G52CON: Concepts of Concurrency

Lecture 20: Revision and coursework feedback

Natasha Alechina
School of Computer Science
nza@cs.nott.ac.uk

Outline of this lecture

- Exam format
- Revision
- Coursework feedback

Exam format

- Change from previous years:
  - No multiple choice question
  - Answer 3 questions out of 5
- Otherwise the same kind of exam questions as in previous exam papers: 02-03, 03-04, 04-05, 06-07, 07-08. 05-06 was a bit different.
- Previous exam papers are available from the Portal → Library tab.
- Other useful sources include the unassessed exercises, and exercises from textbooks

Revision

There are several ways to revise for the exam:

- first you should look at your own notes
- all the slides for each lecture are available on-line, as are the lists of suggested reading, the unassessed exercises and the model answers
- textbooks are useful, particularly if there is something in your notes you don't understand (and slides don't include enough background and code examples)
- unassessed exercises and exercises from textbooks can also be very useful in giving you more practice at solving particular types of problems.

Outline Syllabus

The module covered of four main themes:

- introduction to concurrency;
- design of simple concurrent algorithms in Java;
- correctness of concurrent algorithms; and
- design patterns for common concurrency problems.

Topics covered

- mutual exclusion and condition synchronisation
- atomic actions
- mutual exclusion algorithms: Test-and-Set, Peterson's algorithm
- semaphores
- monitors
- distributed processing: message passing, RPC and rendezvous
- correctness of concurrent programs
- concurrent programming in Java (shared memory & distributed processing)
- common concurrency problems: e.g., Producer-Consumer, Readers and Writers, Client-Server

Lectures and topics

Lectures often cover more than one topic, e.g., a lecture about a concurrency primitive like monitors might contain:

- something about how monitors are implemented
- how they compare to other primitives
- the safety and liveness properties of solutions built using them
- examples of how they are used to solve a one of the common concurrency problems, e.g., bounded buffer problems

Mutual Exclusion & Condition Synchronisation

- mutual exclusion
- condition synchronisation
- interference
- critical sections
- classes of critical sections

Atomic actions

- process switching
- atomic actions
- memory accesses
- machine instructions
- multiprocessors
- mutual exclusion protocols

Mutual Exclusion algorithms

- archetypical mutual exclusion problem
- general form of a solution
- Test-and-Set algorithm
- Peterson's algorithm

Semaphores

- ***P*** and ***V*** operations
- general and binary semaphores
- blocking
- mutual exclusion & condition synchronisation with semaphores
- properties of semaphore solutions & problems of semaphores
- how to use semaphores

Monitors

- components of a monitor
- mutual exclusion & condition synchronisation in monitors
- operations on condition variables
- signalling disciplines
- how to use monitors

Distributed processing

- processes and channels
- message passing: asynchronous & synchronous message passing
- remote invocation: RPC & (extended) rendezvous
- modules & synchronisation

Correctness of concurrent programs

- safety properties: e.g., mutual exclusion, absence of deadlock, absence of unnecessary delay
- liveness properties: e.g., eventual entry
- properties of algorithms: e.g., Test-and-Set, Peterson's algorithm
- proof-based approaches to verification: e.g., assertional reasoning
- model-based approaches to verification: e.g., model checking, CTL specifications

Concurrent programming in Java

- Java threads
- monitors and Java
- Java memory model
- synchronisation: mutual exclusion & condition synchronisation
- distributed processing in Java: `java.rmi`

Producer-Consumer problems

- the role of buffering
- different sizes of buffer: e.g., single buffer, bounded buffer
- implementations: e.g., semaphores & monitors
- applications

Readers and Writers problems

- Reader & Writer processes
- synchronisation requirements
- implementations: e.g., semaphores & monitors
- applications

Client-Server problems

- data flow in concurrency problems
- patterns of communication
- examples:
  - chat server
  - time server using RPC

A sample question

(a) Briefly describe the major stages in the lifecycle of a thread and explain how the transitions between stages occur. (5)

(b) A savings account is accessed by several processes. A process making a deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds. Develop a Java `SavingsAccount` class which allows safe concurrent update of `SavingsAccount` objects. The class should have three public methods:

- `void deposit(int amount)` which adds amount to the current balance;
- `void withdraw(int amount)` which subtracts amount from the current balance;
- `int balance()` which returns the current balance.

