

Producer/Consumer & Condition Variables

ICS432 - Fall 2008
Concurrent and High-Performance
Programming

Henri Casanova (henric@hawaii.edu)

The Shared Queue Example

- Let's take another look at the example from the previous lecture notes
- We have a queue of integer elements
 - insert()
 - remove()
- We have two threads
 - producer: puts integers
 - consumer: removes integers
- Let's look at our implementation, which uses locks

Simple Solution

lock_t mutex; // global variable

```
void producer() {
  int x;
  while(1) {
    x = generate();
    lock(mutex);
    insert(list,x);
    unlock(mutex);
  }
}
```

```
void consumer() {
  int x;
  while(1) {
    lock(mutex);
    x = remove(list);
    unlock(mutex);
    process(x);
  }
}
```

Producer/Consumer

- A common concurrent application model is the **producer/consumer** model
 - A producer thread puts "item" in some "structure"
 - A consumer removes "item" from some structure
- Our previous example was not a true producer/consumer implementation
- The consumer should be waiting for items to be put in the queue
 - In our program, remove() returns -1 when the queue is empty

Producer/Consumer

lock_t lock; // global variable

```
void producer() {
  int x;
  while(1) {
    x = generate();
    lock(lock);
    insert(list,x);
    unlock(lock);
  }
}
```

```
void consumer() {
  int x = -1;
  while(1) {
    lock(lock);
    x = remove(list);
    unlock(lock);
    if (x == -1)
      continue;
    process(x);
  }
}
```

Busy Wait

- The previous implementation uses a **busy wait**
- The Consumer keeps trying to remove an item
- This implementation is correct but **wasteful**
 - When the queue is empty, the consumer uses the CPU to place a function call and do two tests, in an infinite loop
- In some cases, this may not affect overall performance
 - Example: the producer is waiting for the disk/network, and the CPU is not really used anyway and therefore not wasted
- But if anything useful may be done with the CPU while the consumer is "waiting", performance can suffer
 - Typically, there are many processes/threads that could benefit from a few more cycles, e.g., on a server
 - Besides, busy waiting increases heat and power consumption, which may be an issue on embedded systems for instance
- **Bottom line:** busy waits are not advised and frowned upon

“Less Busy” Wait?

- A simple idea to avoid wasting CPU cycles is to make the thread sleep
- The problem: how to pick the time to sleep?
 - Large: program is not responsive
 - Small: CPU cycles wasted
- How about using locks to block the thread?

```
void consumer() {
    int x = -1;
    while(1) {
        lock(mutex);
        x = remove(list);
        unlock(mutex);
        if (x == -1) {
            sleep(10); // sleeps for 10 ms
            continue;
        }
        process(x);
    }
}
```

Using Locks for Blocking

```
lock_t mutex, empty;

void producer() {
    while(1) {
        lock(mutex);
        insert(list, generate());
        unlock(empty);
        unlock(&mutex);
    }
}

void consumer() {
    int x;
    while(1) {
        lock(empty);
        lock(mutex);
        x = remove(list);
        if (list.size != 0)
            unlock(empty);
        else
            continue;
        unlock(mutex);
    }
}
```

- ✓ Lock “empty” must be locked before both threads start
- ✓ Assumes that one can unlock an unlocked lock safely

Using Locks for Blocking

- The previous solution is rather simple
 - Although one could imagine that in more complex situations the use of multiple locks could be dangerous (e.g., deadlocks) and to be avoided
- The drawback is that locks are not an efficient way to wait for an event that may occur in a “long” time
 - Spin locks waste CPU cycles
 - They can use exponential back-off though
 - But then the program may lack responsiveness
- Essentially, we’re in the same situation we were in before and nothing is really solved
- What we need is
 - a way for a thread to wait for something without wasting CPU cycles
 - a way for a thread to wake up another thread with some signal
- Such “wait” and “signal” functionalities can be easily implemented with help from the operating system

Condition Variables

- A basic abstraction for this kind of thread synchronization is a **condition variable**
- A condition variable supports two operations:
 - **wait(cond)**: the thread placing this call goes to sleep (put to sleep by the O/S)
 - **signal(cond)**: when this call is placed, one of the sleeping threads wakes up
- A good way to think of a condition variable is a *queue of blocked threads*
- Calling signal() on a condition variable with no sleeping threads has no effect
- Let’s look at our producer/consumer with condition variables

Wait/Signal

```
void producer() {
    while(1) {
        lock(mutex);
        insert(list, generate());
        unlock(mutex);
        signal();
    }
}
```

```
void consumer() {
    int x;
    while(1) {
        if (list.size == 0) {
            wait();
        }
        lock(mutex);
        x = remove(list);
        unlock(mutex);
    }
}
```

Wait/Signal

```
void producer() {
    while(1) {
        lock(mutex);
        insert(list, generate());
        unlock(mutex);
        signal();
    }
}
```

```
void consumer() {
    int x;
    while(1) {
        if (list.size == 0) {
            wait();
        }
        lock(mutex);
        x = remove(list);
        unlock(mutex);
    }
}
```

- Unfortunately, there is a small problem if we have two or more consumers: **race condition** on list.size!
 - Both threads think the list is empty
 - They both move on
 - One of them ends up calling remove() on an empty list

Strict Producer/Consumer

- In our example, having a consumer call remove() on an empty list 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 on a database
 - the consumer does a write to disk
 - the consumer sends/receives data from the network
 - etc.
- So in a true Producer/Consumer model, a consumer must never be awakened and consume when there is nothing to consume
- So, we need to remove our race condition
- Question: how do we remove race conditions?
- Answer: with a lock!

