Programming Assignment 3: UNIX Threads

Due 10/21/18 at 11:59pm

1 Introduction

For many computing tasks it is sufficient that one instruction be executed immediately after another, sequentially, until the program completes. However, in the case that many tasks need to be executed at once, or that a single task can be appropriately split into sub-tasks, a program's performance can be greatly improved by working on tasks concurrently or in parallel through the use of threads. Sometimes it works just as well to use multiple processes, or multitasking, instead of multithreading, but there are many benefits that result from multithreading.

2 Description

You are given a starter code directory. If you build the code using `make`, 2 executable files are created: `dataserver` and `client`. Here, the `client` runs both executables as parent-child processes using `fork()`. The client process then communicates with the server process using "named pipe" IPC mechanism encapsulated in class `RequestChannel`, which you can use without modification. Now, the client sends a few requests through `RequestChannel` to the server, which responds with some data. There are 3 persons in this system: John Smith, Jane Smith and John Doe. Every request has the format "data <person name>" (e.g., "data John Smith"). The server responds to each such request with a number in range $[0, 99]$ generated randomly. This emulates collecting data from a real system that keeps patient vitals or some other important piece of information, and the client is a small device that shows those data after collecting them from the main data base.

You can `make` the project and run `./client` (with no arguments) to observe execution and behavior of the system. The client process calls `fork()` and runs the dataserver program. Then, it populates a `SafeBuffer` object with 300 requests, 100 each for three hypothetical "users": John Smith, Jane Smith, and Joe Smith. This is done using one single-threaded process. The command-line argument, `-n`, (the getopt loop is provided for you in this assignment) allows you to specify the number of requests *per person*. For example, typing `./client -n 10000` would cause 30000 requests to be generated (10000 per person). Since the program is single threaded, you should observe that the program will run much slower with 100 times more work to do.

In large systems, it is quite possible that the number of requests could grow exponentially. Websites like Amazon receive around 80 million requests daily. Other companies such as Google and Youtube, receive requests numbering in the hundreds of millions. No single process will ever be able to this volume of traffic. So, how do the worlds largest businesses handle it in stride? Instead of making one process do all of the work, multiple processes handle data requests concurrently. In doing so, it is possible to handle large quantities of data since one can continue to add more cores/process threads to the system as load increases.

3 Assignment

1) In this assignment, you are to increase the efficiency of the client-server program given to you by using multithreading. The command line option, `-w`, (look at the getopt loop in the starter code) takes a number for an argument, which defines how many worker threads your program will utilize. Your program must spawn that many worker threads successfully (see requirements below), and these threads must work together to handle each of the $3n$ requests.

2) Multithreading often requires the use of specialized "thread-safe" data structures. In order to facilitate multithreading, you will modify a few classes (e.g., `SafeBuffer` and `Histogram`). `SafeBuffer` can simply wrap around an STL data structure (we recommend `std::queue`), but must have at least a constructor, a destructor, and the operations `push()` and `pop()`. The data going through the `SafeBuffer` must be in FIFO order (see requirements below). Both functions must use locks to ensure that the buffer can be modified concurrently at either end (concurrent modification at *both* ends requires *semaphores*, which we will tackle in the next assignment). The `Histogram` object may be updated simulatenously as well. Therefore, you must make that class thread-safe as well.

In addition to processing the requests using multithreading, and coding a specialized data structure for that purpose, the request buffer must also be initially populated using multithreading. However, writing the code for request threads will be much easier than for the worker threads: you will only ever have 3 request threads, one for each user (John, Jane, and Joe from earlier; note we said 3 request *threads*, not 3 request *thread functions*), and each one simply pushes n requests (corresponding to its respective user) to `request_buffer`. Furthermore, the request threads do not have to run concurrently with the worker threads (and in fact *must* not), which means that the only lines of request-thread-related code in main (apart from setting up the thread parameters) will be three `pthread_create` calls followed by thread `pthread_join` function calls.

