Resource Synchronization Protocols

Real-Time and Embedded Operating Systems

Prof. Li-Pin Chang
ESSLab@NYCU

Motivation

- Tasks share and compete for resources
- Resource access from tasks must be synchronized
  - To avoid race conditions
  - To avoid deadlocks
  - To manage priority inversion
- Acquision (lock) -> use -> release
  - Acquisition is *never* rejected but may be postponed
  - The waiting time must be predictable

Okay, but first…

- What kind of system does not need synchronization on resource access?
  - Non-preemptible scheduling
    - A job releases all resources upon its completion
  - Cyclic executive or frame-based scheduling
    - Resource access is statically programmed

Resources

- Let us consider passive resources only
- When a task is using a passive resource, it is busy on the CPU. It does not suspend on the CPU.
  - E.g., data structures protected by sems or mutexes
- Active resources are not considered here
  - A task suspends on the CPU when using an active resource, e.g., I/O devices

Resources

- Consider *n* types of serially reusable resources $R_1, R_2, \ldots, R_n$
- R_i has v_i indistinguishable units of resource (instances)
  - For example
    - A global buffer has 20 empty slots
    - A DMA controller has 4 channels
  - Acquiring which one of the instances does not matter

Resources

- A job locks a resource before using it
  - The lock attempt may not be granted immediately
  - The task waits until the lock attempt is granted
- Locking is not granted immediately (and is postponed) bcz
  - the resource is not currently available, or granting the lock may cause some undesirable effects
- When a job no longer needs a resource, it unlocks it
  - Unlocking is effective immediately
- Both locking and unlocking may cause context switch

Resources

- Let the duration between the locking and unlocking of a resource be a critical section
- Let all critical sections be properly nested

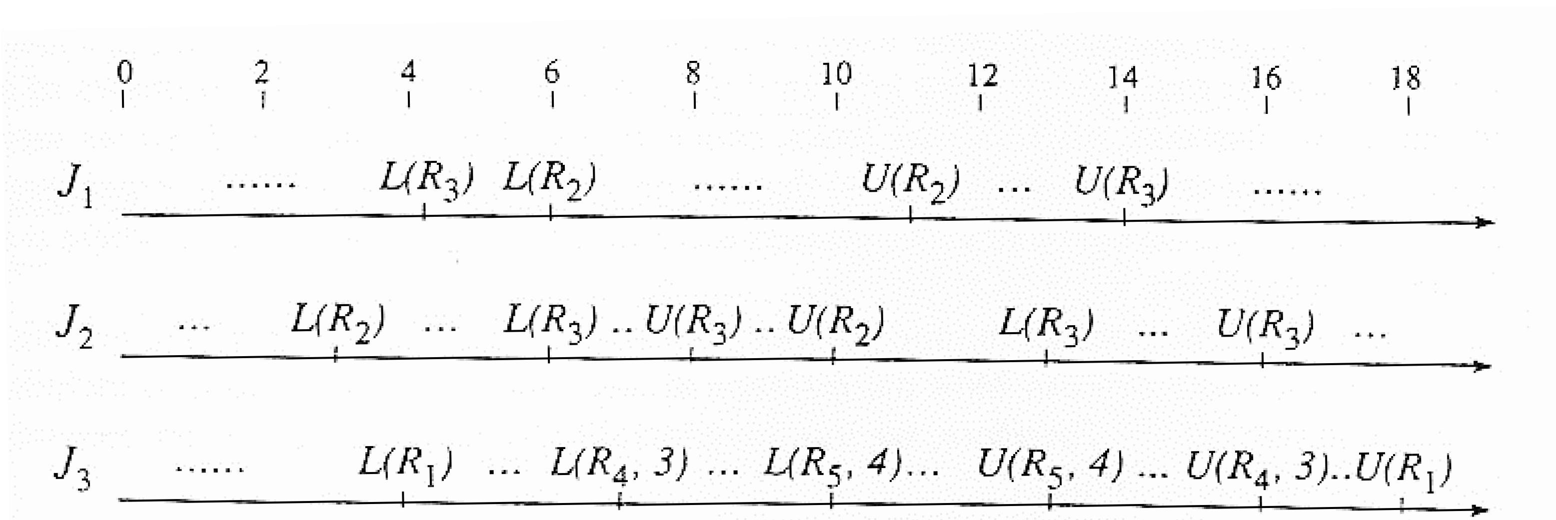


FIGURE 8–1 Examples of critical sections

Resources

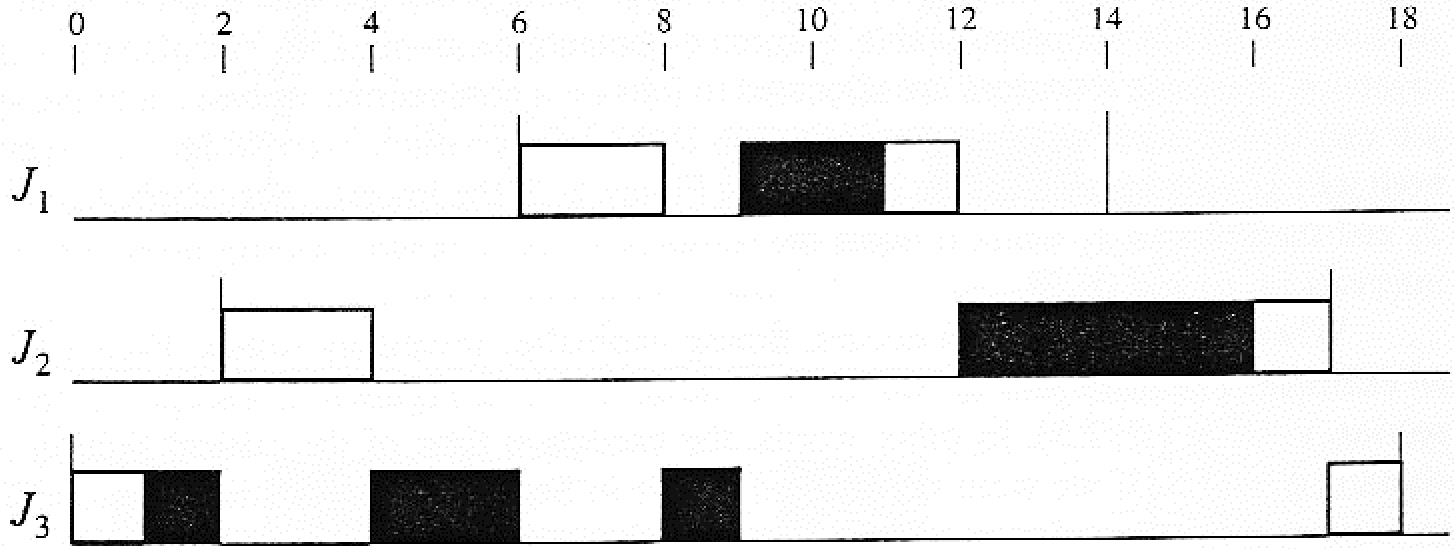


FIGURE 8–2 Example of job interaction due to resource contention.

- Resource contention increases task response time
  - If J1 and J2 do not require resources, they can finish by time 11 and 14, respectively (vs 12 and 17)

Resources

- A job is directly blocked if the requested resource is held by another job
- Use a wait-for graph to describe the run-time resource usage
- Use a bipartite graph to describe resource requirements

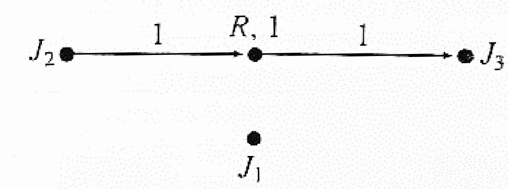
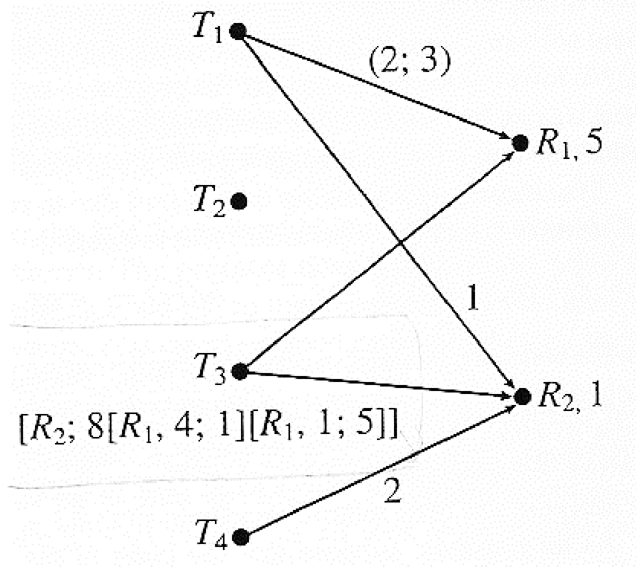


FIGURE 8–5 A wait-for-graph of the system in Figure 8-2 at time 5.



Undesirable Effects - I

- Priority inversion (blocking)
  - A high-priority job can not run because of resource contention from low-priority jobs
    - E.g., the high-priority job tries to lock a resource that is currently locked by a low-priority job
  - In this case, the high-priority job is blocked
    - Note: a low-priority job is never “blocked” by high-priority jobs (why?)
  - May damage the schedulability of HPTs

What happened on Mars?

← Pathfinder, protected by airbags

→ The Pathfinder

**Pictures courtesy of NASA

Pathfinder Mars Landing

Undesirable Effects – I (cont'd)

- Even worse a, job may be *indirectly* blocked by another job, even if the two jobs do not share any resources!!
  - uncontrollable priority inversion

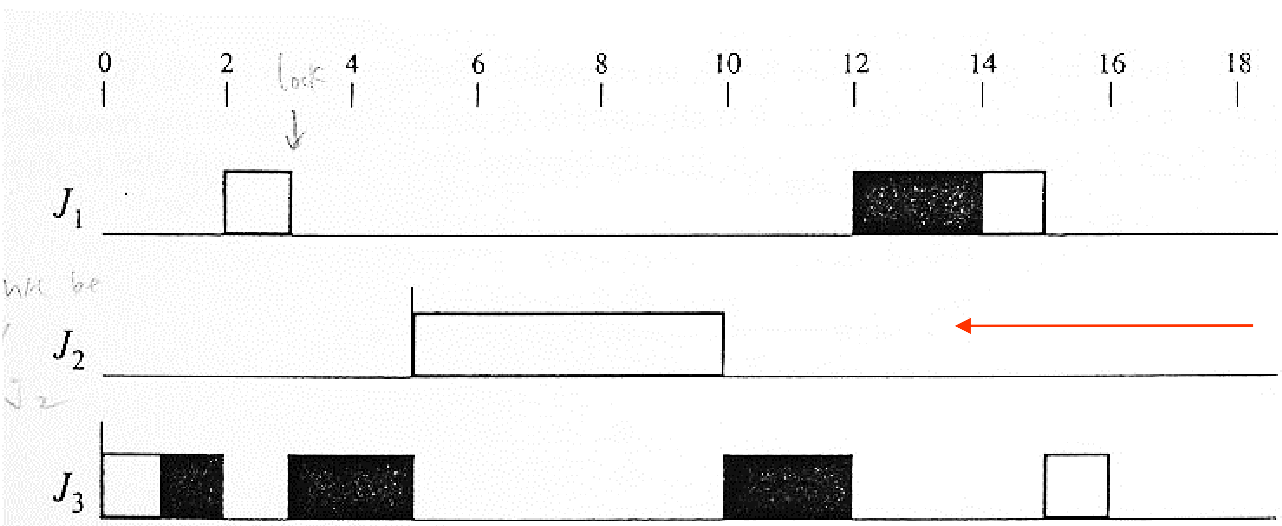


FIGURE 8–4 Uncontrolled priority inversion.

On Pathfinder, this medium-priority job cause a time-out of the high-priority J1!

Undesirable Effects - II

- Timing anomalies
  - Ideally, if a job completes earlier than expected, then the response time of all the other jobs become earlier (at least, not later)
  - But exceptions exist..

