v1.0
20150420

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2014/2015
Lecturer: Alessandro Ricci

[module lab 2.2]
THREAD SAFETY & LIVENESS

THREAD SAFETY DEFINITION

- A central aspect of concurrent programming is writing **thread-safe** code / thread-safe classes
  - a class is *thread-safe* if *it continues to behave correctly when accessed from multiple threads*
    - **regardless of the scheduling or interleaving of the execution of those threads by the runtime environment**
    - **with no additional synchronization or other coordination on the part of the calling code**
- *Correctness* means that a class *conforms to its specification*
  - a good specification defines
    - **invariants** constraining an object's state
    - **post-conditions** describing the effects of its operations
- Thread-safe classes encapsulate any needed synchronization so that clients need not to provide their own

SHARED MUTABLE STATE

- Writing thread-safe code is – at its core – about *managing access to state and in particular to* **shared, mutable state**:
  - **shared**: variable or object could be accessed by multiple threads
  - **mutable**: its value could change during its lifetime
- if multiple threads access the same mutable state variable without appropriate synchronization, the program is *broken*
  - race conditions
- There are three ways to fix it:
  - don't share state variable across threads
  - make the state variable immutable
  - use synchronization whenever accessing the state variable

STATELESS OBJECT

- Stateless objects are *always* thread-safe
  - actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads
- Example - a *factorizer service*
  - source: `pap.lab05.factorizer.FactorizerService`

```
public interface FactorizerService {
  int[] getFactors(long n);
}
```

- **Stateless** implementation
  - check source:

    `pap.lab05.factorizer.StatelessFactorizer`
  - this class is thread-safe *by construction*

STATE-FULL OBJECTS

- Thread safety is undermined as soon as we share and access in R/W stateful objects
  - **mutable** state-full objects, in particular
- Example: adding a *cache* to the factorizer service
  - check source:

    `pap.lab03.factorizer.FactorizerWithCache_unsafe`
  - this class is **not** thread-safe:
    - **check & act** problem => race conditions

RACE CONDITIONS

- The concurrent execution of non-atomic sequence of statements that should be considered atomic generate **race conditions**
  - occur when the *correctness of a computation depends on the relative timing or interleaving of multiple threads* by the runtime (...and getting the right answer relies on lucky timing..)
- Main examples
  - **lost updates**
    - when executing concurrently non-atomic *read-modify-write* operations
      - ex: count++
  - **check-and-act**
    - when a potentially state observation is used to make a decision on what to do next
    - example:

```
If (file X doesn't exist) -- check
then create file X        -- act
```

      - since check+act are not atomic, *the state can change after check and before act*.

COMPOUND ATOMIC ACTIONS

- *check-and-act* and *read-modify-write* are examples of compound actions
  - sequences of operations that must be executed atomically in order to remain thread-safe

ATOMIC COMPOUND ACTIONS IN JAVA : SYNCHRONIZED BLOCKS

- Compound-actions - and atomic statement blocks - in Java can be realized by means of synchronized blocks or methods

```
synchronized(lock){
   statement
   statement
   statement
}
```

- A synchronized block has 2 parts
  - a reference to an object that will serve as the lock
  - block of code to be guarded by the lock
- Mostly used at a *method* level
  - synchronized attribute
  - more about this in next modules when discussing monitors in Java

INTRINSIC LOCK AND ENTRY SET

- Atomic blocks work by exploiting the **lock** embedded in each Java object (more on this in next modules)
  - called **intrinsic lock** or monitor lock
  - functioning as a *guard* for the block
- The lock is automatically acquired and then released by a thread respectively when entering and exiting the block
  - if the lock is already acquired, the thread is blocked (suspended) and added to the entry set
  - when a thread exited the block, one thread of the entry set is selected and re-activated
  - no ordering policy is specified
  - if the lock is not released by the thread inside the block, threads in the entry set are blocked forever (starvation)
- For static methods and fields, the lock is associated to the related Class object
- For **synchronized** methods, the object serving as lock is **this**

