Resource Synchronization Protocols

Embedded OS Implementation

Prof. Ya-Shu Chen
National Taiwan University
of Science and Technology

Motivation

- Access control to resources are applied to secure the validity
  - To avoid race conditions
- A resource can be shared by a number of tasks
  - Which task should be granted for access and which should not be granted?
    - The waiting time must be controllable and predictable
  - Resource sync protocol v.s. resource scheduling

Resources

- Let us consider passive resources only
  - A task is still busy on CPU when it is using a passive resource
  - They do not suspend themselves on CPU
  - E.g., memory protected by semaphores
- Active resources are not considered here
  - A task may becomes idle on CPU and switch to an active resource
  - E.g., I/O devices
  - Active resources, of course, can be modeled as passive resources with loss of parallelism

Resources

- Consider there are ρtypes of serially reusable resources $R_1, R_2, \ldots, R_n$
- For some R_i, there are v_i indistinguishable units of resource
  - A read lock may permit many readers
  - A write lock allows only one writer
    - And also prohibit any other readers

Resources

- When a job attempts to use a resource, we say that it tries to lock it
  - The lock may not be granted immediately
  - The task becomes waiting until the lock is granted
    - A lock is not granted because of 1) there are no such kind of resources available, or 2) to prevent some undesirable effects

- When a job no longer needs a resource, it unlocks it
  - An unlock is immediately granted, naturally

Resources

- Let the duration between the lock and unlock to a resource be a critical section
- Let critical sections be properly nested

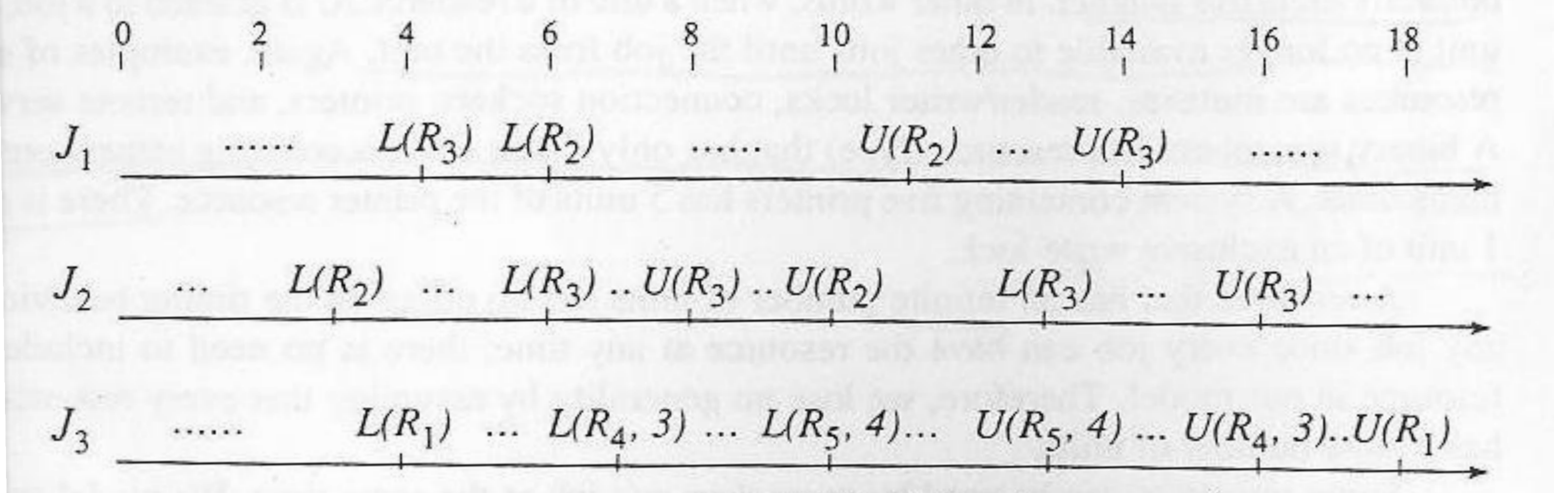


FIGURE 8–1 Examples of critical sections

Resources

白色沒有資源使用
黑色使用資源中

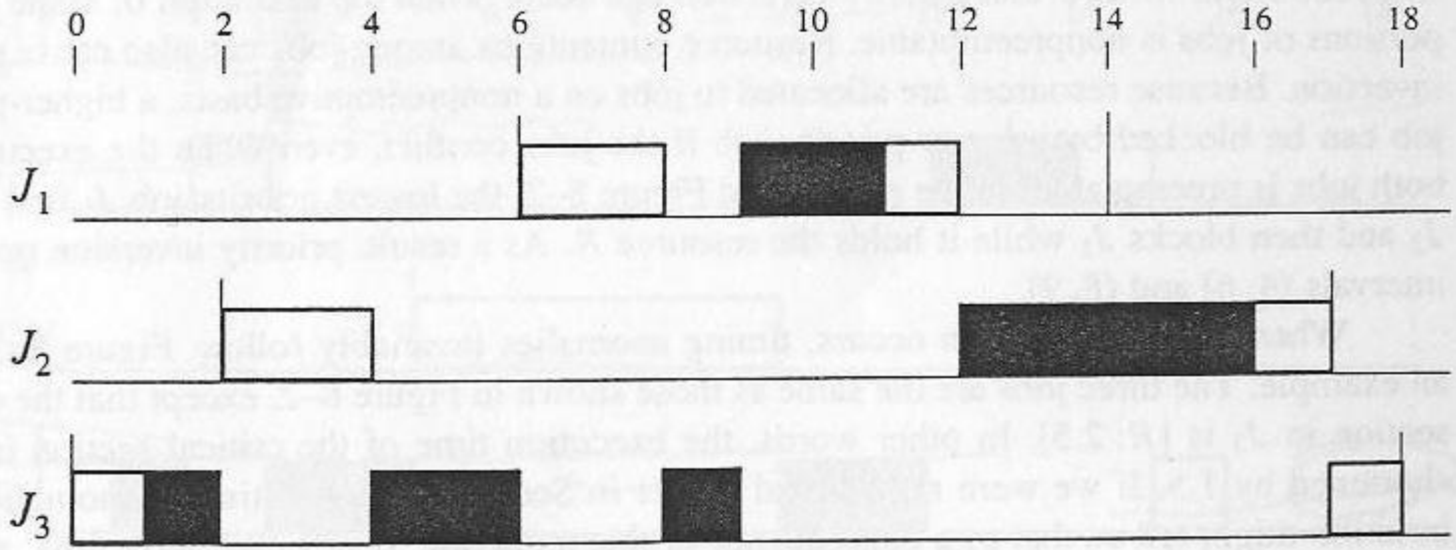


FIGURE 8–2 Example of job interaction due to resource contention.

