Properties IV & Software Transactional Memory

Lecture 18

Alma Rahat

CS-210: Concurrency

24 March 2021

Swansea University
Prifysgol Abertawe

What did we do in the last session?

- Washing machine
- Progress properties

In this lecture...

Learning outcomes.

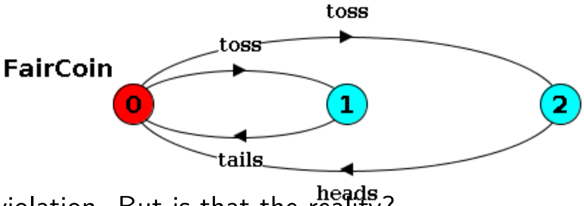
1. To apply modelling techniques to identify progress issues.
2. To devise appropriate ways to eliminate progress issues.
3. To identify the difference between task and data parallelism.
4. To apply data parallelism using the multiverse library for Software Transactional Memory.

Outline.

1. Example: two coins.
2. Fixing Single Lane Bridge.
3. Data parallelism.
4. Examples: bank account and array element swapping.

Was the Coin Really Fair?

```
FairCoin = (toss -> heads -> FairCoin | toss -> tails ->
FairCoin). // a simple coin that produces heads or tails.
progress Heads = {heads} //tests that heads can occur
infinitely many times
progress Tails = {tails} /tests that heads can occur
infinitely many times
progress HeadsOrTails = {heads, tails} //tests that at
least one of heads or tails occur infinitely many times.
```



No progress violation. But is that the reality?

Was the Coin Really Fair?

We will investigate what happens if we add priority to a certain action: the prioritised action gets picked all the time in a choice situation.

- High priority in process P: if the actions come up in a choice, prefer actions $\{a_1, \ldots, a_n\}$ over others.

  ```
  P << {a1, ..., an}
  ```

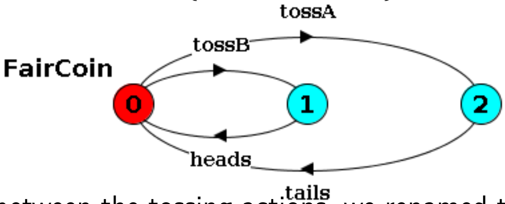
- Low priority in process P: if the actions come up in a choice, prefer other actions over $\{b_1, \ldots, b_n\}$.

  ```
  P >> {b1, ..., bn}
  ```

We will use this to uncover potential progress problems by selecting traces that may be problematic.

Was the Coin Really Fair?

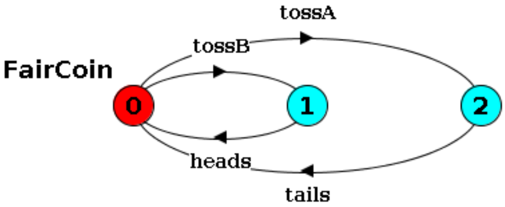
```
FairCoin = (tossA -> heads -> FairCoin | tossB -> tails ->
FairCoin).
progress Heads = {heads}
progress Tails = {tails}
progress HeadsOrTails = {heads, tails}
```



To differentiate between the tossing actions, we renamed them: `tossA` and `tossB`. No progress violations still.

Was the Coin Really Fair?

```
FairCoin = (tossA -> heads -> FairCoin | tossB -> tails ->
FairCoin).
||TestPriority = FairCoin >> {tossA} // prefer tossB over
tossA
progress Heads = {heads}
progress Tails = {tails}
progress HeadsOrTails = {heads, tails}
```

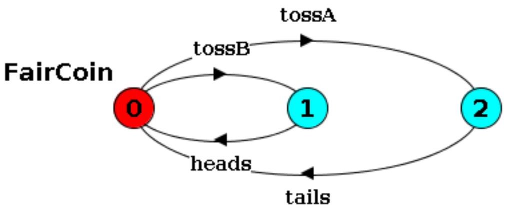


```
Progress violation: Tails
Trace to terminal set of states
Cycle in terminal set:
        tossB
        heads
Actions in terminal set:
        {heads, tossB}
```

We impose priority that `tossB` gets chosen over `tossA`. Now, we end up cycling between states Q_0 and Q_1, violating the `Tails` property.

