Machine Problem 7: Synchronization (Due: 4/21/2017)

Introduction:

You may have noticed that the client from MP6 had several limitations:

1. The request buffer had to be populated all at once, before the worker threads begin, which is not adaptable to real-time, i.e. it is easy to imagine situations where requests have to be processed before it is known whether all requests have arrived.
2. The SafeBuffer implementation of the request buffer was susceptible to grow to infinity.
3. The worker threads are in charge of assembling the final representation of the data, which is an example of poor modularity: a histogram of bin size 10 isn't always the best representation of the data, and changing it shouldn't require changing how responses are receieved. In other words, the worker thread function had "too much to do."

Addressing these problems requires more advanced synchronization than we were ready to handle during MP6, but now we are ready to fix things up a bit.

Background:

A Classic Synchronization Problem

To address the first limitation, one could simply allow the request threads to run in parallel with the worker threads. The only new challenge (and one that you will have to address as part of this assignment) would seem to be ensuring proper termination of the worker threads.

But if we look a little closer, we see that the problem is much more complicated than that. What if, due to the unpredictability of the thread scheduler or the data server, the worker threads processed all the requests in the request buffer before the request threads had finished? Maybe the worker threads wouldn't terminate, but if we stick with the MP6 code then they would try to draw from an empty data structure. This highlights one of the glaring problems with using plain-old STL (or otherwise conventional) data structures in a threaded environment, even if they have mutexes wrapped around them like the SafeBuffer class did: *underflow* is possible. Clearly the worker threads need to wait for requests to be added to the request buffer, but what's the best way to do that?

This is closely tied to the second limitation from the introduction: the request buffer is susceptible to grow to infinity (or in our case, n), or *overflow*. However, placing an artificial ceiling on the number of requests is not a realistic solution. The request threads

need to wait for worker threads to deplete the buffer to a certain point before pushing new requests to it, which simply begs the previous question: what's the best way to do that?

The synchronization concern in MP6 was concurrent modification, or interleaving. The new synchronization concerns of MP7 combine to form one of the classic synchronization problems that will be/has been discussed in lecture: the producer-consumer problem. It's a fairly common programming problem, and one which has a bounty of real-life applications. There are some naïve solutions which may be/may have been discussed in lecture, but there's not room to discuss them here.

Let's Try a New Data Structure!

All the problems discussed so far are shortcomings of the implementation of the request buffer, so the solution could be a new data structure. If that were the case, the new data structure would need to:

- Prevent underflow
- Prevent overflow
- Prevent concurrent modification

Because it is "bounded" on both sides (e.g. no overflow or underflow), we will call this data structure a *bounded buffer*. Now, how can we build a bounded buffer?

Another Synchronization Primitive: Semaphore

We can build a bounded buffer (please forgive the alliteration) using a new (to us) synchronization primitive, called a *semaphore*. A semaphore is a variable that keeps track of how many units of a given resource are available. In our case, the resource is the number of available spaces for requests in the bounded buffer. While the mutex primitive provides operations that lock and unlock the mutex, the semaphore provides operations that decrement and increment the record of the amount of resource available. When a thread or process needs the resource it decrements the semaphore, and if the semaphore's count passes below 0 then the calling thread enters a wait queue and sleeps. When more of the resource becomes available the semaphore count is incremented and, if the count passes from -1 to 0, then the scheduler picks a thread from the semaphore wait queue and wakes it up. Traditionally the semaphore's decrement/wait and increment/wake up operations are respectively called P() and V().

Note that a mutex is simply a semaphore with a maximum count of 1: neither allow more than one thread to hold the lock at any given time.

Putting it all together

Now we can explain how using a bounded buffer to hold requests prevents underflow

and overflow. It uses two semaphores, which we call full and empty, which represent (respectively) the number of requests in the bounded buffer and the number of free spaces in it. When a thread needs to remove a request from the bounded buffer for processing, the bounded buffer internally calls full.P(), removes the next request, then calls empty.V(). The opposite occurs when a thread needs to add a request to the bounded buffer: internally the bounded buffer calls empty.P(), adds the request, then calls full.V(). When the bounded buffer is constructed the counter for the empty semaphore is given an initial value that fixes the maximum size of the bounded buffer, while the counter for the full semaphore is initialized to 0.

