Concurrent Programming (Part II)
Lecture 7: Safety, Liveness & Deadlock

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

- I want to draw together ideas from previous lectures
- Recall that one way of designing concurrent systems is by modelling the system as two types of entities:
  - **Active entities** implemented as **Threads**
  - **Passive entities** implemented as **Monitors** (employing synchronization and conditional synchronization)
- One of our overall aims was to design ’correct’ concurrent programs.
- This Lecture (and the next one) will explain more formally what we actually mean by a ‘correct’ concurrent program.

- *What do we mean by a correct **sequential program**?*

Definition of Concurrent Program Correctness

- A ‘correct’ concurrent program must satisfy certain **properties** – assertions that are true **for every legal execution of the program**
- Correctness takes on a boolean value – either a program is correct or it isn’t – there is no (formal) concept of an ‘almost correct’ concurrent program.
- If the program fails to meet a property only once in 10^40 possible runs … then **formally** it is not correct (informally it may need to be run for the age of the Universe to see the bug … or you might be lucky/unlucky that it shows up after just 5 minutes … !)
- This really means that concurrency correctness needs to be **designed into the system before implementation** (rather than testing incorrectness out of the system after implementation).

What types of Properties ?

- Two main classes of **properties** exist 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 examine aspects of **Safety Properties**, the next lecture will examine aspects of **Liveness Properties**.

Safety Properties – a program should never go into a ‘bad’ state

- One example of the safety property being violated was people ‘disappearing’ in the ornamental garden.
  - This ‘bad’ state involves incorrect updating of attributes by multiple Threads. One cure involves identifying **critical sections of the code** & **applying appropriate synchronization**.
- **Safety invariants** can be used to check/prove objects do not go into ‘bad’ states. For instance, in this case, the total number of people equals sum of turnstiles.

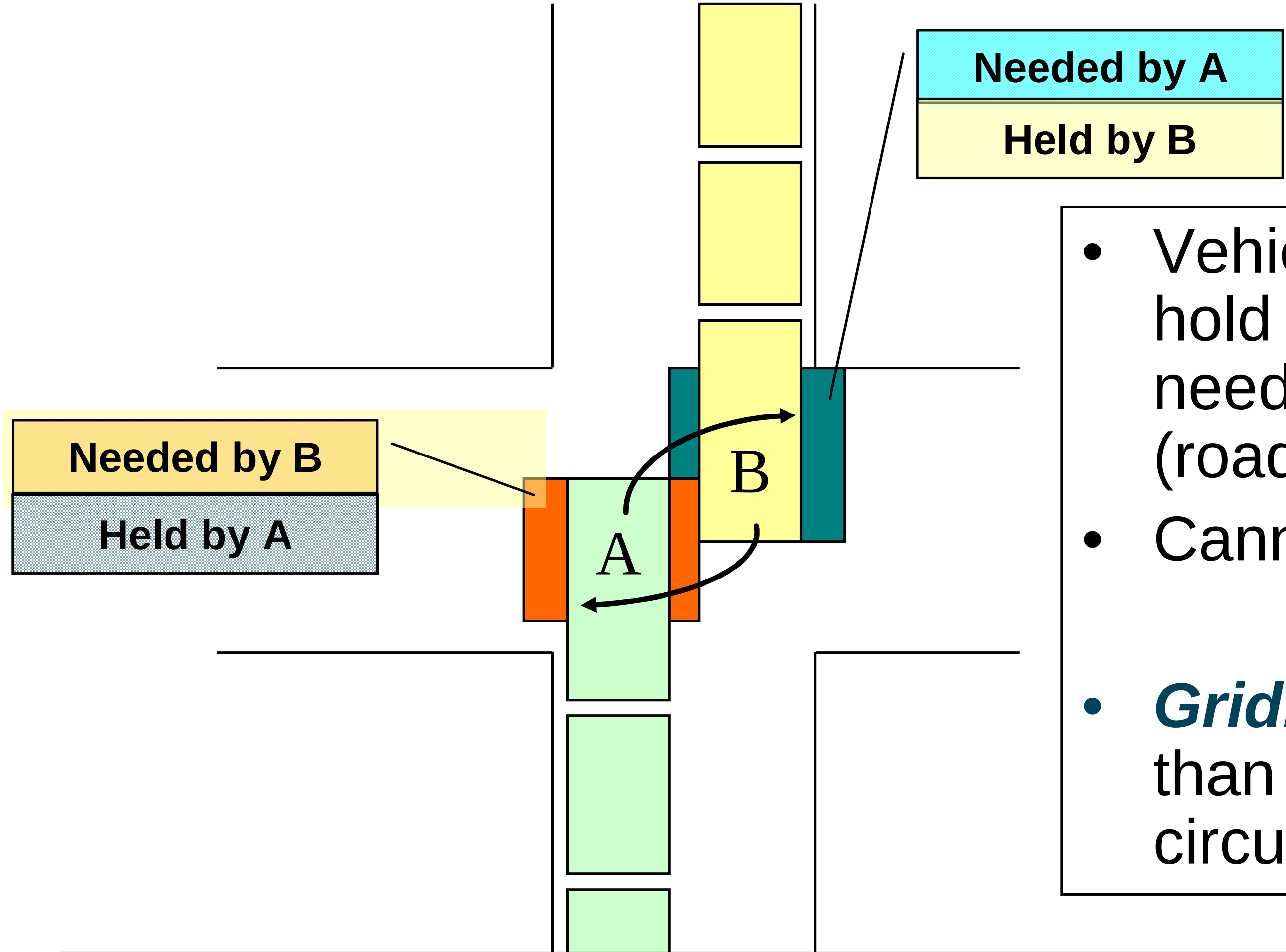
Safety Properties – a program should never go into a ‘bad’ state