Was the Coin Really Fair?

```
FairCoin = (tossA -> heads -> FairCoin | tossB -> tails ->
FairCoin).
||TestPriority = FairCoin << {tossA} // prefer tossA over
tossB
progress Heads = {heads}
progress Tails = {tails}
progress HeadsOrTails = {heads, tails}
```



```
Progress violation: Heads
Trace to terminal set of states
Cycle in terminal set:
        tossA
        tails
Actions in terminal set:
        {tails, tossA}
```

We impose priority that `tossA` gets chosen over `tossB`. Now, we end up cycling between states Q_0 and Q_2, violating the `Heads` property.

Any questions?

Liveness on the Single Lane Bridge

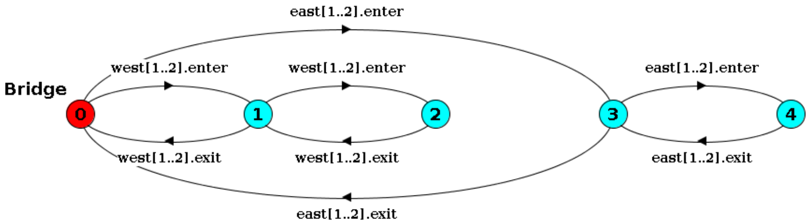
We saw that the potential issue with Single Lane Bridge was that there was a lack of fairness, and it was not picked by the LTSA because of uniform fairness assumption in choices.

Liveness on the Single Lane Bridge

This was the model of the Bridge. Assumption was that cars can enter from either side fairly.

```
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).
```

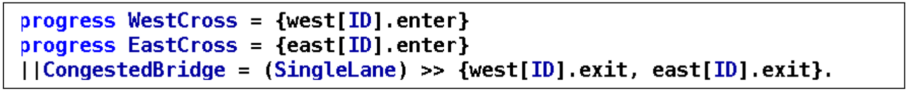


Liveness on the Single Lane Bridge

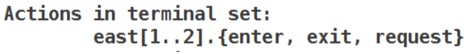
To get rid of unnecessary deadlock warnings, we will first modify the Car process.

```
Car = (enter -> exit -> Car).   previously we had:  Car
= (enter -> exit -> STOP).
```

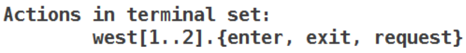
We will now define a couple of progress properties to ensure that both sides gets a go.



```
progress WestCross = {west[ID].enter}
progress EastCross = {east[ID].enter}
||CongestedBridge = (SingleLane) >> {west[ID].exit, east[ID].exit}.
```



```
Actions in terminal set:
        east[1..2].{enter, exit, request}
```

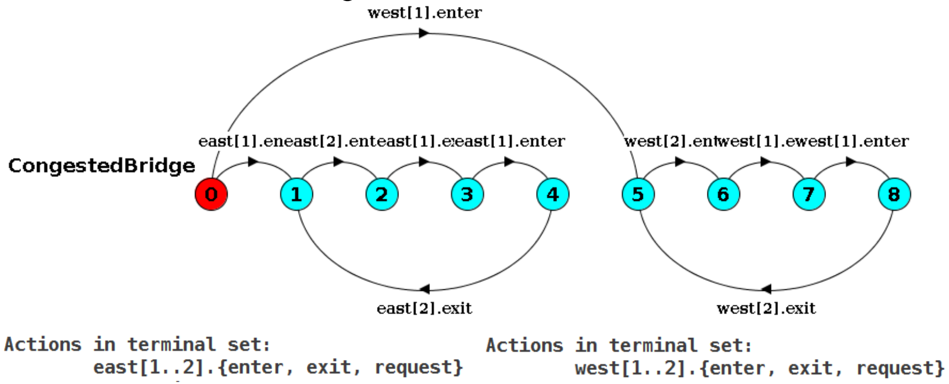


```
Actions in terminal set:
        west[1..2].{enter, exit, request}
```

Both progress properties are violated, and we see that for WestCross violation, in the terminal set only has actions from east[ID], and *vice versa*.

Liveness on the Single Lane Bridge

Here are the terminal sets again.



