Machine Problem 6: UNIX Threads (Due: 4/7/17)

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.

Threading Background:

How do threads differ from processes?

A process is a container for code in execution. It consists of an address space which houses program code and all program data (See Advanced Concepts section for further information). In UNIX/Linux operating systems, processes are maintained as completely separate execution units. Any given process knows absolutely nothing about other processes executing on the system, and processes can only communicate using one of the IPC (Inter-Process Communication) mechanisms offered by the kernel (Named/unnamed pipes, shared memory regions, etc...).

Processes are kept separate to ensure system security. One can conjure up a multitude of reasons why it would be disastrous if one process could easily access the data of another. However, given that modern processors have multiple execution units ("cores"), there is a need to provide a way for multiple instances of sequential code execution to work together.

Context switches of entire processes are extremely expensive. In order to switch one process out for another, the state of the currently executing process must be saved, and the state of the process being switched to must be loaded into registers/memory.

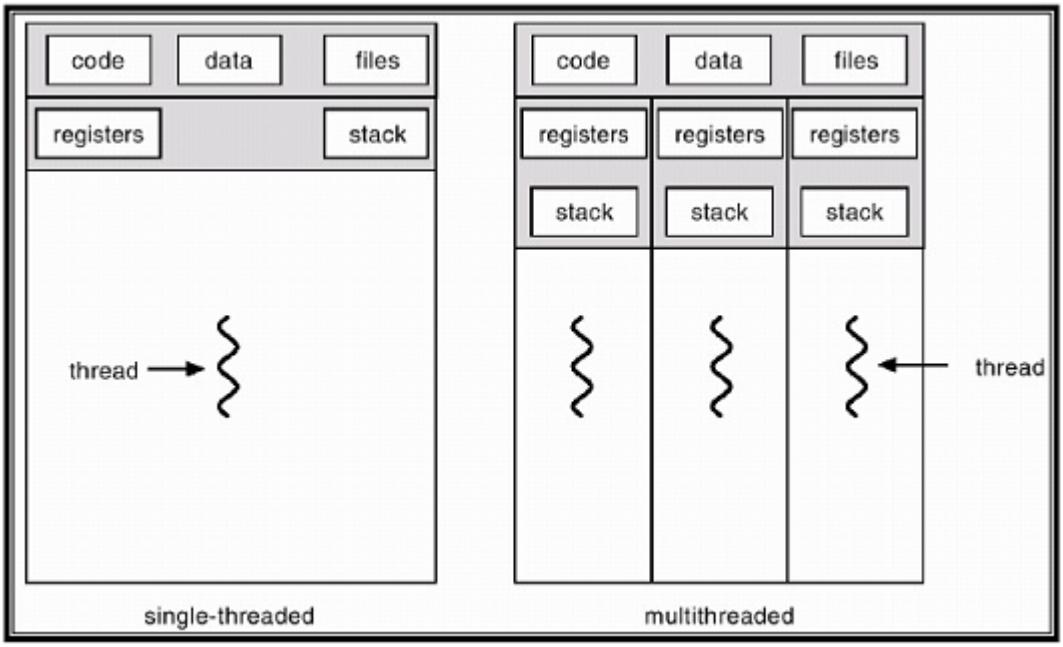
In an environment that must perform multiple tasks at the same time which all contribute to a single goal, switching back and forth between processes would be an unnecessary drain on computing resources. To address this problem, POSIX offers a library specifying light weight execution units known as threads.

Threads are initialized inside of a parent process. They execute within (share) the context of the parent process, so no full context switch is required in order to switch from one thread to another. Some data can be shared between threads executing under the same process, and some must be kept individual. The heap, program code, and open files

are shared between threads, whereas threads must have their own stack space and CPU register values.

As a result of this shared data scheme, switching between threads is as simple as switching out the stack pointer (the %esp register on x86 systems) and saving/replacing CPU register values. This takes considerably less time than switching out entire processes and makes multithreaded execution of code practical.

