Interference IV & Condition Synchronisation

Lecture 9

Alma Rahat

CS-210: Concurrency

23 Feb 2021

What did we do in the last session?

- Testing for interference.
- Feedback.

In this lecture...

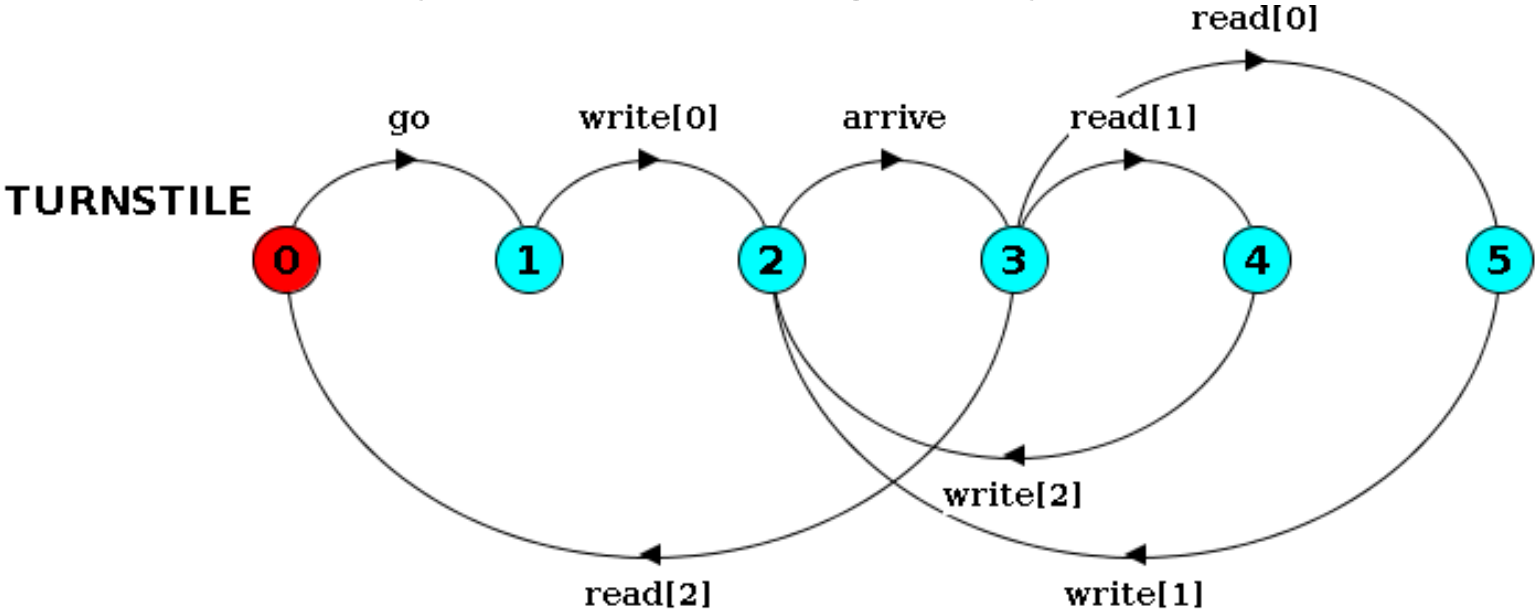
Learning outcomes.

1. To model locks in FSP and ensure that tests pass.
2. To explain what active and passive processes are.
3. To model condition synchronisation.
4. To write Java code for condition synchronisation.

Outline.

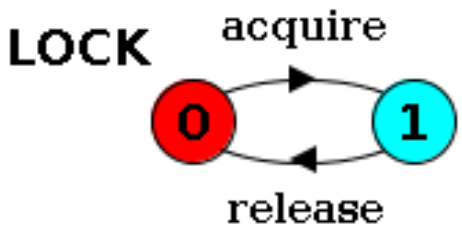
1. Modelling locking and synchronisation.
2. Active and passive processes.
3. Conditional access to shared resources.
4. Model and code.
5. `wait()`, `notify()`, and `notifyAll()`.

How would you model locking and synchronisation?



Remember: `TEST` should remain the same, otherwise we would not know what we did is indeed fixing the problem identified using the original test.

