CPSC 3500 Computing Systems

Assignment #5: A Simple Network File System (NFS)

20 Points

Note: Students may complete this as either an individual or group project. For group projects (maximum 3 members), only one team member needs to submit the work.

Table of Contents

1. Description

1.1 Overview

For this project, your task is to develop a simple client-server Network File System (NFS) that operates on a simulated disk. The server program is designed to handle a single client at a time, meaning that it services only one client over a TCP session throughout its lifetime. Although not a requirement for this project, you may choose to enhance the server's functionality by incorporating either multiprocessing or multithreading to enable it to support multiple clients. However, this is beyond the scope of this project.

You can obtain the source code package for this project from Canvas. The package includes a Makefile that you can adjust as necessary. Additionally, there is a template README file provided, which you should complete according to the instructions detailed later in this document.

For this project, the NFS is constructed on a simulated virtual disk, which is represented by a file. The virtual disk comprises 1,024 disk blocks, numbered from 0 to 1023, with each block having a size of 128 bytes.

The implemented code follows a layered architecture, which is illustrated in Table 1 below.

Table 1

Client-Side Shell (**Shell.cpp** and **Shell.h**)	Processes the network file system commands from the command line at the client side. • 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 a 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	A low-level interface that interacts with the disk. (**DO NOT attempt to modify them!**)

(**BasicFileSys.cpp** and **BasicFileSys.h**)	
Disk (**Disk.cpp** and **Disk.h**)	Represents a "virtual" disk that is contained within a file. (**DO NOT attempt to modify them!**)

The four layers of the NFS system are implemented through individual classes, with the exception of the client-side shell. Each class "has-a" single instance of the layer beneath it. For example, the `FileSys` class, responsible for the file system layer, has an instance of the basic file system, `BasicFileSys`.

There are two primary programs for this project: one for the NFS client and the other for the NFS server.

Table 2

client.cpp	NFS client main program. (**Do not attempt to modify them!**)
server.cpp	NFS server main program. • A listening socket is created to receive a TCP connection from a client. • Upon receiving a connection request, the new socket is saved to a variable called **`sock`**. • After mounting the NFS, a loop is initiated to receive NFS commands from the client. The corresponding file system operation is called using the **`fs`** object provided by the `FileSys` file system. This operation is implemented in the `FileSys` file system and executed, and the response message is sent back to the client. (**You should implement the code as specified above.**)

In the File System layer, `FileSys`, your objective is to develop the file system commands listed in Table 3. It is suggested that you implement these functions in the order specified in the table.

Table 3

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 details the shell commands that you will need to implement in the server-side file system (`FileSys.cpp`). Each command corresponds to a function with the same name. Commands requiring a file or directory name include a name parameter, while the append function has an additional parameter for data and the head function has a second parameter for the number of bytes to print.

The only files on the server side that require modification are `FileSys.cpp`, `FileSys.h`, and `server.cpp`. In `FileSys.h`, modifications are only permitted within the private section of the class. Additional private member functions and data members may be added as necessary. In `server.cpp`, you must establish a TCP socket and bind it to the specified port (from the command-line argument). The server will listen for a client TCP connection and accept the connection with a new socket (sock) upon request. This new socket is required to mount the file system before any client requests can be served.

Subsequently, the server should continue to perform this process until the client terminates the connection: receiving an FS command from the client and invoking the corresponding FS operation that you implemented in `FileSys.cpp`.

Only two files on the client side need modification: `Shell.cpp` and `Shell.h`. Specifically, you must implement the member functions illustrated in Table 4 below:

Table 4

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.

1.2 Basic File System Interface Routines

In order to implement the file system operations on the server side, you will need to utilize routines that are provided by the basic file system. The file system class comprises a basic file system interface, which is represented by a private data member, **bfs**. Below is a description of the routines that have been provided, and which you will need to utilize:

```
// 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);
```

Please note that the basic file system also includes code for mounting (initializing) and unmounting (cleaning up) the basic file system. The basic file system has already been mounted and unmounted in the provided code, so there is no requirement to utilize the mount and unmount functions in your code.

