Project 3: Design and Implementation of File System

April 16, 2024

1 Objectives

1. Design and implement a basic disk-like secondary storage server.
2. Design and implement a basic file system to act as a client using the disk services provided by the server designed above.
3. Study and learn to use the socket API. Sockets are 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 Statement

This project will be developed in several steps to help you understand the concepts and encourage modular project development. You must adhere to the specifications given below. To share your experiences effectively, you are recommended to keep a record of your insights during the coding. This includes writing down any challenges encountered or bugs costing you a long time for debugging. By doing so, you'll avoid forgetting the details of your experiences when writing the report.

Step 1: Design a basic disk-storage system

Description: Implement, as an Internet-domain socket server, a simulation of a physical disk. The simulated disk is organized by cylinder and sector.

- You should include the seek time in your simulation to account for track-to-track time (using usleep(3C), nanosleep(3R), etc). Make the seek time a value in microseconds, passed as a command-line parameter to the disk server program.
- You should accept the number of cylinders and the number of sectors per cylinder from command line parameters. Assume the sector size is fixed as 256 Bytes.
- Your simulation should store the actual data in a real disk file, so you'll need to accept a filename for this file as another command line option.

Hints:

1. You may 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.

2. Given the challenges of socket programming, we have provided some functions. Please refer to the code template provided on Canvas for how to use these functions. If you believe there are bugs in the provided code, please contact the TA on time.

The Disk Server: The server should be able to understand the following commands and give corresponding responses:

- **I:** Information request. The disk returns two integers representing the disk geometry - the number of cylinders, and the number of sectors per cylinder.
- **R c s:** Read request for the contents of cylinder c sector s.
  - The disk returns **Yes** followed by a whitespace and those 256 bytes of information, or **No** if no such block exists.
  - Hint: This will return whatever data happens to be on the disk in a given sector, even if nothing has ever been explicitly written before.
- **W c s l data:** Write a 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 **Yes** to the client if it is a valid write request (legal values of c, s, and l), or returns a **No** otherwise.
  - In cases where $l < 256$, the contents of those bytes of the sector between byte l and byte 256 are undefined, use zero-padded contents.

Hints:

1. The data format that you must use for c, s, and l above is a regular ASCII string, followed by a space character. So, for example, a read request for the contents of sector 17 of cylinder 130 would look like `R 130 17`.
2. Strings are not equivalent to byte arrays: in a string, 0 represents the end of the string, but this is not the case in a byte array. Besides, many bytes cannot be printed as visible characters. Therefore, DO NOT use functions like `strlen()` or `printf()` on the *data*. The disk server will not interact with users directly, so writing raw bytes to the file descriptor is acceptable.

The Disk Client: The client that you write here is mostly for the unit testing of the disk module. The actual front-end user of this "disk" will be the file system implemented in the later steps. The 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. Besides, you need to deal with ASCII invisible characters. For example, add a command that prints data in binary or hexadecimal. Otherwise, it would be difficult to debug your disk server without such a debugging tool.

Command line:

a) Server:

```
./BDS <DiskFileName> <#cylinders> <#sector per cylinder> <track-to-track delay> <port=10356>
```

b) Client:

```
./BDC <DiskServerAddress> <port=10356>
```

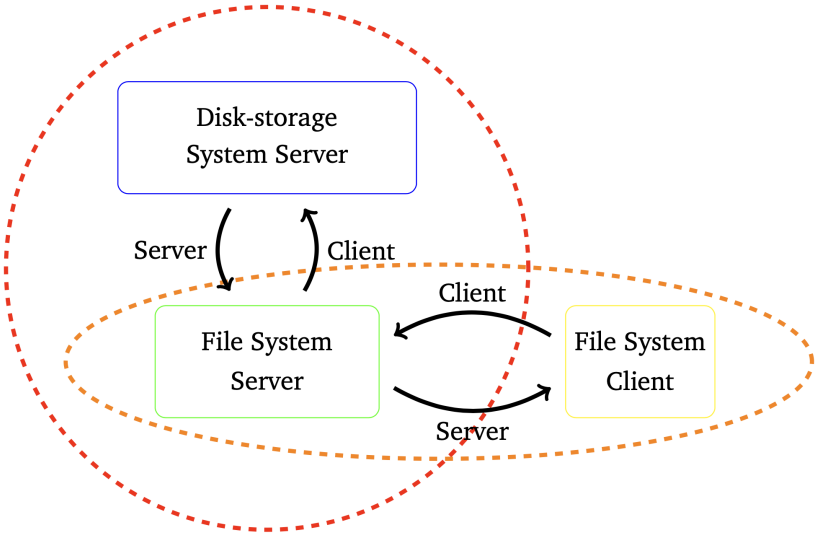

Figure 1: File System Architecture

Step 2: Design a basic file system

Implement an inode file system. The file system should provide operations including:

- Initialize the file system
- Create a file
- Read data from a file
- Write the given data to a file
- Append data to a file
- Remove a file
- Create directories

To provide the above file-like concepts, you need to operate on more than just the raw block numbers that your disk server provides to you. You need to keep track of which blocks of storage are allocated to which file, and the free blocks available on the disk.

