platforms can be found here.

## 2.1 System requirements

Nachos 5.0j requires Java SE Java Development Kit 1.5 or later. To find out your version of Java, run `java -version` in the command prompt. The most up-to-date version of JDK and the JRE can be found here.

Next, we discuss the procedure of installing Nachos on Windows, Linux, and Mac OSX.

## 2.2 Nachos Installation

### 2.2.1 Windows Installation

1. Install Cygwin. Cygwin is a Windows program that emulates Unix commands and processes. In order to install and run Nachos, we must do it in this environment. Be sure to download the 32-bit version of Cygwin since the Windows MIPS cross-compiler is 32-bit only.

2. During Cygwin installation, you will be prompted to select custom packages to install or skip. Search for `make` and `gcc` and include them for installation, see Figures minipage1 and minipage2 for clarification. Once the installation is finished, make sure to run Cygwin before proceeding to the next steps in order for it to initialize its folders properly.

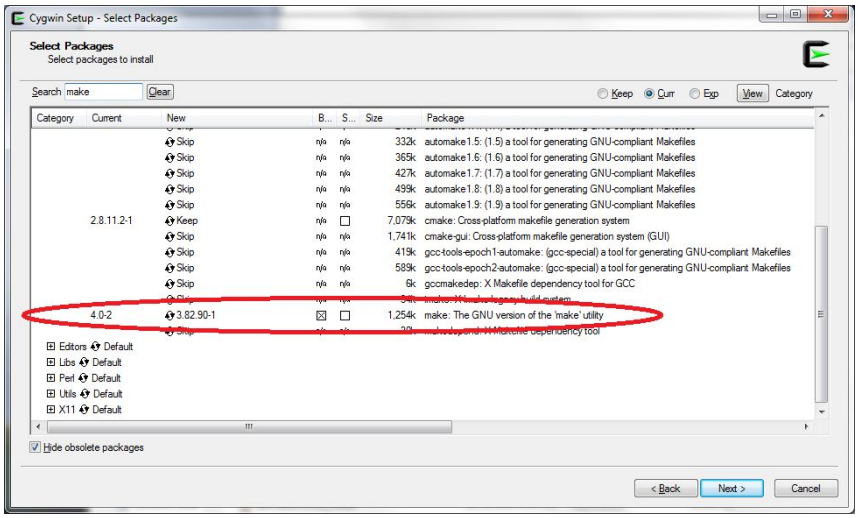


Figure 1: Add Make to your installation

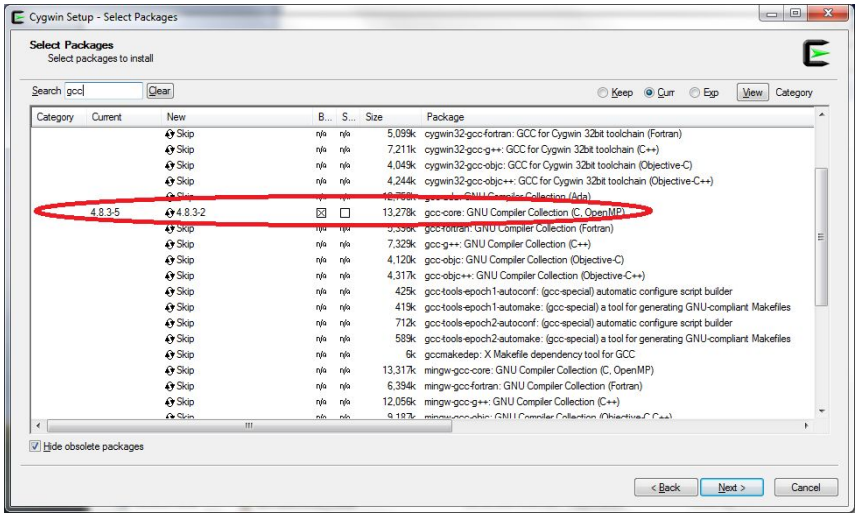


Figure 2: Add gcc to your installation

3. Download Nachos here and save it to a directory that is easily accessible. Since you are using Cygwin, it is recommended that you save it to your Cygwin home directory (See the address bar in Figure figure3).

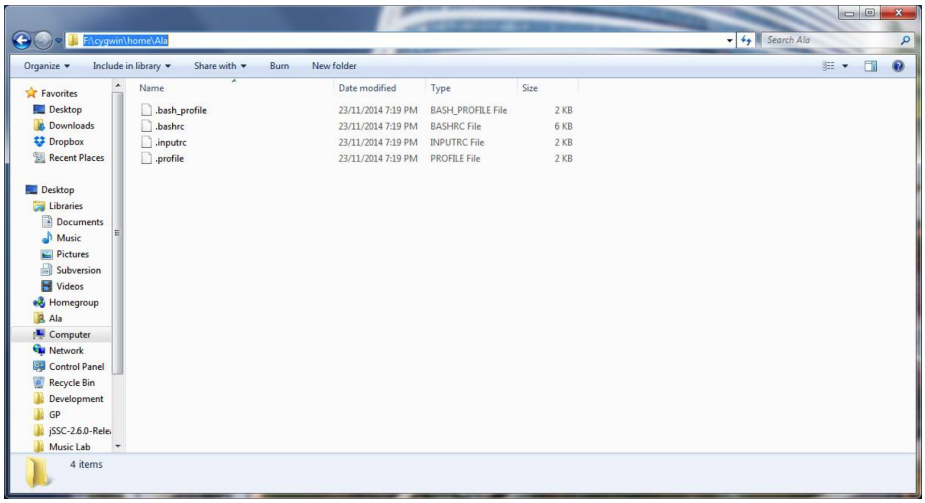


Figure 3: Your empty user directory on Cygwin

4. Extract Nachos by running `tar -zxvf nachos.tar.gz` in the directory it was saved.

5. Now we must add the JDK to the Cygwin path. Navigate to your home directory in Cygwin and look for `.bash_profile` and open it with the text editor of your choice.

6. At the bottom of the file, add the path to your jdk bin folder to the path variable, like in Figure minipage4.

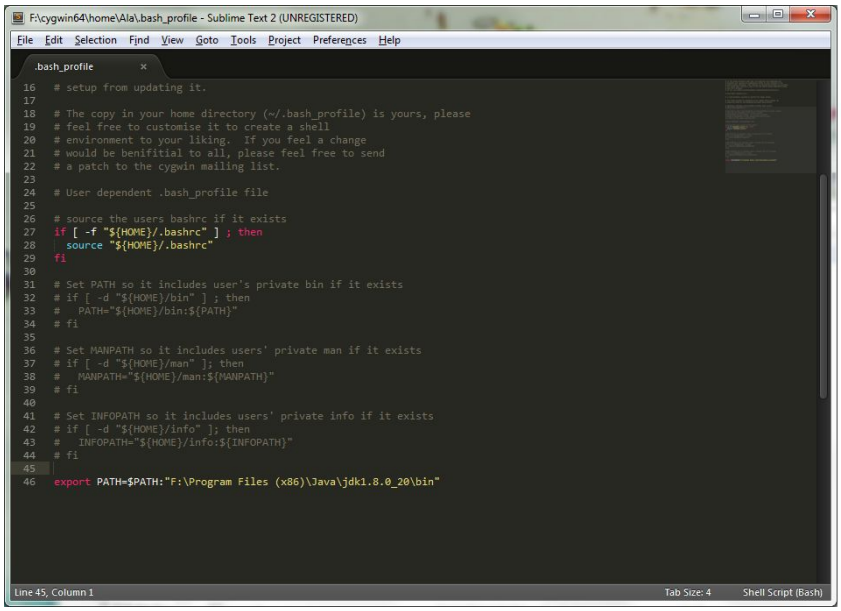


Figure 4: Add jdk/bin to your path variable

7. Run `javac` in Cygwin. If you get a "command not found" error, then double-check your path.

8. We can now compile. Navigate to the `Proj1` directory and run `make`.

9. Once you have compiled proj1, you are now ready to run it. Run `../bin/nachos`, your output should look something like Figure minipage5 if everything was installed correctly.

10. You can add Nachos to your path by including the nachos/bin directory inside of the path export line in `.bash_profile` following the same procedure in Step 6.

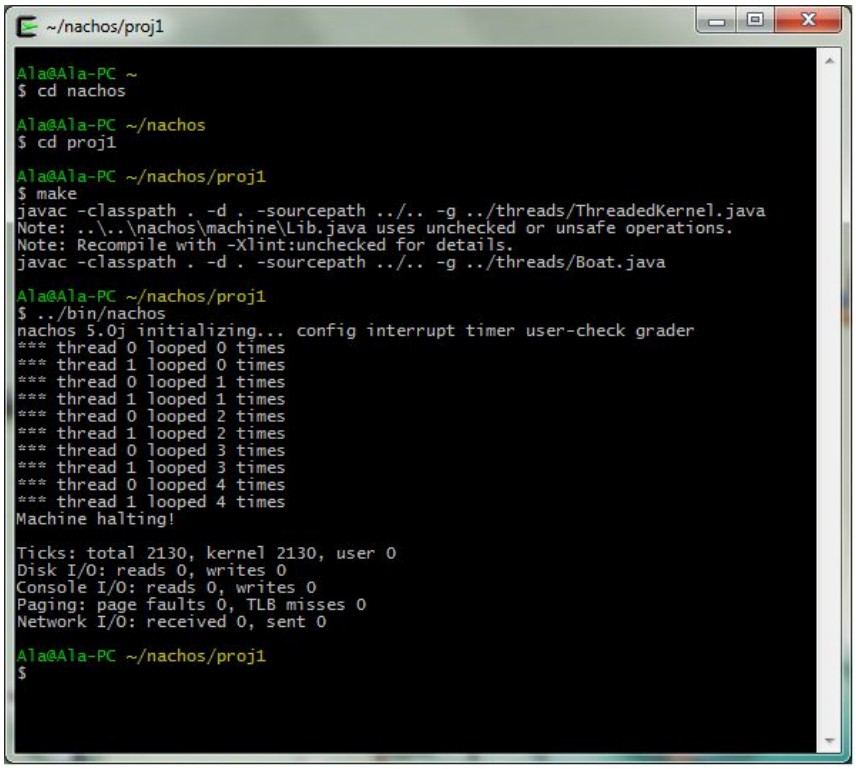


Figure 5: Nachos output in Proj1

11. Nachos is now ready to go! To install the cross-compiler on Cygwin, please proceed to Section subsection2.3.

### 2.2.2 Linux Installation

The Linux installation is largely similar to the Cygwin installation, with the main difference being the bash script editing.

1. Download and unzip Nachos into a directory of your choice using `tar -zxvf nachos-java.tar.gz`.

2. Ensure that you have Java 1.5 or above installed by running `java -version`. If you do not have Java installed, you can download it from the Oracle website here.

3. Once you have Java working, then navigate to nachos/proj1 and run *make*.

4. To make things easier for ourselves, we will add the nachos executable to our PATH variable so that it is easier to run. Open `~/.bashrc` in your favourite text editor.

5. Add the absolute path to the `bin` folder inside Nachos to the path variable by appending the line `export PATH=$PATH:path_to_nachos_bin` to the end of the file, similarly to Figure minipage6.

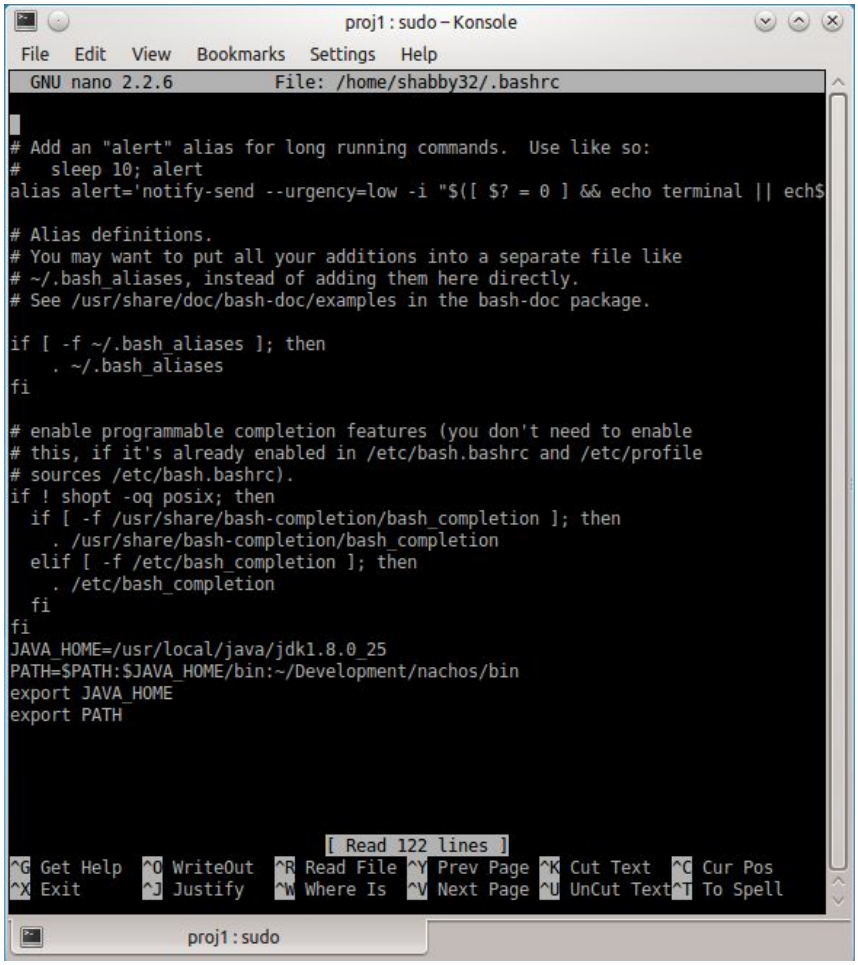


Figure 6: .bashrc contents in Kubuntu

6. We can now try running Nachos. Navigate to the `proj1` folder inside Nachos, and run `nachos`. You should have an output similar to Figure minipage7.