1.3 File System Blocks

There are two categories of files in this project: data files, which store a sequence of characters, and directories. Data files comprise an inode and zero or more data blocks. Directories, on the other hand, consist of a single directory block that stores the contents of the directory, which differs from what was discussed in class.

There are four types of blocks used in the file system, each with a block size of 128 bytes:

- *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. Many students made this mistake unfortunately in the past.)

- *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);
```

At times, you may encounter a situation where it is uncertain whether a block is an inode or a directory. To tackle this issue, both the inode and directory have a magic number located as the first field in memory. When you encounter an unknown block, you should read it as a directory block (or an inode block - either will do). After that, read the magic number. If the magic number is DIR_MAGIC_NUM, then it is a directory, whereas 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.

Given that the size of the blocks is fixed, there are restrictions on the size of files, file names, and the number of entries in a directory. These limitations, as well as other file system parameters, are defined in `Blocks.h` as constants.

1.3 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.) This indicates that indexed allocation is used for files.
- 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** copy 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.

1.4 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).

1.5 Running the Program

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

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

Be reminded that ports **10101 to 10300** are used for this class. In order to avoid conflicts of use, you may calculate your port number in this range:

```
[10101 + (your-SU-ID)% 31 * 5, 10101 + (your-SU-ID)%31*5 + 4]
```

Then, you run the client program in a different terminal:

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

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

At the beginning of the server program, the disk is mounted. The contents of the disk are stored in a file named "DISK". If the file exists, the program uses the disk contents. If the file does not exist, a new disk file is created and block 0 (superblock) and block 1 (home directory) are initialized properly. The program starts with the current directory set to the home directory.

The disk contents persist across different runs of the program, which can be useful for testing purposes. However, if a fresh disk is needed, the DISK file can be removed (deleted). The Makefile will automatically remove the disk when recompiling.

1.6 Implementation Notes

- The size of the block data structures are 128 bytes on cs1 and should be 128 bytes on many other systems.
- Unlike data files, the names of files and subdirectories stored in directories are null terminated (C string instead of C++ string).
- 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.

1.7 Return Codes and Messages

The server program must return the following 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**)
200	*OK*	(Applied to: all operations)

When an error is encountered in the server program, the following steps should be followed:

(1) Send back an appropriate error code and error message to the client.
(2) Return from the function, without exiting the program. The use of exit() or abort() is not allowed.
(3) All errors should be detected before any writes are made to the disk. If an error is detected, the disk should not be modified.
(4) Partially completed operations should not modify the disk at all.
(5) If the command is executed successfully, return code 200 with message OK.

1.8 Message Format

The communication between the NFS client and server follows the message formats given below. It is strictly prohibited to use any other formats. Adhering to this message format will ensure that clients and servers from different teams can work together. Teams can collaborate with each other to test their clients and servers, but only using the executables and not the source code. The message formats are as follows:

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 a message body.
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. Summary

This assignment involves creating a client-server network file system using a virtual disk. The server does not need to handle multiple clients concurrently, meaning that it does not need to be a multithreading or multiprocessing program. The network file system must support the list of commands specified in Table 3. The client program sends a command to the server via its socket; the server receives the command, executes the corresponding local file system operation, and sends back a response. The return code and message must follow the specifications outlined in Section 1.7. Additionally, the message format between the client and server must conform to the format described in Section 1.8. To ensure interoperability between different teams' clients and servers, all teams must adhere to the protocols described in Sections 1.7 and 1.8.

