Machine Problem 6: Synchronization

Due 11/6/2016 at 11:59pm

Introduction

You may have noticed that the client from MP5 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 isn't always the best representation of the data, and changing it shouldn't require changing how responses are received.

Addressing these problems requires a bit more advanced understanding of synchronization than we were ready to handle during MP5, 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 write a new thread function to populate the request buffer with the requests for each patient. Better yet, have a separate thread for each patient. The request threads would run concurrently with the worker threads and terminate when they had made all their requests. The only new challenge (and one that you will have to address as part of this assignment) would 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 MP5 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: **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), in other words it allows **overflow**. And we can't place an artificial ceiling on the number of requests, that wouldn't 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: what's the best way to do that?

The synchronization concern in MP5 was concurrent modification, or interleaving. The new synchronization concerns of MP6 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 with 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 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

You'll eventually figure out that the mechanisms used to prevent overflow and underflow require an assurance of no concurrent modification, so it's easiest to wrap that into the data structure as well even though we had already solved the problem at the function level. 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 does 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, 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 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 semaphore's 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 just use it for the patient response buffers and the problem is solved. There you go.

The Assignment

Code

You are given the same files as in MP5 (dataserver.cpp, request_channel.cpp/.h, functioning makefile) except the client is called client_MP6.cpp because it is substantially different from client_MP5: you have much less code given to you than in the previous assignment. You will have to modify it so that it has the same functionality as the completed client_MP5.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 MP5 ("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 `bounded_buffer` and `semaphore` classes**. We recommend at least giving them their own dedicated .h files, though you can also have corresponding .cpp files if you find it helpful. The `bounded_buffer` class will have to use the `semaphore` class, but it's 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. This results in by far the fastest, most provably correct way to complete this assignment. If you find another way, please tell us about it.

The `bounded_buffer` 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.

Report

1. Present a brief performance evaluation of your code. If there is a difference in performance from MP5, 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"). Discuss how performance changes (or fails to change) with each of them, and offer explanations for both.
    - Be sure that w = 1 and b = 1 are among your data points.
    - *Please* don't copy-paste from your MP5 report, even if some of the answers you come up with are similar for MP6.
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 (such as your semaphore and bounded buffer implementations, if they are not contained in client_MP6.cpp)
- 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.
    - Follows the "Your code may NOT" rules listed earlier.
    - Semaphore and bounded buffer implementations are correct.
        - FREE POINTS: nothing else will work if your semaphore and bounded buffer classes don't work.
        - HINT: they're simple enough that a peer teacher/TA can verify the code's correctness *without* you needing to test it.
    - BONUS (+1): does not use global variables (besides `atomic_out a`), and correctly cleans up all mutexes, threads, and heap memory allocations.
        - Fewer points are available than in MP5 because if you got these points in MP5 then you will have learned how to get them in MP6.
    - BONUS (+20): use **`man 2 setitimer`**, **`man 3 signal`**, **`system("clear")`** and an appropriate signal-handler function to display the response count histograms in the console in real time at 2-

second intervals. The format should be the same as in the final results display. To get these points, demonstrate the functionality to a peer teacher/TA in lab with an execution of your client for n = 10000 requests. Be sure to show them the corresponding code.

- Ensure that this functionality is commented out or otherwise not impacting performance while you gather data for your report.
- HINT: signal-handler functions take an int and return void.
- HINT: if all goes well you shouldn't need to use the `atomic_out` class, but you'll get the points even if you do use it.

- Report:
  - All questions answered reasonably (at least).
  - Demonstrate the time required for finishing using graphs.