- Another example would be ‘writing over’ characters still held at the ‘start' of a bounded cyclic buffer (producer/consumer example)
  - This ‘bad’ state involves monitors applying methods when they are not in an appropriate state (for instance, putting a character into the buffer when it is already full). Its cure is the **appropriate use of conditional synchronization**.
- A **safety invariant,** in this case, is the total number of characters held by the buffer is between 0 and MAX.

Safety Properties – a program should never go into a ‘bad’ state

- Another type of safety property is **deadlock** (although this could also be viewed in some sense as a liveness property)
- We just touched upon deadlock in the Semaphore version of the producer/consumer example (bounded buffer program) – resulting from nested monitors.
- We will now describe this concurrency topic more formally …

Traffic Example – Real Life Deadlock (Gridlock)



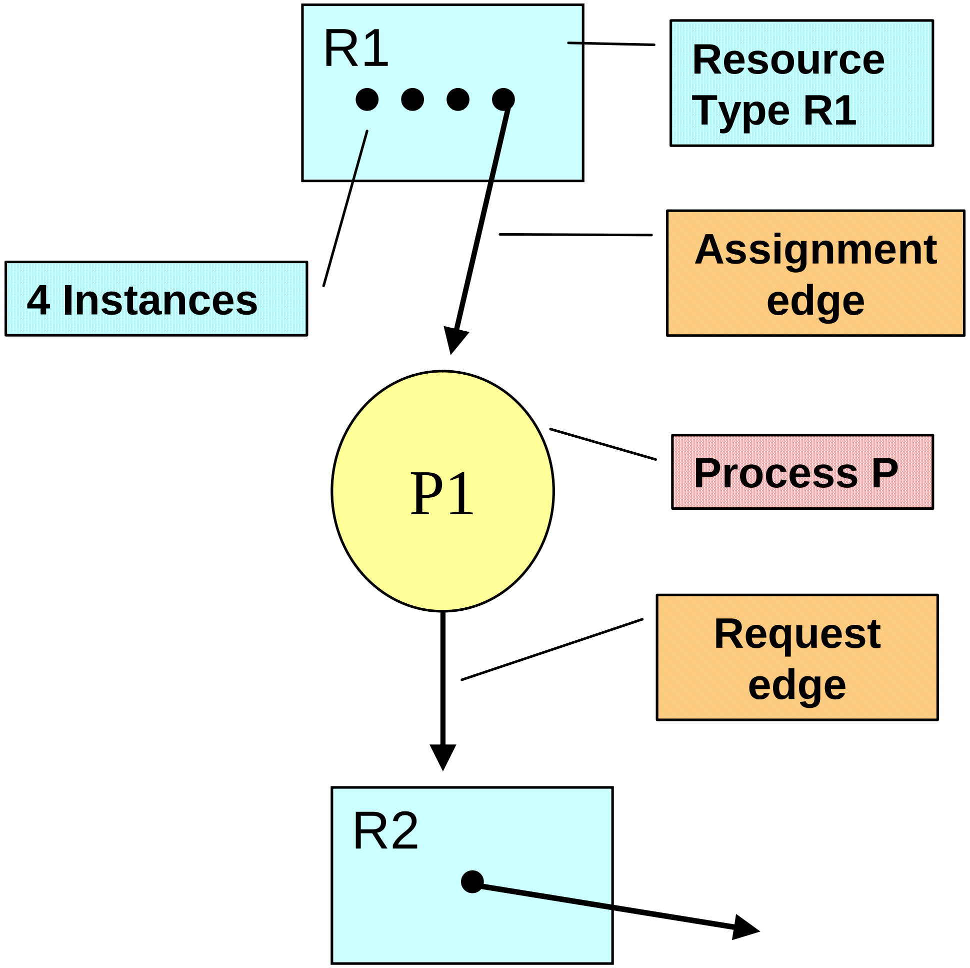
- Vehicles A & B each hold resources needed by the other (road space)
- Cannot back out …
- ***Gridlock*** where more than two vehicles in circular dependency.

