CPSC 3500 Computing Systems

Project 4: A Simple Network File System (NFS)

Assigned: 2:05PM, Friday, 03/01/2019

Due: 11:59PM, Sunday, 03/10/2019

Note: you can form a max group size of 3 for this project. I do not encourage you to take it as an individual project. Only one submission is required for a group.

1. Description

In this project, you will be implementing a simple client-server NFS over a simulated disk. The server is not a multi-client program.

Download the source code package from the Canvas. A Makefile is provided and feel free to modify it when necessary. A README template file is also provided. You should complete the README as specified later.

In this project, the file system is built on top of a virtual disk that is simulated using a file. In this virtual disk, there are 1,024 disk blocks (numbered from 0 to 1023) and each block is 128 bytes.

The provided code implements a layered architecture as follows:

Client-Side Shell (**Shell.cpp** and **Shell.h**)	Processes the network file system commands from the command line. • In mountNFS(): It creates a TCP socket cs_sock and connects it to the NFS server when the NFS is being mounted • In xxx_rpc(): It sends a NFS command to the server, receives a response from the server, and displays the message. • In unmountNFS(): It closes the TCP connection if the NFS is mounted. Client.cpp uses the above APIs. (You should implement your client-side code here.)
Server-Side File System (**FileSys.cpp** and **FileSys.h**)	Provides an interface for NFS commands received from the client via the TCP socket and sends back the responses to the client via the TCP socket. • Data member - fs_sock: the TCP socket used for communication between the NFS server and the client. It is initialized in its mount(). Server.cpp uses the APIs implemented here. (You should implement your server-side FS code here.)

Basic File System (**BasicFileSys.cpp** and **BasicFileSys.h**)	A low-level interface that interacts with the disk. (Do not attempt to modify them!)
Disk (**Disk.cpp** and **Disk.h**)	Represents a "virtual" disk that is contained within a file. (Do not attempt to modify them!)

Each of the four layers is implemented using a class. Except the client-side shell, each class contains ("has-a") a single instance of the lower layer. For instance, the file system class has an instance of the basic file system.

The following two main programs are for the NFS client and server, respectively.

client.cpp	NFS client main program. (Do not attempt to modify them!)
server.cpp	NFS server main program. • Creates a listening socket to accept a TCP connection from a client • By accepting a TCP connection request from a client, saves the new socket to sock. • After mounting the NFS, you uses a loop to receive NFS commands from a client and call the corresponding FS operation in fs which in turn executes the command and sends the response message back to the client (You implemented in the server-side file system). (You should implement the code as specified above.)

Your task is to implement the following file system commands in the File System layer. *It is recommended to implement these functions in the order they appear*:

mkdir <directory>	Creates an empty subdirectory in the current directory.
ls	List the contents of the current directory. Directories should have a '/' suffix such as 'myDir/'. Files do not have a suffix.
cd <directory>	Change to specified directory. The directory must be a subdirectory in the current directory. No paths or ".." are allowed.
home	Switch to the home directory.
rmdir <directory>	Removes a subdirectory. The subdirectory must be empty.
create <filename>	Creates an empty file of the filename in the current directory. An empty file consists of an inode and no data blocks.

append <filename> <data>	Appends the data to the file. Data should be appended in a manner to first fill the last data block as much as possible and then allocating new block(s) ONLY if more space is needed. More information about the format of data files is described later.
stat <name>	Displays stats for the given file or directory. The precise format is described later in the document.
cat <filename>	Display the contents of the file to the screen. Print a newline when completed.
head <filename> <n>	Display the first N bytes of the file to the screen. Print a newline when completed. (If N >= file size, print the whole file just as with the cat command.)
rm <filename>	Remove a file from the directory, reclaim all of its blocks including its inode. Cannot remove directories.

The above list shows the shell commands. You will be implementing these commands in the server-side file system (**FileSys.cpp**). Each command has a corresponding function with the same name. Commands that require a file name or directory name have a name parameter. The append function also has a second parameter for the data, and head has a second parameter for the number of bytes to print.

The only files at the server side that requires modification are FileSys.cpp, FileSys.h and server.cpp. In FileSys.h, you are only allowed to modify the private section of the class. You can add extra data members and private member functions when necessary. In server.cpp, you should create the TCP socket and bind it to the provided port. The server listens for client TCP connections and upon a request accepts the connection with a new socket created. The new socket should be provided to mount the file system before serving any client requests. Then the server should repeat this process until the client terminates the connection: receives the FS command from the client and invokes the corresponding FS operation you have implemented in FileSys.cpp.

