**Machine Problem 7: UNIX Threads**

**Due XX/XX/XXXX**

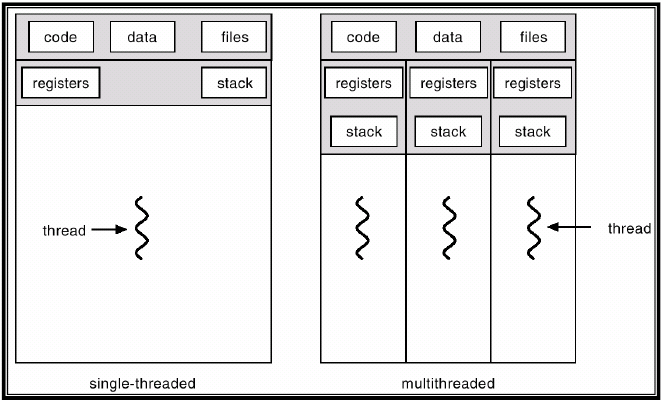
**Introduction**

For many computing tasks it is sufficient that one instruction be executed 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.

**Background**

**How do threads differ from processes?**

The principle difference between threads and processes is that two processes, even if they are instances of the same program, have different address spaces and require the use of special IPC mechanisms to communicate with each other. Threads are created by processes and share the address space of their parent process, executing code from within the process and sharing the global variables of their parent process. In short, two processes execute independently of each other, but multiple threads can execute within a single process.



http://www.pling.org.uk/cs/ops.html

One can think of a thread executing within a process similarly to calling a function within that process, and creating a thread even requires the definition of a “thread function” for the thread to execute: the thread function shares its parent process’s state, memory, and other resources in much the same way that a plain function shares those things with its calling function. And although threads can use pipes, signals, or other IPC mechanisms to communicate with their parent processes or sibling threads, they typically don’t need to.

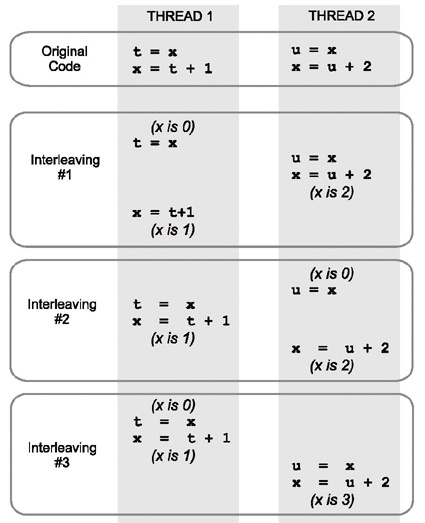
**Advantages of Multithreaded Applications**

Multithreaded applications offer the following advantages, as well as others that aren’t discussed here (**insert citation**):

* *Faster Execution*: threads can take advantage of computer systems which have multiple CPUs, multi-core processors, clusters of machines, and other parallelization tools.
* *Lower Resource Consumption*: threads, which share the address space of a single process, spare the OS the chore of allocating memory for more processes.
* *Better System Utilization*: for example, a file system that needs to retrieve information both from a cache (very fast) and from a hard disk (very slow) can have a separate thread for each task, and neither would have to *block* the other (wait for it to finish).
* *Simplified Sharing and Communication*: as discussed earlier, threads need not suffer from the overhead of IPC since they already share resources.

**Synchronization: A Brief Introduction**

Because they share process resources, threads often attempt to use them at the same time. When this happens, the final result is often determined by the order in which instructions from within the code of different threads are executed. Consider the following example:



http://iwillgetthatjobatgoogle.tumblr.com/post/12633454830/race-conditions-first-approach

The word “interleaving” refers the manner in which lines of code from different threads are scheduled and executed sequentially with respect to their own thread but with no particular order with respect to code from the other threads. In this example the correct final result is clearly 3, but the values of 1 and 2 are possible as well due to interleaving.

In short, when two threads have to modify the same variable or the same data structure, the final result is unpredictable and depends on which thread modifies at first. Fortunately for us, there is a way to keep more than one thread from entering a given section of code at the same time. This prevents interleaving and ensures a somewhat more sequential execution of the code.

**The Mutex: Your most basic, most fundamental synchronization tool**

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 Machine Problem provides some very convenient functions for using mutexes. To use them, you will need to put **#include <pthread.h>** among your includes. Mutex types can be declared like any other variables (i.e. pthread\_mutex\_t pmt;) and passed by address to the relevant API calls (i.e. pthread\_mutex\_init(&pmt, NULL).