Figure #1: Threading Visual Representation



Advantages of Threads

The list below provides some of the most popular incentives for utilizing multithreaded programming. For further reading, there is a variety of readily available literature pertaining to the topic (The author recommends *Programming with POSIX Threads* by Butenhof).

- *Faster Execution*: Threads can take advantage of computer systems which possess multiple CPUs. Instead of being bound to doing one sequence of instructions at a time, the only limitation is the number of processors available. This scales from small personal computers to massive supercomputers.
- *Lower Resource Consumption*: Threads spare the kernel from having to do significant amounts of computation. By sharing the parent's address space, less time is required in kernel mode doing work not pertinent to the problem. Additionally, by utilizing the same address spaces, less memory management must be performed during switching.

- *Better System Utilization*: Most code is inherently parallelizable even though it is traditionally envisioned as sequential. Any number of disjoint tasks can be executed in parallel instead of in a sequence. A classic analogy for parallelization envisions a house with 5 rooms to be painted. Assuming that painting a room is an individual task that cannot be subdivided, up to 5 painters can be hired to paint the house. Painting some rooms may take longer than others, but overall, having 5 painters will always be much faster than just one.
- *Simplified Sharing & Communication*: Since threads can share data structures (given that they have a common heap space), passage of information between threads is simple and does not require kernel intervention (as is usually required in traditional IPC mechanisms).

Assignment:

You are given 8 files: client_MP6.cpp, dataserver.cpp, request_channel.cpp, request_channel.h, SafeBuffer.h, SafeBuffer.cpp, test.sh, and a functional makefile. When you type the command 'make' into your shell prompt, two executables (in addition to the related .o files) will be produced: client and dataserver.

The code given to you is functional. You may type ./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 request buffer 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.

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 3*n requests.

Multithreading often requires the use of specialized "thread-safe" data structures. In order to facilitate multithreading, you will code a thread-safe class called (for example; you are free to rename it) SafeBuffer (see the commented-out line client_MP6.cpp, "//SafeBuffer request_buffer;"). SafeBuffer can simply wrap around an STL data structure (we recommend std::list or std::queue), but must have at least a constructor, a destructor, and the operations push_back and retrieve_front (the provided make_histogram_table output-formatting function also makes use of a size function, but you are free to modify it). push_back adds an element to the back of the buffer and retrieve_front removes an element from the front so that items are added and removed in FIFO order (see requirements below). Both functions must use locks (see Advanced Concepts section) 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). SafeBuffer.cpp and SafeBuffer.h are simply skeleton files, their structure illustrates what this class is expected to do and will make your assignment easier to grade (since the grader knows what to expect) but ultimately you may modify them however you deem best to complete the assignment.

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).

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. (Please do try to keep it to only a makefile. You shouldn't need to do anything fancy or convoluted)
- **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)

