Project 2 (due 3/27/2006, 11:59PM)
An Optimized Skeletal Web Proxy Server

In this project, you will use your web server code as the basis for a proxy server. The proxy server will have no functionality (yet!) other than being an intermediary between a web server and a web client. The proxy server will have to be optimized so that it can work very efficiently when communicating with your own web server running on the same machine.

I GENERAL INSTRUCTIONS

- Read the entire instructions carefully before you start implementing anything!
- You may work individually or in teams of 2. If you team up, be sure to mention what code you used from each team member's Project 1.
- Be warned: this project will require more independent study of technical material than Project 1. This tends to be time consuming. Also, you have to decide on the design of your system.
- Don't get stuck! If something is hard to implement the way I suggest, approximate it and go on!
- If you have any questions, ask me! Use the newsgroup for broad questions.

II DESCRIPTION OF PROJECT STAGES

The project consists of four stages. The stages are not balanced! (Stage 2 should take longer than the rest.)

1. Web Proxy Server and Proxy Capable Client
Using your web server code, implement a multithreaded web proxy server. (You should have as much code as possible in source files that are shared between the web server and the proxy server.) A proxy server is an intermediary between a web server and a client (the client could be another proxy server, however). That is, the proxy server receives requests for pages that would normally be served by different web servers. The proxy server will then need to contact the appropriate server and receive the contents of each page before it can serve them to the client. (Of course, the proxy server could have cached pages from previous requests, but caching is beyond the scope of this project.)

The only kind of HTTP request you need to support in your proxy server is a GET request. Note, however, that a GET request that is addressed to a proxy server has to have an absolute URI (instead of a relative URI). That is, the requested address will begin with "http:" and will specify an internet name, instead of just specifying a relative file path.

Test your proxy server with a standard browser. To set up Netscape to use a proxy, go to Edit->Preferences->Advanced->Proxies->Manual proxy configuration->HTTP Proxy. Additionally, change your web client from Project 1 to support web proxies. You can do this by adding a command line argument. For instance, your client could be executable as:

```
web_client <#threads> [-proxy proxy_addr] <URL>+
```

Make sure that your client issues requests with absolute URLs when connecting to a proxy server.

2. Efficient Web Proxy
Overview. An interesting special case is when your web proxy runs on the same machine as your web server and it receives a request for a file serviced by your web server. In this case, your web server and your web proxy server are essentially two modules of the same local application that just happen to run in different address spaces. This is a nice system structure because the role of the proxy server is conceptually very different from the role of the web server and one could run with or without the other. Web servers are responsible for serving pages (ensuring security, retrieving files from the file system, creating dynamic pages, etc.), whereas proxy servers can do higher-level page manipulation (e.g., compression, encryption, caching, etc.) without knowing anything about how pages are stored on disk or how they are generated dynamically.

When your web server runs on the same machine as your proxy server, transmitting files between them through a socket is inefficient. Instead, the two processes can use shared memory to pass file data efficiently. This optimization is what you need to implement in this stage of the project. Note that this task requires additions both to the web server and the proxy server. Make sure that you can enable and disable the optimization (e.g., use a command line argument, or a compile-time flag). This is because you will need to compare the optimized and unoptimized versions eventually.

Architecture and Technical Concerns. You are free to implement the basic functionality any way you want, but be sure to document your design in your report. Below, I give an outline of the functionality, but there is significant freedom at every step:
- The proxy server receives a request and extracts the address of the target web server.
- If the target web server is on the local machine, the proxy server checks to see if the web server is your own web server. There are two issues here. First, it is a (relatively) hard problem to check whether an internet name is a name (or alias) for the local machine. This project is not about implementing mature internet protocols, so just do something reasonable (like checking if the name maps to the IP address you know for the current host—do not just compare names!) The second issue is how to detect that the web server that will serve the page is your own. Again, any common sense solution will do. Explain the pros and cons of the solution you picked.
- The proxy server issues a command to the web server and the web server knows to return file data in a shared memory segment. There are many ways to do this, but do not cheat: the proxy server should know nothing about accessing files from disk! (Otherwise your solution is not modular and might work for static content but would not work for dynamic content.) Here is one possible implementation: The proxy server allocates the shared memory, then passes an identifier for the shared memory segment to the web server with a special-purpose request (e.g., you can invent a "LOCAL_GET" request). The web server can then supply the (optional) response header and file data in shared memory (in chunks, if the shared memory segment is

not big enough). Note that synchronization of some kind will be needed. Pthread shared mutexes are one way to do this, but there are others, too. Also note that all shared data between the web server and the proxy will have to be in the shared memory segment. Such shared data include, e.g., the size of the file data written, possibly a flag to show whether the web server has yet written the data or not, etc.

