Machine Problem 8: Synchronization (Due: XX/XX/XX)

Introduction:

You may have noticed that the client from MP7 had several limitations:

1. The request buffer had to be populated all at once, before the worker threads begin: not adaptable to real-time
2. The request buffer was susceptible to grow to infinity: potential memory issues
3. The worker threads are in charge of assembling the final representation of the data - poor modularity: a histogram of bin size 10 isnt always the best representation of the data, and changing it shouldnt require changing how responses are received.

Addressing these problems requires a bit more advanced understanding of synchronization than we were ready to handle during MP7, but by this point we should be 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 wouldnt terminate, but if we stick with the MP7 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 whats 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), in other words it allows overflow. And we cant place an artificial ceiling on the number of requests, that wouldnt be realistic. The request threads need wait for worker threads to deplete the buffer to a certain point before pushing new requests to it, but we run into the same problem as before: whats the best way to do that?

The synchronization concern in MP7 was concurrent modification, or interleaving. The new synchronization concerns of MP8 combine to form one of the classic synchronization problems that will be/has been discussed in lecture: the producer-consumer problem. Its a fairly common programming problem, and one which has a bounty of real-life applications. There are some nave solutions which may be/may have been discussed in lecture, but theres not room to discuss them here.

Let's Try a New Data Structure!

All the problems discussed so far are problems with the data structure used for 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, we call this data structure a bounded buffer. This again begs the question, how can the new data structure be made to prevent overflow and underflow?

Another Synchronization Primitive: Semaphore

It can do this through the use of 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 semaphores count passes below 0 then the process 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 process from the semaphore wait queue and wakes it up. Traditionally the semaphores decrement/wait and increment/wake up operations are called P() and V() respectively.

Putting it all together

Now we can explain how the bounded buffer prevents underflow and overflow. It uses two semaphores, which we call full and empty, to represent the number of requests in the request buffer and the number of free spaces remaining in the bounded buffer. When a process needs to remove a request from the bounded buffer for processing, internally the bounded buffer calls full.P(), then removes the next request, then calls empty.V(). The opposite occurs when a process 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 represents the maximum desired size of the bounded buffer, while the counter for the full semaphore is given the initial value of 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.

Because multiple processes/threads might obtain a semaphore (successfully call P() 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 be contained in the bounded buffer just like the full and empty semaphores.

Addressing the last limitation

The bounded buffer data structure suffices to address the first two limitations stated 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 patients, make three response buffers (one per patient) and have the worker threads just sort responses into the correct one. Then, have three statistics threads (again, one per patient) remove responses from the response buffers and process them however. For this assignment the result 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 being the producers and the worker threads being the consumers we have the worker threads being the producers and the statistics threads being the consumers. Since we already have the bounded buffer data structure available to use, we can also use it for the patient response buffers and the problem is solved. There you go.

Assignment:

Code

You are given the same files as in MP7 (dataserver.cpp, reqchannel.cpp/.h, functioning makefile) except the client is called client_MP8.cpp.You will have to modify it so that it has

the same functionality as the completed client_MP7.cpp except that the request threads, worker threads, and statistics threads 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
  - This means that requests and responses must removed from their respective buffers in the order in which they were added to those same buffers.
- Push data requests to the request buffer outside the request thread function Note that any other kind of request may be pushed or sent inside main, but data requests can only be handled by the thread functions
- Modify the frequency count vectors outside the statistics thread function

In addition to the command-line arguments from MP7 (n for number of requests per patient 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 patient 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_MP8.cpp: they must have their own files. Note that the BoundedBuffer class will have to use the Semaphore class, but its not necessary to use the Semaphore class anywhere else.

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.), 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 wait queue operations, since the language on that point has been vague so far. We recommend letting the API do the wait queue operations for you: man 3 pthread_cond_init, man 3 pthread_cond_signal, man 3 pthread_cond_wait, man 3 pthread_cond_destroy. The wait queue member of the Semaphore class can be a single pthread_cond_t variable.

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, return all heap memory to the heap if any was used
- Element addition: add element to underlying data structure, with proper use of semaphores and mutexes. Ensure FIFO order.
- Element removal: remove element from underlying data structure and return it, with proper use of semaphores and mutexes. Ensure FIFO order.

Since requests must be processed in a FIFO order, we recommend 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.

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.

Run the initial client code once, and pay close attention to the final display. It uses the make_histogram_table function to format and output the frequency counts for the requests processed for each patient, which takes as arguments the 3 patient 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) and then displays the output of make_histogram_table. You will need to make sure that this is threadsafe by synchronizing on the same mutexes that the worker 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 patient 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 should 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 (perhaps for the first time), 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 arent changing, you know theres a problem to be diagnosed).

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.

Report

1. Present a brief performance evaluation of your code. If there is a difference in performance from MP7, 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 changes (or fails to change) with each of them, and offer explanations for both.
   - Include b = 1 is among your data points.
   - Please dont copy-paste from your MP7 report, even if some of the answers you come up with are similar for MP8.
3. Describe the platform the data was gathered on and the operating system it was running. Something simple 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 the main. This should be reflected in the makefile. If you have these in the main, 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 (note that no points will be deducted from the main part of the assignment)
   - If you do it using 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 n, b, w. Specially we want to see variations in b and w after setting n=10K at least.