CS 214: Systems Programming, Spring 2018
Assignment 3: A Better Open

Abstract:
Any time you ask for a resource and you get back a reference to that resource, rather than the resource itself, it is likely that resource is abstracted or virtualized. In this project you will do something similar to your malloc project. Rather than replacing malloc() and free(), you'll write a separate set the file commands that will contact a file server to perform read() and write() commands on it rather than talking to the local hard drive.

Introduction:
This project is fairly straightforward in concept. Think about what open() does from your point of view as a user. It either tells you that the file can not be opened and sets errno, or gives you a handle to access it; a file descriptor. In your case your netopen() will ask another machine if the file can be opened by passing its name and access mode to the remote server and listening for a response. Whatever the remote file server says is the state of the file, your netopen() command can report back. In order to figure out the state of the file all the remote file server needs to do is try the open() the file and report back the results: either a file descriptor or -1 and an error number.
So all you need to do then is to have the “net” version of your file commands bundle up the parameters and command they are given, send them to the remote file server and report the result. All your file server needs to do is to listen for communication, get a command with parameters, try to run it and the report back the results.
In order to do this, you need to do some simple network communication, so will will need to learn about sockets, threads and synchronization between threads.

Methodology:
You must first complete the 'base program' segment below, but you are then free to choose which other extensions you implement, with one proviso; you can not implement extension D unless you also implement extension C. If you complete all parts, you will receive 160% credit.

Base Program: (+80%)
You will be providing an interface much like the standard file system calls to allow easy use of files across the network. You should write 'netopen', 'netread', 'netwrite' and 'netclose'. All of these calls should use the same syntax and have the same overall functionality as their local counterparts (except where expressly exempted), but they will ship their parameters your file server where the actual file operations will happen. To your client code, it will look like open and netopen, read and netread, write and netwrite and close and netclose work almost identically, except your net commands are working on files on another machine.

netopen(const char *pathname, int flags)

The argument *flags* must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file in read-only, write-only, or read/write modes, respectively.

RETURN VALUE
netopen() returns the new file descriptor, or -1 in the caller's context if an error occurred (in which case, errno should be set appropriately). In order to avoid error and disambiguate your file descriptors from the system's, make your file descriptors negative (but not -1!).

ERRORS (check open manpage for definition)
reqiured:
- EACCES
- EINTR
- EISDIR
- ENOENT
- EROFS

optional (you may want/need)
- ENFILE
- EWOULDBLOCK
- EPERM

ssize_t netread(int fildes, void *buf, size_t nbyte)

RETURN VALUE
Upon successful completion, netread() should return a non-negative integer indicating the number of bytes actually read. Otherwise, the function should return -1 and set errno in the caller's context to indicate the error.

ERRORS (check manpage for definition)
required:
ETIMEDOUT
EBADF
ECONNRESET

ssize_t netwrite(int fildes, const void *buf, size_t nbyte)

RETURN VALUE
Upon successful completion, netwrite() should return the number of bytes actually written to the file associated with fildes. This number should never be greater than nbyte. Otherwise, -1 should be returned and errno set to indicate the error.

ERRORS
required:
EBADF
ETIMEOUT
ECONNRESET

int netclose(int fd)

RETURN VALUE
netclose() returns zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS
EBADF

netserverinit(char * hostname)
-or-
netserverinit(char * hostname, int filemode) (only if ext. A is implemented)

RETURN
0 on success, -1 on error and h_errno set correctly

ERRORS
required:
HOST_NOT_FOUND
- if you do implement extension A, be sure to #define and implement this code -
INVALID_FILE_MODE

Your file server will be a separate program you write that runs on an iLab machine other than the one your client code runs on. When a net file command in your library is run, the library function should open a connection to your file server and send it a message telling it which function was invoked along with the necessary parameters to run it. Your file server will, based on the message sent to it, run a command on its local file system and send the result back to your library code. These responses may be success, and data, or error code and errno (or h_errno) values to set. On error, your library should take the error code from your file server's response and set the same errno (or h_errno, for network errors) in the calling code's context. You are free to design the messaging protocol however you like.

