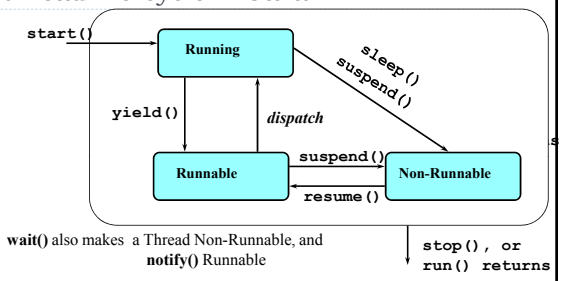
Question asked after Prev. Lecture

Abhik Roychoudhury
National University of Singapore

Q1. From post-it note

- If JVM is running inside one OS process, are the threads of a Java program truly parallel?
- Answer
  - The threads will be time-shared. At any point, you can assume that one thread is scheduled to run. There might be other threads which are schedulable – but only one is running. This corresponds to the sub-states of the **Alive** state in the thread life cycle discussed in today's lecture.

thread life-cycle in Java



Q2. Concurrency and parallelism

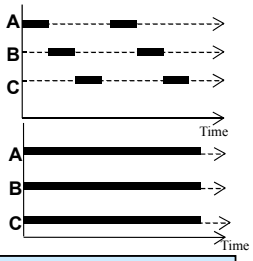
- Why interleaved execution simulates the parallel behavior irrespective of processor speeds?
- Answer: Consider two processes
  - A = a -> A
  - B = b -> B
  - If they are running in two processors a could finish before b, and vice-versa – depending on the processor speeds.
  - This captured by two interleavings (global traces)
    - a -> b -> …
    - b -> a -> …

Definitions

- ***Concurrency***
  - ***Logically* simultaneous processing.**

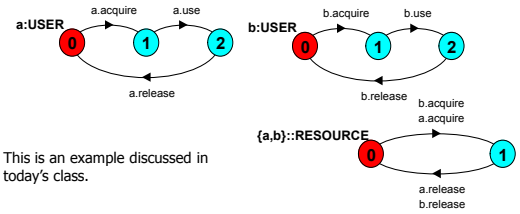
  Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.
- ***Parallelism***
  - ***Physically* simultaneous processing.**

  Involves multiple PEs and/or independent device operations.



Both concurrency and parallelism require controlled access to shared resources . We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.

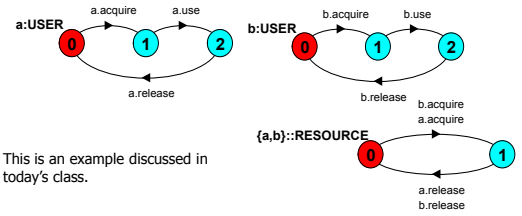
Q3. Any guarantees of mutual exclusion



This is an example discussed in today's class.

```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).
RESOURCE_SHARE = (a:USER || b:USER
                  || {a,b}::RESOURCE).
```

Q3. Reasoning about per-process code



This is an example discussed in today's class.

1. From the RESOURCE process, we can see that after an acquire action, a release action must be executed, before another acquire action can happen.
2. We can also see that *a.acquire* cannot be immediately followed by *b.release* in the RESOURCE process --- since *b.release* is a shared action and for it to take place in the b::USER --- there should have been *b.acquire* prior to it

Important note

- While doing multi-threaded programming
  - We are often resorting to such per-process reasoning.
  - It would be nice to have a global state machine (obtained by composing all the processes) and perform formal verification – but realistically this is often **not** done.
  - For this reason, it is important for the programmer to at least do such per-process reasoning (or some limited reasoning about communication) while writing the code.

