Concurrency, Parallelism and Distribution (CPD)

Concurrency: Models Problems and Solutions

Joaquim Gabarro
(gabarro@cs.upc.edu)

Computer Science
BarcelonaTech, Universitat Politècnica de Catalunya

Study material

Jeff Magee and Jeff Cramer
Concurrency, State Models & Java Programs
John Wiley & and Sons, 2006.
`http://www.doc.ic.ac.uk/~jnm/book/`

1. Problem: Interference

Problem:Interference

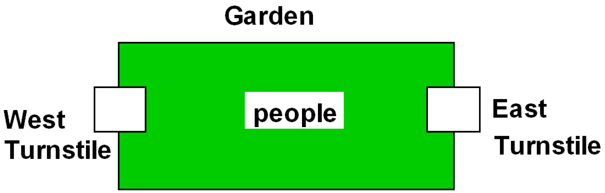
Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference.

How to model and detect.

We model and analyze the Ornamental Garden Problem.

Ornamental Garden Problem

People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.



The concurrent program consists of two concurrent threads and a shared counter object.

Interaction in Java

The simplest way for two or more threads in a Java Program to interact is via an object whose methods can be invoked by a set of threads.

The object people can be invoked by threads east and west.

```
public class Garden {
  public static void main(String[] args){
    Counter people = new Counter("Counter");
    Turnstile west= new Turnstile("West", people);
    Turnstile east= new Turnstile("East",people);
    west.start(); east.start();
}}
```

Class Turnstile

```
class Turnstile extends Thread {
  String name;
  Counter count;
  public final static int MAX = 20;

  Turnstile(String threadName, Counter c)
    { name = threadName; count = c; }

  public void run() {
    try{
     System.out.println(name+ " value is: "+ 0);
     for (int i=1;i<=MAX;i++){
         Thread.sleep(500); //0.5 second
         System.out.println(name+ " value is: "+ i);
         count.increment();
     }
    } catch (InterruptedException e) {}
}}
```

Class Counter

```
public class Counter {
  int value=0;
  String name;

  Counter(String name) {this.name=name;}

  void increment() {
    int temp = value;   //read[v]
    Simulate.HWinterrupt();
    value=temp+1;       //write[v+1]
    System.out.println(name+ " value is: "+ value);
  }
}
```

Contex switch

Context switch is the action of switching from executing one process to another.

```
class Simulate {
    public static void HWinterrupt() {
        if (Math.random()<0.5)
           try{Thread.sleep(200);}
            catch(InterruptedException e){};
            //used instead of Thread.yield()
            //to ensure portability
    }
}
```

Problem

A possible execution of `Garden` give us:

```
West value is: 0
East value is: 0
West value is: 1
East value is: 1
Counter value is: 1
Counter value is: 1
East value is: 2
Counter value is: 2
West value is: 2
Counter value is: 3
East value is: 3
...
West value is: 19
Counter value is: 34
West value is: 20
Counter value is: 35
```

In this case 5 entries are lost!

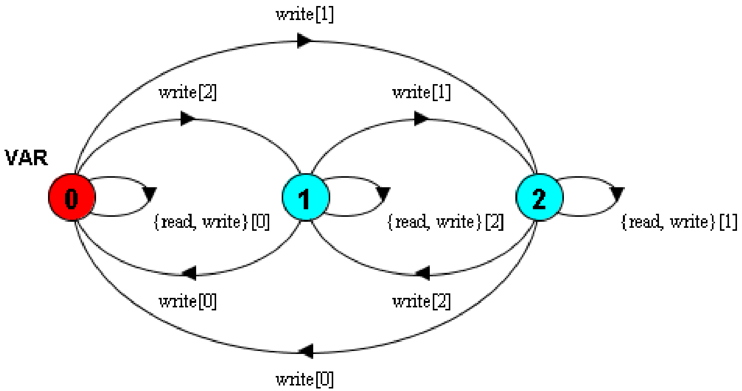
Model

We need to model a variables (counters), turnstiles and interaction between them.

```
VAR = · · ·
TURNSTILE = · · ·
||GARDEN =
     (east:TURNSTILE || west:TURNSTILE
     ||{east,west, people}::value:VAR)
     /{go/{ east, west}.go, end/{east,west}.end} .
```

