Lecture 07:
Advanced Thread programming

Software System Components 2
Behzad Bordbar
School of Computer Science, University of Birmingham, UK
Recap

- How to deal with race condition in Java
- Using `synchronised`
  - Mutual exclusion over a class
  - Mutual exclusion over a method
  - Mutual exclusion over a snippet of code
- Locking using `Lock` interface
Contents

- Reenterancy
- Explicit Lock vs. Intrinsic lock
- Limitations of intrinsic lock
- volatile
- Atomic variables
- Thread safety
- Condition interface
Reenterancy of synchronized

(is also called *intrinsic* lock)

- When a thread requests a lock that is already held by another thread, it gets blocked.
- *Intrinsic* locks are reentrant, if a thread tries to acquire a lock that it already holds, the request succeeds.
- Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.
Reenterancy of synchronize (ctd)

- Reentrancy associate with each lock:
  - acquisition count
  - owning thread
- When a thread acquires a previously unheld lock, the JVM records the owner and sets the acquisition count to one.
- If that same thread acquires the lock again, the count is incremented, and
- when the owning thread exits the synchronized block, the count is decremented. When the count reaches zero, the lock is released.
Explicit Locks: ReentrantLock

- ReentrantLock implements Lock, new to Java 5
- ReentrantLock provides all capabilities provided by intrinsic lock (synchronized):
  - mutual exclusion and memory-visibility guarantees as synchronized.
  - Acquiring a ReentrantLock behaves the same as entering a synchronized block,
  - Releasing a ReentrantLock behaves the same as exiting a synchronized block.

- But why a new mechanism?
- Intrinsic lock has some limitations
Limitations of intrinsic lock

- It is not possible to interrupt a thread waiting to acquire a lock, (Interruptible Lock Acquisition in explicit lock)
- It is not possible to attempt to acquire a lock without being willing to wait for it forever (Polled Timed Lock Acquisition in explicit lock Java 5.0).
- Intrinsic locks also must be released in the same block of code in which they are acquired; this simplifies coding and interacts with exception handling, but makes non-block structured locking impossible (Non-block-structured Locking in explicit lock)

However, in explicit lock failing to use finally to release the lock makes the debugging very hard

See the API for new features.
Volatile

- It is not possible to designate an attribute as synchronized.
- Suppose a class with an attribute `value` which is accessed via a synchronized method of the class. But what if other objects access it directly via another thread.
- Old lesson: make attributes `private` and use synchronized mutator and accessors (if accessed by multiple threads).
- If an attribute is used directly by multiple threads and the access is not channelled through synchronised mutator and accessor it must be made volatile.
Volatile (ctd)

For the sake of efficiency, the compiler loads the value of a variable which is used several times into one of run-time registers.

Use of volatile signals the compiler to reload the attribute’s value directly from its memory location on each use.

This only applies to class variables and instances, variables and methods within a thread have their own copy on the run-time stack and are not going to be interfered with.
Atomic variable

- Java 5 introduces atomic variables to support atomic access to single variables in `java.util.concurrent.atomic`
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicLong
  - AtomicReference<V> // V type of object
  - AtomicReferenceArray<E> …

For example, AtomicInteger:
Atomic variable

class Counter {
    private int c = 0;
    public void increment()
        { c++;
    }
}

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
}
Thread safety

- A class takes proper care to protect its fields with locks, so that it can work with multiple threads, then it is called thread safe.

- Swing is not thread safe

- The general rule is to manipulate Swing components only on the event dispatch thread.

- Consider repaint-paint pattern: repaint can be called on any thread, but it doesn't directly do the repainting. Instead a paint is put on the queue of jobs for the event dispatch thread to execute.

Exceptions:

- `setText` can normally be called from any thread

- A component has not yet been set visible or added to a visible container, then any thread (typically the main thread when it creates objects) can safely work on it.
Liveness

A concurrent application's ability to execute in a timely manner is known as its liveness.