If we were to simulate this situation in Java – how is this deadlock different from the previous example of 'nested monitor' deadlock ?

Traffic Example – Real Life Deadlock (Gridlock) … some modelling analysis

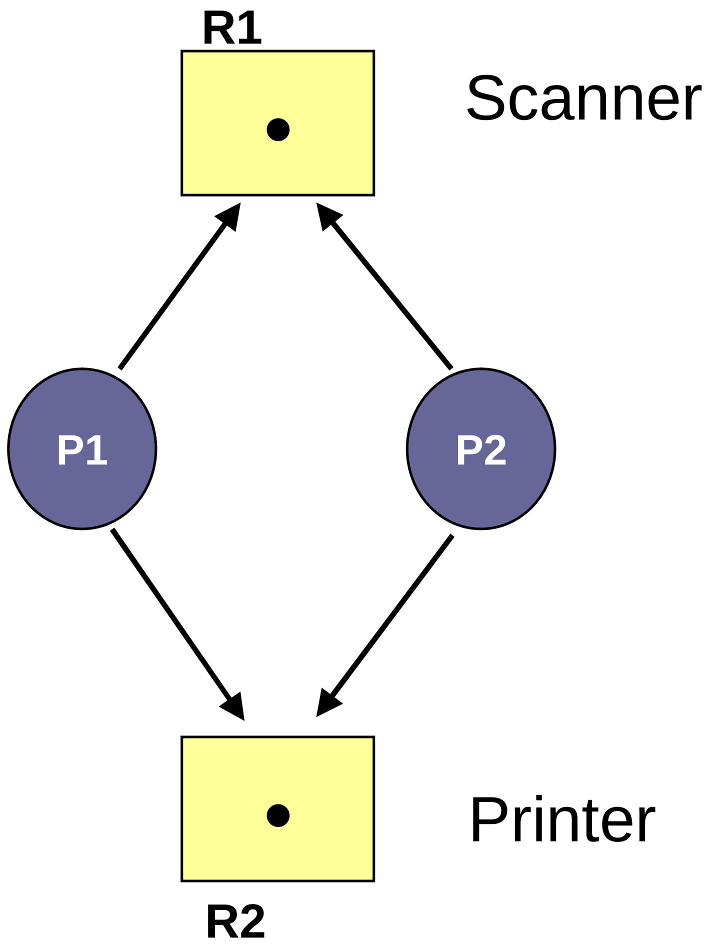
- This ‘deadlock’ is innate within the real-life system.
- If we modelled it using Java concurrency – we would also expect deadlock to occur … since ‘deadlock’ occurs in the real-life situation.
- The Bounded Cyclic Buffer **does not** have deadlock in the real-life situation (i.e. a person putting characters written on paper into pigeonholes and another person taking them out …) This deadlock results from poor implementation of the real-life situation.
- The Traffic Gridlock (‘deadlock’) is generally eliminated in the real-life situation by having a ***box junction (yellow criss-crosses in the UK)*** **-** ***“Do not enter box until your exit is clear” (See Euston Road next to the Station)***
- In this case the Java model of the real-life situation would also have its deadlock eliminated with this mechanism …

