Neal Patel, Duy Nguyen

CSCE 313 – 503

October 25, 2016

Machine Problem 5 Report

**Question**: Describe what your code does and how it differs from the code initially given to you.

**Answer**: We created a buffer class which contains a lock for the buffer, an instance of the buffer (specifically, the request buffer, itself), and the push and pop functions. Since the PARAMS struct is required, we implemented the PARAMS struct by instantiating it with a name, the number of requests per person, the number of worker threads to be utilized, a request buffer (which is a pointer to the original buffer since only one instance of the buffer should exist), and a mutex, which is in charge of locking and unlocking access. We also created a similar struct for the worker threads, which contains similar aspects as the PARAMS struct. In our worker thread function, we realized the worker thread is responsible for popping each request from the buffer and retrieving the response, where each worker thread has its own worker channel. Our code simply loops through that process and locks and unlocks where necessary to configure it for safe client-server interaction. Our request thread function is pretty short; all it does is use the passed in void object (PARAMS object) to push the data into the buffer. The additions we made to the main were to use our classes and structs instead of what the main initially had (plain lists and vectors). We also used the gettimeofday() function to properly make a graph which depicts the client program’s running time.

**Question**: Make a graph that shows how your client program’s running time or n = 1000 varies with the value for w. Describe the behavior of the client program as w increases. Is there a point at which the overhead of managing threads in the kernel outweighs the benefits of multi-threading? Also compare (quantitatively and qualitatively) your client program’s performance to the code you were originally given.

**Answer**:

Graph:

The w markers are as follows: 1, 10 (increment by 10 until 190, that go to 195, which is max possible). When there is only one worker thread, the time is significantly higher than when the number of worker threads is greater than 1. As w increases, time decreases steadily up until when w is 20, which is when w starts to stay almost constant and fluctuate by minor amounts. Generally, overhead is our case is the time it takes to finish a specific number of requests with a specific number of threads versus a fluctuated number of requests with a lower number of threads. From testing, at w= 20, the overhead of managing the threads in the kernel outweighs the benefits of multi-threading. Comparing the run times with the given code and our code, our program executes much faster than the execution with the initial code, meaning the performance is more efficient with our code.

**Question**: What is the maximum number of threads that the host machine will allow your client program to create before throwing an error? What does the operating system do when your client program tries to create more threads than allowed? How does your client program behave in response?

**Answer**: The maximum number of threads the host machine will allow is 195. The operating system shows the back trace and the memory map when trying to create more threads than allowed. Our program displays an error message in order to let the user know that the host system can only accept up to 195 threads.