Project 1 (due 2/17/2006 at 11:59PM)
A Rudimentary but Powerful Web-Server

In this project, you will design and implement a multi-threaded web server for static pages. At a first approximation, a web server is a server program that implements the HTTP protocol. Your web server should be based on a sound, scalable design.

I GENERAL INSTRUCTIONS
- Read these instructions carefully! They may save you a lot of time later.
- You may work individually or in pairs.
- Subsequent projects will be based on this one and not always incrementally! Follow good software engineering practices—you will have to live with the consequences. (As a suggestion: use unit tests, document separately what data are protected by what mutex/condition variable, isolate into libraries code pieces that you can reuse, comment your code thoroughly, make your programs as flexible as possible.)
- The time allocation is rather generous (4 weeks), exactly because everybody has to finish this project before they can advance to the next. Be sure to start early, though!
- You may use any machine that has a reasonable implementation of Pthreads (most likely Solaris or Linux). In the end, your server should run on some GT machine (either CoC or OIT). Pthreads are installed on the OIT machines, so you do *not* need to use CoC resources for development but you may need to do so for experiments.
- If you want to use the CoC computers and you don't have an account, apply for one by filling out an online account request form (http://www.cc.gatech.edu/cns/forms/account_form.html). The administrators already have the class roster and will give you an account - I do not need to approve the applications individually.
- Use the pointers on the web page for reference to concrete technical information (e.g., Pthreads tutorials and examples, specification of the HTTP protocol, thread debugging document, etc.). If you are unfamiliar with network programming, read the socket programming examples on the class web page and the man pages for the `socket`, `bind`, `listen`, `accept`, and `connect` calls. The socket code patterns that you will need for clients and servers are very basic.
- Use the newsgroup for broad questions. For questions that are not of general concern, ask me.

II DESCRIPTION OF PROJECT STAGES
The project consists of three stages:

1. Web Server
Overview. Implement a simple web server. Strictly speaking, you are only required to implement a tiny subset of the HTTP protocol, so formally your server should not be called an HTTP server. It should be good enough for most common browsers, though.

You only need to support "simple" HTTP requests and responses (request: GET <path>, followed by a carriage return and line feed; response: the body of the requested document and connection termination afterwards). Additionally, implement some minimal failure

functionality (e.g., return a "404 Not Found" instead of just dropping the connection). You can find the specification of the HTTP protocol on the Web, for instance, in: http://www.w3.org/Protocols/rfc2616/rfc2616.html You can implement more than the required functionality (some extra credit opportunities exist), but this is not the focus of the project. An easy way to implement the tedious parts of HTTP is to get code from an existing implementation. For instance, `micro_httpd` (http://www.acme.com/software/micro_httpd/) implements much of HTTP in about 150 lines of C code!

Your server should be runnable with a parameter determining the port on which it listens for connections.

Architecture. There are several good designs for high-performance UNIX web servers. The one we will follow consists of a "boss" thread receiving requests and dispatching them to "worker" threads. (If you want to follow a different design, see me!) Worker threads then take over the rest of the communication until the request is satisfied (i.e., the requested file is sent). To avoid the overhead of creating new threads for each incoming connection, you should implement a pool of worker threads that consume work requests produced by the boss thread. The size of the pool (i.e., the number of threads) should be a run-time parameter. It is a good idea to have the scheduling scope for all your threads to be the system scope (i.e., each of your threads is mapped to a different kernel thread so that you get independent scheduling). This is something you can experiment with.

Start your server running, preferably on your local machine. Make it look for files in some directory *on the local disk* (e.g., `/var/tmp/<yourname>`). (This is not too important right now but it will be in later stages, when we will be doing performance measurements.) *For security reasons, make sure your server is restricted in terms of what files it will return to clients!!!* (See "security", below.) Point your browser to your server and make sure that it works. That is, if you have installed your server on machine `nonexistent.cc.gatech.edu`, port 8008, give your browser the URL http://nonexistent.cc.gatech.edu:8008/file1.html (assuming you have a file called `file1.html` in the directory that your server searches).

Security. It is very important that you ensure the security of your files while running the server. Anyone who suspects you might have an unprotected server running can scan the ports and use the server to get your private files! For instance, if I am running a web server on the `/var/tmp` directory of `ocelot.cc.gatech.edu`, port 8008, someone could try to access
http://ocelot.cc.gatech.edu:8008/../../net/hc281/ada/private
and grab file `private` from my directory.

