[Nachos]

Nachos is an instructional operating system designed by Thomas Anderson at UC- Berkeley. Originally written in C++ for MIPS, Nachos runs as a user-process on a host OS. Here we have discussed the Nachos-1 assignment and also described how to work with thread class, scheduling class and synchronization support in Nachos.

Threads, Synchronization & Scheduling

Assignment 1

A1 A2 B1 B2

1 Introduction

Nachos is an instructional software, an attempt to study and modify an operating system. But there is a difference between Nachos and a real operating system. Nachos runs as a single Unix user process whereas real operating systems run on bare machines. However, Nachos simulates the general low level facilities of typical machines including interrupts, vm and interrupt driven device IO.

To complete the first assignment you need not worry about MIPS. Because, for the first assignment your business is with the thread package provided with nachos, not the simulated MIPs hardware. In fact, one does not require writing user level programs to do useful works with nachos. Nachos kernel is a multithreaded program. Each thread is capable of running independently without the intervention of any other threads. And these threads are **green** threads, i.e. managed completely by Nachos itself without depending on POSIX threads. To complete your assignment, you need to understand nachos thread implementation.

Since Nachos is running on our real machine as a single process, a definite question arises: how it can run multiple threads independently within a single process? To get the answer of this question, you have to go through the Nachos thread classes and understand them very carefully. So first task of the assignment is to understand Nachos thread class.

2 Understanding Nachos Threads

Nachos ***Thread*** class is defined in **"nachos-3.4/code/threads/thread.h"** and the interfaces are implemented in "**nachos-3.4/code/threads/thread.cc**". Go through these two files attentively to understand what is going on inside nachos threads. The four functions that you are needed to study are:

1. ***Fork ( )***
2. ***Yield ( )***
3. ***Sleep ( )***
4. ***Finish ( )***

Understand the ***Scheduler*** class since it schedules the threads. The most important functions of this class that need to be understood are:

1. ***ReadyToRun ( )***
2. ***FindNextToRun ( )***
3. ***Run ( )***

Find the details from the document "A roadmap to Nachos".

3 Implementation of Lock in Nachos

The class and interfaces for ***Lock*** and ***Condition*** variables are already defined in nachos. You can find it at "**nachos-3.4/code/threads/synch.h**". The interfaces are implemented in the file "**nachos-3.4/code/threads/synch.cc**". You need to change both these files in order to do the assignment. You will also find Semaphore already implemented. But remember that the existing code must not be modified; since they are used everywhere in nachos source code.

Modifying anything intentionally or unintentionally might cause an error when we will compile nachos next time. So you should be careful so that no existing code is modified. You only need to add (not modify) some code in the existing file and then recompile the nachos. This will create a new nachos executable program which will do the job. We present here a step by step development of the ***Lock*** implementation code so that we can easily understand it.

From your basic OS knowledge, you should know that Lock or Mutex is used to achieve mutual exclusion. If we want that for a certain portion of code or for a shared object in memory, only one thread can enter that region at once, we have to use Lock. The ***Lock*** class is defined partially in the file "**synch.h**". Open that file. You will find :

```
class Lock
{
 public:
 Lock(char*
 debugName); ~Lock();
 char* getName() { return name;
 } void Acquire();
 void Release();
 bool isHeldByCurrentThread();

 private:
char* name;
List *queue;
};
```

So the class ***Lock*** has four public methods (functions). They are

1. ***getName ( )***
2. ***Acquire ( )***
3. ***Release ( )***
4. ***isHeldByCurrentThread ( )***

1. getName()

This function simply returns the name of the ***Lock*** object. When you create a ***Lock*** object, you give it a name as a parameter. getName() returns that name.

2. Acquire()

A thread acquires a lock by calling acquire function of the ***Lock*** object. For example calls like : `myLock->Acquire();`

You have to consider some issues here:

1. Assume the ***Lock*** object is free. So calling thread returns acquiring it. You may like to keep a private variable for the lock class to indicate whether the lock is free or busy.
2. Assume the ***Lock*** object is already acquired by another thread. So the calling thread should be blocked until the ***Lock*** object is set free. To block a thread, call the function ***Thread::Sleep()*** for the currently running thread.

3. Moreover, Consider this case: Two or more threads call the ***Lock::Acquire()*** function and get it busy; so all of them are sleeping. This might be the case that (although it does not always happen) one or more of them wakes up by some external reasons). When it wakes up it will get the ***Lock*** (although Lock is not set free). So ensure inside acquire function – if sleeping threads wake up they should again check the condition whether ***Lock*** is really free).