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 a chain of free blocks. Associated with each block is a cylinder# and sector#. "Writing to a file" is converted to "writing into a disk location identified by (cylinder#, sector#)". All this information should be stored on the disk, as the file-system module in the memory is not persistent, and it could be shut down and restarted with data lost.

The file system needs to support files of at least 16 KB in size, and filenames must be at least 8 Bytes long.

Implement this file system server as another UNIX-domain socket server. On one hand, this program will be a server for one UNIX-domain socket; on the other hand, it will be a client to the disk server UNIX-domain socket from the previous parts, as shown below.

The File-system Protocol: The file system server should be able to accept and respond to the following commands.

- **f:** Format. This will format the file system on the disk, by initializing any/all of the tables that the file system relies on.
- **mk *f*:** Create a file. This will create a file named f in the file system.
- **mkdir *d*:** Create a directory. This will create a subdirectory named d in the current directory.
- **rm *f*:** Delete file. This will delete the file named f from the current directory.
- **cd *path*:** Change the directory. This will change the current working directory to the path. The path is in the format in Linux, which can be either a relative path or an absolute path. When the file system starts, the initial working path is "/". You need to handle **"."** and **".."** as special cases.
- **rmdir *d*:** Delete a directory. This will delete the subdirectory named d within the current directory.
- **ls:** Directory listing. This will return a listing of the files and directories in the current directory. You are also required to return other meta information, such as file size, last update time, etc.
- **cat *f*:** Catch file. This will read the file named *f*, and return the data in it.
- **w *f l data*:** Write data into 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 previously in the file, the file will be extended longer. If the new data is shorter than the data previously in the file, the file will be truncated to the new length.
- **i *f pos l data*:** Insert data to a file. This will insert l bytes of data into the file after the $pos-1^{th}$ character but before the pos^{th} character (0-indexed). If the *pos* is larger than the size of the file, append the l bytes of data to the end of the file.
- **d *f pos l*:** Delete data in the file. This will delete l bytes starting from the *pos* character (0-indexed), or till the end of the file (if l is larger than the remaining length of the file).
- **e:** Exit the file system.

For testing/demonstration purposes, you need to implement a command-line client, similar to the one that you wrote for the disk server in Step 1. The file-system client should work in a loop, allowing the user to type commands in the format of the above protocol, send the commands to the file server, and display the results to the user.

Hints: We suggest following the below steps to implement your file system:

- Implement an in-memory file system, reading commands via STDIN, where the data is stored in memory and disappears after the system reboot.
- Write the data to a raw file to achieve the data persistence.
- Implement the file system as a server, accepting commands passed through a client.

- Replace the raw file with the disk server implemented in Step 1. This should only require modifying a few lines of code.

Command line:
a) Disk Server:

```
./BDS <DiskFileName> <#cylinders> <#sectors per cylinder> <track-to-track delay> <port=10356>
```

b) File-system Server:

```
./FS <DiskServerAddress> <BDSPort=10356> <FSPort=12356>
```

c) File-system Client:

```
./FC <ServerAddr> <FSPort=12356>
```

Step 3: Support multiple users in the file system

A real-world file system is supposed to store data for more than one user. In this step, you need to make some changes to your file system so that it becomes a multi-user system. The key problem to solve here is to separate the stored files and data for different users. Instead, how to support the parallel operations by multiple clients is considered in the "Extension 2 - Multi-Clients" in Step 4 as an optional extension. In other words, in this step, you can safely assume different users only connect to and access the file system in sequential order. One client will finish the actions and disconnect before another client establishes her/his connection and performs actions.

To support the multi-user feature, we first need to design a data structure to store the user information, which should be stored in the file system, instead of memory. Then you need to provide approaches and interfaces to log into your file system, create new users, delete users, etc.

Each user should have her/his home folder, own files, and so on. Regarding the file access control, the users can share some files with other users, or make files only visible to themselves. You need to design a file access protocol to control file read/write permissions.

Step 4: Optional extensions

In this step, we provide some optional extra functions to the basic file system. Implementing any of the following functions will help you earn bonus points. Besides, you are encouraged to implement any other additional functions/optimizations that are useful to your file system. Please clearly indicate the optional functions you have implemented in your file system, and demonstrate your extensions through empirical performance evaluation or qualitative analysis.

