6.824 - Fall 2002

6.824 Lab 2: A concurrent web proxy

Introduction

In this lab assignment you will write an event-driven web proxy to learn how to build servers that support concurrency.

For clarity's sake, we define some terms. The *proxy-server* is the part of your proxy that receives requests from a client, i.e., the part that `accept`s a connection, `read`s the request, and, later on, `write`s back the reply. The *proxy-client* is the part of your proxy that talks to a server, i.e., the part the `connect`s to the server, `write`s the request, and then `read`s the reply.

A rough definition of what it means for your proxy to be concurrent---we'll refine it later---is that, if your proxy is waiting for some event on behalf of one request, it should be able to process other requests. Note that this means that requests may be satisfied in a different order than the order in which they were started. For example, if client A connects, but hangs a few seconds before writing its request, client B could connect, write its request and have its reply back in the time that client A was hanging there. In the Design Requirements, you'll find a more precise definition of what concurrency means in this assignment.

Design Requirements

You should make sure your web proxy satisfies the following requirements.

Consider the following list of events that your web proxy must handle:

- Arrival of a new connection,
- Arrival of some or all of a request from a client,
- The completion of a `connect()` to a server, and
- Arrival of some or all of a response from a server.

While waiting for any of these events for one connection, your web proxy must be able to handle any of these events for other connections.

It is OK if your web proxy blocks in these functions:

- `write`, and
- `gethostbyname`.

In addition, your web proxy should fulfill the following requirements. These are similar to the requirements from the first lab; differences are marked in red.

- GET requests work.
- Images/Binary files are transferred correctly.
- Your web proxy should properly handle Full-Requests (RFC 1945, Section 4.1) up to, and including, 65535 bytes. You should close the connection if a Full-Request is larger than that.
- You must support URLs with a numerical IP address instead of the server name (e.g. http://18.181.0.31/).
- You are not allowed to use `fork()`.
- You may not allocate more than 100MB of memory.
- The restriction of not having more than 32 open file descriptors no longer applies. You can have as many file descriptors as the system allows.
- Your proxy should correctly service each request if possible. If an error occurs, and it is possible for the proxy to continue with other clients or subsequent requests, it should do so. If an error occurs from which the proxy cannot reasonably recover, the proxy should print an error message on the standard error and call `exit(1)`. There are not many non-recoverable errors; perhaps the only ones are failure of the initial `socket()`, `bind()`, `listen()` calls, or a call to `accept()`. The proxy should never dump core except in situations beyond your control (e.g. a hardware or operating system failure).

You do **not** have to worry about correct implementation of any of the following features; just ignore them as best you can:

- POST or HEAD requests.
- URLs of any type other than http.
- HTTP headers, other than the first line with the `GET` request.

If your proxy consistently passes all of our tester's tests (see below), you're done and we'll give you an A. Assuming you handed it in on time. The tester for this assignment is not the same as the tester for the first lab.

Getting started

We have provided a skeleton directory. It is available in `/home/6824/labs/webproxy2.tgz`. The following sequence of commands should yield a compiled version of the web proxy you should extend to pass the tests.

```
% tar xzvf /home/6824/labs/webproxy2.tgz
% cd webproxy2
% gmake
```

The tarball contains `http.C`, `http.h`, `Makefile`, and `webproxy2.C`. `webproxy2.C` contains some example code that the discussion below refers to.

Things you need to know

Recall that the `read` system call will block until there is some data to read. This means you cannot simply use `read` in your concurrent web proxy unless you know for sure that there is

data to read. If you don't watch out, your web proxy might block on a `read` and, *poof*, gone is concurrency.

The `select` system call is your big friend in this assignment. The main point of `select` is that it will tell you which file descriptors can be `read` without the risk of blocking on them. If you use `select` prudently, you will never block on a `read`, which is the first step to concurrency.

You should use `select` in an *event loop*. An event loop is simply a `while()` loop like this:

```
while(1) {
  select(...);   /* blocks until some fd is ready to read */

  /*
   * read the appropriate file descriptors
   * and do whatever is necessary to make progress
   */
}
```

This is how you should structure your web proxy, also. Notice that there's only one event loop with one single `select` in your entire program.

Constructing parameters to `select` is a little tricky, and you should consult the "Using select" article for the details. The example code in `webproxy2.C` contains an event loop with `select`.

Notice that we haven't mentioned `write` yet. The short answer is that, as mentioned above, it is OK if you call `write` and it blocks.

That's it? Almost. Unfortunately, there's one slight problem left. Recall that you need to `connect` to a server to fetch the requested web page. What happens when that server is down? Or what if, because of network problems, establishing the connection takes a long time? Your web proxy will be blocking on the `connect` call thereby taking away concurrency!