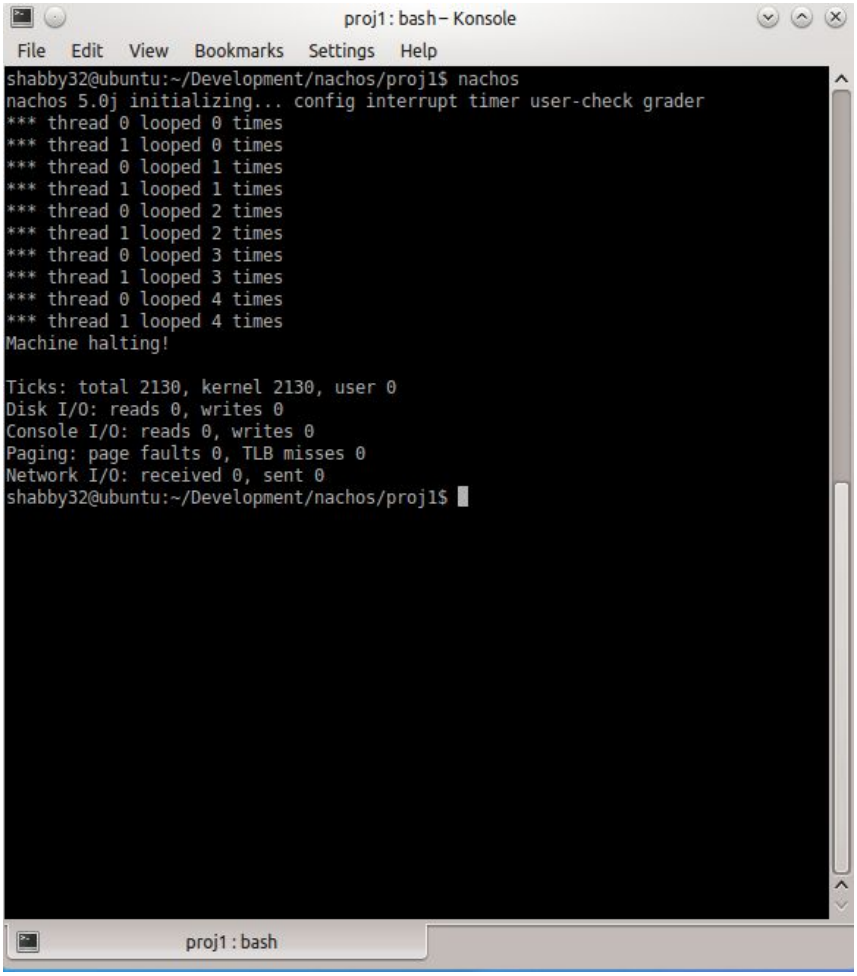


Figure 7: Proj1 run in Kubuntu

**NOTE:** If you encounter the error "Unsupported major.minor version error", you must remove the other Java version and all references to it using the following commands:

1. Update your repository first: `sudo apt-get update`
2. Remove all Oracle and/or Java related files: `sudo apt-cache search java | awk 'print($1)' | grep -E -e '^(ia32-)?(sun|oracle)-java' -e '^openjdk-' -e '^icedtea' -e '^(default|gcj)-j(re|dk)' -e 'ĝcj-(.*)-j(re|dk)' -e 'java-common' | xargs sudo apt-get -y remove`
3. `sudo apt-get -y autoremove`
4. Purge all config files: `dpkg -l | grep ^rc | awk 'print($2)' | xargs sudo apt-get -y purge`
5. Remove Java config and cache directory: `sudo bash -c 'ls -d /home/*/.java' | xargs sudo rm -rf`
6. Remove manually installed Java Virtual Machines (JVM): `sudo rm -rf /usr/lib/jvm/*`

### 2.2.3 Mac OS X Installation

Mac OS X installation is similar to the steps for Linux with the only difference that the path variable should be modified in the file `.bash_profile` in your home directory.

## 2.3 Cross-compiler Installation

Nachos simulates a machine with a processor that roughly approximates the MIPS architecture. The simulated MIPS processor can execute arbitrary user programs. Nachos has two modes of execution, one of which is the MIPS simulator. The second mode corresponds to the Nachos "kernel". In MIPS simulator mode, a MIPS cross compiler is required to compile user programs written in C into COFF binary to be executed in Nachos. One can find many user programs in `nachos/test` directory, which is the default directory to store them.

An instructional machine with MIPS compiler pre-installed will be provided; if you are not using an instructional machine, you must download and install the appropriate cross-compiler from here. Before you start downloading, you need to check the architecture of your system.MAKE SURE you download the correct one as this is crucial. This can be accomplished by typing the command (In Linux and Mac OSX): `uname -a`. Figure minipage8 and minipage8 show the output of `uname -a` on Linux and Mac OSX respectively.

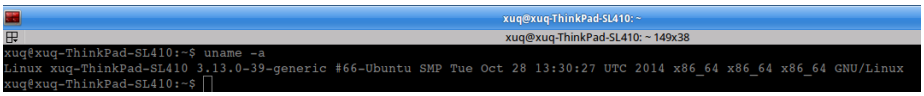


Figure 8: Architecture information in Ubuntu

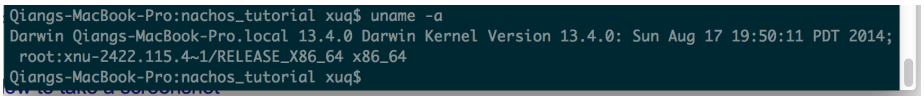


Figure 9: Architecture information in Mac OSX

The work flow of running user programs in Nachos is depicted in Figure mini-page10.

> **Implementation note:** Since a user program is written in C, one can in fact compile using GCC with minor modifications. To do so, one needs to replace the `#include` statements with proper header files (e.g, replacing `#include "stdio.h"` with `#include <stdio.h>`).

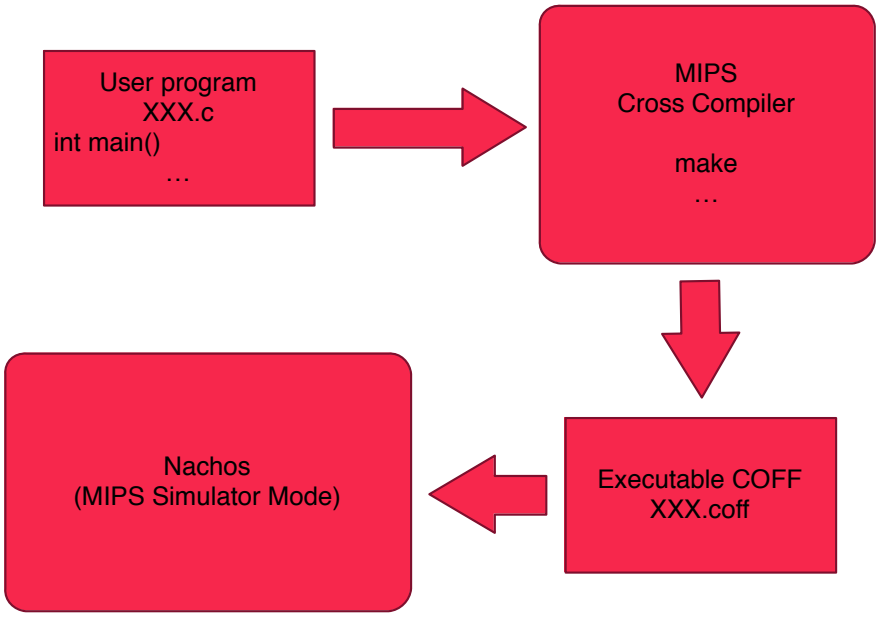


Figure 10: The relationship between MIPS and Nachos

### 2.3.1 Linux Installation

We highly recommend to use 32-bit Linux to install MIPS compiler. The installation is straightforward. Let us take 32-bit Ubuntu 12.04 LTS as an example to illustrate the detailed procedure of MIPS compiler installation.

1. Download an architecture compatible MIPS package from the link. Here,

you need to select `mips-x86.linux-xgcc.tar.gz`.

2. Extract the files: `tar -xzvf mips-x86.linux-xgcc.tar.gz`. The location of this folder will be used in the following step. You can go to this folder and get the absolute path of this folder:
   ```
   cd mips-x86.linux-xgcc/
   pwd
   ```

3. Set environment variable ARCHDIR. The configuration of environment variable depends on the local OS and shell version. Typically, you need to edit file `.bashrc` in your HOME directory. This is a hidden file which is not visible. To see this file, you need to use the command `ls -a`. Open the file with your preferred editor and add two new lines:
   ```
   export ARCHDIR=Your mips cp dir
   export PATH=$ARCHDIR:Your nachos bin dir:$PATH
   ```
   Then save and quit the editor. Note that, **to make this configuration effective, you need to restart your TERMINAL(Not Machine)**.

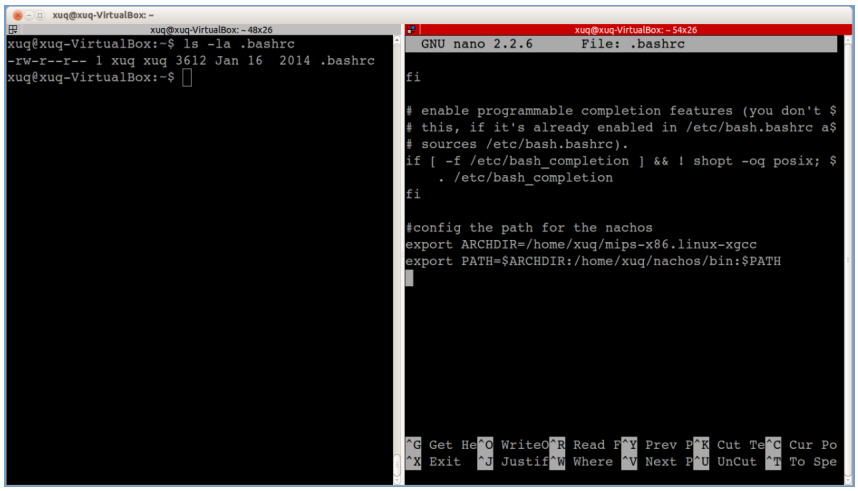


Figure 11: The configuration of environmental variables

4. To this end, the configuration of MIPS is finished. Now we need to test it works. Navigate to `nachos/test`, and run `make clean`. Then run `make`. Proceed to the next step if there are no errors produced here. If you run into errors such as "unrecognized flag" then check your ARCHDIR and where you appended it to the PATH variable.

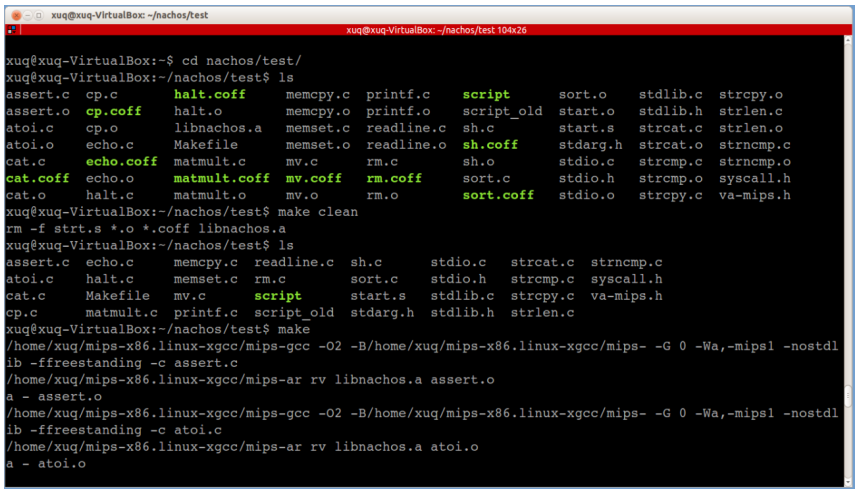


Figure 12: Test configuration of MIPS compiler

5. Navigate to `nachos/proj2` and compile it with *make*, then run `nachos -d ac -x halt.coff`. This allows us to run nachos with a user program (halt.c, translated using the MIPS compiler to halt.coff) as input. Your output should look something like Figure minipage13.