Modelling `synchronized` Method

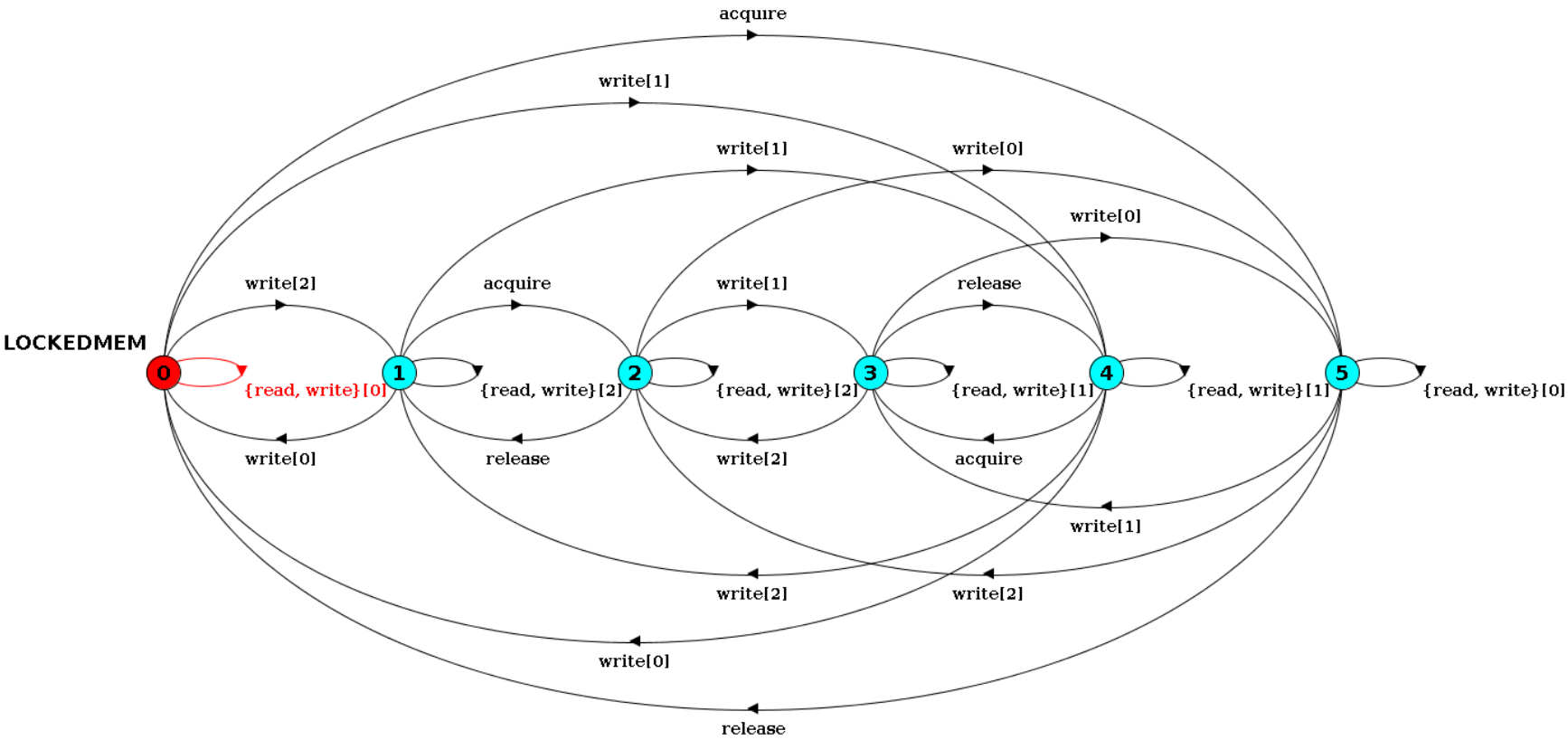


Modelling a lock is straightforward: you `acquire` and then `release`.

