Project - 1
CSE 421/521 – Operating Systems
Due: October 22 @11:59pm, 2012

1. Preparation

Before beginning your work, please read the following carefully:

- Chapters 4-6 from Silberschatz
- Lecture slides on Threads, CPU Scheduling and Synchronization
- Recitation notes on Processes, Threads and Networking in Unix Environment
- RFC1945 (http://ftp.ics.uci.edu/pub/ietf/http/rfc1945.html)

2. Programming Task: Implement a Multi-threaded Web Server

The objective of this project is to implement a multithreaded web server "**myhttpd**" in C/C++ on a UNIX-based platform.

SYNOPSIS: myhttpd [**−d**] [**−h**] [**−l** *file*] [**−p** *port*] [**−r** *dir*] [**−t** *time*] [**−n** *threadnum*] [**−s** *sched*]

DESCRIPTION: myhttpd is a simple web server. It binds to a given port on the given address and waits for incoming HTTP/1.0 requests. It serves content from the given directory. That is, any requests for documents is resolved relative to this directory (the document root – by default, the directory where the server is running).

2.1 OPTIONS:

−d : Enter debugging mode. That is, do not daemonize, only accept one connection at a time and enable logging to stdout. Without this option, the web server should run as a daemon process in the background.

−h : Print a usage summary with all options and exit.

−l *file* : Log all requests to the given file. See LOGGING for details.

−p *port* : Listen on the given port. If not provided, **myhttpd** will listen on port 8080.

−r *dir* : Set the root directory for the http server to *dir*.

−t *time* : Set the queuing time to *time* seconds. The default should be 60 seconds.

−n *threadnum*: Set number of threads waiting ready in the execution thread pool to *threadnum*. The default should be 4 execution threads.

−s *sched* : Set the scheduling policy. It can be either **FCFS** or **SJF**. The default will be **FCFS**.

2.2 PROTOCOL:

myhttpd speaks a simplified version of HTTP/1.0: it responds to GET and HEAD requests according to RFC1945. When a connection is made, **myhttpd** will respond with the appropriate HTTP/1.0 status code and the following headers:

Date : The current timestamp in GMT.
Server : A string identifying this server and version.
Last-Modified : The timestamp in GMT of the file's last modification date.
Content-Type : text/html or image/gif
Content-Length : The size in bytes of the data returned.

If the request type was a GET, then it will subsequently return the data of the requested file. After serving the request, the connection is terminated. The HEAD request will only return the metadata but not actual content.

If the request was for a directory and the directory does not contain a file named "index.html", then **myhttpd** will generate a directory index, listing the contents of the directory in alphanumeric order. Files starting with a "." are ignored.

If the request begins with a '~', then the following string up to the first slash is translated into that user's **myhttpd** directory (ie / home/*<user>/myhttpd/*).

2.3 LOGGING:

By default, **myhttpd** does not do any logging. If explicitly enabled via the **−l** flag, **myhttpd** will log every request in a slight variation of Apache's so-called "common" format: '%a %t %t "%r" %>s %b' all in a single line per request. That is, it will log:

%a : The remote IP address.
%t : The time the request was received by the queuing thread (in GMT).
%t : The time the request was assigned to an execution thread by the scheduler (in GMT).
%r : The (quoted) first line of the request.
%>s : The status of the request.
%b : Size of the response in bytes. i.e, "Content-Length".

Example:

```
127.0.0.1 - [19/Sep/2011:13:55:36 -0600] [19/Sep/2011:13:58:21 -0600]
"GET /index.html HTTP/1.0" 200 326
```

(check http://httpd.apache.org/docs/1.3/logs.html for more info on Apache logs.)

All lines will be appended to the given file unless **−d** was given, in which case all lines will be printed to stdout.

2.4 MULTITHREADING:

The server will consist of **2+n** threads. A pool of n threads will be always ready for executing/serving incoming requests (**n *x execution threads***). The number n is given as a parameter when you start the server (using option: *-n threadnum*). One thread will be responsible for listening to incoming HTTP requests and inserting them in a ready queue (***1 x queuing thread***). Another thread will be responsible for choosing a request from the ready queue and scheduling it to one of the execution threads (***1 x scheduling thread***).

2.5 QUEUING

The queuing thread will be continuously listening to the port *p* for incoming http requests. As soon as a new http request comes to the **myhttpd** server, it will be inserted into the ready queue. For the first *t* seconds after the server is started, there will be no execution and all requests will wait in the ready queue. At time *t+1*, the scheduling thread will start choosing one of the requests in the ready queue (see section 2.6) and assigning them to one of the available execution threads. Meanwhile, new requests may come to the **myhttpd** server and they will be inserted into the ready queue. **NOTE:** When you insert a request into the ready queue, you also need to insert the size of the requested file if the request type is GET. If the request type is HEAD, enter 0 as the file size since no content will be returned. This information will be used in the scheduling process.

2.6 SCHEDULING

The scheduling policy to be used will be set via the [**–s** *sched*] option when **myhttpd** server is first started. The available policies are First Come First Serve (FCFS) and Shortest Job First (SJF). When SJF scheduling policy is selected, you can use the file size information as the job length for scheduling purposes, assuming serving larger files will take longer. The scheduler thread will choose one of the requests in the ready queue according to the scheduling policy selected. The request will then be assigned to one of the available execution threads for service. There will be no scheduling done during the first *t* seconds after the **myhttpd** server is started. This time period will be used to accumulate some requests in the ready queue.

2.7 SYCHNRONIZATION

You need to make sure that you protect the ready queue and other shared data structures across multiple threads to prevent race conditions. We will learn some techniques such as semaphores to achieve this in the synchronization lecture.

3. What to Submit?

1. You need to prepare a **tar package** containing all source files of the project, and call it <team#>.tar. This package should also include a Makefile and README file. The whole package should compile when the tester simply types make in the source code directory. Your README file should contain details and options on how to compile and run the server, if there are any.

   This package should be **emailed** to Prof. Tevfik Kosar (tkosar@buffalo.edu) and cc'ed to the TAs Ying Yang (yyang25@buffalo.edu) and Weida Zhong (weidazho@buffalo.edu) by **October 22nd @11:59pm.**

2. You also need to write a **design document** for your project. In this document, you should include:
   - **a.** The team#, names and email addresses of the team members
   - **b.** Responsibilities of each team member in the project
   - **c.** The data structures you used for the implementation of the queuing, scheduling, multithreading and synchronization components of the server
   - d. How are context switches between threads implemented in your code?
   - **e.** How are your race conditions avoided in your code?
   - **f.** Briefly critique your design, pointing out advantages and disadvantages in your

design choices.

g. Please cite any online or offline resources you consulted while preparing your project, other than the course materials.

h. Attachment 1: README file (see above)

i. Attachment 2: Source code of your project

A **hardcopy** of this design document should be submitted to Prof. Kosar at the beginning of the class on **October 23rd, @9:30am**