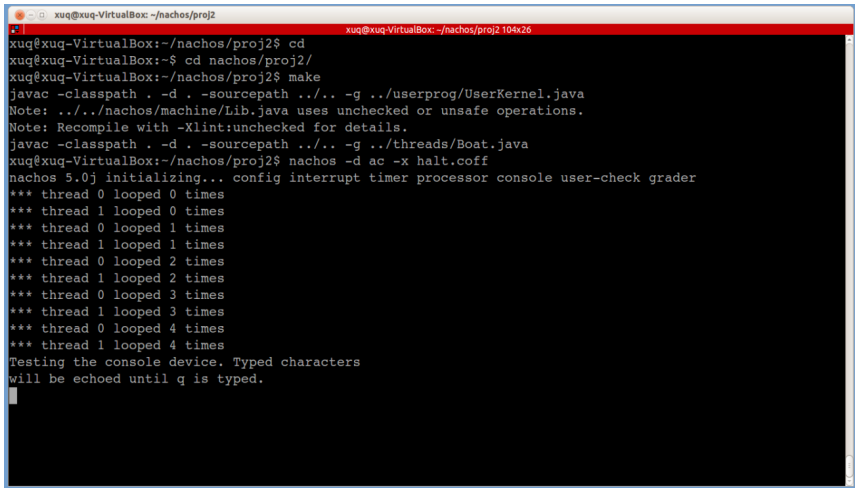


Figure 13: Test the compiled COFF file

### 2.3.2 Mac OSX Installation

The installation of MIPS compiler in Mac OSX are almost identical. The differences are listed as follows:

1. The environment variable configuration file in Mac OSX system is `.bash_profile` rather than `.bashrc`.

2. You should download a different MIPS installation package.

There is a good chance that you will see an error as shown in Figure mini-page14.

```
Qiangs-MacBook-Pro:test xuq$ make
/Users/xuq/mips/bin/mips-gcc -O2 -B/Users/xuq/mips/bin/mips- -G 0 -Wa,-mips1 -nostdlib -ffreestanding -c assert.c
dyld: Library not loaded: /usr/local/lib/libmpc.3.dylib
  Referenced from: /Users/xuq/mips/bin/../libexec/gcc/mipsel-elf/4.8.1/cc1
  Reason: image not found
mips-gcc: internal compiler error: Trace/BPT trap: 5 (program cc1)
libbacktrace could not find executable to open
Please submit a full bug report,
with preprocessed source if appropriate.
See <http://gcc.gnu.org/bugs.html> for instructions.
make: *** [assert.o] Error 4
Qiangs-MacBook-Pro:test xuq$ ls /usr/local/lib/
```

Figure 14: A common error when running MIPS on Mac OSX

This error occurs because you do not have `libmpc` lib installed. To install this library, follow the following steps:

**Step 1** Install the Xcode Command Lines. If you have Xcode installed, use the Download tab in Xcode Preferences to install the Command Line Tools. If you do not have Xcode installed, you can download the Command Line Tools for free from the Apple Developer website.

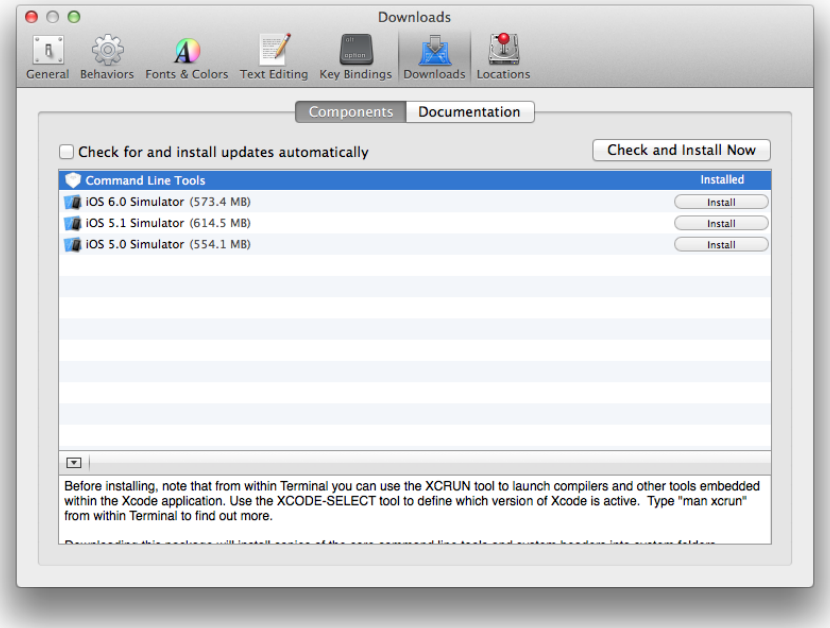


Figure 15: Install command line tools in Xcode

**Step 2** Install GNU Multiprecision library (aka `libmpc`). We recommend that you use the brew package manager to compile and install libmpc. Home-

brew offers a very clean and simple way to install command line tools and libraries on a Mac. To install it. It should be as simple as running:
`ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go/install)"`
To check whether `Homebrew` is installed successfully, you can run command: `brew doctor`. If it is installed successfully, you will see something like Figure minipage16.

```
Qiangs-MacBook-Pro:~ xuq$ brew doctor
Warning: Some directories in /usr/local/share/man aren't writable.
This can happen if you "sudo make install" software that isn't managed
by Homebrew. If a brew tries to add locale information to one of these
directories, then the install will fail during the link step.
You should probably `chown` them:

    /usr/local/share/man/de
    /usr/local/share/man/de/man1

Warning: Broken symlinks were found. Remove them with `brew prune`:
  /usr/local/bin/texdist

Warning: You have MacPorts or Fink installed:
  /opt/local/bin/port

This can cause trouble. You don't have to uninstall them, but you may want to
temporarily move them out of the way, e.g.

  sudo mv /opt/local ~/macports
```

Figure 16: Check whether Homebrew is installed successfully

**Step 3** Install `libmpc`. Use command `brew install libmpc`. And to make sure that the library is correctly installed, you can run command: `ls /usr/local/lib/libmpc.3.dylib`.

**Step 4** Go to `nachos/test` to `make` again. The compilation should be successful as shown Figure minipage17.

```
Qiangs-MacBook-Pro:test xuq$ make
/Users/xuq/mips/bin/mips-gcc -O2 -B/Users/xuq/mips/bin/mips- -G 0 -Wa,-mips1 -nostdlib -ffreestanding -c assert.c
/Users/xuq/mips/bin/mips-ar rv libnachos.a assert.o
r - assert.o
/Users/xuq/mips/bin/mips-gcc -O2 -B/Users/xuq/mips/bin/mips- -G 0 -Wa,-mips1 -nostdlib -ffreestanding -c atoi.c
/Users/xuq/mips/bin/mips-ar rv libnachos.a atoi.o
r - atoi.o
/Users/xuq/mips/bin/mips-ar rv libnachos.a printf.o
r - printf.o
/Users/xuq/mips/bin/mips-ar rv libnachos.a readline.o
```

Figure 17: Run MIPS compiler

### 2.3.3 Windows Installation

We do NOT recommend to use Windows as developing environment for Nachos. We've encountered many errors while executing MIPS in Windows (Typically in Cygwin). That being said, the basic steps of configuration of MIPS in Cygwin is described as follows. Please use with caution.

1. Run `tar -zxvf your_mips_tar` to extract the folder. It is recommended to keep your MIPS folder in the same directory as the Nachos directory for ease of access.

2. Now we must add the MIPS directory to the PATH variable the same way we added the JDK. To do so, first define an ARCHDIR variable that contains the path to the extracted MIPS folder, and then append this variable to the PATH variable. MAKE SURE that ARCHDIR is appended to the path variable before anything else (Figure minipage18). Note that if the folder is placed in your Cygwin home directory, you can use the $HOME variable in `.bash_profile`.

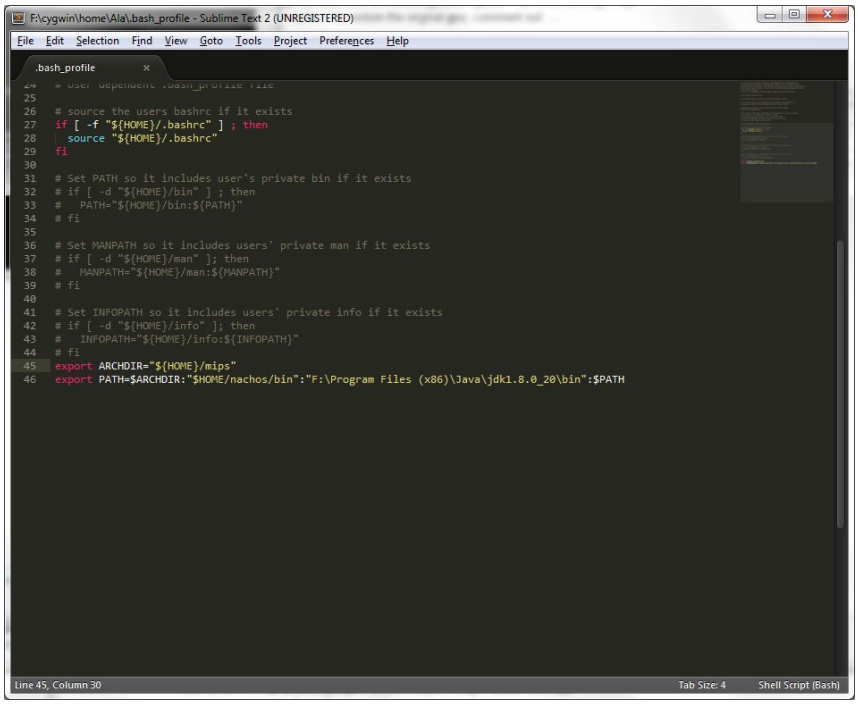


Figure 18: Adding ARCHDIR to the PATH variable

3. Navigate to `nachos/test`, and run `make`. If you run into errors such as "unrecognized flag" then check ARCHDIR and if the path variable has been set properly.

**Tips**

1. Nachos accepts COFF binary files produced by any MIPS compiler be it installed on Linux, Windows or Mac OSX. Therefore, you can compile your user program on the instructional machine and copy to your test directory for execution.

2. Errors in compiling user programs typically come from incompatibility between the compiler and the machine. For example, you may have installed 64-bit MIPS compiler in a 32-bit machine. Make sure you have the right MIPS compiler for your local environment.

## 2.4 Organization of Nachos 5.0j sources

The Nachos API Javadoc can be found HERE. Alternatively, you can produce your own Javadoc by following the instruction in command line or through eclipse. This documentation is very important for you to understand the code structure of Nachos. Nachos 5.0j is composed of 7 main packages as summarized in Table table1.

Table 1: The packages in Nachos

| Packages | |
|---|---|
| `nachos.ag` | Provides classes that can be used to automatically grade Nachos projects. |
| `nachos.machine` | Provides classes that implement the Nachos simulated machine. The key components of Nachos machine are implemented here. |
| `nachos.network` | Provides classes that allow Nachos processes to communicate over the network. |
| `nachos.security` | Provides classes that can be used to protect the host system from malicious Nachos kernels. |
| `nachos.threads` | Provides classes that support a multithreaded kernel. The logic of thread scheduling will be implemented here. |
| `nachos.userprog` | Provides classes that allow Nachos to load and execute single-threaded user programs in separate address spaces. |
| `nachos.vm` | Provides classes that allow Nachos processes to be demand paged, and to use a hardware TLB for address translation. |

### 2.4.1 `nachos.machine`

Among these packages, `nachos.machine` is the most important one. The classes in this package implement the simulated machine, important abstract classes and data structures. The main entry point of the Nachos is also contained

in this package, more specifically, in `nachos.machine.Machine`. Next, we will describe some important classes in `nachos.machine`. For a complete reference of the package, interested reader can check the Nachos Javadoc.

Table 2: Main classes in `nachos.machine`

| Class Summary | |
|---|---|
| ArrayFile | A read-only OpenFile backed by a byte array. |
| Coff | A COFF (common object file format) loader. This class combining with `CoffSection` define the internal structures of user program in Nachos. |
| CoffSection | A CoffSection manages a single section within a COFF executable. |
| Config | Provides routines to access the Nachos configuration. To understand how Nachos parse configure file, going through this class is a good idea. |
| Kernel | An OS kernel. This class is the superclass of `nachos.threads.ThreadedKernel`. |
| Lib | Provides miscellaneous library routines. |
| Machine | The master class of the simulated machine. It also contains the main entry of Nachos. |
| OpenFile | A file that supports reading, writing, and seeking. |
| StubFileSystem | This class implements a file system that redirects all requests to the host operating system's file system. |
| TCB | A TCB simulates the low-level details necessary to create, context-switch, and destroy Nachos threads. |
| TranslationEntry | A single translation between a virtual page and a physical page. Note that TLB entries are of type `TranslationEntry`, the same class used for page table entries. |

### 2.4.2 Others

Since `nachos.machine` implements the simulated Nachos machine, it should remain intact. Modification to Nachos is mostly limited to other packages. For example, the thread related projects will only involve changes to package `nachos.threads`.

According to the project requirement, you only need to focus on the specific packages. In the programming assignment, we will deal with `nachos.threads`, `nachos.userprog`, and `nachos.vm`. See Table table3– table5 for the classes in each package.

Table 3: The class description in `nachos.threads`

| Class Summary | |
|---|---|
| Alarm | Uses the hardware timer to provide preemption, and to allow threads to sleep until a certain time. |
| Communicator | A communicator allows threads to synchronously exchange 32-bit messages. |
| Condition | An implementation of condition variables built upon semaphores. |
| Condition2 | An implementation of condition variables that disables interrupt()s for synchronization. |
| ElevatorController | A controller for all the elevators in an elevator bank. |
| KThread | A KThread is a thread that can be used to execute Nachos kernel code. |
| Lock | A Lock is a synchronization primitive that has two states, busy and free. |
| LotteryScheduler | A scheduler that chooses threads using a lottery. |
| PriorityScheduler | A scheduler that chooses threads based on their priorities. |
| Rider | A single rider. |
| RoundRobinScheduler | A round-robin scheduler tracks waiting threads in FIFO queues, implemented with linked lists. |
| Scheduler | Coordinates a group of thread queues of the same kind. |
| Semaphore | A Semaphore is a synchronization primitive with an unsigned value. |
| SynchList | A synchronized queue. |
| ThreadedKernel | A multi-threaded OS kernel. |
| ThreadQueue | Schedules access to some sort of resource with limited access constraints. |

Table 4: The class description of `nachos.userprog`

| Class Summary | |
|---|---|
| SynchConsole | Provides a simple, synchronized interface to the machine's console. |
| UserKernel | A kernel that can support multiple user processes. |
| UserProcess | Encapsulates the state of a user process that is not contained in its user thread (or threads). |
| UThread | A UThread is KThread that can execute user program code inside a user process, in addition to Nachos kernel code. |

Table 5: The class description of `nachos.vm`

| Class Summary | |
|---|---|
| VMKernel | A kernel that can support multiple demand-paging user processes. |

## 2.5 Execution

The typical work flow of Nachos projects is given in Figure minipage19. Most of Nachos projects require modification of Java codes. For the purpose of testing, you may either write and use Java classes derived from the `AutoGrader` class, or run user programs compiled as COFF binary.