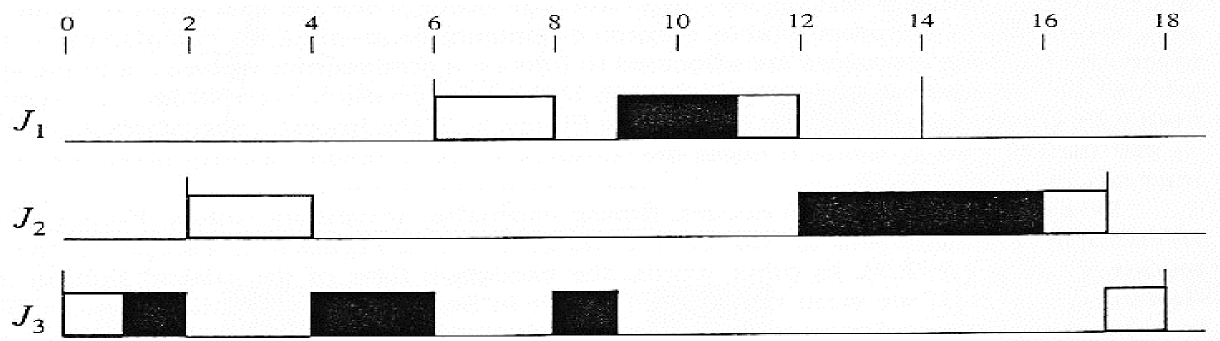


FIGURE 8–2 Example of job interaction due to resource contention.

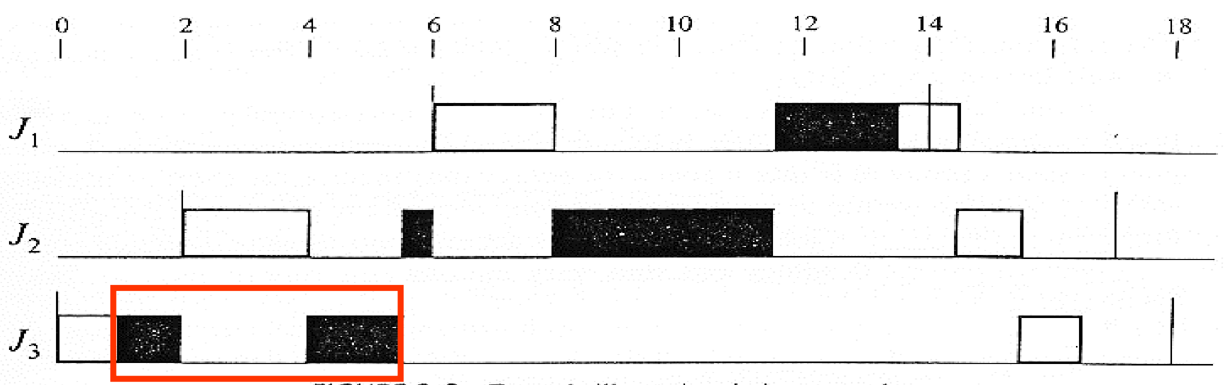
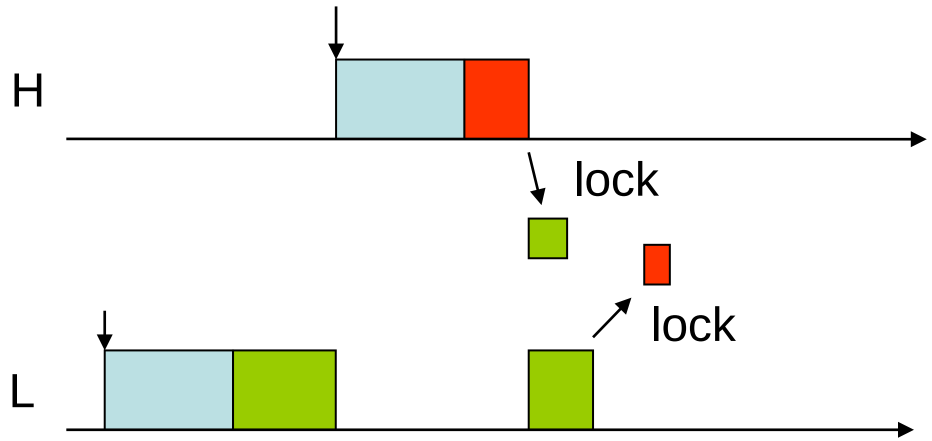


FIGURE 8–3 Example illustrating timing anomaly.

Now the duration of this CS becomes shorter…

Undesirable Effects - III

- Deadlocks
  - A fatal error in real-time systems
  - If a deadlock occurs, then some jobs are involved in circular waiting
  - Deadlock prevention: enforcing a partial order on resource access



Undesirable Effects

- Priority inversions and timing anomalies are inevitable with mutual-exclusive access
  - They should be managed (e.g., bounded)
- Deadlocks are a fatal error
  - They must be avoided

Resource Synchronization Protocols

- Protocol: a set of rules
  - to grant and to postpone lock requests
  - to schedule jobs (priority manipulation)

Non-Preemptible Critical Section

- NPCS: A job becomes non-preemptibe when it is using a resource
  - Any HPT cannot preempt it
  - Equivalent to locking the scheduler

Non-Preemptible Critical Section

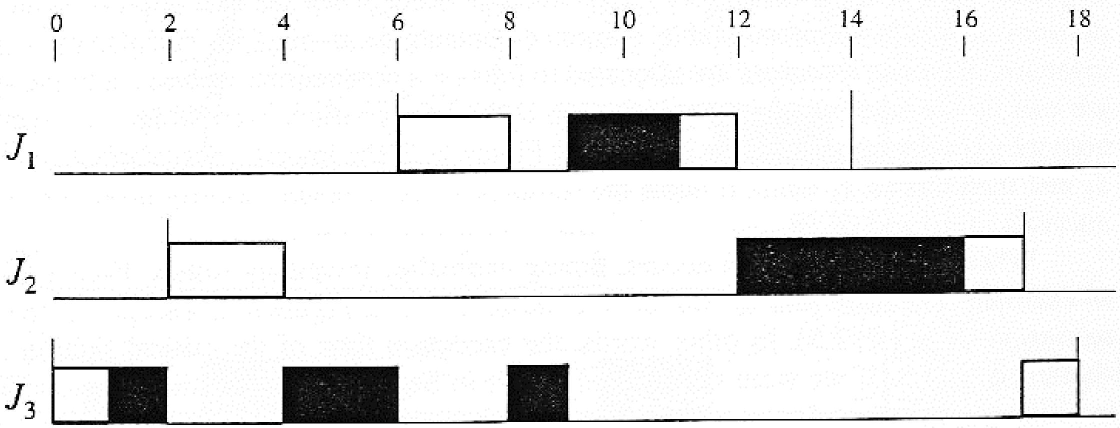


FIGURE 8–2 Example of job interaction due to resource contention.

←Ungoverned
（mutual exclusion only)

NPCS→

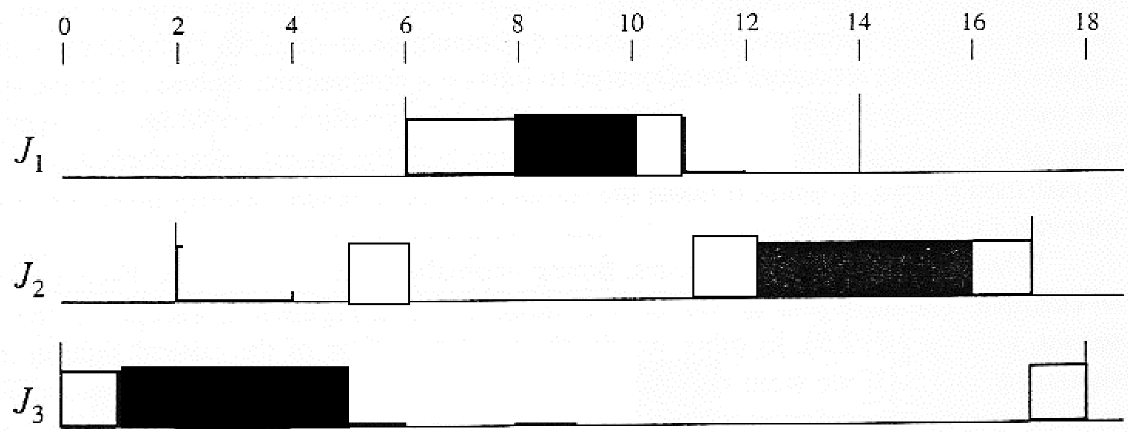


FIGURE 8–2 Example of job interaction due to resource contention.

Non-Preemptible Critical Section

- Deadlocks never occur
  - A job holding a resource can never be blocked (no hold and wait)
  - (why?)

- Uncontrollable priority inversions never occur
  - (why?)

Non-Preemptible Critical Section

- Let tasks in $\{T_1, T_2, ..., T_n\}$ be sorted in the rate-monotonic order and scheduled by RM
- The longest blocking time imposed on task T_i because of resource contention is

$$max_{i+1 \leq k \leq n}(\, s_k \,)$$

- s_k stands for the longest critical section of task T_k

Why at most once?

Non-Preemptible Critical Section

- If tasks are scheduled by EDF, the longest blocking time of T_i is

$$max_{i+1 \leq k \leq n} (s_k)$$

- Still the same (huh?)

Schedulability tests for RM/EDF with NPCS will be discussed later

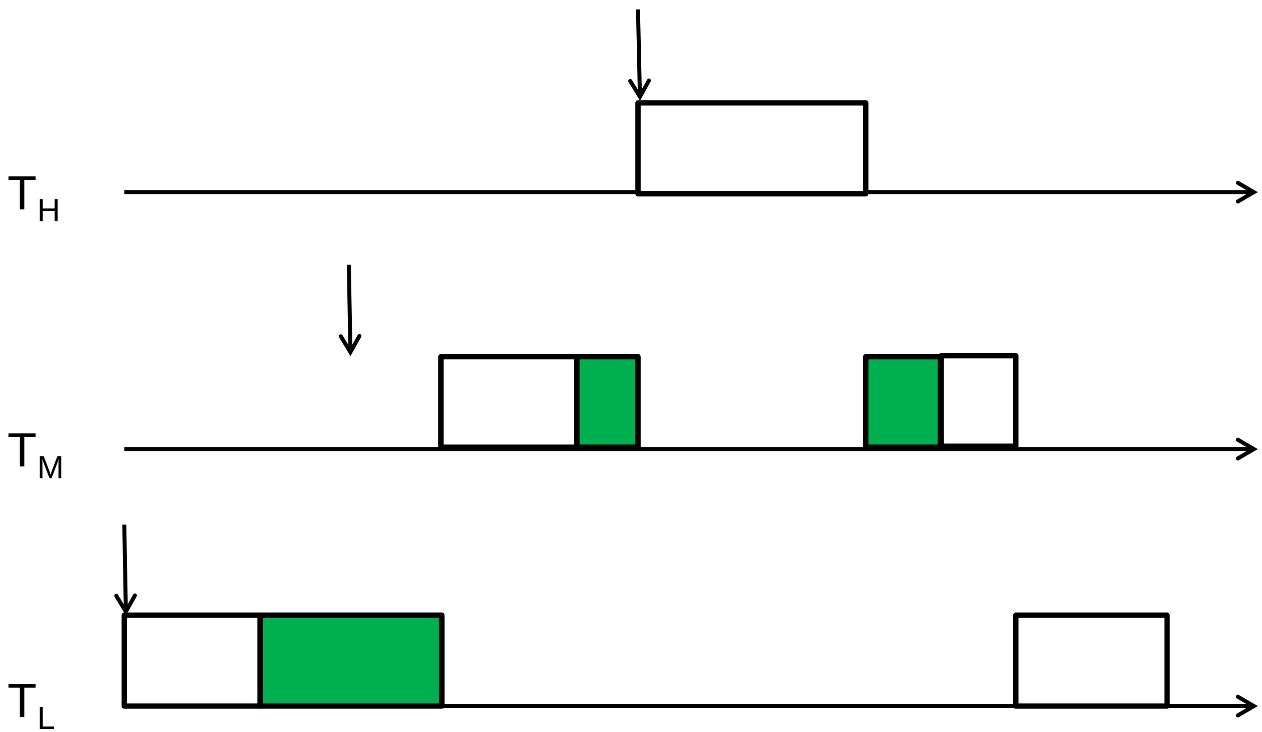
Non-Preemptible Critical Section

- Pros:
  - Easy to implement
  - No need to know resource usage *a priori*
- Cons:
  - Poor response (high priority tasks suffer)
  - An LPT blocks a HPT even if they do not share any resources

Ceiling-Priority Protocol

- Used in Ada real-time programming language
  - Commonly used in aerospace or aviation systems
- Resource usages are known *a priori*
- Consider a task set $T=\{T_a, T_b, T_c, T_d, T_e, T_f,\ldots\}$, sorted by priority (high->low)
  - Let T_b,T_d,T_e, share a resource R
  - When a task T locks R, its priority is boosted to T_b's priority until it unlocks R