- Resource usages may postpone the completion of tasks
  - If J1 and J2 do not require resources, they complete by 11 and 14, respectively

Resources

- A job is said to be directly blocked by another job if it loses in the contending for some resource

- Use wait-for graph to describe run-time resource usage

- Use a bipartite graph to describe resource requirements

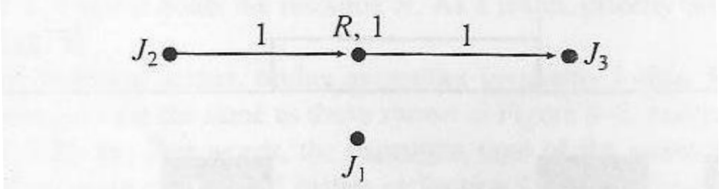
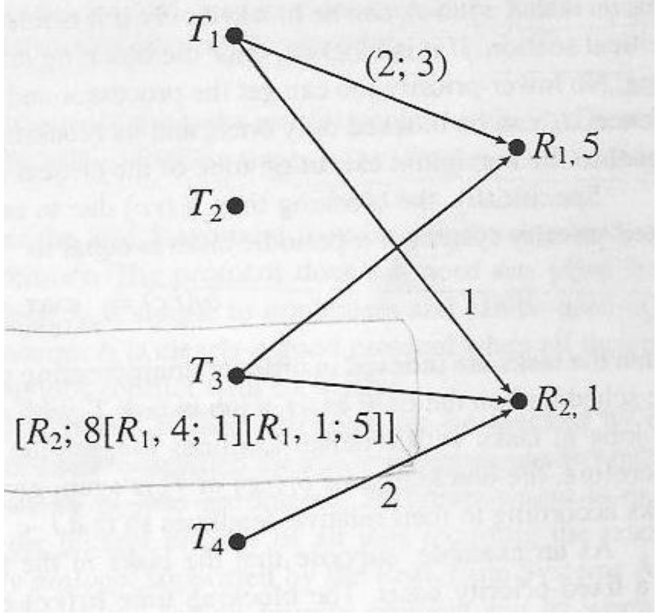


FIGURE 8–5 A wait-for-graph of the system in Figure 8–2 at time 5.



Undesirable Effects

- Priority inversion
  - A high-priority job can not run because of the interference from a low-priority job
    - If the high-priority job tries to lock a resource that is currently locked by another low-priority job
  - In this case, the high-priority job is said to be blocked
    - Note: a low-priority job never be “blocked” by high-priority jobs (why?)

Undesirable Effects

跨優先權
因為2做完給3
但3被1鎖
2被卡住了

- Priority inversion
  – Priority inversion could damage timeliness!
    - A job may be blocked by another job, even if the two jobs do not have any direct resource contention!!

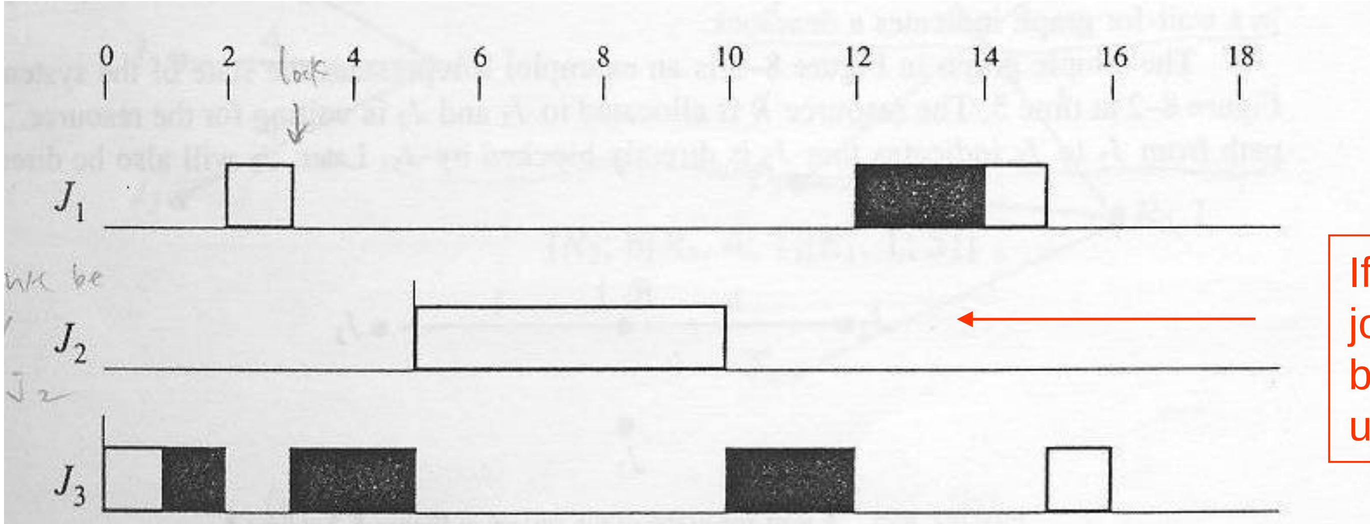


FIGURE 8–4 Uncontrolled priority inversion.

If there are many such jobs, how long J1 is blocks becomes unpredictable!

Undesirable Effects

- Timing anomalies
  - Ideally, if the processing of a job (or a part of it) is accelerated, all jobs in the system enjoy shorter (or equal) response time
  - However, it is not true if jobs contend for resources

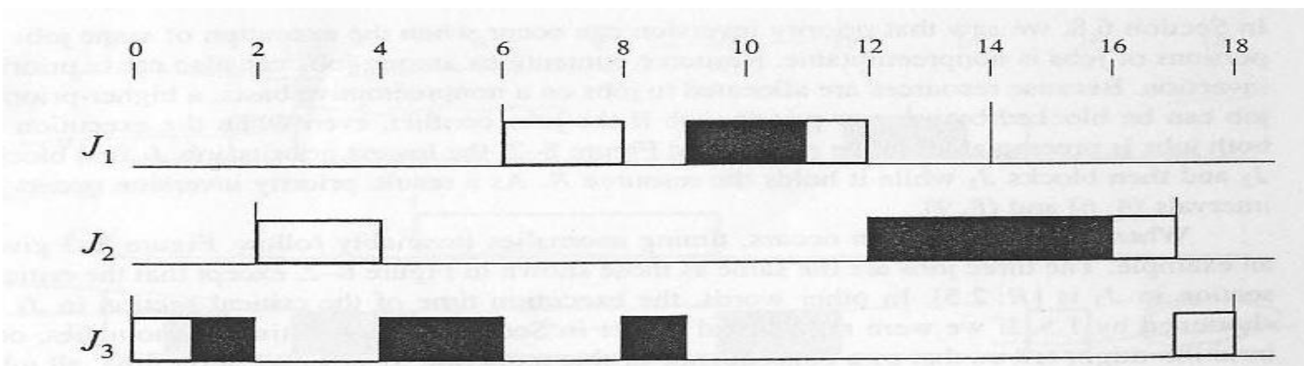


FIGURE 8–2 Example of job interaction due to resource contention.