To implement the network file system, teams must modify the provided files. The only files on the server side that require modification are FileSys.cpp, FileSys.h, and server.cpp. In FileSys.h, teams are only allowed to modify the private section of the class. They can add extra data members and private member functions when needed. In server.cpp, teams must create a TCP socket and bind it to the provided port. The server listens for a client TCP connection and upon a TCP connection request, accepts the connection with a new socket, which is then used to mount the file system before serving any client requests. The server should repeat this process until the client terminates the connection: it receives an FS command from the client and invokes the corresponding FS operation implemented in FileSys.cpp. On the client side, the only files that require modification are Shell.cpp and Shell.h. Specifically, teams need to implement the member functions described in Table 1. No other files should be modified.

Teams must include a README file in their submission, which should contain a description of each team member's names and respective contributions (if it is a group project), at least one test case for each file system command, and the test results. The test cases and results should serve as proof of testing and should be included in the README file. In addition, teams must rate their own project (A/B/C/D/F as specified in Section 4) and provide a brief explanation of their rating. The running of the programs must follow the specifications outlined in Section 1.5.

3. Methodology

The project consists of two main components:
(1) File system operations on the server side, which involve functions in the File System Layer.
(2) Network communication between the client and server.

These two components can be implemented independently and simultaneously. To begin, you can focus on implementing the File System operations without involving client-server communication. You can make a copy of server.cpp and modify it to simply receive commands from the command line (using some code from Shell.cpp) and execute the corresponding FS operations. It is recommended to implement one file system command at a time (following the order shown in Table 3) and perform unit testing for each command.

Once you are satisfied with your File System implementation, you can add the client-server communication code to the original server.cpp and augment each FS operation in FileSys.cpp by sending the response message back to the client after the requested FS operation is completed (whether it was successful or not). For a group project, team members can divide tasks such that some members work on client-server communication to ensure that well-formatted messages can be sent and received between the client and server. Meanwhile, others can work on the file system server side to implement FS operations. It is

recommended that all team members are involved in the development of both components throughout the project.

You may follow the roadmap as described below to meet the deadline:

- Phase #1: read the document and the provided code to understand the project
- Phase #2: work on the file system implementation & testing
- Phase #3: work on the network communication & testing
- Phase #4: work on integration testing and test against other team's client & server
  Each team can announce your server's port # for other teams to test their clients & servers.
- Phase #5: submission

It is important to prioritize design before implementation. Break down the project into smaller components and implement and test each one separately. This ensures that each function works as expected and reduces the likelihood of errors that could cause the entire system to fail. Remember to focus on one function at a time and test it thoroughly before moving on to the next one.

4. Grading Criteria

Label	**Notes**
a. Submission. (1 pt)	- Your Makefile works properly. - All required files are included in your submission.
b. README file (2 pts)	README file meets the 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)
c. 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 project: - A: 14-15 pts - B: 12-13 pts - C: 10-11 pts - D: 8-9 pts - F: 0-7 pts
d. Others (2 pts)	-No obvious memory leaks (if applicable) (-0.5 pt with memory leak). -sockets are closed (-0.5 pt if none) -proper socket reads and writes (-0.5pt if none) -No unsolicited messages (-0.5pt if none)
e. 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!
f. Late submission	Please refer to the late submission policy on Syllabus.

5. Test Cases

You may run the following test cases to check if your code works as expected. Of course, you are encouraged to try more test cases.

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

6. Submitting your Program

Prior to submitting your work, ensure that your code has been successfully compiled and run on **`cs1.seattleu.edu`**, and that your submission includes all necessary files.

The following files must be included in your submission:

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

You should create a package p3.tar including the required files as specified above, by running the command:

```
tar  -cvf   p3.tar  README  *.h  *.cpp  Makefile
```

Then, use the following command to submit `p3.tar` if you are in class `CPSC 3500-01`:

```
/home/fac/zhuy/class/submit3500_01  PA3  p3.tar
```

But, if you are in class `CPSC 3500-02`, use this one instead:

```
/home/fac/zhuy/class/submit3500_02  PA3  p3.tar
```

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

```
======================Copyright(C)Yingwu Zhu=========================
Mon Jan 13 12:43:34 PST 2020
Welcome testzhuy!
You are submitting hello.cpp for assignment p2.
Transferring file.....
Congrats! You have successfully submitted your assigment! 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!