Project 3
Design and Implementation of a Secondary Storage System
CSE 521: Operating Systems

Alwin Tareen
<tareen>
tareen@buffalo.edu
2535-1583

May 4, 2011

The Technical Manual

1 Introduction

The overall objective of this project was to create a series of software modules that accommodated the flow of data from a simulated disk storage system to a file system. The means by which all of the software modules communicated was the Socket API. Each software module was designed to be either a client, or a server.

The client-server framework was used to accomplish several types of software tasks. The simple tasks were to reverse the letters in a text string, and also for a client to provide a listing of all the files on a server's directory. The more complicated tasks were handled by a more elaborate setup of the client-server model, namely, the disk storage filesystem. With this model, we were required to manage such system tasks as file creation and deletion. Also, we had to read and write large amounts of data by sectioning it into individual blocks, and managing the flow and placement of these. Finally, we had to implement a directory structure that provided a small set of UNIX commands, such as making and deleting a directory.

2 The Problem Implementation

The successful implementation of the simulated disk-storage and filesystem, required that we designed several distinct software frameworks. They are described as follows:

The basic client-server These were two very basic software modules that communicated through inet-domain sockets. The client sent a string of text to the server, who processed it with a `reverse()` function, then sent the result back to the client.

The directory listing server This framework was very similar to the basic client-server setup, as it had two modules communicating through inet-domain sockets. However, it had a slightly more complicated data handling methodology. The client sent the UNIX `ls` command, followed by a series of options. The server handled this request by calling the `fork()` and `execvp()` functions. The resulting directory listing output was sent over to the client, who read it from the socket and displayed the contents.

The basic disk-storage system This was a more advanced configuration of the client-server framework, in the sense that it provided a real-time simulation of a physical disk. The system had to manage independent blocks of data passing between the modules. In doing so, we had to implement three basic operating commands: information request, read request, and write request. The blocks of data were managed by using a linear index, based on calculations derived from the number of tracks, and the number of sectors per track.

The file system server This software module required a highly complicated rendering of the client-server model. It was the sort of program that could only be successfully implemented by the most elite, experienced or capable software engineers. Specifically, two tables had to be set up to keep track of the files and data: the File Table, and the File Allocation Table (FAT). The File Table was implemented by using a `struc`, and the FAT was set up by using an `array`. The file system performed basic file operations, such as create and delete file, by interfacing with the disk storage server, as well as a file client. In this manner, the file system performed both client and server roles, concurrently. It facilitated the request and delivery of data blocks between the disk storage module and the file client module.

The file system directory structure server This software configuration provided additional functionality to the file system server, namely, by implementing several basic UNIX functions. The user could now create and delete directories, as well as change and display the current working directory. The algorithms used to implement these directory functions were generally based on the ones used in the previously described file system server.