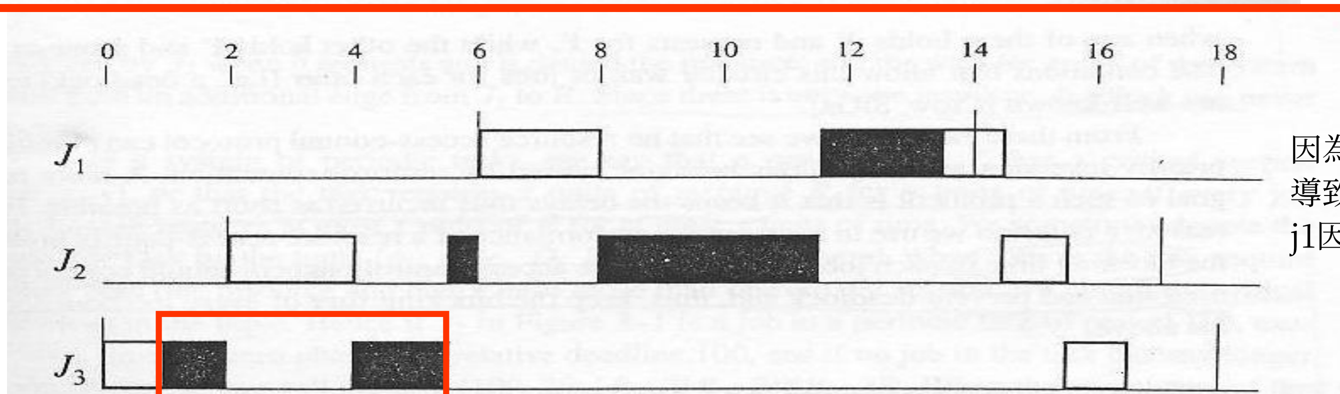


FIGURE 8–3 Example illustrating timing anomaly.

因為j3 更短了
導致j2 黑色現做
j1因為等j2 missdeadline

Now the duration of this CS becomes shorter…

Undesirable Effects

- Deadlocks
  - The most undesirable effect to real-time systems
  - If jobs are in circular waiting, then deadlocks occur
  - Must find a way to impose a partial order on run-time resource access

互相鎖住 互相等待對方

critical
sched lock

Undesirable Effects

- Priority inversions and timing anomalies are spontaneous if there are accesses to resources
  - Can durations for priority inversions be bounded?
  - Can timing anomalies be controllable?
- Deadlocks must be avoided
  - A system suffers from deadlock is inherently incorrect!

Resource Synchronization Protocols

- A set of rules to
  - grant a lock to a resource
  - schedule jobs that use resources

Non-Preemptible Critical Section

- [Mok] NPCS: If a job is using a resource, it won’t be preempted by any other jobs 資源
  - Even if there are no resource conflicts 即使沒有資源衝突
  - Effectively the job runs at the highest priority

- Deadlocks never occur because any job holding resources can not be preempted
- Uncontrollable priority inversions never occur (why?)

Non-Preemptible Critical Section

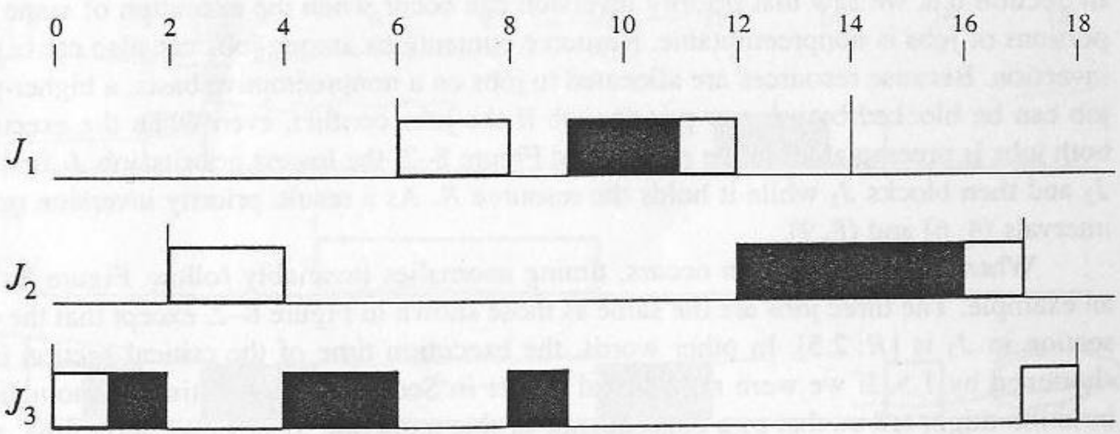
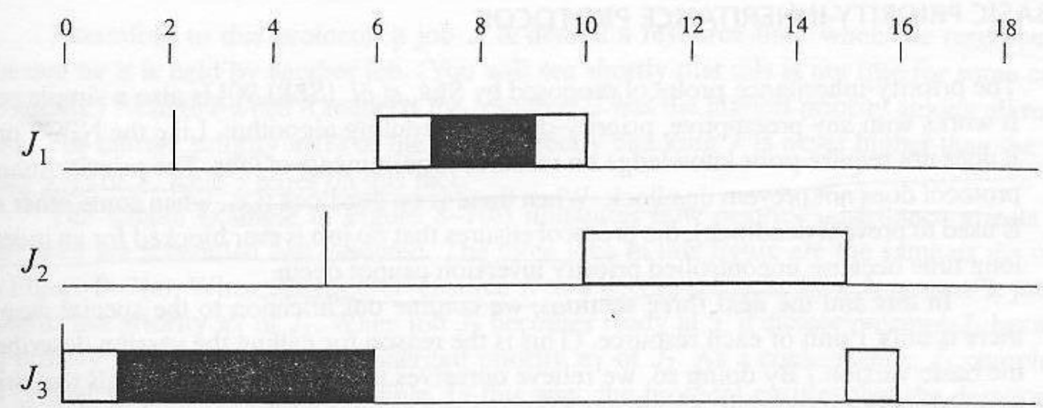


FIGURE 8–2 Example of job interaction due to resource contention.

