Properties II

Lecture 16

Alma Rahat

CS-210: Concurrency

17 March 2021

What did we do in the last session?

- Applications of Amdahl's Law.
- Introduction to properties.
    - Car park example.

In this lecture...

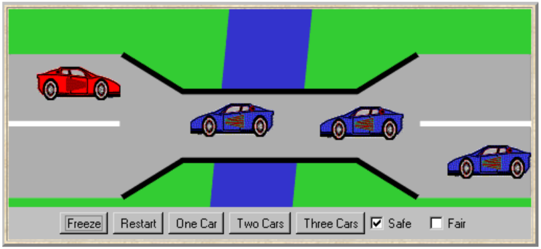
Learning outcomes.

1. To apply safety properties in FSP model development and analyse the system.
2. To apply modelling techniques to identify progress issues.
3. To devise appropriate ways to eliminate progress issues.

Outline.

1. Single lane bridge.
2. Washing machine.
3. Progress and fairness.
4. Example: tossing a coin.
5. Terminal sets.
6. Fixing Single Lane Bridge.

Single Lane Bridge Problem



Source: Chapter 7, Magee & Kramer.

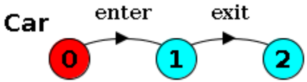
A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in the different directions enter the bridge at the same time.

Single Lane Bridge Problem

- Actions: `enter` and `exit`.
- Processes:
  - Active: `Car`, `Convoy`, `Cars`.
  - Passive: `Bridge`.
- Safety properties: `EntranceOrder`, `ExitOrder` and `SingleCarOnBridge`.

Single Lane Bridge Problem

```
Car = (enter -> exit -> STOP).
```

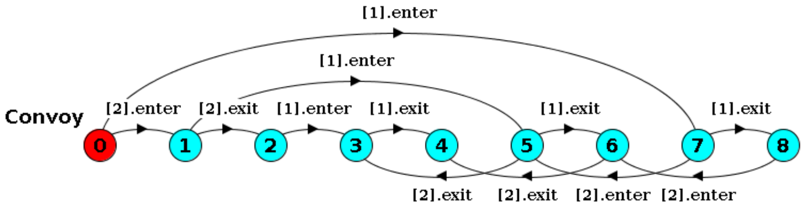


The `Car` model is simple: it just `enters` and then `exits` the bridge. Note that this is only the building block for `Convoy` and `Cars`.

Single Lane Bridge Problem

A Convoy is a queue of cars.

```
const N = 2
range T = 0..N
range ID = 1..N  // lower ID is for the first car on the queue.
Car = (enter -> exit -> STOP).
||Convoy = ([ID]:Car).  // composing multiple cars with IDs.
```

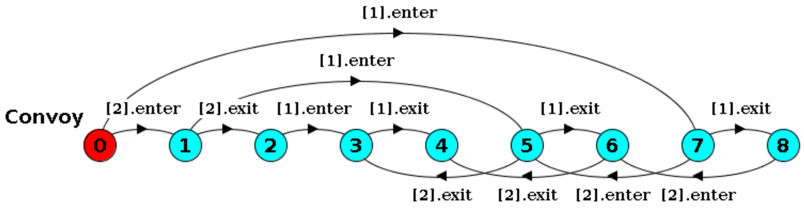


We now have two cars: [1] and [2], each with alphabets enter and exit.
Any issues? Please go to www.menti.com and use the code 78 73 05 3

Single Lane Bridge Problem

A `Convoy` is a queue of cars.

```
const N = 2
range T = 0..N
range ID = 1..N  // lower ID is for the first car on the queue.
Car = (enter -> exit -> STOP).
||Convoy = ([ID]:Car).  // composing multiple cars with IDs.
```

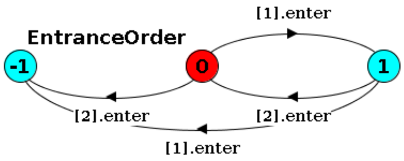


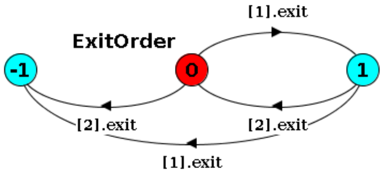
We now have two cars: `[1]` and `[2]`, each with alphabets `enter` and `exit`.Clearly violates the order of entrance and exit.

