CMSC 433 Fall 2013
Michael Hicks
(Some slides due to Rance Cleaveland)

UNIVERSITY OF MARYLAND

Lecture 10
State dependency

State-Dependent Actions

- A method's behavior may depend on the current state of the object
  - Recall the idea of a **precondition**: this is a requirement on the arguments and the object's state for expected execution
  - A method may explicitly check its precondition, or silently fail if not satisfied
    - Precondition on **Line.slope** : object is not a vertical line
      - Else throws an exception
    - Precondition on normal **HashMap**: object is not accessed by multiple threads
      - Else will fail in weird ways

Examples of State Dependency

- Operations on collections
  - Can’t remove an element from an empty queue
  - Can’t add element to full buffer
- Operations involving constrained values
  - Can’t withdraw money from empty bank account
- Operations requiring resources
  - Can’t connect to internet if local network is down
- Operations requiring previous operations
  - Can’t read from a file that has not been opened

State-Dependency and Multithreading

- In single-threaded case, one general option for state-dependency: *balking*
  - Operation cannot be performed, so method refuses to perform it
  - This can be achieved by
    - Ignoring the request
    - Raising an exception (or returning a status code)
- In multithreaded applications, other possibilities are available because another thread might change the state of the object
  - *Guarded suspension*
  - *Optimistic retries*

Balking Via Ignoring

```
public class BoundedCounter {
  private int value = 0;
  private int upperBound = 0;

  //INVARIANT:  in all instances 0 <= value <= upperBound

  public synchronized boolean isMaxed () {
    return (value == upperBound);
  }
…

  //Pre:  none
  //Post:  increment value if not maxed; otherwise, do nothing.
  //Exception:  none
  public synchronized void inc () { if (!isMaxed()) ++value; }
}
```

- Recall BoundedCounter
  - inc method does nothing if counter value at maximum value
  - Balking via ignoring!

Balking Via Exceptions

```
public class BoundedBufferException {

  // Invariant:  number of elements is <= maxSize

  private final int maxSize;
  private ArrayList<Object> elements;
  …
  // Pre:  number of elements is below maxSize
  // Post:  elt is added to end of list of elements
  // Exception:  If number of elements is too high, throw exception.
  public synchronized void put (Object elt) throws Exception {
    if (elements.size() < maxSize) { elements.add(elt); }
    else throw new Exception("Put error");
  }
}
```

- Consider BoundedBufferException class
  - Stores elements in a queue
  - If queue is full / empty, put / take methods are not defined!
  - In this case, exceptions raised

Balking Via Return Codes

```
public class BoundedBufferReturnCode {

// Invariant:  number of elements is <= maxSize

  private final int maxSize;
  private ArrayList<Object> elements;

  // Type of return values
  public class ReturnVal { public final Object obj; public final boolean code; … }

  // Pre:  none
  // Post:  if list is nonempty, return first element and true; otherwise,
  //        return null and false
  // Exception:  none
  public synchronized ReturnVal take () {
    if (elements.size() > 0) {
      Object elt = elements.get(0);
      elements.remove(0);
      return new ReturnVal(elt, true);
    }
    else return new ReturnVal(null, false);
  }
}
```

- Consider BoundedBufferReturnCode
  - Inner class defined for return values for put, take
  - Inner class includes object, boolean indicating whether operation concluded successfully

Observations on Balking

- Operations do not block when state is correct (assuming no infinite loops)
- When an operation balks, it is up to class user to determine what to do
  - Detect "ignoring"
  - Handle exception or act on return code

Guarded Suspension

- For bounded buffers in a multithreaded environment:
  - If the buffer is empty now, a `take()` operation cannot complete
  - Another thread could deposit an element later, and a `take()` could succeed!
- In guarded-suspension approaches to state-dependent actions, threads "go to sleep" until the actions they want to perform are possible
- Needed mechanisms
  - … for going to sleep ("suspend")
  - … for waking up ("resume")

Busy-Waiting

- An old-fashioned mechanism for suspend/resume
  - Use a while loop to test for enabledness of state-dependent action
  - When true: exit loop, perform action
  - E.g.

