CSE201: Monsoon 2020
Advanced Programming

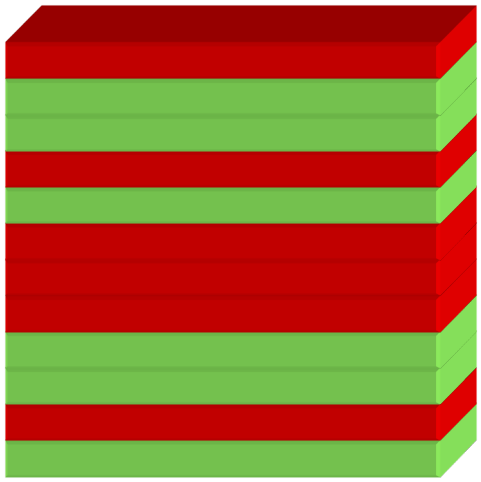
Lecture 20: Mutual Exclusion

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Lecture

- Race conditions
- Mutual exclusion
- Monitor locks

Race Condition



Put green pieces

How can we have alternating colors?

Put red pieces

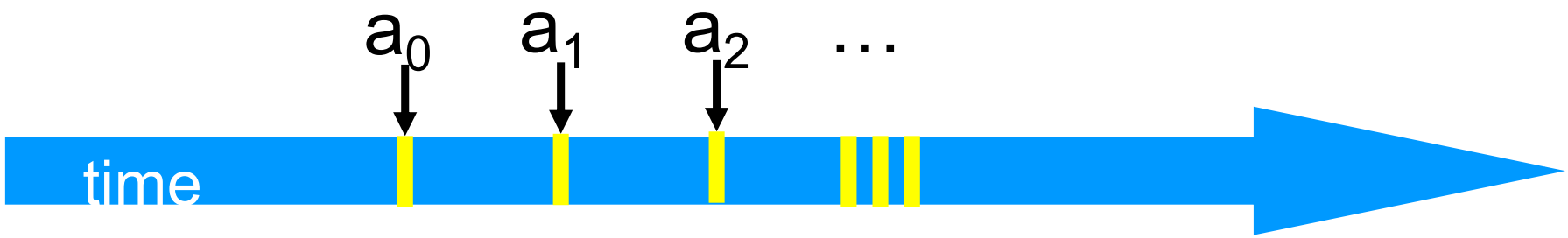
Mutual Exclusion

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
  - *Otherwise it results in a race condition!*

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Threads

- A ***thread*** A is (formally) a sequence a_0, a_1, ... of events
  - Notation: a_0 ➔ a_1 indicates order



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

Concurrent Execution Over Multiple Threads

- Thread A

- Thread B

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

Question

```
class Counter implements Runnable {
    int counter = 0;

    public void run() { counter++; }
    public static void main(String[] args)
                          throws InterruptedException {
        ExecutorService exec =
                    Executors.newFixedThreadPool(2);

        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }

        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }

        System.out.println(task.counter);
    }
}
```

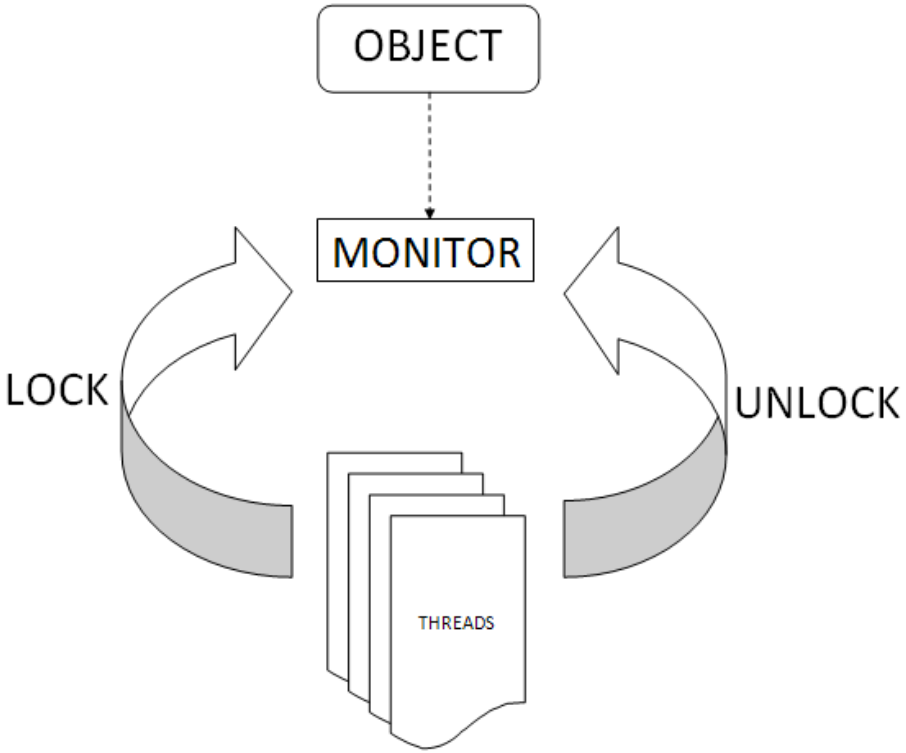
- What will be the output of this program?
  - Race on counter!
  - Buggy code and you will see different answers in different runs

Implementing Mutual Exclusion

```
class Counter implements Runnable {
    int counter = 0;
    // Both the versions of run method below is correct
    public synchronized void run() { counter++; }
 /* public void run() { synchronized(this) {counter++;} } */
    public static void main(String[] args)
                          throws InterruptedException {
        ExecutorService exec =
                    Executors.newFixedThreadPool(2);

        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }

        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }

        System.out.println(task.counter);
    }
}
```