LOCK REENTRANCY (1/2)

- def: **lock reentrancy**
  - when locks are acquired on a *per-thread* basis
    - vs. per-invocation basis
    - per-invocation basis is adopted instead as default locking behavior for Pthreads (POSIX threads) mutex-es
- Java intrinsic locks are reentrant:
  - if a thread tries to acquire a lock that it already holds, the request succeeds

LOCK REENTRANCY (2/2)

- Reentrancy facilitates encapsulation of locking behaviour and thus simplify the development of OO concurrent code

```
public class Widget {
  public synchronized doSomething(){...}
}

public class LoggingWidget extends Widget {
   public synchronized void doSomething(){
     System.out.println(toString()+": calling doSomething");
     super.doSomething();
   }
}
```

- Without reentrancy the above example would lead to a deadlock

PERFORMANCE: POOR CONCURRENCY PROBLEM

- The misuse of atomic blocks can lead to performance problems.
- Example: over constrained factorizer service
  - check source:
    `pap.lab05.factorizer.FactorizerWithCache_overconstrained`
- This solution is thread-safe but not acceptable
  - it enforces a sequentialization of computations that can be done concurrently
  - poor performances
- Careful choice of what parts must be designed and implemented as critical sections
  - examples with the factorizer service with cache
    - `pap.lab05.factorizer.FactorizerWithCache_quite_good`
    - `pap.lab05.factorizer.FactorizerWithCache_good`

AVOIDING LIVENESS HAZARD

- Tension between *safety* and *liveness*
  - using locking to ensure thread safety
    - ...but indiscriminate use of locking can cause "lock-ordering" deadlocks
  - using thread pools and semaphores to bound resource consumption
    - ...but failure to understand activities being bounded can cause resource deadlocks

DEADLOCKS

- A situation wherein *two or more competing actions are waiting for the other to finish, and thus neither ever does*
- Coffman *necessary conditions* for a deadlock to occur (1971)
  - **mutual exclusion condition**
    - a resource that cannot be used by more than one process at a time
  - **hold and wait condition**
    - processes already holding resources may request new resources
  - **no preemption condition**
    - no resource can be removed from a process holding it
    - resources can be released only by the explicit action of the process
  - **circular wait condition**
    - two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds
- Deadlock can only occur in systems where all 4 conditions hold true

DEADLOCKS WITH LOCKS

- It happens when
  - multiple threads wait forever due to cyclic locking dependency
  - simplest case
    - when thread A holds lock L and tries to acquire lock M, but at the same time thread B holds M and tries to acquire L, both thread will wait for ever
    - deadly embrace

DEADLOCKS DETECTION & RECOVERY

- Deadlocks **detection** and **recovery**
  - adopted in databases
    - databases are designed to detect and recover from deadlocks
    - transactions typically acquire many locks, until they commit
    - not so uncommon for two transactions to deadlock
  - identifying the set of transactions that are deadlocked by analyzing *is-waiting* dependency graph
    - looking for cycles
    - if a cycle is found, a victim is selected and the transaction aborted
- No automated deadlock detection / recovery mechanism in JVM
  - if threads deadlock, *that's all folks*
    - we can just shutdown the application
  - "post-mortem" diagnosis support

DEADLOCK DIAGNOSING

- *Thread dump* support provided by the JVM
  - triggered by
    - sending the JVM process a SIGQUIT signal on UNIX (kill -3)
    - pressing CTRL-\ on UNIX
    - pressing CTRL-Break on Windows
- Thread dump content
  - stack trace for each running thread
  - locking information
    - which locks are held by each thread, in which stack frame they were acquired, and which lock a blocked thread is waiting for

LOCK-ORDERING DEADLOCKS

- Simple example:

```
public class LeftRightDeadlock {
  private final Object left = new Object();
  private final Object right = new Object();

  public void leftRight(){
    synchronized(left){
      synchronized(right){
        doSomething();
      }
    }
  }

  public void rightLeft(){
     synchronized(right){
      synchronized(left){
        doSomethingElse();
      }
    }
  }

  private void doSomething(){ System.out.println("something.");}
  private void doSomethingElse(){ System.out.println("something else.");}
}
```

