EECE 315 – UBC

Assignment 5

Overview

This assignment focuses on how to implement a File System in an Operating System. The basic nachos operating system uses stubs to Unix system calls to provide for normal file support. In this assignment, these calls to Unix system calls are no longer used - the File System is handled internally to nachos.

The current implementation of the Nachos file system is fairly basic. A single thread, flat directory structure (i.e. no sub-directories), with limited number of files, fixed file size and limited max file size are the current abilities of the system. In the course of this assignment, it is your responsibility to remove some of these restrictions.

The Nachos version provided with this assignment is no longer using its stub file system. You will also notice that the behavior of the system will change. For example, you will no longer be able to simply run:

./nachos -x ../test/array

Note: All Nachos commands are executed from the **userprog** directory. If you execute them in any other directory, results might be different.

Why? Because Nachos is now looking for the array executable file in the Nachos file system. The array file is not there; it is in the UNIX file system. To run the array program, or any other program, you will first need to load the program into the Nachos file system. Then you will be able to run it. For example, you might run:

./nachos -f -cp ../test/array array -x array

This command will format the Nachos disk and initialize an empty file system on it (the -f flag), copy the array program from the UNIX file system into the Nachos file system (the -cp flag), and then execute the array program from the Nachos file system. The -cp flag, of course, is somewhat unrealistic since it allows you to load files into the Nachos file system from outside. However, since you create Nachos NOFF files on UNIX machines, such a facility is necessary if you are to run those files on Nachos.

Note that you must format the Nachos disk before you can store any files on it for the first time. Failure to do so will result in errors. Formatting the disk erases anything previously stored on the disk and creates a new, empty file system.

Files related to this task can be located in the ***filesys*** directory. We basically focus on creating the files, so we need not execute the code. Sample test files can be located in the *filesys/test* directory. We will use these files to test your code.

Command Name	*Command*
TC1	*./nachos –d f -f -cp ../filesys/test/small small*
TC2	*./nachos –d f -f -cp ../filesys/test/medium medium*
TC3	*./nachos –d f -f -cp ../filesys/test/big big*

-d: This flag is used to turn debugging on for file system (-f) debug statements.

-cp: This flag is used to copy files from UNIX to Nachos file system.

If you execute all the above commands in the *filesys* directory, TC1 and TC2 will execute successfully, whereas TC3 will lead to segmentation fault. This happens because size of test file used in TC3 is bigger than what Nachos can support. Please follow up the debug statements in the console to see what file was created and what data was copied to these files.

Task 1: Implementing variable file size

Default Configuration

The default implementation of the file system requires that when a file is created, a file size be given. This file size is used to allocate an appropriate number of disk sectors for this file. Once this file size is set, it is unchangeable. Details of how files are currently created can be seen in *filesys.cc* in the *FileSystem::Create*method. The effects of the set file size in can be seen *inopenfile.cc* in the *OpenFile::WriteAt* method. In this function there is a check to see if the write position plus the number of bytes to write goes beyond the end of the file:

```
if ((position + numBytes) > fileLength)

        numBytes = fileLength - position;
```

If the write would go past the end of the file, it adjusts the number of bytes to write to stop at the end of the file.

Changes to Make

The first thing to changes it to make the size parameter optional. Changing the following line of code can do this.

*bool FileSystem::Create(char *name, int initialSize) {*

To

*bool FileSystem::Create(char *name, int initialSize = 0) {*
initialSize = 0;

This sets the default value of initialSize to 0 bytes, so that a call to this function does not need to specify the size, and makes it so that files start off with 0 bytes, as they do in Unix.

Just make this change and compile the code. When you execute TC1 and TC2, you will notice that the file size was created with size 0, but there was fewer data copied to the file. The copied data was just metadata about the file, stored in file header. You will learn about file header in next task. The actual data was trimmed due to the default configuration mentioned above.

Now you need to work and modify the Nachos code to increment the file size dynamically.The next step should consist of modifying *WriteAt* in the *openfile.cc*file to allow for a change in the number of bytes. This will include a simple change in numBytes in the FileHeader, if the change in size does not cross a sector boundary. If the size change crosses a sector boundary, you must allocate a new sector, increase the size of numBytes and numSectors in the FileHeader. This should probably be done in a method that you will add to the FileHeader class so that you will have the access you need to the numBytes and numSectors FileHeader class variables.

Testing:

The output of TC1 and TC2 should first create a file of size 0, and then modify the size whenever more data needs to be added to the file.

