Resource Access Control (1)

Real-Time and Embedded Systems (M)

Lecture 13

UNIVERSITY
of
GLASGOW

Lecture Outline

- Definitions of resources
- Resource access control:
  - Non-preemptable critical sections
  - Basic priority inheritance protocol
  - Basic priority ceiling protocol

- Material corresponds to chapter 8 of Liu’s book

Resources

- Resources may represent:
  - Hardware devices such as sensors and actuators
  - Disk or memory capacity, buffer space
  - Software resources: mutexes, locks, queues, etc.

- Assume a system with ρ types of resource named R_1, R_2, …, R_ρ
  - Each resource comprises n_k indistinguishable units
    - Resources with a (practically) infinite number of units have no effect on scheduling; and so are ignored
  - Each unit of resource is used in a non-preemptable and mutually exclusive manner; resources are s*erially reusable*
  - If a resource can be used by more than one job at a time, we model that resource as having many units, each used mutually exclusively

- The system must control access to the resources

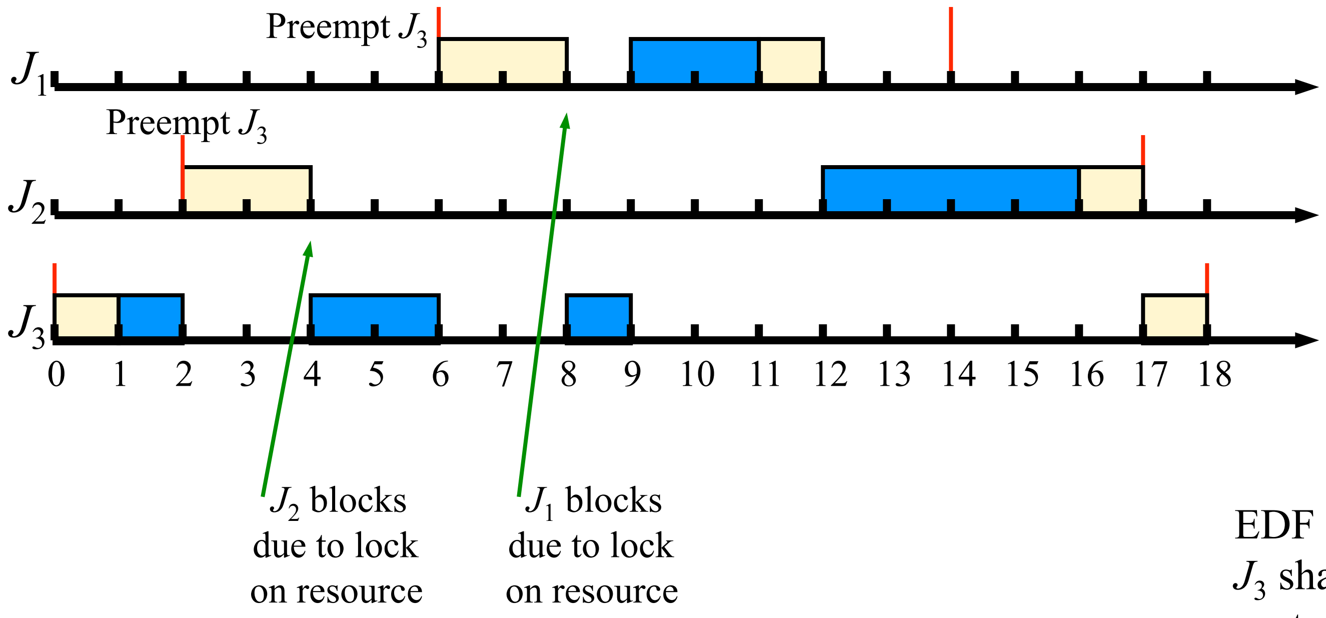
Locks and Critical Sections

- Assume a lock-based concurrency control mechanism
  - A job wanting to use n_k units of resource R_k locks $L(R_k, n_k)$ the resource
  - When the job is finished with the resources, it unlocks them: $U(R_k, n_k)$
  - If a lock request fails, the requesting job is blocked and loses the processor; when the requested resource becomes available, it is unblocked
    - A job holding a lock cannot be preempted by a higher priority job needing that lock