- deadlock
- starvation
- livelock
Deadlock

Locks are intended to ensure safety, blocking threads that might interfere with each other and forcing them to wait. Sometimes this can go too far:

*Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.

Example of deadlock: Suppose there are two shared resources, protected by locks lA and lB. A thread that wishes to use the resources must acquire both resources. Now suppose one method acquires lA first:
Deadlock (ctd)

lA.lock();
lB.lock();
try {
    critical region
} finally {
    lA.unlock();
lB.unlock();
}

while another method acquires them in the reverse order lB first, then lA. If two threads call the two methods, then they might reach deadlock with each thread owning one of the locks but unable to acquire the other.
Starvation

- *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
- Happens when shared resources are made unavailable for long periods by "greedy" threads.
Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work.
One form of deadlock is the following. A thread holds a lock $l$, but it cannot proceed because it is waiting for a certain state of affairs. Unfortunately, no other thread can bring about that state of affairs, because they would first need to acquire the lock $l$.

A *condition* is a way in which the first thread can temporarily release the lock so that other threads can acquire it.
Each condition is attached to a particular lock, and is created by calling the factory method `newCondition()` on that lock.

There are two `Condition` methods that work together:
- `await()`
- `signalAll()`.

When a thread calls `await()` it releases the lock and is blocked.
java.util.concurrent.locks.
Condition interface (ctd)

- `await()` also throws `InterruptedException` in the same way as `Thread.sleep()`, so again you have to decide how to handle those exceptions.

- A thread should only call `await()` or `signalAll()` on a condition if it already owns the corresponding lock. Otherwise an exception will normally be thrown.
The condition object keeps track of all the threads that are waiting on it.

Normally they stay blocked until another thread calls `signalAll()` on the condition.

At that point all the waiting threads are allowed to run again, though they still have to wait their turn to acquire the lock.

When a thread owns the lock again, it is allowed at last to return from `await()` and continue its execution.
Using a condition

- `await()` is called by a thread when it owns the lock but also needs that testable property to hold and discovers that it does not. By calling `await()`, the thread says, "I can't go on, I might as well be blocked, and I'll release the lock."

- Almost always, the `await()` is in a loop like this.
  ```java
  while (property is false) {
    condition.await();
  }
  ```

- When the thread finishes the loop, it knows the property is true, and it also owns the lock so no other thread can access the resource and make the property go false unexpectedly.
Let us rework the TravelAgent example, create 3 classes:

1. Flight.java with attributes: TotalNumberOfSeats and BookedTillNow. Include a constructor to use the fields. Add two methods:
   - ReserveSeat() // for booking a pair of seats
   - CancelReservation() // for cancelling a pair of seats.
   - Reservation takes 1 secs and Cancelation takes 5 secs

2. Threads classes ReservingTravelAgent.java, with Attributes:
   - int ID; // Identity number of the flight
   - Flight flight;
na-Exercise for condition: spec.

In the `run()` method invoke `ReserveSeat()` of flight, to reserve a pair of seats

3. `CancellingTravelAgent.java` with attributes
   - `int ID; // Identity number of the flight`
   - `Flight flight;`

In the `run()` method invoke `CancelReservation()` of flight

((for a sample solution see reworkedTravelAgent01 directory))
In the run() method invoke ReserveSeat() of flight, to reserve a pair of seats

3. CancellingTravelAgent.java with attributes
   - int ID; // Identity number of the flight
   - Flight flight;

In the run() method invoke CancelReservation() of flight

((for a sample solution see reworkedTravelAgent01 directory))
Exercise for condition: spec.

- In Flight class BookedTillNow is a shared resource. Use an explicit lock to make sure correct access via multiple threads to the two methods.

  Do this by adding a new lock

  ((for a solution see reworkedTravelAgent02 directory))

- Use Condition interface to modify the code to allow cancellation of seats and reservation of a pair of seats which are cancelled.

  ((for a solution see reworkedTravelAgent03 directory))