**CIS 657 Programming Assignment 3**

This assignment is a group project. You can work by yourself.

Each group needs to submit Final Report and Nachos source code:

* Create your document – Final Report (doc or pdf) including the **disclosure form**, **screenshot(s) of your output**, **your solution code *– Must be text (image cannot be process with TurnItIn) --* (filename including full path, and your own code (created or modified by you), not Nachos original code)** to the **Turnitin link** on the blackboard (under Assignments).
  + Your design must be clearly explained.
  + Your solution must be clearly described.
  + All of your code must be well documented and commented.
  + Make sure that you receive the confirmation email from the TurnItIn.
* Do a “make clean” in code/build.linux, then compress your Nachos folder on the VM, download the compressed file into your local machine and name the compressed file **Ass3\_yourNetId.zip (Note: Zip only!!)**. Submit the zipped file to the **Assignment Code Submission link** of Blackboard.
  + Make sure that your Nachos runs on the VM and your zip file is uncompressed.

Group size cannot be > 3

Each individual of the team needs to submit project evaluation form

* What the student did/contributed
* What other team mates did/contributed
* This is confidential

Due: Dec. 7 (Friday, end of the day)

Late submission: -10 points by Dec. 8, -20 points by Dec. 9. No later submission is allowed. (This is the final project. So please understand there will be no extensions. This is a hard deadline. You must submit everything by the deadline.)

*Follow the Lab1 instruction and create a new fresh Nachos folder. If you need multitasking or virtual memory, use your assignment 2 implementation.*

1. Distributed File System (DFS)

* Each separate Nachos (emulated as a UNIX process) is a node in the network; a network (emulated via UNIX sockets) provides the communication medium between these nodes (preferably, on different terminals or in different windows).
* When a user process opens a file with a name of the form "node #/string", this refers to a file stored on the server whose network address is the specified integer. For example, you can assume that there are always at most 10 nodes, numbered 0 - 9, so the "node #" here is a single digit.
* A filename that does not begin with "node #/" refers to a file stored on the local node. Thus, for a process running on the node with network address addr (node #), "string" and "addr/string" refer to the same file.
* A server with network address addr (node #) stores all of its files in a UNIX directory named "addr". For example, the node with network address 0 would store a file named "testfile" in a UNIX file named "0/testfile". This implies that if (e.g.) node 0 opens an executable "test-pgm", that executable must be accessible as ./0/test-pgm.
* Your DFS must be designed to work correctly with a completely asynchronous network.
* **The basic Nachos FileSystem** must be implemented.
* You need to present sufficient test cases to show the correctness of your File System.

2. Fast File System (Improve File System Performance)

* No user programs are required for this project.
* Implement the dividing of a file into blocks and blocks and fragments.
* **The basic Nachos FileSystem** must be implemented.
* The DISK must be logically divided into cylinder groups.
* File growth algorithm must match the description in the paper.
* A kernel process that writes to a file that demonstrates the growth algorithm should be implemented.
* Each cylinder group should keep track of its local information.
* Attempt to store all i-nodes for directory in the same cylinder group.
* You need to present sufficient test cases to show the correctness of your File System.

3. Log Structured File System

* No user programs are required for this project.
* Disk space in the file system will be divided into segments.
* Log structure will be implemented that buffers writes together into a segment before being written out.
* Segment summary will describe the contents of a segment.
* You must be able to read these files back from the log structure.
* Inode map to maintain location of all inodes, this will be stored in log structure with the updated versions before write.
* Segment cleaner function to claim space from already written segments.
* You must implement **the basic Nachos FileSystem**.
* You must have kernel program that demonstrates that your project works correctly.
* You need to present sufficient test cases to show the correctness of your File System.

**Basic Nachos FilesSystem**

The files to focus on are:

* filesys.h, filesys.cc — top-level interface to the file system.
* directory.h, directory.cc — translates file names to disk file headers; the directory data structure is stored as a file.
* filehdr.h, filehdr.cc — manages the data structure representing the layout of a file’s data on disk.
* openfile.h, openfile.cc — translates file reads and writes to disk sector reads and writes.
* synchdisk.h, synchdisk.cc — provides synchronous access to the asynchronous physical disk, so that threads block until their requests have completed.
* disk.h, disk.cc — emulates a physical disk, by sending requests to read and write disk blocks to a UNIX file and then generating an interrupt after some period of time. The details of how to make read and write requests varies tremendously from disk device to disk device; in practice, you would want to hide these details behind something like the abstraction provided by this module.

Nachos file system has a UNIX-like interface, so you may also wish to read the UNIX man pages for creat, open, close, read, write, lseek, and unlink (e.g., type “man creat”). Nachos file system has calls that are similar (but not identical) to these calls; the file system translates these calls into physical disk operations. Create (like UNIX creat), Open (open), and Remove (unlink) are defined on the FileSystem object, since they involve manipulating file names and directories. FileSystem::Open returns a pointer to an OpenFile object, which is used for direct file operations such as Seek (lseek), Read (read), Write (write). An open file is “closed” by deleting the OpenFile object.

Many of the data structures in our file system are stored both in memory and on disk. To provide some uniformity, all these data structures have a “FetchFrom” procedure that reads the data off disk and into memory, and a “WriteBack” procedure that stores the data back to disk. Note that the in memory and on disk representations do not have to be identical.