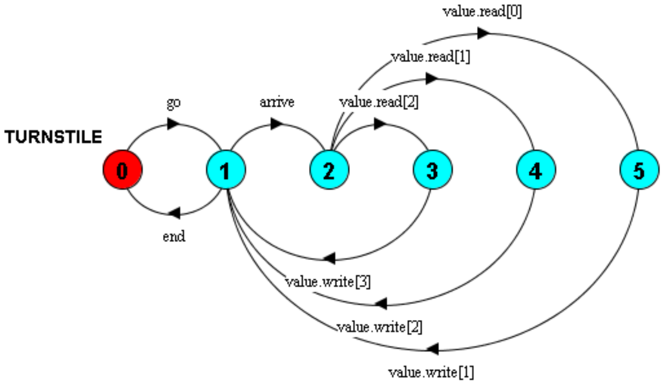
Variable

```
const N = 2
range T = 0..N
VAR = VAR[0],
VAR[u:T] = (read[u] ->VAR[u] | write[v:T]->VAR[v]).
```



Turnstile

```
const N = 2
range T = 0..N
set VarAlpha = {value.{read[T],write[T]}}

TURNSTILE = (go -> RUN),
RUN = (arrive-> INCREMENT | end -> TURNSTILE),
INCREMENT = (value.read[x:T] ->value.write[x+1]->RUN) +VarAlpha.
```



Complete Model

```
const N = 4
range T = 0..N
set VarAlpha = {value.{read[T],write[T]}}

VAR = VAR[0],
VAR[u:T] = (read[u] ->VAR[u] | write[v:T]->VAR[v]).

TURNSTILE = (go -> RUN),
RUN = (arrive-> INCREMENT | end -> TURNSTILE),
INCREMENT =
  (value.read[x:T] ->value.write[x+1]->RUN) +VarAlpha.

||GARDEN =
     (east:TURNSTILE || west:TURNSTILE
     ||{east,west, people}::value:VAR)
     /{go/{ east, west}.go, end/{east,west}.end} .
```

Inconsistent state

Trace to a inconsistent state:

```
go
east.arrive
east.value.read.0
west.arrive
west.value.read.0
east.value.write.1
west.value.write.1
end
people.value.read.1
```

Interference

Interference: Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference.

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects. Mutual exclusion can be modeled as atomic actions

2. Solution: Mutual Exclusion

Solution: Mutual exclusion

Interference: Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference.

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects. Mutual exclusion can be modeled as atomic actions

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword synchronized, which uses a lock on the object.

To ensure mutually exclusive access to an object, all object methods should be synchronized.

Class SynchronizedCounter

```
public class SynchronizedCounter{
  int value=0;
  String name;

  SynchronizedCounter(String name){
    this.name=name;
  }

  public synchronized void increment() {
    int temp = value;    //read[v]
    Simulate.HWinterrupt();
    value=temp+1;        //write[v+1]
    System.out.println(name+ " value is: "+ value);
}}
```

Another Turnstile

```
public class GoodTurnstile extends Thread{
  String name;
  SynchronizedCounter count;
  public final static int MAX = 20;

  GoodTurnstile(String threadName,
                     SynchronizedCounter c){
    name=threadName; count = c;
  }

  public void run() {
    //as before
    ...
  }
}
```

Another Garden

```
public class GoodGarden {
  public static void main(String[] args){
    SynchronizedCounter people
             = new SynchronizedCounter("Counter");
    GoodTurnstile west
              = new GoodTurnstile("West", people);
    GoodTurnstile east
              = new GoodTurnstile("East",people);
    west.start();
    east.start();
  }
}
```

Good counting

Let us execute `GoodGarden`

```
East value is: 0
West value is: 0
East value is: 1
West value is: 1
Counter value is: 1
Counter value is: 2
...
Counter value is: 38
West value is: 20
Counter value is: 39
East value is: 20
Counter value is: 40
```