Single Lane Bridge Problem

We will define a couple of safety properties for *entrance* and *exit*.

```
property EntranceOrder = EntranceOrder[1],
EntranceOrder[id:ID] = ([id].enter ->
EntranceOrder[id%N+1]).
property ExitOrder = ExitOrder[1],
ExitOrder[id:ID] = ([id].exit -> ExitOrder[id%N+1]).
||CheckConvoy = (Convoy || EntranceOrder || ExitOrder).
```



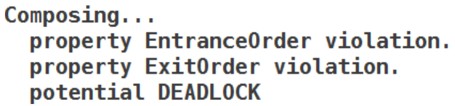


The properties ensures that any invalid sequence of actions is leading to ERROR.

Single Lane Bridge Problem

We will define a couple of safety properties for *entrance* and *exit*.

```
property EntranceOrder = EntranceOrder[1],
EntranceOrder[id:ID] = ([id].enter ->
EntranceOrder[id%N+1]).
property ExitOrder = ExitOrder[1],
ExitOrder[id:ID] = ([id].exit -> ExitOrder[id%N+1]).
||CheckConvoy = (Convoy || EntranceOrder || ExitOrder).
```



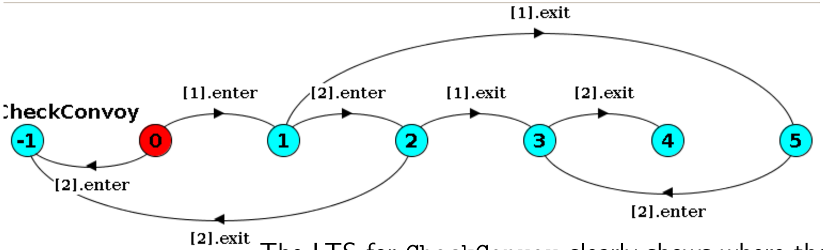
```
Composing...
  property EntranceOrder violation.
  property ExitOrder violation.
  potential DEADLOCK
```

While composing `CheckConvoy`, we can clearly see an example trace to `ERROR`. Note that the deadlock here is simply the end of program, so nothing to worry about.

Single Lane Bridge Problem

We will define a couple of safety properties for *entrance* and *exit*.

```
property EntranceOrder = EntranceOrder[1],
EntranceOrder[id:ID] = ([id].enter ->
EntranceOrder[id%N+1]).
property ExitOrder = ExitOrder[1],
ExitOrder[id:ID] = ([id].exit -> ExitOrder[id%N+1]).
||CheckConvoy = (Convoy || EntranceOrder || ExitOrder).
```

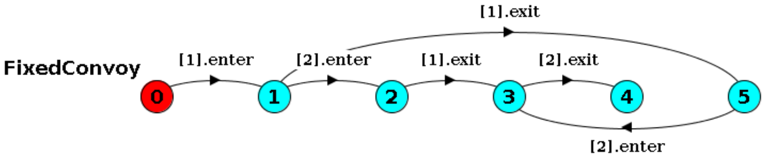


The LTS for CheckConvoy clearly shows where the problems are.

Single Lane Bridge Problem

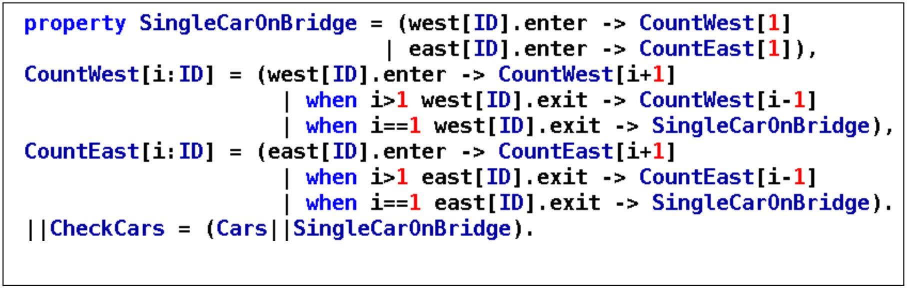
We will create a couple of processes that allow us to model *entrance* and *exit* and associated order of events.

```
Entrance = Entrance[1],
Entrance[i:ID] = ([i].enter -> Entrance[i%N+1]).
Exit = Exit[1],
Exit[i:ID] = ([i].exit -> Exit[i%N+1]).
||FixedConvoy = ([ID]:Car || Entrance || Exit).
||CheckFixedConvoy = (FixedConvoy || EntranceOrder ||
ExitOrder). // for checking the properties.
||Cars = ({west, east}:FixedConvoy).// creating the Cars.
```

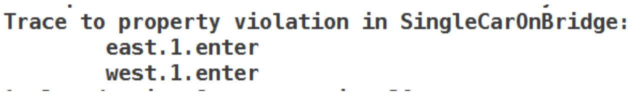


Single Lane Bridge Problem

We will now create the property that check that there is only one car on the bridge.



