ECE 459: Programming for Performance
Assignment 4

Jeff Zarnett & David Cheng

March 26, 2017 (Due: April 3, 2017)

In this assignment, you will be asked to simulate a multi-server application to investigate the impact of Load Balancing on the system's overall performance, but also have an opportunity to apply what you have learned in this course to speeding up a multithreaded application. A basic overview of the operation is as follows: some number of jobs will arrive periodically (according to a Poisson distribution) to the system and they will be assigned to one of N queues either by (a) random assignment or (b) round-robin (i.e., job i is assigned to queue i modulo N). Each queue will have an associated pthread that will dequeue the first job, if any is available, and dispatch it to a processing function. When processing is complete, it outputs some timing data to the results file in a specified format, giving us some information about how long it takes to complete the processing of the task.

Getting started. Fork the provided git repository:

```
ssh git@ecgit.uwaterloo.ca fork ece459/1171/a4 ece459/1171/USERNAME/a4
```

and then clone the provided files.

To submit, push C files containing your code, along with a brief report. The report should explain what you did to optimize the program, explain your load balancing strategy, and overview when load balancing is effective and when it is not.

This program takes a lot of command line arguments so you can test and play around with it. The actual evaluation will be done with something like the default parameters, but could have more queues and changes to the arrival rate etc. to accommodate:

Character	Meaning
n	Number of "servers" (queues)
a	Assignment policy (1 for random, 2 for round robin)
j	Number of jobs
b	Load balancing (0 for off, 1 for on)
l	Lambda (arrival rate parameter)
m	Max Rounds (controls variability of job size)

The output file will be comma separated values ("results.csv"), with one line per completed job. The column order:

1. Job ID
2. Arrival time (the time at which the job enters the queue)
3. Execution time (the amount of time the job took in the `execute` function)
4. Departure time (when the job's execution is finished)
5. Response time (departure time minus arrival time)

Part One: Performance

This is, after all, Programming for Performance, so you can and should attempt to speed up your code as much as you can. You may use any techniques from the course (or from elsewhere) to implement the code, as long as it produces the right output and the program still works as described and you don't leak memory (helgrind being mad about irrelevant races is allowed). The code as written works, but there are no guarantees it is optimal. So feel free to optimize it in any "legal" way.

What do I mean by legal? You can't trivialize the problem, reduce the processing time, or change job generation. So, some rules. You may not alter the `#define` directives at the top of the file. You may not change the timing information (when the values of arrival/departure time, for example, are set). You may not modify the `generate` or `execute` functions. Also, so we can test it, it's best not to alter the way the command line options work.

Some examples: you can modify the enqueue semantics, or how material is written to file, or how the fetch and execute routine runs (parallelize it, perhaps?), modify the data structures if it would make things more efficient.

The file you are modifying here is `speedup.c`.

Part Two: Load Balancing

The next part of this is implementing the load balancing routine. A high level explanation is that you need to migrate jobs from over-loaded queues to under-loaded ones. This keeps things "balanced" within some margin. You should run the load balancing technique **periodically**. How often you wish to run it is a design decision: if you run it too often, you'll waste a lot of time shuffling jobs around; not often enough and you do not see enough benefit.

Each queue uses the First-Come First-Served (FCFS) policy (i.e., all work is considered to be of equal priority).

One strategy for implementation follows, though you are welcome (encouraged) to come up with your own. At each period, measure the average length of the queues; where the length of each queue is measured by the total number of jobs currently waiting in it. If the length of a particular queue is higher than the average, then it is considered as an over-loaded queue and as such you need to migrate some jobs from the end of that queue to the under-loaded queues of that tier. This process continues till you have balanced queues (all queues are approximately equal in length, depending on how many jobs there are).

List of things I don't want you to do: (1) have a single queue for incoming jobs (as this trivializes the load balancing), (2) have > 1 thread per queue (as this can mean that threads sit with the work items and get blocked, meaning load balancing does nothing).

(Hint: you may have an infinite loop, frequently moving the same job between the queues over and over. This is something to avoid!)

This last column of the output file is important; the real goal is to minimize the **average 90th percentile response time** of all jobs. You need to run the system for the following 4 cases and find the average 90th percentile response time of all jobs in each case:

1. Random assignment, no load balancing
2. Round robin assignment, no load balancing
3. Random assignment, with load balancing
4. Round robin assignment, with load balancing

The file you are modifying here is `loadbalance.c`. After completing part one, you might want to move some or all of your changes into this second file. Or, you could make completely different ones.

Evaluation

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. Segfaulting or otherwise crashing solutions earn at most 49%.

Your program will be evaluated over several runs to prevent bad luck on a single run from being the deciding factor. Your program should not leak memory, nor should it have data races that result in invalid or incorrect output.

Part 1: Performance

Average 90th Percentile Time Reduction (25 marks) The more efficient your implementation (i.e., the lower the average 90th percentile response time), the better your grade in this section. You are going to be evaluated against the other students in your class. It's simple; the faster you make your program, the more marks you'll get. If you can achieve a speedup of at least 1.2 then you will get at least 15 of the marks here (after that the marks are relative).

Report (15 marks) 12 marks for explaining how you sped up the program (1-2 pages is appropriate length); 3 marks for clarity of exposition.

Part 2: Load Balancing

Implementation (20 marks) A correct implementation is important. You may use any techniques to implement the load balancing, as long as it produces the right output, and it balances the load.

Average 90th Percentile Time Reduction (25 marks) The more efficient your implementation (i.e., the lower the average 90th percentile response time), the better your grade in this section. You are going to be evaluated against the other students in your class. It's simple; the better the 90th Percentile time reduction, the better your mark. If you can get a speedup of at least 2.0 here, you will get at least 15 of the marks here (after that the marks are relative).

Report (15 marks) 12 marks for explaining the impact of load balancing (1-2 pages is appropriate length); 3 marks for clarity of exposition.