Reminder about preparing for the exam
1) Review the lectures
2) Read first four chapters + sections 8.1 to 8.5 of our book
A. Silberschatz, P. B. Galvin and G. Gagne
Operating systems concepts

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 2 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
- A lock that uses busy waiting (turn) is called a spin lock.

Question: Are all four conditions for correct concurrent behaviour met?
Answer: condition 3 can be violated. But it ensures the other three condition are met.

Condition 3 can be violated.
- Thread 1 leaves its CR, sets turn = 1
- Thread 2 enters its CR
- Thread 2 finishes its CR
- Now, both threads in nonCR and turn = 0
- Thread 1 execute its loop, exiting CR & setting turn= 1
- Now, both threads in nonCR and turn =1
- Thread 1 finishes its nonCR last, goes back to busy loop
- At this point, Thread 1 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
A semaphore is an integer valued variable $s$ with two operations. These two operations are performed by threads on semaphore operation wait() (also called down()) or P() invoked on $s$
- The value $s$ is checked
- If $s > 0$ the value $s$ is decremented (the process can access CR) and continue
- If $s = 0$ process is blocked and goes to sleep
Operation signal() (also called up() or V()) invoked on $s$ decrement it ($s++$). This happens when process leaves the CR. Both execution of wait and signal are atomic and no other process can interrupt this. In particular, both testing and decreasing/increasing happens as one block (Hardware).
semaphores (continue)

Consider N resources
processes p_1, …, and p_k want to access them (processes have no preferences)
Imagine a semaphore with initial value s = N.
p_i wants to access performs a wait().
If s!=0, it reduces the value and access the resource.
If s=0, the process is blocked and go to sleep.
When process finishes leaves CR and calls signal(),
so the value increases. Another process can access.
Example: a bunch of people trying to access 5 cash machines.

Implementation of semaphore

If s=0 and p_i does a wait()
1) p_i is put into a waiting queue of the semaphore
2) control is transformed to the CPU scheduler
3) CPU scheduler chooses another process from ready processes
when another process singal() and p_i is at the head of queue
1) process is restarted by a wakeup() execution.
This sends the process from waiting to ready.
2) the process is put into the ready queue of the CPU
3) CPU follows it scheduling algorithm and runs the process.

Implementation of semaphore

How ensure wait() and signal() are done atomically?
In single processor systems, the interrupt is disabled during execution of wait() and signal().
In multi processor systems, it is possible to disable interrupt for all processors. This may be resource intensive, so other mechanism similar to the spin-lock or other advanced strategies are used.

Check pages 263-267 of our book (Silberschatz et al) for further information about semaphores

Mutex

- When the semaphore ability to count is not needed, we have two values 0 and 1
- ...

Thread 1:
While (true){
    execute non-CR
    s.wait()
    enter CR;
    s.signal()
    execute non-CR
}

Thread 2:
While (true){
    execute non-CR
    s.wait()
    enter CR;
    s.signal()
    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 wants the lock is suspended until the lock is released.
- 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
  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: sss 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 
   (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
   (sample solution at folder flight_travelagent_6)
**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

---

**Operating Systems and Networks**

Lecture 10: Threads in Java (continue)

---

**Recap**

- How to deal with race condition in Java
- Using `synchronized`
  - Mutual exclusion over a class
  - Mutual exclusion over a method
  - Mutual exclusion over a snippet of code
- Locking using Lock 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 synchronise (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 Java5
- 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 (Polling 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 synchronised mutator and accessor (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.

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 IA and IB. A thread that wishes to use the resources must acquire both resources. Now suppose one method acquires IA first.
Deadlock (ctd)

```java
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.

Liveloop

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.

Waiting for a state of affairs

- 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.

Java.util.concurrent.locks.
Condition interface

- 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.

`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;`

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

```java
((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

```java
((for a sample solution see reworkedTravelAgent01 directory))
```

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

```java
((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.

```java
((for a solution see reworkedTravelAgent03 directory))
```