- The segment of a job that begins at a lock and ends at a matching unlock is a *critical section*
  - Use the expression $[R, n; e]$ to represent a critical section regarding n units of R, with the critical section requiring e units of execution time
  - Critical sections may nest if a job needs multiple simultaneous resources
    - E.g. lock R_1, then lock R_2, then lock R_3, …, unlock R_3, unlock R_2, unlock R_1 is represented as $[R_1, n_1; e_1\ [R_2, \mathrm{n}_2; e_2\ [R_3, \mathrm{n}_3; e_3]]]$

Contention for Resources

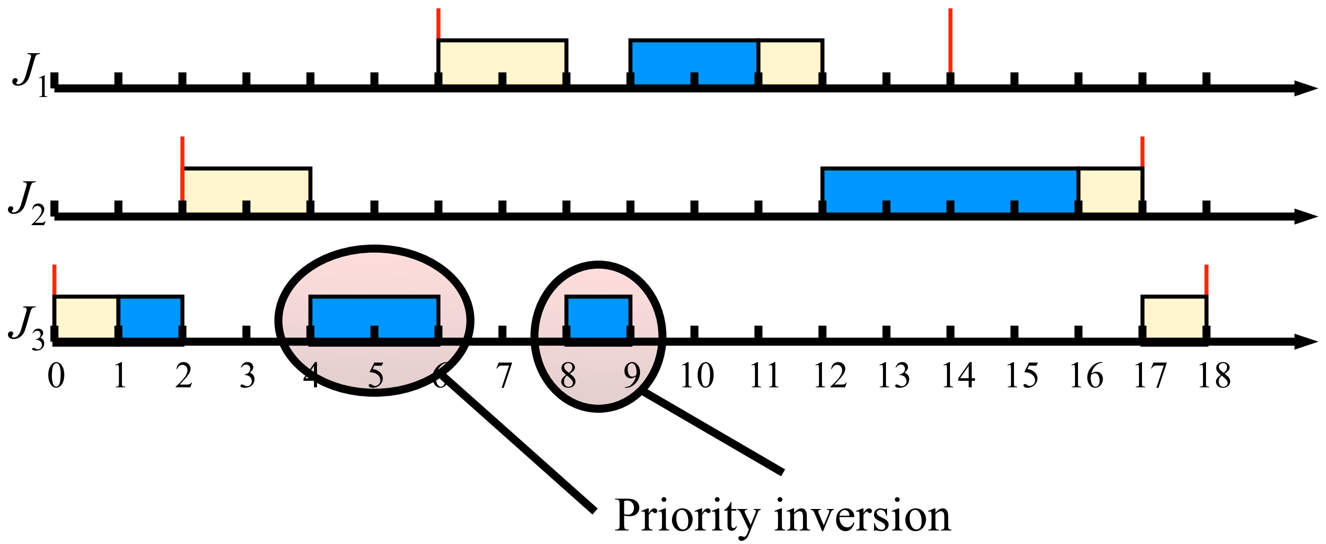
- Two jobs *conflict* with one another if some of the resources they require are of the same type; they *contend* for a resource if one job requests a resource that the other job has already been granted



EDF schedule of J_1, J_2 and J_3 sharing a resource R protected by locks. Red lines indicate release times and deadlines of jobs. Contention for R delays the higher priority jobs

