Concurrent Programming (Part II)
Lecture 8: Liveness Property

Dr Kevin Bryson
K.Bryson@cs.ucl.ac.uk
Room 8.04

Course Web Site on Moodle

http://moodle.ucl.ac.uk/course/view.php?id=753

Enrolment Key: ATOMIC

Aim of Lecture

- Recall the two main classes of **properties** for concurrent programs:
  - **Safety Properties**: assert that nothing ‘bad’ will ever happen during any execution (the program will never enter a ‘bad’ state)
  - **Liveness Properties**: assert that something ‘good’ will eventually happen during every execution
- In this lecture we will introduce **liveness properties**. As mentioned before, **deadlock** is a safety property but can also be viewed as a permanent failure of **liveness**.

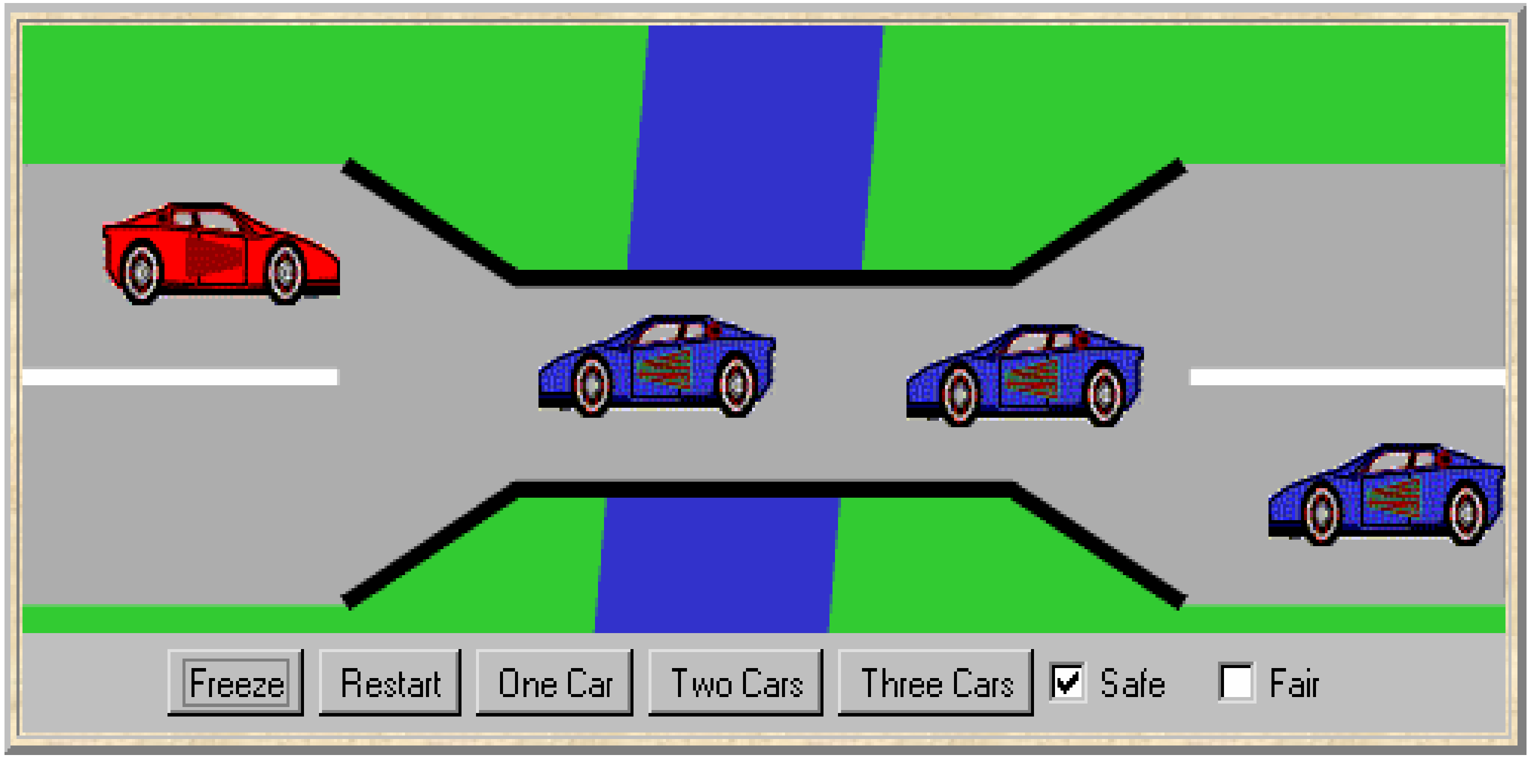
So what is liveness?