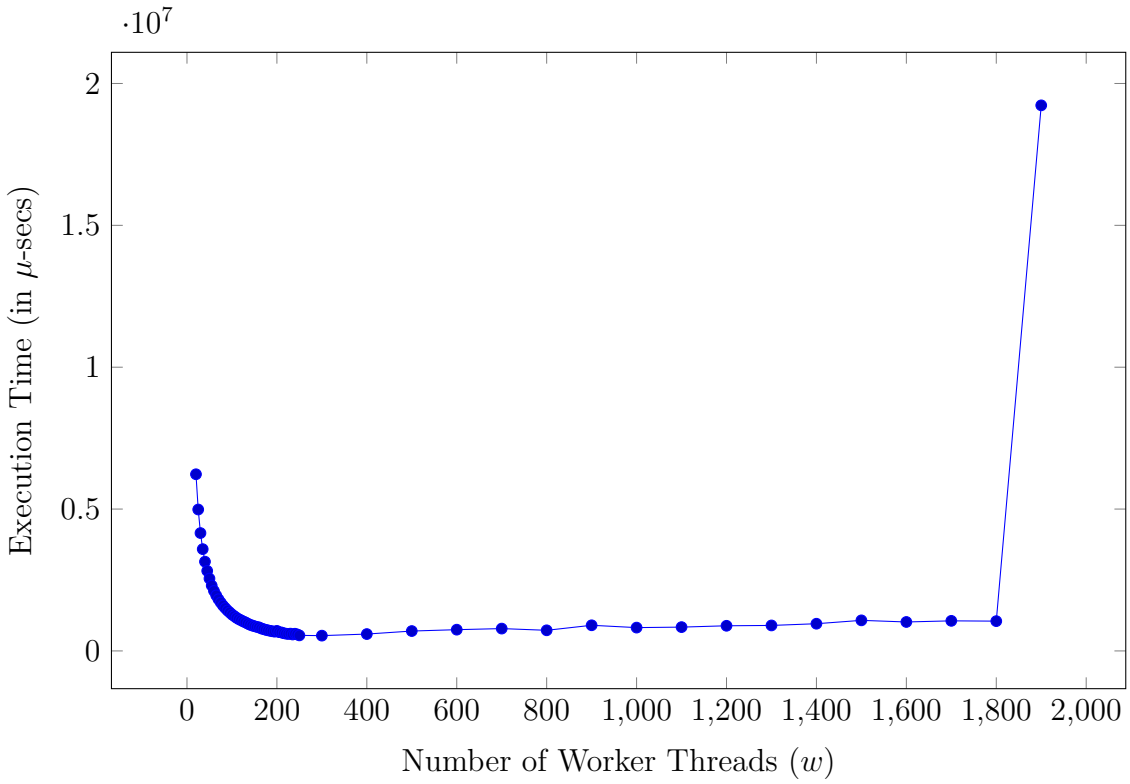
A storage struct is an old-fashioned C struct, which serves mainly to store and organize different variables. An instance of such a struct can be passed to a thread function as the last argument given to pthread_create. The thread function then accesses its arg parameter, a void pointer, and casts it 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 of MP6 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 MP6's 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.

Note Concerning Auxiliary (Helper) classes and functions

The question has often come up, can I use helper functions or helper classes? The answer is almost always yes, you can do whatever you want, *so long as* it doesn't change the expected behavior of the submitted code.

...that practically begs the question: what is the grading script expecting, and how do I avoid breaking it? This time the answer is very easy: *there is no grading script*. For this machine problem, as well as MP7 and MP8, Vocareum is being used only as a submission platform, and grading will mostly be done in lab via demo.

However, you are encouraged to implement your solution as much within the provided framework as possible. It was written the way it was in order to help you understand what needs to be done to complete this assignment, and to help you be more efficient in doing so. Staying within the framework also makes things much more predictable for the graders and will allow you to demo more efficiently. This is highly appreciated by peer teachers and TA's who have to grade multiple sections of demos for any given assignment (although unfortunately we cannot award bonus points for it...).

Advanced Concepts:

Atomic Operations & Race Conditions

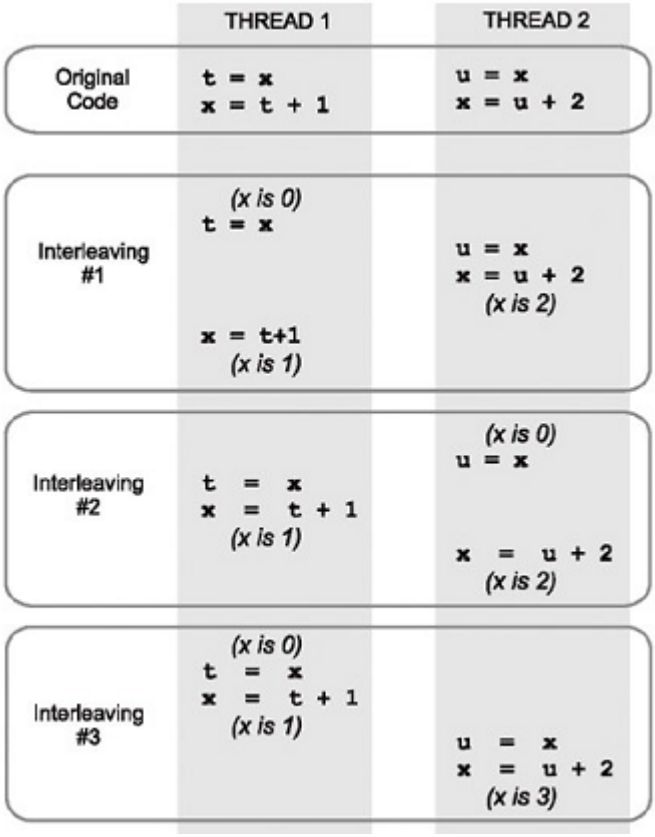
The scheduler (process 0 on UNIX/Linux machines) makes all decisions regarding when processes execute and in what order. The choice of which process to execute next is made by complex algorithms which strive for efficiency and fairness (all processes get a chance). However, the order of execution is not defined by these algorithms (i.e. there is no enforced execution sequence given a list of processes to be run). As a result, processes are switched in and out of execution depending on the will of the system's scheduling algorithm.

UNIX processes are known as preemptive or reentrant (both terms mean the same thing). Reentrant processes are those which can be context switched in the middle of execution and then resumed later. This means that operations such as function calls, file I/O, and even system calls can and will be interrupted during execution and resumed later.

There are some operations that are so fundamental to the system and the hardware that they cannot be interrupted. They either run to completion or not at all. These are known as atomic operations. Most instructions that can be implemented as a single assembly language instruction are atomic. These assembly language statements are executed directly on the processor in one clock cycle. Add with two arguments, subtract with two arguments, load a word from memory, store a word to memory, etc. are all atomic operations. Anything more complex, such as adding three+ arguments, can be divided into more than one assembly instruction. As a result, they are not atomic and can potentially be interrupted.

In the context of normal processes, interruptions are fine. As long as the state is resumed properly, the user is never aware that their program was interrupted (potentially several times). When a program uses threads, however, the results can be disastrous. As a thought experiment, imagine a process containing two threads (1 & 2) and a set of shared global variables. If both threads attempt to execute code to modify any of these variables, the result is dependent on the order in which the two threads execute. The order of thread execution is impossible to predict. By default, there are no guarantees, and every possible situation can and could happen. Situations like this, where the result is dependent on the order of execution, are known as *race conditions*. The figure below illustrates the above situation and shows all possible interleavings of code execution for a given set of instructions.

Figure #3: Race Condition



Synchronization

A segment of code that potentially could be damaging if caught in a race condition is known as a critical section. Critical sections must be executed atomically with respect to all other threads in the system. Even though system calls and functions generally aren't atomic, they can be made atomic through the use of synchronization. This means that when a process calls an function or performs an operation, the function or operation will complete fully or not at all without another process or thread performing the same operation. There are multiple ways of making complex expressions into atomic operations, such as locks and semaphores. Many of these methods will be covered in extensive detail in lecture. Consequently, this section focuses on the synchronization method that you should use for this assignment: the pthread mutex.

A mutex, short for "mutual exclusion," prevents two concurrently running threads from entering the same section of code at the same time. The POSIX API that you'll be using in this maching problem provides some very convenient functions for using mutexes. To use them, be sure to add "#include<pthread.h>" to your program. Mutex types can be declared like any other variables (i.e. pthread_mutex_t temp;) and passed by address to the relevant API calls, such as pthread_mutex_init(&temp, NULL).

