CS 502 Operating Systems — WPI, Fall 2003
Craig E. Wills — Project 4 (30 pts)
Assigned: Monday, November 10, 2003 — Due: Monday, December 1, 2003

Introduction

This assignment is an opportunity for you to use Unix sockets to build a real server. You can use the sample code given in class as a starting point for building a server using sockets. You will be building a server to actually serve Web content. Your server will respond to real HTTP requests and reply with appropriate responses. You can test your client using a standard Web browser.

Basic Objective

Your server will be reading HTTP GET requests sent to a given port number. For example, if your server is started up on the "ccc1" machine on port 4242 then the URL for accessing the text version of the WPI home page is `http://ccc1.wpi.edu:4242/index-t.html`. Because your server should assume that all requests map to files and because the main WPI home page is generated with a CGI script, the home page will not be served correctly by your server.

The content served by the WPI Web server is located in the directory `/www/docs`. This directory is prepended to each request. Thus the given URL maps to the file `/www/docs/index-t.html`. The Web server also has a special rule for requests ending in a "/". These requests appear to be directories, but the Web server appends the file `index.html` on to the end of these requests. Thus a request for `/News/` should map to the file `/www/docs/News/index.html`. You should simply ignore all requests that do not map to a regular file after following the given mapping rules. Web servers have other mapping rules, but these are the only two we will use in this project.

Your server should be started up giving a port number on the command line. Port numbers up to 1024 are reserved so use a port number (16-bit integer) larger than this value. If you receive a "bind failed" error then another server (perhaps one of your's!) is currently bound to that port. You may also see this error message if you try to restart your server immediately after it has just exited. The Unix operating system uses a short time limit (30 secs?) in which it refuses to let a new process bind to a port that has just been used.

In the simplest form your server should be a *single* process that accepts a connection using *accept()*, receives the request and its headers then sends a response header and content back to the client. The server then closes the socket connection used for handling this request.

HTTP Request

Your server first needs to handle an HTTP request. The following is an example request generated by a real browser for `/index-t.html` located on `ccc1.wpi.edu` at port 4242. The first line of the request contains the type of request (you only need to handle `GET`, but other

types such as `HEAD` and `POST` are possible in HTTP). Following the `GET` request is the object to be requested. You will first want to ensure that the request is of type `GET` and then extract the object string. The remainder of the line identifies the HTTP version used by the browser. You can ignore the HTTP version. The remaining lines are HTTP request headers. In HTTP/1.1, the `Host` field is required, but you can ignore this line and all other headers. However, your server needs to read these headers.

```
GET /index-t.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; U; SunOS 5.7 sun4u)
Host: ccc1.wpi.edu:4242
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

To aid you in reading the request a line at a time, the routine *sockreadline()* has been created and put in the file `/cs/cs502/public/example/sockreadline.c`, which you are welcome to copy and use. This routine receives a character at a time from a given socket and stores these characters in a NULL-terminated character buffer. It returns when the newline (`\n`) character is reached.

Note that the HTTP specification expects all lines to be terminated with a CR (carriage return) and LF (line feed) characters. These characters are "`\r\n`" in C/C++. Note also that the HTTP specification expects a "blank" line containing only "`\r\n`" at the end of the request headers. Your server should keep reading lines until it receives such a line.

HTTP Response

There are a number of HTTP response codes, but for this project we will use only two: 200 and 404. If you receive a `GET` request for an object that can be successfully mapped to a file then you should open this file for reading using the system call *open()*. If you successfully open the file then your server should first send the HTTP response "`HTTP/1.0 200 OK\r\n\r\n`" indicating success followed by a blank line. Note that two sets of CR/LF characters are sent. Subsequently you should use *read()* to read all content from the file and send it on to the server. Use *close()* to close the file when done reading and close the socket connection. Your server is done handling this request. Note that you should use these system calls for I/O rather than use routines expecting only text as the object contents may not be text.

If the request is not valid or cannot be mapped to a file that is successfully opened then your server should send back to the client the response "`HTTP/1.0 404 Not Found\r\n\r\n`" indicating failure and close the socket connection.

Client Testing

To test your server, you can use a standard browser. However, to make testing simpler and to be able to see the response headers, you should create a simple Web client. This client should connect to a given port on a given host and send a minimal request string. Use command line arguments to control your client. For example:

```
% webclient ccc1 4242 /index-t.html
```

can be used to request the object from port 4242 on the machine ccc1. Your simple webclient will need to connect to the port and send the `GET` line (ending in CR/LF) followed by a blank line with CR/LF. You can use "HTTP/1.0" as the version. Your client should then receive back the response headers and content from the server and print them to stdout. Beware of requesting images with your simple client as the content will likely will not print well. You should also be able to use your Web client with any Web server by sending to the standard Web server port 80.

Completing a simple server and a simple client and verifying they both work for HTML content is worth 20 out of 30 points for the project.

Multi-Process Server

For an additional five points on the project, you should modify your Web server to fork off a process to handle each request. The child process will use the socket value returned by *accept()* to handle all interaction with the client while the parent process loops back waiting to accept additional connections. This approach will allow requests to be handled in parallel by the server.

Watch out for this approach as the parent and child processes should be sure to close socket connections they are not using. Also be sure to insert code to cleanup "zombie" processes that are left after the child processes terminate. Use *wait3()* to do so such as the following code that collects status information. See the man page for additional information.

```
int pid;
int status;
struct rusage ruse;

while ((pid = wait3(&status, WNOHANG, &ruse)) > 0)
    ;
```

You can either do this in your main server code or in a signal handler, but in the latter case *accept()* will return an error indicating an interrupted system call (`EINTR`), which you need to check for.

Your server should support either forking or non-forking. Use a command line argument (default is non-forking) to indicate which approach to use. For example:

```
% webserver 4242 fork
```

is used to start your server on port 4242 with the forking option. Use "none" or leave out the last argument to indicate a non-forking server.

Multi-Threaded Server

For the remaining five points of the project, enhance your server to also support threads. In this version only *one* process is used, but a new thread is spawned to handle each request. The main thread will loop back to accept additional requests. Remember that sockets are shared amongst all threads so spawned threads should not close the socket used by the main thread for accepting connections. Also remember that all threads run in the same address space so that the use of shared buffers/variables or non-reentrant routines will cause problems.

Use the command-line argument "thread" to indicate your Web server is running in multi-threaded mode.

Performance Testing

If you implement more than one server version you should do some performance testing to see how many requests each approach can handle. A simple test is to repeatedly request the same Web object from the server. You can use your Web client to test, but you will need to run multiple versions. Alternately you can use a web server testing tool.

Submission

Electronically turn in the code for project using the project name *proj4*. You should include a Makefile for compiling "webserver" and "webclient". The file `Makefile` in `/cs/cs502/public/example` can be used to compile the *sockclient* and *sockserver* examples given in class. You can start with this makefile and modify it appropriately. Make sure you use a TAB character to indent the operations for a target as is done in the given file `Makefile`.