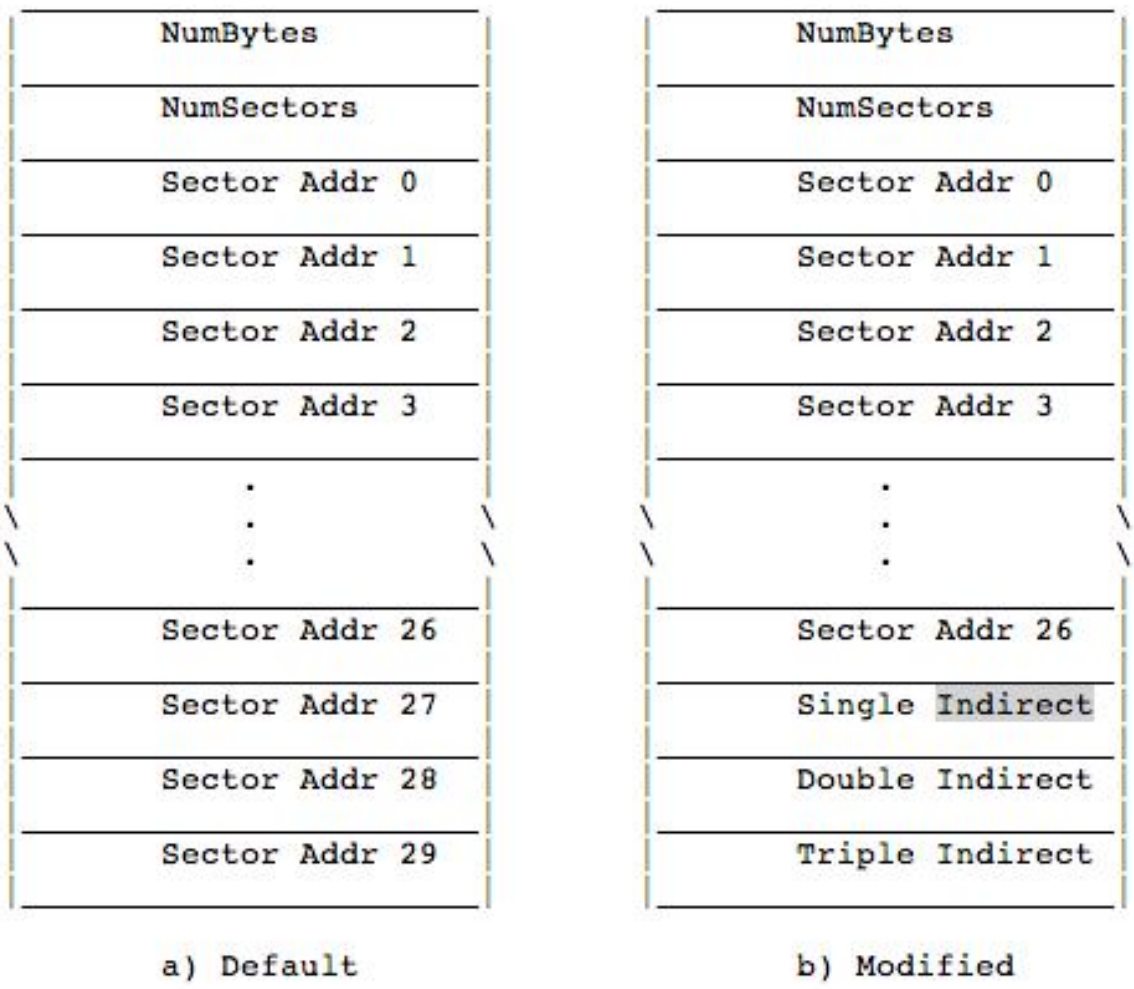
Task 2: Maximum supported file size

Default Configuration

The maximum file size for the default nachos configuration is 3,840 bytes. This is based on the sector size in two different ways. First off, the file's FileHeader itself must fit in a sector, so can be at most 128 bytes, or 32 ints. Two ints are required to specify numBytes and numSectors, leaving 30 ints for the sector numbers used to hold sector numbers (referred to as direct addressing). So each file can use at most 30 sectors, each sector being 128 bytes. 30 Sectors * 128

Bytes/Sector = 3840 Bytes. As the disk size is 128 KBytes, this is the maximum file that you can support. For this assignment you only need to support 6KB.