4. Locks are used to achieve mutual exclusion on some code or some memory area. But the acquire needs to be atomic. i.e. cpu should not be taken away once a thread is inside the acquire or release function. So you should enable/restore interrupt at proper places.

3. Release()

Release function needs to do the following things:-

1. If ***Lock*** object is held by the thread that calls the ***Release()*** function then we will just release the ***lock*** object:

```
void Release()
{
      isHeldBySome = false;
}
```

However before releasing the ***Lock*** object you need to ensure that the calling thread is really the one who holds the ***Lock*** object. How can you do it ?
Simply you can't. Because you have no way of telling who holds the ***Lock***. So you need another variable to be defined in the ***Lock*** class. Assume the name of that variable is ***currentHolder***.

The function ***isHeldByCurrentThread()*** simply returns true if the ***Lock*** object is currently held by the calling thread. You can easily implement it using our previously defined currentHolder variable which keeps track of which thread holds currently the ***Lock*** object :

```
bool Lock::isHeldByCurrentThread()
{
        if(currentThread == currentHolder ) return
        true; else return false;
}
```

You have to modify the ***Acquire()*** function a little bit to track the record of currentHolder variable.

Moreover, when a lock is released, it is the responsibility of the holder thread to wake up those threads from their sleep so that they can proceed further. So, you need to keep track of the threads waiting for a lock. You can do it by keeping a List variable in Lock class that will hold all the threads that are sleeping on a ***Lock***.

List class is already implemented in nachos. You can find it here "**nachos-3.4/code/threads/list.h"** and "**nachos-3.4/code/threads/list.cc**". You need to study these two files to learn about the nachos List class.

Now in ***Release()*** function we need to wake up those threads that are sleeping.

Since the ***Lock*** private variables are shared data structure and two or more threads can call the ***Release()*** function of the same ***Lock*** object in parallel you must ensure that only one is executing at any time so that the consistency of data structure is maintained. So you should turn off /restore interrupts properly.

In summary to implement ***Lock*** in nachos:

You need to add some codes in "synch.h" and "synch.cc" files.

You need to study "synch.h", "synch.cc", "list.h", "list.cc" "thread.h" "thread.cc" "scheduler.h" "scheduler.cc" files.

4 Implementation of Condition Variable

Condition variables make writing concurrent programs easy. They have two basic operations, namely **Wait** and **Signal**. Condition variables supplement monitor procedures. As Nachos is written in C++, there is no automatic support for monitor procedure; however here we use them along with locks, to implement a powerful yet easy synchronization mechanism.

Note that : in Java the "**synchronized**" keyword stands for monitor procedures, and within synchronized code blocks, we sometimes use **wait()**, **notify()** and **notifyAll()** methods. These methods are similar to Nachos' **Condition::Wait()**, **Condition::Signal()** and **Condition::Broadcast()** functions. The class ***Condition*** is defined in **synch.h** and is shown here:

```
class Condition
{
    public:
      Condition(char*
      debugName); ~Condition();
      char* getName() { return (name); }

      void Wait(Lock *conditionLock); void
      Signal(Lock *conditionLock); void
      Broadcast(Lock *conditionLock);

    private:
      char* name;
      List *waitqueue;
};
```

*1. Wait(Lock * conditionLock)*

When a synchronized procedure discovers that it can't continue without some event to happen, it does a **wait** on some condition variable, which corresponds to that expected event. The thread is blocked then.

*2. Signal(Lock * conditionLock)*

Within a synchronized procedure, when some thread finds some event has happened, which corresponds to some condition variable, the tread does a **signal** on that condition variable, and one of the threads waiting for that condition are waken up.

*3. Broadcast(Lock * conditionLock)*

Broadcast() function is similar to Signal() function except that in this case all sleeping threads are wake up rather than only one of them.