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

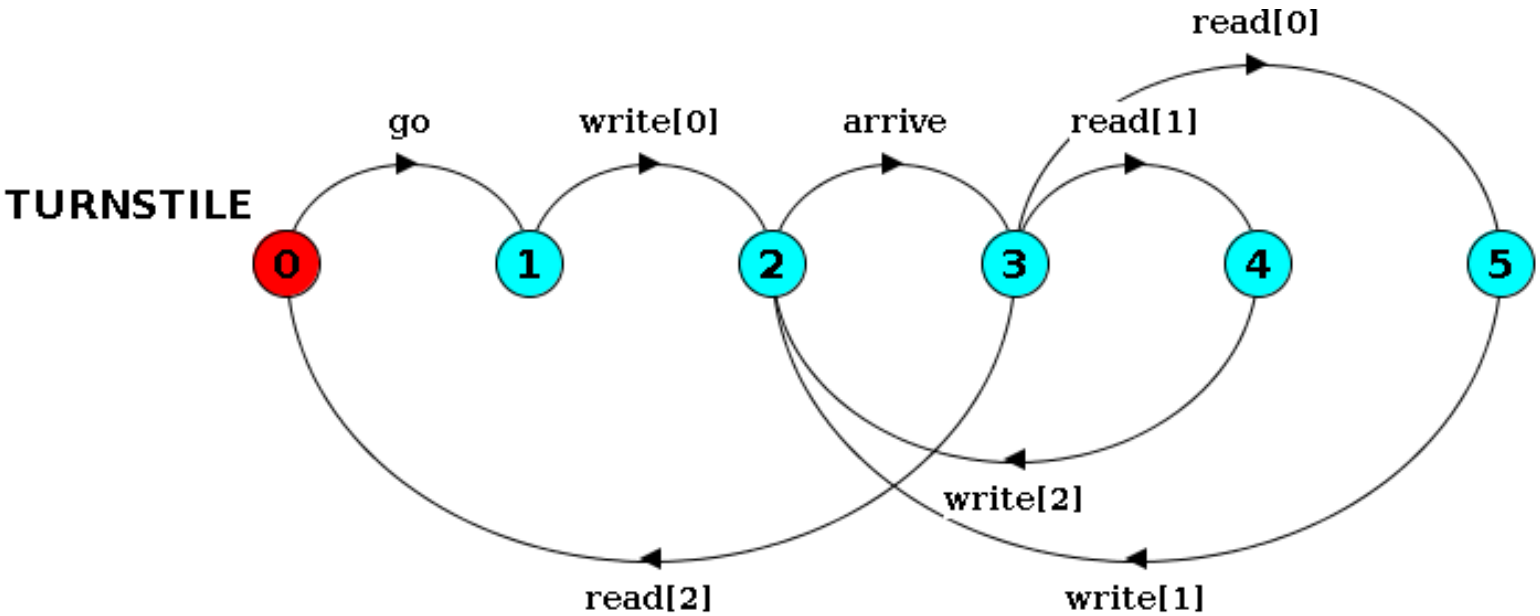
Modelling synchronized Method

```
LOCKEDMEM = (LOCK || MEMORY).
```



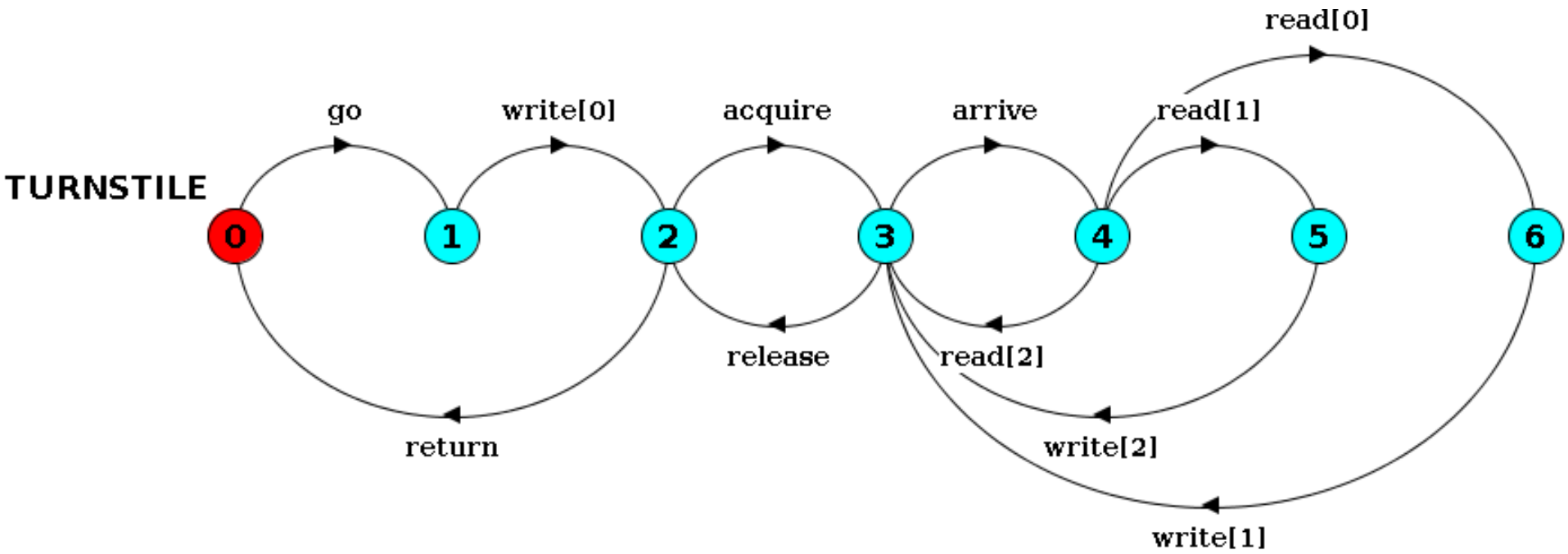
We can compose MEMORY and LOCK together. It does not impose precedence between the alphabets of LOCK and MEMORY. We will impose that later through description of TURNSTILE.