- Keep the shared memory segments around so that they can be used in multiple connections. Ideally, you can keep a shared memory segment per proxy worker thread, but you may run into problems with the maximum number of shared memory segments for a single process, which can be very small (6?) in some systems. If this is the case, then your threads in the proxy server will have to compete for shared memory segments! Limiting the number of threads in your proxy server is not an acceptable solution (unless the limit is over 50)! This would limit the concurrency in the general case just to support a specific optimization. You can implement a synchronized queue of shared memory segments instead.

The mechanism you will use to implement shared memory is standard System V shared memory system calls (`shmget`, `shmat`, etc.). A handout documenting this functionality will be given out in class, and posted online (password will be provided). There is also a Posix shared memory standard (`shm_open`, etc.), but you will not find implementations in many platforms (e.g., to my knowledge, there is a Solaris library but not a Linux one).

Extra Credit Opportunities (attempt only after you have completed everything else). The above construction uses fast inter-process communication for file transfer but not for requests. Requests are short, but the overhead of connecting through a socket every time may still be significant. Your proxy server could use a fast IPC mechanism to communicate with the web server, instead. The problem in this case is that the web server has to wait both on a network socket (`accept`) and on a "local" channel. You can use signals as a (bad) solution, but I would not recommend it due to the inherent complexity of signal handling, especially with multithreaded programs. Instead, you can either use `select` or have two boss threads in your server. One will wait for network connections and one will wait for local connections (e.g., through a System V message pipe). This will also eliminate the need for an ugly special-purpose "LOCAL_GET" request. If you implement this, make sure to compare its performance with the simpler design that uses a socket for requests and you will get up to 10% extra credit.

Also, note that essentially what you need is to "jump" from a worker thread in the proxy to a worker thread in the web server. (In fact, ideally the two processes will have the same number of worker threads.) There are special-purpose IPC mechanisms that are supposed to be good for that. Check out "doors" in Solaris (`man door_create door_bind`, etc.). If you do use doors, make sure to evaluate their effectiveness and you will get extra credit (up to another 10%).

Clearly state in your project which (if any) extra credit opportunities did you implement.

3. Termination Handling
System V shared memory identifiers are persistent with respect to the process. This means that if a shared memory identifier is not explicitly removed, it will stay in the system even after your server has exited. Shared memory segments will be created either by your web server or by your proxy server. You goal in this stage is to do a good job of cleaning up shared memory resources after the server(s) have exited. As your servers are probably programs running forever, the only way for them to exit is through termination by a signal. This could happen, for instance, by pressing Ctrl-C on the keyboard. Define and install signal handlers for some common signals, so that the server cleans up when it exits ("server" being either the proxy server or the web server, depending on which one creates the shared memory). There is no way to do a perfect job—there are signals that cannot be caught, like SIGKILL or SIGSTOP—but take care of some obvious cases. Test to see that your solution works (that is, the shared memory resources are removed).

For signal reference, read the man page for `sigaction` and follow the pointers to other man pages. See also header file `<sys/signal.h>`.

4. Experiments
Using your client, compare the performance of your optimized and unoptimized web server and proxy server pairs. Show the results of experiments (give actual numbers) and explain. Is using shared memory faster or slower? Is it a bandwidth issue or a latency issue? (Think about the kinds of experiments you need to perform to separate the two.) How do you interpret the results? (What do they tell you about the operating system and its implementation of cross-process synchronization and/or sockets?) Also, compare the performance of the web server and proxy server pair to just the web server. How much slower does the proxy server make the system?

III DELIVERABLES

You should turn in the following:
- Your code. Make sure it is obvious how to compile and run everything (i.e., supply a README file). Send everything as in Project 1 (email a code archive me).
- A report outlining your design choices and the results of your experiments.
- A brief explanation of what each team member did.

Good luck! Have fun!!!