Take a moment to think about how this works. When the full semaphore reaches zero there are no more requests left in the buffer, so processes that decrement it past zero wait instead of attempting to draw from an empty queue. When the empty semaphore reaches zero the buffer has reached the maximum allowed size, so processes that decrement it past zero will wait instead of growing the buffer to infinity. Note that a single semaphore is insufficient to accomplish this, since semaphores are only "one-way": they lock when decremented past zero, but can theoretically be incremented up to infinity, hence the need in this case for two semaphores "going in opposite directions."

Because multiple threads might "obtain a semaphore" (call P() without blocking and be granted access to the buffer) at the same time it is still necessary for operations on the underlying data structure to be protected by a mutex to prevent concurrent modification. This mutex should, as in MP6, be part of the bounded buffer implementation just like the full and empty semaphores.

Addressing the last limitation

The bounded buffer data structure suffices to address the first two limitations mentioned in the intro section. So what about the third? The solution is fairly simple: instead of having the worker threads directly modify the frequency count vectors for the three users, make three response buffers (one per user) and have the worker threads sort responses into the correct one. Then, have three "statistics threads" (again, one per user, and the same users as in MP6) remove responses from the response buffers and process them as needed. For this assignment the outcome will still be a histogram with bin size 10, but theoretically it could be anything appropriate.

This solution introduces a separate producer-consumer problem, where instead of the request threads as producers and the worker threads as consumers we have the worker threads workerworkeras producers and the statistics threads as consumers. Since we already have the bounded buffer data structure available to use, we can just it for the user response buffers and the problem is solved. There you go.

Assignment:

Code

You are given the same files as in MP6 (dataserver.cpp, reqchannel.cpp/.h, functioning makefile), except that the client is called client_MP7.cpp and has been changed some. You will have to modify it so that it has the same functionality as the completed client_MP6.cpp, but the request threads, worker threads, and statistics threads will all run in parallel. To make this requirement a bit clearer, here are some rules of thumb to follow:
Your code may NOT (i.e. points will be deducted if it does):

- Wait for any thread to finish before all threads have been started
- Process requests or responses in a non-FIFO order
- Push data requests to the request buffer outside the request thread function, or push quit requests outside of main
- Push anything to the response buffers outside of the worker thread function, to remove anything from them outside of the statistics thread function.
- Modify the frequency count vectors outside the statistics thread function (although it is fine to initialize them inside main)

In addition to the familiar command-line arguments from MP6 ("n" for number of requests per user and "w" for number of worker threads), your program must take an additional command-line argument: "b," for the maximum size of the request buffer. The user response buffers can be of an arbitrary fixed size. On that note, to make any of this possible you will need to implement the BoundedBuffer and Semaphore classes. They must at least have their own dedicated .h files, though you can also have corresponding .cpp files if you find it helpful. However, in any case you will lose points if these classes are defined in client_MP7.cpp: they must have their own files. Note that the BoundedBuffer class will have to use the Semaphore class, but it will not be necessary to use the Semaphore class anywhere else.

For these classes as with MP6's SafeBuffer, a description is provided of the functionality necessary to complete the assignment, but naming and other implementation decisions are ultimately left up to you.

The Semaphore class must support at least the following operations:

- Construction: initialize mutex, counter, and wait queue members. The constructor should take a single int argument for the initial counter value.
- Destruction: Properly clean up all non-primitive members (i.e. calling pthread_mutex_destroy, etc.), and return all heap memory to the heap if any was used.
- P: locks a mutex, then decrements the counter. If counter goes below zero, release mutex and enter wait queue. Else release mutex.
- V: locks a mutex, then increments the counter. If previous counter value was negative and current value is non-negative, signal the wait queue. Then release mutex.

You may be wondering how one is supposed to do all the semaphore's "wait queue" operations, since the language on that point has been vague so far. We recommend letting the POSIX API do the leg work for you: the "wait queue" member of the Semaphore class can be a single pthread_cond_t variable. Check out the man pages man 3 pthread_cond_init, man 3 pthread_cond_signal, man 3 pthread_cond_wait, and man 3 pthread_cond_destroy.

This semaphore implementation exposes you to how semaphores actually work, but there is one very remarkable alternative implementation: simply use the POSIX semaphore API (man 7 sem_overview). This is especially useful when you need to synchronize different processes on the same semaphore, and not just different threads of the same process, and for this reason will be very helpful for MP8. Admittedly this approach is a little bit more difficult to learn, but ultimately both approaches are good to know since they have different strengths and weaknesses. If you choose to use this approach, you will not be required to write an additional Semaphore class or make any of the changes that would go adding the Semaphore class (i.e. no need to add a makefile rule, no need for semaphore.h or semaphore.cpp, etc.). Further you can read the rubric item titled "Semaphore class" as "Semaphore usage." With those clarifications made, please understand that this handout was written with the other approach in mind.