Modelling `synchronized` Method



We want to protect actions from `arrive` till `read[2]` (or return).

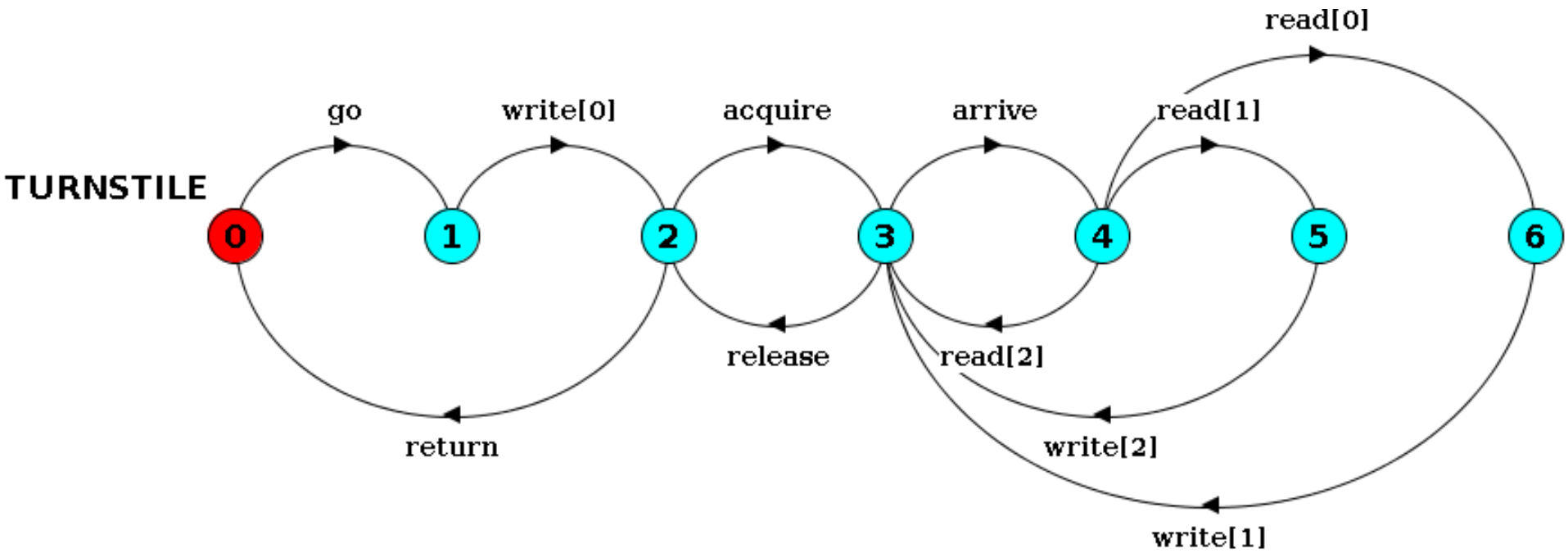
Modelling `synchronized` Method



We introduce additional actions `acquire` and `release` to isolate the actions that were in the method under synchronisation.