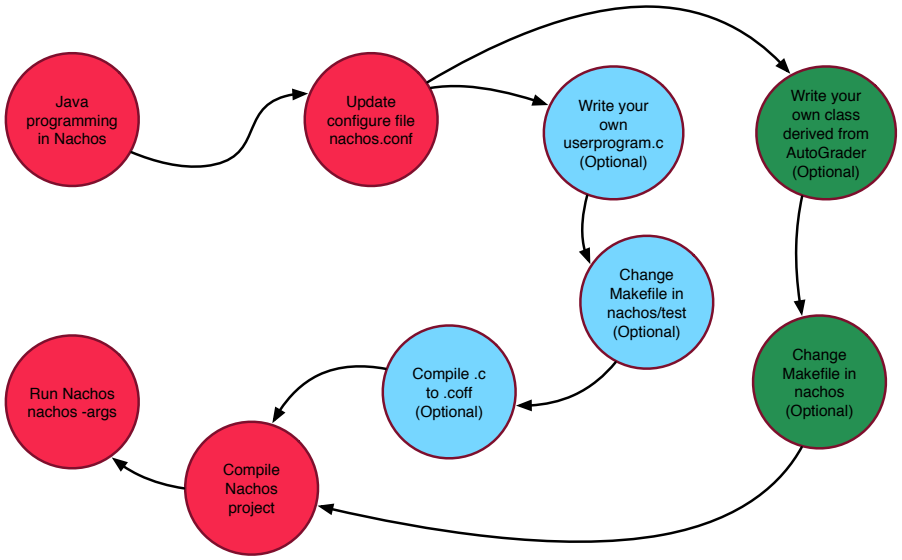


Figure 19: The workflow for Nachos projects

### 2.5.1 Nachos configure file

When Nachos starts, it reads the default configure file `nachos.conf` from the current directory (You can explicitly specify a different configure file by `nachos -[]`). It contains a list of key-value pairs, in the format "key = value" with one pair per line. One can modify the configure file to change the default scheduler, default shell program, to change the amount of memory the simulator provides, or to reduce network reliability. An exmaple configure file is given in Figure minipage20. In different projects, the students may be required to change the settings in `nachos.conf`. For example, for thread related projects, `Kernel.kernel` should be set to `nachos.threads.ThreadedKernel`. The detailed options in the configure file are discussed in Table table6.

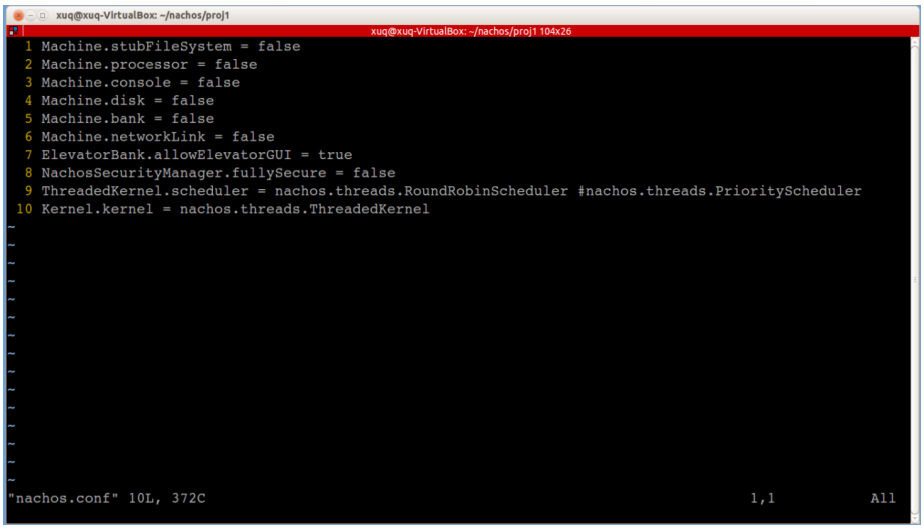


Figure 20: An example configure file

### 2.5.2 Makefile

The Makefile of Nachos allows compiling Nachos in command line. If you are using Eclipse or Netbeans as IDE, you will not make use of Makefile. Two Makefiles are relevant to your project. The first one is under a project directory, e.g, `nachos/proj1` and the second one is in the root directory of Nachos.

The Makefile under the project directory is usually very simple. An example is given below:

```
1 DIR = XXX XXX
2 include ../Makefile
```

In first line, variable `DIRS` specifies the packages to be compiled. The second line includes the main Makefile for Nachos. Generally, writing Makefile is rather complicated. A full description of Make and Makefile can be found here. Fortunately, the main Nachos Makefile has been provided to you already under the root Nachos directory. Generally, there is no need to modify this file UNLESS new classes shall be included in Nachos projects. For example, to use customized AutoGrader-derived classes, you can modify the corresponding line in the Makefile:

```
1 ag = AutoGrader BoatGrader BasicTestGrader.java ThreadGrader1.java
```

### 2.5.3 Command line options

To debug and test Nachos project, you can flexibly use Nachos command line arguments. Running `nachos -h` gives the collection of Nachos command line

| | |
|---|---|
| `Machine.stubFileSystem` | Specifies whether the machine should provide a stub file system. This option should be enabled when reading and writing files on a local disk (e.g., for user programs) |
| `Machine.processor` | Specifies whether the machine should provide a MIPS processor. |
| `Machine.console` | Specifies whether the machine should provide a console. This option should be set true if users need to interact with Nachos through a console (as required by some user programs) |
| `Machine.disk` | Specifies whether the machine should provide a simulated disk. False by default |
| `ElevatorBank.allowElevatorGUI` | True by default. |
| `NachosSecurityManager.fullySecure` | False by default. When we grade, this will be true, to enable additional security checks. |
| `Kernel.kernel` | Specifies what kernel class to dynmically load. The options are `nachos.threads.ThreadedKernel`, `nachos.userprog.UserKernel`, `nachos.vm.VMKernel`, and `nachos.network.NetKernel`. |
| `Processor.usingTLB` | Specifies whether the MIPS processor provides a page table interface or a TLB interface. |
| `Processor.numPhysPages` | The number of pages of physical memory. Each page is 1KB. The default number of pages is 64. |
| `ThreadedKernel.scheduler` | The type of CPU scheduler to use. Options include `nachos.threads.RoundRobinScheduler` and `nachos.threads.PriorityScheduler` |

Table 6: Nachos configuration options

options summarized in Table table7.

Various debug options can also be specified in command line as summarized in Table table8. One can specify new debug flags and corresponding outputs by including them in the Java programs. For example, flag "a" is currently defined in `nachos/userprog/UserProcess.java` as,

```
Lib.debug(dbgProcess, "pageTable is not initialized");
...
private static final char dbgProcess = 'a';
```

It is highly advisable for you to make use of the debug options during development.

Table 7: The detailed Nachos command arguments and their functions

| | |
|---|---|
| -d | Enable some debug flags(see Table table8), e.g. -d ti |
| -h | Print this help message |
| -s | Specify the seed for the random number generator |
| -x | Specify a user program that `UserKernel.run()` should execute, instead of the default Kernel.shellProgram, e.g. `nachos -x halt.coff` |
| – | Specify an autograder class to use, instead of the default `nachos.ag.AutoGrader` |
| -# | Specify the argument string to pass to the autograder |
| -[] | Specifiy a config file to use, instead of the default `nachos.conf` |

Table 8: Nachos debug flags. To use multiple debug flags, clump them all together. For example, to monitor COFF info and process info, run `nachos -d ac`

| | |
|---|---|
| c | COFF loader info |
| i | HW interrupt controller info |
| p | processor info |
| m | disassembly |
| M | more disassembly |
| t | thread info |
| a | process info (formerly "address space"), hence a |

## 2.6 Using Eclipse with Nachos

To set up Nachos in Eclipse, we must import the files correctly. This can be a bit tricky as Eclipse has a few settings that need to be configured correctly. **Before you begin, ensure that your *Nachos* folder is inside of another folder, such as *3SH3*. If we import the *Nachos* folder itself then it will not function correctly in Eclipse**.

1. Select `File > New > Java Project`, and de-select the "use default location" option on the screen, and enter the location of your directory containing Nachos. See Figure minipage21 for clarification. Click `Finish` to proceed.

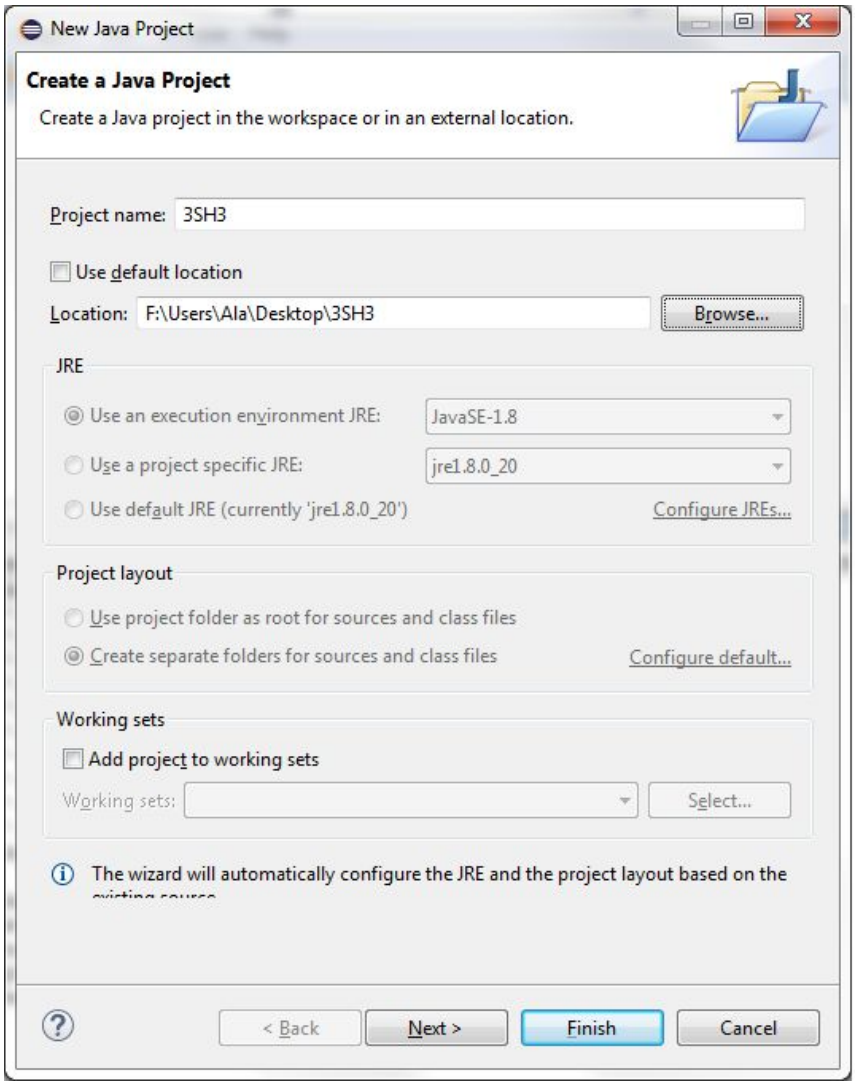


Figure 21: Path set to the parent directory of the Nachos folder

2. Your project directory should now look like Figure minipage22. Right-click the project name and go to `Run As.. > Run Configurations`.

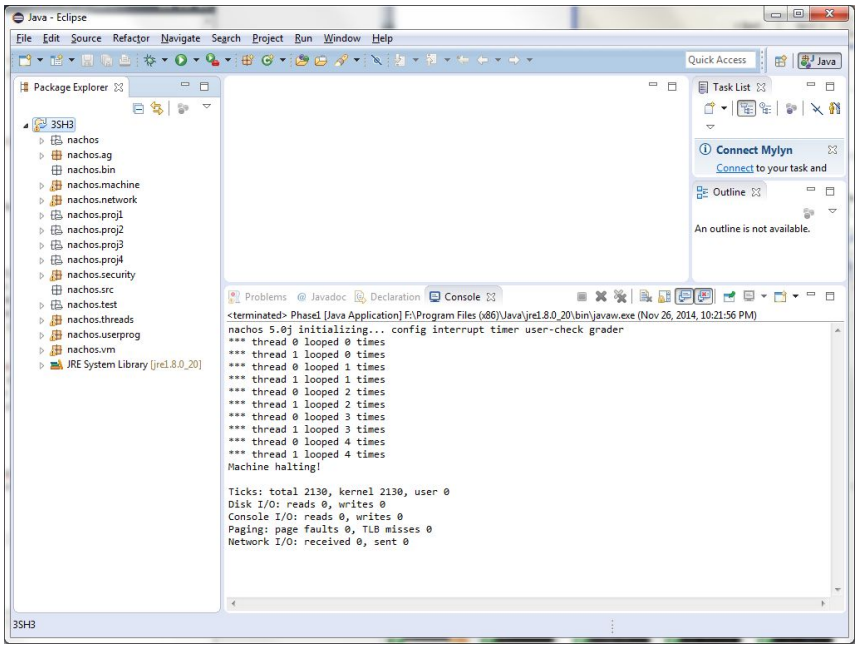


Figure 22: Eclipse Project Workspace Structure

3. Under run configurations, you will need to create a new configuration for each assignment. Generally, you only need to change the command line arguments (if they are required by the assignment) and which Nachos project directory the execution will start from. Double click `Java Application` to create a new configuration.