You should have your file server listening for connections on a port that is greater than 8k (2^13), but less than 64k (2^16). This port can be hard coded in your library. The IP address of the iLab machine that is running your file server must be not be hard-coded, however. Instead, your library should include a function called 'netserverinit' that takes a hostname as an argument and a file connection mode and verifies that the host exists. 'netserverinit' should return -1 on failure and 0 on success. If a net file command is called when 'netserverinit' either has not been run, or has been run on a host name that does not exist, the net file command should return '-1' and set h_errno to HOST_NOT_FOUND. 'netserverinit' is also responsible for making sure your library attaches the file connection mode to every net file command a library sends.

Your file server must also assign remote clients a 'network file descriptor' so that it can tell different clients' requests apart. File descriptors are normally positive, but these are special file descriptors that should never EVER work on the local machine if the user accidentally forgot and tried to use them with regular read() and write(), so your network FDs should be negative.

Your file server only needs to support the flags: O_RDONLY, O_WRONLY and O_RDWR. Any other flag should result in an error.

Your file server should use multiple kernel threads. It should have its server socket in one thread, and on receiving a connection request, should spawn a worker thread to handle the connection and pass it the newly-constructed client connection. The new worker thread will receive the request, decode it, run the appropriate local command and send back its output.

Extension A: (+30%)
Your file server should provide three types of file connections:

0. Unrestricted mode
In unrestricted mode, your fileserver will allow any number of clients to have the same file open with any permissions. If three different clients want to have the file open in write mode, as long as they all open in unrestricted mode, your file server should allow it.

1. Exclusive mode
In exclusive mode, your fileserver will allow any number of clients to have the same file open in read mode, but only at most one to have it open in write mode. If the file can not be opened with write permission due to another client having it open for writing already, your fileserver should report that the permissions are not allowed in such a case (with an open error and appropriate errno).

2. Transaction mode
In transaction mode your file server should allow only one distinct client to have a given file open at a time, for any reason. If any other client wants to open the same file for any reason, it should fail with a lack of permission error code. If some other clients have a file open already in any other mode and another client wants to open the file in transaction mode, its open should fail with a lack of permission error code.

Be sure to create #define codes for these file modes in your .h and modify your netinit function to include a parameter for file mode.

Different clients can have the same file open in different modes simultaneously, so long as all modes allow all permissions. Client 0 may open the file in unrestricted mode, but only ask for read permission. If Client 1 wants to open the same file in exclusive mode with write permission, it should be able to do so without error, since exclusive mode is fine with multiple read connections. If Client 0 instead had the file open in unrestricted mode with read and write permission and Client 1 tried to open the file in write mode, that open request should fail since exclusive mode requires only one client have write permission at a time. If Client 1 got there first and had the file open in exclusive mode with write access and Client 0 tried to open the file with read and write permission in unrestricted mode, that open should fail. It should fail because, even though Client 0 is fine with Client 1's write access, Client 1 expects the file was opened in exclusive mode and that it alone has write access, and that should not ever change until Client 1 closes the file.

