Operating Systems and Networks
Lecture 09: Threads in Java (continue)

Interacting threads

**Aim:** study threads which interact with each other by working on the same data, ensuring they don’t modify the same piece of data simultaneously.

Mutual exclusion
Let us see it with the help of an example

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 two integer value attributes 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`)

Race condition (recap)
- The problem is known as a *race condition* or *data race*.
- It is caused because threads/processes can access shared resources (shared variables, files, …)
- The key to avoiding race condition is to ensure mutual exclusion.
- Mutual exclusion: no two thread can access the critical region simultaneously.
- Critical region (also section) part of the program where shared memory is accessed.

**Question:** what is the critical section in case of the Flight example?

Race condition (ctd) (recap)
Mutual exclusion solves the race condition problem. But to have concurrent programs with correct behaviour the following conditions must satisfy:
1. No two thread may be simultaneously inside their critical regions
2. No assumption may be made about speed or number of CPUs
3. No thread running outside its critical region may block other processes
4. No thread should have to wait forever to enter its critical region
Strict alternation

- Thread 1:
  ```java
  While (true){
    while (turn !=0) {
      /* loop here to discover when turn change to 0 */
    }
    enter CR;
    turn = 1;
    execute non CR
  }
  thread are taking turn in execution.
  Use an variable turn.
  To access the CR, threads continuously test value of turn. This is called busy waiting
  … waste of CPU time
  ```

- Thread 2:
  ```java
  While (true){
    while (turn !=1) {
      /* loop here to discover when turn change to 1 */
    }
    enter CR;
    turn = 0;
    execute non CR
  }
  ```

Condition 3 can be violated.

- Thread1 leaves its CR, sets turn =1
- Thread2 enters its CR
- Thread2 finishes its CR
- Now, both threads in nonCR and turn = 0
- Thread1 execute its loop, exiting CR & setting turn= 1
- Now, both threads in nonCR and turn =1
- Thread1 finishes its nonCR fast, goes back to busy loop
- At this point, Thread1 is not permitted to enter CR because turn=1

So thread2 which is not in its CR is blocking Thread1

Busy waiting

- All above methods require busy waiting.
- Other methods based on busy waiting
  - Dekker’s mutual exclusion algorithm (60s)
  - Peterson’s algorithm (1981)
  - Test and Set Lock (TSL)
  - Negative aspects of busy waiting
    - wasting CPU resources
    - Priority Inversion Problem
    - Avoid waste of resources “Sleep and wakeup”

Busy waiting (ctd)

- Avoid waste of resources Sleep and wakeup
- Sleep is a system call that allows the caller to block itself, until another thread wakes it up (wakeup call)
- We must also take care of the case that the thread is not sleep and it receives a wakeup call
- This leads to the idea of semaphores

semaphores

- E.W. Dijkstra suggested using of a variable value to count the number of wakeups used for further use (semaphore)- designed for processes
  - value 0 means no wakeup calls
- There are two operations
  1. down() invoked on a semaphore checks the value
  2. If value >0 decrement the value (uses on wakeup) and continue
  3. If value = 0 process goes to sleep without completing the down()

The stage involving -Checking the value and changing it and going to sleep – is atomic (implementation issue)
semaphores (continue)

2. `up()` invoked on a semaphore increments the value.

If one or more processes are sleeping on a semaphore unable to complete an earlier `down()` one of them (randomly) is chosen by the system to complete its `down()`.

The stage involving – incrementing a semaphore and waking up a process – is atomic (implementation issue).

Mutex

- When the semaphore ability to count is not needed.
- Mutex has two states: `locked` and `unlocked`.

Thread 1:
```java
While (true){
    execute non-CR
    down(mutex)
    enter CR;
    up(mutex)
    execute non-CR
}
```

Thread 2:
```java
While (true){
    execute non-CR
    down(mutex)
    enter CR;
    up(mutex)
    execute non-CR
}
```

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.

Mutual exclusion in Java

- Thread mutual exclusion is build on objects.
- Every object has its own semaphore.
- Use `synchronised` keyword on the object.
- When using `synchronised` 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 running at the same time in the object.
- But all other thread are active and can be running on other objects available.

synchronised

- `synchronised` can be applied to:
  - Class methods.
  - INSTANCE 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.

synchronised (continue)

What happens if the acquired lock is held by another thread?

The thread that want the lock is suspended until the lock is release.

If we use the `synchronised`, 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 synchronisation part as small as possible.
Mutual exclusion over a class

- Apply `synchronized` to the method with the keyword `static`.
- **na-Exercise**: add `static synchronized` to the method `booking()` of `TravelAgent` class. (solution: folder `flight_travelagent_3`)
- What happens if you remove the `static`? Why? Won’t work.

Mutual exclusion over a method

- 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.

Mutual exclusion over an snippet of code

- 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
  ```java
  static Object object = new Object();
  na-Exercise: Apply `synchronized(new Object())` to the method `booking()` of `TravelAgent` class. (solution: folder `flight_travelagent_4`)
  ```

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**: `ss` create object from the fields (may be even static objects) to represent booking object (for example for complex scenario like which seat to book) and use the lock of that object. (solution: folder `flight_travelagent_5`)

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` (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 already owns 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.

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();`
  ```java
  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.

(sample solution at folder `flight_travelagent_6`)

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`

na-Exercise

Question: In stage 4 of the solution, can I write?

Lock bookinglock = new Lock();

Why not?
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.