CS354/CS350 Operating Systems Winter 2004

Assignment Three (March 14 VERSION)

Design and Preliminary Testing Document Due: Tuesday March 23, 11:59 am
Full Assignment Due: Tuesday March 30, 11:59 am

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 = -DFILESYS_STUB -DRDATA -DSIM_FIX
```

to

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

(Don't forget to add in `-DUSE_TLB` 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. However, they are not exactly the same. 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 The Basic File System

Once you have switched from the stub file system to the basic Nachos file system implementation, your next task should be to read and understand the basic implementation. 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.

The 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 -s 2 creat`"). The Nachos file system has calls that are similar (but *not* identical) to these; the file system translates these calls into physical disk operations.

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 file system must *share* the NachOS simulated disk with your virtual memory system. Chances are that your virtual memory system is not designed to share. Similarly, the basic file system expects to have the disk to itself. This is generally not difficult to fix - a simple strategy is to partition the disk and give each system dominion over one partition - but you must fix it to prevent the two subsystems from interfering with one another.

3 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` and `Remove` 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.

   Ensure that the `Remove(char *filename)` system call works properly. 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 filesystem should actually remove the file, reclaiming all of the disk space used by that file, including space used by its header.

   It should be possible to create a new file with the same name as a removed file immediately after `Remove` has been called. This new new file should not interfere with existing handles to the old file.

2. Create a new system call for opening files called `NewOpen`. This call operates the same as the `Open` system call except that it adds a mode parameter that specifies whether the file is opened for reading (`"r"`), writing (`"w"`), or appending writes (`"a"`). The appending write mode opens the file with the file position just past the last character in the file (i.e., so the next write will append to the file). A file that is opened for writing (`"w"`) is truncated. A file is not truncated when any of the other modes are used. The initial file position is the beginning of the file unless the mode `"a"` or `"a+"` is used, in which case it is the end of the file.

   Each of these modes can be further modified by appending a `"+"`, which specifies that the file can be both read and written (this is called read/write mode). Yes, `"r+"` and `"w+"` specify the same mode.

   Attempting to write to a file opened in read only mode `"r"` or attempting to read from a file opened in write only mode `"w"` or `"a"` should return an appropriate error code.

   The interface for the new system call is

   ```
   OpenFileId NewOpen(char *name, char *mode);
   ```

   If the file can not be opened an appropriate negative error value should be returned (see `code/userprog/errno.h`) when possible. For example, trying to open a file with a mode other than those specified above should result in a failure to open the file and the error EINVAL should be returned.

3. Modify the kernel code for the existing `Open` system call so that it calls the code for `NewOpen` with the mode set to read/write. This will permit existing user test programs that use `Open` to continue to work without having to be altered while also permitting the use of the new `NewOpen` call.

4. 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.

5. Implement the `Seek` system call (the definition and more information is provided in syscall.h). Associated with each open file identifier will be a unique file (seek) position. 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 the same process could open the same file multiple times in the same program and obtain different file identifier each time the file is opened. There should be a unique file position associated with each file identifier.

6. Modify the file system so that it will support files at least as large as 4 Mbytes (**your design may support slightly larger files**). (In the basic file system, each file is limited to a file size of 3840 bytes.) A good design will not be wasteful, e.g., it will not consume enough data blocks to store 4 Mbytes of data if the file is only storing 100 bytes of data. You should have some ideas from class as to how to accomplish this. Note that, since you'll be implementing sparse files (see the next requirement) it will be possible to create a 4 Mbyte file even though the disk only has space for 262144 bytes.

7. Implement a mechanism to allow files to grow. A file should grow when a process tries to `Write` beyond the end of the file. 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` 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`.

   `Seek` calls on a file should succeed unless the seek position specified is beyond the maximum possible file size in your system in which case it should return EINVAL.

   Note that you should not be using any unnecessary blocks to store a file (that is you should implement sparse files).

   For example, if a program executes the sequence of operations

   ```
   id = Create("myfile");
   id = NewOpen("myfile", "w");
   x = Seek(id, 62019);
   x = Write(id, "A", 1);
   Close(id);
   ```

   The resulting file should have a length of 62020. Any subsequent reads from that file to file positions less than 62019 would return zeros. A read from position 62019 would return `"A"`. Any reads from file positions greater than 62019 would return end-of-file (EOF). In general any reads from file positions greater than or equal to the length of the file should return EOF (file position 0 is the start of the file).

8. The basic file system imposes a limit of 10 entries in the root directory. **You should modify this restriction to permit up to 20 entries in the root directory.**

9. Implement new system calls `SpaceId GetId()` and `SpaceId GetParentId()` to return the address space id of the calling process and the parent of the calling process, respectively. If the process has no parent `GetParentId` returns -1 (this will be true of the very first process created).

   Also, implement new system calls:

   - `int GetFileLen(OpenFileId id);`
     `// Return the length of the file in bytes.`
   - `int GetFileDataSectors(OpenFileId id);`
     `// Return the number of data blocks used by the file.`
   - `int GetFileTotalSectors(OpenFileId id);`
     `// Return the total number of blocks used to store the file (excluding the file header).`

   These calls will be used extensively in autotesting so ensure that these are implemented and working otherwise you will likely receive a very poor grade on the implementation portion of the assignment.

10. Implement the system call `CreatePipe`, in the filesystem. It takes a string (pathname) argument and creates an entry in the directory on the filesystem that acts as a pipe. Note that this is an entry in the directory in the same fashion that a file, directory, or symbolic link could be entries in a directory (if symbolic links were implemented in Nachos). Data that is written to this pipe should **not** be stored on disk. Instead, the data should be sent to the first process that calls or has called Read on a descriptor for that pipe. The signature for this call is as follows:

```
int CreatePipe(char *name);
int RemovePipe(char *name);
```

Both calls return a 0 on success and an appropriate negative error code on failure. One can not create a pipe with the same name as another directory entry (in that directory). For example, you can not have a directory that contains a file named "Barney" and a pipe named "Barney".

After the pipe has been created any process can open it using `NewOpen` or `Open` and can close it using `Close`. When opening a pipe with `NewOpen` the specified mode is ignored. Trying to `Seek` on a pipe is an error and an appropriate error code should be returned.

Trying to remove a named pipe by using `Remove` or `RemoveDir` should fail and return an appropriate error code. Likewise trying to remove a file or directory using `RemovePipe` should fail and return an appropriate error code.

A pipe is treated like a FIFO buffer. That is, all writes are to the end of the pipe and all reads are from the front of the pipe.

Removing a pipe that is open should be handled in the same way as removing a file that is open (see the directions earlier in this document for that case).

Note that if a process calls Read on a pipe that doesn't contain any data, it blocks until all data that it has asked to read has been read. If two or more processes call Read on the same pipe they should not receive the same data. The first process should first get the number of bytes that it asked for prior to the second process getting the number of bytes that it asked for and so forth. If two or more processes call write on the same pipe the data should be written to the pipe in the order in which the write calls were performed. The first process should write all of its data first prior to the second process writing all of its data and so forth.

A writer should block until all of the bytes that it wants to write in a particular write call have been read. For example, if a process writes 200 bytes to the pipe prior to anyone calling read on that pipe, it should block indefinitely until all 200 bytes have been read.

Note that a process that calls write with a large number of bytes, N (e.g., 1,000,000) should not cause the kernel to allocate space for and store all N bytes. Instead the kernel should buffer some number of bytes M ($M < N$) which can later be transferred to a process that calls read on that pipe. Remember that the writing process should be blocked during the call and it should remain blocked until all N bytes have been read. If you thin it is appropriate you can modify the existing code for the `Write` and/or other system calls. Note, that you **must not** change any of the system call interfaces or system call numbers (these are provided in a new version of syscall.h).

11. **Ensure that you add code to exception.cc to return ENOSYS to any system calls that you have not implemented, since we'll be running automated testing for this assignment. Failure to do so will be considered and treated as academic dishonesty.**

3.1 File System Testing Requirements

The testing requirements for assignment three are as follows:

1. Demonstrate that it is possible to create and remove files from the root directory, and that removal of open files behaves as required.
2. Demonstrate that a process can use (have open) several files at the same time, and that several processes can use a file simultaneously.
3. Demonstrate that it is possible to open, read from and write to small files.
4. Demonstrate that seek can be used to achieve non-sequential reading and writing of files.
5. Demonstrate that large files (up to the required maximum size) can be created, opened, written to and read from.

6. Demonstrate that `Write` can be used to make files grow.
7. Demonstrate that attempts to exceed file system limits (e.g., number of entries per directory, maximum file size, file system capacity) do not cause your system to crash.
8. Demonstrate that the system calls that return the file size and the number of data blocks return correct values.
9. Demonstrate that you've implemented sparse files properly.
10. Demonstrate that you've implemented `CreatePipe` and `RemovePipe` correctly and that pipes can be used to exchange data properly.

4 What to Submit

For assignment 3 you will submit your design document one week prior to the implementation deadline.

There will be no demos for Assignment 3. Instead, the TAs will download and build your code and run a series of tests to determine whether your implementation meets the requirements. The testing requirements listed in Section 3.1 should give you a good idea of what our tests will be looking for.

Our tests will assume that you have not changed any of the calling signatures or the meanings of any of the NachOS system calls, as defined in the new `syscall.h` posted to the course web page. The tests will not depend on any of the features that you were asked to implement for A1 or A2. It is OK for you to build on your A1 or A2 code, but we do not require it.