Extension 1 - Caching: Reading files can be expensive, incurring many I/Os to the (slow) disk. Imagine an open example: without caching, every file open would require at least two reads for every level in the directory hierarchy (one to read the inode of the directory in question, and at least one to read its data). With a long pathname (e.g., /1/2/3/ ... /100/file.txt), the file system would literally perform hundreds of reads to just open the file. To remedy what would be a huge performance issue, you can use a region in the system memory to cache frequently accessed or important blocks. Similar to the virtual memory mechanism, strategies such as LRU and different algorithms would decide which blocks to keep in the cache. Implement a cache in the memory, select your page replacement algorithm

for the cache, and share your design and performance evaluation (e.g., how many times you have accelerated your file read operations?)

Extension 2 - Multi-clients: You have supported multi-users in step 3 which mainly focuses on separating the user folders/files and user account management. Another thing you can do is to support more than one user to use the file system at the same time. That means your file system server should support parallel requests from multiple clients. Different from the shell problem in Project1, here we have more challenges when more than one user logs in to your file system. For example, user A wants to write a file, while user B wants to read the same file. There is a read/write conflict. You can come up with a solution to solve this conflict. There are still some other problems like this. It's really difficult to handle all such synchronization situations, so we encourage you to try your best to consider as many issues as possible and explain your solutions in the report.

Extension 3 - Journaling: Consider the crash-consistency problem, that is, how to update persistent data structures despite the presence of a system crash. The system may crash or accidentally power off between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again. Given that crashes can occur at arbitrary points through time, how do we ensure the file system keeps the on-disk image in a reasonable state? Many modern file systems use the idea of journaling (also known as write-ahead logging), where a sequence of updates is treated as a single indivisible unit (called transaction), ensuring that either all of the updates are committed to the disk, or none of them are. Try to implement a journaling and transactions mechanism in your file system.

3 Implementation Details

In general, the execution of the programs above is carried out by specifying the executable program name followed by the command line arguments.

- Use Ubuntu Linux and GNU C Compiler to compile and debug your programs. GCC-inline-assembly is allowed.
- See the man pages for more details about specific system or library calls and commands: `sleep(3)`, `gcc(1)`, `mmap(2)`, `usleep(3)`, `nanosleep(3)`, etc.
- When using system or library calls you have to make sure that your program will exit gracefully when the requested call cannot be carried out.
- One of the risks of experimenting with forking processes is leaving unwanted processes hanging which wastes system resources. Please make sure that each process/thread is terminated cleanly when the program exits. A parent process should wait until all its child processes finish, print a message, and then quit.
- Your program should be robust. If any of the calls fail, it should print an error message and exit with an appropriate error code. Please always check for errors when invoking a system or library call.
- Please ensure that your program can work correctly with the tests we provided and exit properly.

4 Material to be submitted

- Create a folder named `Prj3_studentID` that includes your source codes, Makefile, and report. Use meaningful names for the files so their contents are recognizable. Then compress the folder into a file named `Prj3_studentID.tar`, ensuring that only the folder itself is included in the tar file.
- You are not required to submit the executables. Make sure your programs can be compiled with the Makefile before the submission.
- Enclose a README file that lists the files you have submitted along with a one-sentence explanation. Call it `README.md`.
- Please state clearly the purpose of each program at the start of the program file. Add comments to explain your program.
- Test runs: You must show that your program works for all possible inputs. Submit online a single typescript file clearly showing the working of all the programs for correct input as well as a graceful exit on error input. Call it `test.md` or `test.pdf`.
- Submit a short report to present the analysis. You can also write down important methods and observations you find useful when finishing your project. Call it `report.pdf`.
- We provide a code template for this project. Here's how the file structure looks (you can feel free to add other files you need):

```
Prj3_52202191xxxx
├── Makefile
├── README.md
├── report.pdf
├── test.md
├── step1
│   ├── BDS.c
│   ├── BDC.c
│   └── Makefile
├── step2
│   ├── FS.c
│   ├── FC.c
│   └── Makefile
├── step3
│   ├── Makefile
│   └── ...
└── step4
    ├── Makefile
    └── ...
```

- Submit your `Prj3_StudentID.tar` file on Canvas.
- Due date: **23:59 on May. 31, 2024**.
- Demo slots: Around **June 1, 2024**. Demo slots will be posted later on a shared document. Please sign up for one of the available slots.

- You are encouraged to present your design of the project optionally. The presentation time is **16:00-17:40, June 4, 2024 (Course meet time of the 16th week)**. Please pay attention to the course website.