NACHOS REPORT

Anokhi

Priyanka

Nishant

Mansi

Shikhar

Geet

Harsh

Varsh

Priya

Aakansha

Contents

1) What is NACH-OS?

NachOS (Not Another Completely Heuristic Operating System) is an instructional software developed for the understanding of students and experience the implementation of real operating system. Nachos is a multitasking operating system simulation that runs in a single UNIX process. It implements a multi-process model by using user-level threads. Thus, each user-level thread is viewed as its own process under Nachos.

2) Threads and context switching Module

2.1) What is threads?

- Nachos provides a kernel threading package, allowing multiple tasks to run concurrently. Once the user-processes are implemented, some threads may be running the MIPS processor simulation.
- A Nachos thread can be viewed as a little process. Threads are created and destroyed with the Fork and Exit system calls, respectively. The thread object also contains a set of structures similar to a Unix processes PCB.
- The Nachos scheduler maintains a data structure called a *ready list*, which keeps track of the threads that are ready to execute. Threads on the ready list are ready to execute and can be selected for executing by the scheduler at any time. Each thread has an associated *state* describing what the thread is currently doing.
  Nachos' threads are in one of four states:
  - **READY:**
    - The thread is eligible to use the CPU (e.g, it's on the ready list), but another thread happens to be running. When the scheduler selects a thread for execution, it removes it from the ready list and changes its state from READY to RUNNING. Only threads in the READY state should be found on the ready list.
  - **RUNNING:**
    - The thread is currently running. Only one thread can be in the RUNNING state at a time. In Nachos, the global variable *currentThread* always points to the currently running thread.
  - **BLOCKED:**
    - The thread is blocked waiting for some external event; it cannot execute until that event takes place. Specifically, the thread has put itself to sleep via *Thread::Sleep()*. It may be waiting on a condition variable, semaphore, etc. By definition, a blocked thread does not reside on the ready list.
  - **JUST_CREATED:**
    - The thread exists, but has no stack yet. This state is a temporary state used during thread creation. The *Thread* constructor creates a thread, whereas *Thread::Fork()* actually turns the thread into one that the CPU can execute (e.g., by placing it on the ready list).
- In non-object oriented systems, operating systems maintain a data structure called a *process table*. Process (thread) table entries contain all the information associated with a Thread. (e.g., saved register contents, current state, etc.).

- In contrast to other systems, Nachos does not maintain an explicit Thread table. Instead, information associated with thread is maintained as (usually) private data of a *Thread* object instance. Thus, where a conventional operating system keeps thread information centralized in a single table, Nachos scatters its ``**thread table entries''** all around memory; to get at a specific thread's information, a pointer to the thread instance is needed.

2.2) What is context switching?

- Switching the CPU from one thread to another involves suspending the current thread, saving its state (e.g., registers), and then restoring the state of the thread being switched to. The thread switch actually completes at the moment a new program counter is loaded into PC; at that point, the CPU is no longer executing the thread switching code, it is executing code associated with the new thread.

- The routine *Switch(oldThread, nextThread)* actually performs a thread switch. *Switch* saves all of *oldThread*'s state so that it can resume executing the thread later, without the thread knowing it was suspended.

- *Switch* does the following:

1. Save all registers in *oldThread*'s *context block*.
2. What address should we save for the PC? That is, when we later resume running the just-suspended thread, where do we want it to continue execution? We want execution to resume as if the call to*Switch()* had returned via the normal procedure call mechanism. Specifically, we want to resume execution at the instruction immediately following the call to *Switch()*. Thus, instead of saving the current PC, we place the return address (found on the stack in the thread's activation record) in the thread's context block. When the thread is resumed later, the resuming address loaded into the PC will be the instruction immediately following the ``call'' instruction that invoked *Switch()* earlier.
3. Once the current thread's state has been saved, load new values into the registers from the context block of the next thread.

4. At what exact point has a context switch taken place? When the current PC is replaced by the saved PC found in the process table. Once the saved PC is loaded, *Switch()* is no longer executing; we are now executing instructions associated with the new thread, which should be the instruction immediately following the call to *Switch()*. As soon as the new PC is loaded, a context switch has taken place.

- The routine *Switch()* is written in assembly language because it is a machine-depended routine. It has to manipulate registers, look into the thread's stack, etc.

2.3) Execution steps

- In the Thead folder we have 4 main files- main.cc, thread.cc,threadtest.cc and scheduler.cc.

- The execution starts from main function which calls 2 functions:-
    1) Initialize() which is defined in system.cc. Initialize() function defines all the global data-structures for scheduler, Interrupts, Timer and statistics. It also initializes currentThread with "main" and sets its status as running.

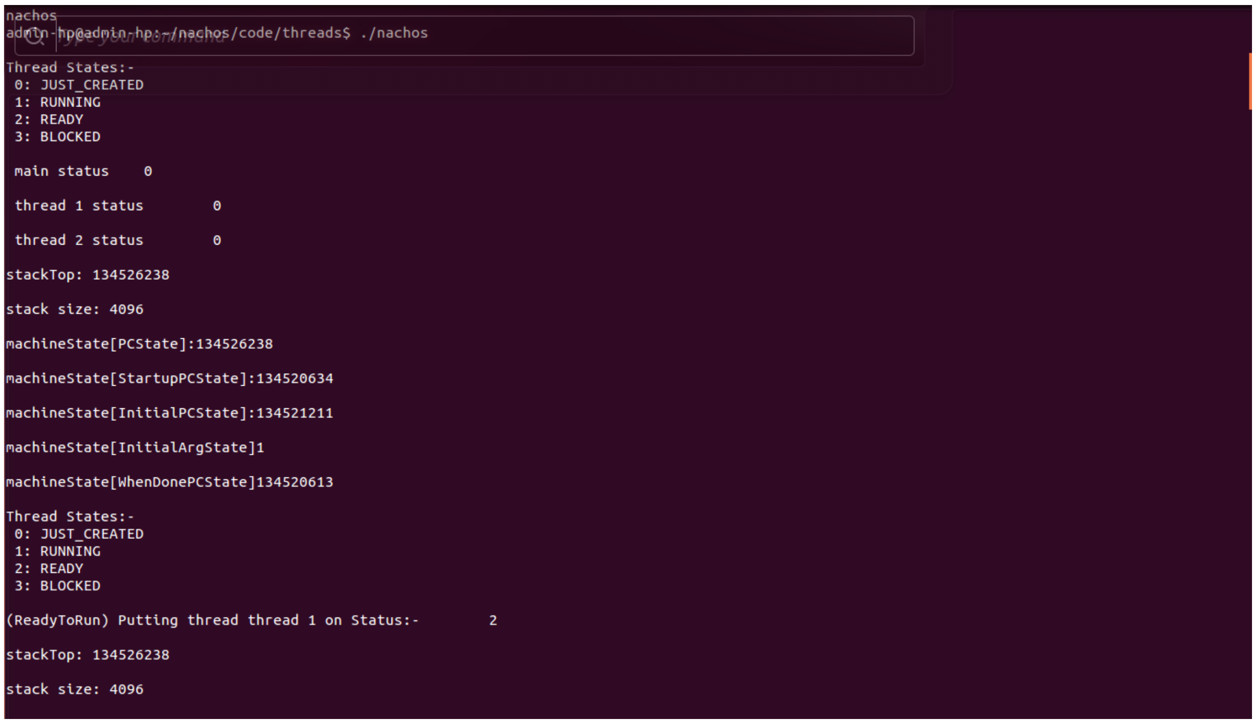
    2) ThreadTest() function which is in threadtest.cc. This function in turn calls ThreadTest1() function. It is in this function that we fork 2 new threads and each forked thread call the function - SimpleThread();

- SimpleThread() in turn calls the yield() function which pre-empts the current running thread. It selects the nextThread to run from the queue maintained in **list.cc** . It also calls readytorun() and run() functions which set the status of the current thread to "READY" and the next thread as running.

- run() function which is defined in **scheduler.cc** is responsible for context switching. It contains the all-important 'switch' function which saves the register contents of current running thread and loads the register contents of the next thread selected for running.

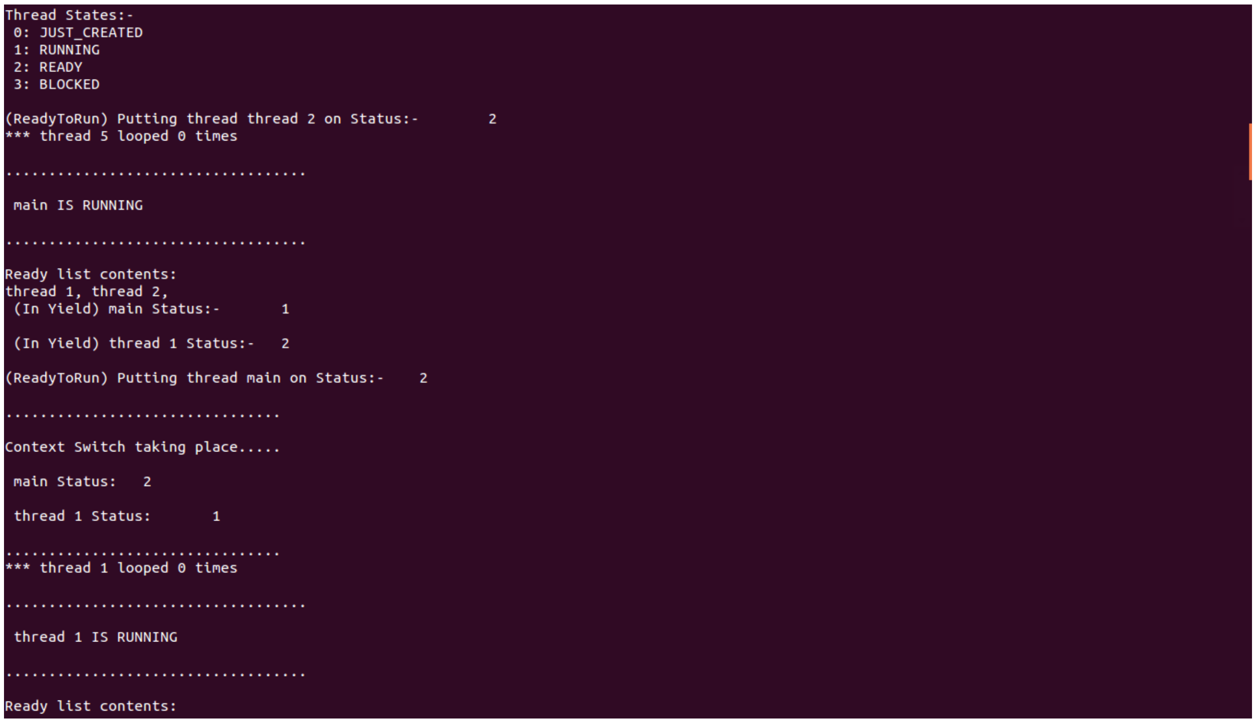
- This process continues untill the for loop in SimpleThread() function ends. Once it ends the main function finishes and calls the Finish() function. This function initializes the 'threadTobeDestroyed' with 'currentThread'.
- It then calls the sleep() function. This function is responsible for changing the status of the thread to 'blocked'. It also finds the next 'ready' thread form the queue if it all there is one and makes it 'running'.