Priority Inversion

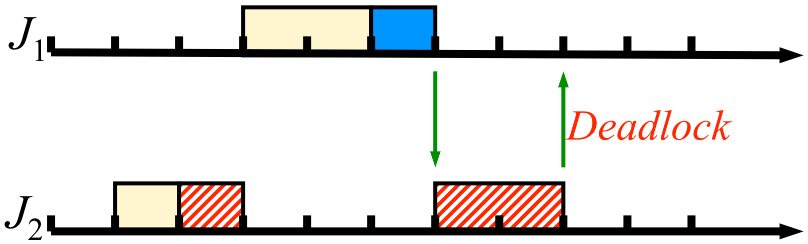
- *Priority inversion* occurs when a low-priority job executes while some ready higher-priority job waits



Contention for resources can cause priority inversions to occur, even if the jobs are preemptable, since a lower-priority job holding a lock on a resource will prevent a higher-priority job requiring that resource from executing

Deadlock

- *Deadlock* can result from piecemeal acquisition of resources; classic example of two jobs needing resources R_X and R_Y
  - If one job acquires locks in the order R_X then R_Y, and the other job acquires them in the opposite order, we can end up with a deadlock



J_1 wants to access blue after 2 units of execution, then red after a further 1 unit

J_2 wants to access red after 1 unit of execution, then blue after a further 3 units

The classic solution is to impose a fixed acquisition order over the set of lockable resources, and all jobs attempt to acquire the resources in that order (typically LIFO order)

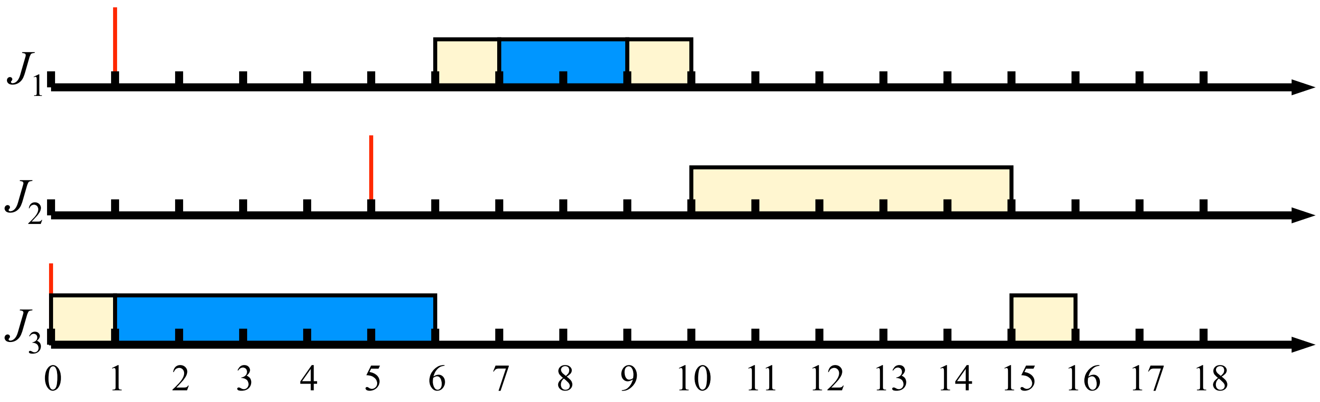
Timing Anomalies

- As seen, contention for resources can cause timing anomalies due to priority inversion and deadlock
- Unless controlled, these anomalies can be arbitrary duration, and can seriously disrupt system timing

- Cannot eliminate these anomalies, but several protocols exist to control them:
  - Non-preemptable Critical Sections
  - Priority inheritance protocol
  - Basic priority ceiling protocol
  - Stack-based priority ceiling protocol

Non-preemptable Critical Sections

- Simplest resource access control protocol: when a jobs acquires a resource it is scheduled with highest priority in a non-preemptable manner



Priority scheduled: J_1 has highest priority. Shading indicates the critical sections, red lines indicate release times for the jobs.