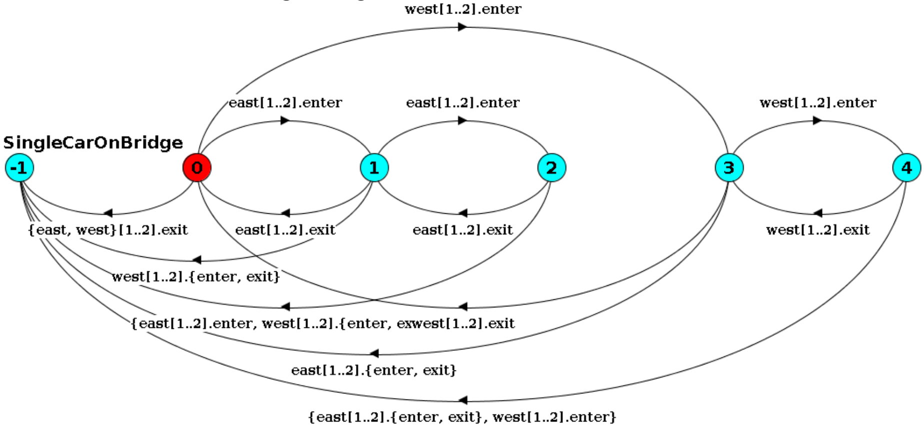
```
property SingleCarOnBridge = (west[ID].enter -> CountWest[1]
                             | east[ID].enter -> CountEast[1]),
CountWest[i:ID] = (west[ID].enter -> CountWest[i+1]
                  | when i>1 west[ID].exit -> CountWest[i-1]
                  | when i==1 west[ID].exit -> SingleCarOnBridge),
CountEast[i:ID] = (east[ID].enter -> CountEast[i+1]
                  | when i>1 east[ID].exit -> CountEast[i-1]
                  | when i==1 east[ID].exit -> SingleCarOnBridge).
||CheckCars = (Cars||SingleCarOnBridge).
```



```
Trace to property violation in SingleCarOnBridge:
        east.1.enter
        west.1.enter
```

Single Lane Bridge Problem

The LTS shows that once we have allowed *east cars* to go ahead, any *west cars* entering will generate an ERROR and *vice versa*.

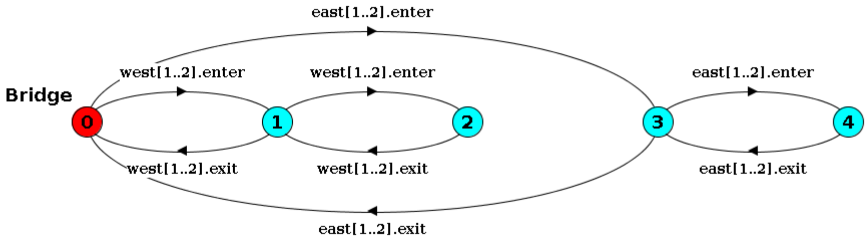


Single Lane Bridge Problem

It is time for us to model the `Bridge`.

```
Bridge = Bridge[0][0],
Bridge[nWest:T][nEast:T] = (
when (nWest == 0 && nEast<N) east[ID].enter -> Bridge[nWest][nEast + 1]
| when (nEast>0) east[ID].exit -> Bridge[nWest][nEast - 1]
| when (nEast==0 && nWest<N) west[ID].enter -> Bridge[nWest+1][nEast]
| when (nWest>0) west[ID].exit -> Bridge[nWest-1][nEast]
).

||SingleLane = (Cars || Bridge).
||CheckSingleLane = (SingleLane || SingleCarOnBridge).
```



Any questions?

