15-440 Spring 2024
Project 1: Transparent Remote File Operations

Important Dates:

Project Handout:	Tuesday January 16, 2024
Checkpoint 1 due:	Friday January 19, 2024, 11:59 PM EST
Checkpoint 2 due:	Friday January 26, 2024, 11:59 PM EST
Final Due:	Friday February 2, 2024, 11:59 PM EST
Submission Limits:	15 Autolab submissions per checkpoint without penalty (5 additional with increasing penalty)

Important Guidance

This project will need to be done in C, in a 64-bit Linux environment, e.g., Andrew Unix server. *It will not run on Windows or Mac !!!*

You will implement a server that lets a remote client read, modify, and delete files in your file space with no security provisions! Please be careful, and do not leave your servers running.

Introduction

In distributed computing systems, it is often necessary to make use of resources located on remote machines. We can write our distributed applications to use the network to communicate between different components running on different machines. However, it becomes very tedious and inelegant to insert ad-hoc networking code every place our software needs to access remote resources.

Instead, we can use layering to hide the network complexities, and provide a clean abstraction for remote resource access. In particular, in this project we will use remote procedure calls (RPCs) to provide access to remote services, but with an interface identical to local services. We will then use this to let an unmodified program access the remote service by interposing the remote procedure calls in place of the local service routines.

In this project, you will build an RPC system to allow remote file operations (`open`, `read`, `write`, …). This will include a server process and a client stub library. To test your remote file access system, you will use existing programs (`cat, ls,` …), but interpose your RPC stubs in place of the C library functions that handle file operations. So it is critical to make your RPC abstraction look as close to local file operations as possible! At the end of this project, you should be able to execute a command like "`cat foo`" but instead of opening and printing the contents of a local file, it will access the contents of file `foo` on the remote server machine.

Requirements/Deliverables

- You will create a server process to provide the remote file services.
- You will create a client stub library to perform RPCs.
- Your code needs to handle the following standard C library calls: `open, close, read, write, lseek, stat, unlink, getdirentries`
- Your code will also need to handle the non-standard `getdirtree` and `freedirtree` calls. We will provide the header and shared library implementing the local version of these functions (see below for a description).
- You will need to write code to marshall and unmarshall parameters and return values, and to maintain any state. You will not be using an IDL or stub generator.
- You are free to design your own protocol for the messages between client and server.
- You may not use RPC packages or libraries that marshall data structures. E.g., You are not allowed to use XDR, boost serialization, etc.
- We will provide sample code for the interposition to get you started.
- We will provide simple applications to test local and RPC file operations (`440cat, 440ls, 440read, 440write, 440tree`)
- This project will use an autograder to test your code. See below on how to submit.
- You will need to write a short (1 page) document detailing your design. See below.
- The late policy will be as specified on the course website, and will apply to the checkpoints as well as the final submission
- Coding style should follow the guidelines specified on the course website

Checkpoint 1 (20%) Due: 11:59 PM, Friday January 19, 2024

Checkpoint 1 is to make sure you get a good start on the basic interposition and networking aspects of this project. For this, start with the example interposition code provided, and extend it to cover all of the required functions. This version will simply "pass through" the calls to the standard library. You will then extend this further to log all of the interposed calls to a remote server. You will write a simple TCP server to print out the names of the functions that were called to standard out. So, running (assuming bash shell syntax):

```
LD_PRELOAD=./mylib.so ../tools/440cat foo
```

should produce output at your server like this:

```
open
read
read
close
```

For checkpoint 1, **your server should not produce any other output**, as this output will be autograded. This means no debugging output, no extra white spaces or newlines, etc. If you are producing debugging output, use `stderr`, not `stdout`. Likewise, do not print anything to `stdout` from your interposition library, as it will interfere with the client program output. No actual RPCs need to be implemented for this checkpoint, and your server does not need to handle concurrent clients, etc.

Checkpoint 2 (20%) Due: 11:59 PM, Friday January 26, 2024

Extend your code to actually perform RPCs for a subset of the standard C file operations. You will need to design and implement a messaging protocol between your interposition client library and your server. You will need to marshal the data and return values. You may wish to keep or extend the logging features from checkpoint 1 for your own debugging. You must have working RPCs for the `open`, `write`, and `close` calls. Your system should be able to correctly create new files remotely on the server. Note: the autograder will check the standard output of the test programs, so your code will fail if you write to `stdout` from your interposition library.

Final (60%) Due: 11:59 PM, Friday February 2, 2024

Fully implement all of the standard C file operations as well as `getdirtree` and `freedirtree` (see description in “New Operations” on page 4). All of these must be implemented as RPCs. Your server will need to handle multiple concurrent clients, and handle clients that use more than one file at a time. Your code should report errors properly (e.g., file not found) to the client programs. (40%)

You will also need to write and submit a 1 page document, describing the serialization protocol you used to transfer data structures between server and client. Highlight any other design decisions you would like us to be aware of. Please submit this as a PDF. (10%)

Your final source code will also be graded on clarity and style. (10%)

Submission Process and Autograding

We will be using the Autolab system to evaluate your code. Please adhere to the following guidelines to make sure your code is compatible with the autograding system.

First, untar the provided project 1 handout into a private directory not readable by anyone else (e.g., ~/private in your AFS space):

```
cd ~/private; tar xvzf ~/15440-p1.tgz
```

This will create a 15440-p1 folder with example code and test tools. You should create your working directory in the 15440-p1 directory. Note: It is important that from your working directory, the provided library (`libdirtree.so`) should be available at ../lib, and the `dirtree.h` file at ../include. Do not copy or symlink these files into the working directory, or put your working directory at a greater depth (e.g., ../../lib to get to the provided files), as this may prevent your code from compiling in Autolab.

Write your code and Makefile in your working directory. You may need to add "`-I../include`" and "`-L../lib`" to the compiler and linker flags in your Makefile. Ensure that by simply running "`make`" in your working directory, both your library and server are built. Please name your library "`mylib.so`" and your server "`server`" and make sure both are in your working directory (i.e., not in a subdirectory). This naming convention and relative file locations are critical for the grading system to build and run your programs.

To hand in your code, **from within your working directory**, create a gzipped tar file that contains your Makefile and sources. E.g.,

```
tar cvzf ../mysolution.tgz Makefile mylib.c server.c
```

Of course, replace these with your actual files, and add everything you need to compile your code (other than the files we provide). If you use subdirectories and/or multiple sources, add these. Do not add any intermediate files (e.g., .o files) or binaries generated during compilation -- just the clean sources. Also, do not add the header, .so file, or binary tools that we have provided -- these will be installed automatically when grading. Your Makefile should expect the header in ../include, and the .so in ../lib. To work correctly with Autolab, when extracted, your tarball should put the Makefile and sources in the current working directory.

Then, log in to **https://autolab.andrew.cmu.edu** using your Andrew credentials. Submit your tarball (`mysolution.tgz` in the example above) to the autolab site. Note that each checkpoint shows up as a separate assessment on the Autolab course page. For your final submission, include your write up as a PDF file in your tarball.

Background: Unix file operations

To access files using the low level Unix/POSIX interfaces, you first open the file, e.g.:

```
fd = open("foo", O_READ);   // open file foo with
                            // flags indicating read only
```

flags indicates options: read-only, read-write, create, zero out (truncate) before writing, etc. The return value is a file descriptor or indicates an error. This descriptor is used in further accesses, e.g.:

```
rv = read(fd, buf, 1024);  // read up to 1024 bytes into buf
rv = write(fd, buf2, 1024);  // write 1024 bytes from buf2
```

The return value is the actual bytes read / written or an indicator of error. The system keeps track of position in the file, which can be changed with lseek. Finally, file is closed:

```
close(fd);
```

All of the operators return a negative value to indicate error. The actual error code is then available in `errno`. Please see man pages for `open(2)`, ... for more details.

New Operations

We have created a new, nonstandard library function, `getdirtree`. This function recursively descends a directory hierarchy, and constructs a tree data structure that represents the directory tree. The tree node data structure and prototype for this function are defined as:

```
struct dirtreenode {
      const char *name;     // name of the directory
      int num_subdirs;           // number of subdirectories
      struct dirtreenode **subdirs;   // pointer to array of
};          // dirtreenode pointers, one for each subdirectory

struct dirtreenode* getdirtree( char *path );
```

The `getdirtree` function allocates the `dirtreenode` structures representing the directory hierarchy rooted at path, and returns a pointer to the root of the tree. If there was an error, `NULL` is returned, and an error code placed in `errno`. We also have implemented `freedirtree` function that recursively frees the memory allocated for the tree data structures. We provide these functions as the `libdirtree.so` shared library and the prototypes in `dirtree.h`. The supplied tool, `440tree`, uses this function and shared library.