Formal Analysis – abstracting the situation - *Resource Allocation Graphs*



- Process P1 is an active entity (e.g. a hungry student)
- Process P1 holds one of the resources R1 (e.g. one of 4 rings on a cooker)
- Process P1 is requesting R2 (a spoon for example) and is waiting for it to be released
- Resource R2 is held by some unspecified process at present
- When R2 released, P1's request is instantaneously transformed to assignment edge.

Consider two processes P1 and P2 both requiring resources R1 and R2 (e.g. a scanner and a printer)



Different ways of allocating 2 resources …

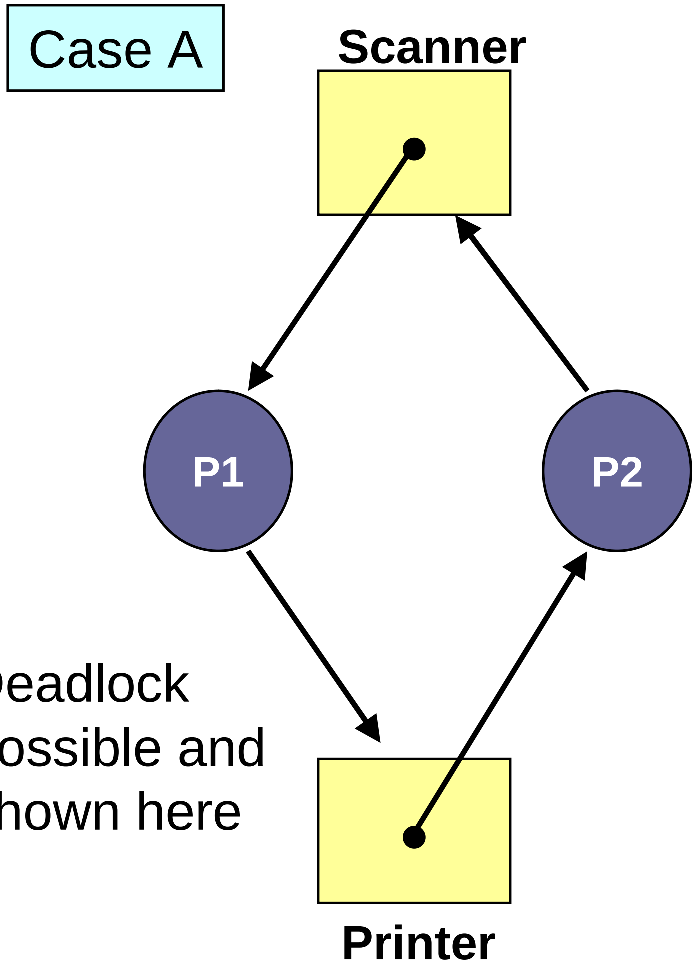
- Let us considered two ways of allocating these resources …