3 The System Diagram

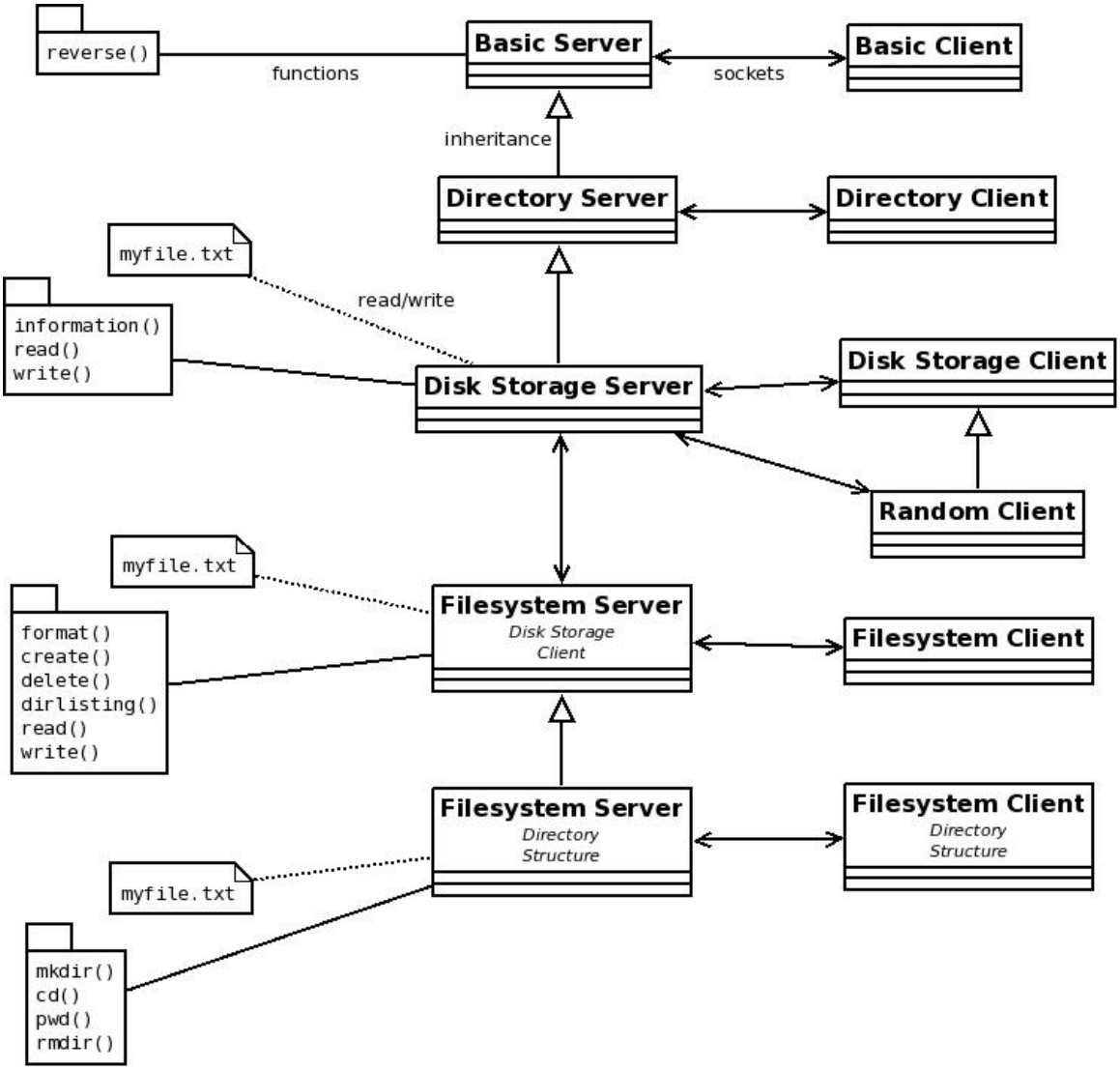

Figure 1: A graphical representation of the software modules used in the Disk Storage Filesystem Project.

4 The Disk Protocol

The server understands the following commands, and it gives the following responses:

- `I`: **Information request**. The disk returns two integers representing the disk geometry: the number of tracks, and the number of sectors per track.
- `R` *c s*: **Read request** for the contents of track *c* sector *s*. The disk returns '1' followed by those 128 bytes of information, or '0' if no such block exists. (This will return whatever data happens to be on the disk in a given sector, even if nothing has ever been explicitly written there before.)

- `W` *c s l* `data`: **Write request** for track *c* sector *s*. *l* is the number of bytes being provided, with a maximum of 128. The `data` is those *l* bytes of data. The disk returns '1' to the client if it is a valid write request (legal values of *c, s* and *l*), or returns a '0' otherwise. In cases where $l < 128$, the contents of those bytes of the sector between byte *l* and byte 128 is undefined.

5 The File System Protocol

The server understands the following commands, and it gives the following responses.

- `F`: **Format**. This command will format the filesystem on the disk, by initializing any or all of tables that the filesystem relies on.
- `C` *f*: **Create file**. This will create a file named *f* in the filesystem. The return codes are as follows:

  0 = successfully created the file.

  1 = a file of this name already existed.

  2 = some other failure.

- `D` *f*: **Delete file**. This will delete the file named *f* from the filesystem. The return codes are as follows:

  0 = successfully deleted the file.

  1 = a file of this name did not exist.

  2 = some other failure.

- `L` *b*: **Directory listing**. This will return a listing of the files in the filesystems. *b* is a boolean flag: if '0' it lists just the names of all the files, one per line; if '1', then it includes the length, in bytes, of each file.
- `R` *f*: **Read file**. This will read the entire contents of the file named *f*, and return the data that came from it. The message sent back to the client is, in order: a return code, the number of bytes in the file (in ASCII), a white-space, and finally the data from the file. The return codes are:

  0 = successfully read the file.

  1 = no such filename exists.

  2 = some other failure.

- `W` *f l* `data`: **Write file**. This will overwrite the contents of the file named *f* with the *l* bytes of `data`. If the new data is longer than the data previous in the file, the file will be made longer. If the new data is shorter than the data previously in the file, the file will be truncated to the new length. The return codes are:

  0 = successfully written to the file.

  1 = no such filename exists.

  2 = some other failure.

Appendices

A The makefile Compilation Script

```
all: bclient bserver dclient drandom dserver fclient fdserver fserver lsclient
lsserver

exes = bclient bserver dclient drandom dserver fclient fdserver fserver lsclient
lsserver

bclient: bclient.c cse421header.h
g++ bclient.c -o bclient

bserver: bserver.c cse421header.h
g++ bserver.c -o bserver

dclient: dclient.c cse421header.h
g++ dclient.c -o dclient

drandom: drandom.c cse421header.h
g++ drandom.c -o drandom

dserver: dserver.c cse421header.h
g++ dserver.c -o dserver

fclient: fclient.c cse421header.h
g++ fclient.c -o fclient

fdserver: fdserver.c cse421header.h
g++ fdserver.c -o fdserver

fserver: fserver.c cse421header.h
g++ fserver.c -o fserver

lsclient: lsclient.c cse421header.h
g++ lsclient.c -o lsclient

lsserver: lsserver.c cse421header.h
g++ lsserver.c -o lsserver

clean:
rm *.o
rm $(exes)
```

B The README Documentation File

```
CSE 521 Project 3: README File
------------------------------
Programmer: Alwin Tareen

Notes:
------
The disk storage server and file system servers required the use of an
external data file, called ‘‘myfile.txt’’. This file was where all the data
blocks were written to and read from. The directory structure was implemented
in a separate source code file, named fdserver.c. The submitted files were:


Header File
-----------
* cse421header.h


Problem 1:
----------
* bclient.c
* bserver.c


Problem 2:
----------
* lsclient.c
* lsserver.c


Problem 3:
----------
* dclient.c
* dserver.c
* drandom.c


Problem 4:
----------
* fclient.c
* fserver.c


Problem 5:
----------
* fdserver.c
```