Your final RPC mechanism needs to encapsulate all of these operations on the server, and provide stubs to access these in the client library. You do not need to handle more advanced features such as `fcntl`, locking, `dup`, handling forks, etc. However, your server needs to be able to handle file accesses from multiple clients at once, and both the server and client library need to be able to handle multiple open files.

Tutorial: C Networking

Note that the following sample code snippets hardcode both the IP address and port number. Do not do this in your code. Instead, read the IP address from the environment variable "server15440" and port number from "serverport15440" (see starter code for examples).

TCP server

This project requires you to write a multi-client network server. The networking functions in C follow the same template as the file operations, but are based around the concept of a socket, rather than a file, and involve more steps and magic incantations. To create a socket:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);  // TCP/IP socket
```

Then, we need to "bind" our socket to a port -- our server will use this port exclusively. To do this, we first populate an ugly address structure and call `bind`:

```
struct sockaddr_in srv;          // address structure
memset(&srv, 0, sizeof(srv));    // zero it out
srv.sin_family = AF_INET;        // will be IP address and port
srv.sin_addr.s_addr = htonl(INADDR_ANY);   // don't care
srv.sin_port = htons(15440);     // port 15440
rv = bind(sockfd, (struct sockaddr *) &srv,
     sizeof(struct sockaddr));  // bind to the specified port
```

Check the return value of `bind`, because it can fail for many reasons (e.g., another program is using the port, left over connection state from previous runs need to time out, etc.). Then, we set up a queue to listen for incoming clients:

```
listen(sockfd, 5);    // listen for clients, queue up to 5
```

Then, we get the connections from clients:

```
struct sockaddr_in client;
socklen_t sa_size = sizeof(struct sockaddr_in);
sessfd = accept(sockfd, (struct sockaddr *) &client, &sa_size);
```

The `accept` call blocks until a client has connected. On success, it returns a new session socket used to communicate with that client. You call it again to get the next client, and another

new session socket. The client address information is provided in the address structure. We can then communicate with the client using the session socket:

```
rv = send(sessfd, buf, 1024, 0);   // send 1024 bytes from buf
rv = recv(sessfd, buf2, 1024, 0);  // receive up to 1024
                                   // bytes into buf2
```

The return values indicate error or actual bytes transferred. Error codes will indicate if the client dropped or disconnected. The `recv` call is blocking. When we are done with this client:

```
close(sessfd);        // done with this client
```

Note that this is the same `close` function used for file I/O ! In fact, `read` and `write` will generally work on the session sockets. Of course, `lseek` does not make sense here. When completely done, we should close the original server socket:

```
close(sockfd);        // server is done
```

Your server needs to be able to handle more than one client at once. This gets tricky, because both the `accept` call to get new clients and the `recv` calls to get data from existing clients are blocking. There are many ways to handle this issue. One simple but inefficient way to do this is to `fork` a child process to handle each client session. The `fork` call will create an exact duplicate the calling process, except the original will get the process id of the copy as a return value, while the copy gets a return value of 0:

```
sessfd = accept(sockfd, (struct sockaddr *) &client, &sa_size);
rv = fork();
if (rv==0) {          // child process
     close(sockfd);  // child does not need this
     do_stuff(sessfd);     // handle client session
     close(sessfd);  // then close client session
     exit(0);        // then exit
}               // parent process
close(sessfd);  // parent does not need this
// should loop back to accept next client
```

This simple structure will have a single server process that listens for new clients, then launches additional processes to handle each client separately.

TCP client

The client code is a bit simpler. First, create a socket:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);  // TCP/IP socket
```

Then, we connect to the server, by populating an address structure and calling `connect`:

```
struct sockaddr_in srv;          // address structure
memset(&srv, 0, sizeof(srv));    // zero it out
srv.sin_family = AF_INET;        // will be IP address and port
srv.sin_addr.s_addr = inet_addr("127.0.0.1");   // server IP
srv.sin_port = htons(15440);     // port 15440
rv = connect(sockfd, (struct sockaddr *) &srv,
     sizeof(struct sockaddr));  // connect to server
```

Check the return value to make sure connection succeeds. Now we can use `send`, `recv`, and `close` as above:

```
rv = send(sockfd, buf, 1024, 0);    // send 1024 bytes from buf
rv = recv(sockfd, buf2, 1024, 0);   // receive up to 1024
                                      // bytes into buf2
close(sockfd);         // client is done
```

Note that the communication channel between the client and server acts as a reliable byte stream, but you cannot expect a particular `recv` to get data from a single `send`. Data from multiple consecutive `send` calls may be picked up by a single `recv`, and data from a single `send` may end up being picked up by several `recv` calls.