Due to the priority in enter action, once we perform one of `east[ID].enter` or `west[ID].enter` we get stuck in the respective terminal set.

Liveness on the Single Lane Bridge

Please go to `www.menti.com` and use the code 6000 7279.

How can we fix this? What strategy can we deploy?

Liveness on the Single Lane Bridge

Please go to `www.menti.com` and use the code 6000 7279.

How can we fix this? What strategy can we deploy?

- Politeness: Do not enter if there is a car waiting at the other side.
- Strict ordering: let a car pass from one side at a time.

Liveness on the Single Lane Bridge

Please go to www.menti.com and use the code 6000 7279.

How can we fix this? What strategy can we deploy?

- Politeness: Do not enter if there is a car waiting at the other side.
- Strict ordering: let a car pass from one side at a time.

Can we just use one of these strategies?

Liveness on the Single Lane Bridge

Please go to www.menti.com and use the code 6000 7279.

How can we fix this? What strategy can we deploy?

- Politeness: Do not enter if there is a car waiting at the other side.
- Strict ordering: let a car pass from one side at a time.

Can we just use one of these strategies? No.

- Politeness: The system may deadlock if two cars are waiting simultaneously.
- Strict ordering: We let one car pass and then wait for a car to come from the other side, we may be waiting indefinitely for a car to appear on the other side.

Liveness on the Single Lane Bridge

Politeness.

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

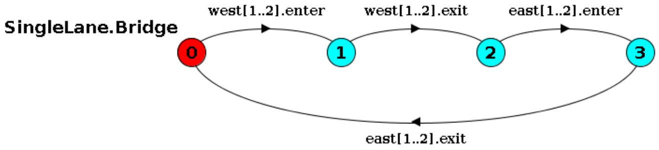
```
Trace to terminal set of states:
        west.1.request
        west.2.request
        east.1.request
        east.2.request
```

All are polite and waiting for others to pass through the bridge. Deadlocked!

Liveness on the Single Lane Bridge

Strict ordering.

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



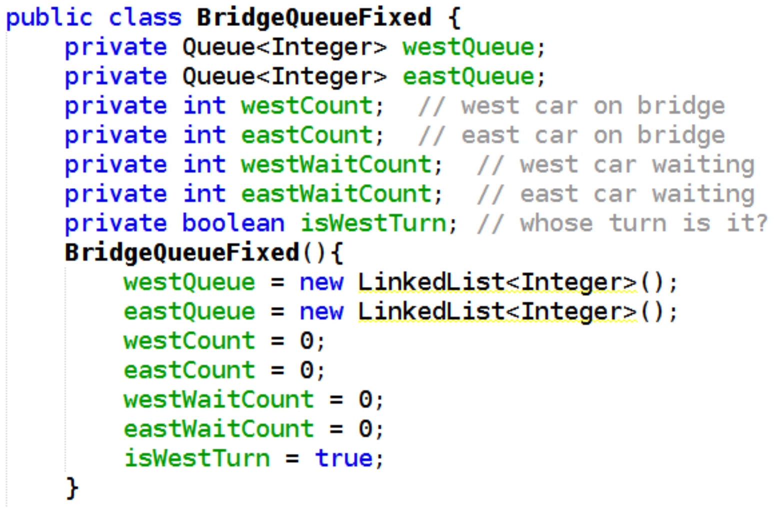
The progress properties we defined passes, but we see from the LTS that we cannot progress if `east[ID]` does not come.

Liveness on the Single Lane Bridge

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

We combine both *politeness* and *strict ordering*. Basically only follow strict ordering if there are cars waiting on the other side.

