Condition Variables

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Back to the Queue Example

- Let’s take another look at our **queue of integers** implementation, which has two methods: insert() and remove()
- We have two kinds of threads
  - **Producers**: put integers into the queue
  - **Consumers**: remove integers from the queue
- Let’s look at our previous implementation, which uses locks to avoid race conditions, ***assuming a non-thread-safe queue implementation***

Simple Solution

```
queue_t queue;
lock_t mutex;
```

```
// producer
  …
  x = generate();
  lock(mutex);
  insert(queue, x);
  unlock(mutex);
  …
```

```
// consumer
  …
  lock(mutex);
  x = remove(queue);
  unlock(mutex);
  process(x);
  …
```

- Typically the producers and consumers do the above repeatedly, in some loop

Producer/Consumer

- The producer/consumer model is very common and very useful
- **A producer:** a threads that repeatedly "generates" items and puts them into some data structure
- **A consumer:** a thread that repeatedly gets items from a data structure and "processes" them
- **A data structure** (often called the "producer-consumer buffer") that allows the above to happen correctly for any number of producers and consumers

Producer/Consumer

- The code two slides ago is not a true producer/consumer implementation: The consumer should **WAIT** for items to be put in the queue whenever the queue is empty
- Let's say that remove() returns -1 when the queue is empty (could throw an exception, etc.)
- Then we could attempt to implement a true producer/consumer as follows….

Producer/Consumer

```
queue_t queue;
lock_t mutex;
```

```
// producer
   …
   x = generate();
   lock(mutex);
   insert(queue, x);
   unlock(mutex);
```

```
// consumer
   …
   while (1) {
      lock(mutex);
      x = remove(queue);
      unlock(mutex);
      if (x == -1) continue;
      process(x);
      break;
   }
```

Producer/Consumer

```
queue_t queue;
lock_t mutex;
```

```
// producer
   …
   x = generate();
   lock(mutex);
   insert(queue, x);
   unlock(mutex);
```

```
// consumer
   …
   while (1) {
      lock(mutex);
      x = remove(queue);
      unlock(mutex);
      if (x == -1) continue;
      process(x);
      break;
   }
```

What's not great about this?

Busy Wait

- Our implementation has a **busy wait** (it "spins")
- The Consumer keeps trying to remove an item while the queue is empty, which burns/wastes CPU cycles
  - Just like a spinlock for a long critical section
- Something useful could be done with the CPU instead of having it just "spin"
  - Typically, many processes/threads could benefit from being scheduled for their time quanta
- Furthermore, busy waiting increases heat and power consumption, which are crucial issues
- **Bottom line: busy waits are at best frowned upon by developers, and typically prohibited**
- Let's try avoiding repeated calls to remove()…

Using a Blocking Lock???

```
queue_t queue;
lock_t mutex;
blocking_lock_t empty(LOCKED);
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(empty);
    unlock(mutex);
    …
```

```
// consumer
  …
  lock(empty);
  lock(mutex);
  x = remove(queue);
  if (queue.size != 0)
      unlock(empty);
  unlock(mutex);
  …
```

- We use a **(re-entrant and blocking)** lock called “empty”
  - Initially in the locked state
- The Consumer blocks until the producer calls unlock(), and does not call unlock() unless it just emptied the queue

Using a Blocking Lock???

```
queue_t queue;
lock_t mutex;
blocking_lock_t empty(LOCKED);
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(empty);
    unlock(mutex);
    …
```

```
// consumer
  …
  lock(empty);
  lock(mutex);
  x = remove(queue);
  if (queue.size != 0)
      unlock(empty);
  unlock(mutex);
  …
```

- Most people don’t like the above solution (and you will never see it used), for good reasons…

Using a Blocking Lock???

```
queue_t queue;
lock_t mutex;
blocking_lock_t empty(LOCKED);
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(empty);
    unlock(mutex);
    …
```

```
// consumer
  …
  lock(empty);
  lock(mutex);
  x = remove(queue);
  if (queue.size != 0)
      unlock(empty);
  unlock(mutex);
  …
```

- **Problem #1**: This assumes that a thread can call unlock() on a lock without having called lock() on it
  - This is often not supported
  - And is known to be fraught with peril anyway from a software maintenance/debugging perspective

Using a Blocking Lock???

```
queue_t queue;
lock_t mutex;
blocking_lock_t empty(LOCKED);
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(empty);
    unlock(mutex);
    …
```

```
// consumer
    …
    lock(empty);
    lock(mutex);
    x = remove(queue);
    if (queue.size != 0)
        unlock(empty);
    unlock(mutex);
    …
```