←Ungoverned

NPCS→

關閉 sched
平行度較差



(a)

**Not the same system :P

Non-Preemptible Critical Section

- Pros:
  - Easy to implement
  - No need to know resource usage *a priori*
- Cons:
  - Low concurrency
  - Poor responsiveness

Non-Preemptible Critical Section

- Let tasks in $\{T_1, T_2, ..., T_n\}$ be sorted in the rate-monotonic order and scheduled by RMS
- The longest blocking time suffered by Ti because of resource contention is

以前最高不用考慮別人
現在有可能被擋住

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

- C_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 suffered by Ti because of resource contention is

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

跟RMS算出來結果一樣
會Preempt一定是週期比我短的
才會發生

- Still the same (why ?)

Schedulability tests?

Priority-Inheritance Protocol

- NPCS is too restrictive, it prevents high-priority jobs from executing even though they have no resource conflicts
  - CPP is restrictive too, jobs may be blocked before they try to lock resources
- PIP relieve such restriction under some certain conditions
  - uncontrollable blocking must not occur

Priority-Inheritance Protocol

繼承priority

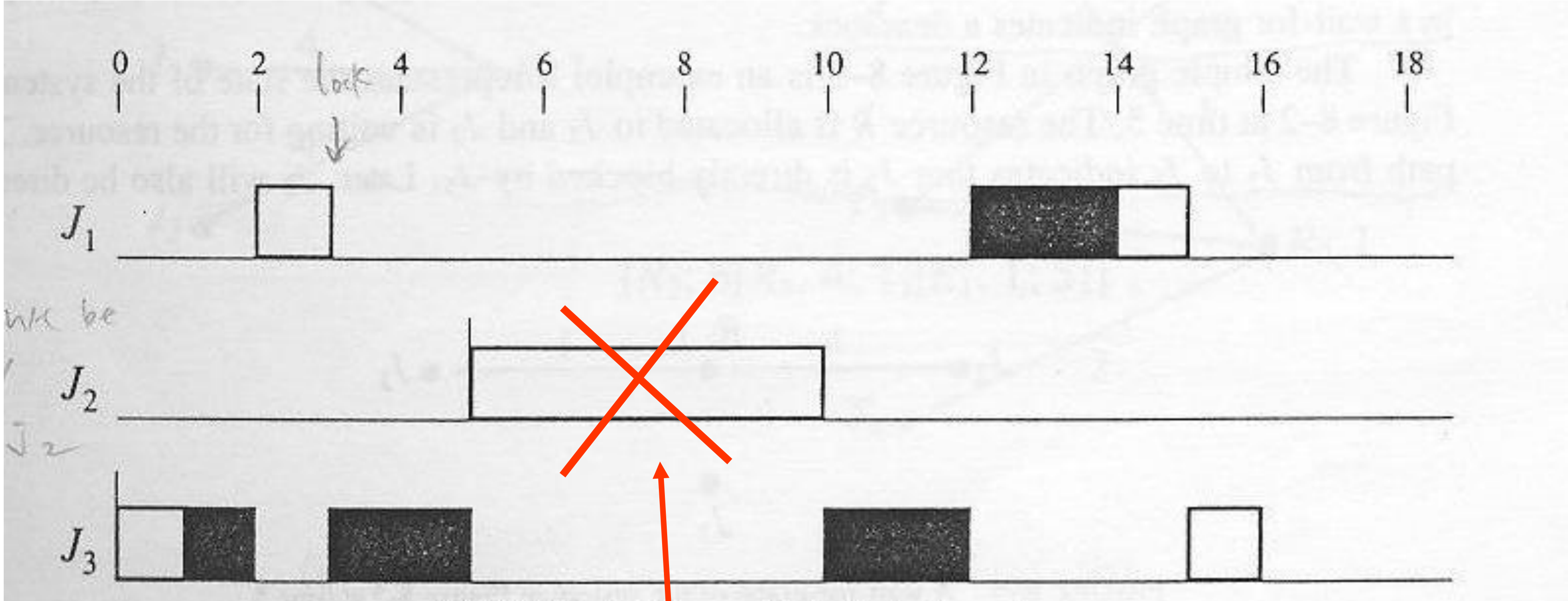


FIGURE 8–4 Uncontrolled priority inversion.

This job should not run because a higher-priority job is already waiting!!

一直檢查J1
如果有J1就不讓J2做

Can J3 inherits J1’s priority? Since J3 blocks J1, logically in the system there is a high-priority job to be executed