2.4) Output



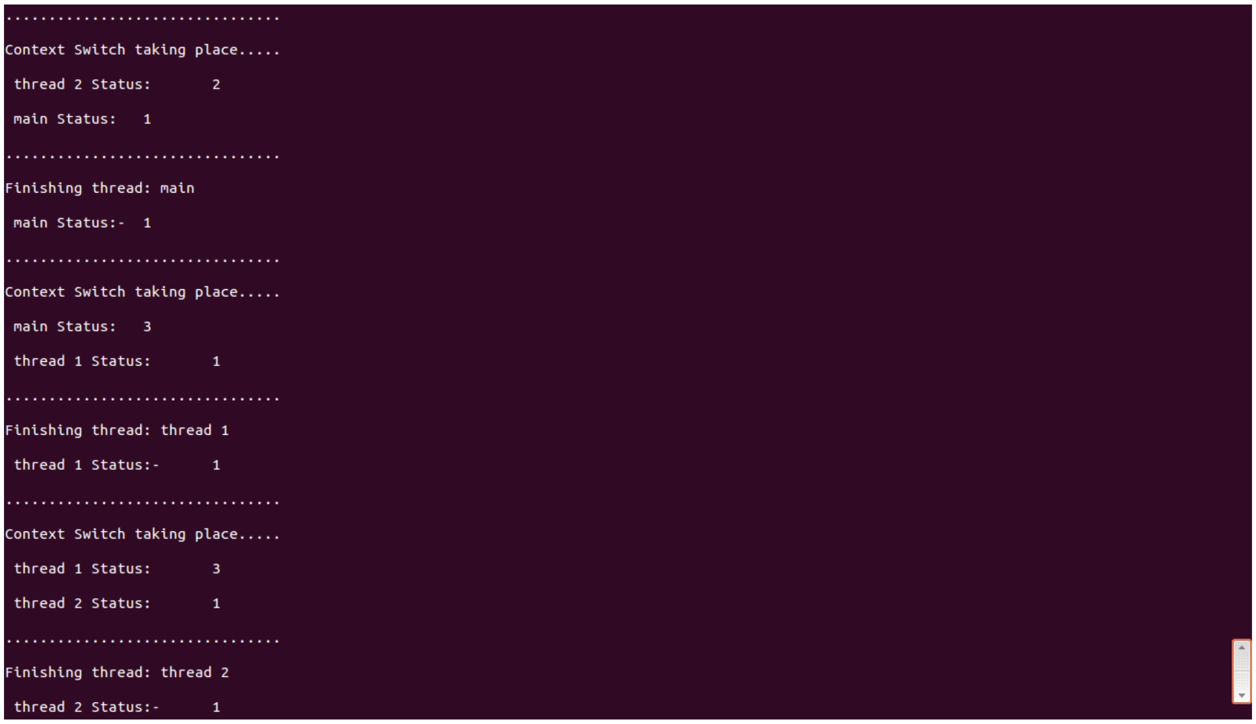
```
nachos
admin-hp@admin-hp:~/nachos/code/threads$ ./nachos

Thread States:-
 0: JUST_CREATED
 1: RUNNING
 2: READY
 3: BLOCKED

 main status    0

 thread 1 status        0

 thread 2 status        0

stackTop: 134526238

stack size: 4096

machineState[PCState]:134526238

machineState[StartupPCState]:134520634

machineState[InitialPCState]:134521211

machineState[InitialArgState]1

machineState[WhenDonePCState]134520613

Thread States:-
 0: JUST_CREATED
 1: RUNNING
 2: READY
 3: BLOCKED

(ReadyToRun) Putting thread thread 1 on Status:-        2

stackTop: 134526238

stack size: 4096
```

Here we can see the different states of threads and also the stack size.

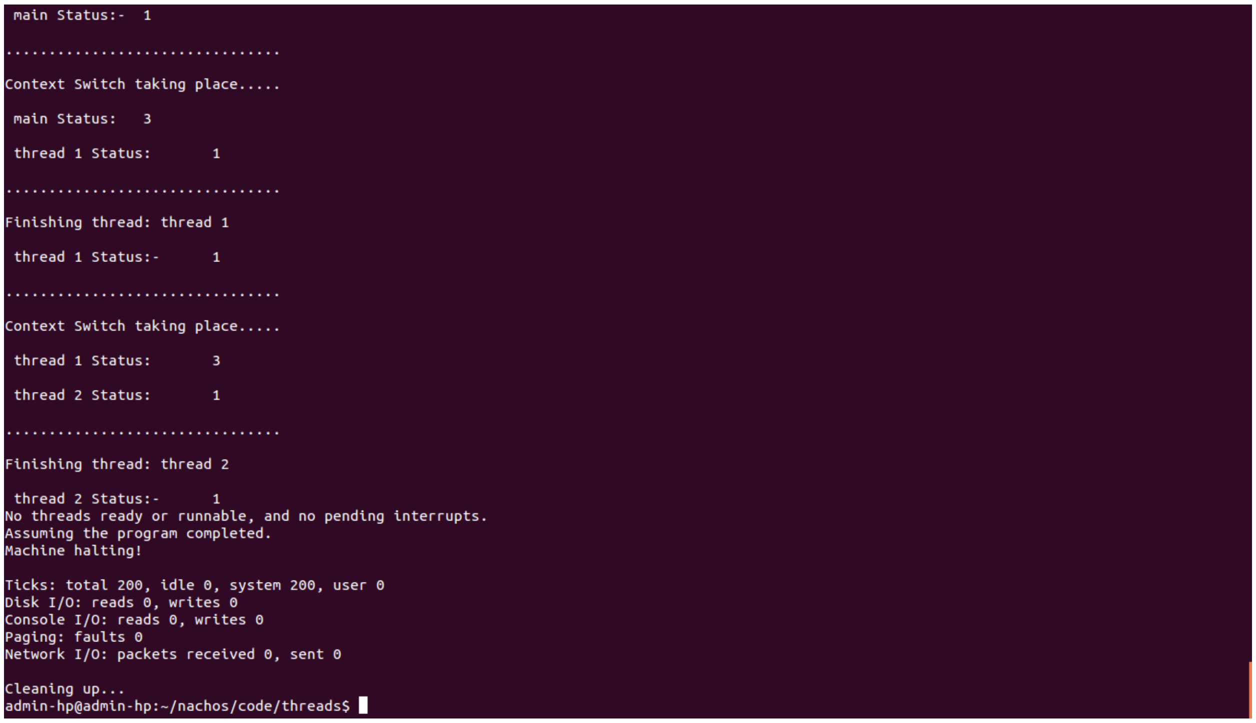


```
Thread States:-
 0: JUST_CREATED
 1: RUNNING
 2: READY
 3: BLOCKED

(ReadyToRun) Putting thread thread 2 on Status:-        2
*** thread 5 looped 0 times

..................................

 main IS RUNNING

..................................

Ready list contents:
thread 1, thread 2,
 (In Yield) main Status:-       1

 (In Yield) thread 1 Status:-   2

(ReadyToRun) Putting thread main on Status:-    2

...............................

Context Switch taking place.....

 main Status:   2

 thread 1 Status:       1

...............................
*** thread 1 looped 0 times

..................................

 thread 1 IS RUNNING

..................................

Ready list contents:
```

Here we can see the particular which thread is running and its status and context switching.


```
...............................
Context Switch taking place.....
 thread 2 Status:       2
 main Status:   1
...............................
Finishing thread: main
 main Status:-  1
...............................
Context Switch taking place.....
 main Status:   3
 thread 1 Status:       1
...............................
Finishing thread: thread 1
 thread 1 Status:-      1
...............................
Context Switch taking place.....
 thread 1 Status:       3
 thread 2 Status:       1
...............................
Finishing thread: thread 2
 thread 2 Status:-      1
```

Here context switching is happening between different threads and its status is also changing.


```
 main Status:-  1
...............................
Context Switch taking place.....
 main Status:   3
 thread 1 Status:       1
...............................
Finishing thread: thread 1
 thread 1 Status:-      1
...............................
Context Switch taking place.....
 thread 1 Status:       3
 thread 2 Status:       1
...............................
Finishing thread: thread 2
 thread 2 Status:-      1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 200, idle 0, system 200, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
admin-hp@admin-hp:~/nachos/code/threads$
```

Here is the final output after completing the all threads.

2.5) Failures

- We faced significant challenge in running user program on Nachos. The basic technique for running a user program is to first compile it using the mips-gcc (for c type files). We then get a file which is in '.coff' format (i.e. common object file format). We then need to convert it to '.noff' format (i.e. Nachos object file format). The resulting '.noff' file is the executable file which can be run on the Nachos. Running the the user program is then pretty straightforward as we need to make a few changes in Makefile and then run it using the command - ./nachos -x.