J_3 locks the resource and significantly delays execution of the other two jobs

Disadvantage: every job can be blocked by every lower-priority job with a critical section, even if there is no resource conflict; very poor timing performance

Priority-Inheritance Protocol

- Aim: to adjust the scheduling priorities of jobs during resource access, to reduce the duration of timing anomalies
- Constraints:
  - Works with any pre-emptive, priority-driven scheduling algorithm
  - Does not require any prior knowledge of the jobs' resource requirements
  - Does *not* prevent deadlock, but if some other mechanism used to prevent deadlock, ensures that no job can block indefinitely due to uncontrolled priority inversion

- We discuss the *basic* priority-inheritance protocol which assumes there is only 1 unit of resource
  - The book discusses how to generalize this to arbitrary amounts of resources

Basic Priority-Inheritance Protocol

- Assumptions (for all of the following protocols):
  - Each resource has only 1 unit
  - The priority assigned to a job according to a standard scheduling algorithm is its *assigned priority*
  - At any time t, each ready job J_k is scheduled and executes at its *current priority*, $\pi_k(t)$, which may differ from its assigned priority and may vary with time
    - The current priority $\pi_l(t)$ of a job J_l may be raised to the higher priority $\pi_h(t)$ of another job J_h
    - In such a situation, the lower-priority job J_l is said to *inherit* the priority of the higher-priority job J_h, and J_l executes at its inherited priority $\pi_h(t)$

Basic Priority-Inheritance Protocol

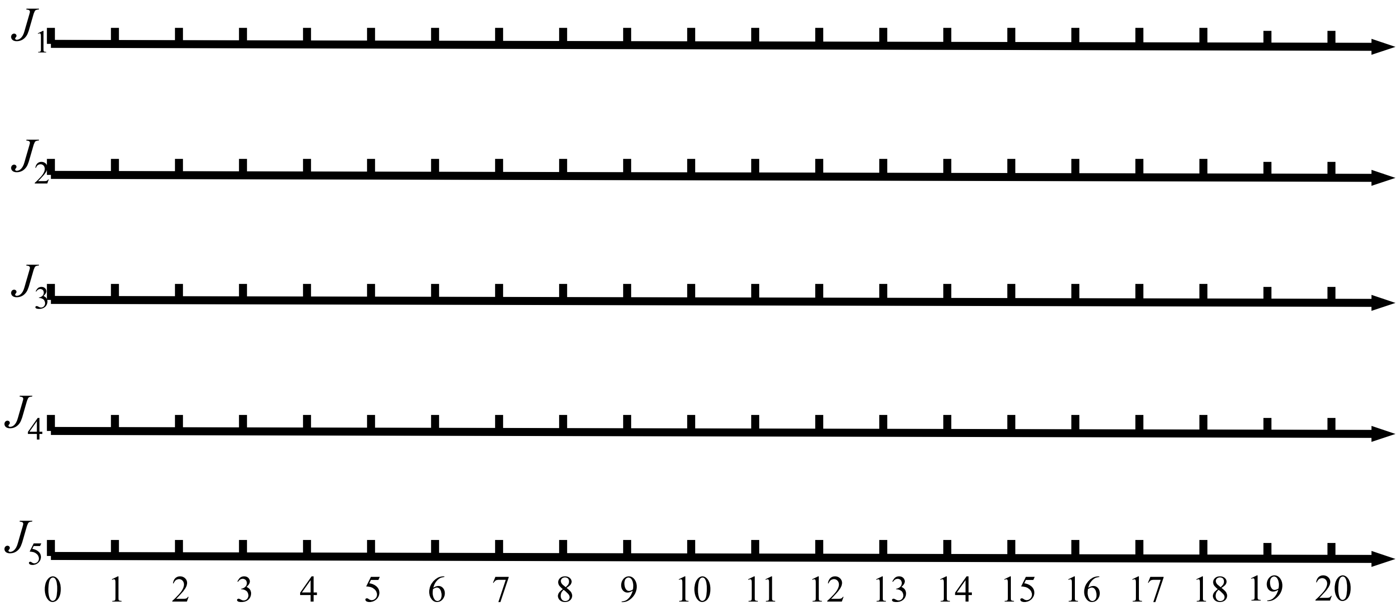
- Jobs are pre-emptively scheduled on the processor in a priority-driven manner according to their current priorities
  - At release time, the current priority of a job is equal to its assigned priority
  - The current priority remains equal to the assigned priority, except when the priority-inheritance rule is invoked
  - Priority-inheritance rule:
    - When the requesting job, J, becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J
    - J_l executes at its inherited priority until it releases R; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquired the resource R

