Roadmap

- Overview of the RTSJ
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- **Resource Control**

Resource Sharing

Lecture aims:

- To introduce the notion of priority inversion
- To illustrate simple priority inheritance and priority ceil emulation inheritance
- To show the RTSJ Basic Model

Introduction and Priority Inversion

- In Java, communication and synchronization is based on a monitor-like construct. Synchronization mechanisms based on mutual exclusion suffer from priority inversion
- This is where a low-priority thread, **L**, enters into a mutual-exclusion zone shared with a high-priority thread, **H**. A medium-priority thread, **M**, then becomes runnable, pre-empts **L** and performs a computationally-intensive algorithm

Priority Inversion Continued

- At the start of this algorithm, a high-priority thread preempts **M** and tries to enter the mutual-exclusion zone
- As **L** currently occupies the zone, **H** is blocked
- **M** then runs, thereby indirectly blocking the progression of **H** for a potentially unbounded period of time
- It is this blocking that makes it very difficult to analyze the timing properties of SOs

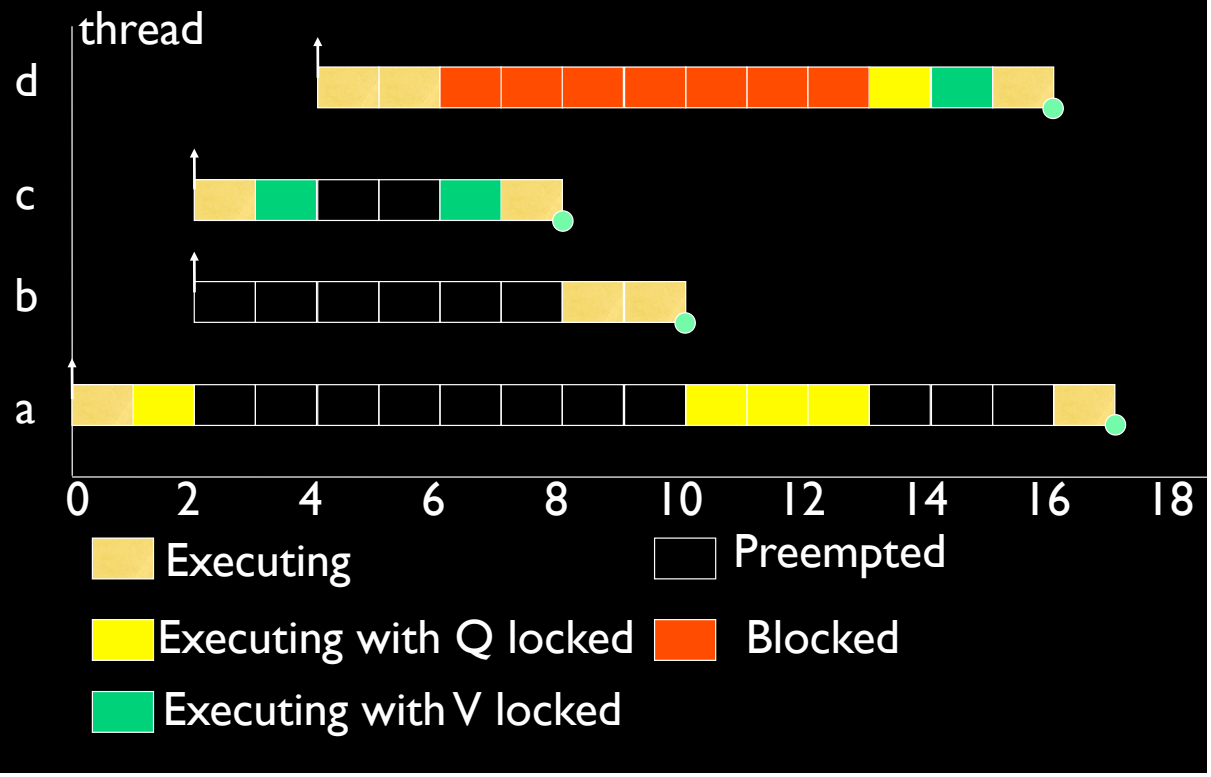
Priority Inversion

- To illustrate an extreme example of priority inversion, consider the executions of four periodic threads: a, b, c and d; and two resources (synchronized objects) : Q and V

thread	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

- Where E is executing for one time unit, Q is accessing resource Q for one time unit, V is is accessing resource V for one time unit

Example of Priority Inversion



Solutions to Priority Inversion

- Priority Inheritance
- Non-blocking communication

Priority Inheritance

- If thread p is blocking thread q, then q runs with p's priority

The RTSJ Basic Model

- Priority inversion can occur whenever a schedulable object is blocked waiting for a resource
- In order to limit the length of time of that blocking, the RTSJ requires the following:
  - All queues maintained by the system to be priority ordered (e.g. the queue of schedulable objects waiting for an object lock
    - Where there is more than one schedulable object in the queue at the same priority, the order should be first-in-first-out (FIFO). Similarly, the queues resulting from calling the Object.wait method should be priority ordered
  - Facilities for the programmer to specify different priority inversion control algorithms
    - By default, the RTSJ requires simple priority inheritance to occur whenever a schedulable object is blocked waiting for a resource

Monitor Control

```
public abstract class MonitorControl {
  protected MonitorControl();
  public static void setMonitorControl(MonitorControl plcy);
  public static void setMonitorControl(Object monitor,
                                       MonitorControl policy);
}
```

Priority Inheritance

```
public class PriorityInheritance extends MonitorControl {
  public static PriorityInheritance instance();
}
```