- ***Case A:*** P1 requests R1 then R2;
  P2 requests R2 then R1

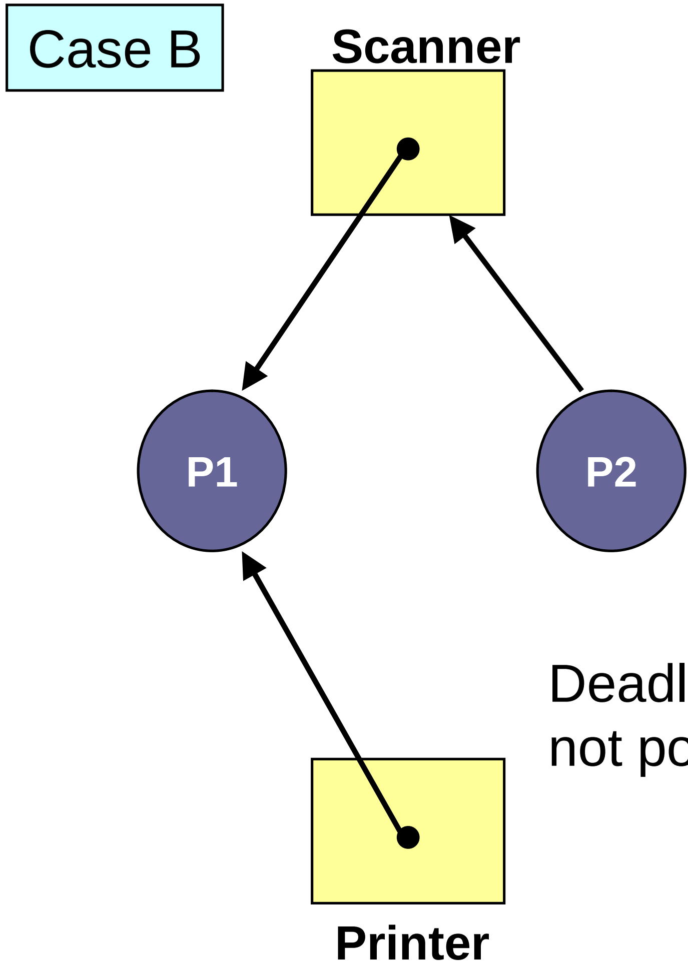
- ***Case B:*** P1 requests R1 then R2;
  P2 requests R1 then R2

- **The ordering is vital** … a “general” rule is that different processes should acquire shared resources in the same order.