- However, we faced problems in the compilation part itself. We were not able to compile successfully the user program. Furthermore, even if we had succeed in compiling the program, we would have still not been able to run it as the programs internally made use of some of the system calls which were yet not implemented.

3) Mutual exclusion model

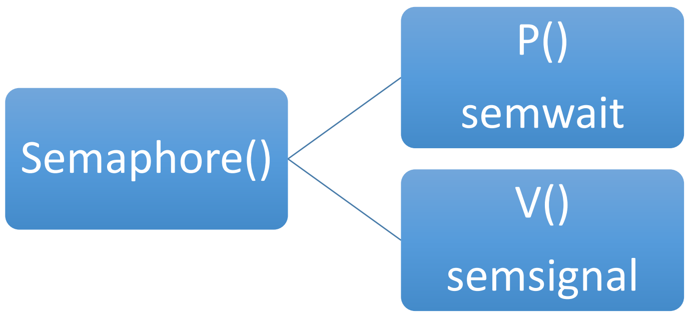
3.1) Introduction

- All the threads of a particular process share the same address space.
- Threads (like processes) have code, memory and other resources associated with them.
- One big difference between threads and processes is that global variables are shared among all threads.
- Because threads execute concurrently with other threads, they must worry about **synchronization** and **mutual exclusion** when accessing shared memory.
- Nachos provide semaphores to achieve synchronization. It also has a public interface to locks and condition variables.
- These locks and condition variables can be used to provide a synchronization abstraction hiding and managing most of the complexity and thus providing high level operations for further use in other modules such as networking.
- It is sometimes useful to allow multiple threads of control to execute concurrently within a single process.

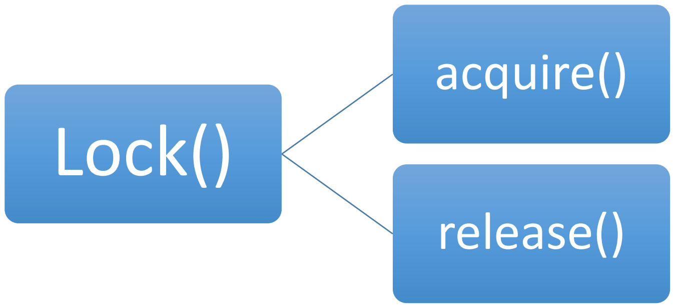
3.2) Understanding the flow

- Using the semaphore class and the public interface we have implemented locks and condition variable.

- Here is a flow that shows how the lock and condition variable work using the already defined semaphore class :

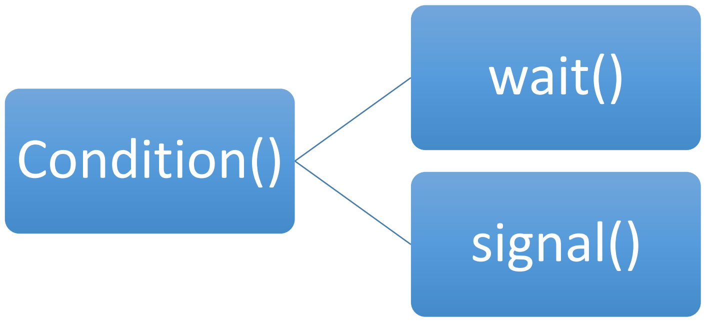


- The above structure defines the semaphore class. It consists of the two functions P() and V().The P() function is equivalent to semwait and V() is equivalent to semsignal.



- The acquire and the release function consist of two semaphore variables mutex and sleep. Mutex is used to ensure that while one of the threads is checking whether the shared resource is free or not, no other thread is allowed to do checking on that resource. The sleep semaphore keeps track of the blocked threads.

    - Thus when a thread acquires a lock no other thread can check the status of the shared resource until the thread which has acquired the lock finishes the checking and more importantly no other thread can use the resource until the thread releases the lock.



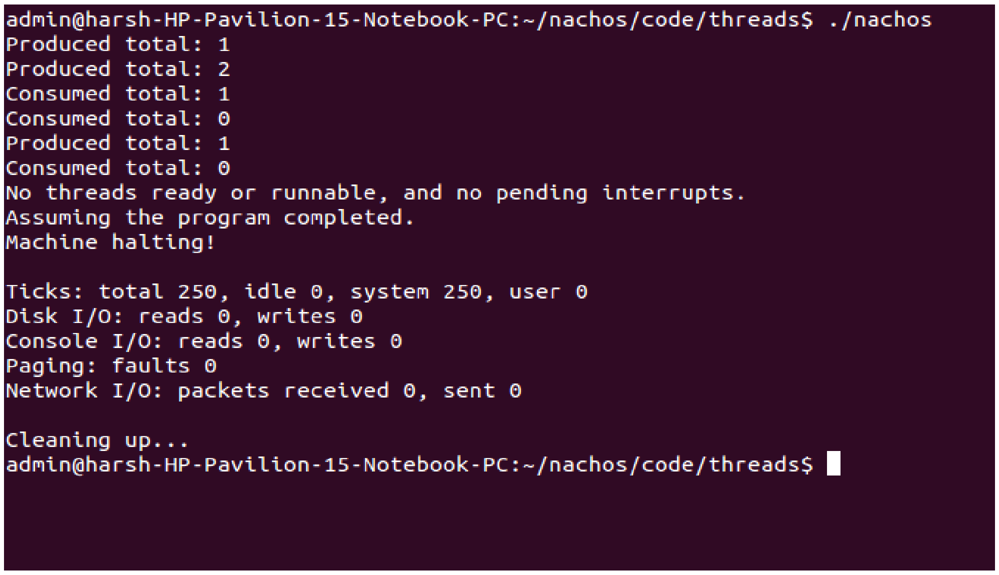
- The condition class consist of wait() and signal() functions. The wait() function suspends the execution of a thread on a certain condition and maintains the queue of similar threads waiting on the same condition. The signal() function releases a thread from a queue when the desired condition for which the thread was waiting for is met. Thus a queue for each condition is maintained.

- Thus using the given structure of semaphore we have created a monitor like structure to ensure synchronization.
- The structure is implemented in the file named synch.cc and synch.h which are files exclusively for thread synchronization.

3.3) Execution Steps

- Now testing of this monitor structure is important and hence we have implemented the producer-consumer problem using this monitor structure.
- We created a sequence which forces monitor to block some threads as our goal is to test monitor not producer consumer problem.
- The producer-consumer test program is written in the threadtest.cc file.

3.4) Output



```
admin@harsh-HP-Pavilion-15-Notebook-PC:~/nachos/code/threads$ ./nachos
Produced total: 1
Produced total: 2
Consumed total: 1
Consumed total: 0
Produced total: 1
Consumed total: 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 250, idle 0, system 250, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
admin@harsh-HP-Pavilion-15-Notebook-PC:~/nachos/code/threads$
```

3.5) Failures

- We first tried to implement software managed TLB in the virtual memory module and studied the memory structure of the nachos for the same. Further we identified the files that simulated the memory for nachos and complied and run that files. The files ran fine, but then we realized that to understand and check how paging works with nachos memory structure we need to run a user program.
- So we tried to compile the user program, but to see errors. We contacted our group members who were working on user program who informed us that they have been trying to solve this error from days but were not successful. So we tried our hands to solve this error, but we too were not successful and hence we needed to drop the module.
- Compilation was to be done by mips-gcc but it returned an error “cc1 not found”.

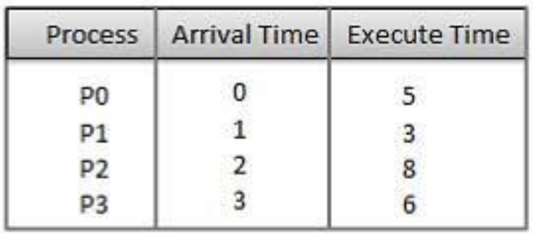
4) Scheduling

4.1) Introduction

Scheduling is the method according to which the threads and processes are given access to the system resources. It is majorly carried out for carrying out multitasking. The scheduler is mainly concerned with the throughput, response time and the latency of the system. Throughput is the total number of processes that complete their execution per unit time. Response time is the time it has to wait before it is executed for the first time. Latency means the total time i.e. th waiting time plus the execution time.

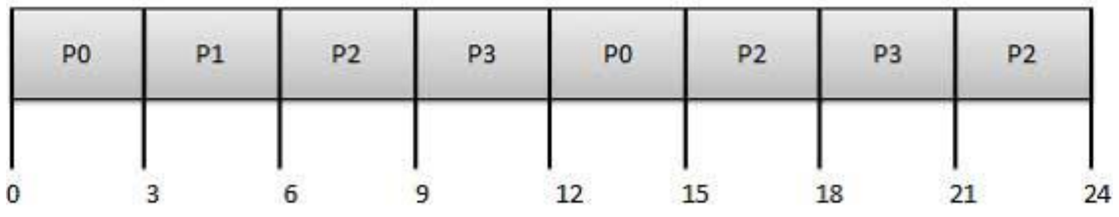
4.2) Round Robin algorithm

It is a preemptive process. Here each program is given a fixed time to execute which is called the quantum. In this algorithm, a process is executed for a given time period and then other process, which also runs for the same time period, preempts it. For the given example below, process P0 is executed for 3 seconds and then the other process waiting in the queue is given the time to execute for 3 seconds and the pattern goes on until all processes are finished.



