CS350 **Operating Systems** **Fall 2006**

Assignment 3: File Systems

Due date: 12:00 (noon), Monday, December 4, 2006

1 Nachos File Systems

Nachos has two file system implementations. As provided to you, Nachos uses the stub file system implementation, which simply translates Nachos file system calls to UNIX file system calls. This is the file system implementation that you have been using for the first two assignments. Nachos also comes with a very basic file system implementation that uses the Nachos simulated disk. For Assignment 3, your task is to improve on this basic implementation.

Your first task will be to switch over from the stub file system implementation (which will no longer be used) to the basic file system that uses the Nachos disk. To use this file system, you will need to rebuild Nachos. First do a

```
make distclean
```

in your build directory. Then, edit `Makefile` so that the symbol `FILESYS_STUB` is no longer defined. Do this by changing the line

```
DEFINES = -DRDATA -DSIM_FIX -DFILESYS_STUB -DRR_QUANTUM
```

to

```
DEFINES = -DRDATA -DSIM_FIX -DRR_QUANTUM
```

Don't forget to add in `-DUSE_TLB` to this list if you wish to continue using the TLB as you did for Assignment 2. Once this is done, you should rebuild Nachos by running `make depend` followed by `make nachos` as usual.

The internal file system interface used by the new basic file system implementation is almost the same as the interface used by the stub file system, with a few differences. For example, `FileSystem::Create` takes one parameter in the stub file system interface, and two parameters in the new basic file system interface. You will need to study the header files in the `filesys` directory to identify the other differences in the interface. Because of these differences, you may need to make a few changes to your existing file-related system call (e.g., `Create`, `Open`) implementations to get Nachos to compile with the new file system.

Once Nachos 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/halt
```

Why? Because Nachos is now looking for the `halt` executable file in the Nachos file system. The `halt` file is not there; it is in the UNIX file system.

To run the `halt` 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/halt halt -x halt
```

This command will format the Nachos disk and initialize an empty file system on it (the `-f` flag), copy the `halt` program from the UNIX file system into the Nachos file system (the `-cp` flag), and then execute the `halt` 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.

You may want to load a number of programs or files into the Nachos file system over and over again (e.g.,when recompiling your test programs.) To do this, it is convenient to put the relevant Nachos commands into a script that can be executed with one command. Probably the simplest way is to put the commands into a text file called `reload` (one Nachos command per line) and run the script with the command "`sh reload`".

There are a number of other file system-related utilities you can run from the Nachos command line. For example, there are utilities that will allow you to display the contents of a Nachos file, to delete Nachos files, and to list the contents of the Nachos directory. See the file `threads/main.cc` for a complete list of the available Nachos command line parameters.

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.

2 File System Design Requirements

The specific requirements for this assignment are as follows:

1. Ensure that the file-related system calls `Create`, `Open`, `Close`, `Read` and `Write` work properly with the basic file system. These calls are already implemented. However, as was noted in Section 1, the new basic file system's interface is not quite the same as the interface used by the stub file system. As a result, you may have to make some small changes to make these system calls work.

   Note that it should still be possible to use the `Read` and `Write` system calls to perform console I/O.

2. Add synchronization to the file system to ensure that `Read` and `Write` operations on each file are atomic. That is, if processes attempt concurrent `Read` or `Write` requests on a file, your operating system should execute the requests one at a time. Similarly, your operating system should ensure that at most one system call (such as `Create` or `Remove` or `Open`) uses a directory at a time. Your synchronization mechanism should be general enough that system calls that use distinct files or directories can proceed concurrently. It should also be general enough so that more than one process can have a file open simultaneously.

3. All file system operations must be atomic. For example, if one process is in the middle of a file write, a process concurrently reading the file will see either all of the change or none of it. For this assignment, it is sufficient to implement locking at the file level.

4. Implement the `Remove(char *filename)` system call, which is used to delete files. When a file is removed, processes that have already opened that file should be able to continue to read and write the file until they close the file. However, new attempts to open the file after it has been removed should fail. Once a removed file is no longer open by any process, the file system should actually remove the file, reclaiming all of the disk space used by that file, including space used by its header.