Possible Resource Allocation Graphs for these different cases …



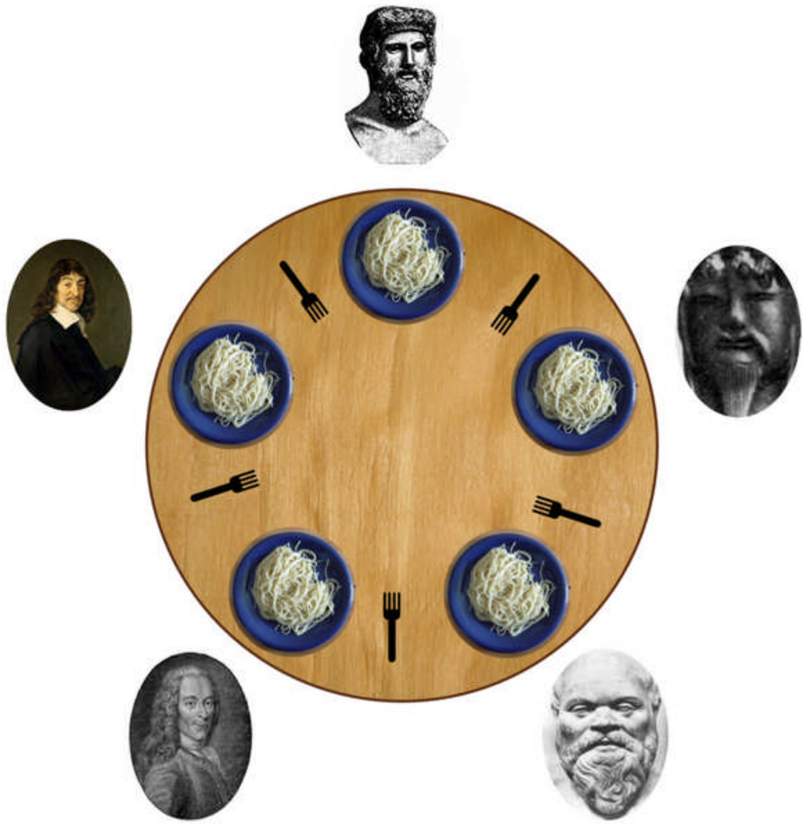
Deadlock possible and shown here



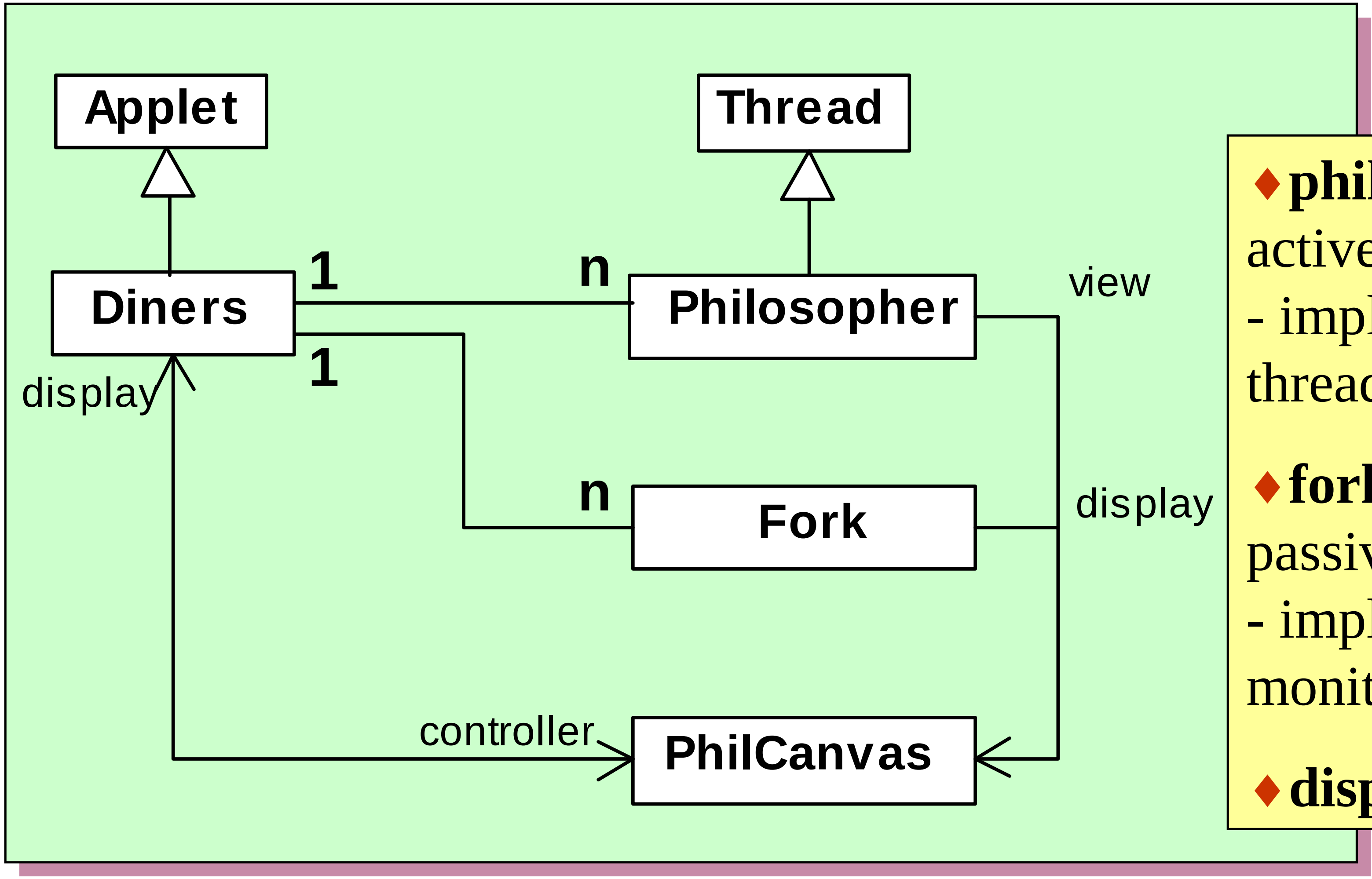
Deadlock not possible

Dining Philosopher Problem

- A classic problem examined by Dijkstra in 1968 – many synchronization algorithms employ this as a test case
- 5 Philosophers sit around table
- They think or eat
- Eat with 2 forks
- Can only afford 5 forks (philosophy doesn't pay as well as computer science!)
- Each philosopher only uses forks to their left and right