The BoundedBuffer class must support at least the following operations:

- Construction: initialize access mutex, full and empty semaphores, and (if necessary) the underlying data structure
- Destruction: properly clean up all non-primitive members, and return all heap memory to the heap if any was used
- Element addition: add element to underlying data structure, maintaining thread-safety by proper use of semaphores and mutexes. Ensure FIFO order.
- Element removal: remove element from underlying data structure and return it, maintaining thread-safety by proper use of semaphores and mutexes. Ensure FIFO order.

Since requests must be processed in a FIFO order, we recommend you use something like std::queue or std::list for the underlying data structure. Adding and removing elements from std::vector at the front is very expensive and will impact the performance of your program.

Unlike with the other machine problems, you are expected to modify the provided makefile as necessary to accommodate your new classes.

Bonus

You have the opportunity to gain bonus credit for this machine problem. To gain this

bonus credit, you must implement a real-time histogram display for the requests being processed.

You may remember the final display from MP6. The make_histogram_table function is used to format and output the frequency counts for the requests processed for each user. This function takes as arguments the 3 user names and pointers to the 3 *frequency_count vectors.

Write a signal-handler function that clears the terminal window (system("clear") is an easy way to do this, check out man 3 system) and then displays the output of make_histogram_table. You will need to make sure that this is thread-safe by synchronizing on the same mutexes that the statistics threads use to modify the *frequency_count vectors.

In main, register your signal-handler function as the handler for SIGALRM (man 2 sigaction). Then, set up a timer to raise SIGALRM at 2-second intervals (man 2 timer_create, man 2 timer_settime), so that your handler is invoked and displays the current user response totals and frequency counts approximately every 2 seconds. To do this, you will need to make sure that your signal handler is given all the necessary parameters when it catches a signal (man 7 sigevent). When all requests have been processed, stop the timer (man 2_timer delete). If you have succeeded, the result will look like a histogram table that stays in one place in the terminal while its component values are updated to reflect the execution of the underlying program.

Is there some purpose for this bonus portion other than making your program output look nice? Yes, there is. Firstly it gives you the opportunity to use signals and signal handlers, which are important both in this course and generally in computer science. Secondly it provides a snapshot of the run-time state of your program, which can be useful for debugging (i.e. if you can tell that the histogram is updating but the values aren't changing, you know there's a problem to be diagnosed). Additionally, it provides a small amount of preparation for MP8.

As with the other parts of the assignment you will lose points for using global variables. However, using global variables for the bonus will result in points being deducted only from the bonus portion of the assignment, not from the main assignment, i.e. you cannot lose points by attempting to earn the bonus credit. Finally, please use the make_histogram_table function to format your output. Programming assignments are much easier to grade when the output is standardized, but again the choice of how to do this is ultimately yours.

Report

1. Present a brief performance evaluation of your code. If there is a difference in performance from MP6, attempt to explain it. If the performance appears to have decreased, can it be justified as a necessary trade-off?

2. Make two graphs for the performance of your client program with varying numbers of worker threads and varying size of request buffer (i.e. different values of "w" and "b") for n = 10000. Discuss how performance is affected by each of them, and offer explanations for both.
    - Include b = 1 is among your data points.
    - Please don't copy-paste from your MP6 report, even if some of the answers you come up with are similar for MP7.
3. Describe the platform that your data was gathered on and the operating system it was running. A simple description like "a Raspberry PI model B running Raspbian OS," or "the CSE Linux server," is sufficient. (Think of this as free points)

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

Rubric

1. BoundedBuffer class (20 pts)
2. Semaphore class (20 pts)
3. Having BoundedBuffer and Semaphore as separate classes, not in client.cpp. This should be reflected in the makefile. If you have these in client.cpp, you will lose 10 points
4. Correct values of the frequency counts (20 pts)
    - This is important since it indicates whether your bounded buffer and semaphore implementations are correct
5. Not having global variables (10 pts)
6. Bonus: using timers to display the counts (20 pts)
    - If you use global variables, you only get 10 bonus pts (again, no points will be deducted from the main part of the assignment)
    - If your implementation uses a separate thread instead of a signal handler, you only get 5 bonus pts
7. Report (20 pts)
    - Should show plots of runtime under varying b and w with constant n, per earlier description. n should be at least 10000.