Assume the arguments to `deposit` and `withdraw` are positive. **Explain your answer**. (10)

(c) Extend your solution to part (b) to incorporate an additional method

- `void transfer(SavingsAccount account, int amount)` which transfers amount from the account `account` to the balance of this account.

Explain clearly the problem(s) that must be addressed in designing the method, how your solution solves these problems and why it is correct. (10)

How to answer

- the number of marks gives some indication of the level of detail required, or the likely difficulty in the case of a problem solving question
- read the question *carefully* and answer what is asked—if it asks for a solution using, e.g., semaphores, then a monitor solution will get zero marks
- *explain* your answer if you are asked to do so
- attempt *all* parts of the question

A sample answer

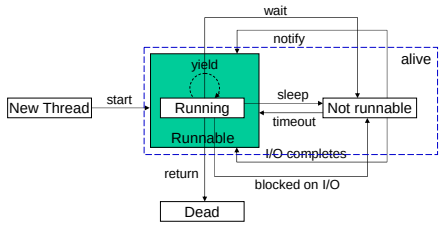
(a) There are three stages in the lifecycle of a thread:

• creation: we make a new Thread by calling the constructor of a subclass of the Thread class or by passing an instance of a class which implements Runnable to the Thread constructor

• alive: calling the start() method on the Thread object invokes its run() method as an independent activity. After a Thread has been started, it is said to be alive. A thread which is alive is either runnable or not runnable. Calling wait() or sleep() or blocking on I/O will make a running thread not runnable. Calls to notify(), timeouts and the completion of I/O will make a not runnable thread runnable.

• termination: the Thread terminates when its run method completes, either by returning normally or by throwing an unchecked exception (RuntimeException, Error or one of their subclasses).

`Thread` lifecycle



A sample answer

```
class SavingsAccount {
    private long balance = 0;

    public void synchronized deposit(int amount) {
        balance = balance + amount;
        notifyAll();
    }

    public void synchronized withdraw(int amount)
      throws InterruptedException {
        while (amount > balance)
            wait();
        balance = balance - amount;
    }

    public long synchronized balance() { return balance; }
}
```


A sample answer

(b) cont.

The current balance is held in the private long variable, balance.

The deposit and withdraw operations are implemented as synchronized methods. deposit simply increases the current balance by amount and signals all threads waiting to make a withdrawal that additional funds are now available. deposit never blocks (except for mutual exclusion), since making a deposit is always legal, whatever the balance.

Withdraw checks to see if there are sufficient funds to cover the withdrawal. If there are, it simply decreases the balance by amount. If there are insufficient funds, it waits until a deposit operation signals that there are more funds available and then tries again. Note that the call to wait() is enclosed in a while loop to ensure that the condition on which we are waiting—there being sufficient funds to cover the withdrawal—is actually true when we update the balance. Using notifyAll() wakes all processes waiting to make a withdrawal, allowing withdrawals to proceed up to but not exceeding the new balance. This could either be one large withdrawal, or several smaller withdrawals.

A sample answer

```
public void transfer(SavingsAccount other, int amount) {
    if (System.identityHashCode(this)<
        System.identityHashCode(other))
           this.doTransferFrom(other, amount);
       else
           other.doTransferTo(this, amount);
}
protected synchronized void doTransferFrom(SavingsAccount other, int amt){
    other.withdraw(amt);
    deposit(amt);
}

protected synchronized void doTransferTo(SavingsAccount other, int amt) {
    withdraw(amt);
    other.deposit(amt);
}
```


A sample answer

(c) cont.