4. Under the "main" tab, enter `nachos.machine.Machine` in the "main class" text box, you can alternatively search for it as well. This will be the main class invoked by Nachos. See Figure minipage23.

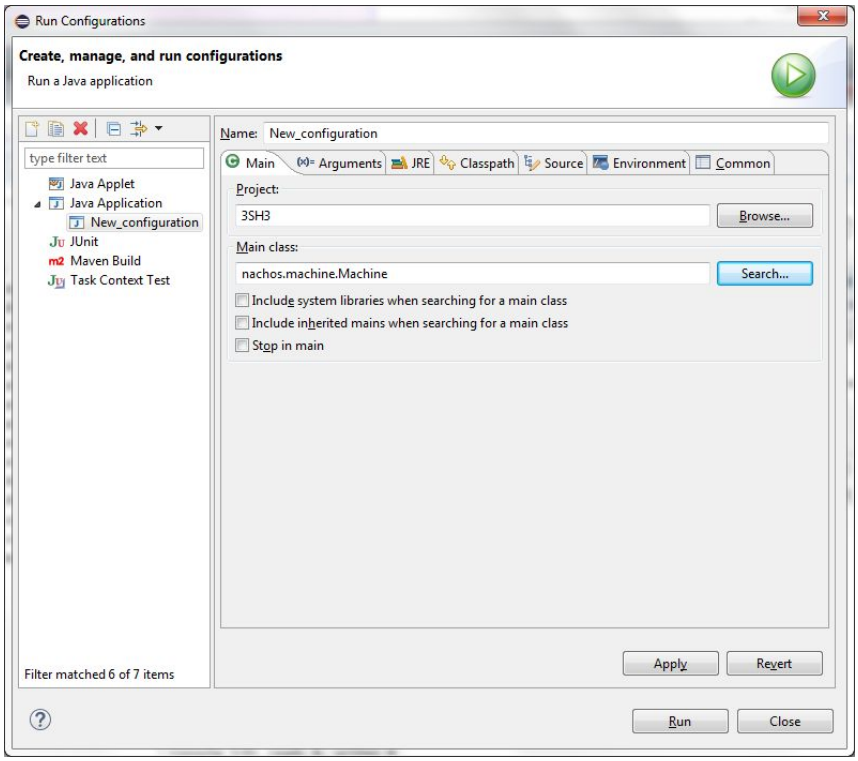


Figure 23: Main tab

5. Now go to the "Arguments" tab, and click "other" in the "working directory" box. In the text box, navigate to `proj1` and input it as the working directory. You will change this directory as the assignments progress (i.e. assignment2 will be proj2 and so on), see Figure minipage24. You can also fill in command line arguments in the top text box if required by the project.

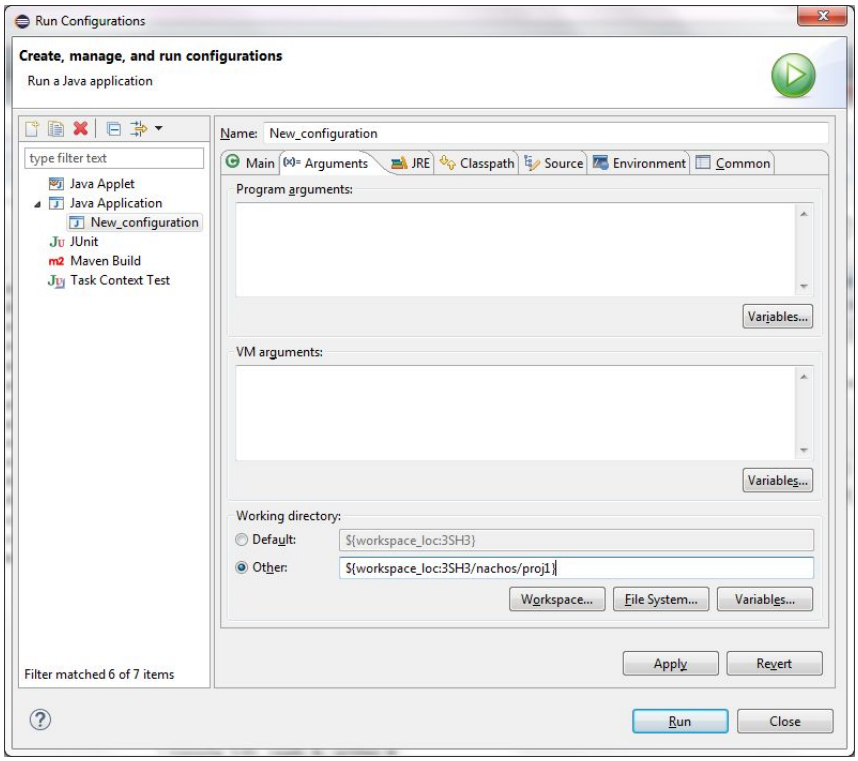


Figure 24: Arguments tab

6. Navigate to the "JRE" tab and ensure that the Runtime JRE is 1.5 or higher.

7. Run the project, you should have an output similar to Figure minipage25.

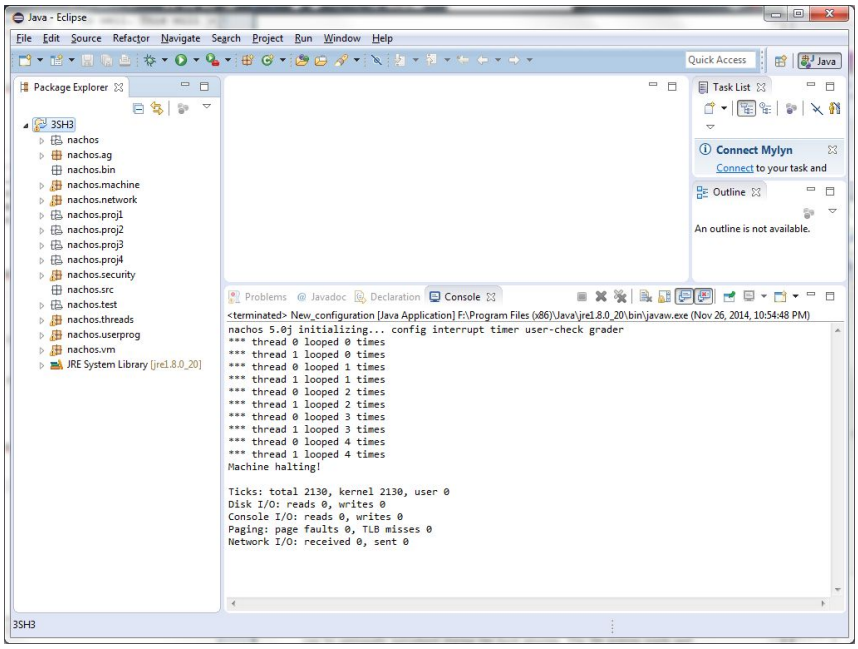


Figure 25: Running proj1 in Eclipse

**Note:** We strongly advise you to use an IDE like Eclipse as it makes debugging and code tracing much simpler.

# 3 Nachos Machine

Nachos simulates a real CPU and hardware devices, including interrupts and memory management. The Java package `nachos.machine` provides these functionalities.

## 3.1 Boot process

The entry point of Nachos is `nachos.machine.Machine.main()`. Upon calling of this method, Nachos is booted. Similar to a real machine, devices including the interrupt controller, timer, elevator controller, MIPS processor, console, network link are initialized with proper parameters specified by the config file. Unlike real machines, the current implementation does not simulate hard disks. Rather, a file system (e.g., `nachos.machine.stubFileSystem`) can be optionally initialized during the boot process. The file system reads and writes from a test directory specified in the config file, which actually locates on the hard disk of the system that Nachos runs on.

The `Machine` object then hands control to the particular `AutoGrader` in use. The `AutoGrader` then creates a Nachos kernel, starting the operating system. In the base `AutoGrader` class, autograder arguments are parsed and the kernel is initialized. To extend the `AutoGrader` class, one simply override the `run()` method.

A Nachos kernel is just a subclass of `nachos.machine.Kernel`. For instance, the thread project uses `nachos.threads.ThreadedKernel`. `UserKernel` extends `nachos.threads.ThreadedKernel` and supports multiple user processes. More details on the Nachos kernel will be discussed in Section section4.

## 3.2 Processor

The `Processor` class simulates a MIPS processor that supports a subset of the R3000 instruction set. Nachos processor lacks co-processor support.

> **Implementation note:** Nachos can not handle floating point operations.

`mainMemory` is byte-addressable and organized into 1KB pages. The actual number of pages used is controlled by the `NumPhysPages` in the config file. `readMem` and `writeMem` take virtual memory addresses as inputs to read and write the associated memory locations. Address translation is handled by the `translate()` method that translates virtual memory addresses to physical addresses Memory management will be detailed in Section section6.

The `Processor` class also allows a kernel to set an exception handler to be called on any user mode exception.

The `Processor` class maintains 38 software-accessible CPU registers including `regPC` and `regSP` for the program counter register and the stack pointer register. After loading a binary user program (in COFF) to the memory,

the PC register is initialized to point to the program entry points, and the SP register points to the top of the stack.

## 3.3 Interrupt management

The `nachos.machine.Interrupt` class simulates interrupts by maintaining an event queue together with a simulated clock. As the clock ticks, the event queue is examined to find events scheduled to take place now.

The clock is maintained entirely in software and ticks only under the following conditions:

- Every time interrupts are re-enabled (i.e. only when interrupts are disabled and get enabled again), the clock advances 10 ticks. Nachos code frequently disables and restores interrupts for mutual exclusion purposes by making explicit calls to `disable()` and `restore()`.
- Whenever the MIPS simulator executes one instruction, the clock advances one tick.

Whenever the clock advances, the event queue is examined and any pending interrupt events are serviced by invoking the device event handler associated with the event. Note that this handler is not an interrupt handler (a.k.a. interrupt service routine). Interrupt handlers are part of software, while device event handlers are part of the hardware simulation. A device event handler will invoke the software interrupt handler for the device, as we will see later. For this reason, the `Interrupt` class disables interrupts before calling a device event handler.

The Interrupt class accomplishes the above through three methods. These methods are only accessible to hardware simulation devices. `schedule()` takes a time and a device event handler as arguments, and schedules the specified handler to be called at the specified time. `tick()` advances the time by 1 tick or 10 ticks, depending on whether Nachos is in user mode or kernel mode. It is called by `setStatus()` whenever interrupts go from being disabled to being enabled, and also by `Processor.run()` after each user instruction is executed. `checkIfDue()` invokes event handlers for queued events until no more events are due to occur. It is invoked by `tick()`.

The Interrupt class also simulates the hardware interface to enable and disable interrupts via the `enable()` and `disable()` methods. These methods are useful in the thread project.

The remainder of the hardware devices present in Nachos depend on the Interrupt device. No hardware devices in Nachos create threads, thus, the only time the code in the device classes execute is due to a function call by the running `KThread` or due to an interrupt handler executed by the `Interrupt` object.

## 3.4 Timer

Nachos provides an instance of a Timer to simulate a real-time clock, generating interrupts at regular intervals. It is implemented using the event driven interrupt mechanism described above. `Machine.timer()` returns a reference to this timer.

Timer supports only two operations:

- `getTime()` returns the number of ticks since Nachos started.
- `setInterruptHandler()` sets the timer interrupt handler, which is invoked by the simulated timer approximately every `Stats.TimerTicks` ticks.

The timer can be used to provide preemption. Note however that the timer interrupts do not always occur at exactly the same intervals. Do not rely on timer interrupts being equally spaced; instead, use `getTime()`.

## 3.5 Serial console

Nachos provides three classes of I/O devices with read/write interfaces, of which the simplest is the serial console. The serial console, specified by the SerialConsole class, simulates the behavior of a serial port. It provides byte-wide read and write primitives that never block. The machine's serial console is returned by `Machine.console()`.

The read operation tests if a byte of data is ready to be returned. If so, it returns the byte immediately, and otherwise it returns -1. When another byte of data is received, a receive interrupt occurs. Only one byte can be queued at a time, so it is not possible for two receive interrupts to occur without an intervening read operation.

The write operation starts transmitting a byte of data and returns immediately. When the transmission is complete and another byte can be sent, a send interrupt occurs. If two writes occur without an intervening send interrupt, the actual data transmitted is undefined (so the kernel should always wait for a send interrupt first).

Note that the receive interrupt handler and send interrupt handler are provided by the kernel, by calling `setInterruptHandlers()`.

> **Implementation note:** in a normal Nachos session, the serial console is implemented by class `StandardConsole`, which uses `stdin` and `stdout`. It schedules a read device event every `Stats.ConsoleTime` ticks to poll stdin for another byte of data. If a byte is present, it stores it and invokes the receive interrupt handler.

## 3.6 Network link

Separate Nachos instances running on the same real-life machine can communicate with each other over a network, using the NetworkLink class. An instance of this class is returned by `Machine.networkLink()`.

The network link's interface is similar to the serial console's interface, except that instead of receiving and sending bytes at a time, the network link receives and sends packets at a time. Packets are instances of the Packet class.

Each network link has a link address, a number that uniquely identifies the link on the network. The link address is returned by `getLinkAddress()`.

A packet consists of a header and some data bytes. The header specifies the link address of the machine sending the packet (the source link address), the link address of the machine to which the packet is being sent (the destination link address), and the number of bytes of data contained in the packet. The data bytes are not analyzed by the network hardware, while the header is. When a link transmits a packet, it transmits it only to the link specified in the destination link address field of the header. Note that the source address can be forged.

The remainder of the interface to `NetworkLink` is equivalent to that of `SerialConsole`. The kernel can check for a packet by calling `receive()`, which returns null if no packet is available. Whenever a packet arrives, a receive interrupt is generated. The kernel can send a packet by calling `send()`, but it must wait for a send interrupt before attempting to send another packet.