An illustrative example of using Condition variables

Consider the producer consumer problem. The producer produces food and consumer consumes food. Producer needs a definite amount of time to produce food. After producing the food it stores the food in a table. The table is of limited size so certain amount of food can be put there. Consumer comes to the table and whenever there is food he consumes it. If the consumer consumes at a higher rate than producer produces then at some time consumer will find that there is no food in the table. What should it do now? Certainly it has nothing else to do other than eating. So it would be good if it goes to sleep (it will help digest the food). Later the producer again produces some food and keeps it in the table. But now consumer is sleeping (it has gone to sleep before finding the table empty). So at one time the table will be filled with food (because producer is sleeping so no one is eating food). Next time when producer comes to the table with food, it finds that table is filled. Hence it needs not to produce any food until the table has some place empty. What it should do now? Since it has nothing else to do, it is better to go to sleep. The case now is that both the producer and the consumer are sleeping. No useful work will be done ever in future. Both will sleep forever. The question here is that when the consumer goes to sleep (because of unavailability of food) there is nobody who will wake it up. Similarly when the producer goes to sleep (because the table is filled with food) no one is waking it up. So there must be someone who will wake both of them up. The consumer sleeps when it finds no food. So it will be best to wake the consumer up again when there is food in the table. The producer only knows it because it produces the food. So it must be the responsibility of the producer to wake any sleeping consumer up. Similarly when producer sleeps it needs to wake up again when there is empty place in the table. Only the consumer creates empty place in the table (by eating some food). So it must be the responsibility of the consumer to wake up sleeping producer.

Considering the above scenario the usefulness of Condition variable comes in mind. Whenever consumer finds no food on the table, it sleeps on the condition “there is some food on the table”. It will sleep until the condition is true. When producer produces food; it fulfills the condition so it wakes up the consumer.

The ***Wait()*** function is called by the consumer when it finds no food. It wants to wait until there is some food true. There is a private variable ***waitQueue*** defined in ***Condition*** class which keeps track of who are waiting on a particular condition (condition means ***Condition*** object).

However : see that there is an argument of Lock object in this function. What does it do? Before calling the ***Wait()*** function of the condition object, the thread at first acquires a ***Lock*** object (to ensure that 2 or more is not accessing the shared

medium simultaneous). In our producer consumer example let we don't allow to access the food table simultaneously. So either the producer or the consumer can come to the table. To ensure that they do not both come to the table simultaneously, we need a Lock object. Before coming to the table, they will first try to acquire the Lock. If someone in the table then others will not be able to acquire the lock. This Lock object is passed as a parameter in the ***Condition::Wait()*** function. But why is it needed by the condition object? Here how it is. Consider again the consumer acquires the ***Lock*** object. So it comes to the table. There it finds no food. So it calls the ***Condition::Wait()*** function. In the ***Condition::Wait()*** function it will go to sleep. But before going to sleep we need that it releases the Lock object so that the producer can come to the table and keep some food.(if it do not release the Lock object and just go to sleep, producer will not be able to come to the table anymore to keep food, so nothing will be done forever). That is why the ***Lock*** object is passed here as a parameter so that before going to sleep we can release the ***Lock***.

Simply Releasing the Lock will not do everything. We assume that the caller does not know about what is going on in ***Condition::Wait()*** function. So if we release the Lock object in wait function we must again acquire it before returning to the caller (setting the Lock in its previous state).

Finally, we know that two or more threads can call the ***Condition::Wait()*** function simultaneously. Since there is a shared data structure here (***waitQueue***), we need to ensure exclusive access to these function. So, do not forget to control the interrupt. You should check it explicitly that the caller calling the Wait() function really acquired the Lock object.

Assume consumer is sleeping (because when it came to the table it found no food there). Next time when producer produces some food and keeps it in the table it will first search whether any consumer is sleeping. So it will call the ***Signal()*** function of the ***Condition*** object to wake up them (in fact only one of them).

5 Solve Multiple Producer-Consumer Problem

Now you have to solve Multiple Producer-Consumer problem using the primitives you have designed (Lock and Condition Variables).

You have to create a Producer class and a Consumer class. Do not add them in threadtest.cc, you have to create separate files producer.h, producer.cc, consumer.h, consumer.cc. Producer will have a produce function and consumer will have a consume function. Create a global buffer, which the producers and the consumers will use.

Illustrate your solution by creating several producers and consumers. Producing and consuming should take some time. So inside the critical region, the producers or consumers execute a for loop upto some random number, get out of critical region, execute a random for loop again before performing next produce or consume.