The key problem is avoiding deadlock due to circular waiting, i.e., if two threads try to transfer money between the same two accounts, we don't end up with one thread holding the lock on each account and unable to obtain the lock on the other. (The problem of ensuring that there are sufficient funds in the account from which the money is being transferred is already solved by the withdraw method)

The problem can be solved by ensuring that threads always acquire the locks in the same order. One way to do this is to associate a unique tag with each object and require all threads to lock objects in tag order. For example, we can use the SavingsAccount's hash codes (e.g., System.identityHashCode() to order them (while it is possible for two objects to have the same hash code, it is unlikely, and this avoids allocating at tag to each new SavingsAccount object created).

Formal Coursework

- More detailed feedback will be on the web when marking is finished (including who wins the prize, and typical mistakes)

Unsynchronized solution 1

```
class SimpleBooking implements Comparable<SimpleBooking> {
    String reference;
    int[] days;
    int room;
    SimpleBooking(String reference, int[] days, int room) {
        Arrays.sort(days);
        this.reference = reference;
        this.days = days;
        this.room = room;}
    public int compareTo(SimpleBooking otherBooking) {
      if (this.days[0] < otherBooking.days[0]) return -1;
      if (otherBooking.days[0] < this.days[0]) return 1;
      return this.reference.compareTo(otherBooking.reference);
    }
```


Unsynchronized solution 2

```
public class Hotel {
  protected HashMap<Integer, TreeSet<SimpleBooking>> rooms =
    new HashMap<Integer, TreeSet<SimpleBooking>>();

  protected HashMap<String, SimpleBooking> bookings =
        new HashMap<String, SimpleBooking>();

  public Hotel(int[] roomNums) {
    for (int i = 0; i < roomNums.length; i++) {
        rooms.put(roomNums[i], new TreeSet<SimpleBooking>());
    }
  }
```


Unsynchronized solution 3

```
boolean roomBooked(int[] days, int roomNum) {
 Arrays.sort(days);
 Iterator<SimpleBooking> it = (rooms.get(roomNum)).iterator();
 while(it.hasNext()) {
   SimpleBooking booking = it.next();
   int[] when = booking.days;
   if (when[when.length - 1] < days[0]) continue;
   if (when[0] > days[days.length - 1]) {
      return false;
    } else {
      for (int j = 0; j < days.length; j++) {
        for (int k = 0; k < when.length; k++) {
          if (days[j] == when[k]) {
            return true;
 }}}}}} return false;}
```

Unsynchronized solution 4

```
boolean bookRoom(String bookingRef, int[] days, int roomNum) {
  if (roomBooked(days, roomNum)) {
    return false;
  } else {
    SimpleBooking booking = new SimpleBooking(bookingRef,
days, roomNum);
    bookings.put(bookingRef, booking);
    (rooms.get(roomNum)).add(booking);
  }
       return true;
}
```


Unsynchronized solution 5

```
void cancelBooking(String bookingRef) throws
                                NoSuchBookingException {
  SimpleBooking booking = bookings.get(bookingRef);
  if (booking == null) throw new
                          NoSuchBookingException(bookingRef);
     bookings.remove(bookingRef);
    (rooms.get(booking.room)).remove(booking);
  }
}
```


Unsynchronized solution 6

```
 boolean updateBooking(String bookingRef, int[] days, int
roomNum) throws NoSuchBookingException {
   SimpleBooking booking = bookings.get(bookingRef);
   if (booking == null) throw new
                        NoSuchBookingException(bookingRef);
   Arrays.sort(days);
   Iterator<SimpleBooking> it =
                        (rooms.get(roomNum)).iterator();
   while(it.hasNext()) {
     SimpleBooking otherBooking = it.next();
     // if otherBooking intersects on days, return false
     // … (as in roomBooked())
        }
// after checked all otherBookings…
```


Unsynchronized solution 7

```
  if (booking.room != roomNum) {
    (rooms.get(booking.room)).remove(booking);
    (rooms.get(roomNum)).add(booking);
  }

  booking.days = days;
  booking.room = roomNum;
  return true;
}
```


Fully synchronized solution

- Same as before, but put "synchronized" on every method.
- This makes sure that the class is thread-safe (only one thread is making booking or checking availability at any time).
- This allows very low concurrency. Clearly it should be possible to for example for several threads to check availability simultaneously.