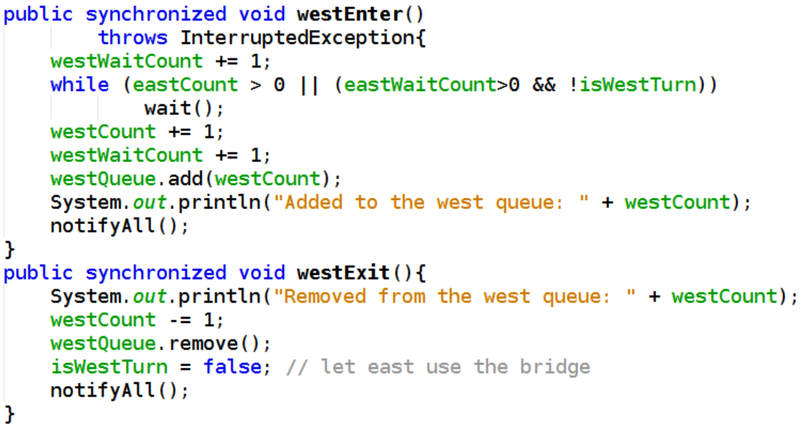
Fixed Code



```java
public class BridgeQueueFixed {
    private Queue<Integer> westQueue;
    private Queue<Integer> eastQueue;
    private int westCount;  // west car on bridge
    private int eastCount;  // east car on bridge
    private int westWaitCount;  // west car waiting
    private int eastWaitCount;  // east car waiting
    private boolean isWestTurn; // whose turn is it?
    BridgeQueueFixed(){
        westQueue = new LinkedList<Integer>();
        eastQueue = new LinkedList<Integer>();
        westCount = 0;
        eastCount = 0;
        westWaitCount = 0;
        eastWaitCount = 0;
        isWestTurn = true;
    }
```

We define additional variables to capture different state information.

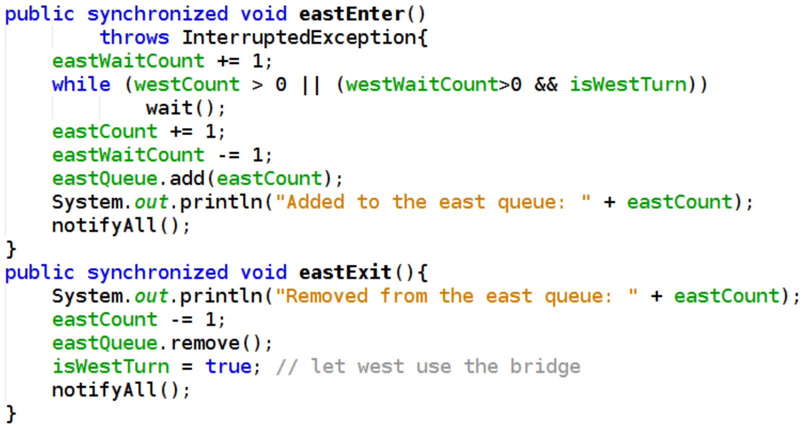
Fixed Code



```
public synchronized void westEnter()
        throws InterruptedException{
    westWaitCount += 1;
    while (eastCount > 0 || (eastWaitCount>0 && !isWestTurn))
            wait();
    westCount += 1;
    westWaitCount += 1;
    westQueue.add(westCount);
    System.out.println("Added to the west queue: " + westCount);
    notifyAll();
}
public synchronized void westExit(){
    System.out.println("Removed from the west queue: " + westCount);
    westCount -= 1;
    westQueue.remove();
    isWestTurn = false; // let east use the bridge
    notifyAll();
}
```

We modify the waiting logic, and increment and decrement various counts according to our model.

Fixed Code



```java
public synchronized void eastEnter()
        throws InterruptedException{
    eastWaitCount += 1;
    while (westCount > 0 || (westWaitCount>0 && isWestTurn))
            wait();
    eastCount += 1;
    eastWaitCount -= 1;
    eastQueue.add(eastCount);
    System.out.println("Added to the east queue: " + eastCount);
    notifyAll();
}
public synchronized void eastExit(){
    System.out.println("Removed from the east queue: " + eastCount);
    eastCount -= 1;
    eastQueue.remove();
    isWestTurn = true; // let west use the bridge
    notifyAll();
}
```

Methods for controlling the east entrance is similar to the west entrance.

Reader-Writers Problem

The single lane bridge is similar to Readers-Writers problem. The context:

- Database access and update.
- Several readers and writer process.
- Simultaneous access where possible: multiple readers, but only one writer.

Challenges.

- Avoid interference and deadlock.
- Ensure fairness and progress.

Here we can analyse the system and come up with a good solution. There are other alternatives, for instance, Software Transactional Memory (STM). This will be our next topic.