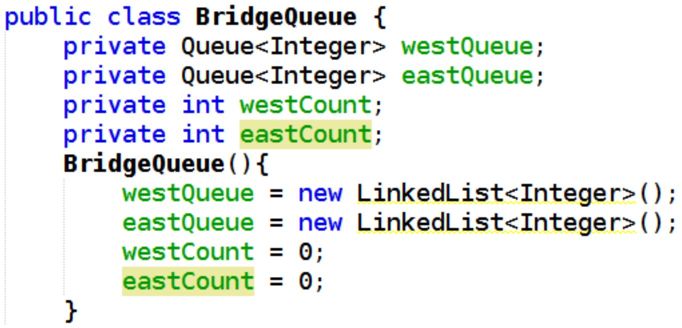
Coding the Bridge

```java
public class Bridge {
    private int westCount = 0;
    private int eastCount = 0;
    public synchronized void westEnter()
            throws InterruptedException{
        while (eastCount > 0) wait();
        westCount += 1;
    }
    public synchronized void westExit(){
        westCount -= 1;
        notifyAll();
    }
    public synchronized void eastEnter()
            throws InterruptedException{
        while (westCount > 0) wait();
        eastCount += 1;
    }
    public synchronized void eastExit(){
        eastCount -= 1;
        notifyAll();
    }
}
```

A simple Bridge class without any queues for cars from west and east; just using simple counters.

Up to date code on Github repository.

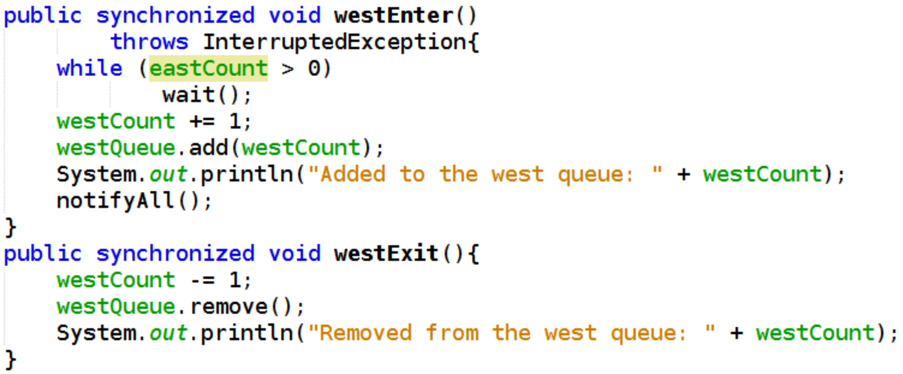
Coding the Bridge



```
public class BridgeQueue {
    private Queue<Integer> westQueue;
    private Queue<Integer> eastQueue;
    private int westCount;
    private int eastCount;
    BridgeQueue(){
        westQueue = new LinkedList<Integer>();
        eastQueue = new LinkedList<Integer>();
        westCount = 0;
        eastCount = 0;
    }
```

An improved version, where we use queues instead to hold cars from west and east. Note that by using linked list first-in-first-out queue implementation, we are making sure that an element that enters the queue first, exits first.

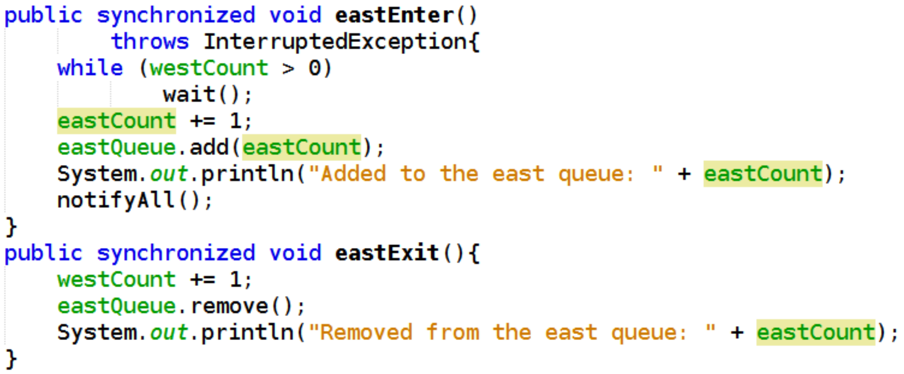
Coding the Bridge



```java
public synchronized void westEnter()
        throws InterruptedException{
    while (eastCount > 0)
            wait();
    westCount += 1;
    westQueue.add(westCount);
    System.out.println("Added to the west queue:  " + westCount);
    notifyAll();
}
public synchronized void westExit(){
    westCount -= 1;
    westQueue.remove();
    System.out.println("Removed from the west queue:  " + westCount);
}
```

It has methods for *westEnter* and *westExit*, adding and removing from the shared `westQueue` respectively.