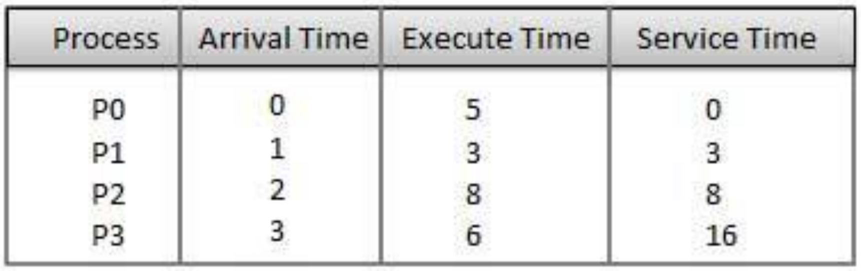
Process	Arrival Time	Execute Time
P0	0	5
P1	1	3
P2	2	8
P3	3	6

Quantum = 3

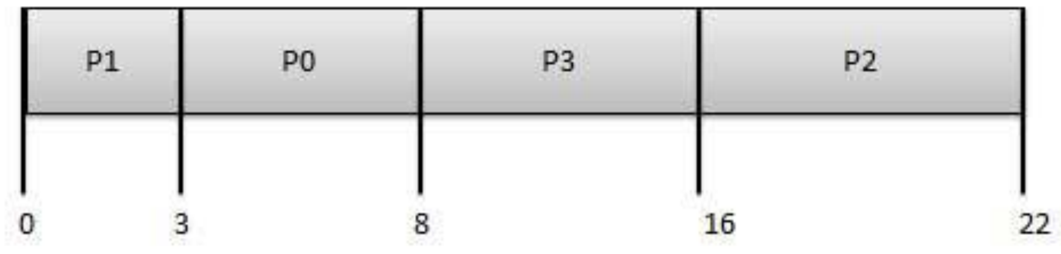


4.3) Shortest Job First algorithm

In this type of algorithm, the process with the shortest execution time is completed first. Its benefit is that small jobs are completed faster but long processes have to wait for a very long time as this process is non preemptive in nature. One issue with this algorithm is that the processor should know what time will the process take to complete its execution. As shown in the below example, the process with the shortest execution time is finished first. Hence the order, P1, P0, P3 and P2 which is in the ascending order of their execution time.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	3
P2	2	8	8
P3	3	6	16



4.4) Multilevel feedback algorithm

In this type of scheduling algorithm, queues are formed according to their priority and then round robin or FIFO is carried out internally.

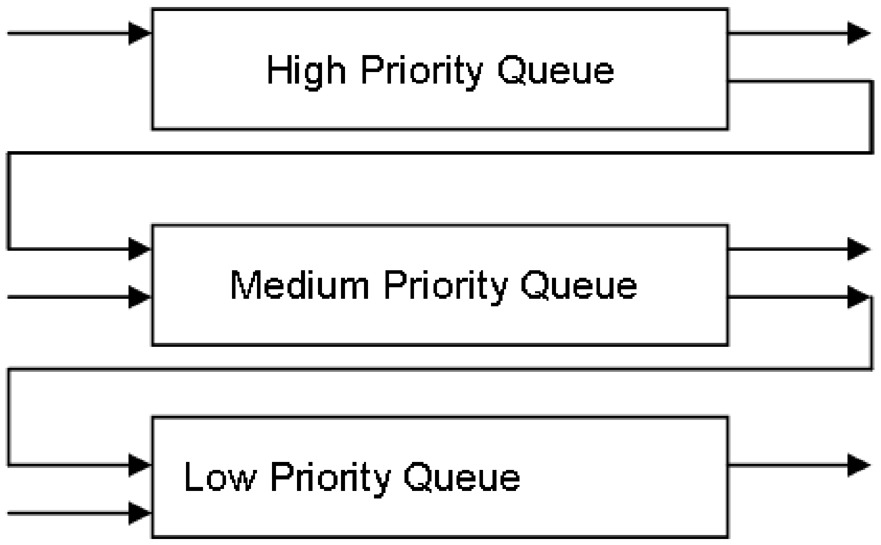
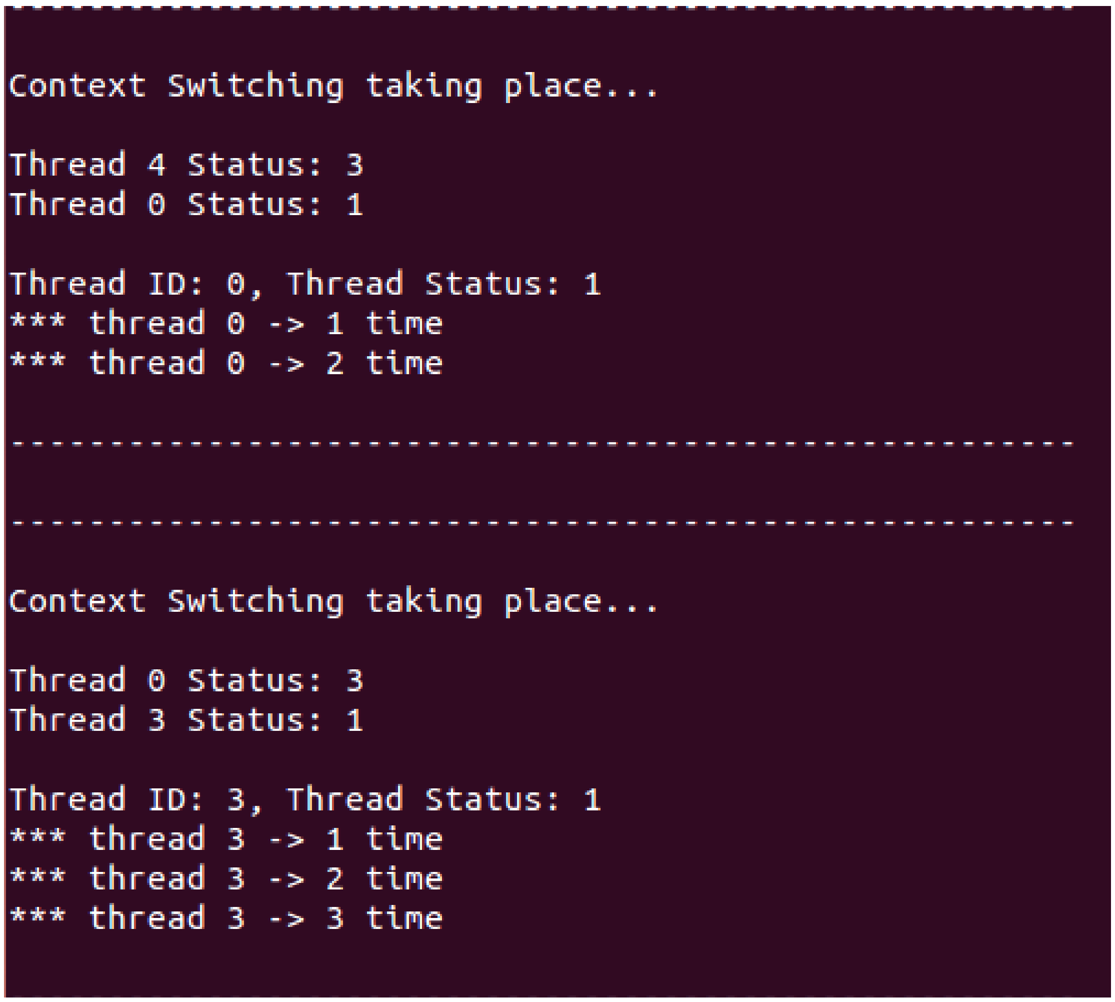


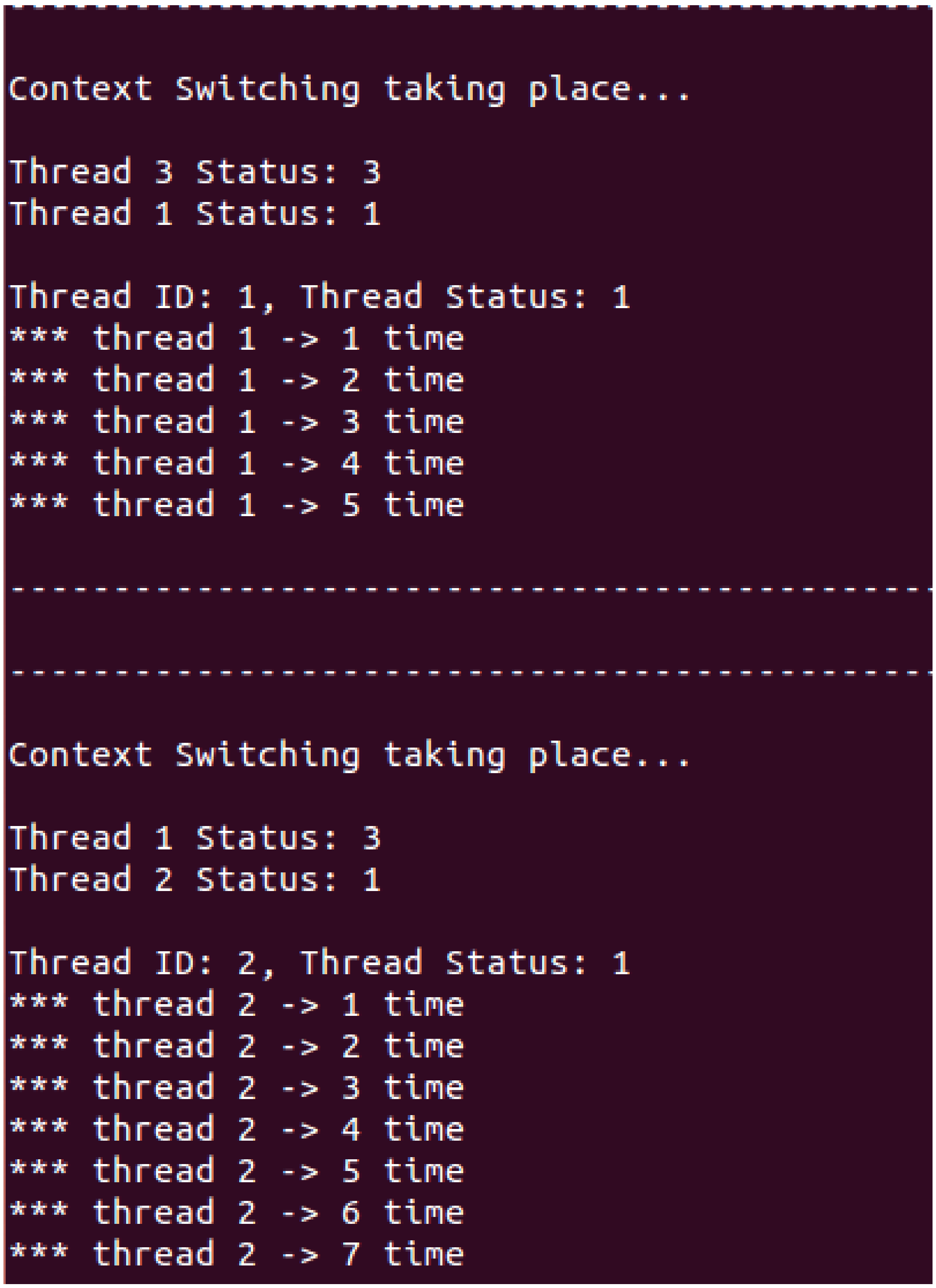
Figure – Multilevel Feedback Queue Scheduling