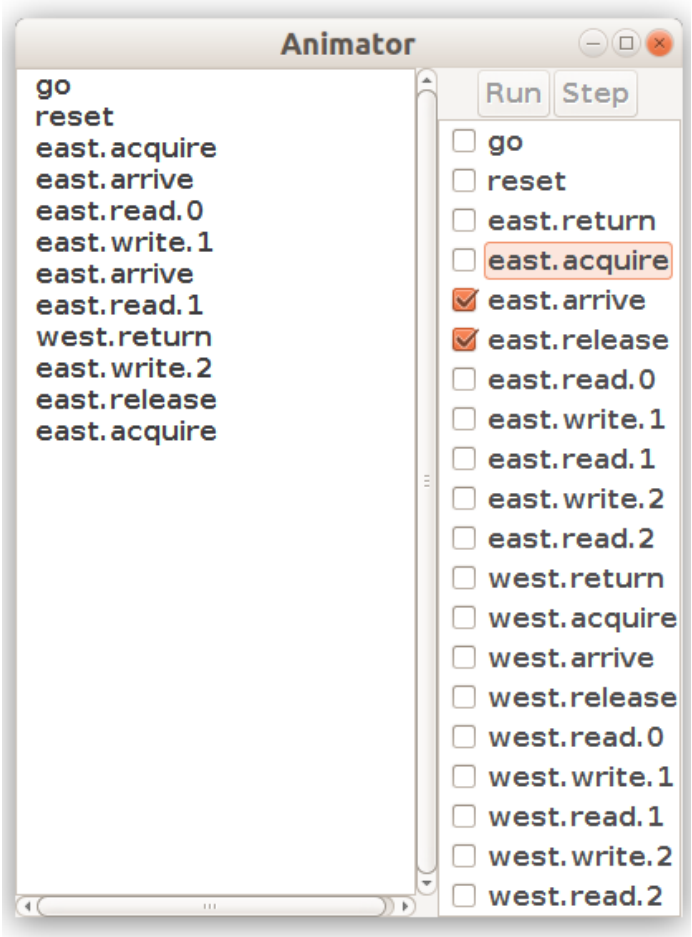
Also, note that we are now using `read[2]` as a way to coming back to `release` if we have done all the incrementing we needed to do.

Modelling synchronized Method




```
TURNSTILE = (go -> write[0] -> RUN),
RUN = (return -> TURNSTILE | acquire -> CRITICAL),
//CRITICAL is the part protected by lock
CRITICAL = (arrive -> INCREMENT | release -> RUN),
INCREMENT = (read[v:0..N-1] -> write[v+1] -> CRITICAL |
read[N] -> CRITICAL).
```

Modelling synchronized Method




```
||GARDEN = (east:TURNSTILE
|| west:TURNSTILE || {east,
west}::LOCKEDMEM) /{reset/{east,
west}.write[0], go/{east,
west}.go}.
```

We remove the relabelling from `read[N]` here as they will be synchronised on their own.

Modelling `synchronized` Method

```
Composition:
TESTGARDEN = GARDEN.east:TURNSTILE || GARDEN.west:TURNSTILE ||
GARDEN.{east,west}::LOCKEDMEM || TEST
State Space:
 7 * 7 * 6 * 5 = 2 ** 12
Analysing using Supertrace (Depth bound 100000 Hashtable size
8000K )...
-- Depth: 0 States: 47 Transitions: 90 Memory used: 49199K
No deadlocks/errors
Analysed using Supertrace in: 2ms
```

This time the test passes with flying colours, so we are proving that mutual exclusion through the use of a lock works.

A quick look at the code...

```
public class SynchronisedCounter extends Counter {

    SynchronisedCounter() {
        super();
    }
    @Override
    public synchronized void increment(){
        int temp = value; //read value
        Simulate.HWInterrupt();
        value = temp + 1; // set value
    }
}
```

A quick look at the code...

```
private boolean synchronised = false;
public void reset(){
    // create counter
    if (!synchronised){
        System.out.println("Running with faulty counter.");
        counter = new Counter();
    }
    else{
        System.out.println("Running with fixed counter.");
        counter = new SynchronisedCounter();
    }
    // create Turnstiles
    west = new Turnstile(counter);
    east = new Turnstile(counter);
}
```

A variable for flipping between faulty and fixed code, and an explicit reset method.

A quick look at the code...

```
public void go() throws InterruptedException{
    reset();
    // create threads
    Thread westThread = new Thread(west, "west");
    Thread eastThread = new Thread(east, "east");
    // start threads
    westThread.start();
    eastThread.start();
    // wait for threads to die
    westThread.join();
    eastThread.join();
}

void flipSynchronised() {
    this.synchronised = !this.synchronised;
}
```

go method is largely the same, but an additional `flipSynchronised` method.