Java synchronized statement

Access to an object may also be made mutually exclusive by using the synchronized statement:

```
synchronized (object) { statements }
```

A less elegant way to correct the example would be to modify the Turnstile.run() method:

```
synchronized(people) {people.increment();}
```

Model

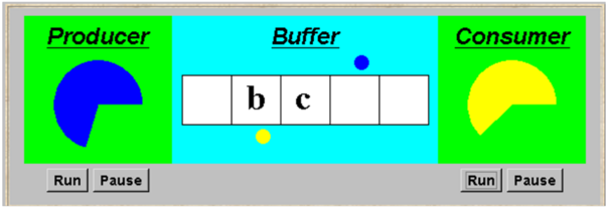
```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

set VarAlpha =
          {value.{read[T],write[T], acquire, release}}

TURNSTILE = ···
RUN = ···
INCREMENT = (value.acquire
                  -> value.read[x:T]->value.write[x+1]
               -> value.release->RUN
            )+VarAlpha.
```

3. Problem: Lack of a Coordination Mechanism

Coordination mechanism

Allow one process to wait for another or notify a waiting process that it can proceed.

Problem: Bounded Buffer



- A bounded buffer consists of a fixed number of slots (size).
- Items are put into the buffer by a producer and a consumer get them.
- It can be used to smooth out transfer rates between the producer and consumer.

LST for Buffer

```
BUFFER(Size=5) = COUNT[0],
COUNT[count:0..N]=
          (when (count<Size) put->COUNT[count+1]
           |when (count>0) get->COUNT[count-1]
           ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER =
           (PRODUCER||BUFFER(5)||CONSUMER).
```

Interface Buffer<E>

The buffer will be implemented as a general-structure class that can buffer any type of Java object.

```
public interface Buffer<E> {

   //put object into buffer
    public void put(E o)
             throws InterruptedException;

   //get an object from buffer
   public E get()
              throws InterruptedException;
}
```

Need of Coordination

- To put an item in the buffer it should be an empty place.
- To get and element from the buffer it should be at least an element on it.

4. Solution: Monitors

Condition synchronization in Java

Java provides a thread wait set per object with the following methods:

- public final void notify()
  Wakes up a single thread that is waiting on this object.
- public final void notifyAll()
  Wakes up all threads that are waiting on this object.
- public final void wait() throws InterruptedException
  - Waits to be notified by another thread.
  - The waiting thread releases the synchronization lock associated with the monitor.
  - When notified, the thread must wait to reacquire the monitor before resuming execution.

Synchronization: from FSP to Java

FSP:
when cond act -> NEWSTAT

Standard coding idiom for expressing guarded methods is simple a while loop invoking a wait.

Java:

```
public synchronized void act()
               throws InterruptedException {
     while (!cond) wait();
     // modify monitor data
     notifyAll()
}
```

Let’s us follow with the implementation

```
public synchronized void put(E o)
              throws InterruptedException {
        while (count==size) wait();
        buf[in] = o;++count;in=(in+1)%size;
        notifyAll();
}

public synchronized E get()
              throws InterruptedException {
        while (count==0) wait();
        E o = buf[out];
        buf[out]=null;
        --count;out=(out+1)%size;
        notifyAll();
        return (o);
 }
```

Monitors

Monitors : Encapsulated data + access procedures
+mutual exclusion + condition synchronization
+single access procedure active in the monitor

- Active entities (that initiate actions) are threads.
- Passive entities (that respond to actions) are monitors.

Each guarded action in the model of a monitor is implemented as a synchronized method which uses a while loop and wait() to implement the guard.

- The while loop condition is the negation of the model guard condition.
- Changes in the state of the monitor are signaled to waiting threads using notify() or notifyAll().

Class Exercise: MICRO_ACCOUNT

Consider a bank dealing with micro accounts.