Any questions?

Approaches for Implementing Concurrency

- Task parallelism. This is what we have been thinking about thus far: how to organise multiple threads and their communication without breaking concurrent correctness using **lock** based (potentially conditional) **synchronisation** mechanisms.
- Data parallelism. A different way to think about concurrency: shared data is protected by accessing it in a sequential manner.

Software Transactional Memory (STM) is a pattern for simplifying concurrent programming and is an option to achieve *data parallelism*, inspired from work in databases.

Issues with Task Parallelism

Complexity. It is not always straightforward to get it right.

Composibility. Difficult to build larger program from smaller components without careful alterations.

Forget the lock. Interference may occur.

Too many locks. Deadlock may occur.

Forget to signal. Waiting indefinitely.

Restoration. If things go wrong, it is not easy to restore to an earlier point.

Testing. Difficult to test due to non-determinism.

Databases usually handle concurrency in accessing shared data rather well in the form of *transaction* handling!

Database Transactions

A database transaction symbolises a unit of work performed within the management system against a database, and treated in a coherent and reliable way independent of other transactions.

Two primary purposes.

1. Reliable units of work: allow correct recovery from failures and keep a database consistent.
2. Provide isolation: between programs accessing (potentially concurrently) the database.

Properties of Database Transactions

ACID: An acronym for a set of properties of database transactions intended to guarantee validity even in the event of failures. In this context, a set of multiple actions are considered as *logically atomic* and thus they are free from concurrency issues.

A tomicity. All changes of data are performed as if they are a single operation: all are performed or none are.

C onsistency. Data is in a consistent state when a transaction starts and when it ends.

I solation. The intermediate state of a transaction is invisible to other transactions.

D urability. After a transaction is completed successfully, changes to data persist and are not undone, even if the system fails.

Software Transactional Memory (STM)

- A pattern for achieving data parallelism.
- An extension or library for many standard programming languages, including Java.
- A developer need not worry about concurrency.
- The primary mechanism is accessing memory through a level of indirection.

Steps to follow.

Step I. Identify data in memory, and mark transactions as part of a transactional log.

Step I. Make a local copy.

Step I. Work on the local copy.

Step I. Commit changes back to memory atomically.

On conflict. Throw away all local operations, and try again.

On success. Mark transactions complete. You copy becomes the new master copy.

Limitations of STM

It is not all plain sailing; if you can design out the concurrent issues that may be preferable depending on the context. In fact, industry still uses the version of concurrency we have looked at so far.

- Not all operations are allowed, e.g. sending or receiving over networks.
- Potentially run expensive operations many times in case of conflicts as our commits fail and we have to start over.
- Difficult to deal with large chunks of memory.
- Still have to worry about priority and contention.

You should decide which approach to take.

Multiverse Library

We will have a look at the following library in Java:

`https://github.com/pveentjer/Multiverse`

You can get the *JAR* files from the following repository:

`https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.multiverse%22%20AND%20a%3A%22multiverse-core%22`

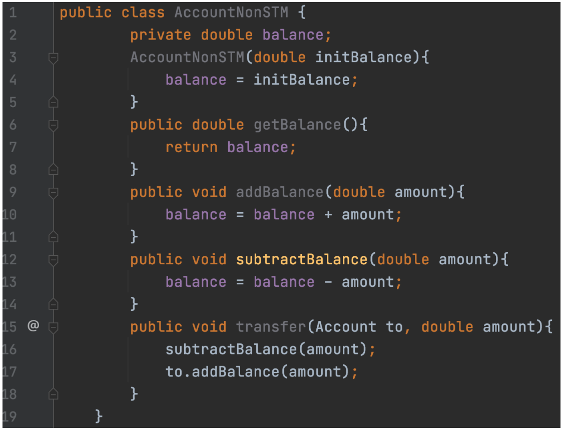
- Latest version is: 0.7.0 (2012).
- Note that ACI are preserved in this, but not D as it may not be connected to a database.

It is available in our git repository:
`https://github.com/AlmaRahat/CS-210-Concurrency/tree/main/java-code/multiverse`

Steps to follow.

1. Import the libraries.
2. Use TxnObject and associated method calls.
3. Wrap the sequence of actions that are needed to be atomicised.