As it is not allowable to increase the number of sectors used for the file's FileHeader, it is necessary to increase the maximum file size by some means of indirect addressing of sectors. Removing some of the ints that were used to address data sectors and use them to address sectors containing indirect addresses to data sectors can do this. The following is an ASCII picture of what is happening



The Single Indirect Address will be the sector number of a sector containing a table that contains the address/sctor numbers of thirty-two data sectors. The Double indirect is the sector number of a sector containing a table that contains 32 single indirect addresses/sector numbers. Finally, the triple indirect address is the sector number of a table containing 32 double indirect addresses/sector numbers. As mentioned previously, 6 KBytes is the maximum file size that you will need to support. This can probably be done with single and double indirect addresses.

Note: For the given test cases, you only need to take care of single indirect addresses and leave the rest.

Easy Changes

The changes those in the definition of the basic filehdr class and the size of its components are fairly minimal. Currently, the size of the dataSectors array in the FileHeader is based on the definition of NumDirect:

*#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))*

As you can see, it is equal to the SectorSize (=128 Bytes) divided by the size of an int (4 bytes) to get the total ints in a sector minus the size of 2 ints for the numBytes and numSectors mentioned previously. This does not leave room for the indirect addresses. To take into account the use of three ints for the indirect, double indirect and triple indirect addresses, it is necessary to subtract off three more ints from the allowable size of the direct addresses specified by NumDirect.It is also necessary to modify the FileHeader class itself to include the indirect addresses.

Harder Changes

It is also necessary to change at least one main function in *filehdr.cc*, the function previously written to allow for files to increase in size - you must provide for when the new sectors need to be addressed using the single of double indirect addresses.

Testing:

The output of TC1, TC2 and TC3 should first create a file of size 0, and then modify the size whenever more data needs to be added to the file.

Please make sure that for TC3 no data is cut out while copying. You should be able to see the bytes of copied data in the printed debug statements.

Bonus Task: Multi-Threaded

Default Configuration

In the default configuration of the File System assignment, there is no threading. This can be seen in the *threads/main.cc* file in the portion in the main block after #ifdef FILESYS. It can be seen that for every command line argument given, a call is made to an appropriate function, without the use of a thread. This means

that once a file system action has started, another will not start until the first is finished. Because of this, there is no protection provided to prevent multiple threads from accessing the same file concurrently - in the case of writes, this can be a major problem.

Command Line Calls

It is important to understand how multiple threads might be run in the file system assignment. When nachos is run, it is called with command line arguments indicating which filesystem functions to run.

nachos -f -l -cp ../test/small small -l -r small -l

This nachos command will format a new nachos disk, list the contents (empty), copy in *small* file, list the contents again (1 file), remove the file *small* and list again (empty). If this is done without threading, these will be done one at a time with a single thread. If done with threads, this will be done with 6 threads running concurrently. It is worth noting, that it is possible that this will have problems when using threads, if the -r argument begins executing before the -cp is finished.

Changes

In order to allow multithread, it is necessary to create threads in *main.cc* that will be forked to the appropriate file system function. This will require looking back at how things were done in previous assignments. The basic idea here is:

create new thread
create new parameter structure
set parameters
newThread->Fork(appropriateFunction, (int)parameters)

In order to prevent multiple threads to access the same file concurrently (for writes or writes during reads) it is probably necessary to construct *a global file table* that tracks all open files. This table will include some sort of file identifier, an int tracking the number of threads accessing the file, a delete flag to indicate when a file has been deleted, read and write access control in the form of semaphores and or condition variables and whatever other global file administrative items you need.

For the concurrency of file reads and writes, the basic problem is that you cannot have a read and a write or a write and a write happening at the same time.

1. So the obvious thing to do is not let any writes occur until there are no files reading - the problem with this is that you may end up starving the thread waiting to write if reads keep occuring.
2. To handle this, you need to allow multiple reads to occur until a write comes along. All reads are allowed to complete, but no new reads are allowed until the current write finishes.

3. If there are multiple reads and writes that have come in while waiting for the first write to finish, they should be handled in the order that they came in. As mentioned previously, the easiest way to handle this is probably with condition variables and semaphores.

Testing:

Explain your code and how do you solve the problems mentioned above. You can use the string files in "/filesys/test" and executable files in "/test" to show the result.

Evaluations

Task 1	
Correct output	50%
Task 2	
Correct output	50%
Bonus task	
Multithreaded command line options	15%
Read & write control (synch problem)	15%

Submission Guidelines

You need to submit all thefiles that you will be modifying for this assignment:

Create and submit the zip file as an attachment to the eece315term2@gmail.com with the subject as follows:
"Assignment5: <Group No.>"

Once you submit the assignment, you will receive an automated reply from the system. You need to show that reply to the TA before they could assess your assignment in the lab, to make sure that you have submitted your assignment correctly.