Condition Variables

Annalise Tarhan

March 4, 2021

1 Study the code in main-two-cvs-while.c. What should happen when you run the program?

The four methods behave predictably. do_fill and do_get access the buffer, add or remove an element, and update num_full and the appropriate pointer. The producer and consumer methods loop for a certain number of times, acquiring the lock, waiting for the buffer to be empty or full using a condition variable, then calling do_fill or do_get, signaling the other, and releasing the lock.

2 Run with one producer and one consumer, and have the producer produce a few values. Start with a buffer with size one and then increase it. How does the behavior of the code change with larger buffers? What would you predict num_full to be with different buffer sizes and different numbers of produced items when you change the consumer sleep string from default to -C 0,0,0,0,0,0,1?

The most obvious difference, besides the producer and consumer producing and consuming multiple values in a row, is that each takes longer turns. With a buffer size of one, they are limited to about six instructions each before they have to wait. With larger buffers, sometimes they only run for a few instructions before they switch, but sometimes they run for much longer.

With the given consumer sleep string, the consumer thread would always sleep after releasing the lock, making it overwhelmingly likely that the producer would add values faster than the consumer removes them, causing the buffer to gradually fill up.

3 If possible, run the code on different systems. Do you see different behavior?

When designing a system and scheduler, there is always a tradeoff between different priorities, including fairness. Context switches are expensive, so minimizing them improves performance at the expense of fairness. I only have one system so I can't compare, but if there is a difference I would expect it to be in how frequently the threads are switched.

4 How long do you think the following execution, with one producer, three consumers, a single shared buffer, and each consumer pausing at point c3 for a second, will take?

The pauses mean that each consumer will wait for a full second after being woken. Since the buffer only holds one value, consumers will have to sleep again after consuming each value. This means that for ten items, it will take at least ten seconds.

5 Now change the size of the shared buffer to 3. Will this make any difference in total time?

Once a thread is awake, it can keep consuming until the buffer is empty. I would expect that because each of the consuming threads has to sleep for a full second before beginning to consume, the producer would have plenty of time to fill the buffer. By this logic, it would take four rounds for the consumers to consume all ten values three at a time, so it would only take four seconds.

Surprisingly (to me) it usually still took at least ten seconds. During those runs, the producer never ran long enough at a time to fill more than two spaces in the buffer. The exception was a run that only took six seconds, and unsurprisingly the buffer was also completely filled during that run. The enforced naps weren't the reason the other runs took so long, it was because the scheduler didn't let the producer run for long enough to be efficient.

6 Now change the location of the sleep to c6, again using a single-entry buffer. What time do you predict in this case?

With efficient scheduling, the three consumers would alternate with the producer, allowing all three to consume an entry before waiting for a second.

Then, they would repeat that sequence three times, followed by one more produce/consume. This is possible since the consumers release the lock before sleeping, and it would only take four seconds. In fact, it took five seconds, with the extra second necessary to process the end of stream markers.

7 Finally, change the buffer size to three again. What time do you predict now?

Because there are three consumers, the per-second batch size was already three entries. Increasing the buffer size does not make a difference.

8 Now look at main-one-cv-while.c. Can you configure a sleep string, assuming a single producer, one consumer, and a buffer size of 1 to cause a problem with this code?

No. The problem with using a single condition variable arises when there is a possibility of waking the 'wrong' thread, which can't happen when there are only two threads alternating.

9 Now change the number of consumers to two. Can you construct sleep strings for the producer and consumers so as to cause a problem in the code?

It's not necessary. Even with a number of items as small as three it fails half of the time with all three threads stuck sleeping. With a larger number, it fails almost every time.

10 Now examine main-two-cvs-if.c. Can you cause a problem to happen in this code? Again consider the case where there is only one consumer, and then the case where there is more than one.

When there is only one consumer, there isn't a problem. With two consumers, there usually is without any intervention. I wasn't able to write sleep statements that worked perfectly every time, but the idea is to force one of the consumers

to sleep after the cond_wait statement at c3 so that by the time it wakes up the other one has already consumed the item in the buffer. Forcing the producer to sleep at the end of its loop at p6 doesn't hurt either.

11 Finally examine main-two-cvs-while-extra-unlock.c. What problem arises when you release the lock before doing a put or a get? Can you reliably cause such a problem to happen, given the sleep strings? What bad thing can happen?

The problem could occur with a buffer with at least two spaces and two consumers. Consider the scenario where the producer fills an element, waking one of the consumers. Just before the consumer starts do_get, the producer is woken and fills another element, triggering the other consumer to wake. That consumer empties the buffer space, but before it updates the use pointer, the first consumer is reawakened. It checks that there is an element at the use pointer, which fails and crashes the program. The available sleep strings aren't fine-grained enough to force this to happen reliably.