## 3.7 Exercise

In this exercise, we trace `KThread.selfTest()` to study the boot process of Nachos. This and the next exercise will help one prepare for the thread and synchronization project. Besides manually going through the source codes, the best way to trace Nachos code is to use break points and the debug mode in Eclipse.

Let us start with the `nacho.machine.Machine.Main` method – the entry point of the boot process. The config file used is in the Project 1 directory.

```
1  Machine.stubFileSystem = false
2  Machine.processor = false
3  Machine.console = false
4  Machine.disk = false
5  Machine.bank = false
6  Machine.networkLink = false
7  ElevatorBank.allowElevatorGUI = true
8  NachosSecurityManager.fullySecure = false
9  ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler
10 Kernel.kernel = nachos.threads.ThreadedKernel
```

The `nacho.machine.Machine.Main` method is given below:

```
1         public static void main(final String[] args) {
2                 ...
3                 processArgs();
4
5                 Config.load(configFileName);
6
7                 ...
8                 createDevices();
9
10                ...
11                autoGrader = (AutoGrader) Lib.constructObject(autoGraderClassName);
12
13                new TCB().start(new Runnable() {
14                        public void run() {
15                                autoGrader.start(privilege);
16                        }
17                });
18        }
```

In Line 3, `processArgs()` processes all command lines arguments. Line 5 reads parameters specified in the configure file. In Line 8, devices are initialized including interrupt, timer, and elevator bank, processor, console, file system and network link as specified by the config file[1]. The number of pages in the main memory is specified to create the processor.

**Question:**

1. Using the config file given above, what is the number of physical pages in the main memory?
2. Why is not any processor object created in this example? How can Nachos simulate a machine without any processor?

In Line 13 – 17 of the `nacho.machine.Machine.Main` method, a new thread control block object `TCB` is created with a `Runnable` object, which if run will create a new Java thread and execute the `autoGrader`'s `start` method with a specific privilege.

```
1         public void nachos.ag.AutoGrader.start(Privilege privilege) {
2                 Lib.assert(this.privilege == null, "start() called multiple times");
3                 this.privilege = privilege;
4
5                 String[] args = Machine.getCommandLineArguments();
6
7                 extractArguments(args);
```

[1] In this example, only interrupt and timer objects are created.

```
8
9                   ...
10
11                  init();
12
13                  ...
14                  kernel = (Kernel) Lib
15                                  .constructObject(Config.getString("Kernel.kernel"));
16                  kernel.initialize(args);
17
18                  run();
19          }
```

In `AutoGrader.start`, Line 7 extracts command line inputs specific to the autograder (See Section section2). Further initialization codes can be included by overriding `init()` called in Line 11. Finally, the specific kernel object is created and initialized. Since in the config file we choose `nachos.threads.ThreadedKernel`, its respective `initialize` method will be called. The method creates a scheduler, the first thread, and an alarm, and enables interrupts. It creates a file system if necessary. Up until the creation of the first thread (Line 16 of `nachos.thread.ThreadedKernel.initialize`), like any other single-threaded Java program, Nachos code is executing on the initial Java thread created automatically for it by Java. Afterwards, Nachos manages its own thread for kernel or user processes. Threads are the basic unit of execution. More detailed discussion on Nachos threads can be found in Section section4.

```
1           public void nachos.thread.ThreadedKernel.initialize(String[] args) {
2                   // set scheduler
3                   String schedulerName = Config.getString("ThreadedKernel.scheduler");
4                   scheduler = (Scheduler) Lib.constructObject(schedulerName);
5
6                   // set fileSystem
7                   String fileSystemName = Config.getString("ThreadedKernel.fileSystem");
8                   if (fileSystemName != null)
9                           fileSystem = (FileSystem) Lib.constructObject(fileSystemName);
10                  else if (Machine.stubFileSystem() != null)
11                          fileSystem = Machine.stubFileSystem();
12                  else
13                          fileSystem = null;
14
15                  // start threading
16                  new KThread(null);
17
18                  alarm = new Alarm();
19
20                  Machine.interrupt().enable();
21          }
```

Line 18 in `AutoGrader.start` runs the kernel `selfTest`, executes and terminates the kernel. After kernel termination, the machine halts.

```
1           void run() {
2                   kernel.selfTest();
3                   kernel.run();
4                   kernel.terminate();
5           }
```

**Question:** Provide a snapshot of the outputs from Nachos running the config file in Project 1. Compare the results when running it multiple times. Are they all the same? Why?

# 4 Threads and Scheduling

Nachos provides a kernel threading package, allowing multiple tasks to run concurrently (see `nachos.threads.ThreadedKernel` and `nachos.threads.KThread`).

## 4.1 KThread and Nachos thread life cycles

All Nachos threads are instances of `nachos.threads.KThread` (threads capable of running user-level MIPS code are a subclass of KThread, nachos.userprog.UThread). A `nachos.machine.TCB` object `tcb` is contained by each `KThread` object and provides low-level support for context switches, thread creation, thread destruction, and thread yield.

Every KThread has a status member that tracks the state of the thread. Certain KThread methods will fail (with a `Lib.assert()`) if called on threads in the wrong state. The life cycle of KThread is illustrated in Figure figure26.

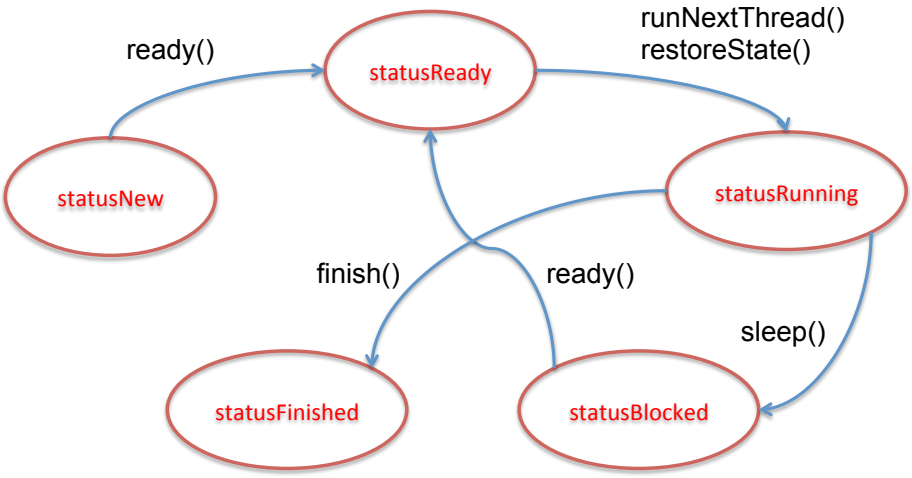


Figure 26: Life cycle of `KThread`

- **statusNew:** A newly created, yet to be forked thread.
- **statusReady:** A thread waiting for access to the CPU. `KThread.ready()` will add the thread to the ready queue and set the status to `statusReady`.
- **statusRunning:** The thread currently using the CPU. `KThread.restoreState()` is responsible for setting status to statusRunning, and is called by `KThread.runNextThread()`.
- **statusBlocked:** A thread which is asleep (as set by KThread.sleep()), waiting on some resource besides the CPU.
- **statusFinished:** A thread scheduled for destruction. Use KThread.finish() to set this status.

Internally, Nachos implements threading using a Java thread for each `TCB` object. The Java threads are synchronized by the `TCBs` such that exactly one is running at any given time. This provides the illusion of context switches saving state for the current thread and loading the saved state of the new thread. For the understanding of `KThread`, we can focus on its public methods.

The life cycle of a `KThread` object starts when it is created. The first `KThread` is created in the `initalize` method of `ThreadKernel`:

```
1         public void initialize(String[] args) {
2                 ...
3                 // start threading
4                 new KThread(null);
5                 ...
6         }
```

The constructor for `KThread` follows the following procedure the first time it is called:

1. Create the ready queue (`ThreadedKernel.scheduler.newThreadQueue()`).
2. Allocate the CPU to the new `KThread` object being created (`readyQueue.acquire(this)`).
3. Set `KThread.currentThread` to the new `KThread` being made.
4. Set the `TCB` object of the new `KThread` to `TCB.currentTCB()`. In doing so, the currently running Java thread is assigned to the new `KThread` object being created.
5. Change the status of the new `KThread` from the default (`statusNew`) to `statusRunning`. This bypasses the `statusReady` state.
6. Create an idle thread.
   (a) Make another new `KThread`, with the target set to an infinite `yield()` loop.
   (b) Fork the idle thread off from the main thread.

After this procedure, there are two `KThread` objects, each with a `TCB` object (one for the main thread, and one for the idle thread). The main thread is not special - the scheduler treats it exactly like any other `KThread`. The main thread can create other threads, it can die, it can block. The Nachos session will not end until all KThreads finish, regardless of whether the main thread is alive.

For the most part the idle thread is also a normal thread, which can be context switched like any other. The only difference is it will never be added to the ready queue (`KThread.ready()` has an explicit check for the idle thread). Instead, if `readyQueue.nextThread()` returns null, the thread system will switch to the idle thread.

If the current thread exists, calling the constructor for `KThread` with a `Runnable` object will simply create a new `TCB` object with the default status `statusNew`. Note that the `KThread` object is not yet added to the ready queue until its `KThread.fork()` method is called. `KThread.fork()` causes the Java thread by the associated `TCB` object to begin execution and calls the `KThread.ready()` method, which puts the `KThread` object in the ready queue.

Note that `KThread.fork()` differs from Unix fork system calls, which returns twice, once in the parent process and once in the child process. `KThread.fork()` only serves to prepare the associated `KThread` for scheduling.

Other public `KThread` methods related to the thread life cycle include, `KThread.run()`, `KThread.sleep()`, `KThread.yield()`, `KThread.finish()`, `KThread.join()`:

- `KThread.run()`
- `KThread.yield()` causes the current thread to relinquish it CPU, add itself to the ready queue and switch to the next thread (or the idle thread) in the ready queue based on the scheduler. Recall that in a time-shared OS, context switches can happen either a thread finishes its execution, its CPU slice is up, it is blocked on some resources, or it yields voluntarily.
- `KThread.sleep()` is called when the current thread has either finished or been blocked. If the current thread is blocked (on a synchronization primitive, i.e. a `Semaphore`, `Lock`, or `Condition`), eventually some thread will wake this thread up, putting it back on the ready queue so that it can be rescheduled. The logic of waking a thread has to be implemented by the respective primitives.
- `KThread.join()` is used by a calling thread to wait for the callee thread to finish before it can proceed. Consider the following example:

```
1          ...
2          KThread t1 = new KThread(new Runnable() {
3                  public void run() {
4                          do something;
5                  }
6          }
7
8          KThread t2 = new KThread(new Runnable() {
9                  public void run() {
10                         do something else;
11                 }
12         }
13
14         ...
15
16         t1.join();
```

```
17          t2.join();
18
19          System.out.println("Finished\n");
20
21          ...
```

The main thread is blocked and would not reach line 19 until both threads finish. Since a current thread cannot join itself, in `KThread.join()`, this thread (the callee) should be different from the current thread (the caller). In the thread and synchronization project, you will be asked to complete the implementation of `KThread.join()`.

- `KThread.finish()` finishes the current thread and schedule it to be destroyed when it is safe to do so.

## 4.2 Scheduler

A sub-class (specified in the nachos.conf) of the abstract base class `nachos.threads.Scheduler` is responsible for scheduling threads for all limited resources, be it the CPU, a synchronization construct like a lock, or even a thread join operation. For each resource a `nachos.threads.ThreadQueue` is created by `Scheduler.newThreadQueue()`. The implementation of the resource (e.g. `nachos.threads.Semaphore` class) is responsible for adding `KThreads` to the `ThreadQueue` (`ThreadQueue.waitForAccess()`) and requesting the `ThreadQueue` return the next thread (`ThreadQueue.nextThread()`). Thus, all scheduling decisions reduce to the selection of the next thread by the `ThreadQueue` objects. Recall the implementation of `KThread.runNextThread()`:

```
1           private static void runNextThread() {
2                   KThread nextThread = readyQueue.nextThread();
3                   if (nextThread == null)
4                           nextThread = idleThread;
5
6                   nextThread.run();
7           }
```

Line 2 calls `readyQueue.nextThread()` to find the next thread in the ready queue for CPU.

The `nachos.threads.RoundRobinScheduler` is the default, and implements a fully functional (though naive) FIFO scheduler. In the Scheduling project, you will be implementing a priority queue scheduling that deals with priority inversion.

The scheduler object is created by the `ThreadKernel.initialize()`:

```
1           public void initialize(String[] args) {
2                   // set scheduler
3                   String schedulerName = Config.getString("ThreadedKernel.scheduler");
```

```
4                   scheduler = (Scheduler) Lib.constructObject(schedulerName);
5                   ...
6           }
```

Thus, by specifying the appropriate scheduler in the config file, one will initialize the respective scheduler object.

**Implement note:**

- Only one type of scheduler can be used for all resources (though each has its own `ThreadQueue`).
- In implementing a new scheduler, one may also have to implement its own `ThreadQueue` class.

## 4.3 Exercise

In this exercise, we finish tracing `KThread.selfTest()` to study the thread package and the default FIFO scheduler.

Recall that `KThread.selfTest()` is called by `AutoGrader.run()` during the boot process.

```
1           public static void selfTest() {
2                   Lib.debug(dbgThread, "Enter KThread.selfTest");
3
4                   new KThread(new PingTest(1)).setName("forked thread").fork();
5                   new PingTest(0).run();
6           }
```

By the time `KThread.selfTest()` is called in Line 4, the main kernel and idle threads have been created.

**Question:** In Eclipse Debug mode, what are the values of the name and the status fields of the `currentThread` object? What about the `idleThread` object?

