Please SCAN

Synchronization

Today: condition variables

Friday: reader/writer locks

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic

Locks

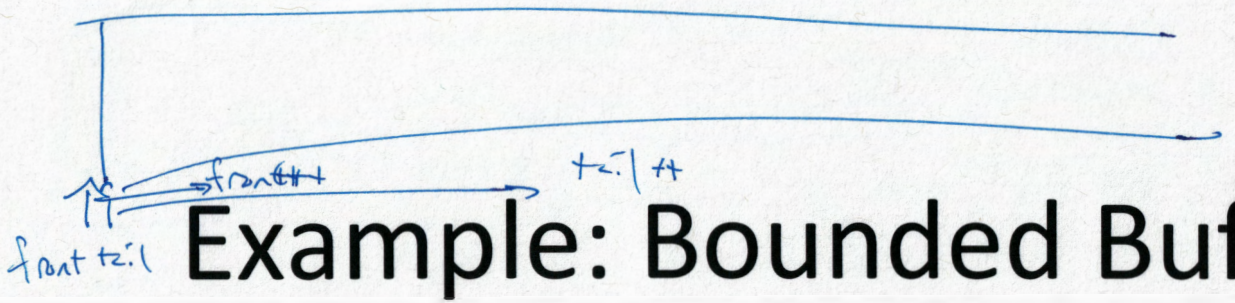
- Lock::acquire
  - wait until lock is free, then take it
- Lock::release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Question

- If tryget returns NULL, do we know the buffer is empty?

- If we poll tryget in a loop, what happens to a thread calling tryput?



Example: Bounded Buffer

```
tryget() {
  lock.acquire();
  item = NULL;
  if (front < tail) {
     item = buf[front % MAX];
     front++;
  }
  lock.release();
  return item;
}
```

```
tryput(item) {
   lock.acquire();
   success = FALSE;
   if ((tail – front) < MAX) {
      buf[tail % MAX] = item;
      tail++;
      success = TRUE;
   }
   lock.release();
   return success;
}
```

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock

- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Condition Variable Design Pattern

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}
```

Hold lock! ASSERT(testSharedState()) sometimes f..?

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state

    // If testSharedState is now true
    cv.signal(&lock);

    // Read/write shared state
    lock.release();
}
```

Example: Bounded Buffer

```
get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(&lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(&lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    while ((tail – front) == MAX) {
        full.wait(&lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(&lock);
    lock.release();
}
```

Initially: front = tail = 0; MAX is buffer capacity

empty/full are condition variables

Question

Does the kth call to get return the kth item put?

Hint: wait must re-acquire the lock after the signaller releases it.

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - tail – front <= MAX
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

A	B
lock X	lock Y
lock Y	lock X

Pre/Post Conditions

```
methodThatWaits() {
   lock.acquire();
   // Pre-condition: State is consistent

   // Read/write shared state

   while (!testSharedState()) {
      cv.wait(&lock);
    }
   // WARNING: shared state may
   // have changed!  But
   // testSharedState is TRUE
   // and pre-condition is true

   // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
   lock.acquire();
   // Pre-condition: State is consistent

   // Read/write shared state

   // If testSharedState is now true
   cv.signal(&lock);

   // NO WARNING: signal keeps lock

   // Read/write shared state
   lock.release();
}
```

Rules for Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Rules for Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait()) {
    condition.Wait(&lock);
}
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Java Manual

When waiting upon a Condition, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) { condition.Wait(lock); }
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()