CSE421/521 Introduction to Operating System
Fall 2007
Project #3
Design and Implementation of Secondary Storage System

November 6, 2007

1 Objective

- Design and implement a basic disk-like secondary storage server.
- Design and implement a basic file system to act as a client using the disk services provided by the server designed above.
- Evaluate the perfromance of the disk server you implemented for various size of request caches.
- Study and learn to use the *socket* API. Sockets will be used to provide the communication mechanism between (i) the client processes and the file system, and (ii) the file system and the disk storage server.

2 Problem Description

This project will be developed in steps to help you get a good understanding of the concepts, and to encourage good modular development. The first two parts will familiarize you with the socket-API. Sockets will serve as the communication backbone for the disk system that you will build in the later parts.

1. (10 points) **Basic client-server:** Create two programs: a client and a server. Let the two communicate through unix-domain sockets. The client will pass a string to the server, the server simply reverses the string, and returns it to the client which prints it out.
2. (10 points) **Directory listing server:** Next, extend part one to implement a directory listing server, and sample client. The directory listing server waits for clients to connect to it. The client provides the server with some data. The data is in the form of an array of parameters to the `ls` program (like the one you used in the first project). The server forks a child process to handle the request. In the child, using one of the `exec*` family of system calls, the `ls` command is executed with the appropriate parameters as indicated by the client. The output of the `ls` program should be written back via the socket to the client. At the client end, display the results transmitted by the server.
3. (25 points) **Basic disk-storage system:** Implement, as a unix-domain socket server, a simulation of a physical disk. The simulated disk is organized by cylinder and sector. You should include in your simulation something to account for track-to-track time (using `usleep(3C), nanosleep(3R)` etc.). Let this be a value in microseconds, passed as a command-line parameter to the disk server program. Also, let the number of cylinders and number of sectors per cylinder be command line parameters. Assume the sector size is fixed at 256 bytes. Your simulation should store the actual data in a real disk file, so you'll want a filename for this file as another command line option. (You'll probably find that the `mmap(2)` system call provides you with the easiest way of manipulating the actual storage. However you are allowed to use file and file API to simulate the storage representing your disk)

The Disk Protocol

The server must understand the following commands, and give the following responses:

- `I`: information request. The disk returns two integers representing the disk geometry: the number of cylinders, and the number of sectors per cylinder.
- `S` b: set the synchronization mode. b is either '0' or '1'. If '0', then disk is in non-synchronized mode, making write commands non-blocking (meaning that write commands will return their status before the data is necessarily written to disk). If '1', then writes do not acknowledge anything until the data is written to disk. The default mode is synchronous (S = '1').
- `R` c s: read request for the contents of cylinder c sector s. The disk returns '1' followed by those 256 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 cylinder c sector s. l is the number of bytes being provided, with a maximum of 256. 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. At what point in time this return code is sent, depends on the setting of the `S` synchronization mode. In cases where $l < 256$, the contents of those bytes of the sector between byte l and byte 256 is undefined—you may implement it any way that you want (for example: old data; zero-fill; etc.).

The data format that you *must* use for c s and l above is a regular ASCII string, followed by a white-space (space, tab or newline) character. So, for example, a read request for the contents of sector 17 of cylinder 130 would look like: `R␣130␣17␣`. And then 257 bytes of data would be returned: the character `1`, followed immediately by the 256 bytes read from that sector.

The Disk Clients

The clients that you need to write here are mostly for testing purposes. The real use of this "disk" will be the filesystem implemented in the later parts of this project.

So, for testing/demonstration purposes, you need to implement two clients:

(a) Command-line driven client: This client should work in a loop, having the user type commands in the format of the above protocol, send the commands to the disk server, and display the results to the user. (It would be difficult to debug your disk server without such a debugging tool anyway.)

(b) Random data client: This client should query the disk for its size (using the `I` command), and then generate a series of N random requests to the disk. Each request should randomly be `W` or `R`, and the cylinder number and sector number should be randomly chosen from the valid range for that disk. Write requests should all write 256 bytes of data. The value of N, and the seed for the random number generator, should be specified on the command line. This client must not print out the data that is read/written - simply printing a single character representing each request (to show progress) is sufficient.

4. (30 points) **File system server:** Implement a flat filesystem that keeps track of files in a single directory (table). The filesystem should provide operations such as: initialize the filesystem, create a file, read the data from a file, write a file with given data, append data to a file, remove a file, etc..

   The following features are **not required** in your implementation:
   - The ability to create and use subdirectories.
   - Advanced access to the filesystem such as filesystem status report (free space, number of files, percentage of used space, fragmentation etc.).
   - Filesystem integrity checks.
   - File permissions.

   You will find that in order to provide the above file-like concepts, you will need to operate on more than just the raw block numbers with which your disk server provides you. You will need to keep track of things such as which blocks of storage are allocated to which file, and the free space available on the disk. A file allocation table (FAT) can be used to keep track of current allocation. Free space management involves maintaining a list of free blocks available on the disk. Two alternative designs are suggested: a bit vector

(1 bit per block) or chain of free blocks. Associated with each block is a cylinder# and sector#. Writing to a file gets converted to writing into a specific cylinder# and sector#. Note that all this information needs to be stored on the disk, as the filesystem module could be shut down and restarted and the disk data should be persistent.

Implement this file system server as another unix-domain socket server. So, this program will be a server for one unix-domain socket; and also be a client to the disk-server unix-domain socket from the previous parts.

