CS421: Introduction to Operating Systems
Fall 1998 : Project #3
Design and Implementation of Secondary Storage System

Instructor: Dr. Davin Milun

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.
- 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.
- Optionally: learn about the System V message queue IPC system.
- Very optionally: learn about the Solaris 2.6 doors IPC system.

2 Problem Description

2.1 Introductory Parts: Due Sunday November 15th

The Introductory Parts will primarily 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. Also, there is the Extra Credit option of using System V message queues.

1. (4 points) **Basic client-server:** Create two programs: a client and a server. Let the two communicate through a stream-based unix-domain socket. The client will pass a null-terminated ASCII string to the server. The server will reverse the string, and return it the reversed string to the client. The client should then print out the string that it received from the server, and quit. The server must continue to run, waiting for the next connection.

   For 2 Extra Credit points (see the Extra Credit section for details) provide a second implementation of part 1, using message queues rather than sockets.

2. (7 points) **Integer output server:** Extend part one to implement an integer output server, and sample client. The server waits for clients to connect to it. The client provides the server with some data.

   The data is in the form of a single character (representing the required base) followed by four bytes (representing an integer). The first byte must be one of: `d`=decimal, `o`=octal, `x`=hexadecimal. The four bytes must be *the four raw bytes* that represent an integer, (*not* the ASCII/string representation of the integer!)

   The server receives that data, converts the integer into the requested base, and sends the resulting number as a null-terminated ASCII string back to the client, and then closes the connection. The server continues running, waiting for the next connection.

   The sample client should take the base and integer from the command line, send them to the server, receive the results, print them out, and then quit.

   For 3 Extra Credit points (see the Extra Credit section for details) provide a second implementation of part 2, using message queues rather than sockets.

3. (7 points) **Trivial web browser:** In this part, you will be writing only a client. The client will be one that can retrieve a document from the World Wide Web, by connecting to any Hypertext Transfer Protocol (HTTP) server on the Internet.

Client command line: `hostname pathname`
The client should connect to TCP port 80 on `hostname`, and send the simplest HTTP protocol GET request: "`GET /pathname`".

The client should then simply print out all the data sent to it from the server, and then quit.

NOTE: This is the bare minimum request that a web server will accept. The full protocol is much richer, and more complex than this. The current version of the protocol is version 1.1, and is described in RFC 2068. Which, as well as thousands of places on the net, you can look at locally as `/ftp/pub/rfc/rfc2068.txt.gz`.

Also note that typing a `~` on the command line will not give a `~` to your program in an argument, because the shell will expand it into the directory name. Therefore, to read a user's homepage, you'll need to put the pathname in quotes on the command line. For example: `part3 www.cse.buffalo.edu "~milun"`

4. (4 points) **mmap demo**. Write a program which `mmap`'s the contents of a file (the filename provided on the command line), and replaces the 23rd to 33rd characters of that file with "`MMAP RULES!`". If the file contains fewer than 33 characters, then the program should print a warning message, and exit. If the file contains more than 2000 characters, then it should also replace character 2000 with an "`A`". When it works correctly, there should be no printed output generated by the program—the contents of the file named on the command line should simply change.

   See the `mmap(2)` man page for details, as well as sample code from recitations. To determine the length of the file, you'll want to see the `fstat(2)` man page.

2.2 Main Parts: Due Thursday December 10th

5. (25 points) **Basic disk-storage system:** Implement, as a stream-based unix-domain socket server, a simulation of a physical disk. The simulated disk is organized by cylinder and sector. The disk server should take as command line parameters: the number of sectors per cylinder; and the number of cylinders on the disk. The number of bytes per sector is fixed at 128 for this particular implementation.

   Your simulation should store the actual data in a real disk file, and this data should be persistent between runs of the server. You'll thus need the filename of this file as another command line parameter. Your program should assume that the file already exists, and should quit with an error message if the file does not exist, or if the file is not large enough to hold the disk data as calculated by the other command line parameters. (You'll probably find that the `mmap(2)` system call provides you with the easiest way of manipulating the actual storage. Also, the easiest way of creating a new empty file from the command line, is probably the `/usr/sbin/mkfile` command.)