4.5) Scheduling in NACH-OS

- In the NachOS code, which was already available, initially, there were two threads being created in the code available. Each was running for a fixed number of times. (i.e. 5 times). First of all we increased the number of threads from two to five and tested it threadtest.cc. Then we made changes and we made the thread run multiple numbers of times according the user input. We are creating threads and assigning the thread id to their respective threads.
- The scheduling algorithm used in the original code was Round robin. We changed that algorithm and implemented the Shortest Job First (SJF) algorithm.
- We changed the nature of the thread scheduling from preemptive to non-preemptive.



```
--------------------------------------------------------------------

Context Switching taking place...

Thread 4 Status: 3
Thread 0 Status: 1

Thread ID: 0, Thread Status: 1
*** thread 0 -> 1 time
*** thread 0 -> 2 time

--------------------------------------------------------------------

--------------------------------------------------------------------

Context Switching taking place...

Thread 0 Status: 3
Thread 3 Status: 1

Thread ID: 3, Thread Status: 1
*** thread 3 -> 1 time
*** thread 3 -> 2 time
*** thread 3 -> 3 time
```



```
----------------------------------------------

Context Switching taking place...

Thread 3 Status: 3
Thread 1 Status: 1

Thread ID: 1, Thread Status: 1
*** thread 1 -> 1 time
*** thread 1 -> 2 time
*** thread 1 -> 3 time
*** thread 1 -> 4 time
*** thread 1 -> 5 time


----------------------------------------------


----------------------------------------------

Context Switching taking place...

Thread 1 Status: 3
Thread 2 Status: 1

Thread ID: 2, Thread Status: 1
*** thread 2 -> 1 time
*** thread 2 -> 2 time
*** thread 2 -> 3 time
*** thread 2 -> 4 time
*** thread 2 -> 5 time
*** thread 2 -> 6 time
*** thread 2 -> 7 time
```

As shown in the above figure, the thread status, its ID and the number of times it is executed is shown. Threads are scheduled according to their execution time. The thread with least execution time will be given higher priority and so on.

Some of the functions inserted in the original code are: -

- void List::SortedInsert(void *item, int sortKey)

To insert new thread in sorted order according to their service time into the queue

- void Thread::Fork1(VoidFunctionPtr func, int arg, int noLoop)

It works same as the fork function, but here the additional argument for set service time of thread is included.

- int reduceLoop()
  It reduces the service time of the thread and returns new service time.
- ThreadStatus getStatus()
  It returns the status of the thread.
- int noLoops()
  It returns the service time of the thread.

Scheduling algorithms used by various operating systems: -

- In NachOS, the threads use round robin algorithm.
- As a part of real time scheduling, Linux uses round robin and FIFO (First In First Out) algorithm.
- Android is based on Linux 2.6.23 kernel and it uses completely fair scheduling.
- Traditional UNIX system also uses round robin algorithm.
- Round robin scheduling algorithm is used in MacOS. MacOS and IOS share the same Darwin kernel and hence round robin algorithm is used in IOS for scheduling threads and processes.
- Windows NT uses multilevel feedback algorithm for scheduling the threads and processes.
- PintOS uses multilevel feedback algorithm for process scheduling.

4.6) Execution Steps

For testing the threads module: -

Open command prompt and do the following:

1. Go to "code" directory folder inside nachos.	cd /nachos/code
2. Type "make"	make
3. Type	./Nacho

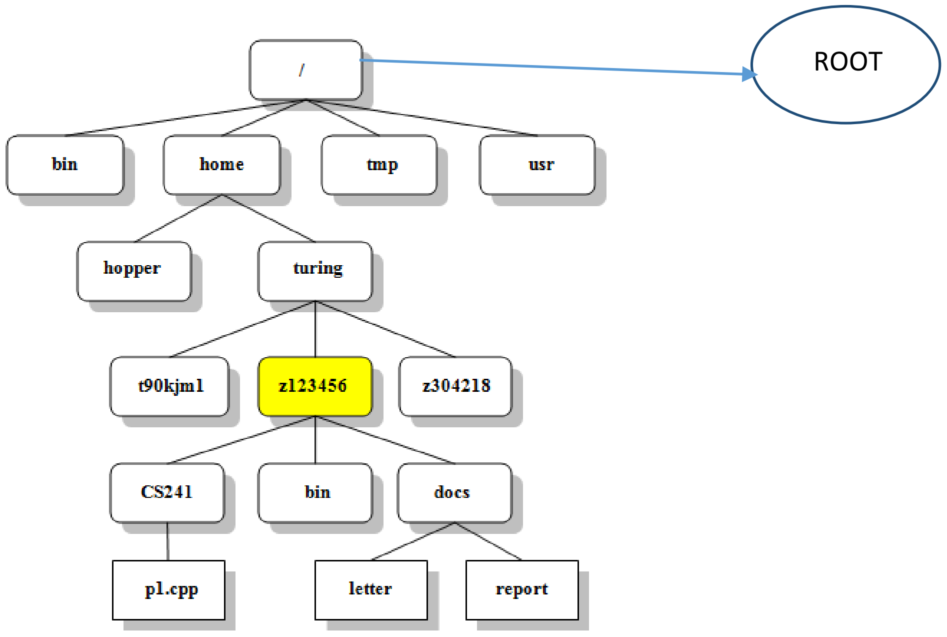
5) File System

5.1) Introduction

File System is a way used by operating system to organize files and its related information in logical manner. File related information includes its parent directory, File size, Date of creation, File access permission etc. File System stores data in such a manner so that when needed it can be retrieved easily and other operations can also be done appropriately. A group of data called chunks is kept in a structure what we call a file. File Systems can be used on many storage devices most common being hard disk. File systems keeps track of files on a partition or a disk.

5.2) UNIX File system structure

In UNIX the File-systems have a multi-leveled hierarchical structure. This structure is called directory tree. The main node is a single directory known as the root node. All the other files are the “descendants” of that node.



The root node (/) is the super user and it has access to all the files with all the permissions. The directories after that are the main directories of that Linux system. In the home directory there are some files which are not located on this system. As the file-system are independent of each other it is possible to mount some other files from other destinations. Here z123456 is the main directory for the user. The user can create as many subdirectories as needed. And the subdirectories can either have files or further subdirectories. Such assigning of files results into the above tree structure of the UNIX file-system.

Every file in the UNIX is assigned with a unique identification number known as Inode. So for any file to be uniquely identified following parameters are used:

- File name
- Directory it belongs to
- Inode of the file

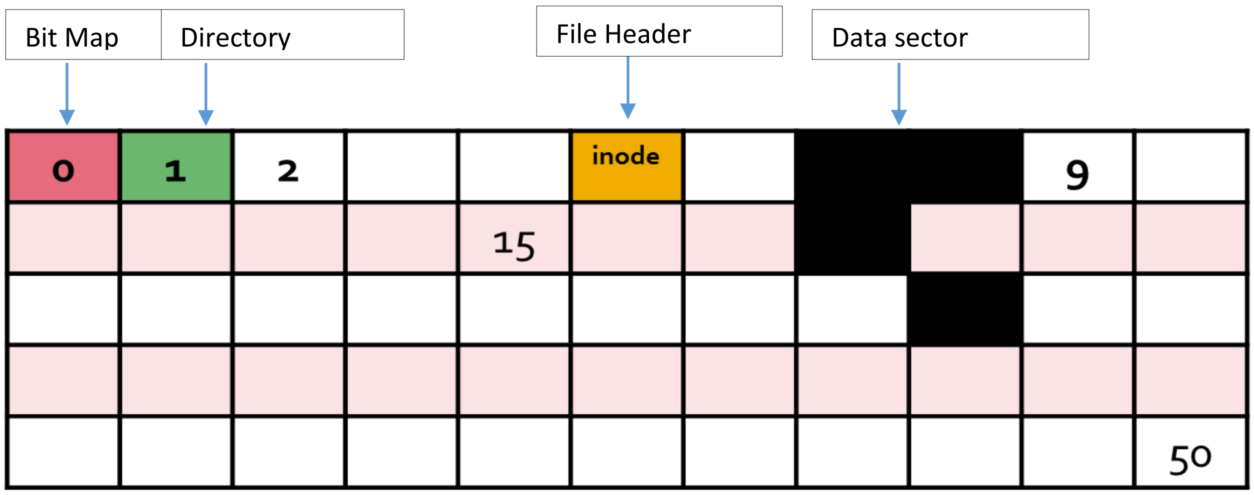
5.3) NACH-OS File system structure

Nachos file-system structure has two types of implementation. The one that we use is known as the “real” implementation. This implementation leads to a flat file structure in Nachos. As a result it does not support multiprogramming.

