i219 Software Design Methodology
10. Multithreaded programming

Kazuhiro Ogata (JAIST)

Outline of lecture

- Thread
- Race condition
- Synchronization
- Deadlock
- Bounded buffer problem

Thread (1)

- Units of execution.
- Can run in parallel (in theory on a uniprocessor computer but actually on a multiprocessor or multicore computer).
- Instances of class Thread (or any of its subclasses) in Java, making it possible to write parallel (or concurrent) programs in Java.

✓ A process in an OS is a running program, equipped with its own memory space; may be called a heavyweight process.
✓ Multiple threads can reside in one process, sharing one memory space; may be called lightweight processes; UNIX on workstations sold by Sun Microsystems (where Java was born) had a lightweight process library.

Thread (2)

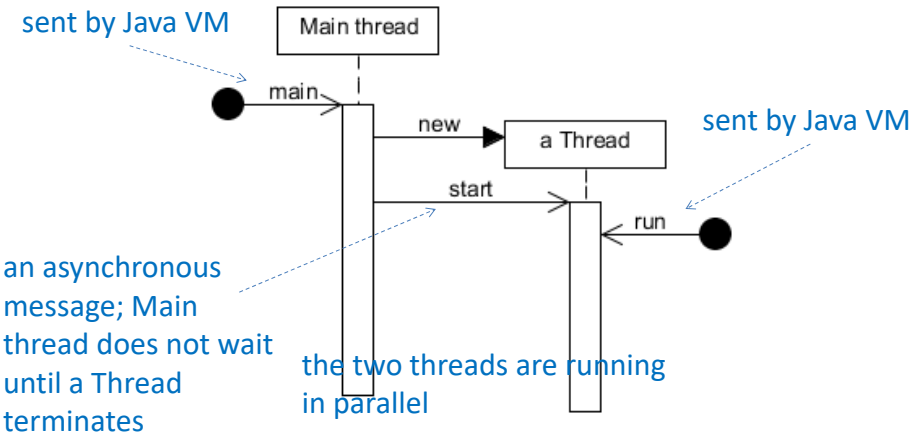
```
public class SubclassOfThread extends Thread {
  …
  public SubclassOfThread(…) { … }
  …
  public void run() { … }
  … }

Thread th = new SubclassOfThread(…);
th.start();
```

a new thread is created

making the thread active (scheduled) and start executing method run() by sending message start() to the thread

when an application is launched, there must be at least one thread that executes main(); such a thread is called the main thread

Thread (3)



Thread (4)

```
public class PrintingThread extends Thread {
   private int times;
   public PrintingThread(int n) { this.times = n; }
   public void run() {  long myId = this.getId();
      for (int i = 0; i < times; i++)
         System.out.println(i + ": I am #" + myId + " thread.");
   }
   public static void main(String[] args) {
      Thread t1 = new PrintingThread(50);
      Thread t2 = new PrintingThread(50);
      Thread t3 = new PrintingThread(50);
      t1.start();  t2.start();  t3.start();
   }
}
```

obtaining the identifier of the thread executing the method

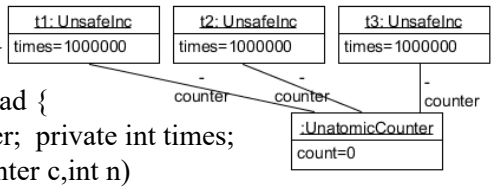
creating three threads, each of which displays what is like

17: I am #10 thread.

50 times.

Race condition (1)

```
public class NonatomicCounter { private int count = 0;
   public void inc() { count++; }
   public int get() { return count; } }

public class UnsafeInc extends Thread {
   private NonatomicCounter counter;  private int times;
   public UnsafeInc(NonatomicCounter c,int n)
   { this.counter = c; this.times = n; }
   public void run() { for (int i = 0; i < times; i++) counter.inc(); }
   public static void main(String[] args) throws InterruptedException {
      NonatomicCounter c = new NonatomicCounter();
      Thread t1 = new UnsafeInc(c,1000000);
      Thread t2 = new UnsafeInc(c,1000000);
      Thread t3 = new UnsafeInc(c,1000000);
      t1.start(); t2.start(); t3.start();
      t1.join(); t2.join(); t3.join();
      System.out.println("Counter: " + c.get()); } }
```



The object is shared by the three threads

Race condition (2)

A launch of the application (UnsafeInc) does not display the followng:

Counter: 3000000

What are actually displayed are as follows:

Counter: 1697864 Counter: 1700446 Counter: 2737760 …

Each time the application is launched, a different result is displayed.