- **Question:** If shared objects are unsafe when they are not synchronized (due to *interference*) – then why not just synchronize everything in the system?
- Well such an approach is likely to result in systems which are not particularly *lively.* If they do not rapidly deadlock, threads are likely to be forever waiting to obtain different object locks.
- *Excessive* locking to guarantee safety needs to be balanced with **liveness** concerns …
- **Definition:** In ***live*** systems, every activity progresses towards completion, every method invoked eventually executes … no threads 'starve' of the CPU resource.

Why might an activity fail to progress?

- An activity/thread may fail to make progress because:
  - **Locking**: A synchronized method blocks one thread because another holds the lock.
  - **Waiting**: A method blocks, e.g. by using Object.wait(), waiting for an event, message or condition yet to be produced by another thread.
  - **Input**: An IO-based method waits for input.
  - **CPU contention**: A thread waits to be scheduled on the CPU.
- We will illustrate problems with liveness using a Single Lane Bridge Simulator.

Single Lane Bridge Simulator (Magee/Kramer Ch. 7)

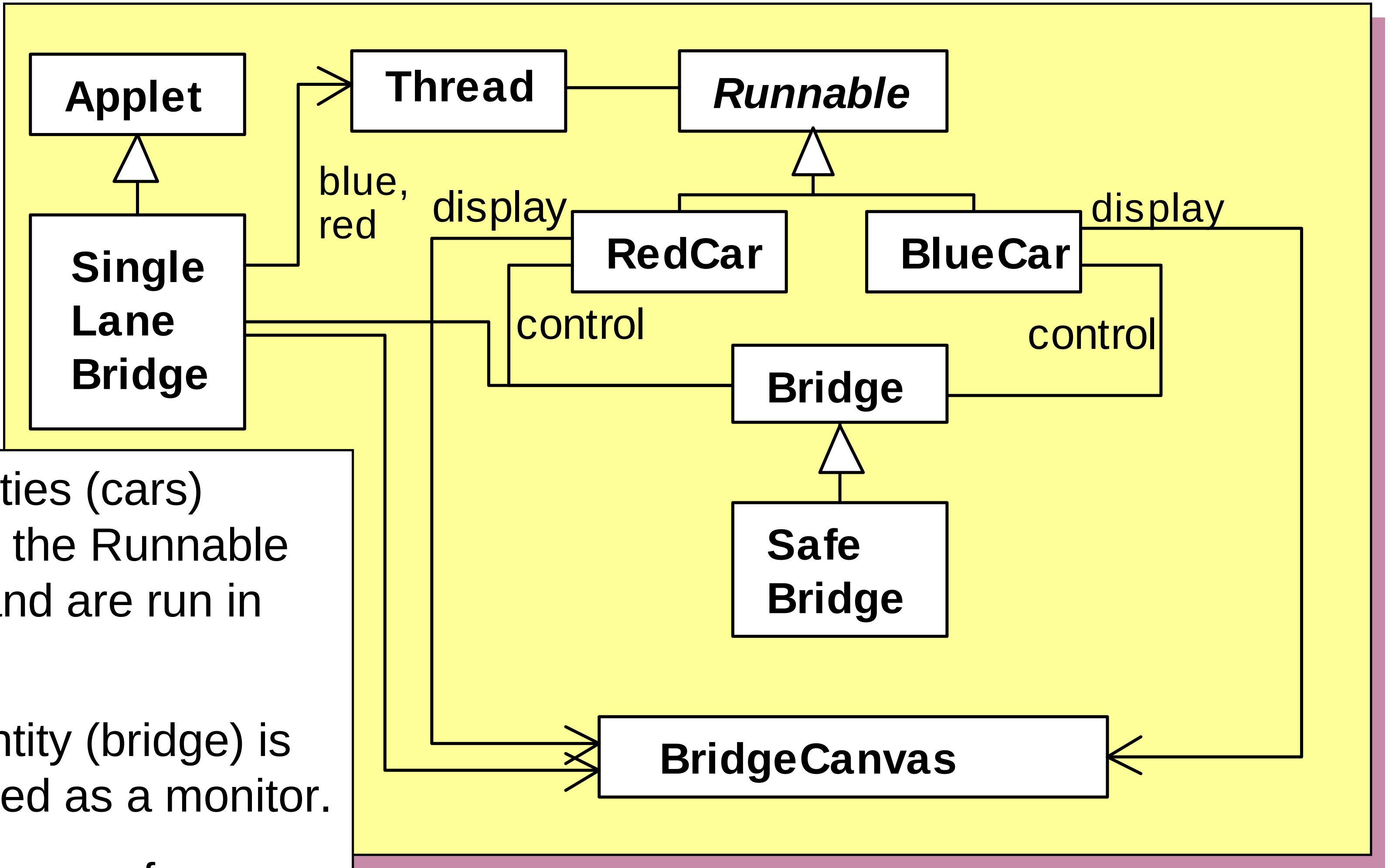


A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time. What does the demo look like …

Single Lane Bridge - How to implement …

- The **active entities** are the cars – these should be implemented either as subclasses of Threads or implement the Runnable interface.
- The **shared passive entity** which the cars interact with is the bridge – this should be implemented as a **monitor**. Thus it should be an object with its state encapsulated so that only one thread can change its state at a time. That is all methods which alter or query its state should be **synchronized**.

Single Lane Bridge - implementation in Java



Active entities (cars) implement the Runnable interface and are run in Threads.

Passive entity (bridge) is implemented as a monitor.

BridgeCanvas enforces no overtaking.

Single Lane Bridge – `BridgeCanvas Class`

An instance of **BridgeCanvas** class is created by SingleLaneBridge – reference is passed to each newly created RedCar and BlueCar object.

```
class BridgeCanvas extends Canvas {

  public void init(int ncars) {…}    // set number of cars

  // move red car with the identity i a step
  // returns true for the period from just before,until just after car on bridge
  public boolean moveRed(int i) throws