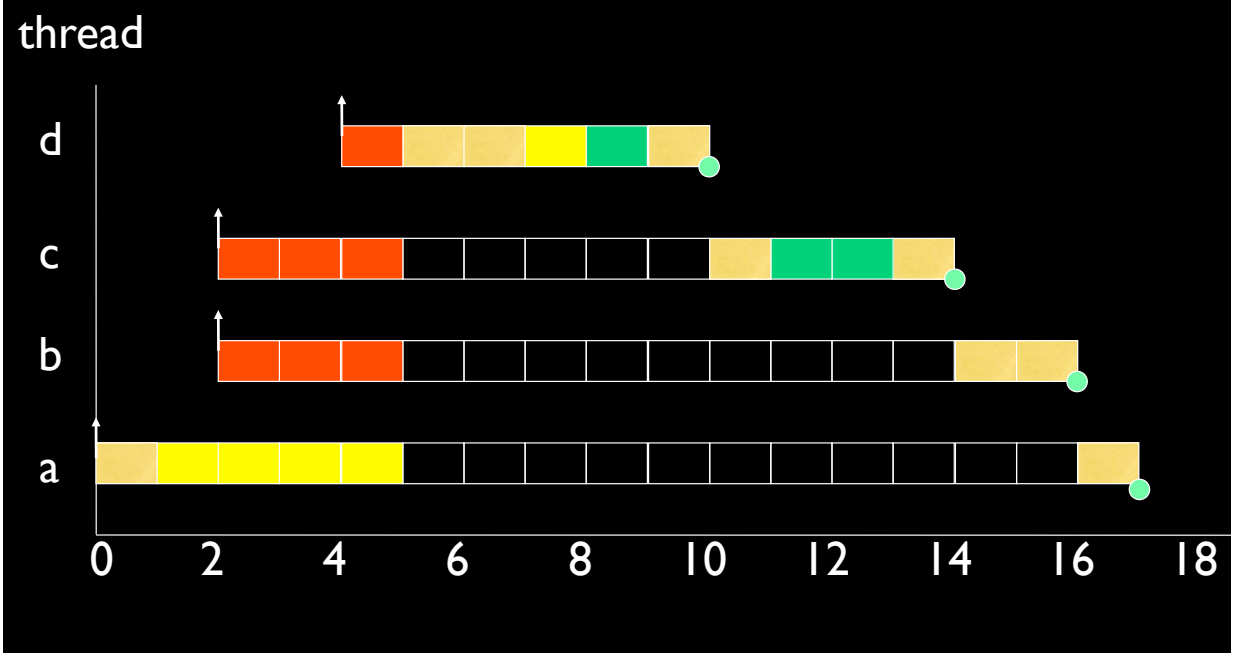

Blocking and Priority Inheritance

- If a thread has `m` critical sections that can lead to it being blocked then the maximum number of times it can be blocked is `m`
- `Priority ceiling emulation attempts to reduce the blocking`

Priority Ceiling Emulation

- Each thread has a static (base) default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined, this is the maximum priority of the threads that use it.
- A thread has a dynamic (active) priority that is the max of its static priority and the ceiling values of any resource it locked
- As a consequence, a thread will only suffer a block at the very beginning of its execution
- Once the thread starts actually executing, all the resources it needs must be free; if they were not, then some thread would have an equal or higher priority and the thread's execution would be postponed

Priority Ceiling Emulation Inheritance



Priority Ceiling Emulation

```
public class PriorityCeilingEmulation
                              extends MonitorControl {
  public static PriorityCeilingEmulation instance(int cl);
  public int getCeiling();
  ...
}
```


NOTE

- Generally, the code used inside a synchronized method (or statement) should be kept as short as possible, as this will dictate the length of time a low priority schedulable object can block a high one
- It is only possible to limit the block if the code doesn't contain:
  - unbounded loops,
  - arbitrary-length blocking operations that hold the lock, for example an arbitrarily-length sleep request

Ceiling Violations

- Whenever a schedulable object calls a synchronized method (statement) in an object which has the Priority-CeilingEmulation policy in force, the real-time virtual machine will check the active priority of the caller
- If the priority is greater than the ceiling priority, the uncheck CeilingViolationException is thrown

Communication between heap-using and no-heap using schedulable objects

- One of the main driving forces behind the RTSJ was to make real-time Java applications more predictable
- Hence, schedulable objects which need complete predictability can be defined not to reference the heap
- This means that they can pre-empt any garbage collection that might be occurring when the schedulable objects are released.

Communication bw heap- and no-heap

- Most large real-time systems will consists of a mixture of heap-using and no-heap schedulable objects. There will be occasions when they need to exchange information
- To ensure that no-heap SOs are not delayed by GC requires
  - all no-heap SOs should have priorities greater than heap-using SOs
  - PCE should be used for all shared synchronized objects,
  - all shared synchronized object should have their memory requirements pre-allocated and
  - objects passed in any communication should be primitive types passed by value or be from scoped or immortal memory
- If these conditions are fulfilled, then there will be no unwanted interference from the GC at inopportune moments

Summary

- Priority inversion is where a high priority thread is blocked by the execution of a lower priority thread
- Priority inheritance results in the lower priority thread inheriting the priority of the high priority thread whilst it is blocking the high priority thread
- Whilst priority inheritance allows blocking to be bounded, a block still occurs for every critical section
- Priority ceiling emulation allows only a single block
- The alternative to using priority inheritance algorithms is to use wait free communication (see book)