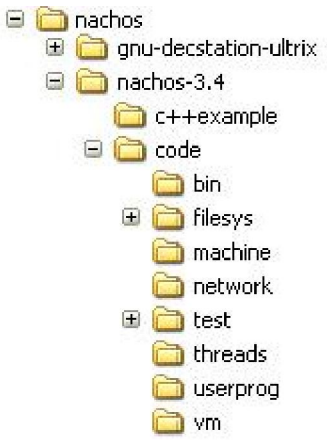
Involuntary Context Switches with the -rs Flag

To aid in testing, Nachos has a facility that causes involuntary context switches to occur in a repeatable but unpredictable way. The -rs command line flag causes Nachos to call Thread::Yield on your behalf at semi-random times. The exact interleaving of threads in a given nachos program is determined by the value of the "seed" passed to -rs . You can force different interleavings to occur by using different seed values, but any behavior you see will be repeated if you run the program again with the same seed value. Using -rs with various argument values is an effective way to force different orderings to occur deterministically .

In theory, the -rs flag causes Nachos to decide whether or not to do a context switch after each and every instruction executes. The truth is that -rs won't help much, if at all, for the first few assignments. The problem is that Nachos only makes these choices for instructions executing on the simulated machine used in the second half of the semester (see below). In the synchronization assignments, all of the code is executing within the Nachos "kernel" running as an application program on the underlying host system (e.g., a SPARC/Solaris workstation where you log in). Nachos may still interrupt kernel-mode threads "randomly" if -rs is used, but these interrupts can only occur at well-defined times: as it turns out, they can happen only when nachos kernel code calls a routine to re-enable interrupts on the simulated machine. Thus -rs may change behavior slightly, but many possibly damaging interleavings will unfortunately never be tested with -rs . If we suspect that your code has a concurrency race during the demo, we may ask you to run test programs with new strategically placed calls to Thread::Yield.
(http://www.cs.duke.edu/~chase/cps110-archive/nachos-guide/nachos-labs-14.html)

6 Helping stuff for the Nachos developer

The Nachos package in the nachos directory has several main subdirectories:



In the initial distribution, you will found a **Makefile**. You can look at the comments in the Makefile ("Compiling and Running Nachos") for information on how to compile Nachos.

The Nachos source code is found in the **code** directory, which is organized into several subdirectories:

- **machine** contains the machine simulation (i.e. the machine "hardware")
- **threads** contains the main program to start Nachos and the core Nachos kernel, including the kernel itself, the implementation of threads, the code to handle context-switching, and synchronization primitives
- **userprog** contains code needed to support user programs and multiprogramming, including support for process address spaces and system calls

- **filesys** contains the implementation of a rudimentary file system

You may add to and modify files in any of these directories except the machine directory - developers can't usually make changes to the hardware, so we can't either! Check there is a warning written at the start of those files

```
//  DO NOT CHANGE -- part of the machine emulation
```

The **test** directory contains sample user programs, along with a Makefile for compiling them.

As you sift through the Nachos source code, the **grep** and **find** commands can be very useful for finding files, function calls, method implementations, or other

things for which you know the name of but can't locate. (In particular, method implementations aren't always where you might expect - e.g. Machine::Run() is in mipssim.cc even though the Machine class is defined in machine.h.)

Here is a sample code you can use to find occurrences of an identifier inside the source tree.

Code

```
#!/bin/sh
# identifier search only, space not supported in string
# usage: findstring <string> *.cpp *.h
st="$1"
shift
for xt in $*
do
    a="find . -name "$xt" -print"
    findfile=`$a`
    for cfile in $findfile
    do
       cnts="grep -c $st $cfile"
       cnt=`$cnts`
       if [ $cnt -gt 0 ]
       then
              echo "---------------------------------- "
              printf "In file "
              echo "$cfile"
              cmd="grep -n $st $cfile"
              res=`$cmd`
              echo "$res"
       fi
    done
done
```

6.1 Compiling and Running Nachos

The instructions for compiling Nachos can be found in the **Makefile** in the **code** directory. They are summarized below. Since Nachos consists of many files, we will be using **make or gmake** to automate the compilation process.