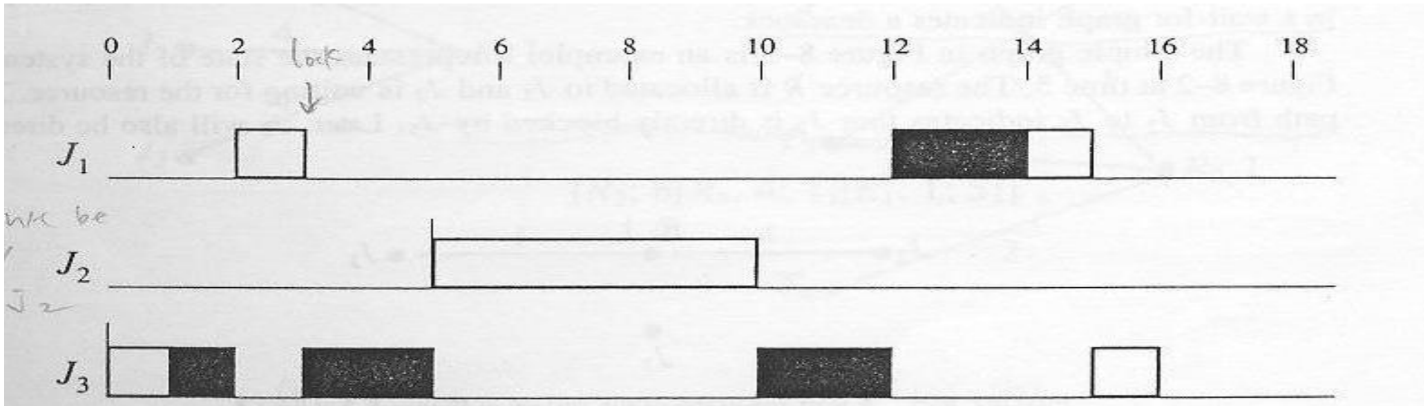
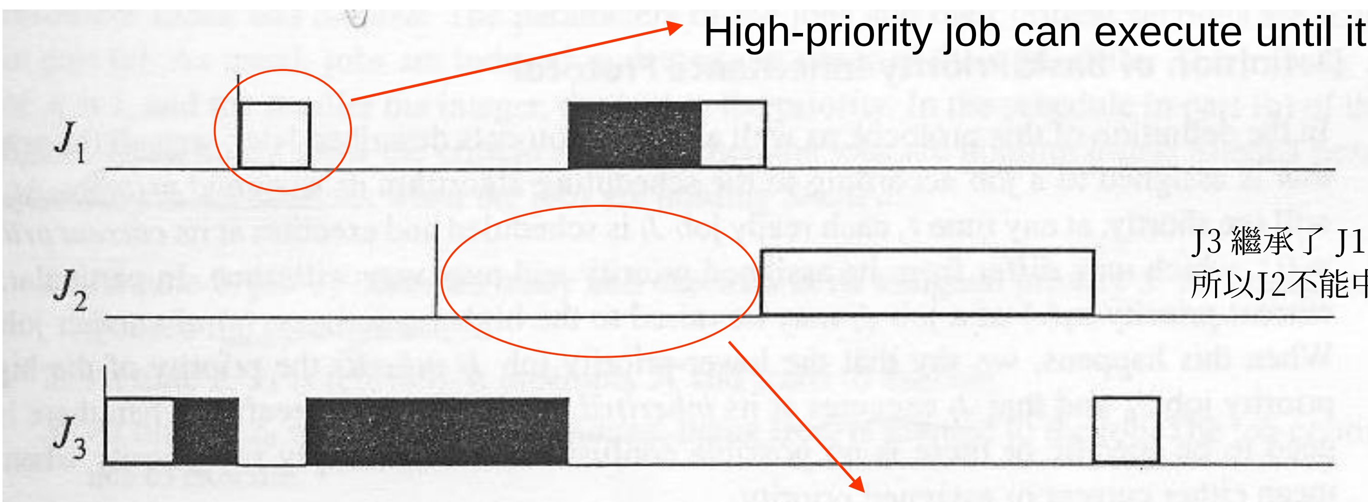


FIGURE 8–4 Uncontrolled priority inversion.



(b)

High-priority job can execute until it is blocked

J3 繼承了 J1的優先權
所以J2不能中止他

Medium-priority job is prevented from executing since a high-priority job is blocked

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.

以[illegible]

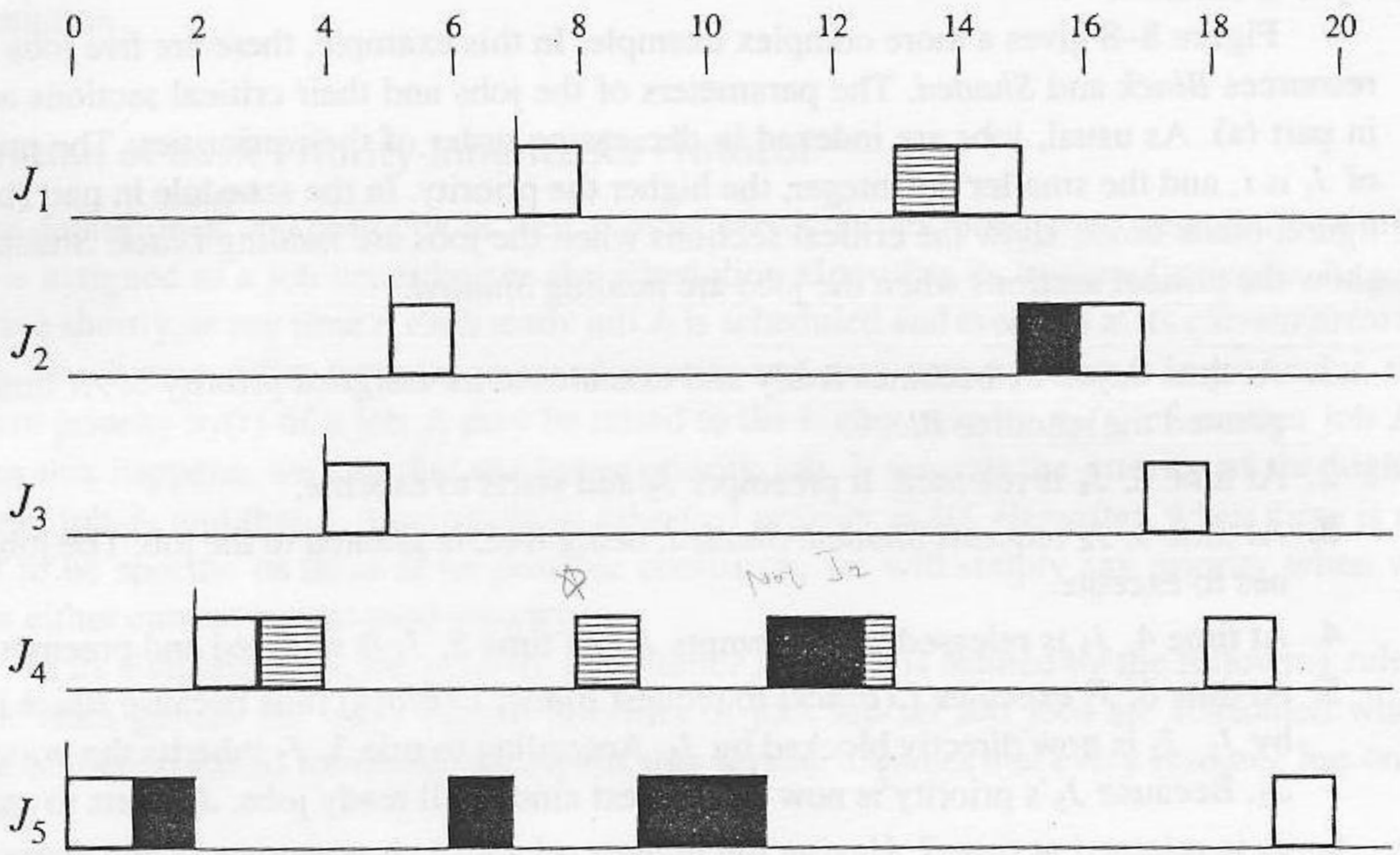
3. *Priority-Inheritance Rule*: When the requesting job J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J. The job J_l executes at its inherited priority $\pi(t)$ until it releases R; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquires the resource R.

Release 吧?