To complement and clarify what is said above, your program should have the following characteristics and satisfy the following requirements:

- The program should function correctly for any number of threads. This does not mean that you have to handle all the errors resulting from large numbers of threads, but only that high numbers of threads do not in and of themselves affect program correctness. Practically speaking this means that your synchronization is provably correct, but not necessarily that your error-handling is robust. You will not have to test your program with more threads than the minimum of $(fd\_max - 2)/2$ (fd_max is the maximum number of file descriptors that a single process can have open at the same time) and however many threads causes `pthread_create` to fail. The value of fd_max

on your system can be checked (on Unix systems) with "ulimit -n" at the command line, while the number of threads that causes `pthread_create` to fail may need to be determined experimentally. Note that both values differ greatly between operating systems (sometimes even between different users on the same operating system...).

- No fifo special files (the named pipes used by the RequestChannel class to communicate between the client and server) should remain after running the client program with any number of threads.
- The client program should not take an unreasonably long time to execute. You can use the sample code as a performance baseline.
- All requests for the program must be handled correctly, which includes being processed in FIFO order. The histogram should report the correct number of requests and should NOT drop any requests.
- Your code may not make use of *any* global variables, as they (in most circumstances) exemplify poor programming practice. The easiest way to avoid this is by using storage structs (more on those later).

4 Deliverables

- **Code:**
  - You are to turn in one file ZIP file which contains the files provided in the sample code, modified to fulfill the requirements of the assignment. Along with this, turn in the report (described below) and any other code that your program requires to run.
  - If your program requires specific steps to compile and run, please provide a README file that describes these steps.
- **Report:**
  - Once you've finished all programming tasks, author a report that answers the following questions about your code:
    1. Describe what your code does and how it differs from the code that was initially given to you.
    2. Make a graph that shows how your client program's running time for n = 10000 varies with the value for w. Include at least 20 data points (more or less evenly spaced) starting at 5 and going to the highest number that will run (*without* reporting some kind of error) on your OS. After making the graph, 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 multithreading? Also compare (quantitatively and qualitatively) your client program's performance to the code you were originally given.

   * Note: Timing must be done internally by your program, rather than by the shell it's running in. A couple of good options for this are clock_gettime() and gettimeofday() (see corresponding manpages: man 3 clock_gettime and man 2 gettimeofday).
   * Note: You may find that the provided test.sh script is a good starting point for gathering data. Feel free to modify it as you find necessary.
3. Describe the platform that you gathered your timing data on (I.e. CSCE Linux server, Raspberry Pi, personal computer, etc...). For any device other than the departmental servers, briefly describe the operating system that it runs (and if you used one of the departmental servers, just say which one you used). Also answer the following sub-questions:
   * What is the maximum number of threads that the host machine will allow your client program to create before reporting an error? What error is reported?
   * 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?

Notes Concerning Global Variables and Storage Structs

Global variables are generally regarded as poor coding practice but we realize that some students haven't yet been taught how to avoid using them in a multithreaded program, where they seem to be a very easy and attractive option. Here, then, is the way we have found easiest and most effective to avoid using global variables: storage structs. (we use "struct" here for our explanation but note that, in C++, "class" works just as well)