- **Problem #2**: Readability suffers because some locks are used for mutual exclusion, and some locks are used for communication, and yet they look the same
    - Even though some disagree (see upcoming Semaphore lecture notes)

Using a Blocking Lock Lock???

```
queue_t queue;
lock_t mutex;
blocking_lock_t empty(LOCKED);
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(empty);
    unlock(mutex);
    …
```

```
// consumer
    …
    lock(empty);
    lock(mutex);
    x = remove(queue);
    if (queue.size != 0)
        unlock(empty);
    unlock(mutex);
    …
```

- **Problem #3**: It is very hard to generalize this use of locks to more complicated programs
  - The "I lock / you unlock" handoff is known to be very difficult to get right, especially with more than 2 threads and more complex patterns

Using a Blocking Lock???

```
queue_t queue;
lock_t mutex;
blocking_lock_t empty(LOCKED);
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(empty);
    unlock(mutex);
    …
```

```
// consumer
  …
  lock(empty);
  lock(mutex);
  x = remove(queue);
  if (queue.size != 0)
      unlock(empty);
  unlock(mutex);
  …
```

Bottom-line:

Using locks for communication is no good!

We need another abstraction

So what do we do now?

- What we need is
  - A way for a thread to wait for “an event” without spinning
  - A way for a thread to signal that “the event” has happened
- Such wait and signal functionalities can be easily implemented with help from the OS
  - The OS can simply move the threads between the READY and the BLOCKED states at will
- There is a troubling similarity with blocking locks, which gets **a lot** of people confused
- If you want to avoid philosophical doubt just remember: locks are for mutual exclusion, while here we’re talking about inter-thread communication
  - And yes, for blocking locks (not spinlocks!), the implementation happens to be almost the same

Condition Variables

- **The basic abstraction for thread communication is a condition variable** (not a great name for it)
- A condition variable supports three operations:
  - **wait():** the thread placing this call goes to sleep (put to sleep by the O/S, i.e., no longer using the CPU)
  - **signal():** when this call is placed, one of the sleeping threads, **if any**, wakes up
  - **broadcast():** when this call is placed, ALL the sleeping threads, **if any**, wake up

Condition Variables

- A good way to think of a condition variable is *a queue of blocked threads*
  - Which is really how the OS implements it anyway
  - A thread gets context-switched out and its PCB is placed in the condition variable's queue
  - It will eventually make its way back to the Ready Queue
- **Important:** when thread A calls signal on a condition variable on which thread B is waiting, **thread B doesn't run immediately at all!**
  - First, thread A gets to finish its time quantum
  - Then, all the threads in the Ready Queue ahead of thread B get to do their time quanta
  - Then, finally, thread B gets to do its time quantum
- Let's look at our producer/consumer with condition variables…

Producer/Consumer?

```
queue_t queue;
lock_t mutex;
cond_t cond;
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
  …
 if (queue.size == 0) {
     wait(cond);
  }
  lock(mutex);
  x = remove(queue);
  unlock(mutex);
  …
```

Producer/Consumer?

```
queue_t queue;
lock_t mutex;
cond_t cond;
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
  …
 if (queue.size == 0) {
     wait(cond);
  }
  lock(mutex);
  x = remove(queue);
  unlock(mutex);
  …
```

- Unfortunately, this doesn't work with **2 consumers**
  - i.e., a consumer might call remove() on an empty queue
- Anybody sees why?

Producer/Consumer?

```
queue_t queue;
lock_t mutex;
cond_t cond;
```

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
  …
 if (queue.size == 0) {
     wait(cond);
  }
  lock(mutex);
  x = remove(queue);
  unlock(mutex);
  …
```

- Problem with two consumers: race condition on queue.size because "testing" following by "doing" is not atomic!
  - The queue has one element in it
  - **Both consumers see the queue as non-empty**
  - They both move on to the critical section one after the other
  - The second one ends up calling remove on an empty queue

This is a Common Bug

- We have seen this several times already: the action of "testing and then setting" is not atomic in code written as: if (some condition) { do_something }
- Back in 1993, 6 cancer patients were overdosed with chemotherapy medicine and died (the "Therac-25" incident)
- From an investigation:
  - "It is clear from the AECL documentation on the modifications that the software allows concurrent access to shared memory that there is no real synchronization aside from data that are stored in shared variables and that the test and set for such variables are not indivisible operations. **Race conditions resulting from this implementation of multitasking played an important part in the accidents.**"

Strict Producer/Consumer

- In our example, having a consumer call remove() on an empty queue once is probably not a big deal and we could live with it
- But for other applications it may not be a good idea
  - the consumer does an update of a database
  - the consumer does a write to disk
  - the consumer sends/receives data from the network to answer customer transactions for on-line reservations
  - ....
- So in a true Producer/Consumer model, a consumer must **never** be awakened and consume when the queue is empty
- We need to remove the race condition on the previous slide
- **Question**: how do we remove race conditions?
- **Answer**: with a lock!

Wait/Signal

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
    …
    lock(mutex);
    if (queue.size == 0) {
      wait(cond);
    }
    x = remove(queue);
    unlock(mutex);
    …
    …
```

moved up

- We just moved the statement “lock(mutex)” before the queue size check
- But now we have a new problem...anybody sees it?
  - Hint: think of what happens if the consumer starts first