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

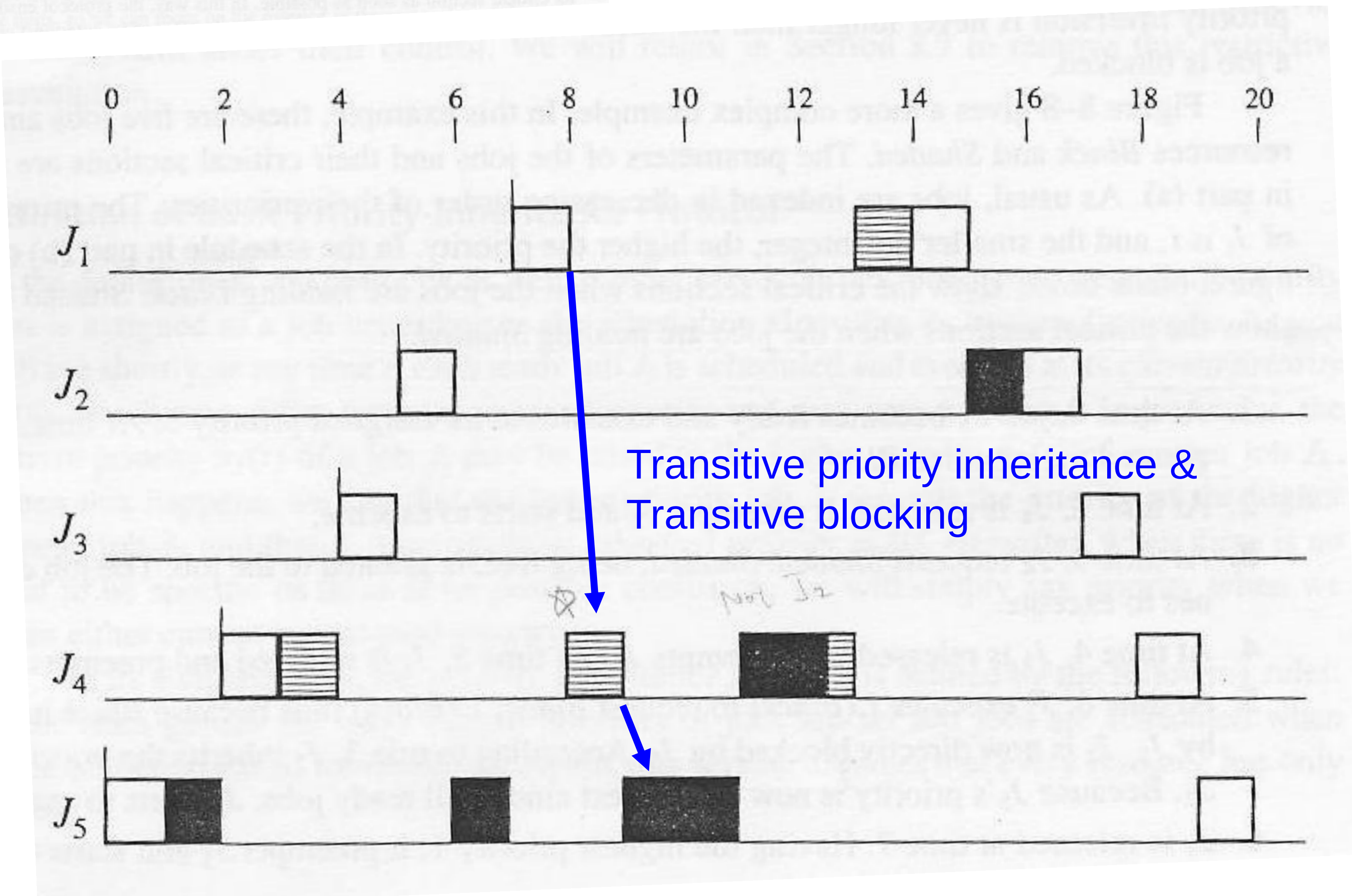
Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[*Shaded*; 1]
J_2	5	3	2	[*Black*; 1]
J_3	4	2	3	
J_4	2	6	4	[*Shaded*; 4 [*Black*; 1.5]]
J_5	0	6	5	[*Black*; 4]

(a)



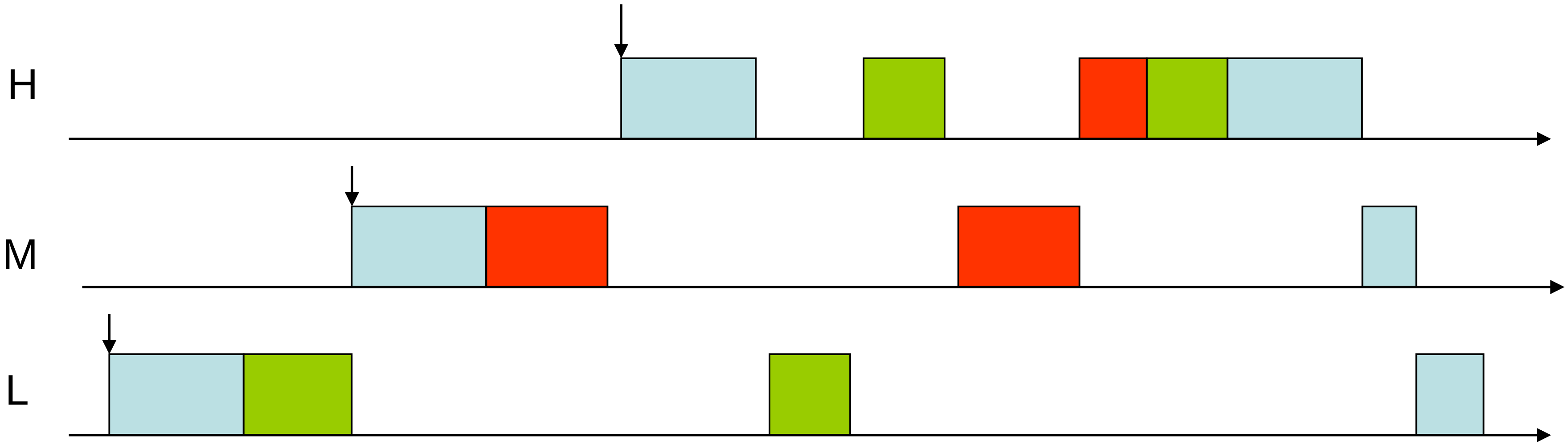
Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[*Shaded*; 1]
J_2	5	3	2	[*Black*; 1]
J_3	4	2	3	
J_4	2	6	4	[*Shaded*; 4 [*Black*; 1.5]]
J_5	0	6	5	[*Black*; 4]

(a)



Priority-Inheritance Protocol

- Is chain blocking prevented?