Ceiling-Priority Protocol



Ceiling-Priority Protocol

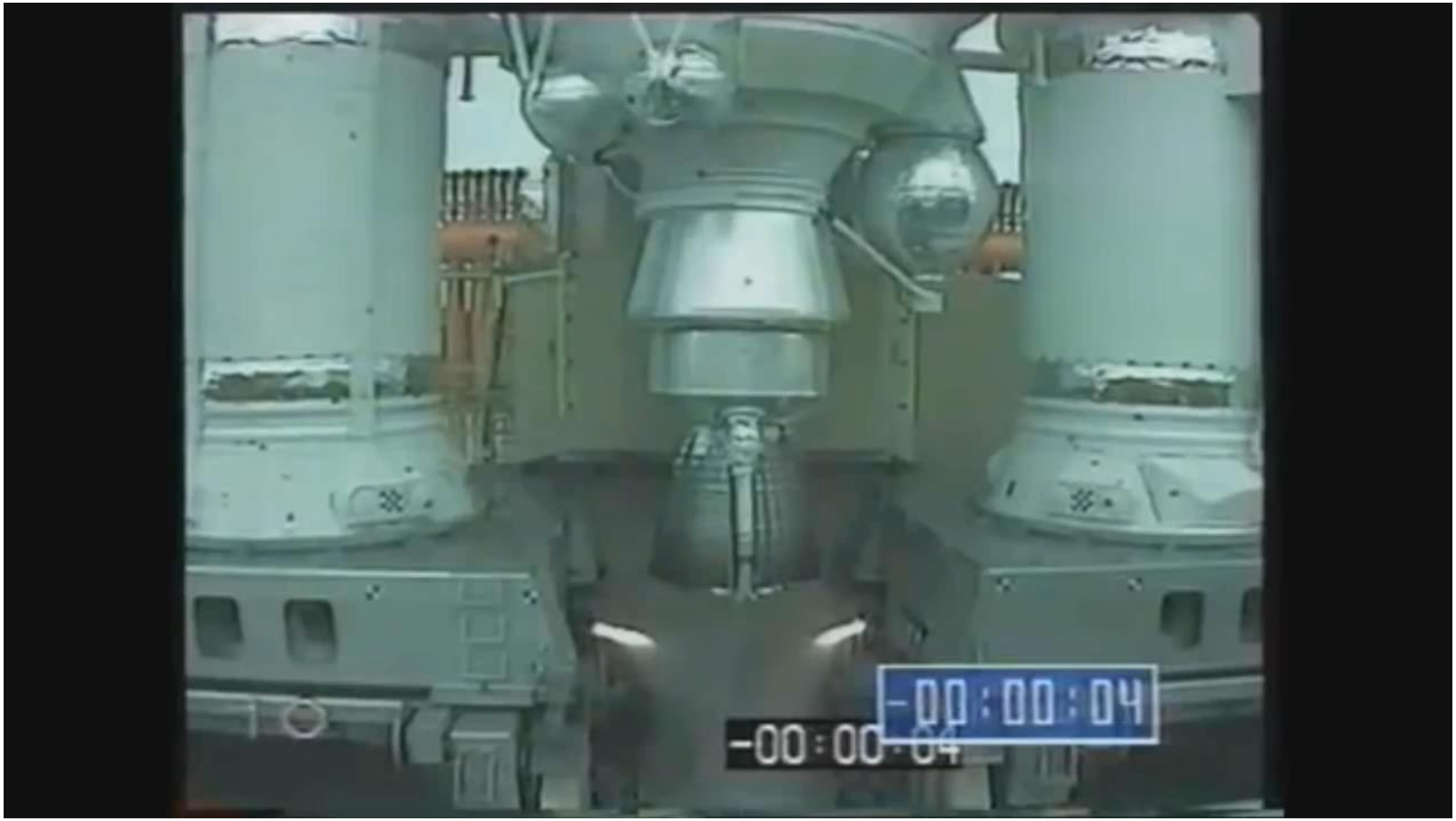
- As known as “highest-locker protocol”
- Better response than NPCS
- Free from
  - Uncontrollable priority inversion
  - Deadlocks
  - These properties directly follow from NPCS

Ceiling-Priority Protocol

- Blocking time
- Consider T_a, T_b, T_c, and T_d
  - T_b and T_d share a resource R
  - T_a and T_c do not use any resources
  - the longest critical sections $c_b=1$ and $c_d=2$

  - Blocking time $b_a=0, b_b=2, b_c=2, b_d=0$
  - NPCS?

What Happened to Ariean 5?



It’s nothing to do with Ada. It’s a software bug.

Priority-Inheritance Protocol

- NPCS is too restrictive because an LPT affects all tasks on locking
  - CPP suffer from the same problem but in a lower degree (highest locker's priority)
- PIP relieves such restriction
  - uncontrollable blocking must not occur
  - Priority is boosted on blocking, not locking

Priority-Inheritance Protocol

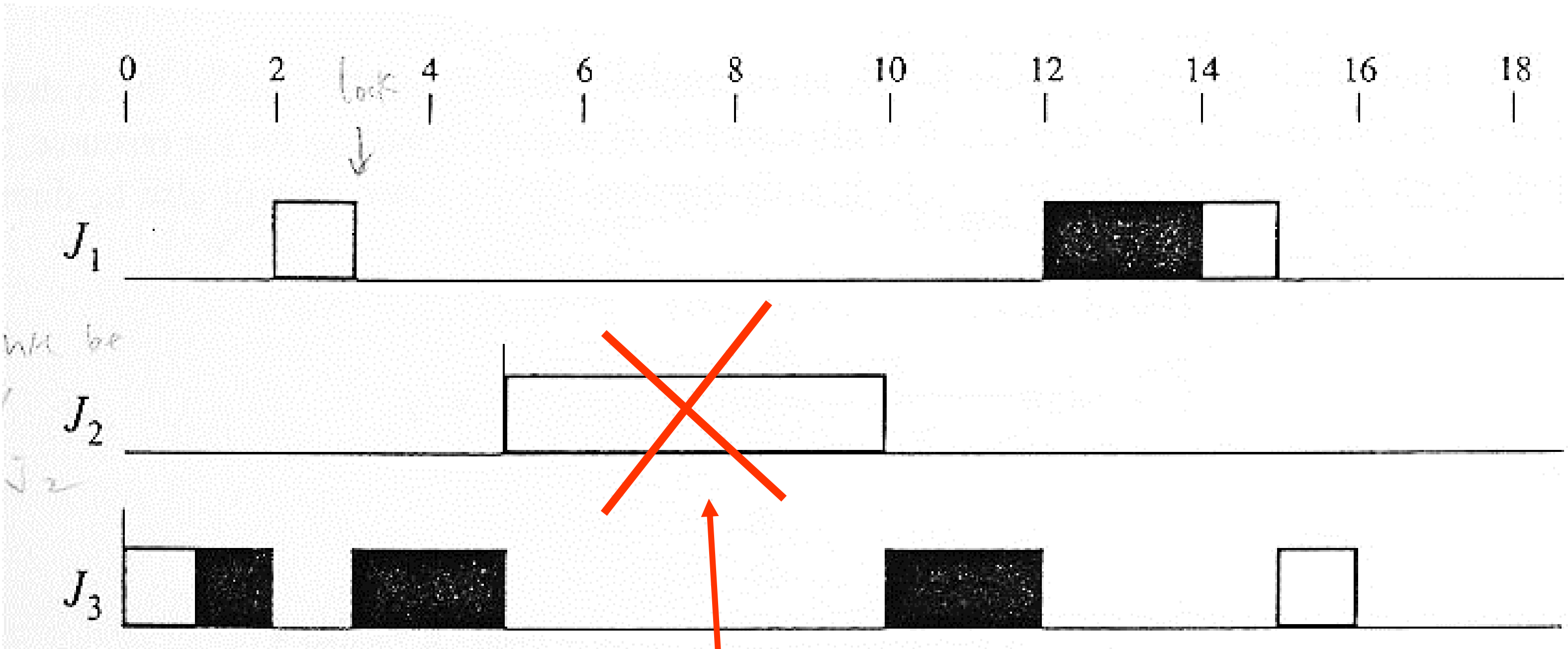


FIGURE 8–4 Uncontrolled priority inversion.

This job should not run because a higher-priority job is already waiting (being blocked)!!

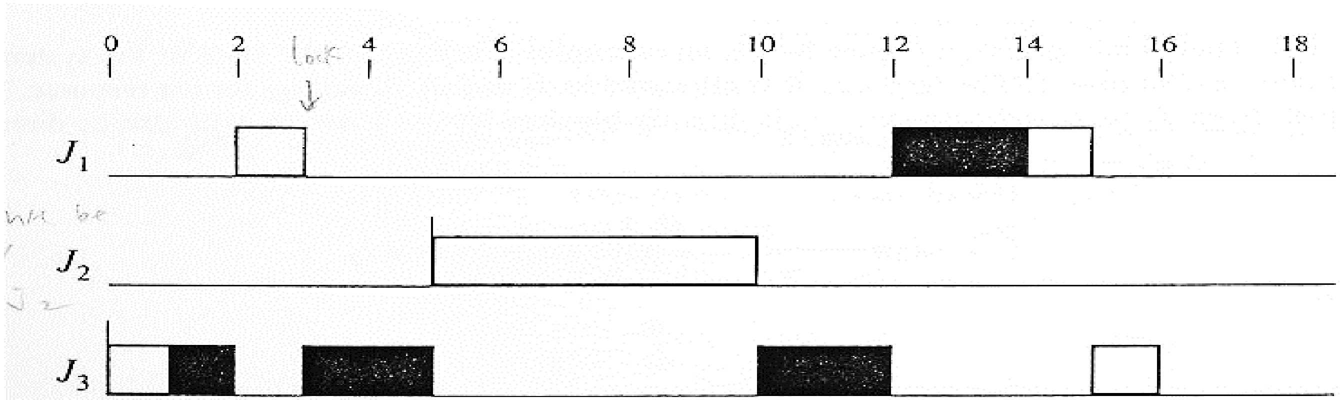
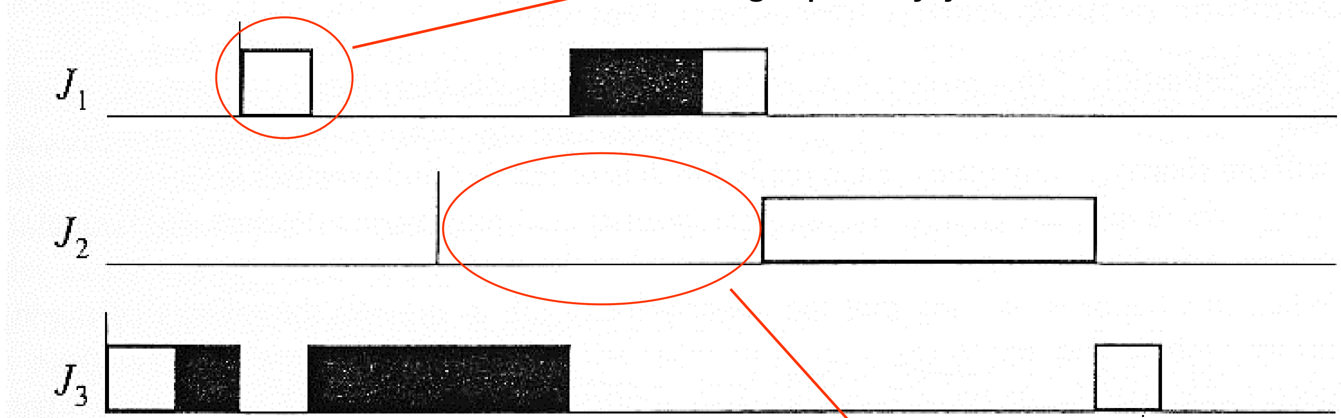


FIGURE 8–4 Uncontrolled priority inversion.

