INB(N)365 Assignment 2
Distributed Communication

Group Work
This assignment can be completed individually or in pairs. If you want to work in a pair find a partner from your class. INB students can only pair with other INB students and the same applies for INN students.

Due Date
Friday Week 13 – 28/10/2011

Weighting
20% of the final grade

Objectives
- Implement a project in the C language on Linux using the Make build system.
- Implement a Client/Server application in C using BSD sockets.
- Build upon Producer/Consumer pattern learned in assignment 1 by using it for thread pools.

Submission
This assignment is to be submitted via the OAS system. Students who complete the assignment in pairs only need to submit one assignment per pair.

http://www.student.qut.edu.au/about/faculties-institutes-and-divisions/faculties/scitech/online-assessment-system

The submitted zip archive (assign2.zip) needs to contain
- the source code implementing the assignment.
- a makefile to build the source code.
- a report in Word 2003 (.doc) or Portable Document Format (.pdf).
- the provided htdocs directory containing the content for the HTTP server.

Source code
- The source code can be contained in single or multiple files.
- The makefile needs to generate binaries called client and server so that the command 'make clean && make' will recompile the source from scratch and the server or client can be run via './server' or './client'.

The report must contain
- a statement of completeness indicating the tasks attempted and any deviations from the specifications or known bugs.
- a description of the data structures, threads, sockets, client, server and how they interact.
- a short description of how to run the program.

Task 1 – A HTTP 1.0 Client and Server

This task is to be completed by both INB and INN students.

Your tasks is to implement a basic, static file serving, single threaded, HTTP 1.0 server and client. A HTTP server is one of the distributed infrastructures that supports the World Wide Web (WWW). A HTTP server listens on port 80 by default and waits for requests from a web browser. You will be able to test your server with a web browser. However, you will still be required to implement a simple client to demonstrate that you understand both sides of Client/Server programming.

The Client and Server will be implemented in the C programming language using BSD sockets on the Linux operating system. This is the same environment used during the weekly practicals. You will have acquired all the necessary knowledge and skills to complete the assignment by attending all the lectures and practicals, and completing the associated reading from the text book.

These programs are to run in a terminal reading input from STDIN and writing output to STDOUT.

HTTP Server

You are required to implement a simple HTTP 1.0 server that implements a subset of the standard. The HTTP 1.1 standard superseded 1.0 in 1997. However, it is simpler to implement the earlier specification. We are not asking you to implement something as complex and robust as Apache. However, we hope that you will enjoy implementing a piece of technology that is frequently used.

All messages between a HTTP client and server are terminated with Carriage Return Line Feed (CRLF) which corresponds to "`\r\n`" in C. Some clients may send only the LF, so your server must handle both cases. A good place to start is to implement two functions for reading and writing a line to a socket. For example,

```
// Read a CRLF terminated line from socket into buf as a null terminated string.
// Note that buf is size bytes long.
// Returns the number of characters received not including CRLF.
size_t readline(int socket, char* buf, size_t size);

// Write a CRLF terminate line to socket.
// Note that str is a null terminated string and is NOT terminated with CRLF.
void writeline(int socket, char* str);
```

You will only be implementing the GET method which requests a resource from the server. A GET request sent from the client looks like following.

```
GET /path/to/file/index.html HTTP/1.0
[headers]
[blank line]
```

Note that the client may send headers. You will need to read and discard them.

The server replies with following if the file was found. You may call your server anything you want in the 'Server: name' header line.

```
HTTP/1.0 200 OK
Server: httpd
Content-Type: text/html
[blank line]
[the contents of index.html]
```

If the file was not found it replies with the following. Note that the 404 message can be any valid HTML.

```
Server: httpd
Content-Type: text/html
[blank line]
<html><body>404 Not Found</body></html>
```

Your server will automatically serve index.html if a request is sent for '/'. Note that the server serves files from a directory called htdocs in the current directory. We have provided htdocs.zip that contains index.html and CGI scripts in Python.

The server will write status updates to STDOUT. They look like the following.

```
Server is listening on port 80
GET '/' from 127.0.0.1
GET '/index.html' from 127.0.0.1
GET '/some.bogus.file' from 127.0.0.1 FILE NOT FOUND
```

Initially you can use a telnet client and web browser to act as the client for your server. If the server is listening on port 80 you can type

```
telnet localhost 80
```

at the command prompt and manually type in the get request such as

```
GET / HTTP/1.0
[blank line]
```

for which the server will reply with a response similar to the one listed above. A web browser will render index.html correctly.

The server will take one command line parameter that indicates the port to listen on. The following command will run the server on port 80.

```
./server 80
```

Exiting the Server

The server will exit upon receiving a SIGINT from the operating system. SIGINT is a type of signal found on POSIX systems. On Linux this means ctrl+c has been pressed and the user wants to exit the program. The INT in SIGINT stands for interrupt. You will need configure a signal handler and ensure the server exits cleanly. Section 4.4.3 on page 167 of the text book has an introduction to signal handlers. Section 21.9.1 on page 837 has a detailed description of signals on Linux.

The Little Unix Programmer's Group has a introduction on signals including configuring handlers in C at http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html.

HTTP Client

The client will take two command line parameters of hostname and port to connect to. The following command will run the client connecting to localhost on port 80.

```
./client localhost 80
```

The client will send a GET request for '/' and print the server response to the screen. It will exit once the communication has completed with the server.

Resources

A more detailed explanation of the HTTP 1.0 and 1.1 protocols can be found at http://jmarshall.com/easy/http/.