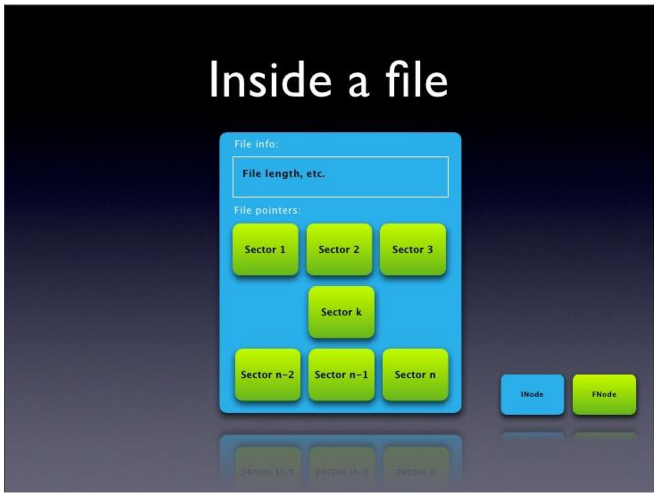
The file system in Nachos is controlled by two major data structures:

- “Root” directory – This directory is the single main directory which contains the name and position of all the files in the Nachos. Though it is said to be a directory, Nachos treats each and every chunk of information or any structure as a file. i.e. it stores everything as a file and operations are done accordingly. This root directory is allocated in the 2nd sector of the file-system.
- Bitmap – It is used to keep track which of the sectors are used or unused. It tells us whether a sector is occupied or empty with the help of which we can decide which sector to allocate to the next file. The Bitmap structure is allocated the first sector of the file-system.

Both the root directory and bitmap are treated as files itself by the Nachos. Also all files created by nachos file system has a magic number written at the starting of each file so as to identify the files uniquely as being a Nachos File.



This is the disk sector of Nachos and the boxes are the sectors in the disk. The first sector is always reserved for the bitmap and the second for the directory. The directory will have the position of the file. The position will be pointing to a sector which would have stored the file header of that file. The file header is similar to that as the Inode number in UNIX system. The file header will be then pointing to the sectors which would have stored the data of that file. All the files created in Nachos are stored in a similar manner and thus the file structure results into a flat structure.



The maximum size of the file would be restricted by number of pointers that would fit into one disk sector.

5.4) Brief of file codes of NACH-OS file system

```
   ---------- FileSystem ----------
    |                          |
 freeMapFile -------------------- directoryFile
    |            |                 |
  BitMap       OpenFile          Directory
                |                 |
             FileHeader        DirectoryEntry
```

1) <u>filesys.h filesysy.cc</u>

It is a top level interface of the file system. This file has the routine to manage the overall operation of the file system. Its basic function is to define the bitmap and directory and assign them with sectors. It is also responsible for keeping both of them open while the Nachos is running. It performs all the operations needed on both these data structures and if there is some change then it write backs to the disk.

2) <u>openfile.h openfile.cc</u>

This file has the routine to help open any file in the Nach-OS. To perform any kind of operation on a file we need to first open the file and that is where this routine is needed. Once the needed operations are performed we need to close the file. The routine will close the file by deleting the data structure of that file which it created while opening the file. It translates the file read and write into disk sector read and write.

3) <u>syncdisk.h syncdisk.cc</u>

This file has the routine to synchronously access the disk. The physical disk that is present in the memory is asynchronous in nature meaning if an interrupt comes then it will stop its present task and attend the interrupt. With this routine we are able to make the interrupt request wait till the present task is finished. This is important because the Nachos disk can handle only one request at a time and due to the interrupt it might fail.

4) <u>filehdr.h filehdr.cc</u>

This file has the routine to manage the file header of the file. It basically manages the data structure (file header) representing the layout of the files data. It is responsible for initializing the file header, giving it the correct value and pointing it to the correct sectors which will be storing the data for the file. It also helps us fetch some file from the disk with the help of its header value. In Nachos the file header does not keep track of the file permission, file owner, last modified date, etc and thus such functions are not performed by this routine.

5) directory.h directory.cc

This file has the routine to manage the directory data structure of the file-system. It has a constructor which initializes an empty directory. Any time a new file is made this routine is responsible for storing that file name and position into the directory. It also helps us fetch some file from the directory or find some file in the directory. It basically transfers the file name into the disk header files.

6) system.h system.cc

This file has a routine responsible for the initialization of the Nachos and the clean-up processes needed. It is responsible for reading the command line arguments while execution and then calling the proper needed function according to the input. It also helps in handling the interrupts faced by the Nachos. All the needed pointers for running the file system from all the file-system files are declared here. It is also responsible for asserting the code for any wrong input. All our major modifications have been made in this file.

7) bitmap.h bitmap.cc

This file has routine responsible for managing the bitmap data structure. It initializes the bitmap and assigns every sector with 1 or 0 according to whether it is occupied or not. It helps us find which sector to place the next file into. It also retrieves back the sector number for a particular file.

5.5) Functions implemented

1. Creating a File:

As explained above Nachos has a flat directory structure which means all the files can be stored only under a single directory. We understood how Nachos creates a directory and extended it further to create a file and store its information in the directory.

There is a directory structure initialized in directory.h which has following information stored:

a. inUse: This shows whether any file is stored in that particular sector or not. “1” if true, “0” if false.

b. sector: sector number of the file.

c. name: Name of the file.

The “Create” function in filesys.h does the creation of file. It is done as follows:

a. It first checks whether there is already a file of this name created earlier using “FetchFrom” function. If already created it takes care not to recreate it again.

b. Next it finds a sector which is free using "Find" function and reserves that sector for a particular file using "Allocate" function and also adds necessary details in the above explained structure using "Add" function.

c. Thus if every step goes well and no exceptions are thrown, it returns success which means file is created successfully.

2. Writing to a File:

Once the file is created, data needs to be written on it. To do this we need to first open the file. This is done through "Open" function of filesys.c . It is done as follows:

a. It first finds the name of the file entered from the main directory file using "FetchFrom" and "Find" function.

b. If the name is found, it returns the sector number on which it is stored.

c. Using "OpenFile" function of openfile.cc it initializes a new header to point to that file and hence the file is opened for reading and writing.

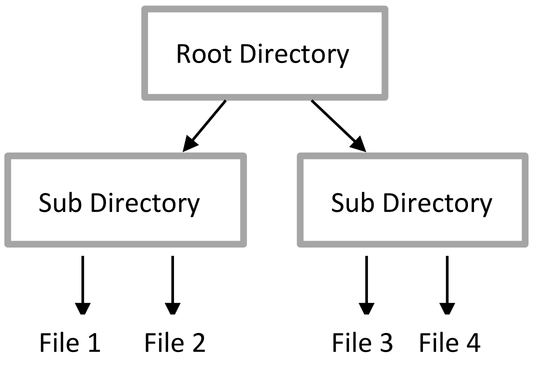
d. Finally the data is written in the opened file using "write" command. The contents for a while are not taken from the user but are already specified in filesys.cc file.

3. Limitation on File Name:

This is done at run time. The filename is written by the user as a command line argument. Before performing any operations on that filename we check whether entered name is within the specified length or not. If not then we terminate the program and display an error message.

4. Hard Code Implementation of Hierarchical Structure of File System:

Since the nachos has a flat file structure, we have tried making it hierarchical though it may not have any user inputs. The nachos allows only single directory to be opened and store the file contents. We have created 2 other subdirectories inside the main root directory. Treating the subdirectories as file, their names and sector numbers have been copied into the root directory. Now when a file is to created it can be created under either of 2 subdirectories but not in the root directory. Implementation looks something like this:



The subdirectory 1 and 2 are allocated fixed sectors 2 and 3 respectively. The information about the files further created will be stored in these 2 directories. Whenever we want to access these files we need to move to the corresponding subdirectory and then read write functions can be applied on it. This is how we have tried implementing a hierarchical file structure.

5.6) Limitations of file system in NACH-OS

1. The nachos file system can open only 1 directory at a time. Hence to move between directories we need to switch to and fro between directories to work on files. This is done using “synchdisk” function. Hence the implementation of hierarchical file system became little difficult.
2. The file structure is flat in nature which creates a lot of problems including disk errors at the booting time.
3. It does not allow multiprocessing.
4. Files have a fixed size assigned while creating. The file cannot be bigger than 4kb. This is because the nachos file-system assumes some fixed memory of its own from the disk at the very beginning and hence there is only limited number of sectors available for file storage. Hence the file size and the number of files that can be stored both are limited.
5. The physical disk can handle only one operation at a time. This is due to the design of nachos. Since it does not allow multiprocessing and also single thread execution, we cannot work on 2 files simultaneously.
6. The size of the directory cannot expand.
7. If the Nachos exits in the middle of an operation than there is a possibility of the disk being corrupted.

5.7) Execution Steps

<u>For testing of file system</u>:

Open command prompt and do the following:
1. Go to “code” directory folder inside nachos. cd /nachos/code
2. Type “make” make
3. Change the directory to filesys. cd filesys
4. Commands to be tested:
   4 a. To format/initializing the file system run ./nachos –f
      This initializes the file system i.e. it allocates 1024 sectors from the total memory available on disk as Nachos file system.
   4 b. To create a file ./nachos –cr filename directoryName
      There are 2 directories created under a root directory. The valid Directory names are <u>subdir1.odt</u> and <u>subdir2.odt</u>. (For this command to execute you first need to run “./nachos –u” command.
   4 c. To write to a file ./nachos –wr filename

The contents to be written into a file are pre-mentioned in system.cc file. The contents would be written only if the file is created first. Else it will throw an exception.

4 d. To see the limitation on the size of filename run "./nachos –cr filename directoryName" with filename having size more than 10 eg: file2345.txt . It will show an error message and will not allow you to create a file.

5.8) Failures

1. The hierarchical file structure can be extended to become user friendly i.e. more than 2 levels of hierarchy can be created and also creation of subdirectories be done by the user. This can be done by allowing more files to remain open and implementing an algorithm to move between directories easily.

Hint: One needs to allocate more sectors for sub-directory. Also there must be initialization of file header for each of the directory and all directories must be kept open till nachos runs. After all file operations all the files must be written back to disk so that changes are saved to disk.

2. Also size of each file created can be increased and also number of files that can be created can be increased. Since the header is stored in 1 disk sector, maximum size of a file is limited by the pointers that will fit in 1 disk sector. The increase in number of files to be created will happen automatically as the hierarchical file structure gets implemented as now you will have more directories to store filenames and hence more headers to locate files and its related information.