- You can just deposit or withdraw one euro in each operation.
- There is a bound *M* to the maximum quantity of money .

```
const M=3
MICRO_ACCOUNT = MONEY[0],
MONEY[i:0..M] = (when i>0 withdrawn->MONEY[i-1]
                 |when i<M deposit ->MONEY[i+1]).
CLIENT=(withdrawn->CLIENT|deposit ->CLIENT).

||BANK = (alice:CLIENT||bob:CLIENT
          ||{alice, bob}::MICRO_ACCOUNT).
```

(1) Write a monitor for `MICRO_ACCOUNT`, complete:

```
public class MicroAccount {
   private int i=0;
   private int M;

   public MicroAccount(int bound) {
     M=bound;
   }
...
}
```

(2) Taking into account the implementation of `MicroAccount` reason about possible deadlocks.

```
public class  Client extends Thread{
   MicroAccount account; String name;
   public Client(String name, MicroAccount account){
      this.name=name; this.account=account;
   }
   public void run(){
     while(true){
       try{
        if(Math.random()<0.5) {
           account.deposit();
           System.out.println(name + " deposit one euro");
           } else {
             account.withdraw();
             System.out.println(name + " withdraw one euro");
           }
          Thread.sleep(3000);
          }catch (InterruptedException e){}
} } }
```

and

```
public class Bank {

   public static void main(String[] args){
      MicroAccount account= new MicroAccount(3);
      Client alice = new Client("Alice", account);
      Client bob = new Client("Bob", account);
      alice.start();
      bob.start();
   }
}
```

5. Problem: Deadlock

Problem: Deadlock

Deadlock occurs in a system when all its constituents are blocked. There are no eligible actions to be performed.

Conditions (necessary and sufficient):

- Serially reusable resources: the processes share resources used under mutual exclusion.
- Incremental acquisition: processes hold on to resources allocated to them while waiting to acquire others.
- No pre-emption: once acquired by a process, resources cannot be pre-empted (forcibly withdrawn).
- Wait-for cycle: a circular chain of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Printer-Scanner System

Processes *P* and *Q* perform the same task, that scanning a document and printing it, by using a printer and scanner.

P acquieres the printer first and *Q* acquieres the scanner first.

```
RESOURCE = (get -> put -> RESOURCE).

P = (printer.get -> scanner.get -> copy
                    -> printer.put -> scanner.put -> P).

Q = (scanner.get -> printer.get -> copy
                    -> printer.put -> scanner.put -> Q).

||SYS = (p:P || q:Q
          || {p,q}::printer:RESOURCE
          || {p,q}::scanner:RESOURCE ).
```

LTS Analyser

Analyser -> Check -> Progress

```
Progress Check...
-- States: 8 Transitions: 11 Memory used: 8359K
Finding trace to cycle...
Finding trace in cycle...
Progress violation for actions:
{p, q}.{copy, {printer, scanner}.{get, put}}
Trace to terminal set of states:
p.printer.get
q.scanner.get
Cycle in terminal set:
Actions in terminal set:
{}
Progress Check in: 16ms
```

6. (Partial) Solution: Timeout

Tentative Solution

Set a timeout. This denies the second deadlock condition of incremental acquisition

```
RESOURCE = (get->put->RESOURCE).

P = (printer.get-> GETSCANNER),
GETSCANNER = (scanner.get->copy->printer.put ->scanner.put->P
                    |timeout -> printer.put->P).

Q = (scanner.get-> GETPRINTER),
GETPRINTER = (printer.get->copy->printer.put ->scanner.put->Q
                    |timeout -> scanner.put->Q).

||SYS = ···
```

Problem:
p.printer.get->p.timeout -> p.printer.put->p.printer.get-> ···

Recommended reading

- In order to get more information you can read Chapters 3,4,5 and 6 of:

  Jeff Magee and Jeff Cramer
  Concurrency, State Models & Java Programs
  John Wiley & and Sons, 2006.
  `http://www.doc.ic.ac.uk/~jnm/book/`
- You can also read the corresponding slides:

  `http://www.doc.ic.ac.uk/~jnm/book/slides.html`