Example: Simple Bank Account



```
public class AccountNonSTM {
        private double balance;
        AccountNonSTM(double initBalance){
            balance = initBalance;
        }
        public double getBalance(){
            return balance;
        }
        public void addBalance(double amount){
            balance = balance + amount;
        }
        public void subtractBalance(double amount){
            balance = balance - amount;
        }
@       public void transfer(Account to, double amount){
            subtractBalance(amount);
            to.addBalance(amount);
        }
    }
```

Here is an example of non-concurrent account. In task parallelism, we will have to hold one account and wait for the other to become available before a transfer can happen. We will see how to do data parallelism in this context.

Example: Simple Bank Account

Live demonstration: code is on github.

Example: Array Swapping

Scenario.

We have a simple `BinaryArray` class, and we want to create a program where it is possible to concurrently swap elements between *two* instances of `BinaryArray`.

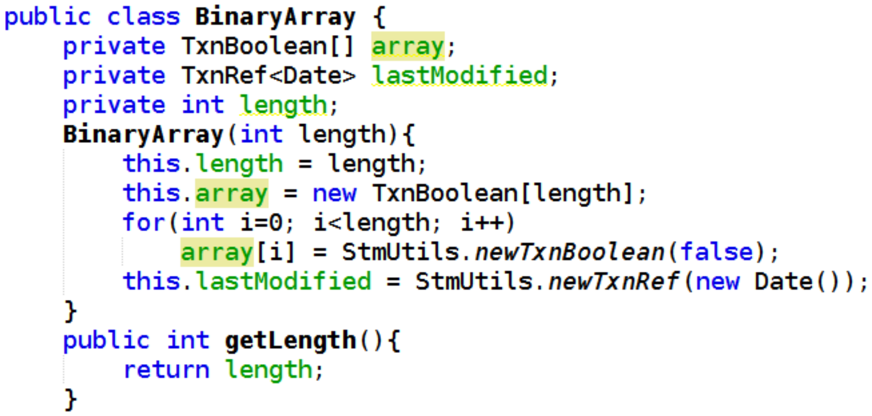
In task parallelism, we would have to think about locking first instance and then second instance for this, which may lead to deadlock. So, we will just make sure that if a `Thread` instance cannot get both, then simply let's go of the first array.

Example: Array Swapping

We will first import the library.

```
import org.multiverse.api.StmUtils;
import org.multiverse.api.references.*;
```

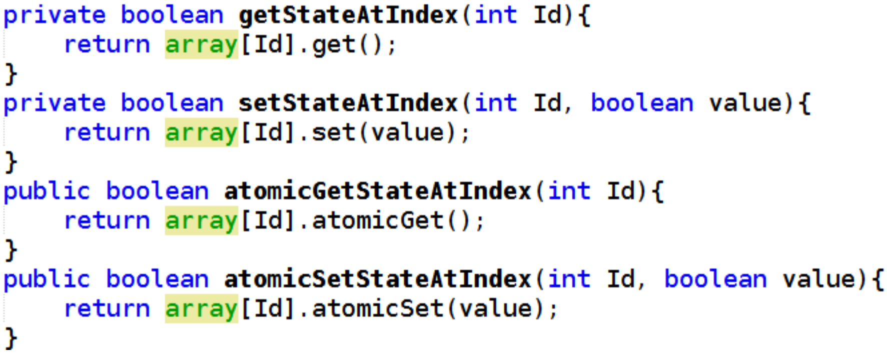
Example: Array Swapping



```java
public class BinaryArray {
    private TxnBoolean[] array;
    private TxnRef<Date> lastModified;
    private int length;
    BinaryArray(int length){
        this.length = length;
        this.array = new TxnBoolean[length];
        for(int i=0; i<length; i++)
            array[i] = StmUtils.newTxnBoolean(false);
        this.lastModified = StmUtils.newTxnRef(new Date());
    }
    public int getLength(){
        return length;
    }
```

- We need a `TxnObject` for this purpose; there are a range of possibilities available in the library.
- To create new instances, we would use a method from `StmUtils`.