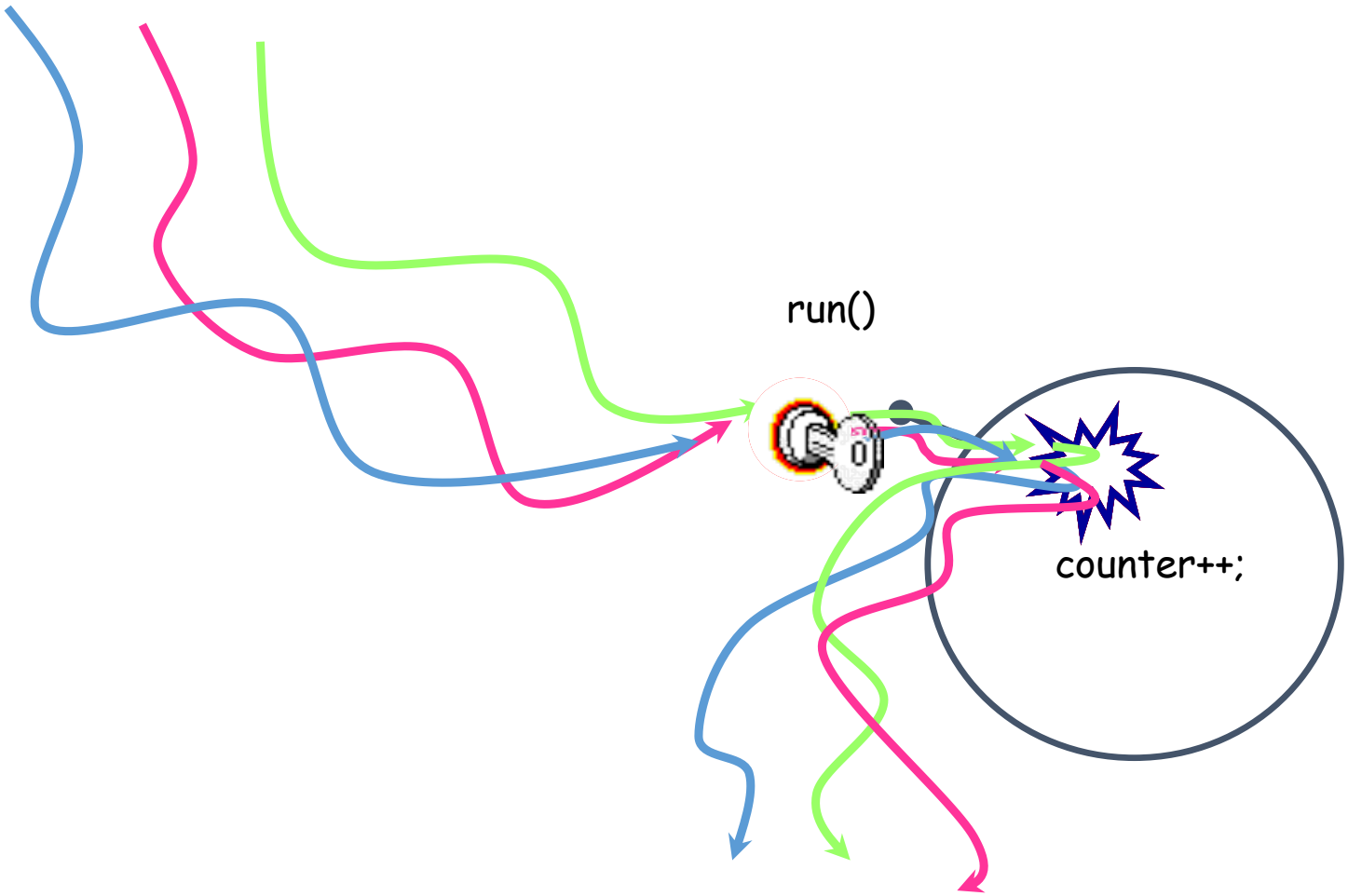
- Critical section
  - The `synchronized` methods (or block) define the critical sections
  - By using `synchronized` keyword we achieved mutual exclusion
    - Now let's analyze this

Monitors



- Each object has a "**monitor**" that is a token used to determine which application **thread** has control of a particular **object** instance
- In execution of a synchronized method (or block), access to the object monitor must be gained before the execution
- Access to the object monitor is queued
- Entering a monitor is also referred to as **locking** the monitor, or **acquiring ownership** of the monitor
- If a thread *A* tries to acquire ownership of a monitor and a different thread has already entered the monitor, the current thread (*A*) must wait until the other thread leaves the monitor

Analyzing our Counter Increment Example



- Only one thread can get the “key” to enter the “run” method i.e., take a lock on monitor
- Rest all threads will be queued to get the lock on monitor
- **Note**: There is no guarantee for fairness, i.e. longest waiting thread need not always get the lock first

Static Synchronized Methods

```
class Counter implements Runnable {
    static int counter = 0;

    public synchronized static void increment() {counter++;}

    public void run() { increment(); }
    public static void main(String[] args)
                          throws InterruptedException {
        ExecutorService exec =
                    Executors.newFixedThreadPool(2);

        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }

        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }

        System.out.println(Counter.counter);
    }
}
```

- Marking a static method as `synchronized`, **associates a monitor with the class** itself
- The execution of synchronized static methods of the same class is mutually exclusive

Next Lecture

- Memory consistency
- Producer consumer problem
- Quiz on Friday at 4.15pm
- Assignment on multithreading next week