InterruptedException{…}

  // move blue car with the identity i a step
  // returns true for the period from just before,until just after car on bridge
  public boolean moveBlue(int i) throws
InterruptedException{…}

  public synchronized void freeze(){…}//  freeze display
  public synchronized void thaw(){…}   // unfreeze display
}
```

BridgeCanvas Class

- ***Look through the original code and try to work out how the overall system is working.***

- The BridgeCanvas class is initialized using the method *init(int ncars)*. This generates ‘ncars’ that are blue and ‘ncars’ that are red. It also positions the cars on the bridge, in terms of (x,y) coordinates.
- This class also has methods *moveRed(int i)* and *moveBlue(int i)*. These move the i’th red or blue car by 2 pixels over the bridge. It does not move a car if it would be too close to the car in front. It also returns the current status of this particular car (on the 1-lane bridge or not).
- THUS this class is *responsible* for keeping track of the (x,y) positions of all the cars and drawing the bridge (**but it is *not responsible for moving the cars***).

Single Lane Bridge – RedCar Class

```
class RedCar implements Runnable {

  BridgeCanvas display; Bridge control; int id;

  RedCar(Bridge b, BridgeCanvas d, int id) {
    display = d; this.id = id; control = b;
  }

  public void run() {
    try {
      while(true) {
        while (!display.moveRed(id));    // not on bridge
        control.redEnter();          // request access to bridge
        while (display.moveRed(id)); // move over bridge
        control.redExit();           // release access to bridge
      }
    } catch (InterruptedException e) {}
  }
}
```

Similarly for the **BlueCar**

RedCar and BlueCar Classes

- ***Look through the original code and try to work out how the overall system is working.***

- These classes are *responsible* for the active behaviour of the cars.
- Each car is modelled using a separate thread (so modelling 3 blue red cars and 3 blue cars involves the creation of 6 Threads).
- Each car tries to move forward until it gets onto the start of the bridge (whereupon the *moveRed/moveBlue* methods return true).
- It has to ask the Bridge controller for permission to actually move onto the bridge:
  - It then just moves forward until it comes to the other side of the bridge (i.e. *moveRed* or *moveBlue* methods returns false).
  - It then tells the bridge controller that it wishes to exit the bridge.