Now after executing Line 4, a new `KThread` object is created and is put to the ready queue. In the debug mode, you should see that the size of `waitQueue` of `readyQueue` increases by 1. In Line 5, a new runnable object `PingTest(0)` is created and its `run()` method is executed. Since no `KThread` is created for this object, it runs in the main thread. As closer look at the `run()` method of `PingTest`, we find that it just yields the execution of the current thread and put it to the ready queue (Line 7) (to make clear which thread is the current thread, we made a slight change to the source code).

```
1           private static class PingTest implements Runnable {
2                   ...
3                   public void run() {
4                           for (int i = 0; i < 5; i++) {
5                                   System.out.println("*** thread " + which + " looped " + i
6                                                   + " times" + " " + currentThread.name);
7                                   currentThread.yield();
8                           }
9                   }
10
11                  ...
12          }
```

If we trace into `KThread.runNextThread`, we find that `readyQueue.nextThread()`. Since there is only one thread, the FIFO scheduler simply return that thread. The first time `currentThread.yield()` is called, the main thread will yield to the forked thread corresponding to `PingTest(1)`, which in turn yields in its `run()` method to the main thread. Thus, the two threads will take turns in their execution until they finish.

**Question:**

1. Provide the output of the program with the modified source code.
2. Modify `KThread.selfTest()` to create 5 `KThread`s for `PingTest` and trace the order of execution (you can name the threads accordingly to see things more clearly).

# 5 User Level Process

Till now, the execution of these kernel threads are similar to that of Java threads except that they are scheduled by the Nachos scheduler. We have yet dealt with simulation of MIPS instruction sets and user level processes. To allow multi-programming, each user level process should have its own address space. The physical memory needs to be allocated so that different processes do not overlap in their usages. Though the execution of any MIPS instruction is only possible when the memory address it refers to is eventually brought in the physical memory, virtual memory provides an illusion of infinite large memory space for each process. In this section, we discuss how user programs can be executed in Nachos; and in next section, we discuss the implementation of Nachos memory management.

## 5.1 Developing and compiling user programs

Nachos cross-compiler allows programmers to write C user programs and compile to COFF binaries that can be executed on the Nachos MIPS simulator. COFF stands for Common Object File Format and is an industry-standard binary format which the Nachos kernel understands. The cross-compiler itself is platform-dependent (though the COFF binary is not) and therefore one needs to make sure the correct one is used.

The following code is an example of C user program (*nachos/test/cp.c*).

```
1  #include "syscall.h"
2  #include "stdio.h"
3  #include "stdlib.h"
4
5  #define BUFSIZE 1024
6
7  char buf[BUFSIZE];
8
9  int main(int argc, char** argv)
10 {
11   int src, dst, amount;
12
13   if (argc!=3) {
14     printf("Usage: cp <src> <dst>\n");
15     return 1;
16   }
17
18   src = open(argv[1]);
19   if (src==-1) {
20     printf("Unable to open %s\n", argv[1]);
21     return 1;
22   }
23
24   creat(argv[2]);
```

```
25    dst = open(argv[2]);
26    if (dst==-1) {
27      printf("Unable to create %s\n", argv[2]);
28      return 1;
29    }
30
31    while ((amount = read(src, buf, BUFSIZE))>0) {
32      write(dst, buf, amount);
33    }
34
35    close(src);
36    close(dst);
37
38    return 0;
39  }
```

The corresponding COFF binary is also located in the *nachos/test* directory. This is the default directory for user programs. Alternatively, one can specify the test directory by including a line `fileSystem.testDirectory = ...` in the config file.

> **Question:** Where and how does Nachos set the test directory? [hint: Check `nachos.machine.Machine.main()`]

Inspecting the *cp.c* source code, we find that it follows the C syntax with familiar functions such as `printf`, `open`, etc. However, a closer look at the test directory and Makefile reveals that some of the functions are in fact implemented by Nachos in the test directory. The function prototypes of system calls such as `creat, open, read, write, close, halt`, etc., are defined in *syscall.h* and their implementations are to be completed in the one of the Nachos projects.

There are multiple stages to building a Nachos-compatible MIPS binary (all of which are handled by the test Makefile):

1. Source files (*.c) are compiled into object files (*.o) by *mips-gcc*.
2. *start.s* is preprocessed and assembled into *start.o*. This file contains the assembly-language code to initialize a process. It also provides the system call "stub code" which allows system calls to be invoked. This makes use of the special MIPS instruction syscall which traps to the Nachos kernel to invoke a system call.
3. An object file is linked with *libnachos.a* to produce a Nachos-compatible MIPS binary, which has the extension *.coff.
4. Note that if you create a new test file (*.c), you will need to append your program name to the variable TARGETS in the Makefile inside test directory

One can run test programs by running "*nachos -x PROGNAME.coff*". In `nachos.conf`, the appropriate kernel and process need to be specified:

```
1  Machine.stubFileSystem = true
2  Machine.processor = true
3  Machine.console = true
4  ...
5  Kernel.processClassName = nachos.userprog.UserProcess
6  Kernel.kernel = nachos.userprog.UserKernel
```

`subFileSystem` is enabled to load the user program.

## 5.2 Loading COFF binaries

COFF (Common Object File Format) binaries contain a lot of information, but very little of it is actually relevent to Nachos programs. Further, Nachos provides a COFF loader class, `nachos.machine.Coff`, that abstracts away most of the details. But a few details are still important. A COFF binary is broken into one or more sections. A section is a contiguous chunk of virtual memory, all the bytes of which have similar attributes (code vs. data, read-only vs. read-write, initialized vs. uninitialized).

Nachos classes that are needed to handle user program are mostly reside in `nachos.userprog`. To support multi-programming, `UserKernel` extends `ThreadedKernel`. When `kernel.run()` is called in `Autograder.run()`, Nachos creates a process and execute the shell program specified by the "-x" argument or in the config file. This is done by first loading the program into the process' address space, at some start address specified by the section (Line 2). A COFF binary also specifies an initial value for the PC register and the stack pointer. Both values will be stored within the `UserProcess` object. Lastly, a user thread (`nachos.userprog.UThread`) is created and put in the ready queue (Line 5).

```
1          public boolean nachos.userprog.UserProcess.execute(String name, String[] args) {
2                  if (!load(name, args))
3                          return false;
4
5                  new UThread(this).setName(name).fork();
6
7                  return true;
8          }
```

## 5.3 User threads

`nachos.userprog.UThread` extends `KThread` with the main difference in how the thread objects run. When the user thread runs on the CPU, the PC register and the stack pointer will be initialized accordingly. The memory address

of the command line arguments `argc` and `argv` will be stored in two argument registers (See `UThread.initRegisters()`). Next, the process' state will be restored after the context switch. If a linear page table is used for memory management (more details see Section section6), an array of translation entries is specified by calling `Processor.setPageTable()`. The page table provides the mapping between virtual address to physical address. From here on, the processor will fetch the instructions from the memory location indicated by the PC register and execute the user program. Execeptions occur in a number of situations, namely, a system call, an invalid instruction, page faults, TLB miss, etc. A complete list of exceptions can be found in `Processor.exceptionNames`.

A user thread may be context switched before it finishes. In this case, registers are stored in `UThread.userRegisters`, which will be restored if the thread is scheduled again. Methods `UThread.saveState()` and `UThread.restoreState()` deal with saving and loading states during context switches.

## 5.4 System calls and exception handling

User programs invoke system calls by executing the MIPS syscall instruction, which causes the Nachos kernel exception handler to be invoked (with the `cause` register set to `Processor.exceptionSyscall`). The kernel must first tell the processor where the exception handler is by calling `Machine.processor().setExceptionHandler()`

The default Kernel exception handler, `UserKernel.exceptionHandler()`, reads the value of the processor's `cause` register, determines the current process, and invokes handleException on the current process, passing the cause of the exception as an argument. Again, for a syscall, this value will be `Processor.exceptionSyscall`.

```
1         public void handleException(int cause) {
2                 Processor processor = Machine.processor();
3
4                 switch (cause) {
5                 case Processor.exceptionSyscall:
6                         int result = handleSyscall(processor.readRegister(Processor.regV0
7                                         processor.readRegister(Processor.regA0),
8                                         processor.readRegister(Processor.regA1),
9                                         processor.readRegister(Processor.regA2),
10                                        processor.readRegister(Processor.regA3));
11                        processor.writeRegister(Processor.regV0, result);
12                        processor.advancePC();
13                        break;
14
15                default:
16                        Lib.debug(dbgProcess, "Unexpected exception: "
17                                        + Processor.exceptionNames[cause]);
18                        Lib.assertNotReached("Unexpected exception");
19                }
20        }
```

The syscall instruction indicates a system call is requested, but doesn't indicate which system call to perform. By convention, user programs place the value indicating the particular system call desired into MIPS register $r2$ (the first return register, $v0$) before executing the syscall instruction. Arguments to the system call, when necessary, are passed in MIPS registers $r4$ through $r7$ (i.e. the argument registers, $a0$ ... $a3$), following the standard C procedure call convention. Function return values, including system call return values, are expected to be in register $r2$ ($v0$) on return. Only the halt system call has been implemented, you will be asked to complete the implementation of the method `UserProcess.handleSyscall` for other system calls. Note that the registers do NOT store the values of the arguments, rather, the virtual memory locations of these arguments. For example, consider a method that handles `open` system call. From *test/syscall.h*, we have

```
1         int open(char *name);
```

Thus, there should only be one argument to the method, say, `handleOpen(int name)`. To get the actual string that stores the file name to be opened, one can use the method `UserProcess.readVirtualMemoryString(name, maxFileNameLength)`, where `maxFileNameLength` is the (programmer defined maximum length of file names).

> **Implementation Note:** When accessing user memory from within the exception handler (or within Nachos in general), user-level addresses cannot be referenced directly. Recall that user-level processes execute in their own private address spaces, which the kernel cannot reference directly. Use `readVirtualMemory()`, `readVirtualMemoryString()`, and `writeVirtualMemory()` to make use of pointer arguments to syscalls.

## 5.5 Exercise

In this exercise, we study how user program is loaded into the process' address space. It will be helpful to the implementation of the multi-programming project.

```
1         private boolean load(String name, String[] args) {
2                 Lib.debug(dbgProcess, "UserProcess.load(\"" + name + "\")");
3
4                 OpenFile executable = ThreadedKernel.fileSystem.open(name, false);
5                 if (executable == null) {
6                         Lib.debug(dbgProcess, "\topen failed");
7                         return false;
8                 }
9
10                try {
```

```
11                      coff = new Coff(executable);
12              } catch (EOFException e) {
13                      executable.close();
14                      Lib.debug(dbgProcess, "\tcoff load failed");
15                      return false;
16              }
17
18              // make sure the sections are contiguous and start at page 0
19              numPages = 0;
20              for (int s = 0; s < coff.getNumSections(); s++) {
21                      CoffSection section = coff.getSection(s);
22                      if (section.getFirstVPN() != numPages) {
23                              coff.close();
24                              Lib.debug(dbgProcess, "\tfragmented executable");
25                              return false;
26                      }
27                      numPages += section.getLength();
28              }
29
30              // make sure the argv array will fit in one page
31              byte[][] argv = new byte[args.length][];
32              int argsSize = 0;
33              for (int i = 0; i < args.length; i++) {
34                      argv[i] = args[i].getBytes();
35                      // 4 bytes for argv[] pointer; then string plus one for null byte
36                      argsSize += 4 + argv[i].length + 1;
37              }
38              if (argsSize > pageSize) {
39                      coff.close();
40                      Lib.debug(dbgProcess, "\targuments too long");
41                      return false;
42              }
43
44              // program counter initially points at the program entry point
45              initialPC = coff.getEntryPoint();
46
47              // next comes the stack; stack pointer initially points to top of it
48              numPages += stackPages;
49              initialSP = numPages * pageSize;
50
51              // and finally reserve 1 page for arguments
52              numPages++;
53
54              if (!loadSections())
55                      return false;
56
57              // store arguments in last page
58              int entryOffset = (numPages - 1) * pageSize;
59              int stringOffset = entryOffset + args.length * 4;
60
```

```
61                  this.argc = args.length;
62                  this.argv = entryOffset;
63
64                  for (int i = 0; i < argv.length; i++) {
65                          byte[] stringOffsetBytes = Lib.bytesFromInt(stringOffset);
66                          Lib.assert(writeVirtualMemory(entryOffset, stringOffsetBytes) ==
67                          entryOffset += 4;
68                          Lib.assert(writeVirtualMemory(stringOffset, argv[i]) == argv[i].l
69                          stringOffset += argv[i].length;
70                          Lib.assert(writeVirtualMemory(stringOffset, new byte[] { 0 }) ==
71                          stringOffset += 1;
72                  }
73
74                  return true;
75          }
```

Recall that `UserProcess.load()` load the executable with the specified name into the user process. In Line 4 – 8, it first opens the executable from the `stubFileSystem`. Note that `ThreadedKernel.fileSystem.open` does not invoke any Nachos system calls, instead, it utilizes the file IO routines from Java.

The Coff constructor takes one argument, an `OpenFile` referring to the MIPS binary file. If there is any error parsing the headers of the specified binary, an EOFException is thrown. Note that if this constructor succeeds, the file belongs to the Coff object; it should not be closed or accessed anymore, except through Coff operations.

There are four Coff methods that operate on the Coff object:

- `getNumSections()` returns the number of sections in this binary.
- `getSection()` takes a section number, between 0 and getNumSections() - 1, and returns a `CoffSection` object representing the section. This class is described below.
- `getEntryPoint()` returns the value with which to initialize the program counter.
- `close()` releases any resources allocated by the loader. This includes closing the file passed to the constructor.