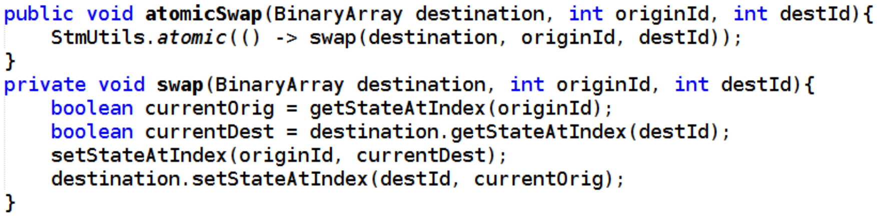
Example: Array Swapping



```
private boolean getStateAtIndex(int Id){
    return array[Id].get();
}
private boolean setStateAtIndex(int Id, boolean value){
    return array[Id].set(value);
}
public boolean atomicGetStateAtIndex(int Id){
    return array[Id].atomicGet();
}
public boolean atomicSetStateAtIndex(int Id, boolean value){
    return array[Id].atomicSet(value);
}
```

- Internally we can use standard `get` and `set` methods from within the class instance.
- Externally this throws errors, so we will only use the `atomic` versions of these.

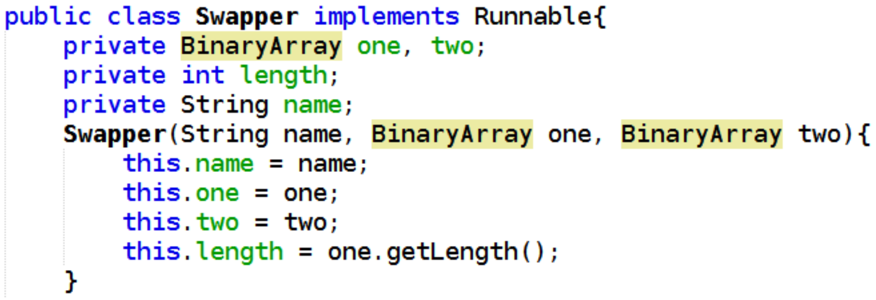
Example: Array Swapping



```
public void atomicSwap(BinaryArray destination, int originId, int destId){
    StmUtils.atomic(() -> swap(destination, originId, destId));
}
private void swap(BinaryArray destination, int originId, int destId){
    boolean currentOrig = getStateAtIndex(originId);
    boolean currentDest = destination.getStateAtIndex(destId);
    setStateAtIndex(originId, currentDest);
    destination.setStateAtIndex(destId, currentOrig);
}
```

- `atomicSwap` is the method that will be called by `Thread` instances.
- The structure we will follow is: `StmUtils.atomic(() -> function(parameters))`.
- `swap` method just contains the set of steps that we need to perform atomically.

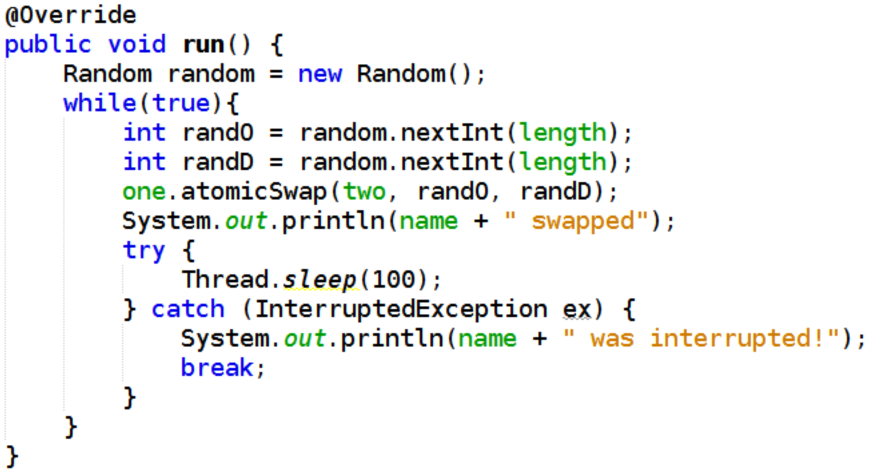
Example: Array Swapping



```
public class Swapper implements Runnable{
    private BinaryArray one, two;
    private int length;
    private String name;
    Swapper(String name, BinaryArray one, BinaryArray two){
        this.name = name;
        this.one = one;
        this.two = two;
        this.length = one.getLength();
    }
```

- The `Swapper` models our threads that will independently swap things around.
- It has a couple of `BinaryArrays`, on which it performs the swap operations.