Wait/Signal

```
// producer
   …
   lock(mutex);
   insert(queue, generate());
   unlock(mutex);
   signal(cond);
   …
```

```
// consumer
   …
   lock(mutex);          <- moved up
   if (queue.size == 0) {
     wait(cond);
   }
   x = remove(queue);
   unlock(mutex);
   …
   …
```

- We now have a new problem: Deadlock
  - The consumer acquires the lock first and waits
  - The producer can never put anything in the queue!
- This is a classic deadlock, but not due to calls to lock/unlock being misplaced
  - Not the same as the classic “lock(lock1); lock(lock2)” and “lock(lock2); lock(lock1);” deadlock bug

Cond. Variables and Locks

- We face a conundrum
  - If we put the lock() after the wait() we have a race condition
  - If we put the lock() before the wait() we have a deadlock

Cond. Variables and Locks

- We face a conundrum
  - If we put the lock() after the wait() we have a race condition
  - If we put the lock() before the wait() we have a deadlock
- What we really want is the following behavior:
  - If a thread holds a lock and calls wait(), then it, somehow, releases the lock while it's blocked!
  - Then, when it wakes up, it, somehow, re-acquires the lock
- Real-life Metaphor:
  - Your family has one car, and the key's on the kitchen counter whenever the car is not in use
  - You **grab** the key to go pick up your friend and get into the car
  - Then you check whether your friend has texted you their location, and they haven't yet….
  - So you **wait** in the car, and in the meantime, no other family member can use the car!
  - "grabbing your phone" should FORCE you to "put the keys back on the counter", the same way "waiting for a condition variable" forces you to "release the lock"

Cond. Variables and Locks

- Luckily we're not in real life but in computer life, so we can just write the code to do what we want :)
- We modify the API as follows: **`wait(cond, lock)`**
  - **`cond:`** what to "wait on"
  - **`lock:`** what to release and re-acquire
  - **`wait()`** can only be called if the lock is acquired
- Pseudo-code of **`wait()`**:

```
void wait(cond_t c, lock_t m) {
    . . .
    unlock(m);        // release the lock
    some_syscall();   // ask the OS to put me to sleep
    lock(m);          // re-acquire the lock
    . . .
    return;
}
```

- **No thread can block WHILE holding the lock**

Wait/Signal

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
    …
    lock(mutex);
    if (queue.size == 0) {
      wait(cond, mutex);
    }
    x = remove(queue);
    unlock(mutex);
    …
```

- A consumer thread calls lock() before checking the size, and if it gets into the if, then wait() releases the lock and will reacquires it whenever the thread gets scheduled again

Wait/Signal

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
    …
    lock(mutex);
    if (queue.size == 0) {
      wait(cond, mutex);
    }
    x = remove(queue);
    unlock(mutex);
    …
```

- A consumer thread calls lock() before checking the size, and if it gets into the if, then wait() releases the lock and will reacquires it whenever the thread gets scheduled again
- There is **still** something wrong...anybody sees it?
  - Hint: A problem with two consumers...
    - It's subtle but very well-known

Wait/Signal

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
    …
    lock(mutex);
    if (queue.size == 0) {
      wait(cond, mutex);
    }
    x = remove(queue);
    unlock(mutex);
    …
```

- There could be a remove on an empty queue!!
  - A consumer gets the lock, the queue is empty, the consumer releases the lock and goes to sleep
  - The producer puts an element in the queue and gets context-switched out right before it calls signal()
  - A **second** consumer shows up, sees the queue as non-empty, and calls remove
  - The producer resumes, and calls signal(), putting the 1st consumer back into the ready queue
  - The first consumer wakes up and calls remove() on empty queue!

How can we fix this?

- The problem is that the producer calls signal() not immediately after putting an item in the queue
- Therefore, the blocked consumer wakes up after another consumer has had time to grab the item that was “intended” for the blocked consumer
- So, perhaps we can put the call to signal() inside the critical section???
  - Even though It seemed natural to first unlock the lock, and then call signal, since after all the first thing the consumer will have to do after waking up is reacquire the lock

Moving signal()?

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    signal(cond);
    unlock(mutex);
    …
```

```
// consumer
    …
    lock(mutex);
    if (queue.size == 0) {
      wait(cond, mutex);
    }
    x = remove(queue);
    unlock(mutex);
    …
```

- Above we've moved the call to signal() before the call to unlock()
- But **calling signal just puts the consumer back on the ready queue**, and the consumer doesn't necessarily run right now
- In fact, another consumer that's was on the ready queue will run first!
- **So this does not fix anything**
  - In fact, it's possible that calling unlock() and then signal() could be a bit more efficient (shorter critical section)

So, how can we fix this?????

