Programming Assignment 4: Synchronization

Due: 11/1/18 Thursday at 11:59pm

Introduction

You may have noticed that the client from PA3 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.
2. The request buffer was susceptible to grow to infinity, which can make this program impractical small limited-memory devices. Even with larger memory, you may want to limit memory usage to certain maximum, which you cannot do for PA3.
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 received.

Addressing these problems requires more advanced synchronization than we were ready to handle during PA3, 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 wouldn't terminate, but if we stick with the PA3 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), in other words it allows overflow. However,

placing an artificial ceiling on the number of requests is not a realistic solution. The request threads need wait for worker threads to deplete the buffer to a certain point before pushing new requests to it, which brings us back to our earlier question: what's the best way to do that?

The synchronization concern in PA3 was concurrent modification, or interleaving. The new synchronization concerns of PA4 combine to form one of the classic synchronization problems that have 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. In class, we also saw some naive effort to solve this problem without success.

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. Now we must ask, how can we build a bounded buffer?

A Synchronization Primitive: Condition Variable

Condition variable is an excellent way for asynchronously waiting for certain event/condition. We have seen in class the alternative of such asynchronous wait and notification is some busy-spin wait which is either inefficient or wasteful in terms of CPU cycles. POSIX provides `pthread_cond_t` type that can waited upon and signaled by one or more threads. In PA4, we can use this type to guard both overflow and underflow from happening.

You will need one condition variable for guarding overflow, and another one for guarding underflow. Each producer thread (i.e., request threads) waits for the buffer to get out of overflow (i.e., buffer size is less then the maximum) encoded in condition # 1 and then pushes an item. It also notifies the consumer threads (i.e., worker threads) by signaling condition # 2 that data is now available. This wakes up all waiting consumer threads (if any) one at a time. The logic for the consumer threads is similar with the difference that consumer threads wait on condition # 2 and signals condition # 1. Please check the man pages and the L11 slide set for reference.

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

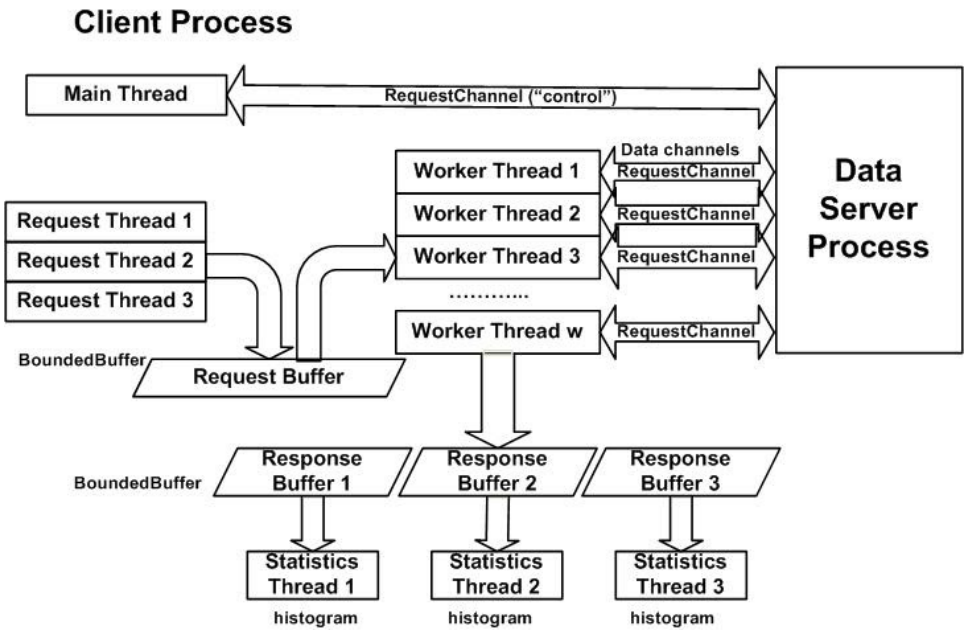

Figure 1: Structure of PA#4.

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.

Assignment

Code

You are given the almost the same set of files as in PA3 (`dataserver.cpp, reqchannel.cpp/.h, makefile, Histogram.h/.cpp, reqchannel.h/.cpp`, and `client.cpp`), except the buffer class is now in `BoundedBuffer.h/.cpp`.

Your code must also incorporate the following modifications compared to PA3:

- Start all threads (i.e., request, worker and statistics) simultaneously. Unlike PA3, there is no episode here. All threads must run together. If that is not the case, the BoundedBuffer objects will fill up and stop the Producers and the program should deadlock.

- Worker threds do not modify the Histogram object directly. Rather they push the responses onto the correponding statistics buffers. There are 3 statistics buffers - one

for each person. Each of these statistics bufferes is cconsumed by a statistics thread who updates the histogram independently from other threads (because there is one stat thread per stat buffer).

- `BoundedBuffer` class should need 2 synchronization primitives: a mutex and two condition variables. You should not need any other datastructures or types. You are NOT allowed to use `Semaphore` type which would be an alternative.
- You must specify a 3rd command-line argument for the capacity of the `BoundedBuffer`s: it is `b`. You should include it in the `getopt()` function. The capacity of the `request_buffer` (i.e., between request and worker threads) should be `b` while that of the 3 `stat_buffers` should be each `b/3` (also ensure that these buffers are at least 1 in size).

Overall, the structure of the program should look like Fig. 1.

Bonus

You have the opportunity to gain bonus credit for this Programming Assignment. To gain this bonus credit, you must implement a real-time histogram display for the requests being processed.

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 `Histogram::print()` function.

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. You can use global variables for the bonus part.

Note that this is an example of asynchronous/real-time programming where the program performs certain operations based on the clock instead of in synchrnous manner. Such technique is useful when a program itself is busy doing its main work, while the asynchronous part is in charge of dealing with real-time events (e.g., printing something every few seconds, wrap-up computation after a deadline expires).

Report

1. Present a brief performance evaluation of your code. If there is a difference in performance from PA3, 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 $n = 10$K at the minimum. Discuss how performance changes (or fails to change) with each of them, and offer explanations for both.

What to Turn In

- The full solution directory including all cpp/h files and a `makefile`
- Completed report

Rubric

1. BoundedBuffer class (20 pts)
    - Your program cannot have a `Semaphore` class. Having one would result in 20 lost points
2. Not having global variables (10 pts)
3. Cleaning up fifo files and all dynamically allocated objects (10 pts)
4. Correct counts in the histogram (20 pts)
5. Existence of all threads (request threads, worker threads and stat threads) and their concurrent operation (20 pts)
    - Missing any of these threads or not running them simultaneously would result in all points in this category being lost
6. Report (20 pts)
    - Should show plots of runtime under varying n, b, w. Specially we want to see variations in b (range $[1, 200]$) and w (range $[1, 500]$) after setting $n = 10$K at least.
    - Having $w < 500$ will result in 5 lost points
    - Requiring $b = 3 * n$ will result in 15 lost points, because that indicates that you did not need any overflow or underflow checks.
7. Bonus: using timers to display the counts (20 pts)
    - If your implementation uses a separate thread instead of a signal handler, you only get 5 bonus pts