A quick look at the code...

```
Have another go?
input should be:
 0 to exit the program,
 1 to iterate with same settings,
 2 to flip between faulty counter and fixed counter, and run the new settings.
1
Running with faulty counter.
199
Have another go?
input should be:
 0 to exit the program,
 1 to iterate with same settings,
 2 to flip between faulty counter and fixed counter, and run the new settings.
2
Running with fixed counter.
200
Have another go?
input should be:
 0 to exit the program,
 1 to iterate with same settings,
 2 to flip between faulty counter and fixed counter, and run the new settings.
```

No matter how many times you try, the code runs properly, even with the interrupt.

Atomic Variables

Swansea University Prifysgol Abertawe

Non-blocking code – for example, tries to do something, when fails, retries, rather than waiting on locks – can be a good alternative from performance perspective (for simple counting operations).

Atomic variables in Java allow non-blocking operations on variables using low-level machine instructions.

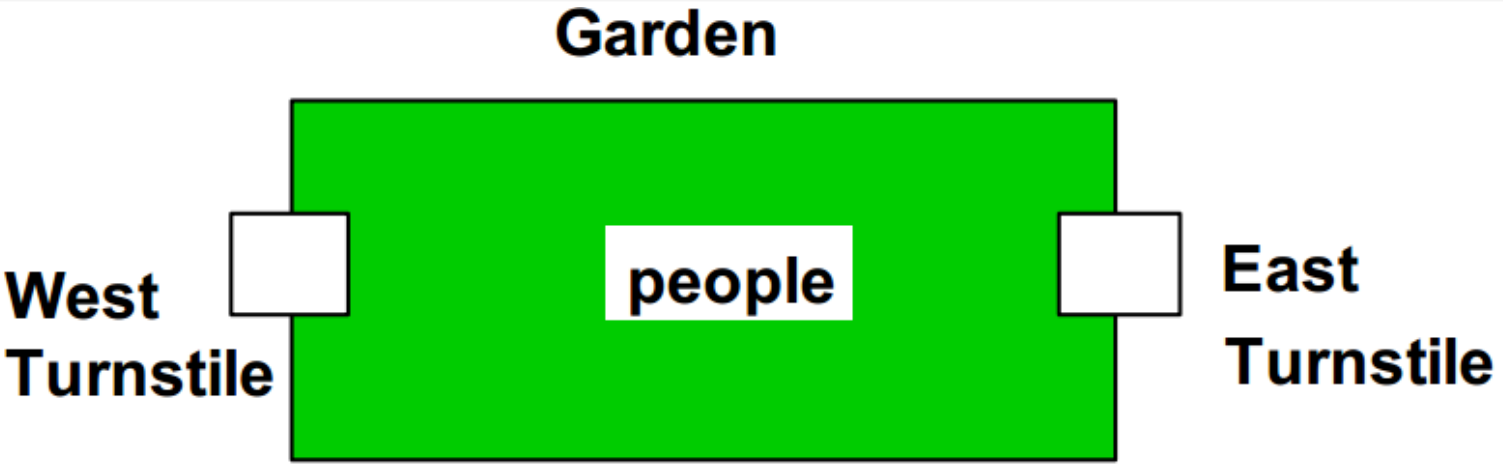
Typical options: `AtomicInteger`, `AtomicLong`,`AtomicBoolean`, and `AtomicReference`.

```
import java.util.concurrent.atomic.AtomicInteger;
public static void main(String[] args) {
    AtomicInteger atomicInteger = new AtomicInteger(100);
    System.out.println(atomicInteger.get());
    atomicInteger.getAndIncrement();
    System.out.println(atomicInteger.get());
}
```

Any questions?