Solving this involves some nasty UNIX trickery. After you create the new socket, but before you call `connect`, you should set the `O_NONBLOCK` flag on the socket, using `fcntl`. This is what `set_nonblock` in `webproxy2.C` does. After that, you can call `connect`. However, because of the `O_NONBLOCK` flag, `connect` will return immediately, whether it really connected to the server or not! There's several things we need to do:

- if `connect` returns -1 and sets `errno` to something else than `EINPROGRESS`, then `connect` has failed and you should handle it as best as you can,
- if `connect` returns -1 and sets `errno` to `EINPROGRESS`, then things are up in the air and we need to wait for `connect` to finish---wait, but not block! You achieve this by having the `select` in your main event loop also check for writability on the file

descriptor used in `connect`. Once `select` tells you that the file descriptor is writable, we know that `connect` completed. Now we still need to find out whether or not `connect` was able to succesfully establish the connection. Who knows; maybe it failed. The correct way to do this is to call `getpeername` (see man page) on the socket. If `getpeername` returns -1, you know that the connection could not be established and you should handle the crisis as best as you can. If `getpeername` returns 0, then things are fine, and you can start writing stuff to the server, but not before **you disable the `O_NONBLOCK` flag** on the file descriptor. You can use `clear_nonblock` in the example code to do this.

Function `do_connect` in the example code shows how to `connect` to a server; `connect_ok` shows how to check whether the connection was established successfully.

You may want to consult "Using select" article for more details.

Running and testing the proxy

Just as in the first assignment, your web proxy should take a port number as its only argument.

When you think your proxy is ready, you can run it against the test program `webproxy2-test`, our tester, the source of which is in `~6824/labs/webproxy2/webproxy2-test.C`. Run the tester with your proxy as an argument:

```
% /home/6824/labs/webproxy2/webproxy2-test ./webproxy2
```

Note that this may take several minutes to complete. The test program runs the following tests:

Ordinary fetch

This test is the "normal case". We send a normal HTTP 1.0 GET request and expect the correct web page.

Fetch by name

This tests whether you can handle a server name in the request, rather than an IP address. We send a normal HTTP 1.0 GET request and expect the correct web page.

Split request

This tests sends the HTTP request in two chunks.

Large request

The tester sends a large request; the request contains a large header that causes it size to be exactly 65535 bytes.

Large response

The tester fetches a web page larger than the maximum amount of memory available to your web proxy.

Zero-size response

The tester fetches a zero-length web page.

Recover after bad connect

The tester sends a request with a URL that specifies a port on which no program is listening; this should cause connect() to fail. The tester makes sure your web proxy handles that failure by sending a subsequent normal request.

Malformed request

The tester sends an HTTP request that is not syntactically correct. After that, it tries to fetch a valid page to see if it your proxy is still doing ok.

Premature client close()

The tester sends a partial HTTP request and then closes the connection. After that, it tries to fetch a valid page to see if it your proxy is still doing ok.

Infinitely long request

The tester swamps your proxy with a request larger than 65535 bytes. The tester expects your proxy to close the connection. After that, it tries to fetch a valid page to see if it your proxy is still doing ok.

Blocking server, concurrent request

The tester starts a request for a web page from a server that will block for 15 seconds halfway through sending its reply. 2 seconds later, the tester starts a request for a web page from a normal server. The reply for the second request should come back first.

Silent client, concurrent request

The tester starts a request for a web page, but waits 15 seconds before actually finishing the request. Meanwhile, another client does a normal request. The reply for the second request should come back first.

Infinite server, concurrent request

This test has been revoked.

The tester starts a request for a web page from a server, whose reply never stops. While your proxy is dealing with this constant data stream, it has to handle a second, normal, request from the tester.

Unresponsive server, concurrent request

The tester starts a request for a web page from a server that cannot be reached. While your proxy is waiting to connect to the dead server, it has to handle a second, request from the tester.

Stress test

The tester starts 2 concurrent requests from the Infinite server (see above) to keep your proxy busy, and then expects your web proxy to handle a barrage of normal requests, split requests, malformed requests, and requests with large responses all at the same. A part of these requests is done to the Blocking server that waits 15 seconds halfway through the reply.

Collaboration policy

You must write all the code you hand in for the programming assignments, except for code that we give you as part of the assigment. You are not allowed to look at anyone else's solution (and you're not allowed to look at solutions from previous years). You may discuss the assignments with other students, but you may not look at or copy each others' code.

Handin procedure

You should hand in a gzipped tarball `webproxy2-handin.tgz` produced by `gmake dist`. Copy this file to `~/handin/webproxy2-handin.tgz`. Set permissions for this file to 600 (`chmod 600 ~/handin/webproxy2-handin.tgz`). We will use the **first** copy of the file that we can find after the deadline---we try every few minutes.