Reader and writers solution

```
 import java.util.concurrent.locks.*;

// in the Hotel class:

  private ReadWriteLock rwl = new
                                ReentrantReadWriteLock();
  private Lock rl = rwl.readLock();
  private Lock wl = rwl.writeLock();
```

Reader and writers solution 2

```
 public boolean roomsBooked(int[] days, int[] roomNums)
{
   Arrays.sort(days);
   rl.lock();
   try {
     // same check as before…
   } finally {
     rl.unlock();
   }
}
```


Reader and writers solution 3

```
 public boolean bookRoom(String bookingRef, int[]
days, int roomNum) {
   wl.lock();
   try {
 // Check if the room is already booked
 // on the requested days.
 // If not, add the new booking to the current
 // bookings.
   } finally {
     wl.unlock();
   }
}
```


Reader and writers solution 4

- and similarly for other methods (**`cancelBooking`** and **`updateBooking`** use the write lock).
- it is reasonably straightforward to extend to having bookings with multiple rooms (still very good if you did it!)

Fine-grained solution

- However, can still do better: at the moment we lock the whole hotel if we want to book a room; maybe we could lock just the room we are interested in and allow other users to check/book other rooms at the same time
- I will call it fine-grained (concurrency control) solution
- This is tricky with a single room per booking, and even harder for multiple rooms, because when you want to change the booking you need to collect locks on multiple rooms
- Will try to explain a version of extended fine-grained solution (will be on the web, too)

Fine-grained solution 2

- **`Booking`** class keeps rooms, days, and compareTo method as **`SimpleBooking`** before (order by the first day of the booking, equals by the booking reference)
- **`BookingList`** a class to hold bookings for one room. The list is ordered by the compareTo.
- **`BookingList`** has a read-write lock
- Two synchronized hash maps, one (**`bookings`**) connecting booking references to **`Booking`** objects, and another (**`rooms`**) connecting room numbers to **`BookingLists`**

Fine-grained solution 3

```
public boolean roomsBooked(int[] days, int[] roomNums) {
      Arrays.sort(roomNums);
// Acquire read locks on the booking lists for all the rooms
      for (int i = 0; i < roomNums.length; i++)
          roomBookings(roomNums[i]).rl.lock();
      try {
          // Check if any of the rooms are booked
           …
      } finally {
          for (int i = roomNums.length - 1; i >= 0; i--)
              roomBookings(roomNums[i]).rl.unlock();
      }
  }
```

Fine-grained solution 4

```
 public boolean bookRooms(String bookingRef, int[] days, int[]
roomNums) {
  Arrays.sort(roomNums);
 for (int i = 0; i < roomNums.length; i++)
   roomBookings(roomNums[i]).wl.lock();
   try {
// Check if any of the rooms are already booked on the
requested, if not add the new booking to the current bookings.
     Booking booking = new Booking(bookingRef, days,
roomNums);
     for (int i = 0; i < roomNums.length; i++)
       roomBookings(roomNums[i]).add(booking);
       bookings.put(bookingRef, booking);
     }
     return true;
   } finally { // unlock the bookings lists }
```


Fine-grained solution 5

- To update or cancel, in principle need to (1) get the Booking object (2) get the room numbers from it and write-lock the BookingLists
- Pitfalls: several threads may get a reference to the same Booking and one of the may cancel it (remove its reference from bookings) while the other modify it and leave it in the bookingList for some rooms.
- As a result, there will be no booking reference corresponding to some days when a room or rooms are booked.

Fine-grained solution 6

- Fix: **`cancelBooking(bookingRef)`** removes **`bookingRef`** from **`bookings`** first, and then synchronizes on the **`Booking`** object **`b`** corresponding to the **`bookingRef`**: this means that it waits until any other thread which could be cancelling or updating **`b`** finishes.
- **`updateBooking`** also synchronizes on **`b`**, and as a first thing checks if the **`bookingRef`** is still in the **`bookings`**. Then it obtains the locks on **`bookingLists`** for the rooms in the old booking and for the new rooms, in order (to avoid deadlocks).

Any further questions

- More detailed coursework feedback on the web later
- If you have revision questions mail me and if there are a lot I will organise an extra revision lecture