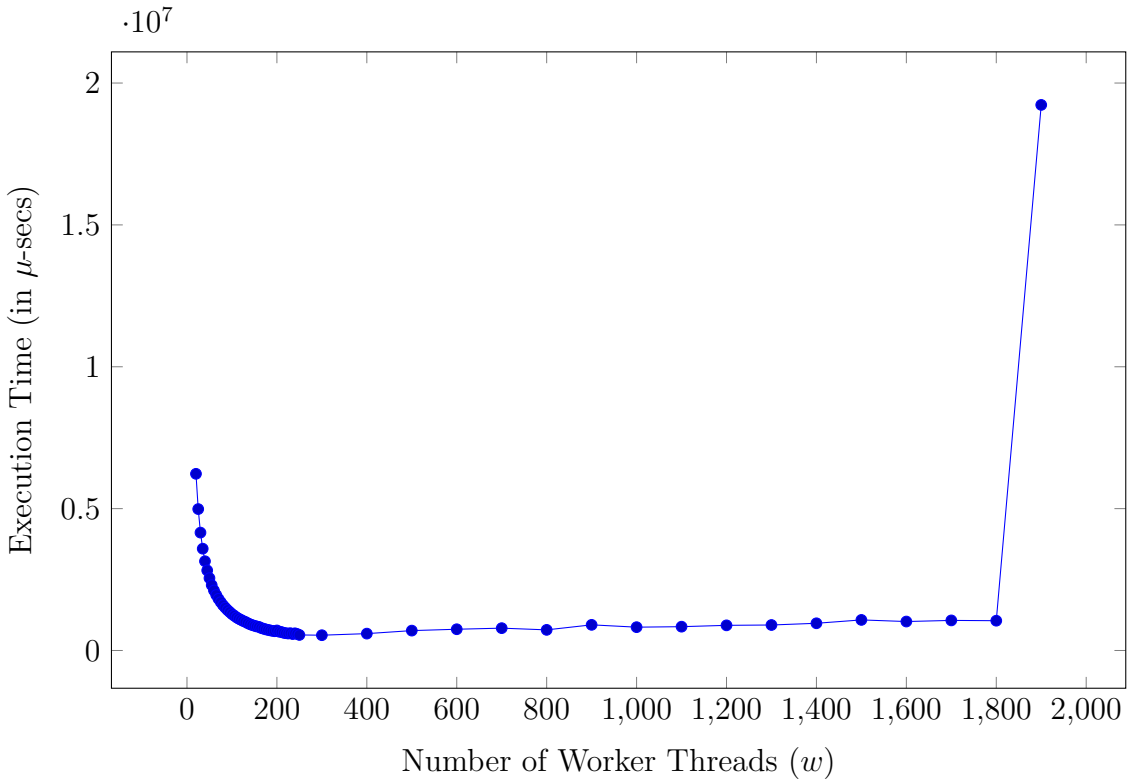
A storage struct is an old-fashioned C struct, which serves mainly to store and organize different variables. An instance of such a struct can be passed to a thread function as the last argument given to `pthread_create`. The thread function then accesses its arg parameter, a void pointer, and casts it back to the type of the storage struct that is being used, and is then able access all the struct's fields and methods. If a programmer cleverly defines the struct's fields based on the needs of his program, he has a very clean and efficient way to pass parameters to a thread function.

Instances of storage structs can be declared and initialized in main, and any memory allocated for them is valid so long as it remains in scope. This means that pointers to storage structs that have been allocated on the stack in main, and to their data fields, can be passed to thread functions and will be valid until they go out of scope. If heap memory is used, it will be valid until explicitly deleted. Pointers and storage structs can thus be used to solve the scoping problems previously solved by global variables, making the latter unnecessary.

You may find it helpful to go over the example code in the man page for `pthread_create`, since it includes an example of storage struct usage (look for the `thread_info` struct.

Notes Concerning the Graph for your Report

Figure #2: Sample Timing Data



This graph was created using data gathered on a MacBook Pro 2011 running OS X Sierra, with n held constant at 10000. You will not be expected to gather this many data points (there are more than 60 in this graph), or run anywhere near as many as 1900 threads, for your report: gathering this much data required increasing the number of available file descriptors as root, as well as more error-handling code that you will be expected to write.

It does, however, demonstrate the expected behavior as w increases, and is intended to encourage you to test your program with larger numbers of threads than you might initially be inclined to. After all, the report requires you to test right up to the default limitations of your OS (but no further).

If, in gathering data for your report, you find that the default limitations of your OS (namely, available file descriptors and thread creation resources) prevent you from building a graph that demonstrates the full range of expected behavior (i.e. performance increases, then flattens out, then suddenly becomes horrible), not to worry: you are only required to run as many threads as your OS will allow *by default*. But you will not be able to complete Part 3 of the report by doing any less.