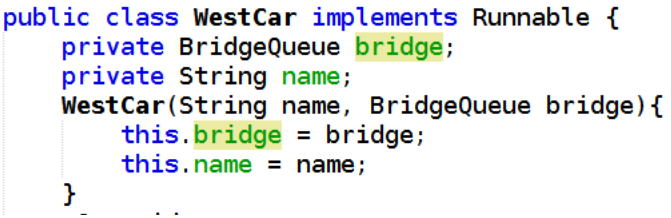
Coding the Bridge



```
public synchronized void eastEnter()
        throws InterruptedException{
    while (westCount > 0)
            wait();
    eastCount += 1;
    eastQueue.add(eastCount);
    System.out.println("Added to the east queue: " + eastCount);
    notifyAll();
}
public synchronized void eastExit(){
    westCount += 1;
    eastQueue.remove();
    System.out.println("Removed from the east queue: " + eastCount);
}
```

Similar methods for *eastEnter* and *eastExit*, adding and removing from the shared `eastQueue` respectively.

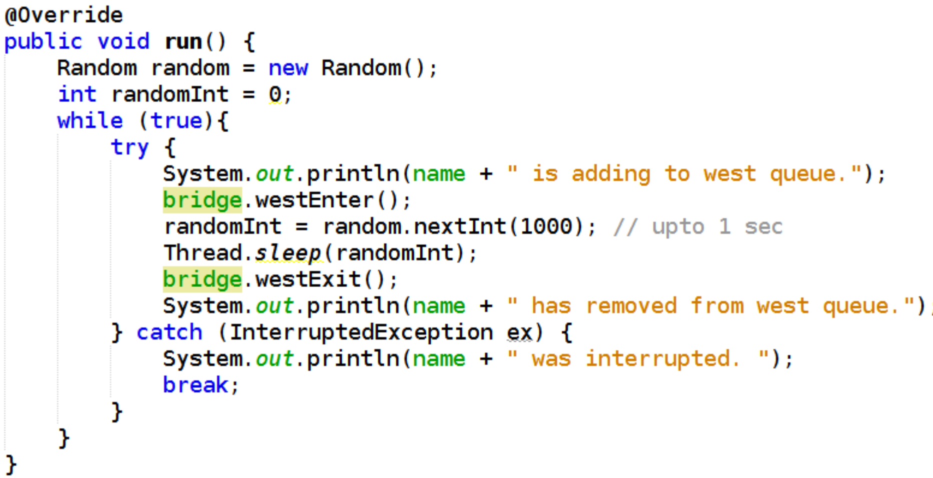
Coding the Bridge



```java
public class WestCar implements Runnable {
    private BridgeQueue bridge;
    private String name;
    WestCar(String name, BridgeQueue bridge){
        this.bridge = bridge;
        this.name = name;
    }
```

Cars are coming from west and east. So, we are going to implement each west and east as Threads. Here, we have the `WestCar` implementation: it has the shared `BridgeQueue` instance, and also a `name`.

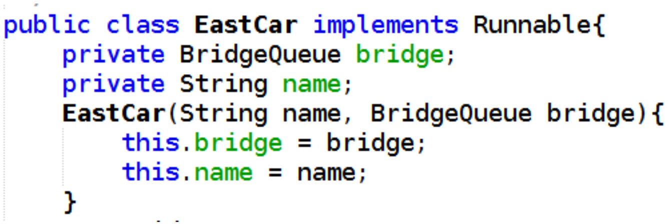
Coding the Bridge



```
@Override
public void run() {
    Random random = new Random();
    int randomInt = 0;
    while (true){
        try {
            System.out.println(name + " is adding to west queue.");
            bridge.westEnter();
            randomInt = random.nextInt(1000); // upto 1 sec
            Thread.sleep(randomInt);
            bridge.westExit();
            System.out.println(name + " has removed from west queue.")
        } catch (InterruptedException ex) {
            System.out.println(name + " was interrupted. ");
            break;
        }
    }
}
```

When this `WestCar` is run, it randomly queues a car, and waits for a bit before removing it from the queue.