High-priority job is blocked



(b)

Medium-priority job is not executed because there is a blocked job with a higher priority

When blocking J1, J3 runs on behalf of J1

Priority-Inheritance Protocol

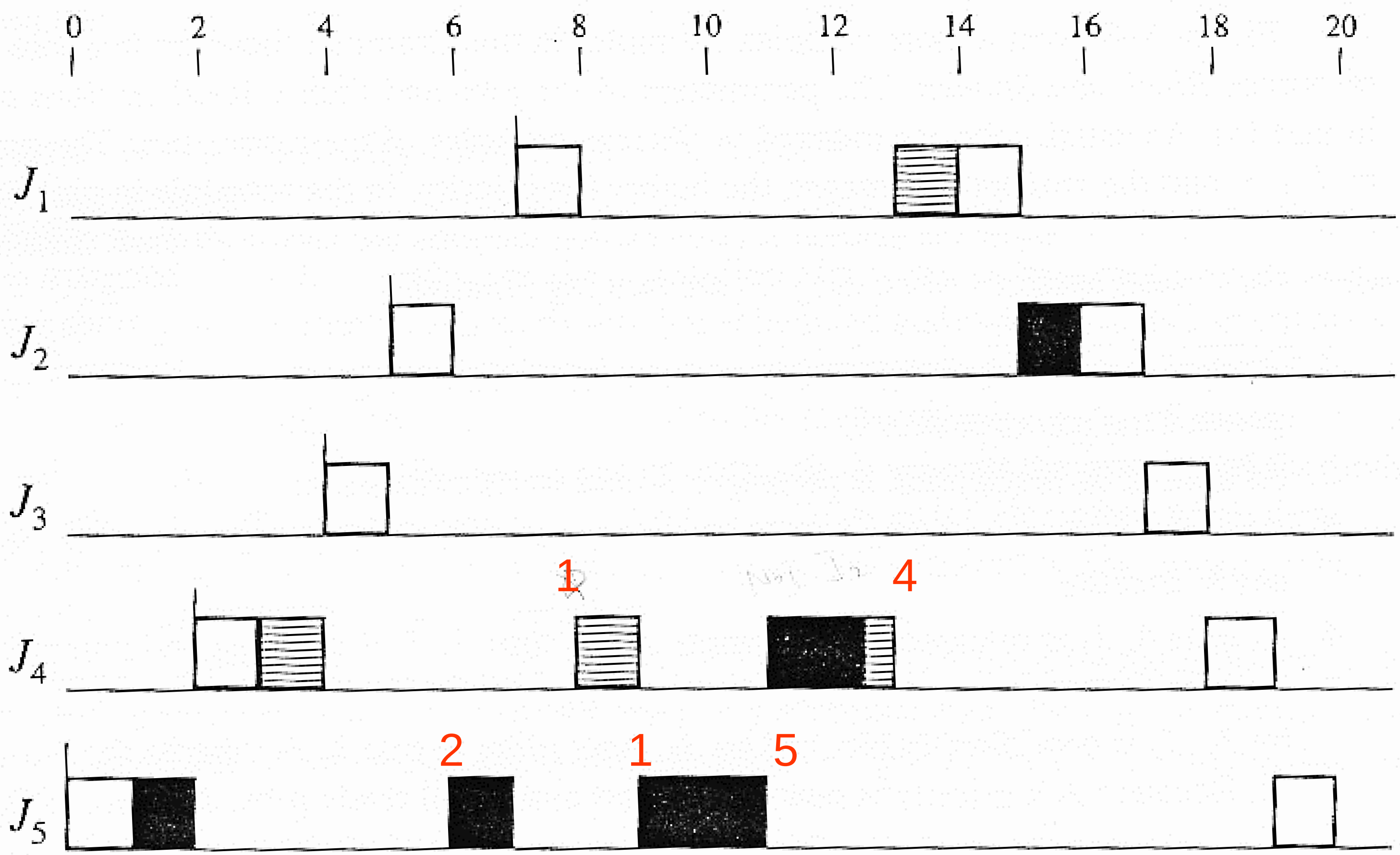
Rules of the Basic Priority-Inheritance Protocol

1. *Scheduling Rule*: Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.
2. *Allocation Rule*: When a job J requests a resource R at time t,
   - **(a)** if R is free, R is allocated to J until J releases the resource, and
   - **(b)** if R is not free, the request is denied and J is blocked.
3. *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. The job J_l executes at its inherited priority $\pi(t)$ 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 acquires the resource R.

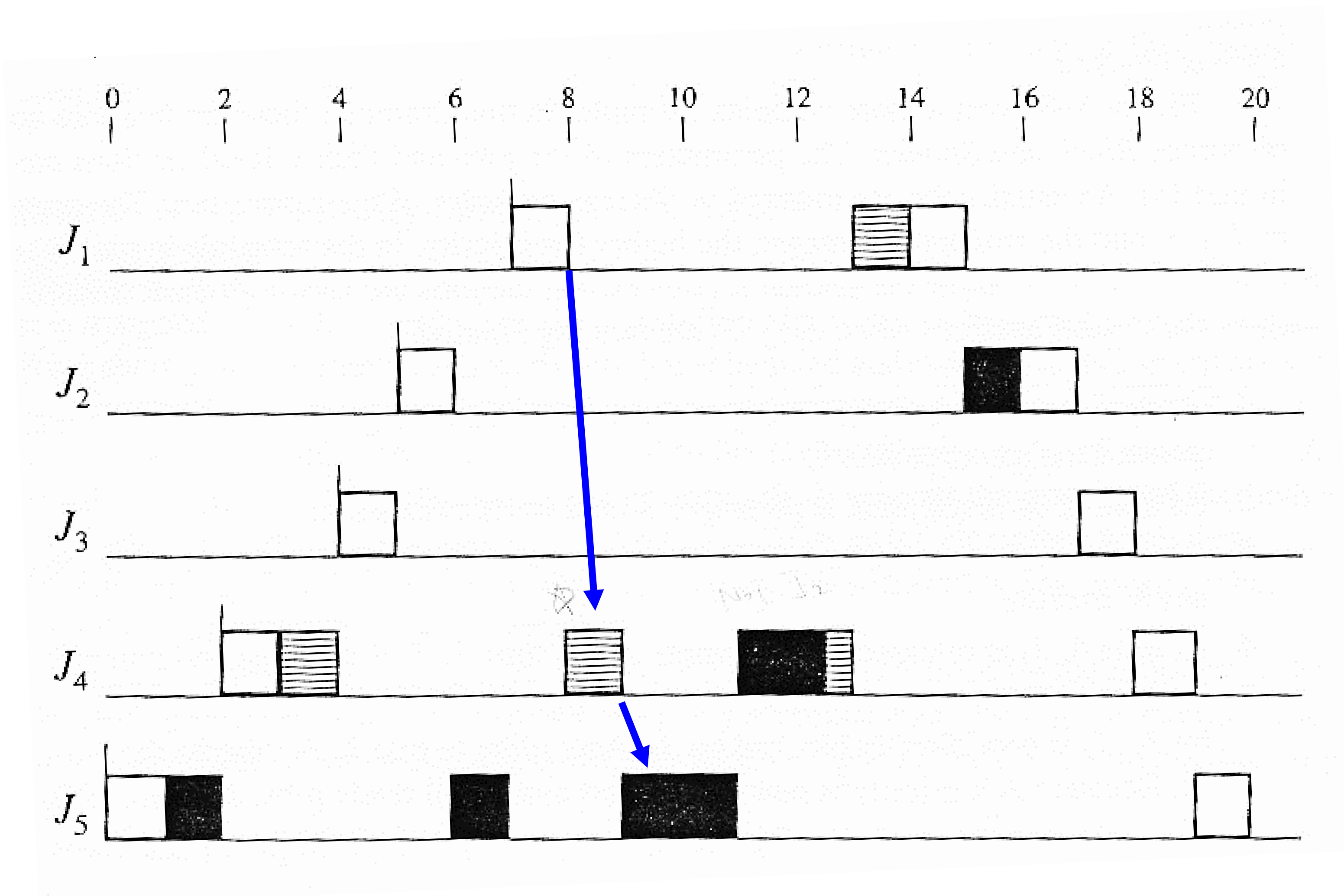
Note: a job may inherits priority from one or more jobs (nested inheritance)

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

(a)

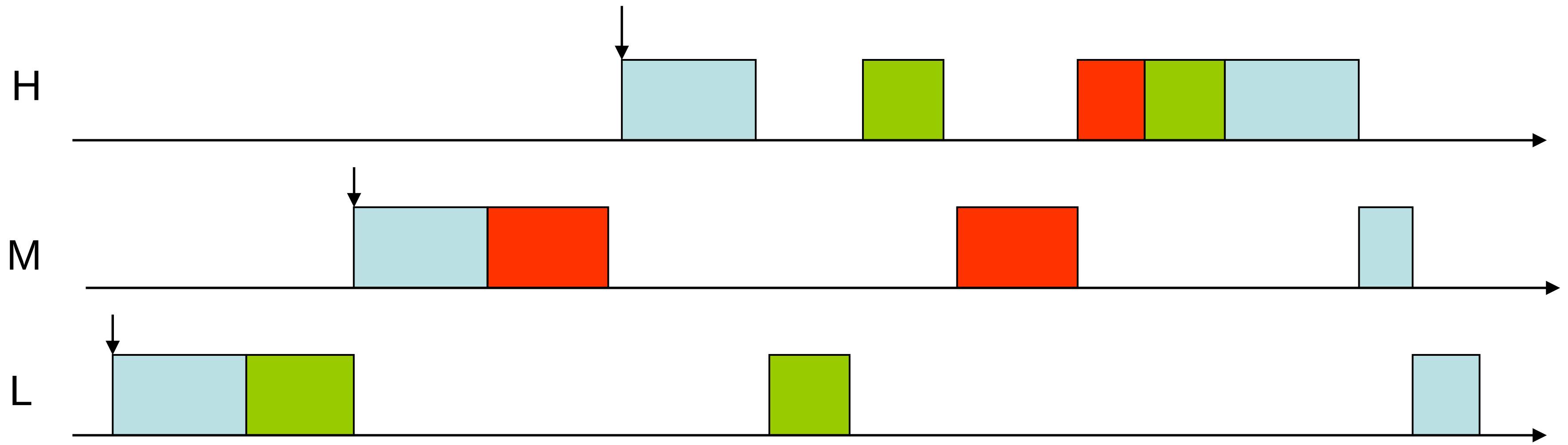


Transitive priority inheritance & Transitive blocking



Priority-Inheritance Protocol

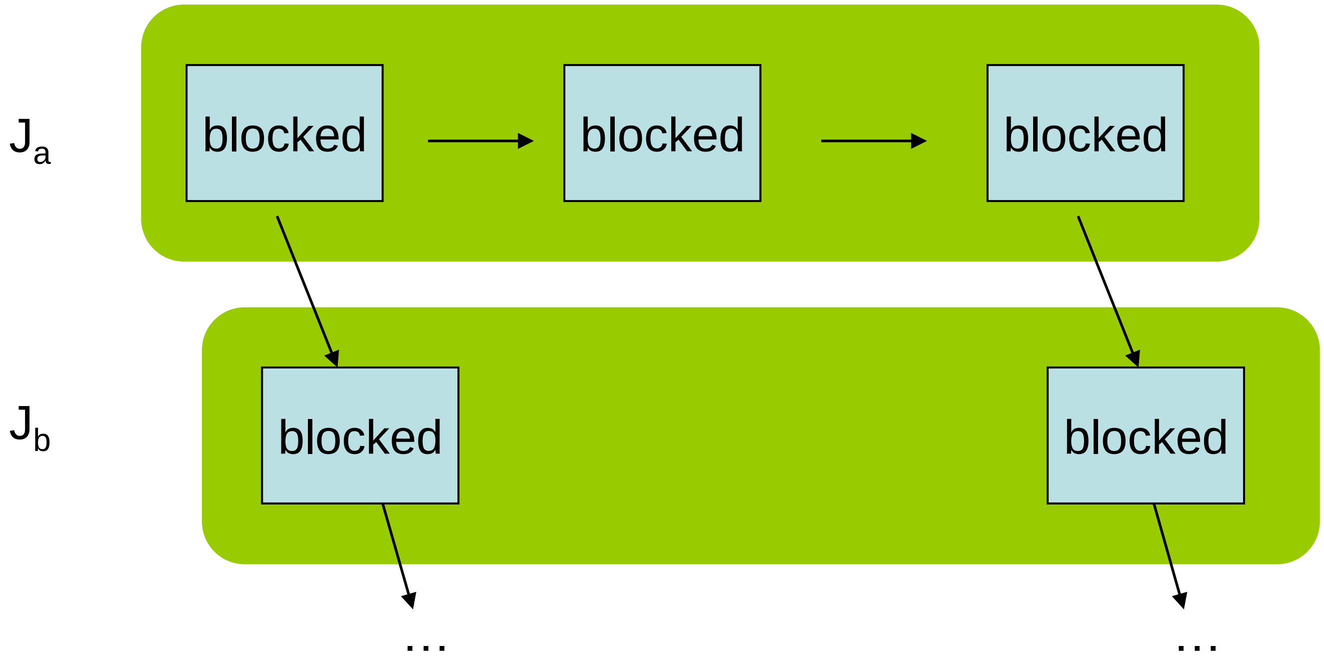
- Chain blocking



- There can be multiple undergoing critical sections under PIP
- There is only one undergoing critical section under NPCS and CPP