The only files at the client side that requires modification are Shell.cpp and Shell.h. Specifically, you need to implement the following member functions:

mountNFS(string fs_loc)	The parameter fs_loc is in the form of server:port. It represents the server name and port number the NFS server is running on. Here, you should create the socket cs_sock and connect it to the server process running on that port. If successful, you should set is_mounted to be true.
unmountNFS()	Simply close the socket if the NFS is mounted successfully.
mkdir_rpc	Send the mkdir command to NFS server, receive the response and display the message.
ls_rpc	Send the ls command to NFS server, receive the response and display the message.

cd_rpc	Send the cd command to NFS server, receive the response and display the message.
home_rpc	Send the home command to NFS server, receive the response and display the message.
rmdir_rpc	Send the rmdir command to NFS server, receive the response and display the message.
create_rpc	Send the create command to NFS server, receive the response and display the message.
append_rpc	Send the append command to NFS server, receive the response and display the message.
stat_rpc	Send the stat command to NFS server, receive the response and display the message.
cat_rpc	Send the cat command to NFS server, receive the response and display the message.
head_rpc	Send the head command to NFS server, receive the response and display the message.
rm_rpc	Send the rm command to NFS server, receive the response and display the message.

Basic File System Interface Routines

To implement the server-side file system operations, you will need to utilize routines provided by the basic file system. The file system class contains a basic file system interface, specifically private data member **bfs**. Here is a description of the provided routines you will need to use:

```
// Gets a free block from the disk.
short get_free_block();

// Reclaims block making it available for future use.
void reclaim_block(short block_num);

// Reads block from disk. Output parameter block points to new block buffer.
void read_block(short block_num, void *block);

// Writes block to disk. Input block points to block buffer to write.
void write_block(short block_num, void *block);
```

Note that basic file system also provides code for mounting (initializing) and unmounting (cleaning up) the basic file system. The basic file system is already mounted and unmounted in the provided code so there is no need to use the **mount** and **unmount** functions in your code.

File System Blocks

There are two types of files: data files (that store a sequence of characters) and directories. Data files consist of an inode and zero or more data blocks. Directories consist of a single directory block that stores the contents of the directory.

There are four types of blocks used in the file system:

- *Superblock:* There is only one superblock on the disk and that is always block 0. It contains a bitmap on what disk blocks are free. (The superblock is used by the Basic File System to implement get_free_block() and reclaim_block() - you shouldn't have to touch it, but be careful not to corrupt it by writing to it by mistake.)

- *Directories:* Represents a directory. The first field is a magic number which is used to distinguish between directories and inodes. The second field stores the number of files located in the directory. The remaining space is used to store the file entries. Each entry consists of a name and a block number (the directory block for directories and the inode block for data files). Unused entries are indicated by having a block number of 0. Block 1 always contains the directory for the "home" directory.

- *Inodes:* Represents an index block for a data file. In this assignment, only direct index pointers are used. The first field is a magic number which is used to distinguish between directories and inodes. The second field is the size of the file (in bytes). The remaining space consists of an array of indices to data blocks of the file. Use 0 to represent unused pointer entries (note that files cannot access the superblock).

- *Data blocks:* Blocks currently used to store data in files.

The different blocks are defined using these structures defined in **Blocks.h**. These structures are all **BLOCK_SIZE** (128) bytes.

```
// Superblock - keeps track of which blocks are used in the filesystem.
// Block 0 is the only super block in the system.
struct superblock_t {
  unsigned char bitmap[BLOCK_SIZE]; // bitmap of free blocks
};

// Directory block - represents a directory
struct dirblock_t {
  unsigned int magic;           // magic number, must be DIR_MAGIC_NUM
  unsigned int num_entries;     // number of files in directory
  struct {
    char name[MAX_FNAME_SIZE + 1]; // file name (extra space for null)
    short block_num;               // block number of file (0 - unused)
  } dir_entries[MAX_DIR_ENTRIES];  // list of directory entries
};
```

```
// Inode - index node for a data file
struct inode_t {
  unsigned int magic;            // magic number, must be INODE_MAGIC_NUM
  unsigned int size;             // file size in bytes
  short blocks[MAX_DATA_BLOCKS]; // array of direct indices to data blocks
};

// Data block - stores data for a data file
struct datablock_t {
  char data[BLOCK_SIZE];        // data (BLOCK_SIZE bytes)
};
```

You will use the basic file system interface routines to read and write these blocks. For instance, to read the home directory (block 1):

```
struct dirblock_t dirblock;
bfs.read_block(1, (void *) &dirblock);
```

In some cases, you will not know whether the block is a directory or an inode. To address this issue, both the directory and inode contain a magic number located as the first field (same spot in memory). For an unknown block, read the block as a directory block (or an inode block – it doesn't matter). Then read the magic number. If the magic number is DIR_MAGIC_NUM, then it is a directory. If the magic number is INODE_MAGIC_NUM, then it is an inode.

TIP*:* Create a private method *is_directory* than returns true if the block corresponds to a directory and false otherwise.