If you had administrator privileges, you could restrict the files your server can reach through the `chroot` command or system call (see the man pages). Unfortunately, this is not an option on the lab machines. The easiest way to protect yourselves is to scan the filename that the web client is trying to retrieve and make sure it is an approved one. For instance, you could make sure that the filename retrieved has no more than one "/" character.

Hardcoding. Please do not hard code machine names, port numbers, or your local directory in the code you submit. The only thing you may hardcode is if the webserver is searching for files relative to the current directory where it executes (.).

2. Testing Tools
Overview. Now that the web server is running, you will need some means for evaluating its performance as well as the performance of future enhancements to the design. You probably don't have access to a few hundreds of human users who can put a load on your server. Therefore, we will need to write a client program to simulate multiple user requests. Write such a web client. It should be a program entirely independent from your web server, that is, a different executable, runnable on a different machine if needed. Of course, you may reuse code from the server, if needed.

The client should be multithreaded for performance. The number of threads it will create should be a parameter. Each thread can access the web server a fixed number of times (e.g., 10), requesting files uniformly at random. (In practice, uniform random accesses may not be a good way to evaluate performance. Nevertheless, real-world access patterns are not the focus of this project.) The set of all files the clients can access should be a compile-time or run-time parameter. Make sure that your client keeps track of how many bytes it retrieved from the server. This could be either a global variable or a per-thread variable. In the end, your program should report how many bytes it got from the server.

Warning: some network calls that your client may use are *not* thread safe but have thread safe variants. The usual `gethostbyname` call is one of these. Overall, you are responsible for ensuring the thread safety of the calls you use.

Extra credit. In addition to your multithreaded client, you may use existing tools and/or benchmarks for testing webserver performance (e.g., `httperf`).

3. Experiments
Overview. Now test your server using your client. For the experiments, you would preferably need a machine where you are the only active user, but this could be any old and slow workstation in a public lab. Run both the server and the client on the same machine. Make several copies of a file in `/var/tmp/` or any other local directory. Make your client access all the files at random. Does your server fail when overloaded? Does it just die or does it gracefully refuse connections? Make sure the server is quite robust. Test its limits. (Make sure you are running on files stored locally, otherwise you are only testing the limits of the network connection. :-) Vary the number of threads in the client and the number of threads in the server. Vary the number of files and file sizes of the files you are retrieving by a couple of orders of magnitude. Keep notes of your server's throughput in terms of MB/s retrieved by the client. Do you see any difference for different numbers of server or client threads?

Overall, you are responsible for coming up with a reasonable test plan. You do not have to demonstrate that many threads are better, but make sure you experiment along a couple

of axes of variation and that you give a credible explanation for your observations. This is not a one-way process: your explanation should influence your experimentation and vice-versa.

Experimentation Suggestions and Hints (these are not requirements!): Most likely you will be working on a uniprocessor, non-RAID system. The performance benefits of threading in this environment become evident only on some workloads. Since your server is multithreaded by nature (it has the boss thread and at least one worker), the difference between 1 worker thread and 30 may be minimal (e.g., < 10%). (You are already getting a performance boost compared to the case of a single thread that both accepts connections and services requests.) To see significant benefits from using more threads, you need substantial I/O, but also some CPU activity to execute while other threads are doing I/O.

I/O is a little tricky: since you are making repeated requests, your files are likely cached, so no disk I/O may occur. If you make more files (e.g., I tried my client with 2000 copies of the `index.html` file from the CoC web page), you can be sure to get a lot of disk I/O in your workload. If you are working on the console, you will be able to tell right away whether your requests are serviced from the disk or not, by just listening to the disk noise. Now, if you make sure that your server threads have something to do on the CPU (e.g., you do a floating point calculation on each byte of the retrieved file), you will notice that your performance varies a lot depending on the number of worker threads in the server.

There are other ways to exploit concurrency by balancing the activity of different subsystems. For instance, you can try to have your client access a few (20-30) large files (e.g., 2MB or more). (Why does this (not) work?) Of course, for a really dramatic increase in the observed performance, you would need a multi-CPU machine with a RAID, but do **not** try your experiments on server machines that other students are sharing!!!

III DELIVERABLES
You should turn in the following **by email to me**:
- Your code. Make sure it is obvious how to compile and run everything (i.e., supply a README file).
- In your README or report, give the name of a specific public GT machine where everything has been tested and works.
- A report of your observations, answers to questions I asked above, etc. Use specific performance numbers to justify any claims you make. This is clearly an open-ended assignment, but, as a rule of thumb, you can get full points if you are brief but accurate.
- Use as subject [CS4210] Project 1.

Good luck! Have fun!!!