Priority-Inheritance Protocol

- Both transitive and chain blocking could occur in PIP



Priority-Inheritance Protocol

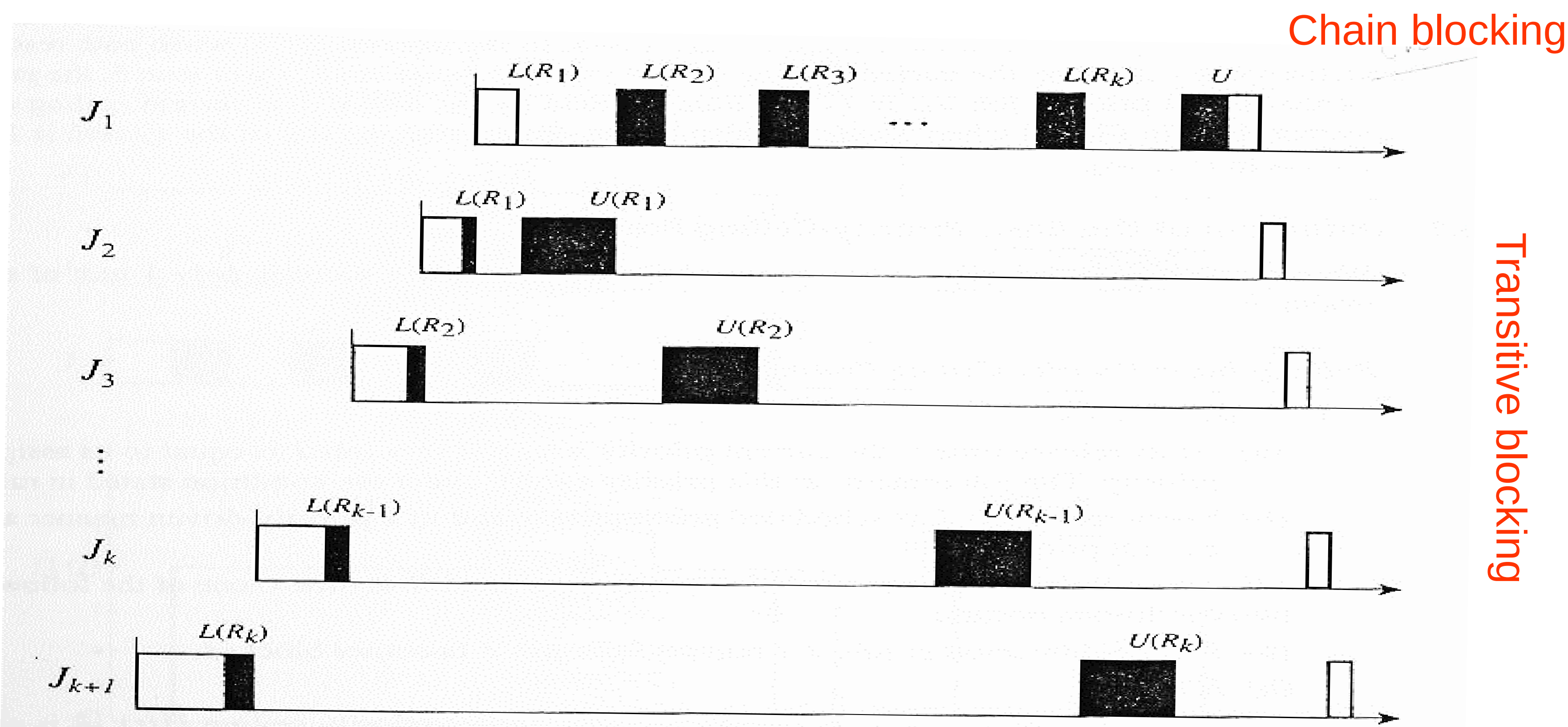
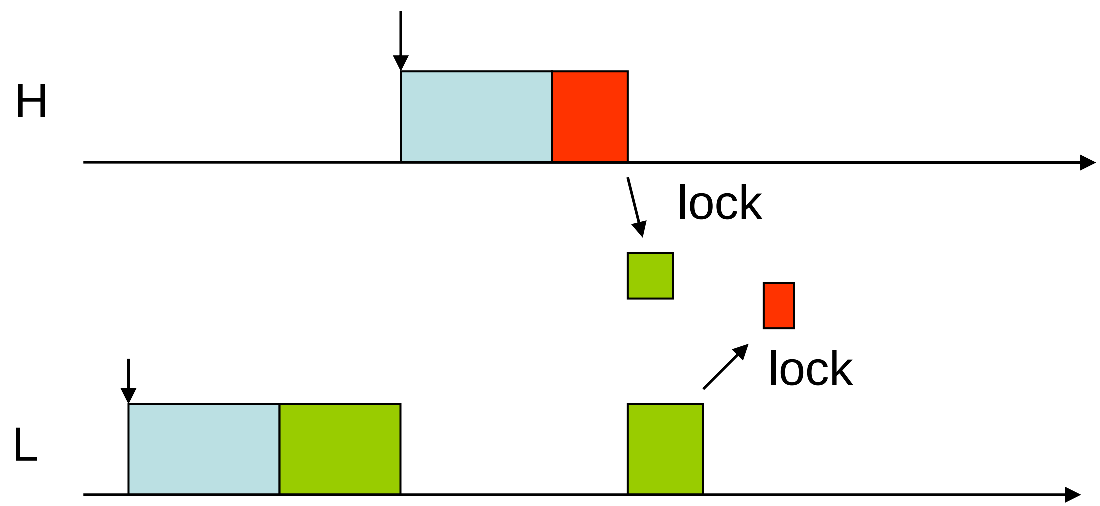


FIGURE 8–9 A worst-case blocking scenario for priority-inheritance protocol.

J1 can be blocked up to min(v,k) times, each of the duration of the outmost CS. v and k stand for different resources J1 requires and the number of low-priority tasks, respectively

Priority-Inheritance Protocol

- Deadlocks



PIP allows hold and wait

Priority-Inheritance Protocol

- Pros:
  - Simple
  - Better response than NPCS and CPP
  - Free from the uncontrollable priority
- Cons:
  - Suffering from deadlocks
  - Chain/transitive blocking

Priority-Ceiling Protocol

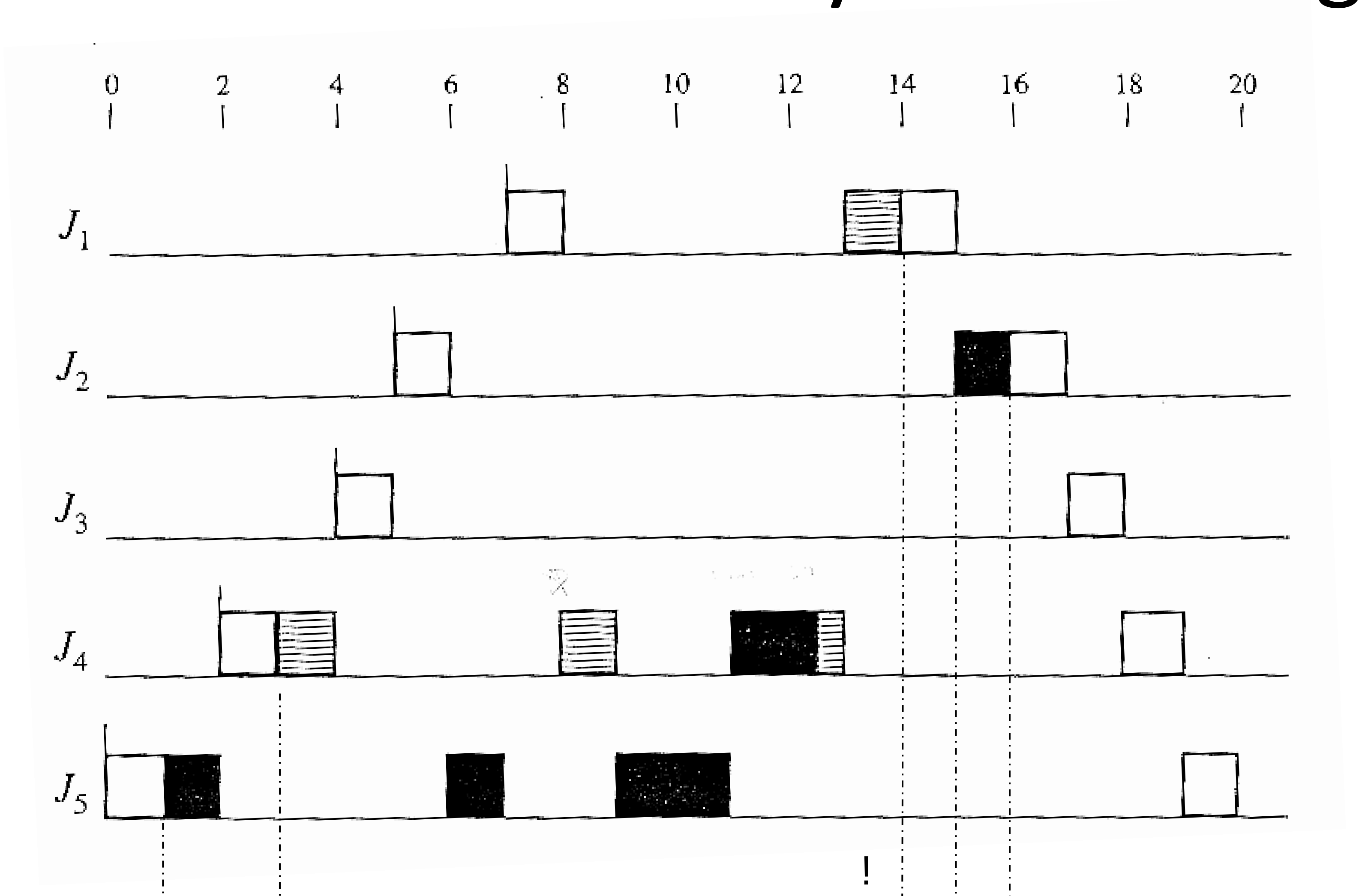
- PCP (fixes) improves upon PIP in terms of
  - Prohibiting chain/transitive blocking
    - To reduce the worst-case blocking time
  - Prohibiting circular waiting
    - To avoid deadlock

Priority-Ceiling Protocol

- Assumptions
  - All tasks have unique and fixed priorities
  - Resource usages of tasks are known *a priori*
- Terms
  - $\Pi(R)$: priority ceiling of resource R
    - The highest priority of all tasks that require R
  - $\Pi^{\wedge}(t)$: current system ceiling at time t (caution!)
    - The highest among the ceilings of all the resources that are *currently in use*