The `CoffSection` class allows Nachos to access a single section within a COFF executable. Note that while the MIPS cross-compiler generates a variety of sections, the only important distinction to the Nachos kernel is that some sections are read-only (i.e. the program should never write to any byte in the section), while some sections are read-write (i.e. non-const data). There are four methods for accessing COFF sections:

- `getFirstVPN()` returns the first virtual page number occupied by the section.

- `getLength()` returns the number of pages occupied by the section. This section therefore occupies pages `getFirstVPN()` through `getFirstVPN() + getLength() - 1`. Sections should never overlap.
- `isReadOnly()` returns true if and only if the section is read-only (i.e. it only contains code or constant data).
- `loadPage()` reads a page of the section into main memory. It takes two arguments, the page within the section to load (in the range 0 through getLength() - 1) and the physical page of memory to write.

Line 20 - 28 read each section of the COFF executable and ensures the sections do not overlap. Line 31 – 38 extract the arguments of the user programs and check if the argument array fits in one page. Line 44 sets the program pointer to point at the program entry point. Line 49 sets the stack point to the top of the stack. Line 54 – 55 load the COFF sections into memory.

Finally, the arguments are then stored at the last page in Line 57 – 72.

**Questions:**

1. What are the variables `this.argc` and `this.argv`?
2. In Line 64 – 72, how exactly are the arguments stored? [Hints: not all arguments are of the same length.]

Next, we run the Nachos in debug mode with config file in *nachos/proj2*[2]. The shell program is *halt.coff*, which is compiled from *halt.c*.

```
1  #include "syscall.h"
2
3  int main()
4  {
5      halt();
6      /* not reached */
7  }
```

We can see from the source code *halt.c* that it does nothing but invokes the system call `halt()`. Set an appropriate breakpoints in `UserProcess` and answer the following questions.

[2]In Eclipse, you can set the working directory to *nachos/proj2* in run/debug configuration.

**Questions**

1. How many sections are there in *halt.coff*? What are they?
2. How many pages do the sections occupy?
3. How many page does the stack occupy?
4. What are the values of the initial PC and SP?
5. Change the shell program to *test/matmult.coff*. Answer the above questions when `Dim = 20` and `Dim = 2000`.

# 6 Nachos Memory Management

As discussed in Section section3, the processor provides registers and physical memory, and supports virtual memory.

The physical memory is byte-addressable and organized into 1-kilobyte pages. A reference to the main memory array is returned by `getMemory()`. Memory corresponding to physical address `m` can be accessed in Nachos at `Machine.processor().getMemory()[m]` The number of pages of physical memory is returned by `getNumPhysPages()`.

When it comes to virtual memory for multi-programming, there are two key aspects that need to be addressed in the implementation. First, how is the physical memory allocated among user processes? Since virtual memory allows the size of the address space of a user process to be larger than that of the physical memory, naturally, questions like which part of the address space should be in the physical memory and what happens in presence of a page fault arise. In the class, we have discussed types of page faults and various strategies for page replacement including FIFO, LRU, etc. The second question is how to map virtual addresses to physical addresses. Users should only be concerned with virtual addresses. However, actual memory references should be with respect to physical addresses. In what follows, we will investigate how the two aspects are implemented in the default Nachos.

## 6.1 Memory allocation

The default Nachos implementation assumes uni-programming, namely, only one user process runs and occupies the physical memory at a time (note this is different from having multiple kernel threads). When a process is executed, its program and data are loaded to the main memory. Furthermore, the entire address space fits in memory. Therefore, virtual memory and physical memory are in fact identical in this case.

```
1         protected boolean UserProcess.loadSections() {
2                 ...
3
4                 // load sections
5                 for (int s = 0; s < coff.getNumSections(); s++) {
6                         CoffSection section = coff.getSection(s);
7
8                         Lib.debug(dbgProcess, "\tinitializing " + section.getName()
9                                         + " section (" + section.getLength() + " pages)")
10
11                        for (int i = 0; i < section.getLength(); i++) {
12                                int vpn = section.getFirstVPN() + i;
13
14                                // for now, just assume virtual addresses=physical addres
15                                section.loadPage(i, vpn);
16                        }
17                }
```

```
18
19                  return true;
20          }
```

In Line 15, `section.loadPage` loads a page in the section into a physical memory page indexed by `vpn`.

> **Implementation notes:** Nachos user programs do not make use of malloc() or free() and thus effectively have no dynamic memory allocation needs (or equivalently, no heap). In the current implementation, a fixed number of pages is used for the process' stack, e.g., 8 pages. Therefore, the complete memory needs of a process is known when it is created. This eases the task of static memory allocation when no on-demand paging is used.

## 6.2 Address translation

Nachos processor supports VM through either a single linear page table or a software-managed TLB (but not both). The mode of address translation is actually used is determined by nachos.conf, and is returned by `hasTLB()`.

In both cases, the `TranslationEntry` class is the data structure in Nachos to store information related to a page. Each `TranslationEntry` object contains the physical page number, the virtual page number, whether the page is ready only, whether it has been used recently (e.g., to implement the CLOCK algorithm), and whether the entry is valid (if not, accessing it will result in a page fault). The `Processor.translate` method translates virtual address into a physical address, using either a page table or a TLB.

```
1           private int translate(int vaddr, int size, boolean writing)
2                   throws MipsException {
3               ...
4
5               // calculate virtual page number and offset from the virtual address
6               int vpn = pageFromAddress(vaddr);
7               int offset = offsetFromAddress(vaddr);
8
9               TranslationEntry entry = null;
10
11              // if not using a TLB, then the vpn is an index into the table
12              if (!usingTLB) {
13                      if (translations == null || vpn >= translations.length
14                                      || translations[vpn] == null || !translations[vpn
15                              privilege.stats.numPageFaults++;
16                              Lib.debug(dbgProcessor, "\t\tpage fault");
17                              throw new MipsException(exceptionPageFault, vaddr);
18                      }
19
```

```
20                      entry = translations[vpn];
21              }
22              // else, look through all TLB entries for matching vpn
23              else {
24                      for (int i = 0; i < tlbSize; i++) {
25                              if (translations[i].valid && translations[i].vpn == vpn)
26                                      entry = translations[i];
27                                      break;
28                              }
29                      }
30                      if (entry == null) {
31                              privilege.stats.numTLBMisses++;
32                              Lib.debug(dbgProcessor, "\t\tTLB miss");
33                              throw new MipsException(exceptionTLBMiss, vaddr);
34                      }
35              }
36
37              // check if trying to write a read-only page
38              if (entry.readOnly && writing) {
39                      Lib.debug(dbgProcessor, "\t\tread-only exception");
40                      throw new MipsException(exceptionReadOnly, vaddr);
41              }
42
43              // check if physical page number is out of range
44              int ppn = entry.ppn;
45              if (ppn < 0 || ppn >= numPhysPages) {
46                      Lib.debug(dbgProcessor, "\t\tbad ppn");
47                      throw new MipsException(exceptionBusError, vaddr);
48              }
49
50              // set used and dirty bits as appropriate
51              entry.used = true;
52              if (writing)
53                      entry.dirty = true;
54
55              int paddr = (ppn * pageSize) + offset;
56
57              if (Lib.test(dbgProcessor))
58                      System.out.println("\t\tpaddr=0x" + Lib.toHexString(paddr));
59              return paddr;
60      }
```

### 6.2.1 Software-managed TLB

The default TLB size is 4. The kernel can query the size of the TLB by calling `getTLBSize()`, and the kernel can read and write TLB entries by calling `readTLBEntry()` and `writeTLBEntry()`. TLB is used to cache recently used page table entries and to expedite address translation. If TLB is enabled (`usingTLB = true`),

In Line 24 – 34 of `Processor.translate`, the input `vpn` will be looked up among the TLB entries. If not found, a `exceptionTLBMiss` exception is generated and handled by the exception handler (to be implemented). Otherwise, the associated page entry will be checked for validity and the respective dirty and used bits will be set accordingly.

In the case of TLB misses, an appropriate TLB replacement policy shall be implemented to substitute existing TLB entries with new ones.

TLB can be used in conjunction with per-process page table or a global inverted page table for address translation.

### 6.2.2 Per-process page table

If the processor does not have a TLB, `Processor.translate` (Line 13 – 20) looks up the page table in `translations` and retrieves the entry corresponding to the virtual address. If the page table is `null`, the respective entry is `null` or invalid then a page fault exception will be generated.

Per-process page table is set up by calling `Processor.setPageTable()` by the user process. On a real machine, the page table pointer would be stored in a special processor register. The user process is responsible for populating the per-process page table initially.

```
1          public UserProcess() {
2                  int numPhysPages = Machine.processor().getNumPhysPages();
3                  pageTable = new TranslationEntry[numPhysPages];
4                  for (int i = 0; i < numPhysPages; i++)
5                          pageTable[i] = new TranslationEntry(i, i, true, false, false, fal
6          }
```

As indicated in Line 5, in uni-programming, it is sufficient to initialize the page table with one-to-one mapping between the physical and virtual memory.

# A Common Object File Format (COFF)

The Common Object File Format (COFF) is a specification of a format for executable, object code, and shared library computer files used on Unix systems. COFF is mostly replaced by ELF.

## A.1 COFF header

At the beginning of an object file, or immediately after the signature of an image file, there is a standard COFF header of the following format (See table center10). More details can be found at [].

Table 9: COFF Header format

| Offset | Size | Field | Description |
|---|---|---|---|
| 0 | 2 | Machine | Number identifying type of target machine |
| 2 | 2 | NumberOfSections | Number of sections; indicates size of the Section Table, which immediately follows the headers. |
| 4 | 4 | TimeDateStamp | Time and date the file was created. |
| 8 | 4 | PointerToSymbolTable | File offset of the COFF symbol table or 0 if none is present. |
| 12 | 4 | NumberOfSymbols | Number of entries in the symbol table. This data can be used in locating the string table, which immediately follows the symbol table. |
| 16 | 2 | SizeOfOptionalHeader | Size of the optional header, which is required for executable files but not for object files. An object file should have a value of 0 here. The format is described in the section Optional Header. |
| 18 | 2 | Characteristics | Flags indicating attributes of the file. |

## A.2 Section table

A section table contains a collection of section table entries each containing 40 bytes of information listed in Table center10. The number of entries in

the Section Table is given by the NumberOfSections field in the file header. Entries in the Section Table are numbered starting from one. The code and data memory section entries are in the order chosen by the linker. In an image file, the virtual addresses for sections must be assigned by the linker such that they are in ascending order and adjacent, and they must be a multiple of the Section Align value in the optional header.

# B Q&As – Questions Raised During Nachos Projects

## B.1 Virtual memory

$\mathcal{Q}_a$: *Do I need to keep both global invert page table (IPT) and a per-process page table?*

*Ans*: No.

$\mathcal{Q}_b$: *Where is the swap file stored? How to read and write swap file?*

*Ans*: A swap file is a regular file in the stubFileSytem that can read and written using `OpenFile.read()` and `OpenFile.write()`. You may implement a private class for operations (open, delete, swap in, swap out, etc.) related to swap file. Swap file is organized in pages. Similar to IPT, you can use a hash table to store the mapping between <process id, virtual page number> and the index of swap page entry.

$\mathcal{Q}_c$: *With on-demand paging, shall I allocate all free physical pages when executing a process (`loadSections`)?*

*Ans*: This is implementation dependent. Some OS reserves a pool of free physical pages to reduce the latency for handling page faults. In Linux, code section of the process is on disk (the executable becomes part of the swap file). In your implementation, you may choose to put as many pages in the physical memory as possible and let on-demand paging handle the rest if no free physical pages are available.

$\mathcal{Q}_d$: *How to generate page faults using matmult.c?*

*Ans*: In matmult.c, 3 integer arrays of size DIMxDIM are statically allocated. By changing the value of DIM, you can make the total memory requirement exceeds the # of pages of physical memory. (Alternatively, you can also reduce the # of physical pages in the config file.

Table 10: Section table entries in COFF

| Offset | Size | Field | Description |
|---|---|---|---|
| 0 | 8 | Name | An 8-byte, null-padded ASCII string. There is no terminating null if the string is exactly eight characters long. For longer names, this field contains a slash (/) followed by ASCII representation of a decimal number: this number is an offset into the string table. Executable images do not use a string table and do not support section names longer than eight characters. Long names in object files will be truncated if emitted to an executable file. |
| 8 | 4 | VirtualSize | Total size of the section when loaded into memory. If this value is greater than Size of Raw Data, the section is zero-padded. This field is valid only for executable images and should be set to 0 for object files. |
| 12 | 4 | VirtualAddress | For executable images this is the address of the first byte of the section, when loaded into memory, relative to the image base. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation. |
| 16 | 4 | SizeOfRawData | Size of the section (object file) or size of the initialized data on disk (image files). For executable image, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize the remainder of the section is zero filled. Because this field is rounded while the VirtualSize field is not it is possible for this to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be 0. |
| 20 | 4 | PointerToRawData | File pointer to section's first page within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a four-byte boundary for best performance. When a section contains only uninitialized data, this field should be 0. |
| 24 | 4 | PointerToRelocations | File pointer to beginning of relocation entries for the section. Set to 0 for executable images or if there are no relocations. |
| 28 | 4 | PointerToLinenumbers | File pointer to beginning of line-number entries for the section. Set to 0 if there are no COFF line numbers. |
| 32 | 2 | NumberOfRelocations | Number of relocation entries for the section. Set to 0 for executable images. |
| 34 | 2 | NumberOfLinenumbers | Number of line-number entries for the section. |
| 36 | 4 | Characteristics | Flags describing sections characteristics. |

$\mathcal{Q}_{\mathrm{e}}$:

*Ans*:

$\mathcal{Q}_{\mathrm{f}}$:

*Ans*:

$\mathcal{Q}_{\mathrm{g}}$:

*Ans*: