CSCE 455/855 Distributed Operating Systems

Spring 2001
Steve Goddard

Programming Assignment 1 (PA1), January 29

Due: 6:00pm February 12

_____

The focus of this semester's Distributed Operating Systems course is an implementation of the FTFS (Fault Tolerant Distributed File-System). The most basic feature any file system provides is the ability to create, open, read, write, close, and delete (unlink in Unix) files.

The assignment is constructed in three parts. In the first part, you will implement a simple file system API that makes calls to the host API for the routes creat(), open(), read(), write(), close(), and unlink(). In the second part, you implement your API using a client/server paradigm in which the two processes communicate using Unix sockets. The client is the process performing file operations. The server receives file operation requests and performs them on behalf of the client. The client and server may (but need not) be running on separate machines. In the third part, you will use RPC rather than sockets for communicating between the client and server.

Part I: Implement an FTFS API for a single process

Implement a simple FTFS API as C or C++ function calls that are compiled into one or more .o files that a user links into their program to use the FTFS file system. The functions implemented in the API are:

```
/* see man 2 open for details on the operation of these functions */
int FtfsOpen(const char *pathname, int flags);
int FtfsOpen(const char *pathname, int flags, mode_t mode);
int FtfsCreate(const char *pathname, mode_t mode);

/* see man 2 read for details on the operation of this function */
size_t FtfsRead(int fd, void *buf, size_t count);

/* see man 2 write for details on the operation of this function */
size_t FtfsWrite(int fd, void *buf, size_t count);

/* see man 2 close for details on the operation of this function */
size_t FtfsClose(int fd);

/* see man 2 unlink for details on the operation of this function */
size_t FtfsUnlink(const char *pathname);
```

You will need to use the macros va_start, va_arg, and va_end to implement the FtfsOpen() function. See man va_start on a Linux box or an ANSI C reference for these macros.

Make sure you test this API thoroughly and document your tests. This part is not meant to be difficult. Its purpose is to familiarize you with the basic Unix file API and the use of variable

number arguments. The next two parts will also be based on this API. However, they will require a client server solution.

Information on Sockets

Below is a brief summary of the basic socket calls, the role they play in the client and the server, and the order in which they are called. The information is presented here to give you a starting place, but you should also study the example code provided with the project, which is located in

http://www.cse.unl.edu/~goddard/Courses/CSCE451/StandardHandouts/IPCexample.tar.gz.

While both of these approaches will help, there is no substitute for reading the manual pages. They are easily found using the man or xman commands.

The first important point is that the socket interface assumes that the client-server relationship holds between two processes. One is the server, which waits for requests at a fixed port address. The fixed address is the equivalent of waiting at a known phone number. The client requests service from the server by calling its number from anywhere it likes. The telephone analogy is not perfect, though, since the processes involved also execute system calls which are the equivalent of building and destroying the telephones, as well as establishing their numbers!

The first system call of concern is the socket() call, which creates a file descriptor attached to a socket structure. This creates the communication point in the process that calls it. Both the client and the server call socket(). Next, the server uses the bind() call to establish its address, and listen() to define the number of pending calls that may wait for attention from the server. The server then calls accept() when it wishes to accept a request for service. Note that accept() returns a new socket, unrelated to the original socket created by the server, which provides a connection to the client making the request which has just been accepted.

After the client creates its socket using the socket()call, it needs to make a connection to the server. The client can first bind its socket to a specific address, although it need not do so. Whether it picks its port address or not, it requests a connection to the server using the connect() call. At this point the client and server are in communication. Each process reads and writes to the sockets as they would read and write to files. When finished, the client and server discard the sockets assuming the close() system call.

The example code available from the class Web page illustrates the use of sockets. The following pseudo code provides a summary of the system calls required to establish a connection.

```
      Server                        Client
fd = socket()                 fd = socket()
bind(fd, server address)      [bind(fd,optional explicit client address)]
listen(fd, queue-length)

nfd = accept(fd, ... )        connect(fd, server-address);

read requests                 write requests

write answers                 read answers
close(nfd)                    close(fd)
close(fd)
```

See the handouts on System Calls and IPC and Sockets for more information on how these calls actually work.

