ECE 459: Programming for Performance
Assignment 1

Patrick Lam

December 28, 2016 (Due: January 30, 2017)

In this assignment, you'll work with a program which requests a resource across the network. I've provided a single-threaded implementation which uses blocking I/O to get the resource. You will reduce the latency of this operation by sending out multiple requests simultaneously (to different machines). In part 1, you'll use pthreads to do this, while in part 2, you'll use nonblocking I/O.

Setup

After setting up your ssh key at `http://ecgit.uwaterloo.ca`, fork the provided git repository at `git@ecgit.uwaterloo.ca:ece459/1171/a1` :

```
ssh git@ecgit.uwaterloo.ca fork ece459/1171/a1 ece459/1171/USERNAME/a1
```

and then clone the provided files.

You should do this assignment in Linux, as the provided `Makefile` was only tested on Linux and isn't very robust. Use a virtual machine at your peril. It might work OK since the program's bottleneck is the response time of the remote execution.

You may log into `ece459-1.uwaterloo.ca` with the ssh key that you set up at `ecgit`. Use your uwuserid with that ssh key. I'll be testing your solutions on that machine.

Assignment code walkthrough

You will find the file `paster.c` in the provided file. This code uses `libcurl` to fetch a set of `PNG` files from the network and `libpng` to paste the files together.

I've provided a web API which returns portions of some pictures that I took. You can see this in a browser by visiting `http://machine:4590/image?img=N`, where `machine` is one of `patricklam.ca`, `berkeley.uwaterloo.ca` and `ece459-1.uwaterloo.ca`, and where `N` $\in [1, 3]$. This API returns a 200×3000 horizontal strip, and uses an HTTP response header to tell you which strip you got.

The provided code repeatedly fetches image segments until it has them all, puts them in an array, and then produces an output file, `output.png`, with the pasted-together image.

Part 0: Resource Leaks (5 marks)

I inadvertently left a resource leak in the provided code. Resource leaks sap performance. Find it (valgrind helps), fix it, and document it in your report.

Part 1: Pthreads (45 marks)

Use the `pthread` library, create a threaded version of the provided program. Your program should create as many threads as the `num_threads` variable (which reads the value from the `-t` command line option) and distribute the work between the 3 provided servers. Make sure all of your library (standard glibc, libcurl, and libpng) calls are *thread-safe* (for glibc, e.g. `man 3 rand` to look at the documentation).

We will look at your code to ensure that it uses `pthread` calls properly, and we will execute your code to verify that it produces the correct output. Code that doesn't compile on `ece459-1` will get at most 39%.

(10 points) In your report, describe how you know that your threaded code uses only thread-safe calls, with pointers to the appropriate documents, and why your code is free of race conditions.

Also, time your executions with the serial version and parallel version (take an average of 3 runs each; for the parallel version, investigate $N \in \{4, 64\}$) and discuss how well parallelization works.

Part 2: Nonblocking I/O (45 marks)

In this part, you will write a single-threaded version of `paster` which uses nonblocking I/O to request multiple versions of the image simultaneously. You will need to use the `curl_multi` API as well as either `select` or `epoll`. Once again, distribute the work between the 3 provided servers.

Your solution should *not* use pthreads. However, it should have multiple concurrent connections to servers open. In this case, the `-t` command line option indicates the number of connections to keep open at once. The `-i` option always indicates which image to fetch.

Again benchmark your work and report comparative results. Discuss the performance of all three versions. Is it what you expected?

Alternate option. You may instead provide a client-side JavaScript solution which initiates multiple requests, integrates the results, and displays the resulting image. I'm thinking of something like a solution with `node-pngjs`. Last year, 3 students took this option. It is actually fairly easy to write this code, but you need to figure out everything on your own. I will not provide information on how to accomplish this (but come talk to me if you're interested).

Part 3: Applying Amdahl's Law and Gustafson's Law (5 marks)

The `paster_parallel` code clearly has a parallel and a serial part. In your report, estimate the number of seconds typically spent in the serial part, and explain how you arrived at that number. Discuss why Amdahl's Law and Gustafson's Law apply, or don't apply, to `paster_parallel`.

Submitting

To submit, simply push your fork of the git repository back to `ecgit.uwaterloo.ca`. We will be marking `Makefile`, `src/paster.c`, `src/paster_parallel.c`, `src/paster_nbio.c`, and `report.pdf`. (You can modify the provided

`report/report.tex` and create it with `make report`; do not submit a `doc` file!). Running `make` in the `assignment-01` folder should produce three files: `bin/paster`, `bin/paster_parallel`, and `bin/paster_nbio`.

Rubric

The general principle is that correct solutions earn full marks. However, it is your responsibility to demonstrate to the TA that your solution is correct. Well-designed, clean solutions are therefore more likely to be recognized as correct.

Solutions that do not compile will earn at most 39% of the available marks for that part. Segfaulting or otherwise crashing solutions earn at most 49%.

Part 0 (5 marks): Self-explanatory.

Part 1 (45 marks): (35 marks for implementation) A correct solution must:

- start the appropriate number of threads (5 points);
- have each thread do work, distributed among the 3 servers (10 points);
- code safety: prevent buffer overflows and clean up all allocated resources, as verified by valgrind (10 points); and
- avoid data races and produce the correct output (10 points).

(10 marks for report) 8 marks for including the necessary information; 2 marks for clarity.

Part 2 (45 marks): (40 marks for implementation) A correct solution must:

- properly initialize the `curl_multi` handle with the appropriate number of individual `curl_easy` handles (10 points);
- process results from the `curl_multi` handle (`select`/`multi_perform`/`multi_info_read` or `multi_perform_socket`) (10 points);
- replace finished handles with new requests while requests remain (10 points);
- code safety: prevent buffer overflows and clean up all allocated resources (5 points); and
- produce the correct output (5 points).

(5 marks for report) 4 marks for information, 1 for clarity of exposition.

Part 3 (5 marks): Self-explanatory.