Why?

Race condition (3)

The reason: count++; is not atomic and at least consists of three basic things: (1) read count (fetching the content v of count), (2) compute (calculate $v+1$), and (3) write count (store the result of $v+1$ into count).

count++; is processed by three threads simultaneously without any protection or in an arbitrary way.

When each thread ti ($i = 1,2,3$) performs read$_i$ count, compute$_i$ and write$_i$ count, there are ${}_9C_3 \times {}_6C_3$ (1680) possible scenarios.

One possible scenario:

read$_1$ count, read$_2$ count, read$_3$ count, compute$_1$, compute$_2$, compute$_3$, write$_1$ count, write$_2$ count, write$_3$ count

After the scenario, what is stored in count is 1 but not 3, although each thread increments count. The effects of two increments are lost.

Race condition (4)

Race condition is a situation in which objects (or resources) shared by multiple threads are used *without any protection* (or *in an arbitrary way*) by those threads, which may cause a different outcome each time when the program is launched.

One possible remedy is to use *synchronization* mechanisms, controlling threads so that at most one thread is allowed to use shared objects (or resources) at any given moment.

Synchronization (1)

Each object is equipped with one lock that can be used to synchronize threads such that a thread that has acquired such a lock is allowed to enter a section in which shared objects (or resources) can be used.

Two ways to use such locks:

1. Synchronized methods: … synchronized … *m*(…) { … }

When a thread *t* executes *o*.*m*(…), *t* first tries to acquire the lock *l* associated with an object *o*. If *t* has acquired *l*, *t* is allowed to invoke m(…). Otherwise, *t* waits until *t* has acquired *l*. When *t* finishes executing m(…), *t* releases *l*.

2. Synchronized statements: synchronized (*o*) { … }

When a thread *t* executes the statement, *t* first tries to acquire the lock *l* associated with an object *o*. If *t* has acquired *l*, *t* is allowed to enter the body … Otherwise, *t* waits until *t* has acquired *l*. When *t* leaves the body … , *t* releases *l*.

Synchronization (2)

```
public class AtomicCounter {private int count = 0;
   public synchronized void inc() { count++; }
   public synchronized int get() { return count; } }

public class SafeInc1 extends Thread {
   private AtomicCounter counter;  private int times;
   public SafeInc1(AtomicCounter c,int n)
   { this.counter = c; this.times = n; }
   public void run() { for (int i = 0; i < times; i++) counter.inc(); }
   public static void main(String[] args) throws InterruptedException {
      AtomicCounter c = new AtomicCounter();
      Thread t1 = new SafeInc1(c,1000000);
      Thread t2 = new SafeInc1(c,1000000);
      Thread t3 = new SafeInc1(c,1000000);
      t1.start(); t2.start(); t3.start();
      t1.join(); t2.join();t3.join();
      System.out.println("Counter: " + c.get()); } }
```

Synchronization (3)

```
public class SafeInc2 extends Thread {
   private NonatomicCounter counter;  private int times;
   public SafeInc2(NonatomicCounter c,int n)
   { this.counter = c; this.times = n; }
   public void run() {
      for (int i = 0; i < times; i++)
         synchronized (counter) { counter.inc(); } }
   public static void main(String[] args) throws InterruptedException {
      NonatomicCounter c = new NonatomicCounter();
      Thread t1 = new SafeInc2(c,1000000);
      Thread t2 = new SafeInc2(c,1000000);
      Thread t3 = new SafeInc2(c,1000000);
      t1.start(); t2.start(); t3.start();
      t1.join(); t2.join(); t3.join();
      System.out.println("Counter: " + c.get()); } }
```