DYNAMIC LOCK-ORDER DEADLOCKS

- When locks to lock are established dynamically
  - basic example: Transfer money between bank accounts

```
public class Test1a {

  private static final int NUM_THREADS = 20;
  private static final int NUM_ACCOUNTS = 5;
  private static final int NUM_ITERATIONS = 10000000;
  private static final Random gen = new Random();
  private static final Account[] accounts = new Account[NUM_ACCOUNTS];

  public static void transferMoney(Account from,  Account to, int amount)
                                        throws InsufficientBalanceException {
    synchronized (from) {
      synchronized (to) {
        if (from.getBalance() < amount)
          throw new InsufficientBalanceException();
        from.debit(amount);
        to.credit(amount);
      }
    }
  }

  ...
}
```

```java
public class Test1a {

  private static final int NUM_THREADS = 20;
  private static final int NUM_ACCOUNTS = 5;
  private static final int NUM_ITERATIONS = 10000000;
  private static final Random gen = new Random();
  private static final Account[] accounts = new Account[NUM_ACCOUNTS];

  public static void transferMoney(Account from,  Account to, int amount)
          throws InsufficientBalanceException {...}

  static class TransferThread extends Thread {
    public void run() {
      for (int i = 0; i < NUM_ITERATIONS; i++){
        int fromAcc = gen.nextInt(NUM_ACCOUNTS);
        int toAcc = gen.nextInt(NUM_ACCOUNTS);
        int amount = gen.nextInt(10);
        try {
          transferMoney(accounts[fromAcc],accounts[toAcc],amount);
        } catch (InsufficientBalanceException ex){}
      }
    }
  }

  public static void main(String[] args) {
    for (int i = 0; i < accounts.length; i++){
      accounts[i] = new Account(1000);
    }
    for (int i = 0; i < NUM_THREADS; i++){
      new TransferThread().start();
    }
  }
}
```

ORDERING LOCKS

- Deadlock came because the two threads attempted to acquire the locks *in a different order*
  - if they asked for the locks in the same order, *there would be no cyclic locking dependency* and therefore no deadlock
- A program will be free of lock-ordering deadlocks *if all threads acquire the locks they need in a fixed global order*
  - verifying consistent lock ordering requires a global analysis of your program's locking behavior

```java
public class AccountManager {

  private final Account[] accounts;

  public AccountManager(int nAccounts, int amount){
    accounts = new Account[nAccounts];
    for (int i = 0; i < accounts.length; i++){
      accounts[i] = new Account(amount);
    }
  }

  public void transferMoney(int from, int to, int amount)
                                   throws InsufficientBalanceException {

    int first = from;
    int last = to;

    if (first > last){
      last = first;
      first = to;
    }

    synchronized (accounts[first]) {
      synchronized (accounts[last]) {
        if (accounts[from].getBalance() < amount)
          throw new InsufficientBalanceException();
        accounts[from].debit(amount);
        accounts[to].credit(amount);
      }
    }
  }
}
```

```java
public class Test1b {

  private static final int NUM_THREADS = 20;
  private static final int NUM_ACCOUNTS = 5;
  private static final int NUM_ITERATIONS = 10000000;
  private static final Random gen = new Random();

  static class TransferThread extends Thread {
    AccountManager man;
    TransferThread(AccountManager man){
      this.man = man;
    }
    public void run() {
      for (int i = 0; i < NUM_ITERATIONS; i++){
        int fromAcc = gen.nextInt(NUM_ACCOUNTS);
        int toAcc = gen.nextInt(NUM_ACCOUNTS);
        int amount = gen.nextInt(10);
        try {
          man.transferMoney(fromAcc,toAcc,amount);
        } catch (InsufficientBalanceException ex){
        }
      }
    }
  }
  public static void main(String[] args) {
    AccountManager man = new AccountManager(NUM_ACCOUNTS,1000);
    for (int i = 0; i < NUM_THREADS; i++){
      new TransferThread(man).start();
    }
  }
}
```