RFC 1945 is the specification of HTTP 1.0 and can be found at http://ftp.ics.uci.edu/pub/ietf/http/rfc1945.html and the HTTP 1.1 RFC 2616 can be found at http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Task 2 – A Thread Pool for the HTTP 1.0 Server

This task is to be completed by both INB and INN students. The server you have implemented in Task 1 can only handle a single HTTP request at a time. You are required to extend the server to accept multiple concurrent connections using a thread pool.

The server will allow up to 30 clients to use the system at the same time. You will need to implement a thread pool where each incoming connection is handled in a separate thread. A thread pool reduces the cost of constantly creating and destroying threads to handle requests. Instead, threads are reused to handle connections. It also limits the number of incoming connections able to be processed at once.

You can reuse the producer consumer pattern from the first assignment, where the main thread accepts incoming sockets and places them in a queue (the producer) and 30 threads remove sockets from the queue and process the connection (the consumer). The text book describes thread pools in section 4.4.4 on page 168.

Task 3 – A CGI Implementation for the HTTP 1.0 Server

This task is only required to be completed by INN students. INB students may wish to complete this task as a challenge but you will receive no extra marks.

The Common Gateway Interface (CGI) is a standard defined in RFC 3875. It allows web servers to pass generation of web pages to another program. These programs are typically written in scripting languages such as Perl or Python but can be written in any language. Upon a HTTP POST request, data is passed to an external program via Environment Variables. The program writes a header and HTML response to standard output. The program output is captured by use of UNIX pipes.

Once again the technology we are asking you to implement, CGI, is not state of the art in communication with server side programs. Modern web servers use more efficient methods for running various programming languages on the server side. However, it is fairly straight forward to implement CGI and it is language agnostic.

HTTP POST

HTTP POST is typically used to submit form data for server side processing. The format of this data is (key, value) pairs that represent named variables. In the example below, the body of the message contains the variables animated, alive, food, firstname and lastname. Note that the header of a HTML message is terminated by a blank line.

```
POST /path/to/script.py HTTP/1.0
From: brains@zombie.qut.edu.au
User-Agent: ZombieBrowser/0.28
Content-Type: application/x-www-form-urlencoded
Content-Length: 76
[blank line]
animated=true&alive=false&food=brains&firstname=Randall&lastname=Skeffington
```

Any of the HTTP methods, including POST, can be used for client server communication as long as the client and server agree on the format of the data being exchanged in the body of the message. However, you will be using it for processing HTML form data.

Environment Variables

Environment Variables create the operating environment in which a process runs. They are a set of named values that can be manipulated and often affect the way a process runs. On Linux each process has its own set of private Environment Variables. For example HOME holds the path to the current users home folder. With the Bash shell you can reference environment variables by placing a $ in front of the variable name. The export command will list all environment variables.

```
vmuser@ubuntu:~$ echo $HOME
/home/vmuser
```

The `putenv()` function allows you to set Environment Variables from C. The following example illustrates setting the REQUEST_METHOD variable to POST.

```
putenv(“REQUEST_METHOD=POST”);
```

UNIX Pipes

UNIX pipes connect standard output from one program to the standard input of another. Pipes are one of the oldest forms of Inter Process Communication (IPC) on UNIX like systems. For example, executing the following command takes the standard output of the list command and pipes it into the word count program for which the -l parameter specifies to count the number of lines received on standard input.

```
vmuser@ubuntu:~$ ls -la | wc -l
156
```

For a high level overview of pipes please see section 6.2.1 of the Linux Programmer's Guide at http://tldp.org/LDP/lpg/node10.html. TLDP is The Linux Documentation Project and contains many guides and HOWTOs on Linux programming and administration.

Use of the `pipe()` and `fork()` system calls are required to orchestrate pipes in C. The `fork()` system call is used to fork a child process that is communicated with via pipes. In the example of a CGI implementation, the parent process is the HTTP server and the child process is the CGI program that generates the web page. Therefore, your HTTP server will need to `fork()` a child process that executes the required Python script.

The `pipe()` system call takes two integers that are file descriptors for reading and writing of the pipe. Both the parent and child processes have two file descriptors each. For example, if the parent process writes into the pipe the child process can read from the pipe on the other end.

Please see section 6.2.2 of the Linux Programmer's Guide to see the complete details of orchestrating pipes in C at http://tldp.org/LDP/lpg/node11.html.

Putting it All Together

Your server will set several Environment Variables based upon the HTTP POST request from the client. It will fork a child process to run the requested script capturing the output via UNIX pipes, sending this output to the client as the HTTP response.

You will need to modify your request handler from Task 1 to handle HTTP POST requests that execute CGI scripts. A file will only be executed by CGI if its execute bit is set. The use of the `stat()` function can determine file properties as described in `man 3 stat`. If the requested file is not executable the server will send a HTTP 500 response.

Your server will set three Environment Variables for the script being executed. These are REQUEST_METHOD, QUERY_STRING and CONTENT_LENGTH. Where the REQUEST_METHOD is POST, the QUERY_STRING is the body of the HTTP request (the named variables sent from the client), and CONTENT_LENGTH is the 'Content-Length:' field from the HTTP POST requests header. Once these Environment Variables have been set, the script can be executed using `execl()` as described in `man 3 execl`. The server will write the response to the client via the use of UNIX pipes.

The server will write status updates to STDOUT. They look like the following.

```
Server is listening on port 80
POST '/cat.py' from 127.0.0.1
POST '/dog.py' from 127.0.0.1 NOT EXECUTABLE
```