Synchronization (4)

```
public class PseudoAtomicCounter { private static int count = 0;
   public synchronized void inc() { count++; }
   public synchronized int get() { return count; } }
public class PseudoSafeInc1 extends Thread {
   private PseudoAtomicCounter counter;  private int times;
   public PseudoSafeInc1(PseudoAtomicCounter c,int n)
   { this.counter = c; this.times = n; }
   public void run() { for (int i = 0; i < times; i++) counter.inc(); }
   public static void main(String[] args) throws InterruptedException {
      PseudoAtomicCounter c1 = new PseudoAtomicCounter();
      PseudoAtomicCounter c2 = new PseudoAtomicCounter();
      PseudoAtomicCounter c3 = new PseudoAtomicCounter();
      Thread t1 = new PseudoSafeInc1(c1,1000000);
      Thread t2 = new PseudoSafeInc1(c2,1000000);
      Thread t3 = new PseudoSafeInc1(c3,1000000);
      t1.start(); t2.start(); t3.start();
      t1.join(); t2.join(); t3.join();
      System.out.println("Counter: " + c1.get()); }}
```

What's wrong?

Synchronization (5)

```
public class PseudoSafeInc2 extends Thread {
   private NonatomicCounter counter; private int times;
   public PseudoSafeInc2(NonatomicCounter c,int n)
   { this.counter = c; this.times = n; }
   public void run() {
      for (int i = 0; i < times; i++)
         synchronized (this) { counter.inc(); } }
   public static void main(String[] args) throws InterruptedException {
      NonatomicCounter c = new NonatomicCounter();
      Thread t1 = new PseudoSafeInc2(c,1000000);
      Thread t2 = new PseudoSafeInc2(c,1000000);
      Thread t3 = new PseudoSafeInc2(c,1000000);
      t1.start(); t2.start(); t3.start();
      t1.join(); t2.join(); t3.join();
      System.out.println("Counter: " + c.get()); } }
```

What's wrong?

Deadlock (1)

```
public class DeadlockInc extends Thread {
   private NonatomicCounter counter1, counter2;
   public DeadlockInc(NonatomicCounter c1,NonatomicCounter c2)
   { this.counter1 = c1; this.counter2 = c2; }
   public void run() {
      synchronized (counter1) { for (int i=1; i < 5000; i++) ;
         synchronized (counter2) {
            counter1.inc(); counter2.inc(); } } }
   public static void main(String[] args) throws InterruptedException {
      NonatomicCounter c1 = new NonatomicCounter();
      NonatomicCounter c2 = new NonatomicCounter();
      Thread t1 = new DeadlockInc(c1,c2);
      Thread t2 = new DeadlockInc(c2,c1);
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println("Counter1: " + c1.get() + ", Counter2: " + c2.get()); } }
```

Deadlock (2)

Thread t1 tries to acquire the lock associated with c1 and the lock associated with c2, while thread t2 tries to acquire the lock associated c2 and the lock associated with c1.

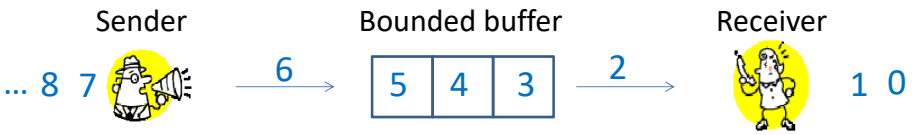
If t1 has acquired the lock l1 associated with c1 and t2 has acquired the lock l2 associated with c2, t1 will wait forever until l2 is released and t2 will wait forever until l1 is released – deadlock.

Deadlock is a situation in which nothing will never happen.

Deadlock (3)

```
public class NoDeadlockInc extends Thread {
   private NonatomicCounter counter1, counter2;
   public NoDeadlockInc(NonatomicCounter c1,NonatomicCounter c2)
   { this.counter1 = c1; this.counter2 = c2; }
   public void run() {
      synchronized (counter1) { for (int i=1; i < 5000; i++) ;
         synchronized (counter2) {
            counter1.inc(); counter2.inc(); } } }
   public static void main(String[] args) throws InterruptedException {
      NonatomicCounter c1 = new NonatomicCounter();
      NonatomicCounter c2 = new NonatomicCounter();
      Thread t1 = new NoDeadlockInc(c1,c2);
      Thread t2 = new NoDeadlockInc(c1,c2);
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println("Counter1: " + c1.get() + ", Counter2: " + c2.get()); } }
```

Bounded buffer problem (1)



Let us write a program such that a thread (called Sender) sends data (represented as 0, 1, 2, …) to another thread (called Receiver) via a bounded buffer.

The bounded buffer is shared with the two threads (Sender & Receiver). First, the bounded buffer uses neither synchronized methods nor synchronized statements.