    `while (!`*enabled*`) ;` // Suspend via spinning

    // Resume
- Considerations
  - Consumes computing resources
  - Enabled-ness condition might become false after loop terminates, so synchronization should be used

`wait()`/`notify()`/`notifyAll()`

- A more modern mechanism in Java for suspending / resuming
  - To suspend, a thread performs a `wait()`
  - Other threads perform `notify()` / `notifyAll()` to enable resumption of suspended threads
- Benefits
  - No consumption of cycles while suspended
  - Synchronization taken care of (we will see how in a moment)
- Dangers
  - A suspended thread is dependent on other threads to wake it up
  - If no other thread performs `notify()` / `notifyAll()`, then thread sleeps forever

How `wait` / `notify` / `notifyAll` Work

- In addition to its intrinsic lock, every Java object as a *wait-set*
  - Wait-set contains threads that are waiting on the object
  - Threads in the wait-set are suspended
- Threads enter wait-set of object `obj` by performing `obj.wait();`
  - Thread is added to wait-set of `obj`
  - Thread releases all its locks on `obj`, but not other locks
  - Thread is then suspended
- Other threads can release waiting threads by performing `obj.notify()` or `obj.notifyAll()`
  - `obj.notify():` one waiting thread selected "at random" for resumption
  - `obj.notifyAll():` all waiting threads selected for resumption
- When thread selected for resumption the following happens behind the scenes
  - It is removed from wait-set
  - It tries to reacquire its locks on the object it was waiting on
    - If it succeeds, it proceeds
    - Otherwise, it blocks waiting for the lock to become available
  - **Note: thread should double-check condition for state-dependent action when it resumes!**

Example: `BoundedBufferWait`

```
// Pre:  number of elements is below maxSize
// Post:  elt is added to end of elements, waiting threads notified
// Exception:  If number of elements is too high, suspend.
public synchronized void put (Object elt) throws InterruptedException {
  while (elements.size() == maxSize) wait();
  elements.add(elt);
  notifyAll();
}
```

- In `put()` / `take()` operations, `wait()` executed when state does not allow action
- When an operation succeeds, waiting threads notified
- When a thread wakes up, it must check that condition it was waiting for holds!
  - **This is why loop is used with `wait()` inside. You should do this always unless you have an ironclad argument for not needing a loop!**
  - Just because a thread is resumed does not mean it is safe to proceed
- When a thread modifies the state of the object (e.g. by successfully adding an element) it must notify sleeping threads
- `InterruptedException`?
  - `wait()` is a blocking operation, meaning it could never terminate
  - Any thread can be interrupted (a topic for a later date) by another thread
  - This exception is raised in this case, because a blocked thread may need some cleanup

`notify()` / `notifyAll()`

- Consider `take()` operation in `BoundedBufferWait`

```
public synchronized Object take () throws
InterruptedException {
  while (elements.size() == 0)
    wait();
  Object elt = elements.get(0);
  elements.remove(0);
  notifyAll();
  return elt;
}
```

- Doesn’t this introduce a race condition?
  - `notifyAll()` called before return of element
  - Could this cause problems?
- Answer: no
  - `notify()` / `notifyAll()` do not release locks
  - So lock on buffer only released when `take()` operation terminates

Why `notifyAll()`?

- `put()` / `take()` use `notifyAll()` rather than `notify()`
  - It seems wasteful to wake everyone up!
  - Why not just wake up one thread?
- There is a reason!
  - Waiting threads are potential concerned with different conditions
    - Putters are waiting for buffer not to be full
    - Takers are waiting for buffer not to be empty
  - If you use `notify()`, you only wake up one thread
  - If you wake up the wrong thread, you can wind up in a deadlock!

`notify()` and Deadlock

- Suppose `put()`, `take()` reimplemented with `notify()` rather than `notifyAll()`, e.g.

```
public synchronized void put (Object elt) throws
InterruptedException {
    while (elements.size() == maxSize) wait();
    elements.add(elt);
    notify();
  }
```

- Now supposed we have:
  - `BoundedBufferWait` with `maxSize == 1`
  - Four threads T_1, …, T_4
  - A deadlock can happen!

Deadlock Scenario

Time	T1	T2	T3	T4	elements.size()	Wait-set
0	take (w)				0	T1
1		take (w)			0	T1, T2
2			put		1	T2
3			put (w)		1	T2, T3
4				put (w)	1	T2, T3, T4
5	take (0)				0	T3, T4
6	take (w)				0	T1, T3, T4
7		take (w)			0	**T1, T3, T4, T2**

<u>Legend</u>
op(w) – operation waits / "rewaits"
op(i) – operation begun at time i completes
op – operation begins and completes without waiting

When To Use `notify()`

- Only use `notify()` if
  - Every thread in wait-set is guaranteed to be waiting on same condition
  - Condition is guaranteed to be true when thread executing `notify()` surrenders its lock on object
- Otherwise: use `notifyAll()`

Timed Waiting

- Problem with `wait()`: unbounded waiting
  - You do not know how long a thread might wait before being able to continue
  - In some applications this leads to unacceptable performance variability
- Variant: `wait(long millis)`
  - Wait for at least specified # of milliseconds
  - At time-out, exit wait-set
  - How do you tell if exit from wait-set is due to notification or timeout?
    - You don't
    - You have to check this yourself
- Intermediate between balking, guarded suspension

