Lecture 06:
Mutual exclusion in Java

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

Recap
- Interacting threads
- Example of a race condition
- Theory of concurrency
- Mutual exclusion
- Locks
- Semaphores
- Monitors

Contents
- 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

Flight booking system (recap)
Consider a flight with 10 seats and a number of travel agents each trying to book two seats on the flight.
1) Write a class `Flight` with to integer value attribute for total number of seats and number of booked seats (make them static to simplify)
2) Write a class `TravelAgent` which is implemented as a thread. `TravelAgent` checks if there are two seats available and books them.
3) In `Flight`, create 10 `TravelAgent` and let them book.
At the end make sure that all threads `join()`.
4) Print the number of booked seats. (see folder `flight_travelagent_01` for a solution)

Flight booking system (ctd) (recap)
- Although each travel agent checks for availability of seat, we end up with too many seats booked.
- Tidy up the code by refactoring the `run()` method of `TravelAgent` by extracting method. Then include some print statement (or use a debugger) to explore why. (see folder `flight_travelagent_2`)

Back to Java
- To avoid data race, whenever two threads access the same data, you must ensure mutual exclusion.
In case of read/write
- multiple reader can read at the same time
- reader and writer must never access the same data at the same time
- two writer must never run at the same time
Question: how this can be achieved in Java.
Thread mutual exclusion is built on objects.
Every object has its own semaphore.
Use `synchronized` keyword on the object.
When using `synchronized` on an object, the JVM makes sure that at most one thread at any given instance has locked that specific object. In other words, only one thread can run at the same time in the object.
But all other threads are active and can be running on other objects available.

`synchronized` can be applied to:
- class methods
- block of code

In each case:
1) mutex (mutual exclusion) lock of an object is acquired.
2) the code is executed.
3) the mutex lock is released.

What happens if the acquired lock is held by another thread? The thread that wants the lock is suspended until the lock is released.
If we use the `synchronized`, are all the three stages happening in an atomic manner? Java takes care of all low-level details such as creating, obtaining, releasing,…
To improve the performance make the synchronization part as small as possible.

Apply `synchronized` to the method with the keyword `static`.

Add `static synchronized` to the method `booking()` of `TravelAgent` class (solution: folder `flight_travelagent_3`)

**Question:** What happens if you remove the `static`?
Try it?
Why?
Won’t work.

Apply `synchronized` to the method
We noticed this does not provide the expected result in the case of our example.
Synchronization stops multiple threads working on the same object, but it does not excludes the same method on different object.
Adding `static` ensures that only one of these objects exists.
You can also use singleton pattern – lots of wrong implementations around. (out of the scope of our work)

Add `static` ensures that only one of these objects exists.

Apply `synchronized(mutex)`.
Explicitly include an object (mutex) whose lock should be acquired.
Hence we need to provide an object.
1) using lock of any object
   `static Object object = new Object();`

   **Exercise:** Add `synchronized(object)` to the method `booking()` of `TravelAgent` class, where `object` is static (solution: folder `flight_travelagent_4`)

   **Exercise:** Would it work if non-static object is used? Why? Test your answer by modifying to `synchronized(new Object[])` to the method `booking()` of `TravelAgent` class (solution: folder `flight_travelagent_4`)

Mutual exclusion in Java

`synchronized`

Mutual exclusion over a class

Mutual exclusion over a method

Mutual exclusion over an snippet of code
Mutual exclusion ... snippet of code (ctd)

2) use the object which its details being updated (in our running example BookedSeat and TotalNumberOfSeats are not objects)
na-Exercise: You can use the lock of the class TravelAgent by synchronized (TravelAgent.class). (solution: folder flight_travelagent_5)

Using locks - the general idea

- Suppose there is some resource to be shared by several threads, accessing it only one at a time.
- You create a lock to protect that shared resource.
  - acquire the lock
  - critical region (accessing the shared resource) ...
  - release the lock
- Then no thread can be executing its critical region unless it owns the lock, and that means only one thread at a time.

Using locks in Java

- use Lock interface from java.util.concurrent.locks
- To acquire (or grab it), you call l.lock().
- If no other thread already owns it, then you now own the lock and your code continues executing.
- If some other thread does already own the lock, then you have to wait. (The lock itself keeps a list of the threads waiting to own it. there may be several, of course.) Your thread is blocked. Its code stops executing until it gets ownership of the lock

Using locks in Java (ctd)

- If you own the lock but don't need it any more, you call l.unlock(). Then, if any other threads are blocked waiting for it, one of them is given the lock and may start executing again.
- Don't call unlock unless you already own the lock. An exception (e.g. IllegalMonitorStateException) is likely to be thrown
- Various classes implement the Lock interface. It's normally easiest to use ReentrantLock.
na-Exercise: Study the Lock interface and ReentrantLock description. Look for java.util.concurrent.locks

lock-try-finally-unlock

- You must remember to release locks in Java.
- even if the critical region throws an exception
- Otherwise no other thread can access the resource
- Use
  l.lock();
  try {
    critical region
  } finally {
    l.unlock();
  }

na-Exercise

Use the lock on the travelagent example (code available at Lecture06_src\flight_travelagent_02)

Sketch of a solution:
1) in class TravelAgent create a new field for lock Lock mylock;
2) create a constructor with a field Lock
3) Modify run() by locking/unlocking before/after booking();
4) In the main() method of Flight, create a lock to be shared and pass it to the TravelAgent constructor. Use ReentrantLock() implementation of Lock() interface
(some sample solution at folder flights_travelagent_6)
Question: In stage 4 of the solution, can I write?
Lock bookinglock = new Lock();
Why not?
www.cs.bham.ac.uk/~bxb
Question: Experiment to see what happens if one
unlocks a thread which is not locked?
-Comment out the line mylock.lock(); in TravelAgent
class and see
IllegalMonitorStateException
is thrown