This is PIP, not PCP

Demonstration of System Ceiling



System ceiling

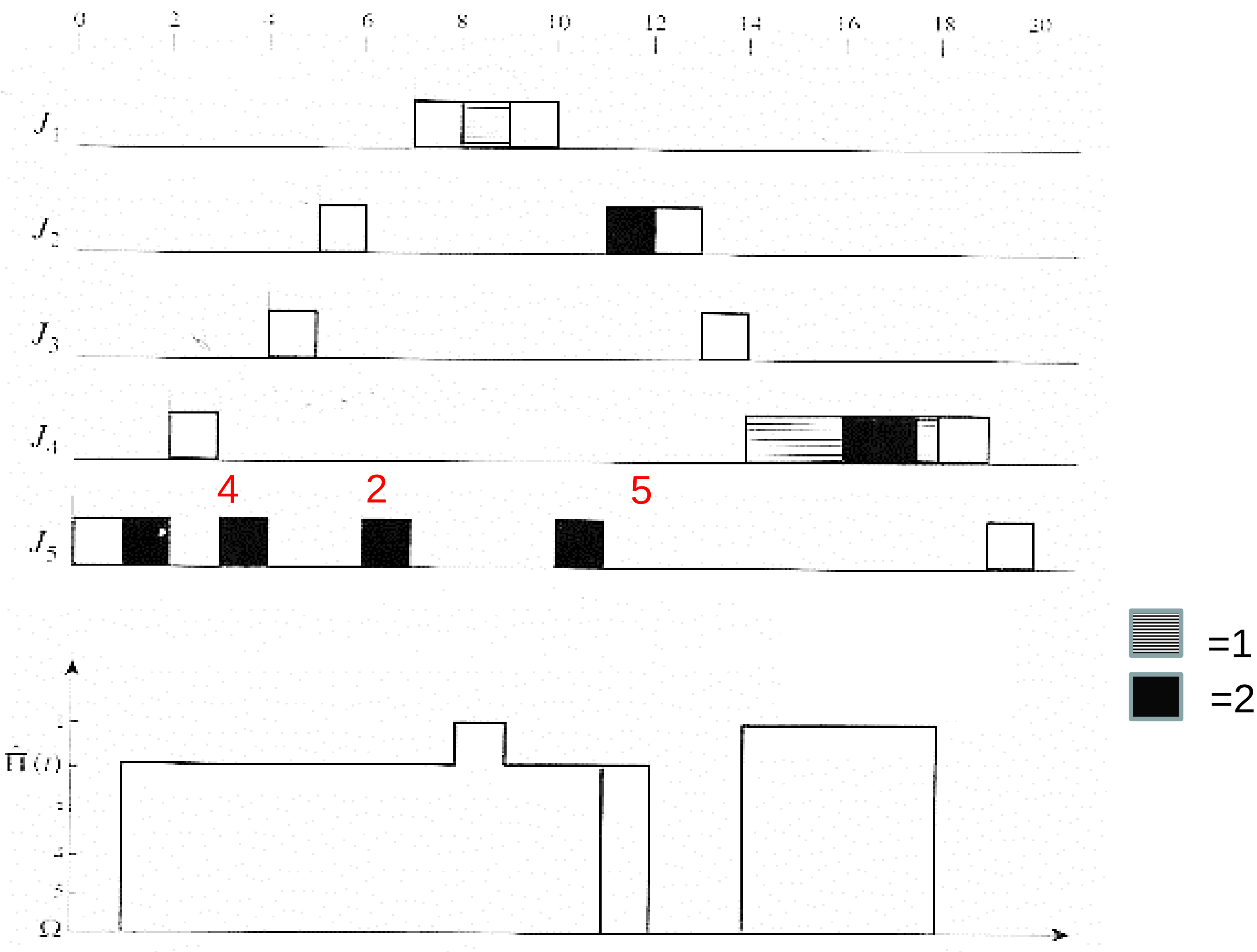
Ω 2 1 Ω 2 Ω

Priority-Ceiling Protocol

Rules of Basic Priority-Ceiling Protocol

1. *Scheduling Rule*:
   (a) At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.
   (b) Every ready job J is scheduled preemptively and in a priority-driven manner at its current priority $\pi(t)$.
2. *Allocation Rule*: Whenever a job J requests a resource R at time t, one of the following two conditions occurs:
   (a) R is held by another job. J's request fails and J becomes blocked.
   (b) R is free.
      (i) If J's priority $\pi(t)$ is higher than the current priority ceiling $\hat{\Pi}(t)$, R is allocated to J.
      (ii) If J's priority $\pi(t)$ is not higher than the ceiling $\hat{\Pi}(t)$ of the system, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\hat{\Pi}(t)$; otherwise, J's request is denied, and J becomes blocked.
3. *Priority-Inheritance Rule*: When 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).

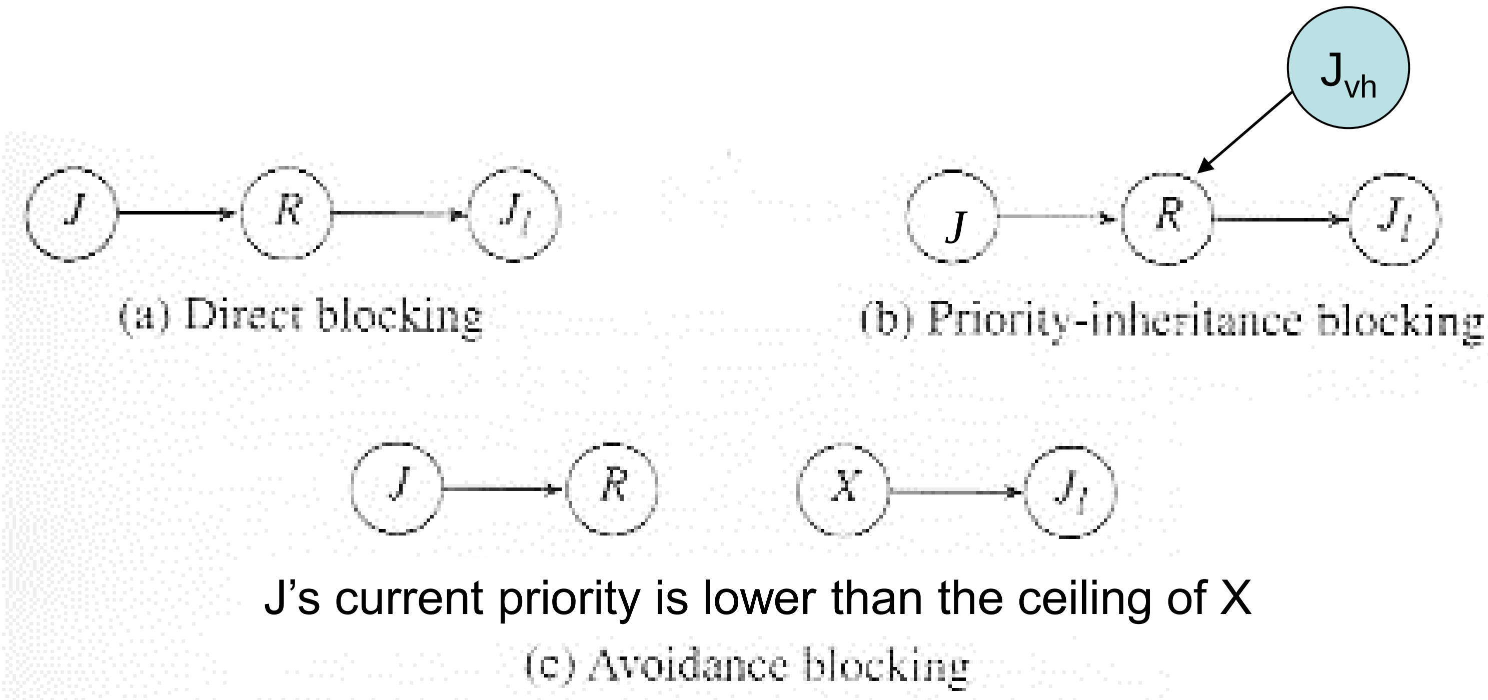
Priority-Ceiling Protocol



Priority-Ceiling Protocol

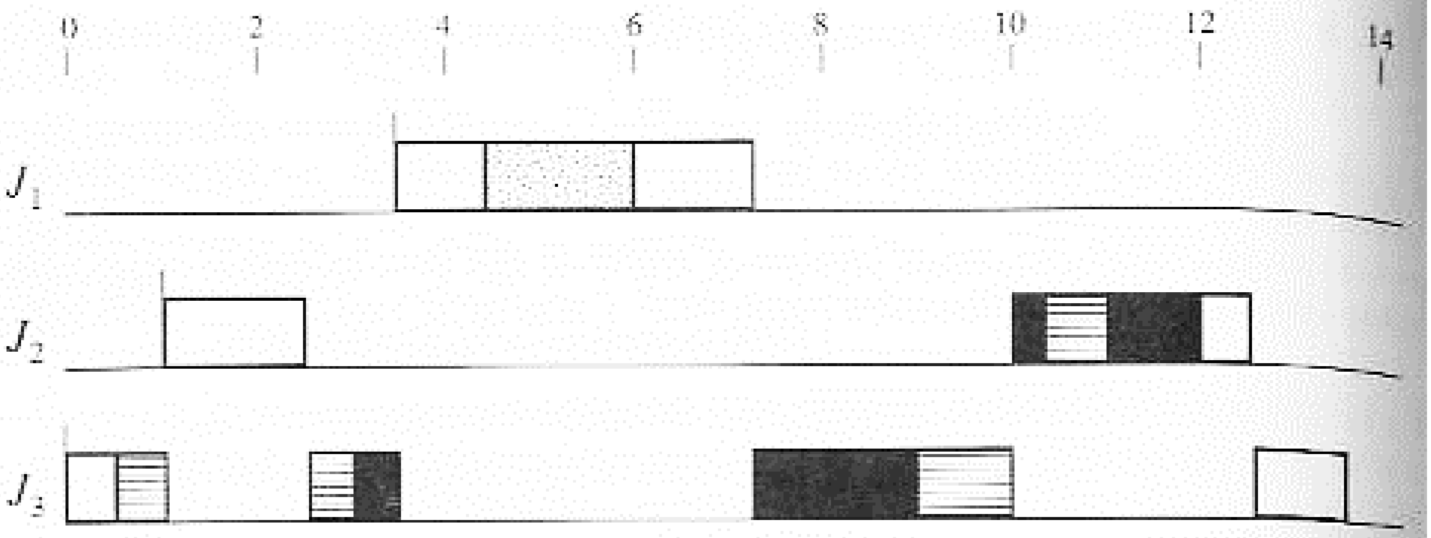
- PCP may postpone a resource request even if the resource is currently available
  - To prevent undesirable effects, specifically, deadlocks and chain/transitive blocking
- There are three types of blockings in PCP
  - Direct blocking
  - Priority-inheritance blocking
  - Avoidance/ceiling blocking (new)

Priority-Ceiling Protocol



Priority-Ceiling Protocol

- PCP avoids uncontrolled blocking
  - Through priority inheritance
- PCP also avoids deadlocks



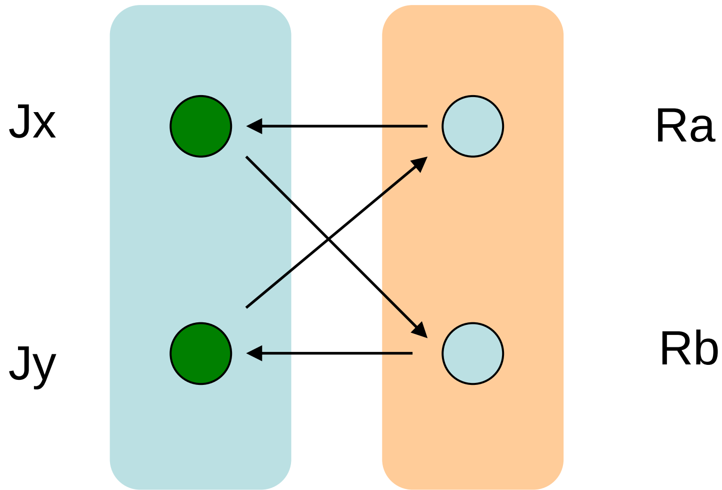
$\Pi(Dotted) = \pi_1$, $\Pi(Black) = \Pi(Shaded) = \pi_2$

FIGURE 8–12 Example illustrating how priority-ceiling protocol prevents deadlock.

Priority-Ceiling Protocol

- PCP avoids deadlocks

  (informal) Proof:

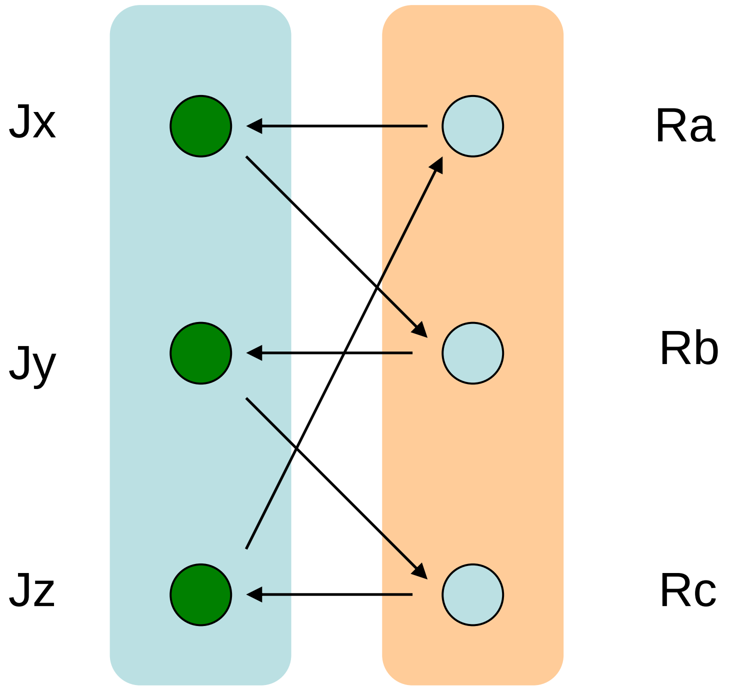


Jx and Jy can not simultaneously lock resources!!