1. **pthread\_mutex\_lock(pthread\_mutex\_t \**mutex*)**
   * Locks *mutex*. The calling thread is said to be “in possession of” *mutex*. Any other thread that calls pthread\_mutex\_lock on mutex before the initial calling thread call pthread\_mutex\_unlock(*mutex*) will block until the appropriate call to pthread\_mutex\_unlock is made in the calling thread. This is the desired behavior, but it is *crucial* that the two threads use the *same* pthread\_mutex\_t object, otherwise they will not become synchronized! For information the error values it might return, and other details about this function, type **man 3 pthread\_mutex\_lock** into your command prompt or try the following url: <http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_mutex_lock.html>
2. **pthread\_mutex\_unlock(pthread\_mutex\_t \**mutex*)**
   * Releases a mutex that had previously been locked by pthread\_mutex\_lock, so that another thread can call pthread\_mutex\_lock. More info at **man 3 pthread\_mutex\_unlock** or <http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_mutex_lock.html>
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* (in this machine problem, *attr* can always be NULL and *mutex* will automatically be initialized using default values). It returns 0 if mutex intialization was successful and an error value if it failed. For more information, try **man 3 pthread\_mutex\_init** or <http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_init.html>
4. **pthread\_mutex\_destroy(pthread\_mutex\_t *\*mutex*);**
   * Uninitializes the mutex referenced by *mutex*. Don’t attempt to do this on a locked mutex, you’ll have a bad time. Also try **man 3 pthread\_mutex\_destroy**, or <http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_init.html> for more information.

**The Assignment**

**(Subtitle: Your Mission, if you Choose to Accept It)**

**Code**

You are given 5 files: client\_E7.cpp, dataserver.cpp, request\_channel.cpp, request\_channel.h, and an already-functioning makefile. When you type make all into your command prompt, two executables will be produced: client and dataserver. At this stage, it’s perfectly fine to just type ./client to observe execution of the initial program. “client” forks off and runs the dataserver, then populates a request buffer with 100 requests each for data on 3 different patients: John Smith, Jane Smith, and Joe Smith. This is done using one, single-threaded process, and if you typed ./client –n 10000 you would observe that the program takes a great deal longer to execute since it has to process 100 times as many requests.

**Your job is to improve the efficiency of client\_E7.cpp by using multithreading.** The parameter that controls the number of threads to use is –w, so an example invocation of the finished program would be ./client –n 10000 –w 250, which will have the client process a total of 30000 requests (10000 requests x 3 patients) using 250 worker threads. Note that the on-campus computer science servers, as well as most laptops, will not let you use as many as 250 threads, so you won’t be required to use as many as 250 threads. However, the Raspberry Pi *will* allow you to use that many threads, and that may be useful in testing your code.

To achieve multithreading, you will have to learn about the POSIX API calls that are used to create threads and wait for them to terminate (HINT: **man 3 pthread\_create** and **man 3 pthread\_join**). However, you will also have to correctly use mutexes within your worker thread function to ensure that neither requests nor responses are lost to interleaving.

One very easy way to measure whether your program is correct (take heed, the graders will look at this) is whether the final histogram counts are correct. If you notice, the initial program’s output has three lines (John Smith total: \_\_, Jane Smith total: \_\_, Joe Smith total: \_\_) that each should add up to whatever the parameter “n” was that was passed to client (100 is the default). These numbers are calculated by calling std::accumulate on each of the frequency count vectors that correspond to the responses to data requests for John Smith, Jane Smith, and Joe Smith, i.e. they verify that n requests were sent and n responses were received. The lines below them simply count up how many responses from the data server fell within the given range of values (the dataserver only sends responses within the range 0 to 99 inclusive, and they are randomized). In the finished program, your worker thread function will update the histograms instead of the main function.

**Report**

Once you’ve finished the above coding tasks, draw up 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 “w”. The first point should be w = 1 and the last point should be w = [the maximum number of threads the OS will give to your client program].
   * 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?
   * Compare your client program’s performance to the code you were originally given.
   * **NOTE:** Timing may ***NOT*** be done with a stopwatch, it must be done within your code. One excellent (but not the only) option is described under **man 2 gettimeofday**.
     + ***On that note***, make sure to comment out all non-error-reporting output operations within the block of code that’s being timed *before you attempt to gather actual data*. Such operations are expensive (relatively speaking) and will skew your results.
3. Describe the platform you gathered this data on. Something simple like “a Raspberry PI model B running Raspbian OS,” or “the CSE Linux server,” is sufficient.
   * What’s the maximum number of threads it will allow your client program to create?
   * 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?

**What to Turn In**

* All original .cpp/.h files, and the makefile, with whatever changes you made to complete the assignment
* Any additional files you used to compile/run your program
* Completed report

**Criteria for Success**

* Code:
  + No fifo files remain after running the client program with a sufficiently large number of threads.
  + Client program doesn’t take *unreasonably* long to execute.
  + Histograms and final response counts are correct.
  + BONUS (+5): does not use global variables (besides atomic\_out a), and correctly cleans up all mutexes, threads, and heap memory allocations.
* Report:
  + All questions answered reasonably (at least).
  + Graph meets given requirements.