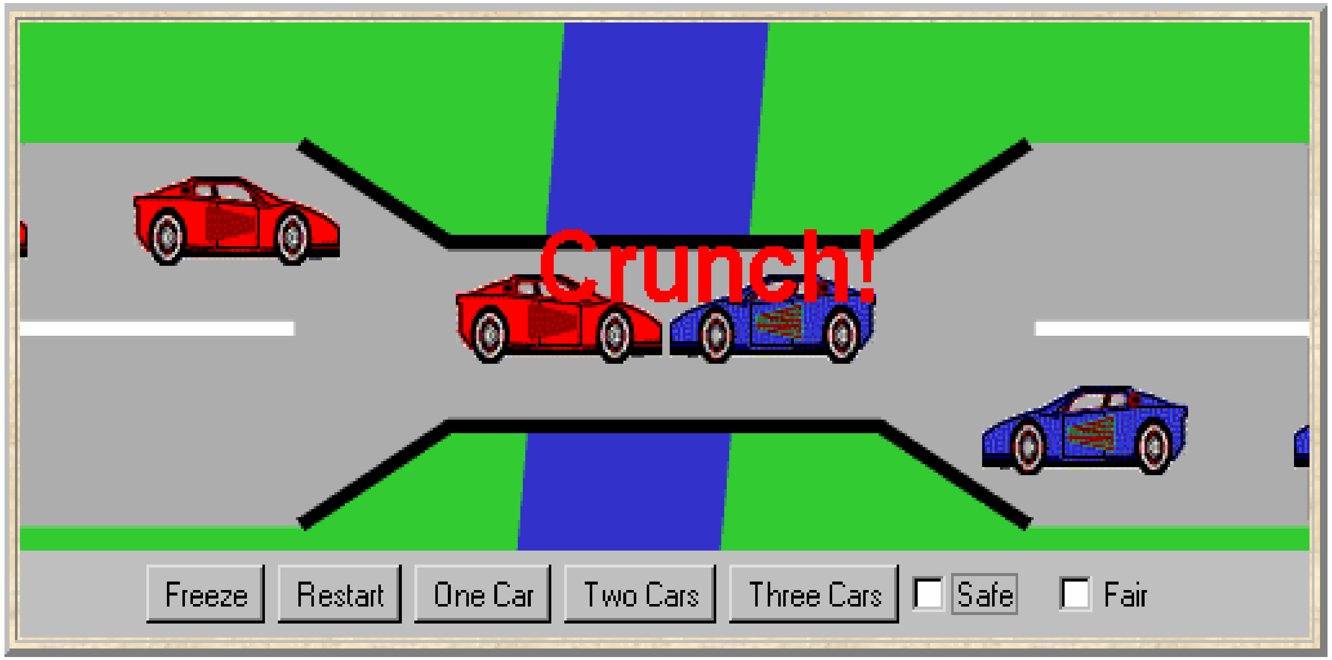
Single Lane Bridge - class Bridge

```
class Bridge {
  synchronized void redEnter()
     throws InterruptedException {}
  synchronized void redExit()  {}
  synchronized void blueEnter()
     throws InterruptedException {}
  synchronized void blueExit() {}
}
```

Class **Bridge** provides a null implementation of the access methods i.e. no constraints on the access to the bridge.

Discuss – what will be the result of this ?

Single Lane Bridge



This is similar to the box junction example in lecture 7, with Gridlock/Deadlock resulting …

How can we stop this happening ?

Discuss suitable constraints for the bridge.