Since the blocks are a fixed size, there are limits on the size of files, size of file names, and the number of entries in a directory. These constants, along with other file system parameters, are also defined in **Blocks.h**.

Data File Format

Here are the rules concerning data files:

- Data files consists of a single index block and zero or more data blocks. (An empty file uses 0 data blocks.)

- The command create creates an empty file. This creates an inode but no data blocks as the file is empty.
- The data string passed into **append** is null terminated, you can use **strlen** to determine its size.
- When appending data using **append**, *do not add a null termination character*. If appending "ABC" to the file, exactly three characters are appended. Do not store null characters in the file; instead, use the size data member to determine the end of the file.
- When appending data, you need to add characters where you left off. If there is room in the last block, that block needs to be filled before adding a new block. If the data to append does completely fit in the last block, completely fill in the last block first, then create a new data block for the remainder. (No file should ever use a data block that is empty - if the append fits exactly into the last block, then fill the last block and do *not* allocate a new block.)
- There is no limit* on the size of the data to append so it may be necessary to create two or more data blocks with a single call to append. (* You will have to check for situations where the append would exceed the maximum file size, however.)
- Only create a new block when it is absolutely necessary to create one. For instance, a file consisting of exactly 128 bytes should have only one (completely full) data block.
- Since the data is not null terminated, it is recommended that **append** copies characters one at a time and that **cat** displays characters one at a time. You may be tempted to use C string functions (such as **strcpy**) or using << on the entire block but they rely on a null termination character being present. C++ strings (std::string) aren't appropriate or necessary either.
- Due to the nature of the shell and the limited commands available, it is impossible to append special characters to a file including '**\0**', '**\n**', and a space.

Stat Format

For directories, print out the directory name and directory block number.

```
Directory name: foo/
Directory block: 7
```

For data files, print out the inode block, number of bytes stored in the file, and the number of blocks the file consumes (including the inode), and the block number of the first data block in the file (or print 0 for the block number if the file is empty and has no data blocks).

```
Inode block: 5
Bytes in file: 170
Number of blocks: 3
First block: 2
```

Empty files store 0 bytes and take up 1 block (inode). Non-empty files take up at least 2 blocks (one inode block and at least one data block).

Running the Program

You should start your server program first by providing the port #:

```
./nfsserver  <port>
```

Then, you run the client program:

```
./nfsclient  <server_name:port>
```

The client program will run indefinitely until the user enters '**quit**' for a command.

In the server program, the disk will be mounted at the beginning of the program. The disk is stored in the filename "DISK". If the disk file exists, it will use those disk contents. If the disk file does not exist, it will create a new disk file and properly initialize block 0 (superblock) and 1 (home directory). The current directory is always the home directory at the start of the program.

The disk is persistent across different runs of the program. This can be helpful for testing, since you don't have to run all your commands in one go. However, in some cases, you may want to start with a fresh disk; simply remove (delete) the file DISK. The Makefile will also automatically remove the disk when you recompile.

Implementation Notes

- The size of the block data structures are 128 bytes on cs1 and should be 128 bytes on many other systems. If you are concerned with portability problems, uncomment the sanity check code at the beginning of **main** to double check.
- Unlike data files, the names of files and subdirectories stored in directories are null terminated.
- Neither **get_free_block** nor **reclaim_block** initializes or clears out the corresponding block in any way. Your implementation should not rely on blocks being "empty".
- Be sure that **rmdir** and **rm** actually reclaim blocks that are part of the deleted directory or file. This can be tested for by removing a file, creating a new file, and running stat on that new file to see if the block is indeed reused. This test is possible since **get_free_block** deterministically returns the free block with the lowest number.

Error Checking

The server program must return the following error codes and messages when appropriate:

```
Code Message
500 File is not a directory          (Applies to: cd, rmdir)
501 File is a directory              (Applies to: cat, head, append, rm)
502 File exists                      (Applies to: create, mkdir)
503 File does not exist              (Applies to: cd, rmdir, cat, head, append, rm, stat)
504 File name is too long            (Applies to: create, mkdir)
505 Disk is full                     (Applies to: create, mkdir, append)
506 Directory is full                (Applies to: create, mkdir)
```

```
507 Directory is not empty            (Applies to: rmdir)
508 Append exceeds maximum file size  (Applies to: append)
```

After sending back the error code and error message to the client, return from the function. Do NOT exit the program. (Calling exit() or abort() in this assignment is not allowed! After all, you wouldn't want your OS to blue screen / kernel panic just because of a file system error.) In addition, all errors should be detected before any writes are made to the disk. In an error condition, the disk should not be modified - operations should either complete fully and successfully, or not modify the disk at all - partially completing operations is a serious bug.

Message Format

We use the following message formats for the communication between the NFS client and server. You are NOT allowed to use other formats. By complying with this message format, different teams' client and server can work with each other. You may work with other teams to test the client and server, but only use other's executables instead of source code!