Q4. Link between processes & Java code

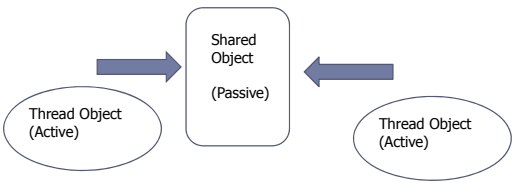
- You said
  - RADIO = on -> off -> RADIO
  - Here on, off are atomic actions. How do you know?
- Answer:
  - It is the other way round. Whatever is atomic, I show as an action in the process equations.
  - Now in reality, the **on** action could be a method call in Java, say **on(),** which is atomically executed because it is written as a synchronized method.
    - **synchronized on(){ …**
  - This also shows some of the linkage between process equations and multi-threaded Java code.

Shared Objects

Abhik Roychoudhury
CS 3211
National University of Singapore

Reading material: Chapter 4 of Textbook.

Thread Communication



Arbitrary interleaving of accesses possible.

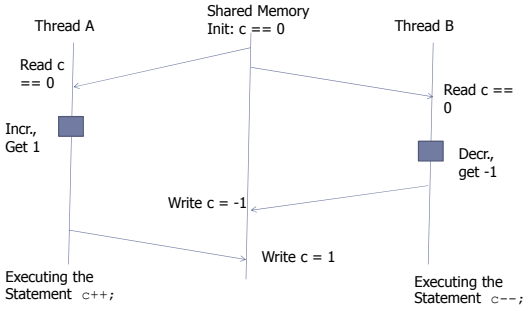
Interference between threads

```
class Counter {
       private int c = 0;
       public void increment() {
                          c++;
       }
       public void decrement() {
                          c--;
       }
       public int value() {
                          return c;
       }
}
```

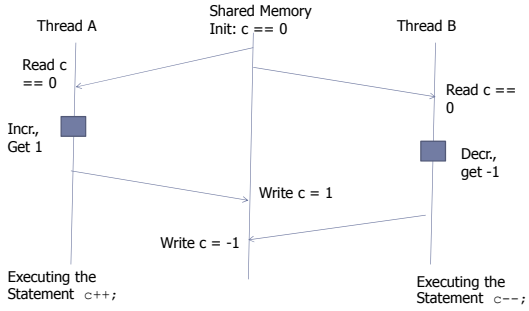
"Correct" operation: One execution of increment adds 1
One execution of decrement subtracts 1

Inter-thread interference from prevent the result from being so. Why?

Interference between Threads – (1)



Interference between Threads – (2)



What do we need?

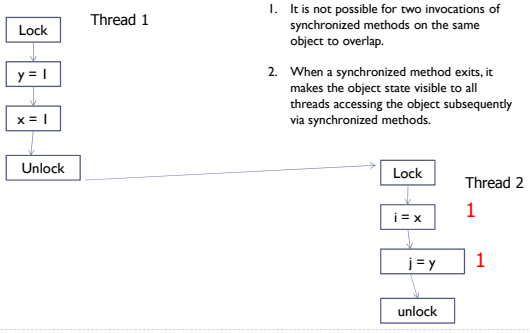
- Mutually exclusive access to the counter
- How to do that?
  - Language level construct – Lock.
  - Acquire lock prior to any access of counter.
  - Release lock after any access of counter.
- Does it require locking discipline then?
  - Well, accesses happen through methods of the shared object
    - In this case, objects of the Counter class
    - Mark these methods as "synchronized"
    - Avoid managing locks for each call of these methods !!

Synchronized Methods

Java programming provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*

```
public class SynchronizedCounter {
      private int c = 0;
      public synchronized void increment() {
                              c++;
      }
      public synchronized void decrement() {
                              c--;
      }
      public synchronized int value() {
                              return c;
      }
}
```


Why synchronized methods?

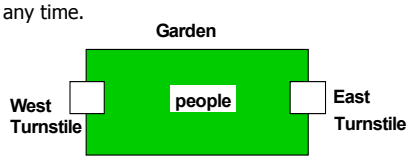


1. It is not possible for two invocations of synchronized methods on the same object to overlap.
2. When a synchronized method exits, it makes the object state visible to all threads accessing the object subsequently via synchronized methods.