Priority-Ceiling Protocol

- Theorem: PCP avoids deadlocks

  (informal) Proof:



If any one job successfully locks some resources, then at least one of the other jobs can not lock resources

Priority-Ceiling Protocol

- Jobs seem having longer response under PCP than under PIP
  - PCP has priority-ceiling blocking but PIP doesn't
- In fact, the worse-case blocking time of PCP is not longer than PIP
  - No transitive blocking and chain blocking in PCP, tasks are blocked by up to *one* critical section

Priority-Ceiling Protocol

- Theorem: any job governed by PCP can be blocked for at most one critical section

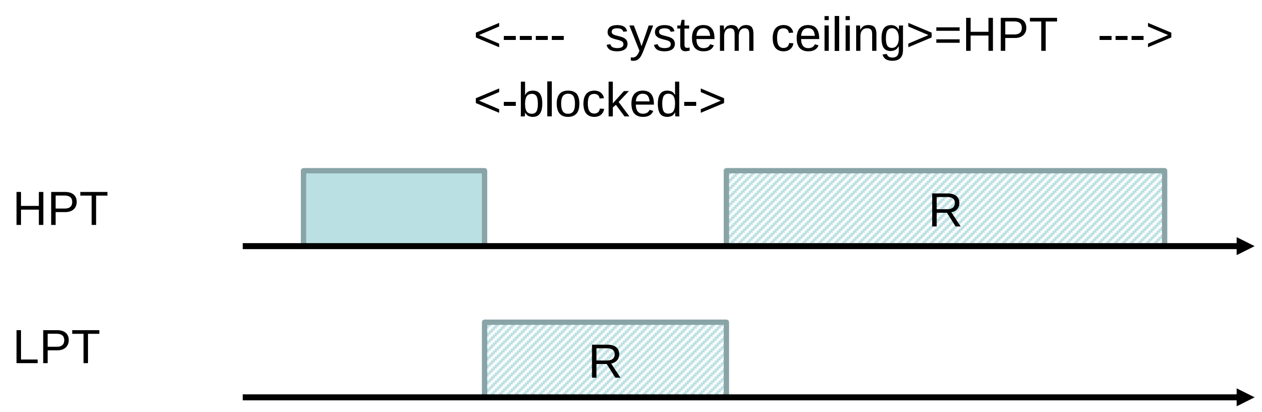
  Proof:

  – A job can be blocked by only once (no chain)

  – No transitive blocking

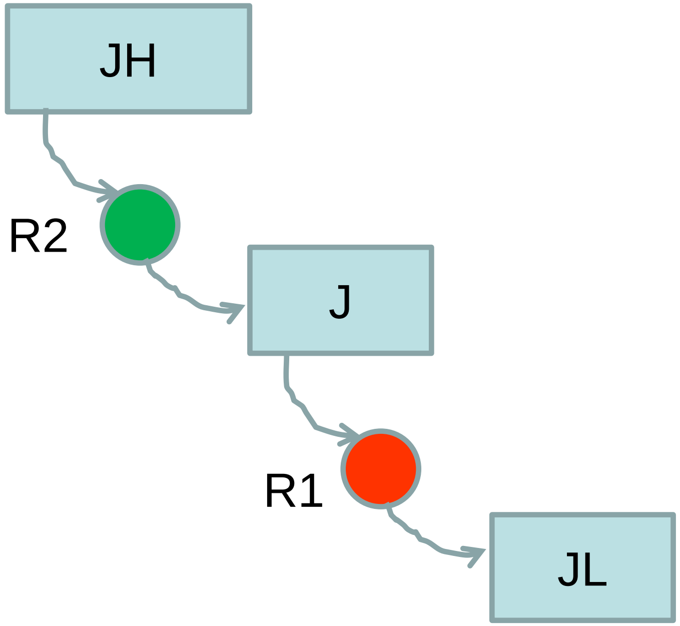
Priority-Ceiling Protocol

- A job can be blocked only once
  - When HPT acquires R, no other tasks < HPT could hold a resource that HPT needs. Otherwise, HPT cannot acquire R (due to system ceiling)



Priority-Ceiling Protocol

- No transitive blocking
  - If JL locks R1 then J cannot lock R2



$J_H \rightarrow J \rightarrow J_L$

Priority-Ceiling Protocol

- Priority inheritance blocking and priority ceiling blocking

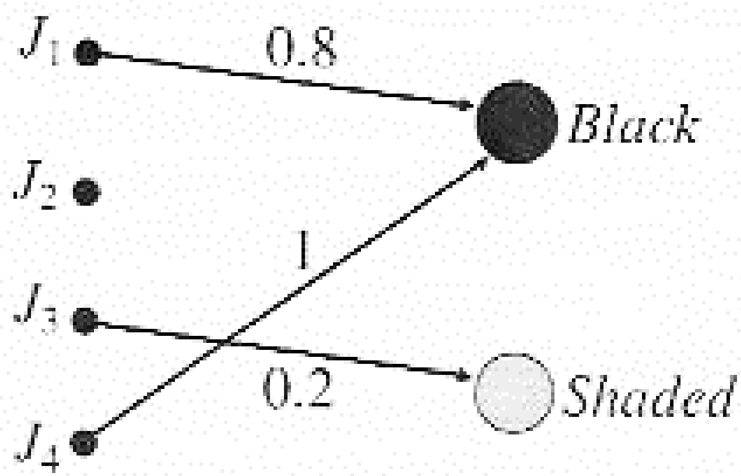


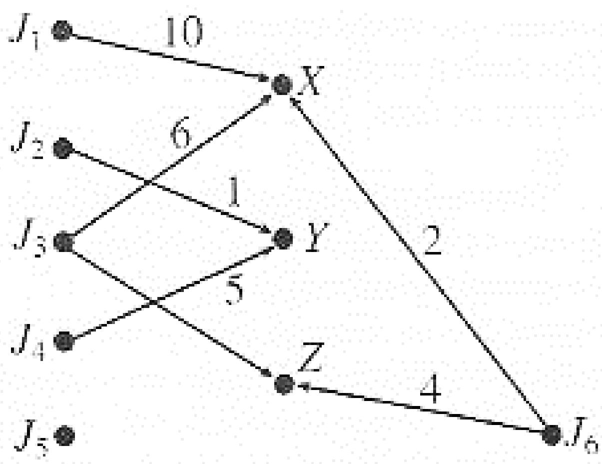
FIGURE 8–14 Example on duration of blocking.

- J4 can block J3 by inheriting a priority from J1
- J4 can block J3 by raising the system ceiling to π(Black)

Different scenarios, but the durations are the same

Priority-Ceiling Protocol

6,6,5,4,4



(a)

	Directly blocked by					Priority-inher blocked by					Priority-ceiling blocked by				
	J_2	J_3	J_4	J_5	J_6	J_2	J_3	J_4	J_5	J_6	J_2	J_3	J_4	J_5	J_6
J_1		6			2										
J_2	*		5			*	6			2	*	6			2
J_3		*			4		*	5		2		*	5		2
J_4			*					*		4			*		4
J_5				*					*	4				*	

max(2,4)

Priority-Ceiling Protocol

- For any job J that requires some resource(s), its priority-inheritance blocking time and its ceiling blocking time are the same
  - PI & PC blockings happen through the same path in the resource allocation graph
  - But remember: ceiling is to manage “locking”, priority is to manage “scheduling”
    - So PI & PC blocking times will be different for tasks not using any resource(s)

Priority-Ceiling Protocol

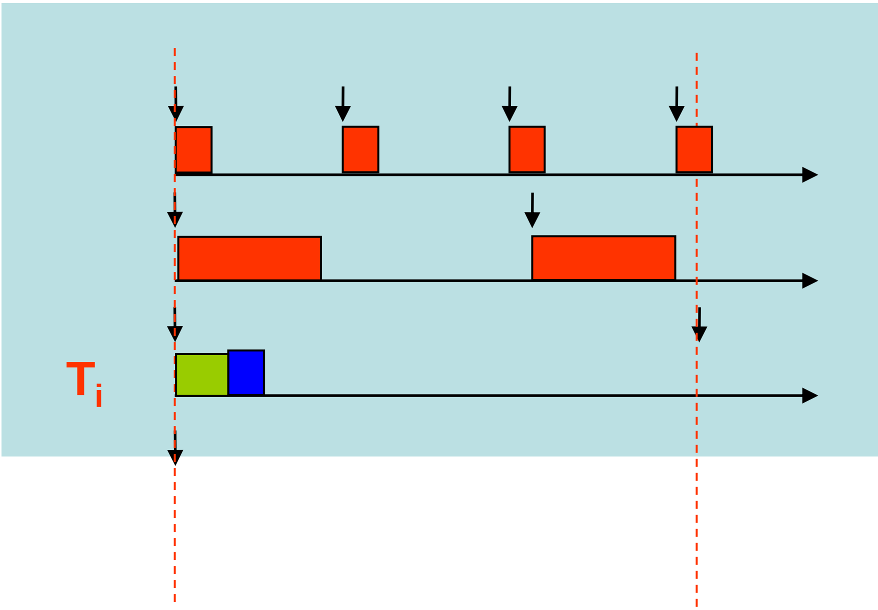
- A collection of tasks $\{T_1, T_2, ..., T_n\}$ governed by PCP are schedulable if

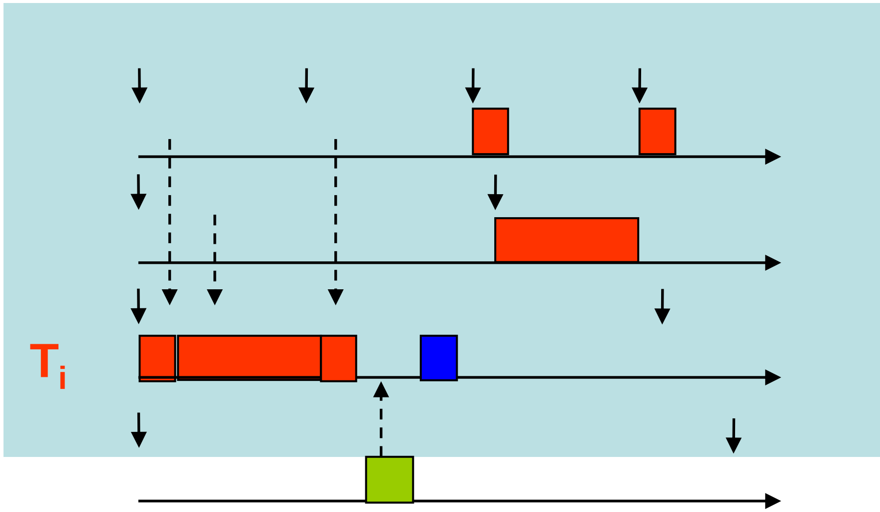
$$\forall_{i=1...n} \left\{ \frac{b_i}{p_i} + \sum_{j=1}^{i} \frac{c_j}{p_j} \leq U(i) \right\}$$

- Where b_i is the longest blocking time imposed on T_i

Priority-Ceiling Protocol

$$\forall_{i=1\ldots n}\left\{\frac{b_i}{p_i}+\sum_{j=1}^{i}\frac{c_j}{p_j}\leq U(i)\right\}$$





The critical instant of T_i (left) with bi included in ci, and (right) with blocking time

- With task blocking time determined, we can also use response time analysis

- What are the tests for NPCS and CPP?
  - The same tests (U bound and rsp time analysis) can be used, as “blocking time” remains the same regardless of the synchronization protocols

Applicability

- So far we are focused on RM, and the following protocols are applicable
  – NPCS, CPP, PIP, and PCP
- For EDF, we previously mentioned that NPCS and CPP can be used as well
  – EDF does not work with PIP and PCP, however
  – A similar but different approach, SRP, is to be discussed

Stack-Resource Policy

- PCP is not directly applicable to EDF, because with EDF there is no task priority, so
  - it is not possible to define ceilings
  - priority inheritance cannot operate
- In EDF, a task preempts longer-period tasks
- So "preemption levels" can be defined to be inversely proportional to task periods
  - It replaces the notion of "task priority"

Stack-Resource Policy

- Resource/system ceiling
  - Use preemption levels
- Inheritance
  - Use deadlines
- Scheduling (!)
  - A task cannot execute until its preemption level is higher than the current system ceiling

Stack-Resource Policy

Rules of Basic Stack-Based, Preemption-Ceiling Protocol

0. *Update of the Current Ceiling*: Whenever all the resources are free, the preemption ceiling of the system is Ω. The preemption ceiling $\hat{\Psi}(t)$ is updated each time a resource is allocated or freed.
1. *Scheduling Rule*: After a job is released, it is blocked from starting execution until its preemption level is higher than the current ceiling $\hat{\Psi}(t)$ of the system and the preemption level of the executing job. At any time t, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.
2. *Allocation Rule*: Whenever a job J requests for a resource R, it is allocated the resource.
3. *Priority-Inheritance Rule*: When some job is blocked from starting, the blocking job inherits the highest priority of all the blocked jobs.

Here, "priority" stands for "deadline", and preemption levels reflect task periods!!