Example: BoundedBufferTimedWaiting

```
public synchronized void put (Object elt, long allowedDuration)
  throws Exception, InterruptedException {
  long startTime = System.currentTimeMillis();
  long timeLeft = allowedDuration;
  while (elements.size() == maxSize) {
    wait(timeLeft);
    // Check if buffer has space
    if (elements.size() < maxSize) {
      elements.add(elt);
      notifyAll();
      break; // Break out of loop
    }
    else {
      // Check if time has expired
      long elapsed = System.currentTimeMillis() - startTime;
      timeLeft -= elapsed;
      if (timeLeft <= 0) throw new Exception ("Timeout");
    }
}
```

- Argument to put includes upper bound on time to wait
- The handling of resumption includes a check for how much time has elapsed
- When “re-waiting” the new timeout value must be recalculated based on how waiting has already occurred!

Nested Monitor Lockout

- Suppose we want to build a layer on top of `BoundedBufferWait`
  - New class should not insert null objects into buffer
  - A new invariant is being defined: buffer should contain no null objects
- An approach: **instance confinement**
  - Make a new class for enforcing new invariant
  - Include a private field containing a `BoundedBufferWait` object
  - Implement a new put method to handle null objects
- Does it work?

BoundedBufferWaitNoNull

```
public class BoundedBufferWaitNoNull {

  private final BoundedBufferWait buffer;

  BoundedBufferWaitNoNull (int capacity) {
    buffer = new BoundedBufferWait(capacity);
  }

  public synchronized boolean put (Object elt) throws InterruptedException {
    if (elt != null) {
      buffer.put(elt);
      return true;
    }
    else return false;
  }

  public synchronized Object take () throws InterruptedException {
    return buffer.take();
  }
}
```

`BoundedBufferWaitNoNull` Does Not Work

- What happens if a thread calls take on a `BoundedBufferWaitNoNull` object when the buffer is empty?
  - Object calls `buffer.take()`
  - Since buffer is empty, thread enters wait-set, releases lock on inner `BoundedBufferWait` object
  - Thread still holds lock on BBWNN object, though
- Deadlock!
  - This phenomenon is called *Nested Monitor Lockout*
  - Issue is that lock is held on outer object even though waiting is occurring on inner object
  - While outer-object lock is held, no other thread can use it!

Solving Nested Monitor Lockout

- Don’t synchronize in outer class

  But sometimes you need to, in order to preserve new invariants

- Reprogram inner class so that object on which locking is to be performed is provided as argument to inner-class constructor
  - Requires reprogramming methods in inner so that this lock is used
  - Solves problem, at cost of rework of inner class

BoundedBufferWaitLockParam

```
public class BoundedBufferWaitLockParam {

  final int maxSize;
  final ArrayList<Object> elements;
  final Object syncLock;

  BoundedBufferWaitLockParam (Object lock, int maxSize) {
    this.maxSize = maxSize;
    elements = new ArrayList<Object> ();
    syncLock = lock;
  }
…
  public void put (Object elt) throws InterruptedException {
    synchronized (syncLock) {
      while (elements.size() == maxSize) syncLock.wait();
      elements.add(elt);
      syncLock.notifyAll();
    }
  }
…
}
```

Optimistic Retrying

- Another mechanism for handling state-dependency
- Implement an operation *optimistically* as follows:
  - Make copy of current state
  - Apply operation if it is applicable
  - "Commit" updated state if the the copy still matches the current state, else abort or retry
  - Copying and commit require locking; operation doesn't
- Why even do this?
  - Locking is expensive
  - If operations "usually succeed", and contention for object is low, it may be more efficient to do this
  - ConcurrentHashMap implemented with this technique

Example: `BoundedCounterOptimistic`

```
// State copying method
public synchronized int current () { return value; }

// Commit method
public synchronized boolean commit (int oldState, int newState) {
  if (value == oldState) {
    value = newState;
    return true;
  }
  else return false;
}

public void inc () {
  for (;;) { // Retry-based
    int currentState = current();
    if ((currentState < upperBound) && (commit(currentState, currentState+1))) break;
    else Thread.yield();
  }
}
```

- Only state-copying, commit methods are synchronized!
  - Other methods call these, also rely on thread-confinement due to local variables
  - For more complicated classes state copying can be performed piecemeal, so long a invariants respected
- New state commitment is simple in this application
  - With more complex objects, need to ensure that creation of new state does not induce changes that cannot be undone