Bounded buffer problem (2)

```
public class NonatomicBBuf<E> {
  private Queue<E> queue;
  private int noe = 0;
  private final int capacity;
  public NonatomicBBuf(int cap) {
    this.queue = new EmpQueue<E>();
    this.capacity = cap;
  }
  public void put(E e) {
    if (noe < capacity) {
      queue = queue.enq(e);
      noe++;
    }
  }
```

Only when the bounded buffer is not full, an element e is put into it.

Bounded buffer

When the bounded buffer is empty, null is returned.

```
  public E get() {
    if (noe > 0) {
      E e = queue.top();
      queue = queue.deq();
      noe--;
      return e;
    }
    return null;
  }
}
```

Bounded buffer problem (3)

Sender

```
import java.util.*;

public class FSender1<E> extends Thread {
   private NonatomicBBuf<E> buf;  the bounded buffer
   private List<E> msgs;   data to be sent
   public FSender1(NonatomicBBuf<E> buf,List<E> msgs) {
      this.buf = buf; this.msgs = msgs; }
   public void run() {
      for (int i = 0; i < msgs.size(); i++)
         buf.put(msgs.get(i));
   }
}
```

What Sender does is to put the data to be sent into the bounded buffer.

Bounded buffer problem (4)

Receiver

```
import java.util.*;

public class FReceiver1<E> extends Thread {
   private NonatomicBBuf<E> buf;  the bounded buffer
   private List<E> msgs;   in which the received data are stored
   private int nom;    the number of data to be received
   public FReceiver1(NonatomicBBuf<E> buf,List<E> msgs,int nom) {
      this.buf = buf; this.msgs = msgs; this.nom = nom; }
   public void run() {
      for (int i = 0; i < nom; i++)
         msgs.add(buf.get());
   }
}
```

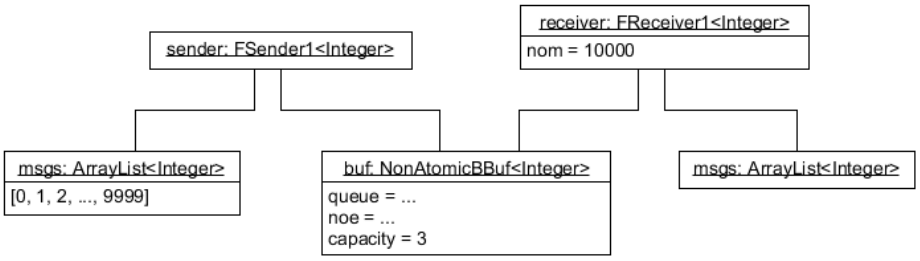
What Receiver does is to get data from the bounded buffer

Bounded buffer problem (5)

```
import java.util.*;

public class FBBProb1 {
  public static void main(String[] args) throws InterruptedException {
    NonatomicBBuf<Integer> buf = new NonatomicBBuf<Integer>(3);
    List<Integer> msgsSent = new ArrayList<Integer>();
    for (int i=0; i < 10000; i++) msgsSent.add(i);
    List<Integer> msgsReceived = new ArrayList<Integer>();
    int nom = msgsSent.size();
    FSender1<Integer> sender = new FSender1<Integer>(buf,msgsSent);
    FReceiver1<Integer> receiver = new FReceiver1<Integer>(buf,msgsReceived,nom);
    sender.start(); receiver.start();
    sender.join(); receiver.join();
    System.out.println("msgsSent: " + msgsSent);
    System.out.println("msgsReceived: " + msgsReceived);
    if (msgsReceived.equals(msgsSent)) System.out.println("Success!");
    else System.out.println("Failure!"); } }
```


Bounded buffer problem (6)

```
NonatomicBBuf<Integer> buf = new NonatomicBBuf<Integer>(3);
List<Integer> msgsSent = new ArrayList<Integer>();
List<Integer> msgsReceived = new ArrayList<Integer>();
int nom = msgsSent.size();
FSender1<Integer> sender = new FSender1<Integer>(buf,msgsSent);
FReceiver1<Integer> receiver = new FReceiver1<Integer>(buf,msgsReceived,nom);
```



Bounded buffer problem (7)

The first version does not successfully deliver the data 0, 1, 2, …, 9999 to Sender from Receiver.