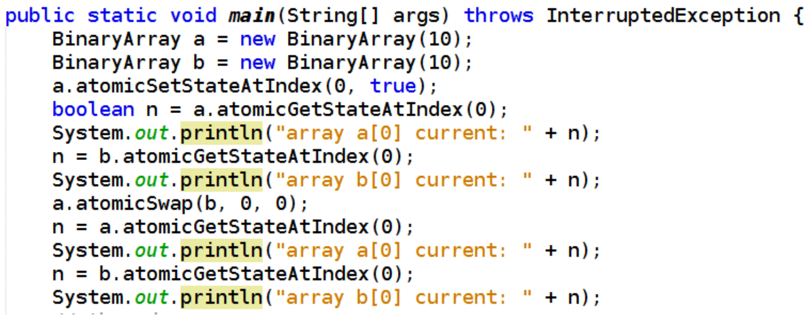
Example: Array Swapping



```
@Override
public void run() {
    Random random = new Random();
    while(true){
        int randO = random.nextInt(length);
        int randD = random.nextInt(length);
        one.atomicSwap(two, randO, randD);
        System.out.println(name + " swapped");
        try {
            Thread.sleep(100);
        } catch (InterruptedException ex) {
            System.out.println(name + " was interrupted!");
            break;
        }
    }
}
```

- This is the `run` method in the `Swapper`: arbitrarily swaps values between two arrays.

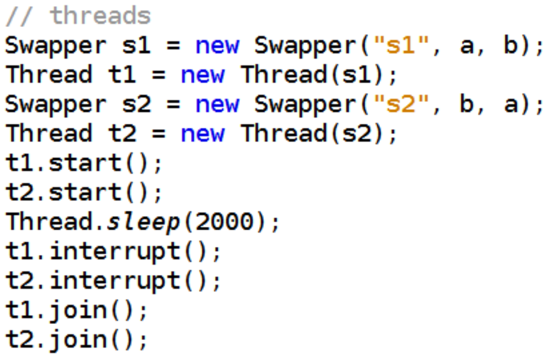
Example: Array Swapping



```
public static void main(String[] args) throws InterruptedException {
    BinaryArray a = new BinaryArray(10);
    BinaryArray b = new BinaryArray(10);
    a.atomicSetStateAtIndex(0, true);
    boolean n = a.atomicGetStateAtIndex(0);
    System.out.println("array a[0] current: " + n);
    n = b.atomicGetStateAtIndex(0);
    System.out.println("array b[0] current: " + n);
    a.atomicSwap(b, 0, 0);
    n = a.atomicGetStateAtIndex(0);
    System.out.println("array a[0] current: " + n);
    n = b.atomicGetStateAtIndex(0);
    System.out.println("array b[0] current: " + n);
```

- Firstly, we show standard calls to the methods in `BinaryArray` in the `main` method.

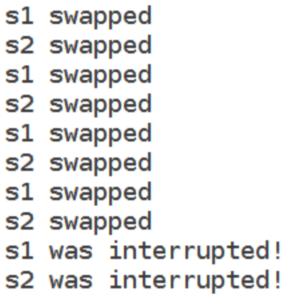
Example: Array Swapping



```java
// threads
Swapper s1 = new Swapper("s1", a, b);
Thread t1 = new Thread(s1);
Swapper s2 = new Swapper("s2", b, a);
Thread t2 = new Thread(s2);
t1.start();
t2.start();
Thread.sleep(2000);
t1.interrupt();
t2.interrupt();
t1.join();
t2.join();
```

- We generate a couple of independent threads, and let them perform swapping on the arrays.

Example: Array Swapping



```
s1 swapped
s2 swapped
s1 swapped
s2 swapped
s1 swapped
s2 swapped
s1 swapped
s2 swapped
s1 was interrupted!
s2 was interrupted!
```

- Example output; it just works out of the box.

Any questions?

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.