A list of relavant API calls along with short descriptions of each is provided below:

1. **pthread_mutex_lock(pthread_mutex_t *mutex)**

   When a process calls lock on a mutex, one of two things will occur. If no other process has the mutex locked, the mutex becomes locked, and the process continues on into the critical section that follows. If another process has control over the mutex (i.e. it is locked by another process), the process that just called for the lock is put to sleep. Any other processes that call lock while the mutex is locked will also be put to sleep.

   All threads must utilize the same pthread_mutex_t object or they will not be synchronized.

   For more information about the use of this function, consult the manual page using the shell command `man 3 pthread_mutex_lock`.

   Important note about POSIX thread manpages: Your Linux/UNIX machine (such as a raspberry pi) may not come with certain manpages by default. In order to acquire them, you will need to instal lthe package `manpages-posix-dev` using the installation mechanism required by your distribution. For the raspberry pi running raspbian, the command is: `sudo apt-get install manpages-posix-dev`.

2. **pthread_mutex_unlock(pthread_mutex_t *mutex)**

   Calling this function releases (unlocks) a mutex that had previously been locked by pthread_mutex_lock. If there is another process waiting to lock the mutex, that process is woken up and given the lock. Afterwards, the whole process starts over again. For more information, consult `man 3 pthread_mutex_unlock`.

3. **pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)**

   This function initializes the mutex referenced by *mutex* to the attributes specified by *attr*. (Note: In this machine problem, attr can always be NULL and *mutex* will automatically be initialized using default values). It returns 0 if mutex initialization was successful or an error value if it failed. For more information, consult `man 3 pthread_mutex_init`.

4. **pthread_mutex_destroy(pthread_mutex_t *mutex)**

   Uninitializes the mutex passed as an argument. As always, for more information, consult `man 3 pthread_mutex_destroy`.

Rubric:

- Request Threads (20 points)
  1. Initial code modified to create 3 parallel request threads in main. (9 points)
  2. request_buffer usage
     - No data requests pushed outside the request threads. (2 points)
     - No quit requests pushed outside main. (2 points)
     - Each request thread pushes n requests. (2 points)
  3. All request threads joined before any worker threads are started. (5 points)
- Worker Thread Functionality (30 points)
  1. Worker threads process all requests and then terminate (and are joined) properly. (10 points)
  2. Final display (10 points)
     - Displays results as histogram with bin size 10 *after* worker threads have been joined. (5 points)
     - Final frequency counts are correct. (5 points)
  3. Proper thread-safety used for request_buffer and *frequency_count vectors. (10 points
     - SafeBuffer (or a similar data structure, i.e. FIFO and provably thread-safe) used for request_buffer. (5 points)
     - No race conditions, especially when incrementing *frequency_count vectors. (5 points)
- Globals (or rather, lack thereof) (10 points)
  1. No global variables used. (-1 point per global variable used, up to -10 points)
- Cleaning up resources (20 points)
  1. No memory leaks. (10 points)
  2. All fifo special files removed via "quit" and proper destructor usage (even in case of errors) (10 points)
     - fifo special files (man 7 fifo) will be present during program execution while they are being used, but all 10 points will be lost *if any fifo files remain after program termination.*
- Report (20 points)
  1. Results demonstrated using plots/graphs (5 points)
  2. Run-time variation observed in the plots (9 points)
     - Plot should be linear with w for some range, but should begin flatten out (or even decrease!) after a certain point when thread/channel management starts to outweigh the benefits of parallelization.
  3. Answers provided to the 3 questions in the Report section of the handout (6 points, 2 for each question)

Bibliography:

[1] Northwood, Chris. "Computer Science Notes ⇒ Operating Systems." Operating Systems. N.p., n.d. Web. 03 June 2016.

[2] Iwillgetthatjobatgoogle. "Race Conditions: First Approach." I Will Get That Job At Google. N.p., n.d. Web. 03 June 2016.