6.2 Making your ubuntu OS ready for Nachos development

Fedora will compile nachos straight way; but for Ubuntu Linux you need slight modifications. **Firstly,** Ubuntu does not come with gmake, it comes with make. Nachos uses gmake. But make and gmake are same here. So you can create a soft link of make named gmake.

```
sudo ln -s /usr/bin/make /usr/bin/gmake
```

Secondly, your Ubuntu linux may not have g++ installed. Nachos uses C++, so g++ is needed for its compilation. Write g++ in the command prompt, and if you find that you do not have this program, install it by :

```
sudo apt-get install g++
```

6.3 Compiling Nachos for the First Time

1. Make sure we are in the code directory of the Nachos distribution.
2. **Type gmake** and press enter**.** This compiles Nachos and produces the executable **nachos**. **(gmake or make both works)**

We may get the following compiler warning:

```
../code/threads/synch.cc: In member function
`bool Lock::IsHeldByCurrentThread()':
../code/threads/synch.cc:196: warning: control reaches end of non-void function
```

It is safe to ignore this for now - it should go away once we complete the implementation of the Lock class in the first Nachos assignment.

Try running Nachos: nachos -u will print out some information about Nachos' commandline arguments, or you can run nachos -T 1 to run the built-in thread test - you should see output about threads 0 and 1 looping.

6.4 Recompiling Nachos After Changes

If we have only changed C++ code in existing Nachos files, then

1. Make sure we are in the directory where the changed source code is.
2. **Type gmake (or make)**

If we have added any new files (.cc or .h files) or added or deleted #includes in any of the existing files, then Refer to section “6.5 Adding new files to Nachos source tree”

6.5 Adding new files to Nachos source tree

When you modifty existing nachos code, compilation is no problem. But if you want to add new file/class, you need to specify to compile it in the Makefile. Observe one thing, that the compiled outputs .o files reside in several folders, but the source file is exactly in one folder. For example machine.cc is in machine folder but after compilation machine.o is in vm, threads, userprog and so on. This is because, purpose of threads, userprog or vm or network directory is different. Each one contains the necessary version required in that folder. For example code

conained inside blocks `#ifdef USER_PRORGRAM …. #endif` will be compiled in userprog folder.

Check first lines of Makefile in several folders. Here 2 given. You can easily see there are differences in DEFINES, which take effect during compilation.

threads

```
DEFINES = -DTHREADS
INCPATH = -I../threads
- I../machine
HFILES = $(THREAD_H)
CFILES = $ (THREAD_C)
C_OFILES = $(THREAD_O)
include ../Makefile.common

include ../Makefile.dep
```

userprog

```
DEFINES = -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB
INCPATH = -I../bin -I../filesys -I../userprog -I../threads
- I../machine
HFILES = $(THREAD_H) $(USERPROG_H)
CFILES = $ (THREAD_C) $ (USERPROG_C)
C_OFILES = $(THREAD_O) $(USERPROG_O)
# if file sys done first!
# DEFINES = -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS
# INCPATH   =   -I../bin   -I../filesys   -I../userprog   -
I../threads -I../machine
# HFILES = $(THREAD_H) $(USERPROG_H) $(FILESYS_H)
# CFILES = $(THREAD_C) $(USERPROG_C) $(FILESYS_C)
# C_OFILES = $(THREAD_O) $(USERPROG_O) $(FILESYS_O)

include ../Makefile.common


include ../Makefile.dep
```

Check the Makefile in code folder. You can see, this makefile contains the main calls necessary to compile the entire tree. Makefile in code folder merely refers the makefiles in nested folders. And also observe – Makefile.common contains info about what output files are necessary for each folder.

If you have checked all these, you are ready to understand the following walk-through. Here we work in the threads directory, because assignment-1 was on threads and synchronization. Classes may be added to any directories (except machine) as needed.

Adding a new Class Abul in nachos source tree

Step 1

Goto threads directory and write `Abul.h` for class header and `Abul.cc` for class implementation.

```
// Sample Abul.h :

class Abul{

public :

 char *getAbul();

};
```

```
// Sample Abul.cc :

#include "Abul.h"

char * Abul::getAbul() {

return "Abul" ;

}
```

Step 2

(Assuming you wrote the class correctly and no there is no compilation error)

Goto code folder and open `Makefile.common`. Find a block like

```
THREAD_H =../threads/copyright.h\
      ../threads/list.h\
      ../threads/scheduler.h\
```

Add ../threads/Abul.h in this block. (If you add it at the last, make sure to add a backslash with the entry that was previously the last entry. If you add it somewhere before the last, add it like `../threads/Abul.h\`

Find block like

```
THREAD_C =../threads/main.cc\
      ../threads/list.cc\
      ../threads/scheduler.cc\
```

Add ../threads/Abul.cc as described above.