NFS Client to NFS Server Request Message Format

The message sent to the NFS server is in text format with one single command line ending with "\r\n".
For example, for the command: ls
The message should be: ls\r\n

For the command: mkdir dirone
The message should be: mkdir dirone\r\n

NFS Server to NFS Client Response Message Format

The message sent to the NFS client is in text format with two header lines ending with "\r\n", one blank line with "\r\n", and optionally a message body if the length is not zero.
The first header line is response status line: Status_code Status_message\r\n
The second header line is message body length line: Length:size_in_bytes\r\n.

For example, for a command: cat myfile
If the file exists, then the response message should be:

```
200 OK\r\n
Length:18\r\n
\r\n
 I am a CS student!
```

If the file does not exist, then the response message should be:

```
503 File does not exist\r\n
Length:0\r\n
\r\n
```

2. Submitting your Program

For a group project, only one submission is required.

Before submission, you must make sure that your code can be compiled and run on Linux server cs1. You must run the submission command on cs1. You must make sure that your Makefile works properly!

The following files must be included in your submission:

- README
- *.h: all .h files
- *cpp: all .cpp files
- Makefile

You should create a package p4.tar including the required files as specified above, by running the command:

```
tar  -cvf  p4.tar  README  *.h  *.cpp  Makefile
```

Then, use the following command to submit p4.tar:

```
/home/fac/zhuy/class/SubmitHW/submit3500  p4  p4.tar
```

If submission succeeds, you will see the message similar to the following one on your screen:

```
=====================Copyright(C)Yingwu Zhu========================
Wed Jan 17 21:53:58 PST 2018
Welcome testzhuy!
You are submitting array.cpp for assignment p4.
Transferring file.....
Congrats! You have successfully submitted your assignment! Thank you!
Email: zhuy@seattleu.edu
===================================================================
```

You can submit your assignment multiple times before the deadline. Only the most recent copy is stored.

The assignment submission will shut down automatically once the deadline passes. You need to contact me for instructions on your late submission. Do not email me your submission!

3. Implementation Suggestions

You can split the project into two major parts:

(1) Network communication between the client and file system server.
(2) File system operations at the server side: the File System Layer's functions.

These two parts can be implemented independently and simultaneously. For a group, some can work on network communication to make sure the well-formatted messages can be sent/received between the client and server; others can work on the file system server side (the functions of the File System Layer as you can always assume that the parameters of the file system functions are received, thus you do not have to wait for the network communication code to complete). For a group, I highly encourage all team members are exposed to and involved in the two parts throughout the project!

Again, always "Design goes first!"

Implement and test one function at a time!

4. Grading Criteria

Label	**Notes**
4a. Compilability, Complete submission, Coding Format & Style (1 pt)	- Your Makefile works properly. - All required files are included in your submission. - Clean, well-commented code.
4b. README file (2 pts)	README file follows the specified requirements, addressing the following: • Team member's names and respective contributions (-0.5 pt if none) • For each file system command, you should run at least a test, and show the test results. (-1 pt if none) • Your own rating on the functionality in 4c: A/B/C/D/F? and explanation! (-0.5 pt if none)
4c. Functionality (15 pts)	All file system operations are implemented correctly and behave as specified. The messages follow the required format. The appropriate messages are displayed. Using the following category to rate your functionality: - A: 14-15 pts - B: 12-13 pts - C: 10-11 pts - D: 8-9 pts - F: 0-7 pts
4d. Resource management, No messy & debugging messages (2 pts)	-No memory leaks (if applicable). -sockets are closed (-0.5 pt if none) - byte orders are properly handled (-0.5pt if none) - proper socket reads and writes (-0.5pt if none) -No messy & debugging & testing messages (-0.5pt if none)
4e. Overriding policy	If the code cannot be compiled or executed (segmentation faults, for instance), it results in zero point. If the submission is incomplete (e.g., missing files), it results in zero point. If you modify the files that you are NOT allowed to revise, it results in zero point!
4f.	Please refer to the late submission policy on Syllabus.

Late submission	

5. Test cases

You may run the following test cases to check if your code works as expected.

Test case	Display message
ls	empty folder
mkdir dir1	success
mkdir dir2	success
ls	dir1 dir2
cd dir1	success
create file1	success
append file1 helloworld!	success
stat file1	Inode block: xx Bytes in file: 11 Number of blocks: xx First block: xx
ls	file1
cat file1	helloworld!
head file1 5	hello
rm file2	503 File does not exist
cat file2	503 File does not exist
create file1	502 File exists
create file2	success
rm file1	success
ls	file2
home	success
ls	dir1 dir2
stat dir1	Directory name: dir1 Directory block: xx

rmdir dir3	503 File does not exist
rmdir dir1	507 Directory is not empty
rmdir dir2	success
ls	dir1