Dining Philosophers in Java (Magee & Kramer Ch. 6)



- **philosophers**: active entities - implement as threads
- **forks**: shared passive entities - implement as monitors
- **display**

Dining Philosophers - Fork monitor

```
class Fork {
  private boolean taken=false;
  private PhilCanvas display;
  private int identity;

  Fork(PhilCanvas disp, int id)
    { display = disp; identity = id;}

  synchronized void put() {
    taken=false;
    display.setFork(identity,taken);
    notify();
  }

  synchronized void get()
     throws java.lang.InterruptedException {
    while (taken) wait();
    taken=true;
    display.setFork(identity,taken);
  }
}
```

"taken" encodes the state of the fork

Dining Philosophers - Philosopher Thread

```
class Philosopher extends Thread {
  ...
  public void run() {
    try {
      while (true) {                          // thinking
        view.setPhil(identity,view.THINKING);
        sleep(controller.sleepTime()); // hungry
        view.setPhil(identity,view.HUNGRY);
        right.get();                          // got right fork
        view.setPhil(identity,view.GOTRIGHT);
        sleep(500);
        left.get();                           // eating
        view.setPhil(identity,view.EATING);
        sleep(controller.eatTime());
        right.put();
        left.put();
      }
    } catch (java.lang.InterruptedException e){}
  }
}
```

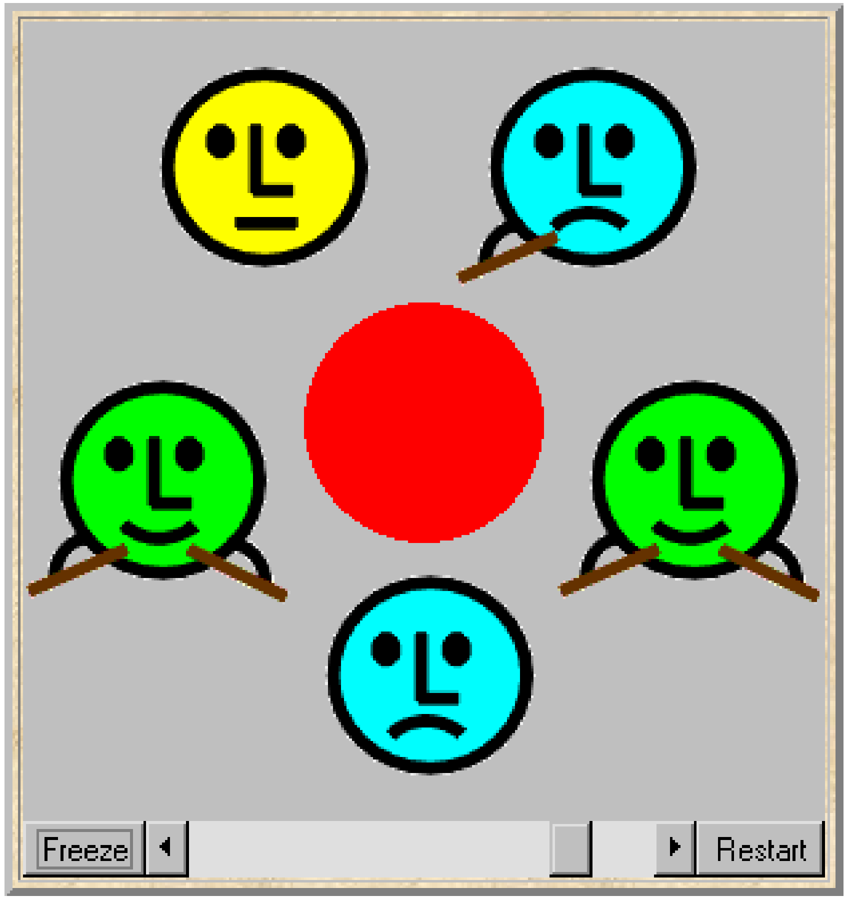
Follows from the behaviour of philosophers!