* Complete the basic file system by adding synchronization to allow multiple threads to use file system concurrently. Currently, the file system code assumes it is accessed by a single thread at a time. In addition to ensuring that internal data structures are not corrupted, your file system must observe the following constraints (these are the same as in UNIX):
  + The same file may be read/written by more than one thread concurrently. Each thread separately opens the file, giving it its own private seek position within the file. Thus, two threads can both sequentially read through the same file without interfering with one another.
  + All file system operations must be atomic and serializable. For example, if one thread is in the middle of a file write, a thread concurrently reading the file will see either all of the change or none of it. Further, if the OpenFile::Write operation finishes before the call to OpenFile::Read is started, the Read must reflect the modified version of the file.
  + When a file is deleted, threads with the file already open may continue to read and write the file until they close the file. Deleting a file (FileSystem::Remove) must prevent further opens on that file, but the disk blocks for the file cannot be reclaimed until the file has been closed by all threads that currently have the file open.
  + Hint: to do this part, you will probably find you need to maintain a table of open files.
* Modify the file system to allow the maximum size of a file to be as large as the disk (128Kbytes). In the basic file system, each file is limited to a file size of just under 4Kbytes. Each file has a header (class FileHeader) that is a table of direct pointers to the disk blocks for that file. Since the header is stored in one disk sector, the maximum size of a file is limited by the number of pointers that will fit in one disk sector. Increasing the limit to 128KBytes will probably but not necessarily require you to implement doubly indirect blocks.
* Implement dynamically extensible files. In the basic file system, the file size is specified when the file is created. One advantage of this is that the FileHeader data structure, once created, never changes. In UNIX and most other file systems, a file is initially created with size 0 and is then expanded every time a write is made off the end of the file. Modify the file system to allow this; as one test case, allow the directory file to expand beyond its current limit of ten files. In doing this part, be careful that concurrent accesses to the file header remain properly synchronized.

4. Google File System

* Creation of ChunkServer, Master Server and client routines.
* No C-user programs are required for this project.
* Replication and management of chunks
* Handling record appends
* Add or remove data node any time. This would be configurable.
* Master machine will listen for Heart Beats sent after configurable interval to the master machine.
  + If the master machine does not hear any HB from any slave machine for few HB counts(configurable), will mark it as dead and replicate its data to another machine to have consistency of replication count.
* You need to present sufficient test cases to show the correctness of your implementation.

5. Nucleus Message Passing

Implement four system calls as described in the papers, to be called by user programs (i.e. C

programs):

1. SendMessage: An asynchronous system call

2. WaitMessage: A synchronous system call

3. SendAnswer: An asynchronous system call

4. WaitAnswer: A synchronous system call

*REFERENCE:*

1. Send Message (receiver, message, buffer)

- copy message into the first available buffer within the pool and delivers it in the queue of named receiver

- the receiver is activated if waiting on a message

- Sender continues after being informed of the “identity of the message buffer” (why this is important, because later on, answer if any will be sent back using that message buffer ==> connection set up)

1. Wait Message(sender, message, buffer)

- delay requesting process until a message arrives in the process's queue

- on return, i.e., when process is ready to proceed), it is supplied:

+ name of sender

+ contents of message

+ identity of the message buffer (why need this? ok, later on, we will see some kind of piggy back, the same message buffer will be used to store the answer. So this identity is some kind of connection between sender and receiver)

+ the buffer is removed from the queue, and made ready to transmit answer

1. Send Answer(result, answer, buffer)

- copy an answer in which message has been received (the identity in charge...)

- deliver it to queue of original sender (there is no sender name in the call, how do i know what is original sender? again by the identity of message buffer given when call WAIT MESSAGE)

- the answering process continues immediately

- the sender of the message is activated if it is waiting for the answer

1. Wait Answer(result, answer, buffer)

- delays the requesting process until answer arrives in a given buffer (hence, send message need to know the identity of message buffer where the message was sent, so that it know where to wait on)

- On arrival, the answer is copied into the process, and the buffer is returned to the pool.

- Result: specify whether the answer is:

+ response from another process or

+ dummy answer generated by system nucleus in case of non-existing receiver

* Implement the buffers correct in the Kernel.
* Processes should be able to communicate with any arbitrary process running in the system, but you need to work out some way of identifying them (at compilation time, so we can’t use the PID).
* The same message buffer should be used for the same two processes to communicate.
* If a process terminates (and you are required to show an example of this happening). This can be done by invoking the Exit system call mid-conversation. The system should send a dummy message out to the waiting process.
* If a process dies while it has sent messages, then those messages should still be receivable by their co-communicators.
* A limit on the total number of messages sent by the process should be configurable and enforceable.
* No other process should be able to interfere with communication (save ids).
* You need to present sufficient test cases to show the correctness of your implementation.

6. Mach Message Passing

* Implement the MachOS Port class in the Nachos Kernel.
* User programs should attempt to communicate via ports.
* Implement two system calls (these can take port Ids) rather than port objects.
  + Send
  + Receive
* Implement the Network Server, this does NOT have to run in user application task, so it may be implemented in Nachos as a kernel process.
* Network servers should be able to communicate with other Nachos processes.
* Communication via ports should be many to 1 (i.e. should only be 1 receiver to a port).
* Bi-directional communication requires two ports.
* Messages in this case should be strings.
* Threading is not required for this project.
* Asynchronous messaging can be done in lieu of the network server. In this case, you are required to build out signals and associated system calls in Nachos (but only for this 1 type of signals).
* Demonstrate your message passing primitives work by using them to implement a “network chat” application among multiple users.
* You need to present sufficient test cases to show the correctness of your implementation.