Priority-Inheritance Protocol

Chain blocking

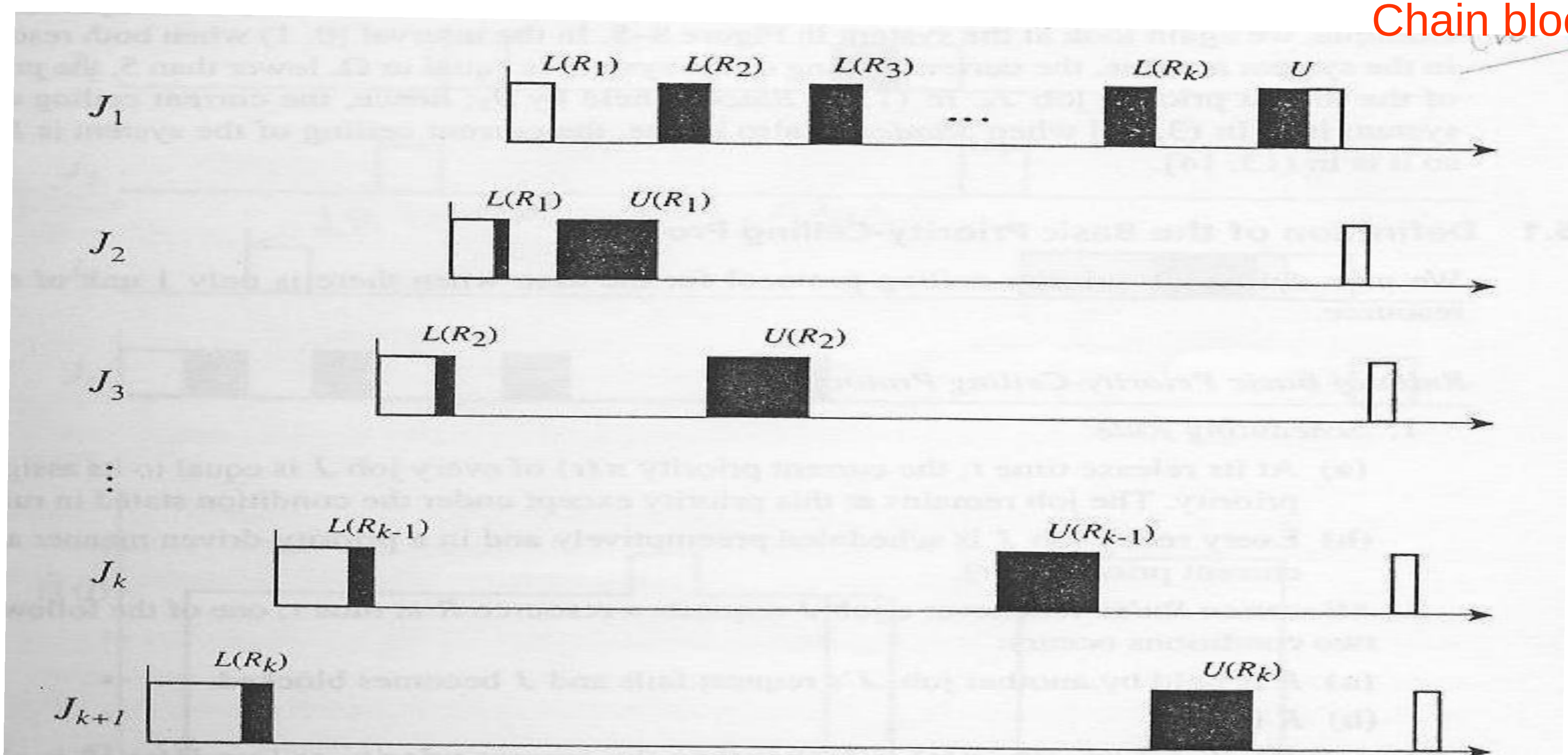


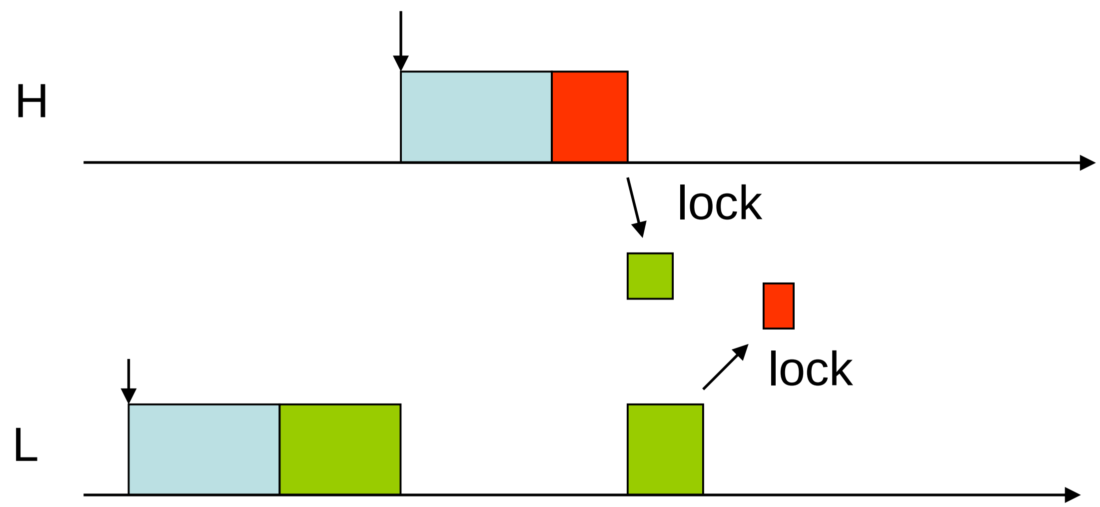
FIGURE 8–9 A worst-case blocking scenario for priority-inheritance protocol.