Hint: This can be done by implementing doubly indirect blocks.

6) Network Drivers

6.1) Overview

Although distributed systems have become increasingly important commercially, most instructional operating systems do not have a networking component. To address this, the main aim of the project is to write a significant and interesting distributed application.

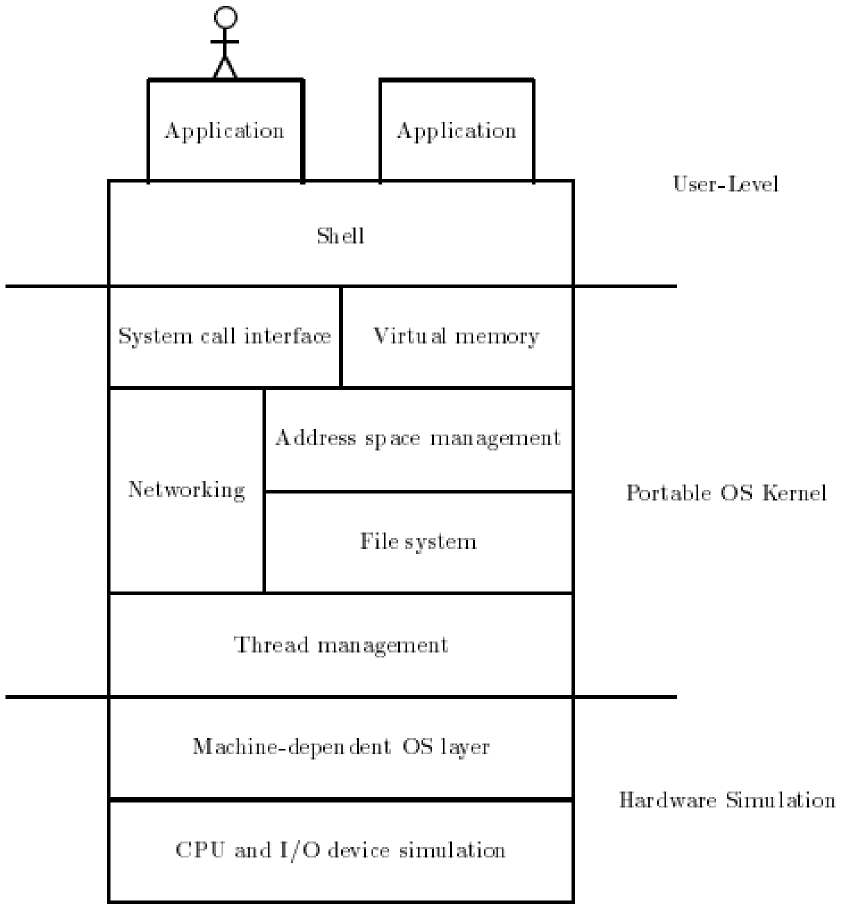


Figure 1: Nachos Software Structure

At the hardware level, each UNIX process running Nachos represents a uniprocessor workstation. We simulate the behavior of a network of workstations by running multiple copies of Nachos, each in its own UNIX process, and by using UNIX sockets to pass network packets from one Nachos "machine" to another. The Nachos operating system can communicate with other systems by sending packets into the simulated network; the transmission is actually accomplished by socket send and receives. The Nachos network provides unreliable transmission of limited-size packets from machine to machine. The likelihood that any packet will be dropped can be set as a command-line option, as can the seed used to determine which packets are “randomly” chosen to be dropped. Packets are dropped but never corrupted, thus checksums are not required.

Nachos is using a simple **POST OFFICE PROTOCOL** on top of the network to illustrate the benefits of layering. The post office layer provides a set of **MAILBOXES** that serve to route incoming packets to the appropriate waiting thread. Messages sent through the post office also contain a return address to be used for acknowledgements. The first task is to provide reliable transmission of arbitrary-sized packets, and then to build a distributed application on top of that service. To use arbitrary-sized packets, one needs to split any large packet into fixed-size pieces, add fragment serial numbers, and send them one by one.

Reliability is more interesting, requiring a careful analysis and design to be implemented correctly. The reliability of the network is a number between 0 and 1. It tells the chances that the network will lose a packet.

Perhaps the biggest limitation of our current implementation is that we do not model network performance correctly, because we do not keep the timers on each of the Nachos machines synchronized with one another.

6.2) Post Office Protocol (POP)

Post office protocol is an application layer Internet standard protocol used by local e-mail clients to retrieve e-mail from a remote server over TCP/IP connection.

The purpose of POP is to basically retrieve mail from Internet service provider (ISP). If somebody sends an email, it usually cannot be delivered directly to the receiver’s computer. ISP's mail server receives email from the internet; it will look at each message and if it finds one addressed to the receiver, that message will be filed to a folder reserved for receiver’s mail. This folder is where the message is kept until receiver retrieves it.

6.3) Networking in NACH-OS

Phase 1: Study of all the code associated with nachos networking.

1) post.h - Nachos consists of data structures for providing the abstraction of unreliable, ordered, fixed-size message delivery to mailboxes on other (directly connected) machines. Messages can be dropped by the network, but they are never corrupted. Our post office delivers packets to a specific buffer (Mailbox), based on the mailbox number stored in the packet header. Mail waits in the box until a thread asks for it; if the mailbox is empty, threads can wait for mail to arrive in it. Thus, the service our post office provides is to de-multiplex incoming packets, delivering them to the appropriate thread. With each message, you get a return address, which consists of a "from address", which is the id of the machine that sent the message, and a "from box", which is the number of a mailbox on the sending machine to which you can send an acknowledgement, if your protocol requires this.

   The mailbox address uniquely identifies a mailbox on a given machine. A mailbox is just a place for temporary storage for messages. The mailbox header defines the destination mail box, sender's mail box and the size of message data. Then a mail is initialized by concatenating the headers to the data, by defining the format of incoming/outgoing message.

   The message is layered in form of:
   - network header
   - post office header
   - data

   After this, a temporary mailbox id defined in which messages are stored only till the receiver retrieves the message. Finally a post office is created which will have collection of mail boxes. The post office is in synchronization with two main operations:

   - Send - send a message to a mailbox on a remote machine
   - Receive - wait until a message is in the mailbox, then remove and return it

   Incoming messages are put by the Post Office into the appropriate mailbox, waking up any threads waiting on Receive.

2) post.cc – It consists of routines to deliver incoming network messages to the correct address, i.e. a mailbox for incoming messages.

   The implementation synchronizes incoming messages with the threads waiting for those messages, which involves-
   a) Initializing a single mail message
   b) Initializing a mailbox within post office, so that it can receive incoming messages.
   c) De-allocate the mail box within the post office.

d) Print the message header.

3) nettest.cc - After sending the message, we need to test out message delivery between two "Nachos" machines, using the Post Office to coordinate delivery. For that on two different terminals, we run two different copies of Nachos with machine ID's 0 and 1 (these are hardcoded).

- ./nachos -m 0 -o 1 &
- ./nachos -m 1 -o 0 &

Address allocation:
To: destination machine, mailbox 0
From: our machine, reply to: mailbox 1

We need to test out message delivery, by doing the following:

1. Send a message to the machine with destination ID, at mail box #0
    a. The Post office constructs packet header and mail header
    b. Adds data to the packet
    c. Sends the packet to proposed destination.
2. Wait for the other machine's message to arrive (in our mailbox #0)
3. Send an acknowledgment for the other machine's message
4. Wait for an acknowledgement from the other machine to our original message
5. Halt

This is how two machines in nachos communicate through POP in application layer.

Phase 2: Pre-requisites

We need an implementation of condition variables, which is not provided as part of the baseline threads implementation. The Post Office won't work without a correct implementation of condition variables.

For the correct implementation of condition variables, some changes should be done in the synch.cc code in the threads of nachos.

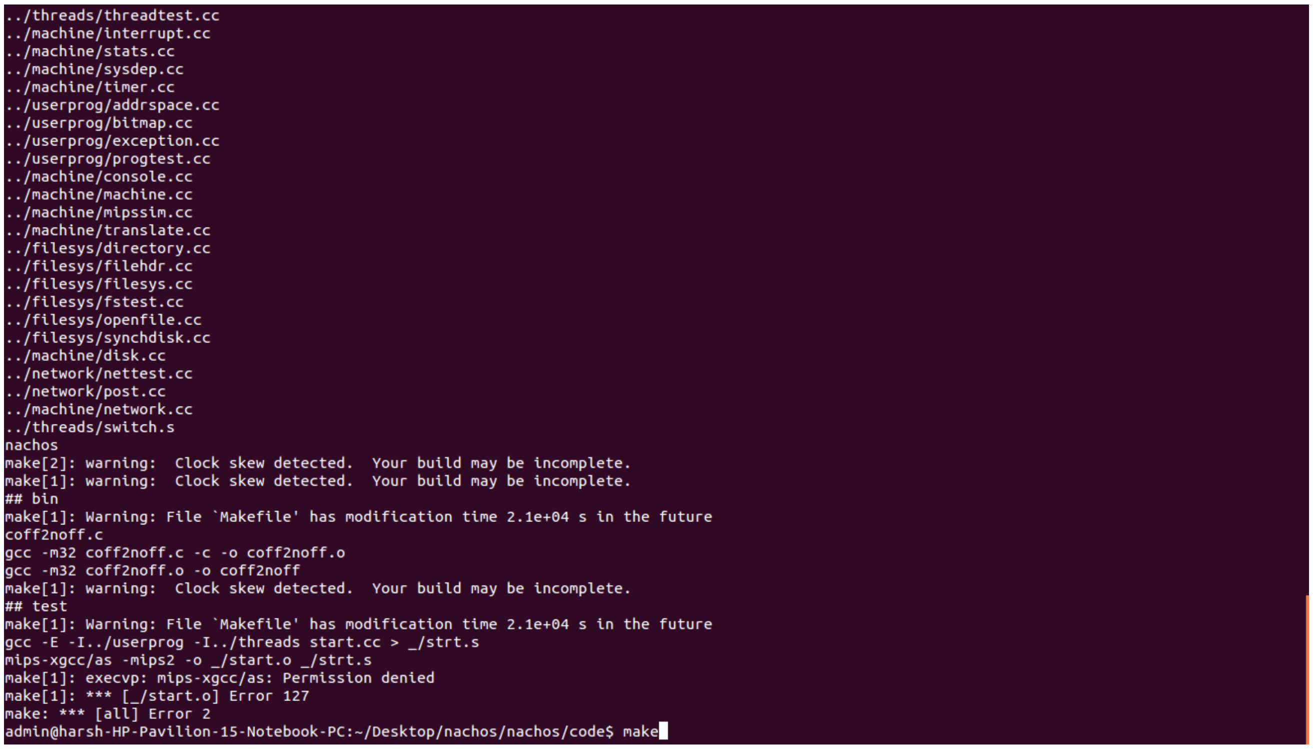
Phase 3: Implementation and testing

There are 3 command line arguments for networking:

- -n sets the network reliability
- -m sets the machine’s host id
- -o takes other machine’s id to test a simple network establishment between two machines

6.4) Execution Steps

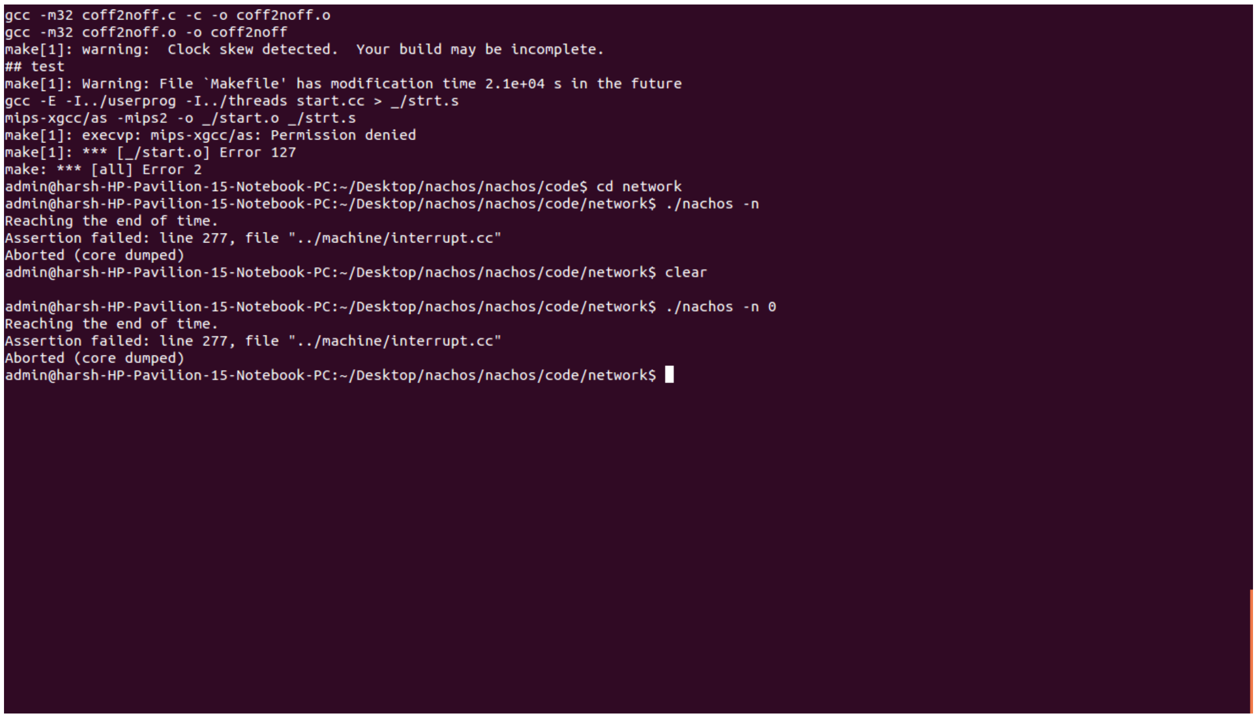
1) First of all, we 'make' the code of nachos which was successful with no errors.



```
../threads/threadtest.cc
../machine/interrupt.cc
../machine/stats.cc
../machine/sysdep.cc
../machine/timer.cc
../userprog/addrspace.cc
../userprog/bitmap.cc
../userprog/exception.cc
../userprog/progtest.cc
../machine/console.cc
../machine/machine.cc
../machine/mipssim.cc
../machine/translate.cc
../filesys/directory.cc
../filesys/filehdr.cc
../filesys/filesys.cc
../filesys/fstest.cc
../filesys/openfile.cc
../filesys/synchdisk.cc
../machine/disk.cc
../network/nettest.cc
../network/post.cc
../machine/network.cc
../threads/switch.s
nachos
make[2]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
## bin
make[1]: Warning: File `Makefile' has modification time 2.1e+04 s in the future
coff2noff.c
gcc -m32 coff2noff.c -c -o coff2noff.o
gcc -m32 coff2noff.o -o coff2noff
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
## test
make[1]: Warning: File `Makefile' has modification time 2.1e+04 s in the future
gcc -E -I../userprog -I../threads start.cc > _/strt.s
mips-xgcc/as -mips2 -o _/start.o _/strt.s
make[1]: execvp: mips-xgcc/as: Permission denied
make[1]: *** [_/start.o] Error 127
make: *** [all] Error 2
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code$ make
```

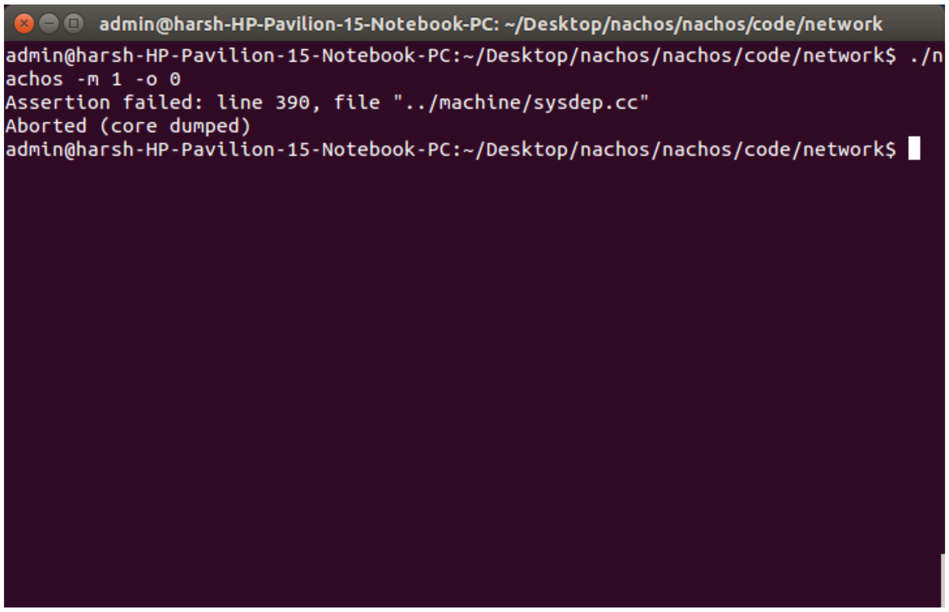
2) Then, in the network folder, as per the documentation we tried out different command-line arguments, but each resulted in segmentation fault, core dumped error.

   a) ./nachos –n 0



```
gcc -m32 coff2noff.c -c -o coff2noff.o
gcc -m32 coff2noff.o -o coff2noff
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
## test
make[1]: Warning: File `Makefile' has modification time 2.1e+04 s in the future
gcc -E -I../userprog -I../threads start.cc > _/strt.s
mips-xgcc/as -mips2 -o _/start.o _/strt.s
make[1]: execvp: mips-xgcc/as: Permission denied
make[1]: *** [_/start.o] Error 127
make: *** [all] Error 2
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code$ cd network
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code/network$ ./nachos -n
Reaching the end of time.
Assertion failed: line 277, file "../machine/interrupt.cc"
Aborted (core dumped)
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code/network$ clear

admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code/network$ ./nachos -n 0
Reaching the end of time.
Assertion failed: line 277, file "../machine/interrupt.cc"
Aborted (core dumped)
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code/network$ 
```

b) On different terminals,
   ./nachos –m 0 –o 1 and
   ./nachos –m 1 –o 0



```
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code/network$ ./n
achos -m 1 -o 0
Assertion failed: line 390, file "../machine/sysdep.cc"
Aborted (core dumped)
admin@harsh-HP-Pavilion-15-Notebook-PC:~/Desktop/nachos/nachos/code/network$ 
```

3) Then, we again read the documents carefully and with some research it was known that networking requires correct implementation of locks and condition variables.

This requires changes in synch.cc in threads folder, which was done and then again we tried to run the network. But, it still showed some errors.

6.5) Failures

- We tried to understand the full implementation of networking with POP.
- We thoroughly traced the movement of packets in communication with two different systems.
- We could not produce a working representation of networking module of Nachos.
    - PROBLEM FACED: We could not satisfy the values of condition variables and locks for POP implementation.

7) Bibliography

- http://phoenix.goucher.edu/~kelliher/cs42/oct07.html
- https://www.cs.duke.edu/~narten/110/nachos/main/node15.html
- http://www.cs.odu.edu/~cs471/soumya/synch.html
- http://people.csail.mit.edu/rinard/osnotes/h3.html
- http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html
- http://homes.cs.washington.edu/~tom/nachos/
- http://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm
- https://www.prismnet.com/~mcmahon/Unix_Tutorial/file_structure.html
- http://www.tutorialspoint.com/unix/unix-file-system.htm
- http://www.ida.liu.se/~TDDB63/material/begguide/filesystem.html
- http://www.slideshare.net/jobo.zh/nachos-filesystem-presentation-823345
- http://www.cas.mcmaster.ca/~qiao/courses/cs3mh3/roadmaps/3.4/filesys.html
- http://www.cs.rit.edu/~mpv/course/os2/FileSystems.html
-