Find block like

```
THREAD_O =main.o list.o scheduler.o synch.o synchlist.o …..
```

Add `Abul.o` with this list. If you break a line make sure to append a backslash.

Step 3

Goto threads directory

Execute

```
gmake  depend
```

Open the Makefile in threads directory. And you will find that there is a block

	`Abul.o: ../threads/Abul.cc ../threads/Abul.h` automatically created. Whenver you include new headers in Abul.cc or include Abul.h somewhere in original nachos code (that's the ultimate reason for adding a new class, anyway) you need to execute `gmake depend` again, so that the Makefile will be updated accordingly.

Step 4	Now you can stay at threads directory and execute gmake to see whether it compiles. If everything is ok, then you will see an incremental compilation that compiles files `Abul.h` and `Abul.cc` only.
	
	Now we will test them via `threadtest.cc`
	
	Open threadtest.cc
	
	Add a new include there
	
	`#include "Abul.h"`
	
	And inside `void ThreadTest()` function write
	
	`Abul myAbul;`
	
	`printf("%s\n",myAbul.getAbul());`

Step 5	Execute
	
	`gmake depend`
	
	Execute
	
	`gmake`
	
	To see this great work in action execute
	
	`./nachos -t`

The **make depend** step is important for updating dependencies if you have done something that might cause them to change. It is always safe to do make depend even if you have only changed existing C++ code; it just shortens the compilation process if you only do it when necessary.

6.6 Using an IDE

Nachos source tree is large and we found that, it is just a pain (not comfortable though a bit educative) to search, browse and manage this monster "plainly" during development process. So it is instructive to use some automatic weapons (Also Known As IDE)

Simply using gedit or nedit doesn't help much. We found using Eclipse or Netbeans useful. Eclipse is a rich platform for development in several languages including Java, C, C++, Python, PHP, Prolog and so on. Actually a plugin is all, that is needed to add support for a new language may even be designed by you. By default, Fedora includes a "Fedora Eclipse" configured for java and C++ projects, but we found downloaded versions of Eclipse CDT better (www.eclipse.org). Netbeans is relatively new, and sometimes its performance is even better. You can download Netbeans for C++ development from its website.

Eclipse and Netbeans, both features intellisense, code folding (like visual studio), highlighting, browsing and visualization options. When an object name is followed by period "." it automatically shows the available member variables and functions. When a function name is followed by parentheses, it automatically shows parameter list. Not only these, select an identifier in the main edit window, and right click the mouse – you will find options like "Open Declaration" "Open Type Hierarchy" or "Open Call hierarchy" and so on. "Open Declaration" will find the declaration of that variable/function. We found that these functionalities make studying/following the code comfortable which in turn makes life easier.

But the problem is, you have to configure the IDE for the first time. Nachos wasn't developed using Netbeans / Eclipse, so you can't just load it with all the bells and whistles automatically. You have to open a CDT project in Eclipse Europa / create a project from existing makefile in Netbeans and import nachos source tree there, set some options and so on. But once you configure it, a project and settings folder will be created and it will persist. Whenever you close the IDE, it will save the present view and state of the workspace and next time you open eclipse, it will be restored automatically.

Nachos generally needs to compile and run with several options like –c, -x, -rs, -t according to our preference which may change during development. So it is better to use eclipse just for editing and browsing purpose. We used to open a terminal window along with eclipse, setting the "current directory" equal to nachos workspace. Whenever compilation was needed we moved to the terminal and issue compilation/run options.

If you don't find Netbeans / Eclipse interesting, you can try emacs or KATE (KDE Advanced Text Editor) or KDevelop. KATE has medium support for code highlighting and browsing history for files.

6.7 Important Notes:

If make is behaving strangely or after running Nachos if it doesn't seem like the changes we just made have taken effect, we can do the following:

1. Make sure we are in the build directory of the Nachos distribution.
2. **Type make clean**. These removes all of the .o files generated by the compilation process so that the next make will recompile everything from scratch.

3. **Type make depend.**
4. **Type make.** Compilation will take a bit longer since everything must be recompiled, but you'll be working with a clean slate.

7 References

The Nachos source code itself is a very good documentation which will help you most to understand and develop Nachos. Besides, **A Road Map Through Nachos** by Thomas Narten and **Nachos Primer** by Matthew Peters, Robert Hill, Shyam Pather are very good documents. You are highly instructed to study those.

Special thanks to Sukarna Barua and Tanvir Al Amin.