Nachos Assignment #4: Build a file system

Tim Brecht
Computer Science 4321/5421
Due date: Tuesday – December 2, 1997, 2:30 p.m.

The multiprogramming and virtual memory assignments made use of the Nachos file system. The fourth phase of Nachos is to actually build this file system. As in the first two assignments, we give you some of the code you need; your job is to complete the file system and enhance it.

The first step is to read and understand the partial file system we have written for you. Run the program 'nachos -f -cp test/small small' for a simple test case of our code – '-f' formats the emulated physical disk, and '-cp' copies the UNIX file 'test/small' onto that disk.

The files to focus on are:

fstest.cc — a simple test case for our file system.

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.

Our 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"). Our file system has calls that are similar (but *not* identical) to these; the file system translates these calls into physical disk operations. One major difference is that our file system is implemented in C++. 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.

While our code implements all the major pieces of a file system, it has some limitations. Your job will be to fix these limitations. Keep in mind that there are, of course, interactions between the various parts of this assignment. For instance, how you choose to implement large files may affect the performance of your system for part 4.

The assignment is to do items 1, 2, and 3. Note that part 2 does not necessarily have to be a separate step. That is, you may wish to go directly from part 1 to 3 as long as the files in part 3 can be as large as the disk (but be careful that you have something to show for your effort if you do not get all of part 3 implemented). A discussion of how you would implement the suggestions in part 4 and 5 are good examples of potential exam questions.

1. 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 modified 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.

2. Modify the file system to allow the maximum size of a file to be as large as the disk (128 KBytes). In the basic file system, each file is limited to a file size of just under 4 KBytes. 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 128 KBytes will probably but not necessarily require you to implement doubly indirect blocks.

3. Implement 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. (Extra credit). Improve the performance of your file system.

   Here are a couple of approaches you might take to improving performance:

   Optimize disk seek and rotational latency. To read or write a sector, the disk head must be moved to the correct track and then the system must wait for the correct sector to rotate under the disk head. In disk-bound systems, these delays can be a cause of poor performance. This can be fixed by (i) laying disk blocks from the same file on the same track (cf. article by McKusick et al.) and (ii) when multiple requests are queued for the disk, scheduling first those that would have the shortest delay, while at the same time avoiding starvation of disk requests.

   Modify the file system to keep a cache of file blocks. When a request is made to read a block, check to see if it is stored in the cache, and if so, no disk access is needed since a copy can be returned immediately. One enhancement is that instead of always immediately writing modified data to disk, dirty blocks can be kept in the cache and written out sometime later — this is called "write-behind". Another enhancement is to automatically fetch the next block of a file into the cache, when one block of a file is read, in case that block is about to be read — this is called "read-ahead."

   You are limited to a cache that is 64 disk sectors in size (roughly 6% of the size of the disk); this space limit must include the memory you use for open files, the bitmap, etc., if you keep those in memory.

For each of these you implement, explain what you expect the performance improvement to be on the benchmarks, and why the other alternatives (the ones you did not implement) would yield fewer performance gains for the same effort.

5. Implement a hierarchical name space. In the basic file system, all files live in a single directory; modify this to allow directories to point to either files or other directories. To do this you will need routines to parse path names into the sequence of directories where to find the file. You will also need to implement routines to change the current working directory (cf. the UNIX 'cd' command) and to print the contents of the current directory.

   For performance, allow concurrent updates to different directories, but use mutual ensure that updates to the same directory directory are performed atomically (for example, to ensure that a file is deleted only once).