The reason must be that the methods put(E e) and get() in the class NonatomicBBuffer<E> are not synchronized.

```
public class AtomicBBuf<E> { …
  public synchronized void put(E e) { … }
  public synchronized E get() { … }
  … }
```

The only difference between NonatomicBBuffer<E> & AtomicBBuf<E> are that the two methods are synchronized (and the name of the class & constructor).

Bounded buffer problem (8)

```
public class FSender2<E> extends Thread { … }
public class FReceiver2<E> extends Thread { … }
```

AtomicBBuf<E> is used instead of NonatomicBBuffer<E> in these classes.

```
public class FBBProb2 { … }
```

AtomicBBuf<E>, FSender2<E> & FReceiver2<E> are used instead of NonatomicBBuffer<E>, FSender1<E> & FReceiver1<E> in the class.

Bounded buffer problem (9)

The second version still does not successfully deliver the data 0, 1, 2, …, 9999 to Sender from Receiver.

```
public synchronized void put(E e) {
  if (noe < capacity) { queue = queue.enq(e); noe++; } }
```

When put(e) is invoked, if the bounded buffer is full, e is not stored in the bounded buffer.

```
public E get() {
  if (noe > 0) { E e = queue.top(); queue = queue.deq(); noe--; return e; }
  return null; } }
```

When get() is invoked, if the bounded buffer is empty, null is returned.

It seems to solve them if Sender waits until the buffer becomes non-full when it is full and Receiver waits until it becomes non-empty when it is empty.

Bounded buffer problem (10)

In FAtomicBBuf<E>, the two methods are modified as follows:

```
public synchronized void put(E e) {
  while (noe >= capacity) ;
  if (noe < capacity) {
    queue = queue.enq(e); noe++; } }
```

```
public synchronized E get() {
  while (noe <= 0) ;
  if (noe > 0) {
    E e = queue.top(); queue = queue.deq()
    noe--; return e; }
  return null; }
```

```
public class FSender3<E> extends Thread { … }
public class FReceiver3<E> extends Thread { … }
```

FAtomicBBuf<E> is used in these classes.

```
public class FBBProb3 { … }
```

FAtomicBBuf<E>, FSender3<E> & FReceiver3<E> are used in the class.

The third version is not a solution. Why?

Bounded buffer problem (11)

The class Object provides wait() and notifyAll().

When a thread *t* holds the lock *l* associated with an object *o* and executes *o*.wait(), *t* releases *l* and waits until *o*.notifyAll() is executed by some other thread.

If a thread executes *o*.notifyAll(), all threads that waits by executing *o*.wait() are waken up and try to acquire the lock associated with *o*.

In MonitorBBuf<E>, the two methods are modified as follows:

```
public synchronized void put(E e)
  throws InterruptedException {
  while (noe >= capacity) this.wait();
   if (noe < capacity) {
     queue = queue.enq(e); noe++;
     this.notifyAll(); } }
```

```
public synchronized E get()
  throws InterruptedException {
  while (noe <= 0) this.wait();
  if (noe > 0) { E e = queue.top();
    queue = queue.deq(); noe--;
    this.notifyAll();
    return e; }
  return null; }
```


Bounded buffer problem (12)

```
public class Sender<E> extends Thread { … }
public class Receiver<E> extends Thread { … }
```

MonitorBBuf<E> is used in these classes.

```
public class BBProb { … }
```

MonitorBBuf<E>, Sender<E> & Receiver<E> are used in the class.

This version successfully delivers the data 0, 1, 2, …, 9999 to Sender from Receiver.

Note that you should use try { … } catch (…) { … } in Sender<E> & Receiver<E>.

Summary

- Thread
- Race condition
- Synchronization
- Deadlock
- Bounded buffer problem

Appendix

```
public interface Queue<E> {
   Queue<E> enq(E e);
   Queue<E> deq();
   E top(); }

public class EmpQueue<E> implements Queue<E> {
   public Queue<E> enq(E e) { return new NeQueue<E>(e,this); }
   public Queue<E> deq() { return this; }
   public E top() { return null; } }

public class NeQueue<E> implements Queue<E> {
   private E head;  private Queue<E> tail;
   public NeQueue(E e,Queue<E> q) { this.head = e; this.tail = q; }
   public NeQueue<E> enq(E e)
   { return new NeQueue<E>(head,tail.enq(e)); }
   public Queue<E> deq() { return tail; }
   public E top() { return head; } }
```