Wait/Signal

```
void producer() {
    while(1) {
        lock(mutex);
        insert(list, generate());
        unlock(mutex);
        signal();
    }
}
```

```
void consumer() {
    int x;
    while(1) {
        lock(mutex);
        if (list.size == 0) {
            wait();
        }
        x = remove(list);
        unlock(mutex);
    }
}
```

- We just moved the statement "lock(mutex)" at the beginning of the while(1) loop
- But now we have a new problem...

Wait/Signal

```
void producer() {
    while(1) {
        lock(mutex);
        insert(list, generate());
        unlock(mutex);
        signal();
    }
}
```

```
void consumer() {
    int x;
    while(1) {
        lock(mutex);
        if (list.size == 0) {
            wait();
        }
        x = remove(list);
        unlock(mutex);
    }
}
```

- We just moved the statement "lock(mutex)" at the beginning of the while(1) loop
- But now we have a new problem...
- **Deadlock:**
 - The consumer locks the mutex lock and waits
 - The producer can never put anything in the queue!

Cond. Variables and Locks

- At this point 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
 - The solution is to "combine" condition variables and locks
 - We modify the API as follows
 - **wait(cond,mutex)**
 - Unlocks the mutex and go to sleep
 - When waking up, lock the mutex again
- ```
void wait(cond_t c, lock_t m) {
 ...
 unlock(m);
 ... // sleep
 lock(m);
 return;
}
```
- This way, **no thread is sleeping AND holding the lock at the same time**

## Wait/Signal

```
void producer() {
 while(1) {
 lock(mutex);
 insert(list, generate());
 unlock(mutex);
 signal(cond);
 }
}
```

```
void consumer() {
 int x;
 while(1) {
 lock(mutex);
 if (list.size == 0) {
 wait(cond, mutex);
 }
 x = remove(list);
 unlock(mutex);
 }
}
```

- We place the lock() at the beginning of the while(1) loop
- But now the mutex will be unlocked during the call to wait
- There is still something wrong...

## Wait/Signal

```
void producer() {
 while(1) {
 lock(mutex);
 insert(list, generate());
 unlock(mutex);
 signal(cond);
 }
}
```

```
void consumer() {
 int x;
 while(1) {
 lock(mutex);
 if (list.size == 0) {
 wait(cond, mutex);
 }
 x = remove(list);
 unlock(mutex);
 }
}
```

- There could be a remove on an empty queue
  - A consumer gets the lock, the queue is empty, and the consumer sleeps
  - The producer puts an element in the queue and gets interrupted right before it calls signal()
  - A **second** consumer sees the list as non-empty, and calls remove
  - The producer resumes, and calls signal
  - The first consumer wakes up and proceeds with a remove on an empty queue!

## Wait/Signal

```
void producer() {
 while(1) {
 lock(mutex);
 insert(list, generate());
 unlock(mutex);
 signal(cond);
 }
}
```

```
void consumer() {
 int x;
 while(1) {
 lock(mutex);
 while (list.size == 0) {
 wait(cond, mutex);
 }
 x = remove(list);
 unlock(mutex);
 }
}
```

- Solution: Use a **while** loop instead of a conditional
- If a consumer is awakened but the list is in fact empty (because another consumer has already consumed the last element in the queue), it will loop and wait again
- This avoids so-called “spurious wake-ups”
- Typical: ALWAYS use a while loop around a wait()

## Where are we?

- So, now we have a clean implementation of the producer-consumer with locks and condition variables
- The pattern in the previous program is classical and can be reused in many applications
  - Always combine condition variables with locks
  - Always do a while loop around a wait() to avoid spurious wake-ups
- 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

```
void producer() {
 while(1) {
 lock(mutex);
 while(list.size >= N) {
 wait(cond_notfull, mutex);
 }
 insert(list, generate());
 unlock(mutex);
 signal(cond_not_empty);
 }
}
```

```
void consumer() {
 int x;
 while(1) {
 lock(mutex);
 while (list.size == 0) {
 wait(cond_not_empty, mutex);
 }
 x = remove(list);
 unlock(mutex);
 signal(cond_notfull);
 }
}
```

## Conclusion

- At this point, we have everything we need to write concurrent programs
  - **Locks** for efficient mutual exclusion
  - **Condition variables** for non-wasteful thread synchronization and/or communication
- And in fact, this is exactly what's available in C for writing multi-threaded programs
- In the next lecture we're going to look at multi-threading in C and introduce the Pthread API