

Lecture 38: Java Concurrency

CSC 131
Fall, 2011

Kim Bruce

Synchronized blocks

- Control access w/ synchronized blocks:
 - `synchronized(someObj){...}`
 - Must hold lock to access. Release when exit.
- Synchronized methods:
 - Implicitly use "this" as lock on whole object

Shared Variables

- Variables read/written by more than one process are vulnerable to race conditions.
 - Even `++n` is vulnerable, as not atomic
 - But there are "atomic" types like `AtomicInteger`
- If multiple threads access the same mutable state variable you have two options:
 - Make the state variable immutable
 - Use synchronization whenever accessing the state variable

Shared Variables

- Visibility of changes:
 - If one thread executes synchronized block, and then another thread enters a block with same lock, then current values of variables accessible by first are visible to second when acquires lock
 - Without synchronization, no guarantees!
 - May reorder, may be in cache or register or ...
- If synchronization not necessary, then label vble as `volatile` to force changes to be visible

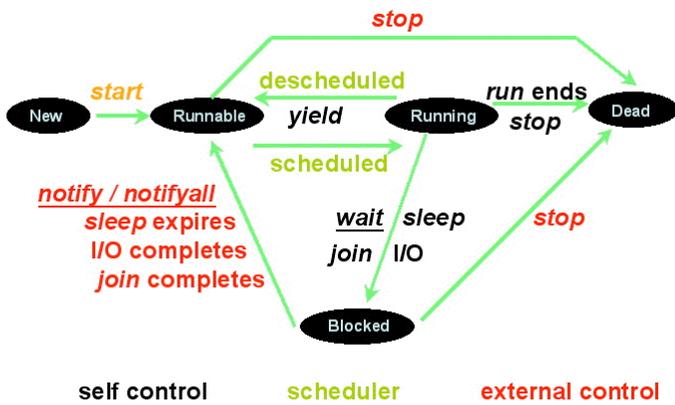
Conditional Waiting

- Every object has a wait set
- `wait()`: release lock & pause until another thread calls `notify` or `notifyAll`.
- `notify()`, `notifyAll()`: wake up waiting threads, which try to grab lock
 - *Can only be used in synchronized code*
 - *Notify wakes up single thread -- arbitrary choice*
 - *NotifyAll wakes up all waiting threads*
 - *Much better than busy-waiting (spin-locks)*

Thread States in Java

- New -- declared, but not yet started
- Runnable -- ready to run
- Running -- currently running
- Blocked -- on I/O, wait on monitor, sleep, join
- Dead -- run has ended

Concurrency in Java



Monitor in Java

```
public class BoundedBuffer {
    protected int numSlots;
    private int[] buffer;
    private int takeOut = 0, putIn = 0;
    private int count=0;

    public BoundedBuffer(int numSlots) {
        if(numSlots < 0) {
            throw new IllegalArgumentException(
                "numSlots <= 0");
        }
        this.numSlots = numSlots;
        buffer = new int[numSlots];
    }
}
```

From Mitchell, hmwk 14.7

```
public synchronized void put(int value)
    throws InterruptedException {
    while (count == numSlots) wait();
    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    count++;
    notifyAll();
}

public synchronized int get()
    throws InterruptedException {
    while (count == 0) wait();
    int value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--;
    notifyAll();
    return value;
}
```

Java Critique

- Brinch Hansen - designer w/Hoare of Monitors hates Java concurrency!
 - Doesn't require programmer to have all methods synchronized,
 - can leave instance variables accessible w/out going through synchronized methods, it is easy to mess up access w/concurrent programs.
 - Felt that should have had a monitor class that would only allow synchronized methods.

Java Threads

- Portable since part of language
 - Easier to use than C system calls
 - Garbage collector runs in separate thread
- Difficult to combine sequential/concurrent code
 - Using sequential code in concurrent -- may not work
 - Java collection classes have synchronized wrappers!
- Using concurrent in sequential programs
 - Useless synchronization
 - 10-20% useless overhead

Rough Spots

- Fairness not guaranteed
 - Choose arbitrarily among equal priority threads
- Wait set is not FIFO queue
 - notifyAll notifies all waiting threads, not necessarily highest priority, longest-waiting, etc.
- Nested monitor problem can cause deadlock.

Nested Monitor Lockout Problem

```
class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
        synchronized(list) {
            list.addLast(x); notify();
        }
    }
    public synchronized Object pop() {
        synchronized(list) {
            if (list.size() <= 0) wait();
            return list.removeLast();
        }
    }
}
```

Releases lock on Stack object but not lock on list; a push from another thread will deadlock

Concurrent Garbage Collector

- How much concurrency?
- Need to stop thread while mark and sweep
- Copy collector may not need to stop all program threads
- Problem
 - Program thread may change objects during collection
- Solution
 - Prevent read/write to that memory area
- Details subtle; generational, copying GC

Java 5: util.concurrent

- Doug Lea utility classes
 - A few general purpose interfaces
 - Implementations tested over several years
- Principal interfaces & implementations
 - Sync -- protocols to acquire and release locks,
 - e.g. Semaphore w/ acquire, release methods
 - BlockingQueue -- classes to insert and delete objects
 - support put, take that block (like bounded buffer)
 - Executor -- executes Runnable tasks
 - You provide control of threads

Java 5 Concurrency Features

```
class BoundedBuffer {    <- array based queue
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

Java 5 Concurrency cont.

```
public Object take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
```

- Advantage: Separate queues for nonEmpty and nonFull conditions on same lock.

More Message Passing

Ada Tasks

- Synchronous message passing
- Tasks have some features of monitors
- Exports entry names (w/ parameters)
- Entry names have FIFO queues

Accepting an entry

```
select
  [when <cond> =>] <select alternative>
  {or [when <cond> =>] <select alternative>}
  [else <statements>]
end select
```

```
task body Buffer is
  MaxBufferSize: constant INTEGER := 50;
  Store: array(1..MaxBufferSize) of CHARACTER;
  BufferStart: INTEGER := 1;
  BufferEnd: INTEGER := 0;
  BufferSize: INTEGER := 0;
begin
  loop
    select
      when BufferSize < MaxBufferSize =>
        accept insert(ch: in CHARACTER) do
          Store(BufferEnd) := ch;
        end insert;
        BufferSize := BufferSize + 1;
        BufferEnd := BufferEnd mod MaxBufferSize + 1;
      or when BufferSize > 0 =>
        accept delete(ch: out CHARACTER) do
          ch := Store(BufferStart);
        end delete;
        BufferSize := BufferSize - 1;
        BufferStart := BufferStart mod MaxBufferSize + 1;
      or
        accept more (notEmpty: out BOOLEAN) do
          notEmpty := BufferSize > 0;
        end more;
      or
        terminate;
    end select;
  end loop;
end Buffer;
```

*Caller only blocked
in accept.
but only one entry
can be executed at a
time.*