Assignment 6

April 6, 2015

Preliminaries

For this assignment, you will add 3 features to your HTTP server:

- socket listening
- multi-process support
- access logging

You may start from either your A6 solution or mine.

Your final program will be called `forking-httpd`, and should live in the file `forking-httpd.c`. The provided `Makefile` will automatically build `forking-httpd` when `forking-httpd.c` is present.

Your new `forking-httpd` program will need to parse additional command line arguments with `getopt`. As before, it takes a single non-option argument the path to the document root, the directory from which web pages are to be served. Parts A and C add additional options.

The program should exit with code `EX_USAGE` (from `sysexits.h`) if an invalid command line option is provided.

Part A: Socket Listening (75 points)

For the first part, make the new HTTP server accept HTTP connections on a socket instead of on standard input. The HTTP handling is exactly as it was for A4; the only difference is that the new program, rather than reading a request from standard input and writing the response to standard input, will receive connections on a socket.

Your program should take two optional command line options, `-h` and `-p`. `-h`, if provided, specifies the hostname (or IP address) to which to bind the listening socket; if not provided, bind to '::' (in `AF_INET6`). `-p` provides the port; if no `-p` option is provided, use port 4080.

Your program needs to do the following:

1. Parse the command line options
2. Create a socket
3. Bind it to the specified host and port (use `getaddrinfo` to resolve them to a socket address)
4. Listen and wait for connections on the socket
5. For each connection received, read an HTTP request from the resulting socket, write the HTTP response back to the socket, and close the client socket.

Run your program by running `./forking-httpd` with appropriate options. It does not need `socat` or `listen.sh`.

Part B: Forking (15 points)

The Part A server now listens (and does not need `socat` to do so). However, it can only respond to one request at a time. This makes it very slow and unresponsive.

Modify your program to fork a separate process for each connection. That is, in step (5) above, rather than reading the request and preparing the response in the main process, your program needs to fork a new child process for the request.

The parent process should close the client socket and wait for the next incoming socket connection.

The child process should close the server socket (since it doesn't need it), read the request from the client socket, write the response, and exit.

The parent process does need to make sure that it reaps its children, or arrange for them to be reaped. This can be done by setting a flag of `SA_NOCLDWAIT` for the `SIGCHLD` signal with `sigaction`. We will talk about this in class.

Part C: Logging (10 points)

It is useful to maintain logs of the visits to a web server. For part C, modify your web server to take a new command line argument, `-l`, that specifies the path to a log file. For each request, record the following in the log file:

1. The time, as an ISO-style timestamp (2015-04-07 03:20:22).
2. The IP address and port from which the connection came (make this human-readable with `getnameinfo`).
3. The requested path.
4. The value of the `Referer` header, if one is present.

Now, there is one problem with this. The full request data is only available in the child processes. However, if each child process just writes to the output file, the output lines may get jumbled up. We need a way to make sure that lines are written one at a time.

The way we will do this is to have each child log to a *logging socket* instead of writing its logs directly to the log file. This should be a datagram (`SOCK_DGRAM`) socket in `AF_UNIX` (see `unix(4)` for some detailed documentation on that). You can create the socket address for such a socket as follows:

```
struct sockaddr_un log_addr = {
    sizeof(struct sockaddr_un),
    AF_UNIX,
    socket_path
};
```

For the socket path, use `/tmp/httpd.log.UUUU.PPPP.sock`, where `UUUU` is the current user's user ID (call `getuid` to get this) and `PPPP` is the process ID of the master HTTPD process (call `getpid` to get this).

By using a datagram socket, we can make each log entry a single datagram; the log listener will then receive each datagram and write it out as a line in the log file.

For the logging, you will need to write code to do the following:

1. In request-handling processes, open a connection to the logging socket, write the log message as a single datagram using `send`, and close the socket.
2. In the log-writing process, open a server socket and bind it to the socket pipe. Then wait for datagrams with `recv`, and write each one to the log file.

There are 2 different designs you can use for part 2. One is to make the master httpd that listens for server connections also be the logging process. In this design, you will need to use `select` to wait for incoming connections and for incoming log messages.

The other design is to have the main process fork a separate log listening process. This log listening process should listen for the datagrams and write them out to the log file. The master process should stop the logging process (by sending it the `SIGTERM` signal using `kill`) when it shuts down. This means that it should trap the standard shutdown signals (`SIGTERM` and `SIGINT`) to arrange for a clean shutdown.

Either design is acceptable. Implement whichever you think will be easier.

Log File Rotation (5pts extra credit)

When the log file grows past 2MB, close it, rename it to a new file (the original file name with the current date and time appended; e.g. if the log file is `access.log`, rename it to `access.log-20150125T153911`), and open a fresh log file.