Part II: Minimal Client-Server

Use the client/sever paradigm to implement the FTFS API. You should write two programs, a server program (process) and a client program (process); the programs should be compiled separately, so that they each have their own executable image. The two processes will communicate with each other in a connection-oriented manner using sockets, but the communication should be as transparent as possible. That is, if possible, the client program should be identical to the program from Part I with only the API implementation changing.

Your API will now have client side and server side implementations with message passing via sockets connecting the two. You must decide whether to choose a connection-oriented or connection-less protocol for your API. If necessary, you may create an initialization routine, but try not to. If necessary, you may also create a shutdown routine to free resources.

During initialization (either an explicit call or in the first call to one of your API routines—an implicit initialization) or for each call (depending on your protocol choice), the client-side API implementation should `connect()` to the server process (after creating the socket, of course). The client API will accept the inputs, marshal them into a message, send the message to the server, and await the reply. When the reply is received, it extracts the return parameters from the message and returns them to the caller.

The server process should `accept()` an incoming connection request (after creating, binding, and listening on the socket), extract the parameters from the message and make the requested call to the FTFS API implementation you created for PART I (perhaps with a name change if necessary). You may assume a default base directory for your server-side implementation if that helps.

Additional Notes

1. Note that because of byte ordering conventions ("big-endian" versus "little endian") on different machines, your client and server may not correctly interpret the contents of an exchanged message. If your client and server are running on different machines with different byte ordering conventions (e.g., one is running on a DEC workstation and one is running on a SUN SPARC), they will not interpret the content of each others' structures correctly. You should use the `htonl()` and `ntohl()` functions to insure that all machines interpret the byte orderings in the same manner.

2. In writing your code, make sure to check for an error return from all system calls. If there is an error, the system declared global variable, `errno`, will give you information about the type of error that occurred:

   ```
   #include <errno.h>

   ......

   if ( (sockid = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
   ```

```
        printf("error creating client socket, error%d\n",errno);
        perror("meaning:"); exit(0);
    }
```

See the man pages `errno(2)` and `perror(2)` for a description of the error codes and the use of `perror`.

3. Make sure you close every socket (file descriptor) that you use in your program. If you abort your program, the socket may still hang around and the next time you try and bind a new socket to the port ID you previously used (but never closed), you may get an error. Also, please be aware that port ID's, when bound to sockets, are system-wide values and thus other students may be using the port number you are trying to use. With this in mind, I suggest you use the last four digits of your SS # and add it to 10,000 to create your port number. The reason for the addition is that port numbers below 5000 are reserved for system use, and those between 5,000-10,000 are used by lots of local applications. If you follow this rule the likelihood of colliding with another program using the same port at the same time is very low.

Part III: RPC
Required for CSCE 855 students. 20% bonus for CSCE 455 students!

Implement your FTFS API using RPC. The client and server do the same tasks as in Part II, except that they communicate using RPC instead of socket calls. You should use `rpcgen` to create the client and server stubs as well as the template code needed.   You will need to read the man pages and/or other sources to learn how to use RPC and `rpcgen`.

Grading Policy for Programs

The programs you hand in should work correctly and be documented. When you hand in your programming assignment, you should include:

1. A program listing containing in-line documentation.

2. A separate (typed) document of approximately two pages describing the overall program design, a verbal description of "how it works" including the basics of what the system is doing underneath, and design tradeoffs considered and made. Also describe possible improvements and extensions to your program (and sketch how they might be made).

3. A separate description of the tests you ran on your program to convince yourself that it is indeed correct. Also describe any cases for which your program is known not to work correctly.

4. A make file that compiles your program(s).

Please hand in your source files for all parts of this project.

The program should be neatly formatted (*i.e.*, easy to read) and structured and documented according to the guidelines distributed in class. Use the handin program to submit your program(s) for grading. This is assignment 1. Your grade will be determined as follows:

```
Program Listing
      works correctly 40%
      in-line documentation 15%
      quality of design 25%
Design Document 15%
Thoroughness of test cases 05%
```

START EARLY. THIS IS HARDER THAN IT LOOKS!