- Resource allocation: when a job J requests a resource R at time t:
  - If R is free, R is allocated to J until J releases it
  - If R is not free, the request is denied and J is blocked
  - J is only denied R if the resource is held by another job

Basic Priority-Inheritance Protocol

- Consider an example system, with parameters are shown on the right →
- Jobs J_1, J_2, J_4 and J_5 attempt to lock their first resource after one unit of execution; J4 accesses  after an additional 2 units of execution

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[; 1]
J_2	5	3	2	[; 1]
J_3	4	2	3	
J_4	2	6	4	[; 4 [; 1.5]]
J_5	0	6	5	[; 4]



Basic Priority-Inheritance Protocol

- Properties of the Priority-inheritance Protocol
  - Simple to implement, does not require prior knowledge of resource requirements
  - Jobs exhibit different types of blocking
    - Direct blocking due to resource locks
    - Priority-inheritance blocking
    - Transitive blocking
  - Deadlock is *not* prevented
    - Although it can be prevented by using additional protocols in parallel
  - Can reduce blocking time compared to non-preemptable critical sections, but does not guarantee to minimize blocking

Basic Priority-Ceiling Protocol

- Sometimes desirable to further reduce blocking times due to resource contention
- The *basic priority-ceiling protocol* provides a means to do this, provided:
  - The assigned priorities of all jobs are fixed (e.g. RM scheduling, not EDF)
  - The resources required by all jobs are known a priori

- Need two additional terms to define the protocol:
  - The *priority ceiling* of any resource R_k is the highest priority of all the jobs that require R_k and is denoted by $\Pi(R_k)$
  - At any time t, the current priority ceiling $\Pi(t)$ of the system is equal to the highest priority ceiling of the resources that are in use at the time
  - If all resources are free, $\Pi(t)$ is equal to Ω, a nonexistent priority level that is lower than the lowest priority level of all jobs

Basic Priority-Ceiling Protocol

- Scheduling rules:
  - Jobs are scheduled in a preemptable priority-driven manner
  - On release time, the current priority of a job is equal to its assigned priority
  - The current priority remains equal to the assigned priority, except when the priority-inheritance rule is invoked

- Resource allocation rule:
  - When a job J requests a resource R held by another job, the request fails and the requesting job blocks
  - When a job J requests a resource R at time t, and that resource is free:
    - If J's priority $\pi(t)$ is higher than current priority ceiling $\Pi(t)$, R is allocated to J
    - If J's priority $\pi(\mathrm{t})$ is not higher than current priority ceiling $\Pi(\mathrm{t})$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\Pi(t)$; otherwise, the request is denied, and J becomes blocked
  - Unlike priority inheritance: *can* deny access to an available resource

