Httpserver - A Simple Server Implementation

Compiling

httpserver compiles on Ubuntu VM with no errors or warnings.

Testing

Testing of httpserver was done in parts.
First I implemented many threads and had them each just print something to see they were there.
Then I implemented the actual multithreading so they can individually respond to client requests. To simulate multiple clients I created a bash file and placed various calls for PUT, GET, and HEAD via curl with an & at the end of which one so they run simultaneously.

Example:
```
curl -T longfile localhost:8080/longfile &
curl -I longfile localhost:8080/longfile &
curl localhost:8080/longfile &
curl -T abc localhost:8080/abc &
```

I then implemented HEALTHCHECK (valid and invalid calls). I tested it with curl commands.

Example:
```
curl -I localhost:8080/healthcheck &
curl -T healthcheck localhost:8080/healthcheck &
curl localhost:8080/healthcheck &
```

Lastly I implemented the logging mechanism and checked it with the previous scripts.

Testing concurrent handling of requests, with errors and large files:
```
for i (1..100)
do
      curl -T longfile localhost:8080/longfile >&/dev/null &
      curl -I longfile localhost:8080/longfile >&/dev/null &
      curl -I longfile localhost:8080/nofile >&/dev/null &
      curl -I longfile
localhost:8080/filenameverylong11111111111111111111111111111111111111111111111111111111111
1111111111111 >&/dev/null &
      curl localhost:8080/longfile >&/dev/null &
      curl -T abc localhost:8080/abc >&/dev/null &
      curl -I longfile localhost:8080/no_r_permission >&/dev/null &
      curl -I localhost:8080/healthcheck >&/dev/null &
      curl -T healthcheck localhost:8080/healthcheck >&/dev/null &
      curl localhost:8080/healthcheck >&/dev/null &
done
```

<u>Then came implementation of errors:</u>
Healthcheck errors:

I realized I counted the logged requests twice.

Logging errors:

I realized I calculated the needed space incorrectly when the number of characters was divisible by 20.

When logging 400 errors, I initially used the same approach as when logging other errors. I then realized that I could not rely on the command or the file name used. Instead I had to use the buffer itself up to the first EOL if it exists.

Performance issues:

I notice that logging was responsible for a vast majority of the run time. This was because I wrote to the logfile with pwrite 1 line at a time. So instead I kept the data in a large buffer and wrote it as infrequently as I could.

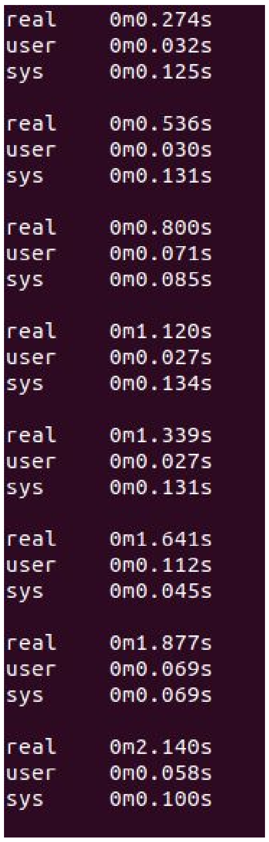
I initially used the unix queuing mechanism, but came to the conclusion that connections might fail sooner than I would like. So I ended up implementing my own queue.

Questions

Run GET 8 times on a file that is 400 MiB long with and without multithreading. Is there any difference in performance? What is the observed speedup??

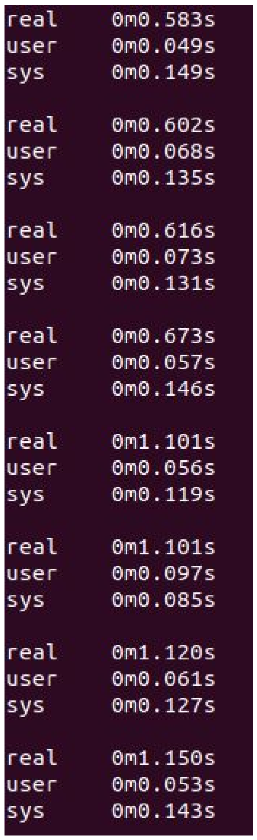
As can be seen in the images below, the implementation without multithreading handled the first few calls faster. In fact it managed to handle the first two GETs before the multithreading managed to finish any of its calls. However, by the end the implementation with multithreading finished in 1.15 seconds while the implementation without multithreading finished in 2.14 seconds.
So, yes there is a difference in performance. For just 2 calls the multithreading is less efficient, but any more than that and it becomes more and more efficient because it is able to handle the requests simultaneously.

No multithreading



```
real    0m0.274s
user    0m0.032s
sys     0m0.125s

real    0m0.536s
user    0m0.030s
sys     0m0.131s

real    0m0.800s
user    0m0.071s
sys     0m0.085s

real    0m1.120s
user    0m0.027s
sys     0m0.134s

real    0m1.339s
user    0m0.027s
sys     0m0.131s

real    0m1.641s
user    0m0.112s
sys     0m0.045s

real    0m1.877s
user    0m0.069s
sys     0m0.069s

real    0m2.140s
user    0m0.058s
sys     0m0.100s
```

With Multithreading



```
real    0m0.583s
user    0m0.049s
sys     0m0.149s

real    0m0.602s
user    0m0.068s
sys     0m0.135s

real    0m0.616s
user    0m0.073s
sys     0m0.131s

real    0m0.673s
user    0m0.057s
sys     0m0.146s

real    0m1.101s
user    0m0.056s
sys     0m0.119s

real    0m1.101s
user    0m0.097s
sys     0m0.085s

real    0m1.120s
user    0m0.061s
sys     0m0.127s

real    0m1.150s
user    0m0.053s
sys     0m0.143s
```

What is likely to be the bottleneck in your system? How much concurrency is available in various parts, such as dispatch, worker, logging? Can you increase concurrency in any of these areas and, if so, how?

I believe the bottleneck is passing the actual data when large files are involved. In the test above we can see that with multithreading that the first 4 requests took about the same amount of time and were handled in parallel, whereas the next 4 had to wait for the previous requests to be handled.

The dispatcher can work at the same time as the threads, but can only accept one request at a time and put it in the queue. The workers can run parallel. They might take CPU from each other depending on the number of threads active and the available number of processors from the machine. Logging can be done in parallel, preferably using as few calls to pwrite as possible (writing large buffers into the log in each call when possible).

The concurrency of the logging can be improved by writing into multiple files (one per worker). The information can be aggregated when inspecting the log files.

For this assignment you are logging the entire contents of files. In real life, we would not do that. Why?

In real life we would not log the entire contents because this would result in massive log files and fill the hard disk. Furthermore, it is not particularly informative, we could instead only log something like the header and the number of bytes passed.