Single Lane Bridge – SafeBridge

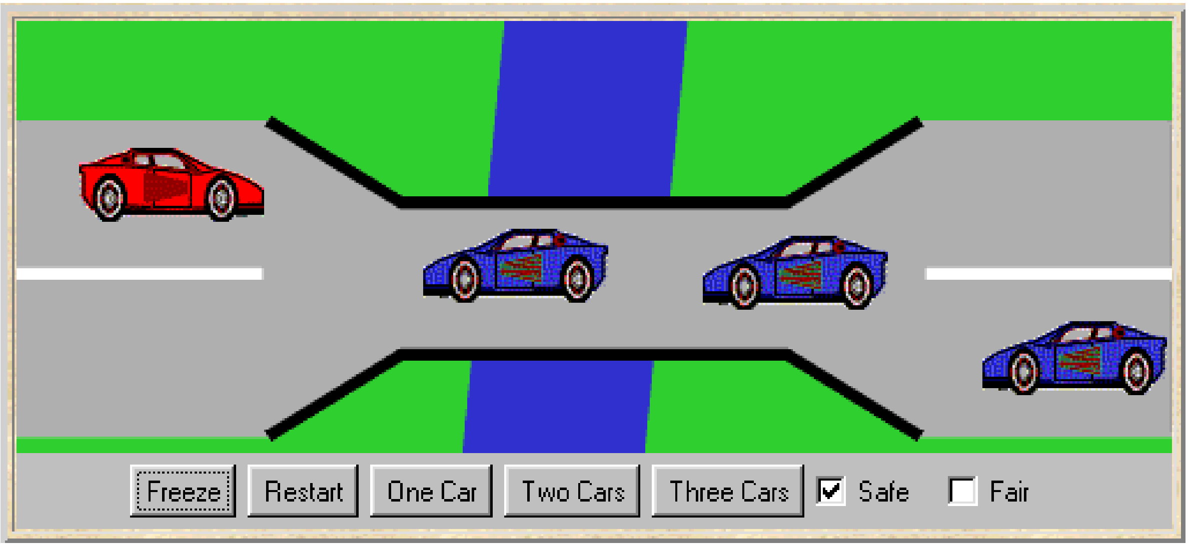
How does this stop the cars crashing?

To avoid unnecessary thread switches, we use ***conditional notification*** to wake up waiting threads when last car exits bridge …

```
class SafeBridge extends Bridge {

  private int nred  = 0; //number of red cars on bridge
  private int nblue = 0; //number of blue cars on bridge

  // Monitor Invariant:       nred≥0 and nblue≥0 and
  //                       not (nred>0 and nblue>0)

  synchronized void redEnter()
      throws InterruptedException {
    while (nblue>0) wait();
    ++nred;
  }
  synchronized void redExit(){
     --nred;
      if (nred==0)notifyAll();
  }
  synchronized void blueEnter()
      throws InterruptedException {
    while (nred>0) wait();
    ++nblue;
  }
  synchronized void blueExit(){
    --nblue;
    if (nblue==0)notifyAll();
  }
}
```

Liveness - single lane bridge

But does every car **eventually** get an opportunity to cross the bridge? This is a **liveness** property.



Liveness - single lane bridge

It is clear that when there are more than 3 cars in the simulator – either the red or blue car threads do not **progress** (depending on which type of car gets to the bridge last).
These threads are suffering from **starvation.** This is a **liveness** failure, quite often depending on both structure of the program and scheduling policies.

Fairness means that eventually every possible execution occurs, including those in which cars do not starve.

Discuss suitable constraints for this bridge to stop this starvation.

Real-world Fairness Mechanisms for Single-Lane Bridge

Different Mechanisms:

- Time-slicing of around 30 seconds each way.
- Detectors for cars that are waiting at red.

Single lane bridge - FairBridge

```
class FairBridge extends Bridge {

  private int nred  = 0; //count of red cars on the bridge
  private int nblue = 0; //count of blue cars on the bridge
  private int waitblue = 0;   //count of waiting blue cars
  private int waitred = 0;    //count of waiting red cars
  private boolean blueturn = true;

  synchronized void redEnter()
      throws InterruptedException {
    ++waitred;
    while (nblue>0||(waitblue>0 && blueturn)) wait();
    --waitred;
    ++nred;
  }

  synchronized void redExit(){
    --nred;
    blueturn = true;
    if (nred==0)notifyAll();
  }                              ...CONTINUED ...
```

Single lane bridge - FairBridge

```
   synchronized void blueEnter(){
       throws InterruptedException {
     ++waitblue;
     while (nred>0||(waitred>0 && !blueturn)) wait();
     --waitblue;
     ++nblue;
   }

   synchronized void blueExit(){
     --nblue;
     blueturn = false;
     if (nblue==0) notifyAll();
   }
}
```

The main change is that the monitor now keeps track of number of red and blue cars waiting, as well as the number actually on the bridge.

Summary

- While a program must remain *safe*, **excessive** locking (i.e. **excessively** large *lock scopes*) often impacts on the *liveness* of a program.
- We examined reasons why activities/threads may fail to progress.
- Safety - Adding conditional synchronization to make a safe bridge without deadlock.
- Liveness - Adding conditional synchronization to make a fair bridge without liveness problems.