- It doesn't matter where we call signal()
- The problem remains: as a consumer I might be awakened because the queue is not empty, but by the time I run on the CPU the queue could have become empty!
- This is called a "spurious wake-up"
  - Real-life metaphor: You're in a coffee who and you asked the barista to come wake you up when the bathroom is free, but by the time you get to the bathroom somebody has gotten in it in the meantime
- The way to avoid spurious wake-ups for producer-consumer is to use a **while** loop!

A while loop!

```
// producer
    …
    lock(mutex);
    insert(queue, generate());
    unlock(mutex);
    signal(cond);
    …
```

```
// consumer
    …
    lock(mutex);
    while (queue.size == 0) {
      wait(cond, mutex);
    }
    x = remove(queue);
    unlock(mutex);
    …
```

- Solution: Use a **while** loop instead of an if statement
- If a consumer is awakened but the queue is in fact empty (because another consumer has already consumed the last element in the queue), it will loop, check again, and wait again
- Basically, don't trust the "wake up you're good to go" blindly, always double check that you're really good to go
  - Because while you were sleeping, all kinds of stuff could have happened

Finally!!!

- So, now we have a clean implementation of the producer-consumer with locks and condition variables
- The pattern in the previous program is a classic and can be reused in many applications
  - Always combine condition variables with locks
  - Always do a while loop around a wait() (unless you really know there is a single consumer)
- Note how difficult it is to reason about concurrency
- This is why we always very much hope that we can re-use a known pattern, e.g., producer/consumer
  - Getting creative with concurrency can be appealing, but is often fraught with peril
- If you can make your program be producer-consumer-like, do it

A Bounded Queue

- The typical producer-consumer model uses a bounded queue: there cannot be more than N elements in the queue
  - Producers may wait because the queue is full
  - Consumers may wait because the queue is empty

- Let's look at how one can write this program…

Wait/Signal

```
queue_t queue;
lock_t mutex;
cond_t cond_not_empty, cond_not_full;
```

```
// producer
  …
  lock(mutex);
  while(queue.size >= N) {
    wait(cond_not_full, mutex);
  }
  insert(queue, generate());
  unlock(mutex);
  signal(cond_not_empty);
  …
```

```
// consumer
  …
  lock(mutex);
  while (queue.size == 0) {
    wait(cond_not_empty, mutex);
  }
  x = remove(queue);
  unlock(mutex);
  signal(cond_not_full);
  …
```

Wait/Signal

```
queue_t queue;
lock_t mutex;
cond_t cond_not_empty, cond_not_full;
```

```
// producer
  …
  lock(mutex);
  while(queue.size >= N) {
    wait(cond_not_full, mutex);
  }
  insert(queue, generate());
  unlock(mutex);
  signal(cond_not_empty);
  …
```

```
// consumer
  …
  lock(mutex);
  while (queue.size == 0) {
    wait(cond_not_empty, mutex);
  }
  x = remove(queue);
  unlock(mutex);
  signal(cond_not_full);
  …
```

- Note that picking good names for the locks and the condition variable is key to program readability

A Barrier

- Say you want to have threads wait for each other at some point in the code
  - Once a thread first reaches some point in the code, then it blocks until all the other threads reach that same point
- This is called a “barrier”
- How can we implement this with locks and condition variables?
- One easy option: keep track of how many threads have arrived at the barrier so far
  - If I am not the last one, increment the count and block
  - If I am the last one, unblock everybody
- Let’s try to come up with pseudo-code together before we look at the solution…

Example: Barrier

```
int count = 0;
lock_t mutex;
cond_t cond;
```

```
void barrier() {
   lock(mutex);
   count++;
   if (count == num_threads) {
     broadcast(cond);
   } else {
     wait(cond, mutex);
   }
  unlock(mutex);
}
```

Conclusion

- At this point, we have everything we need to write concurrent programs
  - Locks for mutual exclusion
    - Spin, blocking, hybrid
  - Condition variables for thread synchronization and/or communication without busy loops
- Next up: Doing condition variables in Java
- In the meantime, let's look at Homework Assignment #5 (individual, pencil-and-paper)…