Tutorial: Library call interposition

Most programs in modern systems use dynamic or shared libraries -- the library functions are not part of the executable binary, and instead are loaded at run time. This gives us the opportunity to interpose on standard library calls from existing programs. In Linux and other Unix-like systems, we do this by creating our own shared library implementing functions we want to interpose, and telling the system to load our library first when executing a program.

For example, suppose we want to track the memory allocations performed by some program `myprog`. When we normally run `myprog`, the system will load the libraries it needs, and dynamically link in the library functions used. In particular, standard C functions, such as `malloc`, will be loaded from `libc.so`. Instead of making use of the libc functions, we can create our own version of `malloc()` and `free()` that will log all calls. We then compile them into a shared library (".so" file in Linux). We use the "`LD_PRELOAD`" environment variable to tell the system to load our library first when executing `myprog`, e.g., assuming bash shell:

```
LD_PRELOAD=./mylib.so myprog
```

Now, when `myprog` is run, the system links in our functions first, before loading any other libraries. The unmodified `myprog` program will now use our version of `malloc` and `free`, instead of the ones from the standard C library.

You will have to create a shared library that implements versions of the file operations (`open`, `read`, ...) that communicate to your server to perform the operations at the remote machine rather than locally, and interpose these into existing binary programs that make local calls. If we can interpose on any dynamically-linked binary, why are custom 440* tools needed? The 440 tools are written to directly use `open`, `close`, `read`, etc. In contrast, most of the standard, small Unix tools do not actually use the `open`, `close`, … file APIs; rather they use the richer buffered file API that includes `fopen`, `fprintf`, etc. This family of functions is much larger and more complex. Internally, these functions do call the equivalent of `open`, `close`, `read`, etc., but use functions internal to the C library, or perhaps direct OS system calls. The linker-level interposition we use here cannot catch such calls made from within the C library.

Notes / Hints

- Do not print anything to `stdout` from your interposition library, or extra output (extra white space or newlines) from your server; otherwise your code will fail in the autograder. Use `stderr` if you are printing any debugging output.
- Check the man pages for the file operations to make sure you are implementing the right behaviors and returning the right values and error codes (e.g., `man 2 read`).
- Watch out for `stat`. On some versions of Linux (more specifically, those with older versions of the glibc library), program binaries actually link to `__xstat`, not `stat`. So if you are having trouble interposing on `stat`, check to see if the system uses `__xstat`, which has an additional parameter for version. The standard Andrew Linux machines and Autolab should use `stat`. If you interpose on both, your code should work on any Linux version.
- Watch out for C strings -- many functions assume null-terminated strings. However, getting the length of a string (`strlen`) does not include the null character. So, for example, sending a message like this:

  ```
  char *buf = "hello";
  send( sessfd, buf, strlen(buf) );
  ```

  will not send the null terminator. Depending on what you do with the message, you may need to add the null terminator at the receiving end, or ensure it gets sent as well.
- Do not hardcode the server IP address or port in your client library or server. Instead, read the IP address from the environment variable "server15440" (as in the example client.c). Read the port number from "serverport15440" (for both the client and server).
- You will need to add the absolute path of the directory containing `libdirtree.so` to your `LD_LIBRARY_PATH` environment variable in order to run the `440tree` program. Otherwise, you will get an error that it could not load the shared library.
- Make sure you develop your code in a private directory. By default, AFS home directories are readable by anyone. Please place your coursework in a private directory (e.g. ~/private), or change the permissions on your home directory.
- It is recommended you use `send` and `recv` for networking from your interposition library. Although `read` and `write` can be used for networking, you will have complications as you have interposed on these calls!
- On a related note, the `read`, `write`, and `close` operations can also be used for other I/O (e.g., pipes, stdout, sockets, etc.). So, for example, since all calls to `read` will go to your library, a program that reads from a socket will be directed to your code. Can you make sure such uses work correctly with your interposition library? (Hint: did you hand out that file descriptor? Hint 2: Can you avoid handing out a file descriptor that the system may give out?)
- You do not need to do anything special to handle read-write or write-write conflicts due to concurrent clients accessing the same file. Just pass the operations on to the underlying filesystem at the server, and let it handle this.
- Does `freedirtree` really need to be an RPC?