DEADLOCKS BETWEEN COOPERATING OBJECTS

- More subtle deadlocks can happen in cooperating objects, in which no methods explicitly acquire two locks, but where this happens indirectly
- A common example: *Observer pattern*
  – observers observing observed object
  – different control flows executing methods for observers and observed
- The more general problem
  – event-oriented pattern implementation in OO languages
    - coupling controls among sources and observers of events
  – MVC case study

```java
interface IObserved {
  int getState();
  void register(IObserver obj);
}

class MyEntityA implements IObserved {

  private List<IObserver> obsList;
  private int state;

  public MyEntityA(){
    obsList = new ArrayList<IObserver>();
  }

  public void register(IObserver obs) {
    obsList.add(obs);
  }

  public synchronized int getState() {
    return state;
  }

  public synchronized void changeState1() {
    state++;
    for (IObserver o: obsList){
      o.notifyStateChanged(this);
    }
  }

  public synchronized void changeState2() {
    state--;
    for (IObserver o: obsList){
      o.notifyStateChanged(this);
    }
  }
}
```

```java
interface IObserver {
  void notifyStateChanged(IObserved obs);
}

class MyEntityB implements IObserver {

  List<IObserved> obsList;

  public MyEntityB(){
    obsList = new ArrayList<IObserved>();
  }

  public synchronized void observe(IObserved obj){
    obsList.add(obj);
    obj.register(this);
  }

  public synchronized
             void notifyStateChanged(IObserved obs) {
    synchronized(System.out){
      System.out.println(
            "state changed: "+obs.getState());
    }
  }

  public synchronized int getOverallState() {
    int sum = 0;
    for (IObserved o: obsList){
      sum += o.getState();
    }
    return sum;
  }
}
```

```java
class MyThreadA extends Thread {
  MyEntityA obj;

  public MyThreadA(MyEntityA obj){
    this.obj = obj;
  }

  public void run(){
    while (true){
      obj.changeState1();
      obj.changeState2();
    }
  }
}

class MyThreadB extends Thread {
  MyEntityB obj;

  public MyThreadB(MyEntityB obj){
    this.obj = obj;
  }

  public void run(){
    while (true){
      log("overall state: "+
                obj.getOverallState());
    }
  }

  private void log(String msg){
    synchronized(System.out){
      System.out.println("["+this+"] "+msg);
    }
  }
}
```

```java
public class Test2 {
  public static void main(String[] args) {

    MyEntityA objA = new MyEntityA();
    MyEntityB objB = new MyEntityB();
    objB.observe(objA);

    new MyThreadA(objA).start();
    new MyThreadB(objB).start();

  }
}
```

AVOIDING DEADLOCKS

- A program that never acquires more than one lock a time cannot have lock-ordering deadlock
- if we must acquire multiple locks, lock ordering must be part of the design
  - minimizing the number of potential locking interaction
  - document a lock-ordering protocol for locks that may be acquired together
- Alternative technique: *timed locks*
  - detecting and recovering from deadlocks using `tryLock` feature of `Lock` classes instead of intrinsic lock

OTHER LIVENESS HAZARD

- **Starvation**
  - typically manifested when using *priorities*
  - basic thread support for priorities in Java thread is “deprecated”
    - platform dependent
    - liveness problems
- **Poor responsiveness**
  - e.g. executing long-term tasks in GUI thread
  - can also be caused by *poor lock management*
    - if a thread holds a lock for long time - for instance while iterating on a large collection and performing substantial work on each element - other threads that need to access that collection may have to wait long time
- **Livelock**
  - when threads cannot make progress because they keep retrying an operation that will always fail
  - solution: introducing some *randomness* into the retry mechanism
    - breaking the synchronization that causes the live-lock

BIBLIOGRAPHY

- *“Java Concurrency in Practice”*, Brian Goetz, Addison Wesley