The Filesystem Protocol

The server must understand the following commands, and give the following responses. (See the Appendix for some *suggested, but not required,* algorithms to implement some of these operations.)

- `F`: format. Will format the filesystem on the disk, by initializing any/all of tables that the filesystem relies on.
- `C` f: create file. This will create a file named f in the filesystem. Possible return codes: 0 = successfully created the file; 1 = a file of this name already existed; 2 = some other failure (such as no space left, etc.).
- `D` f: delete file. This will delete the file named f from the filesystem. Possible return codes: 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' it includes other information about each file, such as file length, plus anything else your filesystem might store about it.
- `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. Possible return codes: 0 = successfully read 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. A return code is sent back to the client. Possible return codes: 0 = successfully written file; 1 = no such filename exists; 2 = some other failure (such as no space left, etc.).
- `A` f l `data`: append to file. This will append the l bytes of `data` to the end of the current contents of the file named f. A return code is sent back to the client. Possible return codes: 0 = successfully appended to file; 1 = no such filename exists; 2 = some other failure (such as no space left, etc.).

For testing/demonstration purposes, you need to implement a command-line driven client, similar to the one that you wrote for the disk server. It should work in a loop, having the user type commands in the format of the above protocol, send the commands to the disk server, and display the results to the user. (It would be difficult to debug your disk server without such a debugging tool anyway.)

5. (15 points) **User level commands** To make the filesystem useful, implement a few basic command-line user level commands. Each of these should be very short programs. They are each simply a client which talks to the filesystem unix-domain socket.

   Implement:

   (a) `mycat` *filename* : show the contents of *filename*.

   (b) `mywrite` *filename* : Read stdin and overwrite *filename* with all the data from stdin.

   (c) `myappend` *filename* : Append all the data from stdin to *filename*.

   (d) `myls` : show the list of files in the directory.

   (e) `mylsl` : show the long form list of files in the directory.

   (f) `myrm` *filename*: remove *filename*.

   (g) `mycp` $filename_1$ $filename_2$: copy $filename_1$ to $filename_2$ (overwriting $filename_2$ if it already exists).

(h) `mymv` *filename*$_1$ *filename*$_2$: rename *filename*$_1$ to *filename*$_2$ (overwriting *filename*$_2$ if it already exists).

(i) `mynewfs` : format the filesystem, removing all contents.

6. (10 points) Technical Report. Provide a detailed printed technical report. Use a format similar to project2.

3 Miscellaneous

1. Read chapters 5 and 6 of your text book.
2. Your programs should be robust. If any of the calls fail, it should print error message and exit with appropriate error code. Always check for failure when invoking a system call. By convention, most UNIX calls return a value of negative one (-1) in case of an error, and you can use this information for a graceful exit. Use perror(3) library routine to print error message when appropriate.
3. Please comment your code well. C/C++ needs it.

4 Material to be Submitted

1. Submit the **source code** for all the programs. Use meaningful names for the file so that the contents of the file is obvious from the name. Also provide a README file.
2. Test runs: It is very important that you show that your program works for all possible inputs. Submit a script file for each program clearly showing the working for correct input as well as graceful exit on error input.
3. You are required to submit a Makefile for your project. It should be set up so that just typing `make` in your submission directory should correctly compile all programs for all parts.
4. You need to submit a printed Technical Report. It should contain two parts:
   (a) A users manual, describing how to interact with your programs
   (b) A technical manual, describing your design disk, and of your filesystem. You must give complete technical details of the format of all data-structures used in your program, as well as the main algorithms, etc..

   This Technical Report should be a professional looking document. Submit a pdf or MS Word document.

5 Due Date

12/5/2007. Submit on-line before midnight.

6 Appendix: Possible algorithms

Here are some hints/thoughts/suggestions of rough algorithms that you might use to implement some of the filesystem-level features. You are not required to actually use any of these. They are simply here to guide your thinking.

The concept is that there is a fixed sized FAT in the first (few) sector(s) of the disk. Each FAT entry is big enough to hold a block number (cylinder and sector). There is one FAT entry corresponding to each block on the disk. The value in each FAT entry represents the next block in the file, following the block represented by the FAT position. The special mark of EOF (1) indicates that there are no further blocks in the current file. The special mark of EMPTY (0) represents that the block corresponding to this FAT entry is not part of any file.

The directory is a fixed number of sectors, following the FAT. Each directory entry contains space for a fixed-length filename, the length of the file, a pointer to the FAT index corresponding to the first sector of the data of the file, plus any other data that you might want to store (deleted flag, etc.). For zero-length files, the EOF marker is put into this FAT position.

Create: search the directory for that filename; if it already exists, fail;
search for an empty directory entry;
search for a free block in the FAT;
record the filename in that directory entry;
record the cylinder and sector numbers in that directory entry;
record the filelength = 0 in the directory entry;
mark the FAT entry field in the directory entry with an EOF marker

Read: search the directory for that filename; if none exists, fail;
read the length field from the directory entry, and return it;
get the first block number in the directory entry;
read the data from that sector, and return the data;
search for next block of the file;
just keep doing that until we reach the end of file

Write: search the directory for that filename; if none exists, fail;
get the first block number in the directory entry;
write the first sector's worth of data into that block;
if there are more sectors already allocated to the file, use next one;
if no more, then need to allocate free block;
to allocate a free block, update the FAT entry of the current block to pointing the newly allocated block;
and keep writing until everything is written;
update the file length in the directory entry;
if the new file was shorter, update other FAT entries to EMPTY