Example:

```
User / Library / Client side:                               File server/Remote side:
netserverinit("basic.cs.rutgers.edu", "exclusive");
// --- checks that hostname "basic.cs.rutgers.edu" exists...
//   - it does!, return 0, do not set errno
int netfd = netopen("/.autofs/ilab/ilab_users/deymious/test.c", O_RDWR)
//inside netopen function:
//  - make a socket
//  - connect to file server

                                                            //--- got a connection request
                                                            //--- new worker thread
                                                            //--- give new socket to worker thread

// - got a socket connected to the file server!
//--- sends: "exclusive,/.autofs/ilab/ilab_users/deymious/test.c,O_RDWR" to file server --->

                                                            // --- worker thread gets message
                                                            // --- worker thread decodes message
                                                            // --- worker thread checks global state:
                                                                    - file exists
                                                                    - file isn't open yet
                                                                    - file can be read and written to
                                                            // --- worker thread modifies global state:
                                                                    - this file is open
                                                                    - this file is in exclusive mode
                                                                    - this file has read and write accesses
                                                            // --- worker thread sends return value &
                                                                    error code
                                              <--- sends a file descriptor to library over socket---///
                                                            // --- worker thread exits

// - netopen returns the file descriptor to user and does not set errno
// - now 'netfd' equals a valid remote file descriptor
//         - presume for this example the remote file descriptor's value is -92
```

```
int bytesread = netread(netfd, &buf, 100)
//inside netread function:
//  - make a socket
//  - connect to file server
                                                        //--- got a connection request
                                                        //--- new worker thread
                                                        // –- give new socket to worker thread

// - got a socket connected to the file server!
                    //--- sends "-92,read,100" to file server --->

                                                        // --- worker thread gets message
                                                        // --- worker thread decodes message
                                                        // --- worker thread checks global state:
                                                                 - client #-92 has read/write access
                                                                 … request is OK!
                                                        // --- worker thread reads 100 byes from file
                                                        // --- worker thread packs return value and
                                                          bytes into message
             <--- sends “100, QUASIFLAPDOODLE ...” to library
                                                        // –- worker thread exits
// - copy bytes into &buf
// - returns 100 and does not set errno
```

Your file server must keep state for all clients and all files, so that global state must be synchronized across all connections (and threads).

Extension B: (+20%)

In order to provide greater bandwidth for 'large' file transfers, your file server will multiplex 'large' reads and writes over multiple connections. For these purposes, we'll consider 'large' to be any operation that requires more than 4K (2^12) bytes. For instance, doing a 8K read would result in a 2-connection multiplex, with 4K segments of the file being processed by 2 different fileserver threads and going over 2 different sockets from the file server to your client library.

To handle 'large' reads, your fileserver should send a configuration message, instead of a data response, to your library informing it how many multi-read connections will be made, which ports the connections will come in on, and which segment of the file will be sent over each port (i.e. port 9123 will get the first 4K, 9124 the second, and so on).

You shouldn't have more than 10 sockets open over all multiplexed connections. If a client asks to do a 4MB write, it will have at most 10 multiplexed sockets. If a client asks to do a 1MB write and 9 of the multiplexed sockets are in use, it still gets only one socket to send its entire 1MB write over.

Extension C: (+15%)
Your fileserver should queue up operations per file. There should be one queue of operations that is shared between all your fileserver's worker threads for each file that is being currently accessed. Any request that can not currently be serviced does not result in a error, but is reinserted at the front of the queue so that it is serviced as soon as possible.

For instance, if a client wants to open a file in transaction mode that is already open, your file server should requeue that request and the thread that is handling that request should periodically check to see if that file is closed (or it should block on the command, and be informed when the file is closed). Once the file has been closed, the transaction mode request is next in the queue, and should succeed.

Only requests that can not be serviced due to policy rules (opening in transaction mode twice, opening with write permission in exclusive mode twice, etc.) should be requeued because they might eventually succeed. If a request results in a filesystem error, for instance trying to open a file that does not exist, that request can not ever be successful, and should not be requeued. It should immediately send a message to the the library with an appropriate error code.

Extension D: (+15%)
In order to keep from holding open queued connections around forever, you should implement a monitor thread. Once every three seconds, you should have a monitor thread wake up and, for each file service queue, lock it and examine its contents. If any request has been waiting for more than 2 seconds, the monitor thread should invalidate it. The worker thread hosting the request should dequeue the it and send an appropriate error status and errno to the client library (e.g. a 'would block' errno and 'timeout' h_errno is appropriate).

Please submit a zipped tar file named Asst3_netFiles.tgz containing:
libnetfiles.h
libnetfiles.c
netfileserver.c
Makefile
testplan.txt
readme.pdf