CS350 **Operating Systems** **Fall 2004**

Assignment Three

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.

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` 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. 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 filesystem should actually remove the file, reclaiming all of the disk space used by that file, including space used by its header.

4. Implement the `Seek` system call. Each open file must have its own 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.

5. 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 just under 4 KBytes.) A good design will not be wasteful, e.g., it will not require a file header to have enough data block pointers to point to 64 KBytes of data if the file is only 1 Kbyte long. You should have some ideas from class as to how to accomplish this.

6. 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 the `Read` call 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 is allowed, and should grow the file appropriately.

7. Implement a hierarchical directory structure. 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. The basic file system imposes a limit of 10 entries per directory. It is `not` necessary for you to relax this restriction.

   Your system should create a new directory when it is given a `CreateDir` system call. For example, `CreateDir(''foo'')` would create a directory (called "foo") under the original root directory. `CreateDir(''a/b'')` would create a directory "b" under the directory "a/", provided that directory "a/" already exists. (A call to `CreateDir` should never create more than one directory. All components of the pathname except the last should already exist, else `CreateDir` should return an error.) The same holds for `Create` used to create a file in a subdirectory.

   The `RemoveDir` system call should be able to remove directories. It is an error to remove a directory that is not empty.

   It is an error to `Open` a directory file. However, you should ensure that the NachOS filesystem list (`nachos -l`) and dump (`nachos -D`) utilities are able to list or print the entire directory hierarchy.

   To keep things simple, you may assume that legal file names have a simple form. Each filename consists of one or more pathname components, separated by "/" and optionally terminated by "/". Each pathname component consists at least one and no more than nine alphanumeric characters (only). The following are examples of legal pathnames:

   - `foo`
   - `foo/zam`
   - `foo16x/GHHHHk`

   The following are examples of illegal pathnames:

- `/`
- `flip//flop`
- `this is an illegal/.*@/../one`

Of course, illegal pathnames supplied as system call parameters should not cause your system to crash. Instead, the call should return an error.

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 a hierarchical directory structure can be constructed using `CreateDir`, that files can be created and used in non-root directories, and the non-root files and directories can be removed.
8. 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.

Your test suite should include other tests that you think are appropriate.