Nachos Assignment 3: File System

Sanzheng Qiao
Department of Computing and Software

October 22, 2007

Due: March 27, Tuesday, 11:59 pm.

As in the previous assignments, we give you some of the code you need; your job is to complete the file system and enhance it.

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 Nachos code implements all the major pieces of a file system, it has some limitations. Your job will be to fix these limitations.

The assignment is to do items 1, 2, and 3.

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 (128K bytes). In the basic file system, each file is limited to a file size of just under 4K bytes. 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 128K bytes 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.