5. Implement the `GetFileLen` system call, which returns the length, in bytes, of an opened file.

6. Implement the `Seek` system call. Each time a file is opened, NachOS returns an `OpenFileId` to the calling process. There should be a *separate* file (seek) position associated with each such `OpenFileId`. The `Read` and `Write` system calls modify this position implicitly, while the `Seek` system call lets a process explicitly change an open file's seek position so it can read or write any portion of the file. Note that negative offset values are permitted. This permits, for example, seeking to -5 bytes from the end of the file.

7. Modify the file system so that it will support files as large as 64 Kbytes. (In the basic file system, each file is limited to a file size of 3840 bytes.) You should have some ideas from class as to how to accomplish this.

8. Implement a mechanism to allow files to grow. A file should grow when a process tries to `Write` beyond the end of the file. In either case, the file should grow enough to accommodate the `Write` operation that causes the growth. Of course, a file should not be allowed to grow larger than the maximum file size supported by the system, or beyond the available capacity of the disk.

   Note that `Read` and `Seek` calls should not cause a file to grow. A `Read` beyond the current end of the file must return an end-of-file indication as described in `userprog/syscall.h`. A `Seek` beyond the current end of the file must return an error.

9. Implement named pipes. The system call `CreatePipe(char *name)` is used to create a named pipe. If a pipe with the specified name does not already exist, the kernel should create one. If a pipe with the specified name already exists, this call should succeed, but needn't do anything. The system call `RemovePipe(char *name)` is used to delete the specified pipe, destroying any data that may remain in the pipe. `RemovePipe`'s behaviour should be similar to `Remove`'s: if any processes have the pipe opened at the time that it is removed, those processes should be able to continue using the pipe until they close it. However, no further opens of the pipe should be allowed. Once all such processes have closed the pipe, the pipe and any data that remain in it can be deleted.

   Once created, named pipes are used by way of the existing `Open`, `Read`, `Write`, and `Close` system calls. The data written to a named pipe should be organized (conceptually) as a FIFO queue. A `Write` call appends data to the back of the queue. A `Read` call consumes data from the front of the queue. If a `Read` call requests k bytes of data from a pipe when the pipe contains fewer than k bytes of data, then the `Read` call should *block* until k bytes of data are available. Furthermore, each pipe should have a maximum capacity of 256 bytes. If a `Write` system call attempts to write k bytes of data into a pipe which does not have sufficient free capacity to hold that much data, then the `Write` system call should *block* until there is space available in the pipe for all k bytes. (Note, however, that an attempt to read or write more than 256 bytes of data from/to a pipe in a single system call should result in an immediate error, since waiting will never allow such a request to succeed.)

   Named pipes should be persistent, like files. Furthermore, pipe names should be treated like file names: when a named pipe called "foo" is created, the pipe's name should appear in the file system directory. Since both file names and pipe names appear in the same directory, they may conflict: at any time, your system may have a file named "foo" or a pipe named "foo", but not both. Furthermore, if a "foo" entry appears in the directory, your system must have some way of determining whether it refers to a file or to a pipe. Unlike regular files, however, pipes are not expandable beyond their fixed maximum size of 256 bytes. Furthermore, the `Seek` and `GetFileLen` system calls should not work with named pipes. An attempt to use either call on a named pipe should result in an error.

3 Getting Started

Your first step should be to switch from the stub file system to the basic Nachos file system implementation, as described in Section 1. Next, read and understand the basic file system implementation, as it will be your starting point. The files to focus on in the `filesys` directory 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. This is the Nachos equivalent of a UNIX i-node.

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.

Some of the data structures in the Nachos 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.

You may implement Assignment 3 directly on top of the base NachOS code that you can download from the course account. Alternatively, you may build on the code that you developed for Assignment 1 or Assignment 2. If you are building on NachOS from Assignment 2, it is important to remember that the NachOS simulated machine includes *two* simulated disks, `Kernel::swapDisk` and `Kernel::synchDisk`. The former is used as backing storage by your virtual memory system. The latter is used to hold the file system.