Dining Philosophers – Creation

Creates philosopher
Threads and fork Monitors

```
for (int i =0; i<N; ++i)
  fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
  phil[i] =
      new Philosopher
         (this,i,fork[(i-1+N)%N],fork[i]);
  phil[i].start();
}
```

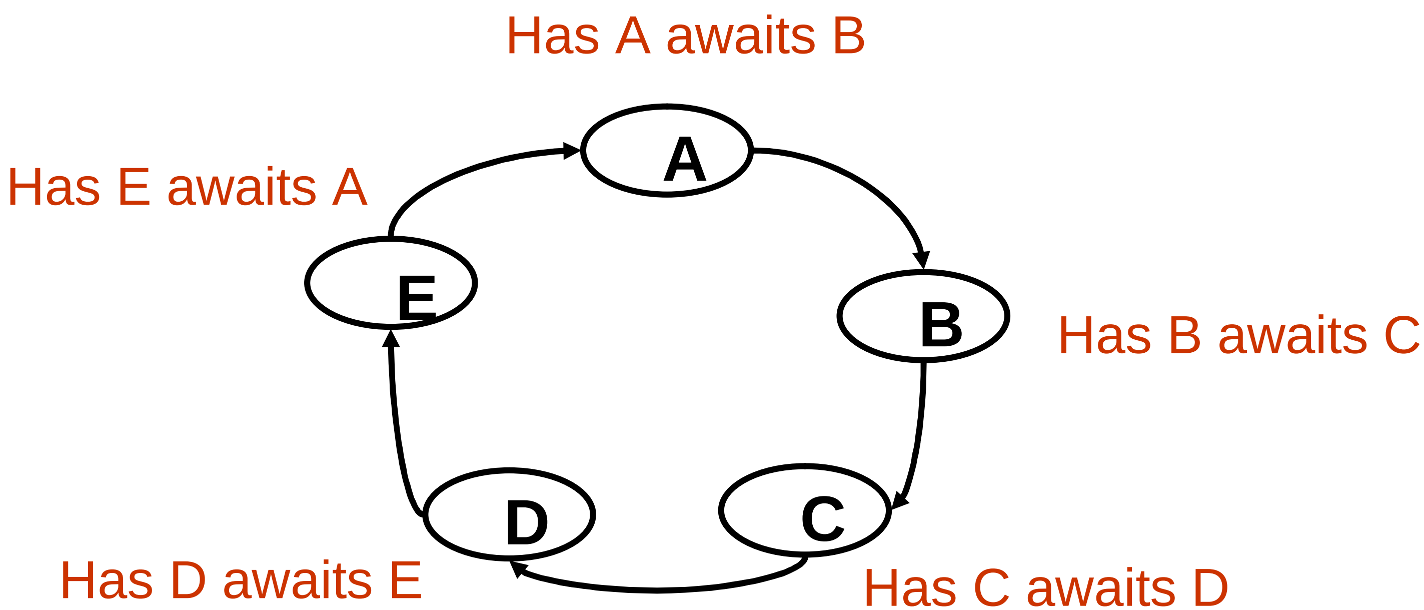
Lets see what happens …



The slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating.

Deadlock in Dining Philosopher

- If each philosopher has acquired their left fork the threads are mutually waiting for each other (wait-for cycle).
- Potential for deadlock exists independent of thinking and eating times – but probability is increased if these times are short.
- Deadlock can be avoided if every second philosopher went for his left fork instead of his right … this destroys the wait-for cycle !



Deadlock Conditions

- Deadlock exists if all 4 of these conditions hold in a system:
  - ***Shared resources with mutual exclusion:*** if a resource is being used, other processes need to wait.
  - ***Hold-and-wait:*** Processes hold only resources while waiting to acquire additional resources
  - ***No pre-emption:*** Resources cannot be pre-empted (forcefully withdrawn) - only released voluntarily by a process.
  - ***Wait-for Cycle:*** A cycle of processes exists such that each holds a resource requested by next process in the cycle (and refused).

Summary

- Safety and liveness properties
- Different types of Safety properties including the safety property deadlock.
- How deadlock occurs – Resource Allocation Graphs
- The classic dining philosophers problem
- The four conditions that are needed for deadlock