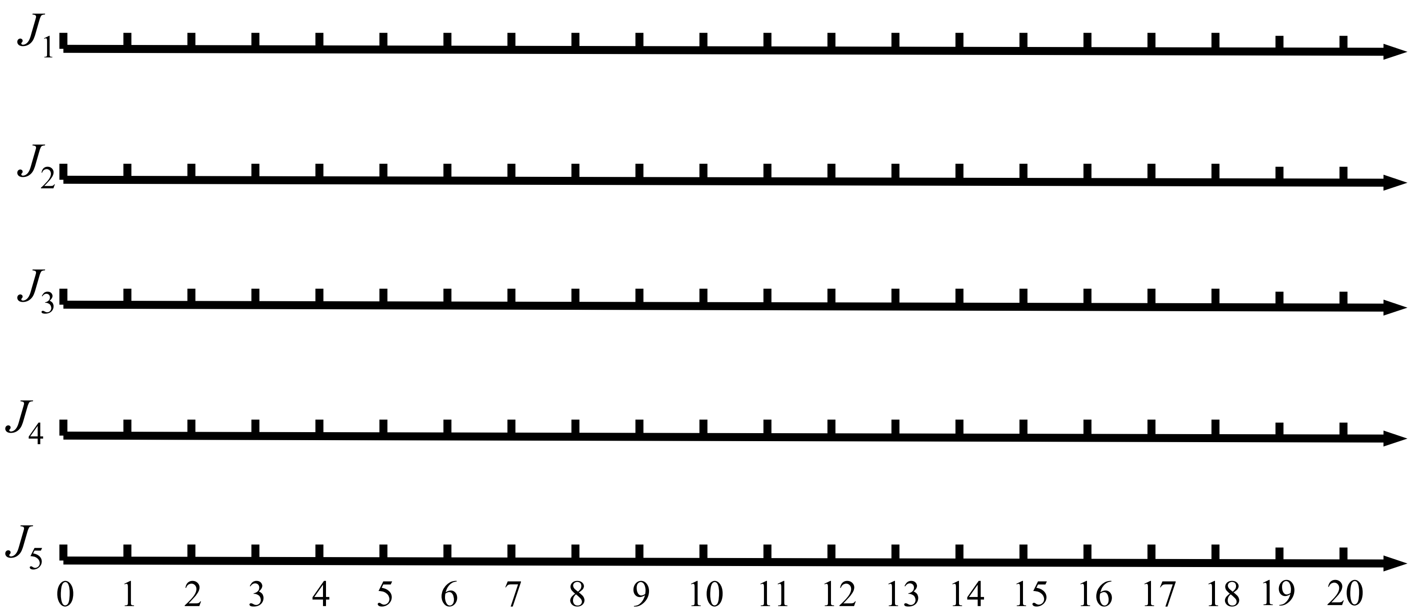
Basic Priority-Ceiling Protocol

- Priority-inheritance rule:
  - When the requesting job, J, becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J
  - J_l executes at its inherited priority until the time when it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it was granted the resource(s)

Basic Priority-Ceiling Protocol

Consider an example system, with parameters are shown on the right →
Jobs J_1, J_2, J_4 and J_5 attempt to lock their first resource after one unit of execution; J_4 accesses after an additional 2 units of execution

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[; 1]
J_2	5	3	2	[; 1]
J_3	4	2	3	
J_4	2	6	4	[; 4 [; 1.5]]
J_5	0	6	5	[; 4]



Basic Priority-Ceiling Protocol

- If resource access in a system of preemptable, fixed priority jobs on one processor is controlled by the priority-ceiling protocol:
  - Deadlock can never occur
  - A job can be blocked for at most the duration of one critical section
    - There is no transitive blocking under the priority-ceiling protocol

- Differences between the priority-inheritance and priority-ceiling protocols:
  - Priority inheritance is greedy, while priority ceiling is not
    - The priority ceiling protocol may withhold access to a free resource, causing a job to be blocked by a lower-priority job which does not hold the requested resource – termed avoidance blocking
  - The priority ceiling protocol forces a fixed order onto resource accesses, thus eliminating deadlock

Summary

- Defined resources, explaining timing anomalies and the need for resource access control
- Illustrated operation of three resource access control protocols:
  - Non-preemptable critical section
  - Basic priority inheritance protocol
  - Basic priority ceiling protocol

Next: more resource access protocols; practical aspects