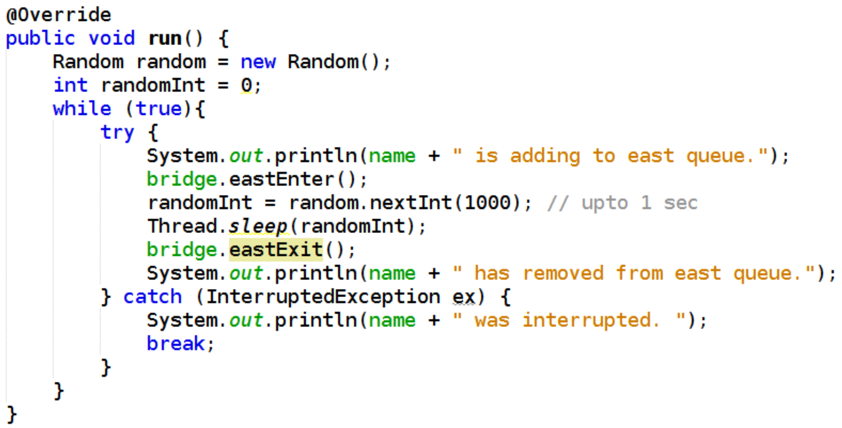
Coding the Bridge



```
public class EastCar implements Runnable{
    private BridgeQueue bridge;
    private String name;
    EastCar(String name, BridgeQueue bridge){
        this.bridge = bridge;
        this.name = name;
    }
```

`EastCar` is the same in principle, but works on the shared `eastQueue`.

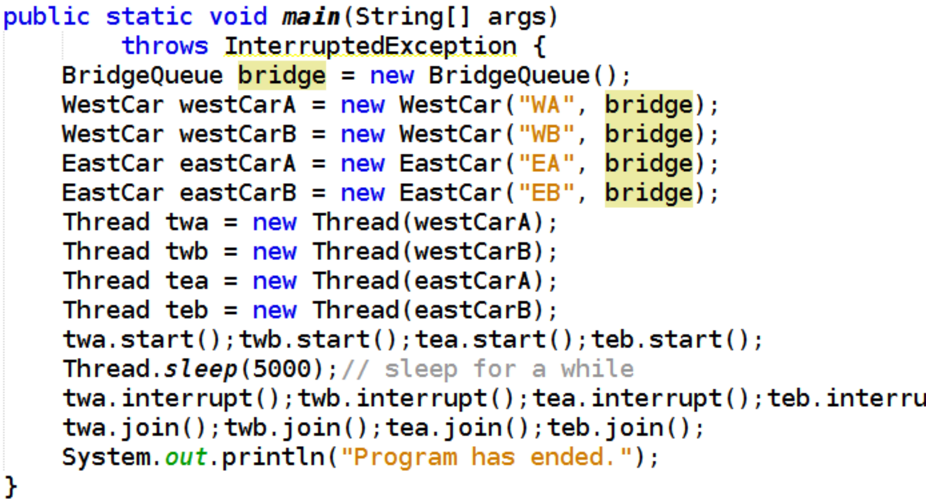
Coding the Bridge



```java
@Override
public void run() {
    Random random = new Random();
    int randomInt = 0;
    while (true){
        try {
            System.out.println(name + " is adding to east queue.");
            bridge.eastEnter();
            randomInt = random.nextInt(1000); // upto 1 sec
            Thread.sleep(randomInt);
            bridge.eastExit();
            System.out.println(name + " has removed from east queue.");
        } catch (InterruptedException ex) {
            System.out.println(name + " was interrupted. ");
            break;
        }
    }
}
```

`EastCar` is the same in principle, but works on the shared `eastQueue`.

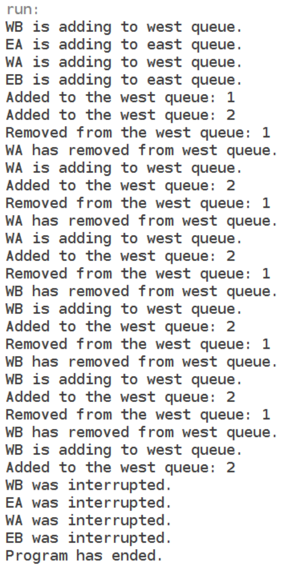
Coding the Bridge



```
public static void main(String[] args)
        throws InterruptedException {
    BridgeQueue bridge = new BridgeQueue();
    WestCar westCarA = new WestCar("WA", bridge);
    WestCar westCarB = new WestCar("WB", bridge);
    EastCar eastCarA = new EastCar("EA", bridge);
    EastCar eastCarB = new EastCar("EB", bridge);
    Thread twa = new Thread(westCarA);
    Thread twb = new Thread(westCarB);
    Thread tea = new Thread(eastCarA);
    Thread teb = new Thread(eastCarB);
    twa.start();twb.start();tea.start();teb.start();
    Thread.sleep(5000);// sleep for a while
    twa.interrupt();twb.interrupt();tea.interrupt();teb.interru
    twa.join();twb.join();tea.join();teb.join();
    System.out.println("Program has ended.");
}
```

In the main program, we just create the `BridgeQueue` and create four different threads to work on it.