   The general scheme under which the disk-storage system should work is that a client will connect to the server, and then issue a sequence of Disk Protocol requests until it is done with needing the server, and then disconnect from the server. It is perfectly acceptable for your server to be an iterative server, so that it can only be connected to a single client at any one time.

   ### The Disk Protocol

   The server must understand the following commands, and give the following responses. **All numeric data specified in this protocol spec is to be sent in raw encoded integer form.** (Since we're using Unix Domain Sockets, both ends of the socket will always be on the same system, so endian-ness is not an issue.) Note that the return codes are not numeric data, but are the single characters '0' etc.

   - `I`: Information request. The disk returns a return code of '0', followed by three integers representing, in order, the disk geometry: the number of cylinders on the disk, the number of sectors per cylinder, and number of bytes per sector (128).
   - R*cs*: Read request for the contents of cylinder *c* sector *s*. The disk returns '0' followed by the 256 bytes of information from that sector, or '1' if no such block exists. (This command will return whatever data happens to be on the disk in a given sector, even if nothing has ever been explicitly written to that sector before.)
   - W*csl*`data`: Write request for cylinder *c* sector *s*. *l* is the number of bytes being provided, with a maximum of one sector (128). The `data` is those *l* bytes of data. The disk returns '0' to the client if it is a valid write request (legal values of *c*, *s* and *l*), or returns a '1' otherwise. In cases where $l < 128$, the contents of those bytes of the sector between byte *l* and byte 128 is undefined—you may implement it any way that you want (for example: leave the old data; zero-fill; etc.).
   - `Q`: Quit request. The disk returns a return code of '0', and then shuts itself down, causing the disk server program to terminate.

- `D`: Disconnect request. The disk returns a return code of '0', and then closes the connection to the client. This is the polite way for a client to disconnect, (rather than the client just closing the connection without warning).
- Any other character should simply be ignored: no return code should be sent at all.

The data format that you *must* use for *c s* and *l* above is the raw integer data. So, for example, a read request for the contents of sector 17 of cylinder 130 would look like: `R`, the 4 bytes representing the integer 130, the 4 bytes representing the integer 17. And then 129 bytes of data would be returned: the character `0`, followed immediately by the 128 bytes read from that sector.

The Sample Disk Client

The client that you need to write here is 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 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.) The client only needs to do minimal sanity checking on the entered data. This is a mini test client, not a major part of the project.

6. (35 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.
   - Filesystem status reports (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 that 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 of current allocation, and mapping of blocks to files. Free space management involves maintaining a list of free blocks available on the disk. (Some possible methods are: a bit vector (1 bit per block), chain of free blocks, or simply searching for free blocks in the FAT.) 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 meta-information also 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 stream-based 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 part. (And you'd be well advised to structure your program similarly, so that you'll have separate parts, in separate functions, that do each of these two sockets. In particular, your file system server should probably have functions to implement each of the Disk Protocol client requests, called something like `disk_read`, `disk_write`, etc.)

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.) **All numeric data specified in this protocol spec is sent as ASCII strings.**

- `F`: Format. Will format the filesystem on the disk, by initializing any/all of tables that the filesystem relies on. Possible return codes: 0 = format completed successfully; 1 = format failed.
- `C` *f*: Create file *f*. This will create a new, empty, 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 *f*. 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 of type *b*. This generates a listing of the files in the filesystems. It returns a return code of '0', followed by a space, followed by an ASCII decimal string representing the number of files in the listing, followed by a space, followed by one line of data for each file. *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 each file (such as modification date, etc.).

- `R` *f*: Read file *f*. 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, a white-space, the number of bytes in the file (as an integer sent as an ASCII decimal string), 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`. *l* should be sent as an ASCII string, followed by a space. (Note that *l* is not limited by the size of a sector, and need not be a multiple of sector size.) If the new data is longer than the data previously in file *f*, then 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 wrote the 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.).
- Any other character should simply be ignored: no return code should be sent at all.

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.)