4.1 Interference

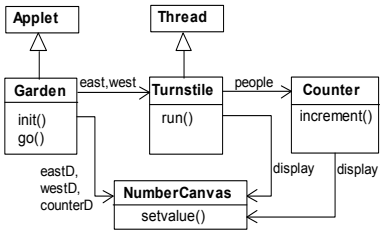
Ornamental garden problem:

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.

ornamental garden Program - class diagram



The **`Turnstile`** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for 0.5 second and then invoking the **`increment()`** method of the counter object.

ornamental garden program

The **`Counter`** object and **`Turnstile`** threads are created by the **`go()`** method of the Garden applet:

```
private void go() {
  counter = new Counter(counterD);
  west = new Turnstile(westD,counter);
  east = new Turnstile(eastD,counter);
  west.start();
  east.start();
}
```


Turnstile class

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  public void run() {
    try{
      display.setvalue(0);
      for (int i=1;i<=Garden.MAX;i++){
        Thread.sleep(500); //0.5 second between arrivals
        display.setvalue(i);
        people.increment();
      }
    } catch (InterruptedException e) {}
  }
}
```

The **`run()`** method exits and the thread terminates after **`Garden.MAX`** visitors have entered.

Counter class

```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value;    //read value
    Simulate.HWinterrupt();
    value=temp+1;        //write value
    display.setvalue(value);
  }
}
```

```
class Simulate {
        public static void HWinterrupt(){
                if (Math.random() < 0.5) Thread.yield();
        }
```

Hardware interrupts can occur at **arbitrary** times.

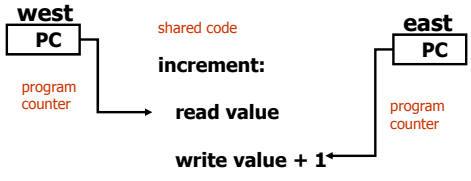
The **`counter`** simulates a hardware interrupt during an **`increment()`**, between reading and writing to the shared counter **`value`**. Interrupt randomly calls **`Thread.yield()`** to force a thread switch.

ornamental garden program - display

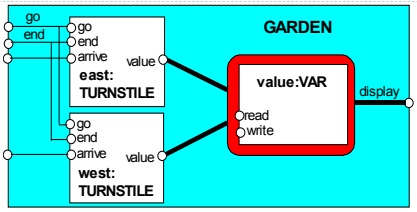
After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. ***Why?***

concurrent method activation

Java method activations are not atomic - thread objects **`east`** and **`west`** may be executing the code for the increment method at the same time.



ornamental garden Model



Process **VAR** models read and write access to the shared counter **value**.

Increment is modeled inside **TURNSTILE** since Java method activations are not atomic i.e. thread objects **east** and **west** may interleave their **read** and **write actions**.

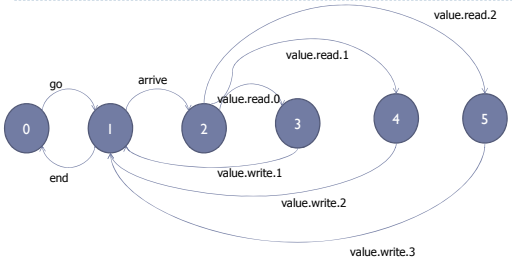
ornamental garden model

```
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,display} ::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

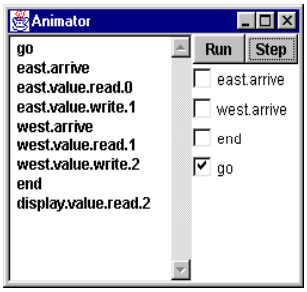
The alphabet of process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The alphabet of **TURNSTILE** is extended with **VarAlpha** to ensure no unintended free actions in **VAR** ie. all actions in **VAR** must be controlled by a **TURNSTILE**.

State Model for Turnstile



checking for errors - animation



Scenario checking - use animation to produce a trace.