An Ornamental Garden

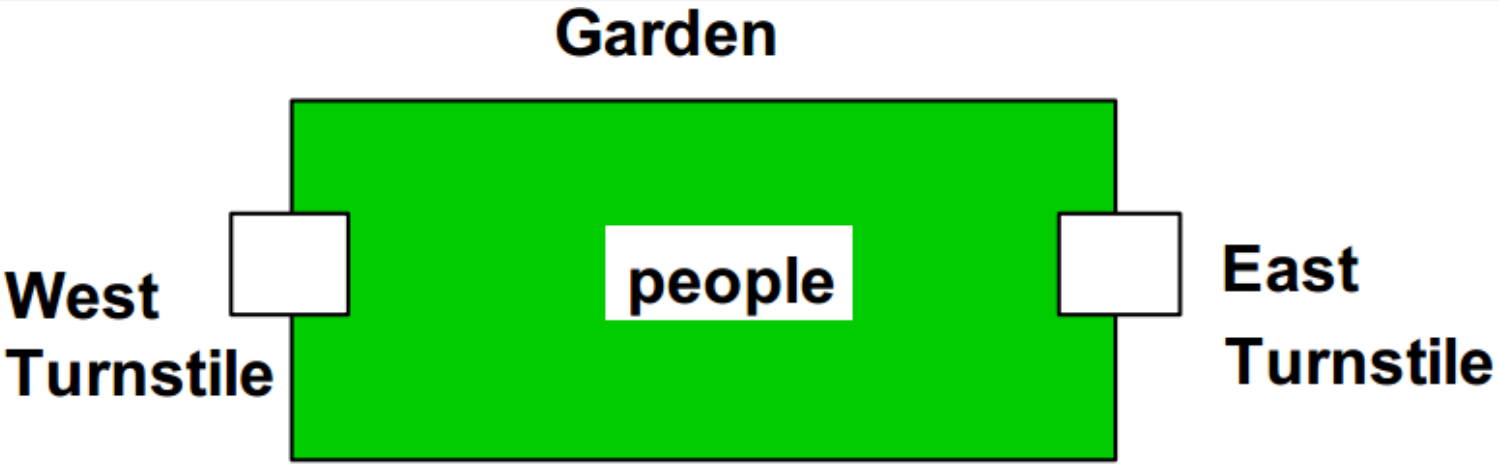
Consider a garden that has two turnstiles that allow people to enter the garden (no need to let them leave!). The turnstiles run parallelly and share a single counter, and each increment this counter 100 times. Ideal way to generate a race condition.



Active and Passive Processes

An Ornamental Garden

Consider a garden that has two turnstiles that allow people to enter the garden (no need to let them leave!). The turnstiles run parallelly and share a single counter, and each increment this counter 100 times. Ideal way to generate a race condition.

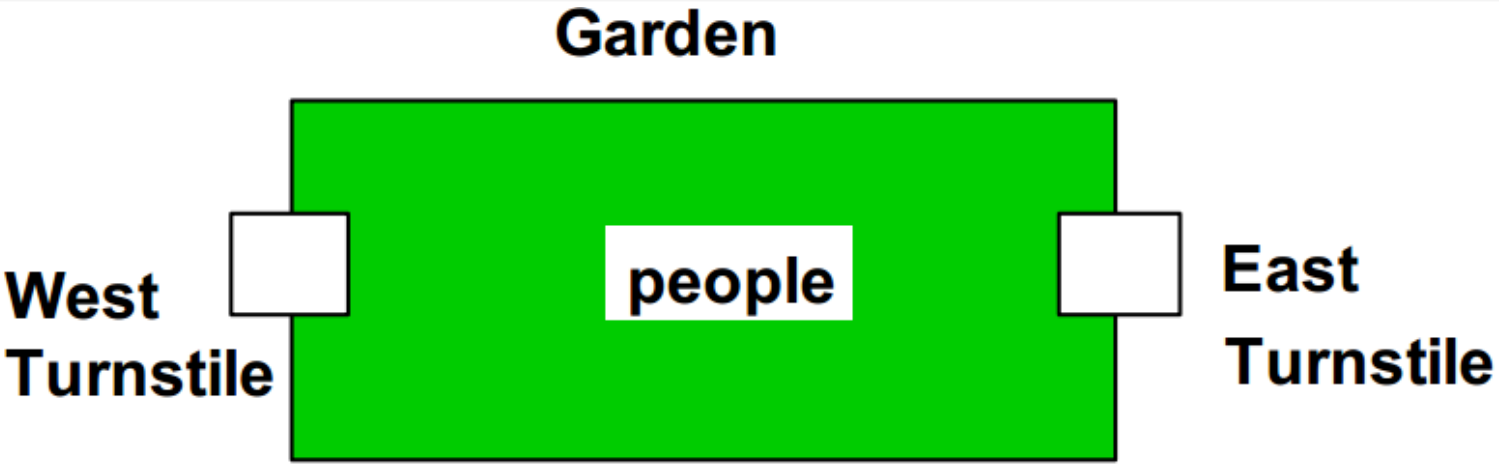


In this scenario, we identified three component processes: west turnstile, east turnstile, and a counter. Which of the processes are **initiating** actions, and which are **responding**?

Active and Passive Processes

An Ornamental Garden

Consider a garden that has two turnstiles that allow people to enter the garden (no need to let them leave!). The turnstiles run parallelly and share a single counter, and each increment this counter 100 times. Ideal way to generate a race condition.