7. (18 points) **User level commands** To make the filesystem useful, implement a few basic command-line user level commands. Each of these should be a short program. Each is an independent client program which talks (using the Filesystem Protocol) to the filesystem unix-domain socket from the previous part.

   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.

3 Extra Credit

In this project, I am providing a way for students to receive extra credit. This extra credit can be worth at most 40 points for the best conceivable project, however it's unlikely that anyone will receive that many extra credit points.

The curve for this project will be based on the regular 100 required points. The extra credit will be used as an 8% extra grade, added in to your overall course grade, (but your course grade will still be divided by the regular total of 100%, not 108%). Thus the extra credit will result in grade improvements for some, and grade deterioration for none in the class.

Ways of obtaining extra credit:

- Do the Extra Credit options within the Intro Parts.
- Extend the filesystem, by adding multiple directories. To to this you should extend the filesystem protocol by adding the `M` (make directory), `N` (change directory), `O` (delete empty directory) commands, and corresponded part 6 commands to take advantage of them. (Up to 15 points, depending on level of implementation.)
- Extend the filesystem is some of the other ways described above in the list of things that you are not required to implement. (Points dependent on what you implement, and how well.)

- Implement the communication between the filesystem and the disk server using System V Message Queues, rather than Sockets. Note that this is *instead* of implementing the Sockets version. However, the communication between the client programs and the disk server must still be done using sockets. (5 points)
- Implement the communication between the filesystem and the disk server using the Solaris Doors mechanism, rather than Sockets. Note that this is *instead* of implementing the Sockets version. However, the communication between the client programs and the disk server must still be done using sockets. (10 points)

You are permitted to provide two separate implementations, if you want to do both of these extra last two extra credit options. If you write the file server well, it should only require changing one "module" of your server.

4 Hints and Miscellaneous comments

1. There is a graphical Java implementation and demonstration of (an older version of) this filesystem project, available for you to run as `http://www.cse.buffalo.edu/~jmkirsch/java/VFileSystem.html`. (It was written for me by Jacob Kirsch, as an independent study project.) It should give you a reasonable guide as to how the whole system fits together, and what data is transfered between which clients/servers at what points.
2. Read the notes that you got from Makin' Copies. There is lots of material and examples and explanations relevant to this project, especially in the last part of the notes. Also, read chapters 11 and 12 of your text book.
3. See `~milun/EXAMPLES/messagequeues/` for example Message Queues code. Also, see the `msgget(2), msgsnd(2), msgrcv(2)` man pages.
4. If you're planning on trying Solaris Doors, you're mostly on your own. You probably want to see the following man pages: `door_create(3X)`, `door_call(3X)`, `door_info(3X)`, `door_return(3X)`
5. Your program *must* 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 any system or library call. By convention, most UNIX calls return a value of NULL or negative one (-1) in case of an error (but always check the `RETURN VALUES` section of each man page for details), and you can use this information for a graceful exit. Use `cerr, perror(3)`, or `strerror(3)` library routines to generate/print appropriate error messages.

5 Material to be Submitted

1. Submit the **source code** for all the programs. Use meaningful names for the files so that the contents of the file is obvious from the name.
2. Submit a `README` file describing which program file corresponds to which part of the project. Also include any special notes, instructions, etc..
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 parts.
4. You need to submit a printed Technical Report for the Main Parts (parts 4–6), including any Main Parts Extra Credit sections. 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. It should preferably be typeset (using LaTeX, FrameMaker, IslandWrite, MS Word etc.), and should definitely not be handwritten (except possibly for diagrams).

6 Due Dates

- Submit on-line the Introductory Parts by 11:59pm of Sunday November 15.
- Submit on-line the Main Parts by 11:59pm of Thursday December 10.
- The printed Technical Report is due in class the following day, Friday December 11.

7 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;
record the filename 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

- Alternatively, you could store the directory (as described above) in a fixed place on the disk (such as the first few sectors); and have the directory entry point at the first data sector. And have every data sector use the first few bytes to point at the next data sector of that file, or EOF if none. In other words, implement something equivalent to a linked list of sectors. Note that you'll still need to keep a list/table of available/allocated sectors somewhere though.