Is this trace correct?

checking for errors - exhaustive analysis

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

```
TEST       = TEST[0],
TEST[v:T]  =
     (when (v<N){east.arrive,west.arrive}->TEST[v+1]
     |end->CHECK[v]
     ),
CHECK[v:T] =
    (display.value.read[u:T] ->
       (when (u==v) right -> TEST[v]
       |when (u!=v) wrong -> ERROR
       )
    )+{display.VarAlpha}.

TESTGARDEN =  (GARDEN || TEST)
```

Like **STOP**, **ERROR** is a predefined FSP local process (state), numbered -1 in the equivalent LTS.

ornamental garden model - checking for errors

```
||TESTGARDEN = (GARDEN || TEST).
```

Use ***LTSA*** to perform an exhaustive search for **ERROR**.

```
Trace to property violation in TEST:
   go
   east.arrive
   east.value.read.0
   west.arrive
   west.value.read.0
   east.value.write.1
   west.value.write.1
   end
   display.value.read.1
   wrong
```

LTSA produces the shortest path to reach **ERROR**.

Interference and Mutual Exclusion

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.

Methods with mutually exclusive access can be modeled as atomic actions.

4.2 Mutual exclusion in Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword **`synchronized`**.

We correct **`COUNTER`** class by deriving a class from it and making the increment method synchronized:

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
     {Counter(n);}

   synchronized void increment() {
        Counter.increment();
   }
}
```


mutual exclusion - the ornamental garden

Java associates a ***lock*** with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.

Java synchronized statement

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

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

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

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

Why is this "less safe"?

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

A "less safe" way

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  public void run() {
    try{
      display.setvalue(0);
      for (int i=1;i<=Garden.MAX;i++){
        Thread.sleep(500); //0.5 second between arrivals
        display.setvalue(i);
        synchronized(people){ people.increment();}
      }
    } catch (InterruptedException e) {}
  }
}
```


Why is it less safe?

The lock is not embedded in the counter object itself.

Every "user" of the counter object (in this case the turnstile threads) will have to take the responsibility of imposing the lock, prior to manipulating the shared counter object.

This is an issue we will always face while programming mutually exclusive access to shared objects in Java.

Important programming trick for multi-threaded programming!

Recursive locking in Java

If a thread t acquires a lock on an object o, t can repeatedly lock o.

The lock counts how many times it has been acquired by the same thread, and does not allow another thread to access object o.

This allows the synchronized methods to be recursive, e.g. consider

```
public synchronized void increment(int n){
      if (n>0){
            ++value;
             increment(n-1);
      } else return;
}
```

What would happen on a call to increment(5) if recursive locking was not allowed in Java?

4.3 Modeling mutual exclusion

To add locking to our model, define a **LOCK**, compose it with the shared **VAR** in the garden, and modify the alphabet set :

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

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

Modify **TURNSTILE** to acquire and release the lock:

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


Revised ornamental garden model - checking for errors

A sample animation execution trace

```
go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
correct
```

Use TEST and ***LTSA*** to perform an exhaustive check.

Is TEST satisfied?

COUNTER: Abstraction using action hiding

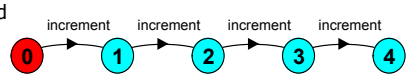
To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For **SynchronizedCounter** we hide **read**, **write**, **acquire**, **release** actions.

```
const N = 4
range T = 0..N

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

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
             -> (when (x<N) write[x+1]
                 ->release->increment->INCREMENT
                )
             )+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```


COUNTER: Abstraction using action hiding

Minimized LTS:



We can give a more abstract, simpler description of a **COUNTER** which generates the same LTS:

```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).
```

This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.

Summary

- Concepts
  - **process interference**
  - **mutual exclusion**
- Models
  - **model checking for interference**
  - **modeling mutual exclusion**
- Practice
  - **thread interference in shared Java objects**
  - **mutual exclusion in Java (synchronized objects/methods).**