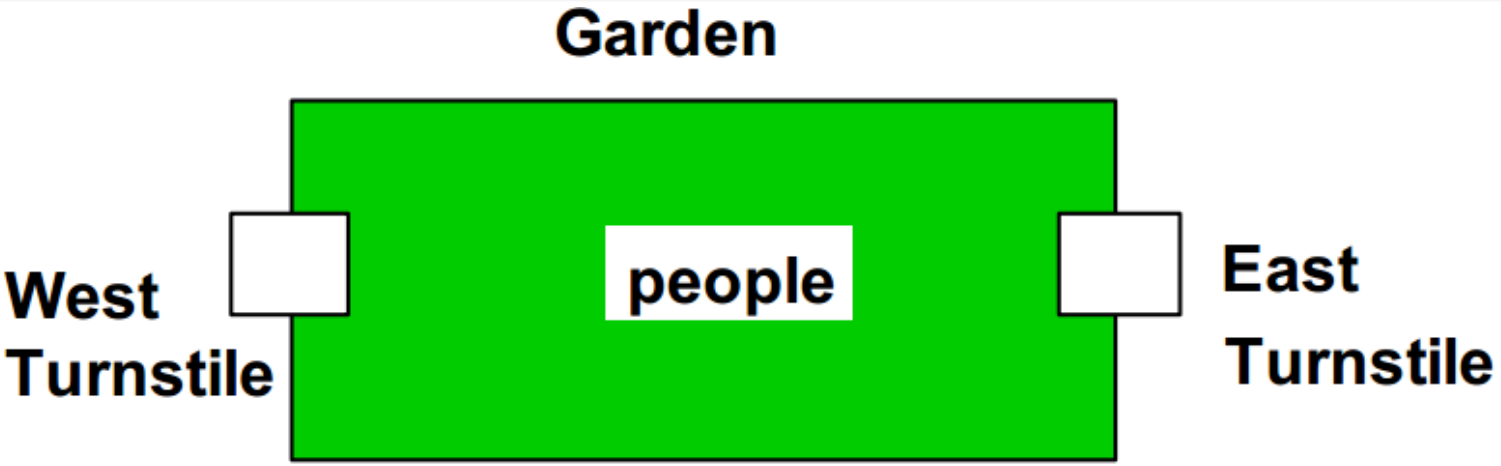


In this scenario, we identified three component processes: west turnstile, east turnstile, and a counter. Please go to `www.menti.com` and use the code 66 55 56 7.

Active and Passive Processes

An Ornamental Garden

Consider a garden that has two turnstiles that allow people to enter the garden (no need to let them leave!). The turnstiles run parallelly and share a single counter, and each increment this counter 100 times. Ideal way to generate a race condition.



Initiators East and West turnstiles – people enter through these.

Responder Counter – it is incremented as a consequence.

Active and Passive Processes

Swansea University
Prifysgol Abertawe

Active process is a process that initiates action: Implement as a `Thread` in Java.

Passive process is a process that responds to actions: Implement as a Monitor class in Java.

So far, when we had shared resources, we only cared about synchronising the access to the resources between threads, so that issues due to concurrent access, such as interference, do not occur. What if there are additional conditions for accessing the shared resources?

Any questions?

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Question 1: What are the actions?
Please go to `www.menti.com` and use the code 98 52 75 0.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Question 1: What are the actions?
Please go to `www.menti.com` and use the code 98 52 75 0.
`enter` and `leave`.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Question 2: What are the active processes? Please go to `www.menti.com` and use the code 98 52 75 0.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Question 2: What are the active processes? Please go to `www.menti.com` and use the code 98 52 75 0.

Active Initiators: `Entrance` and `Exit`.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Question 3: What are the passive processes? Please go to `www.menti.com` and use the code 56 84 75 0.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

Question 3: What are the passive processes? Please go to `www.menti.com` and use the code 56 84 75 0.

Passive Responder: `Controller`.

Conditional Access to Resources

Scenario

There is a car park with one entrance and one exit, and n spaces. The controller only allows a car in *if there is at least one space left* and *lets one leave if there is at least one car in the car park*.

The resource – spaces – is going to be an attribute of the monitor class: **Controller**, and it will be accessed by the **Entrance** and **Exit** threads based on the two conditions.

We will model the scenario first, and then see how it can be implemented in Java.

Summary

- Active processes should be implemented as Threads.
- Passive processes will contain the shared resources as attributes, and should be a monitor, controlling access.