- PCP: check system ceiling on locking
- SRP: check system ceiling on scheduling

Preemption levels

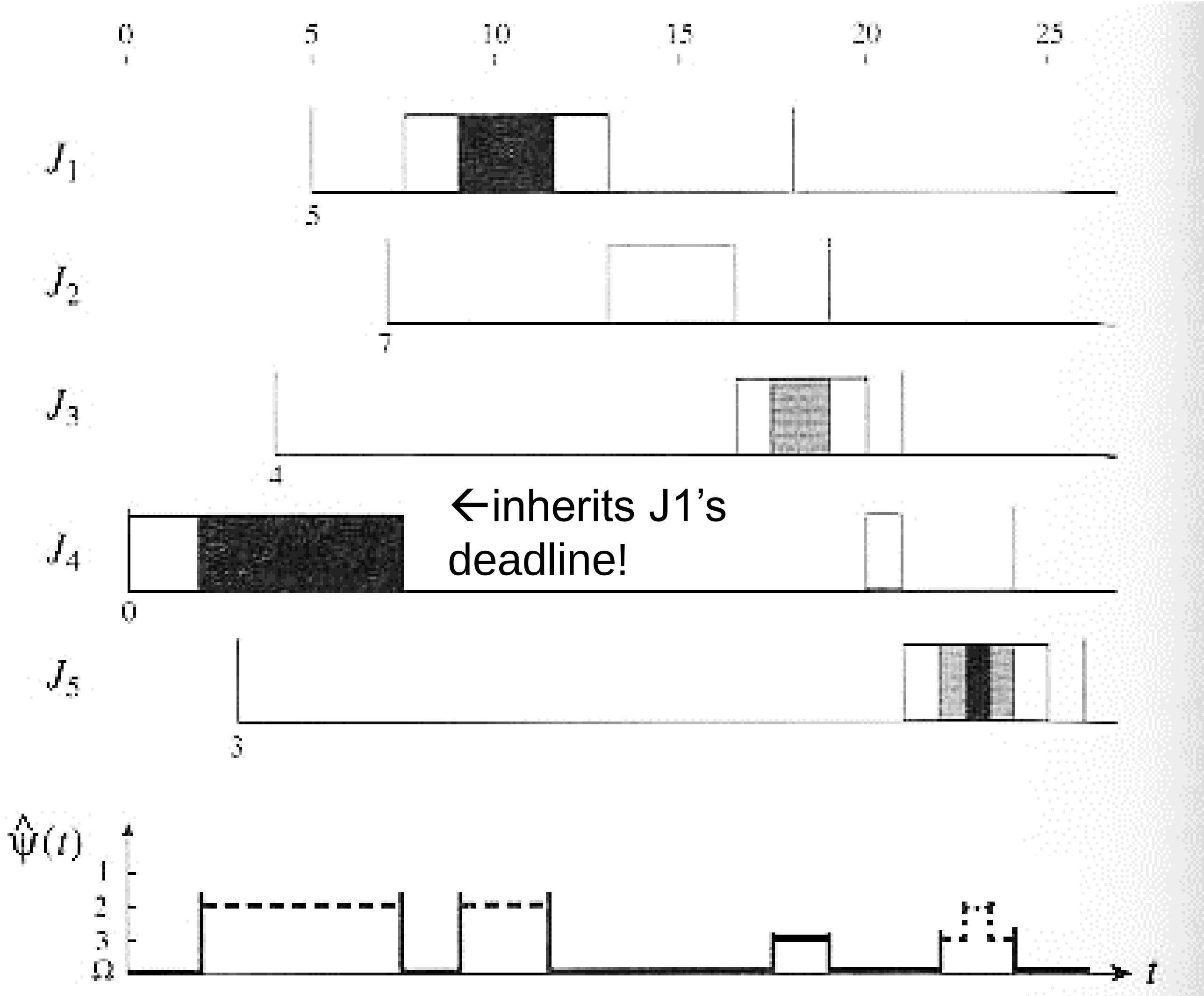
2

1

3

5

4



Black=2
Gray=3

FIGURE 8–20 Example of priority ceilings of ~~multiple unit~~ resources.

Stack-Resource Policy

- SRP avoids uncontrolled priority inversion
  - Protected by deadline-inheritance
- SRP avoids transitive blocking and chain blocking
  - When a job start executing, all that resources it needs have been released
  - Governed by system ceiling
- SRP prevent deadlocks from happening
  - A job is never be blocked once it start executing

Stack-Resource Policy

- Blocking time
  - Direct blocking (?)
    - No blocking on locking
  - Deadline-inheritance blocking
    - Blocked on scheduling
  - Preemption-ceiling blocking
    - Blocked on scheduling
- Use a similar technique for PCP/RM to calculate task locking times of SRP/EDF

Stack-Resource Policy

- A collection of tasks $\{T_1, T_2, ..., T_n\}$ governed by SRP are schedulable by EDF if

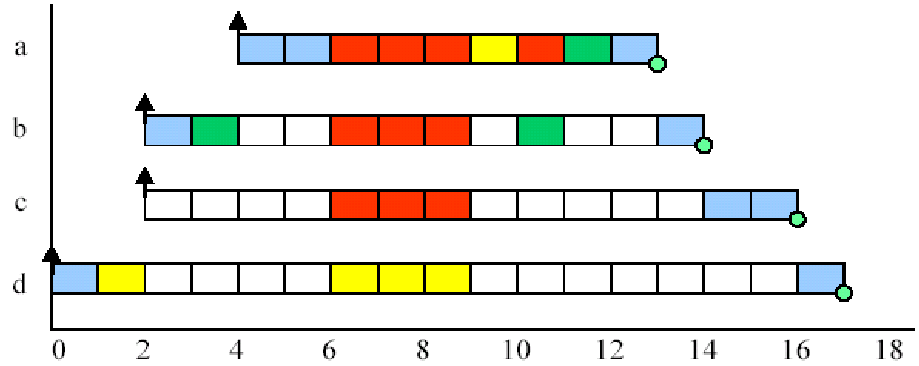
$$\forall_{i=1}^{n} \left(\frac{b_i}{p_i} + U \right) \leq 1$$

- U = the total CPU utilization of all tasks
- The same test applies to NPCS and CPP

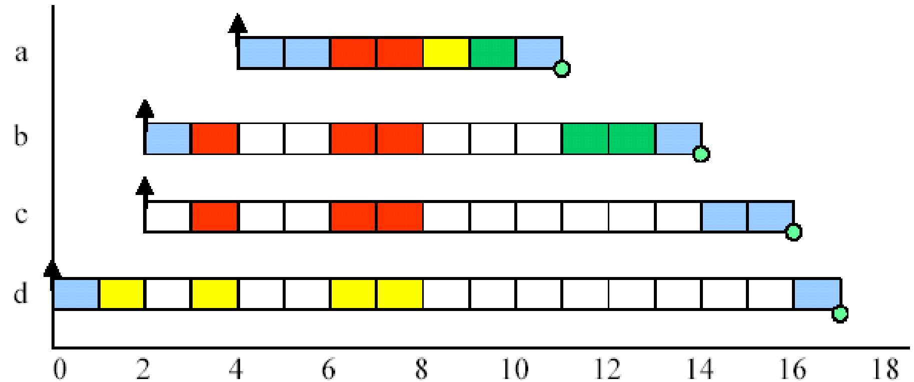
Stack-Resource Policy

- Pros:
  - Free from indirection priority inversion and deadlocks
  - Easy to implement, works with EDF
- Cons:
  - Response of HPTs is bad

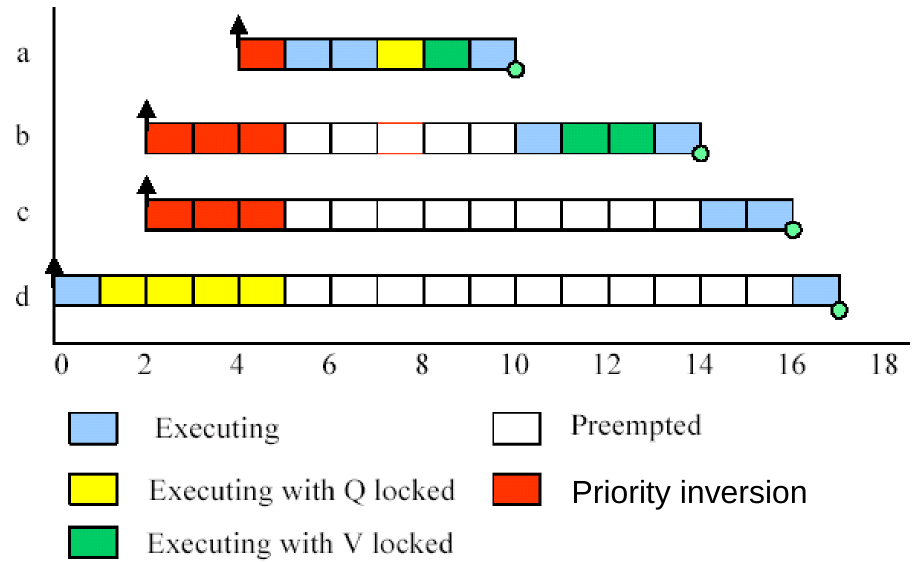
Ungoverned



PCP+RM



SRP+EDF



	Uncontrollable priority inversion	Bounded blocking time	Blocked at most once	Deadlock avoidance
Non-preemptible Critical Sections **(NPCS)**	No	Yes	Yes	Yes
Ceiling-Priority Protocol **(CPP)**	No	Yes	Yes	**Yes**
Priority-Inheritance Protocol **(PIP)**	No	Ouch	No	No
Priority-Ceiling Protocol **(PCP)**	No	Yes	Yes	Yes
Stack-Resource Policy **(SRP)**	No	Yes	Yes	Yes

Summary

- Check list
  - Priority inversion
  - Blocking time
  - Blocking duration
  - Deadlock
  - Response

- Backup slices…

Priority-Ceiling Protocol

- Context-switch for independent tasks
  - Two context switches are accounted to the preempting job
  - Thus every job is added to 2*CS
- With PCP, a job can be blocked for at most once
  - Additional 2*CS is accounted to a job with a non-zero blocking time

Priority-Ceiling Protocol

- A collection of tasks $\{T_1, T_2, \ldots, T_n\}$ governed by PCP are schedulable if

$$\forall_{i=1\ldots n} \left\{ \frac{b_i}{p_i} + \sum_{j=1}^{i} \frac{f(j) \times CS + c_j}{p_j} \leq U(i) \right\}$$

- f(x)=4 if b_x !=0
- f(x)=2 if b_x == 0

Stack-Resource Policy

- Preemption ceilings of a multi-instance resources R increases as the free units in R decreases

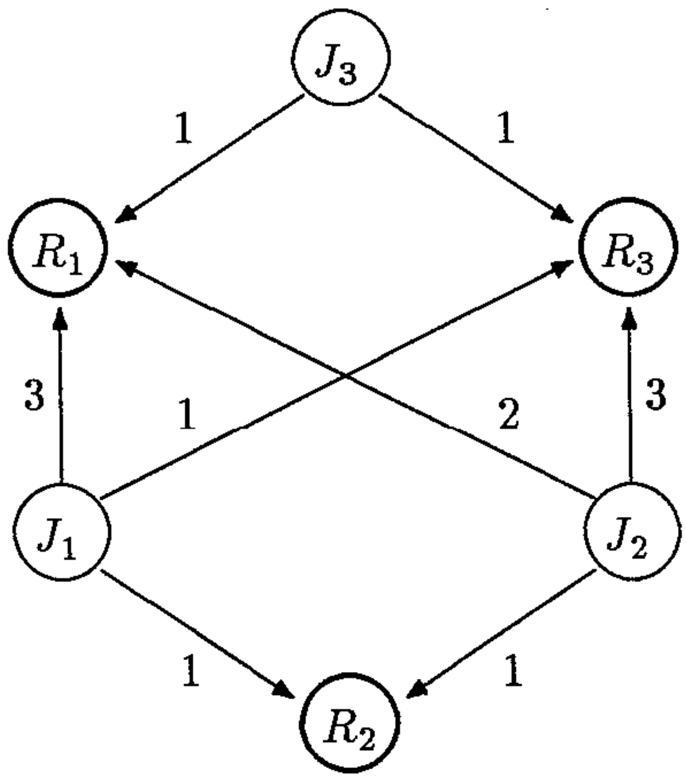
- Let $\ulcorner R \urcorner_k$ be the priority ceiling (preemption ceiling) of resource *R* there are *k* units available
  - $\ulcorner R \urcorner_k$ is the highest preemption level of all tasks that require more than k units (>) of resource R
  - These task jobs may be blocked when trying to lock resource R

	Avail	Need						
R	N_R	J1	J2	J3	$\lceil R \rceil_0$	$\lceil R \rceil_1$	$\lceil R \rceil_2$	$\lceil R \rceil_3$
R_1	3	3	2	1	3	2	1	0
R_2	1	1	1	0	2	0	0	0
R_3	3	1	3	1	3	2	2	0

Figure 4: Ceilings of Resources.

Job 1 need 3 units of R1

$\lceil R1 \rceil_2$: Ceiling of R1 when there are two residual instances of R1



Preemption levels:
J3 highest
J1 lowest

Stack-Resource Policy

- With PCP+RM, a task may be blocked (for once) in the middle of execution
  - Extra two units of the cxtsw overhead
- With SRP+EDF, a task will not be blocked once it starts execution
  - Last-In First-Out, this is why SRP is named after "stack" resource policy