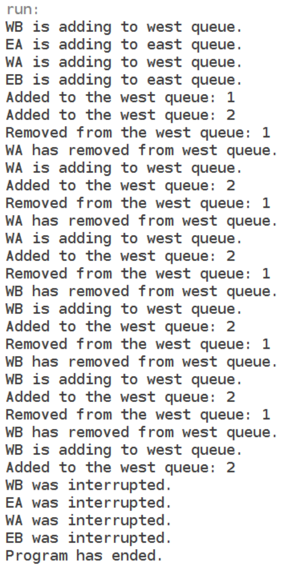
Coding the Bridge



```
run:
WB is adding to west queue.
EA is adding to east queue.
WA is adding to west queue.
EB is adding to east queue.
Added to the west queue: 1
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
WB was interrupted.
EA was interrupted.
WA was interrupted.
EB was interrupted.
Program has ended.
```

This is the output of the program. Is there anything surprising in the output? Please go to `www.menti.com` and use the code 3987 5039.

Coding the Bridge



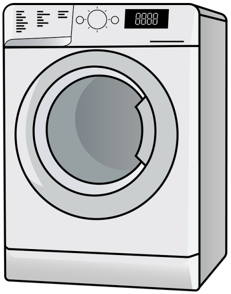
```
run:
WB is adding to west queue.
EA is adding to east queue.
WA is adding to west queue.
EB is adding to east queue.
Added to the west queue: 1
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
WB was interrupted.
EA was interrupted.
WA was interrupted.
EB was interrupted.
Program has ended.
```

This is the output of the program. Is there anything surprising in the output? Please go to `www.menti.com` and use the code 3987 5039.

EA and EB never gets to go through the bridge and add to the queue, as cars continue to come from the west. This is called *starvation*, i.e. some of the threads never get to do anything, and keep waiting on others.

Any questions?

Example: Washing Machine and Safety



In a washing machine, you can `turn it on` and then `turn it off`. Once turned on, it has a `closed door`. At this stage, you can `open` the machine, and find it to be `empty`. Now, you can `load` it and make it `full`. When full, you can `unload` it and make it go back to original `empty` state. When it is full, you can close the door, and `start` the washing process, where you `wash`, `rinse`, and subsequently `dry`. At any point during the washing process, you should be able to `pause` and `resume` current operation.
There is a safety *cycle* property: wash must come before rinse, and both must happen before dry.
Produce FSP code for the model and the safety property.

Working through the problem

In a washing machine, you can `turn it on` and then `turn it off`. Once turned on, it has a `closed door`. At this stage, you can `open` the machine, and find it to be `empty`. Now, you can `load` it and make it `full`. When full, you can `unload` it and make it go back to original `empty` state. When it is full, you can close the door, and `start` the washing process, where you `wash`, `rinse`, and subsequently `dry`. At any point during the washing process, you should be able to `pause` and `resume` current operation.

FSP

```
Machine = (on -> ClosedDoor[0]),
ClosedDoor[i:0..1] = (when i==0 open -> Empty |
                      when i==0 off -> Machine |
                      when i==1 open -> Full |
                      when i==1 start -> Cycle[0]),
Full = (unload -> Empty |
            close -> ClosedDoor[1]),
Empty = (load -> Full | close -> ClosedDoor[0]),
Cycle[i:0..2] = (when i==0 wash -> Cycle[i+1] |
                 when i==1 rinse -> Cycle[i+1] |
                 when i==2 dry -> ClosedDoor[1]|
                 pause[i] -> Paused[i]),
Paused[i:0..2] = (resume[i] -> Cycle[i]).

property CycleProperty = (wash -> rinse -> dry
                          -> CycleProperty).
```

Summary

Fairness and progress are important properties through which we can ensure liveness of a system. It is important to identify these and then design out such subtle issues.