多個共享資源

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

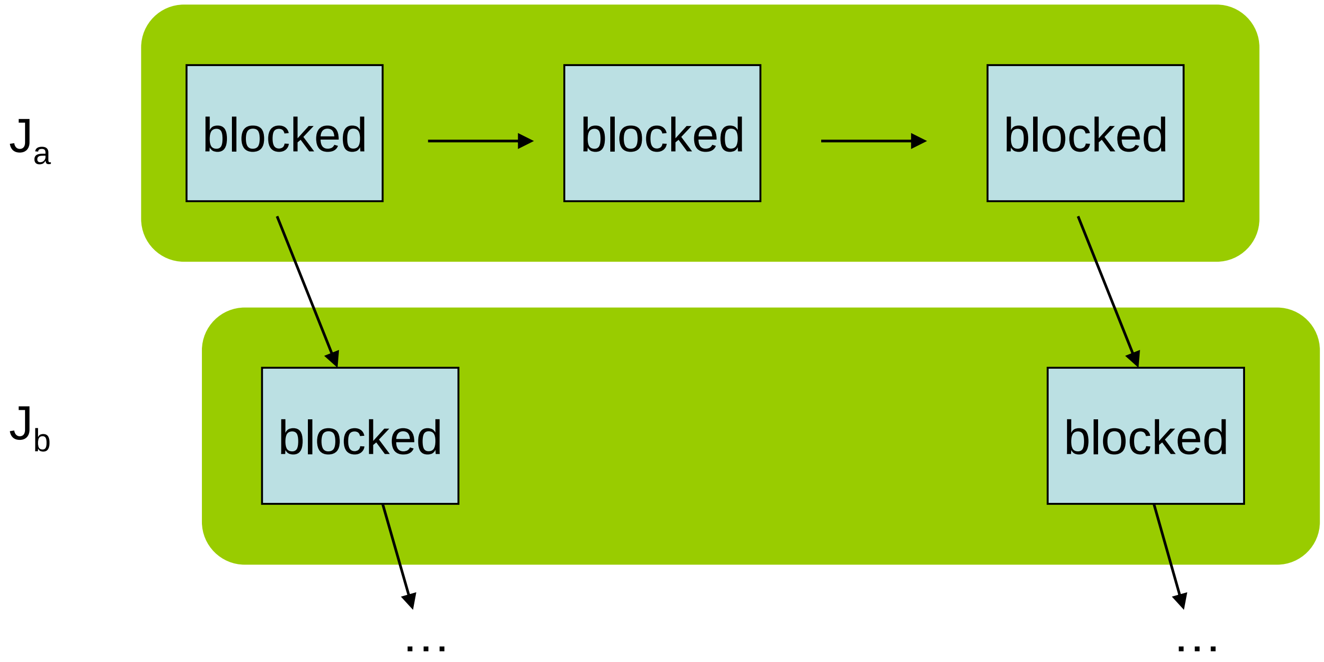
Priority-Inheritance Protocol

- Deadlocks

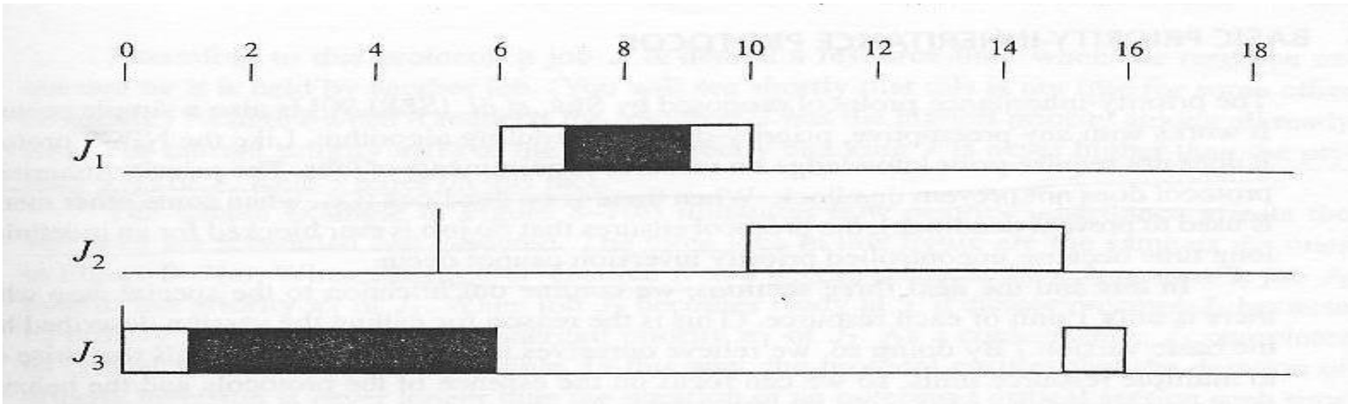


Priority-Inheritance Protocol

- In PIP, blocking time might be lengthy (still bounded though…)



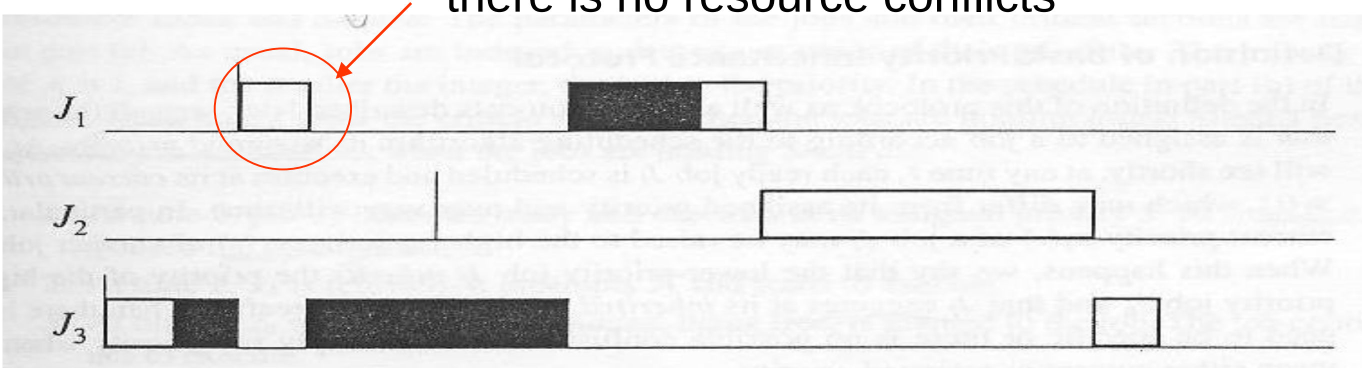
Priority-Inheritance Protocol



(a)

NPCS

Actually more high-priority jobs can run, provided there is no resource conflicts



(b)

PIP

Priority-Inheritance Protocol

- Pros:
  - Simple
  - High concurrency (high responsiveness)
- Cons:
  - Suffering from deadlocks
  - Blocking time is not well managed
    - Chain blocking
    - Transitive blocking

Priority-Ceiling Protocol

- PIP avoids uncontrolled priority inversions, but jobs suffer still from
  - Deadlocks
  - Lengthy blocking time (chain blocking, transitive blocking)

- PCP aims at resolving the shortcomings of PIP

Priority-Ceiling Protocol

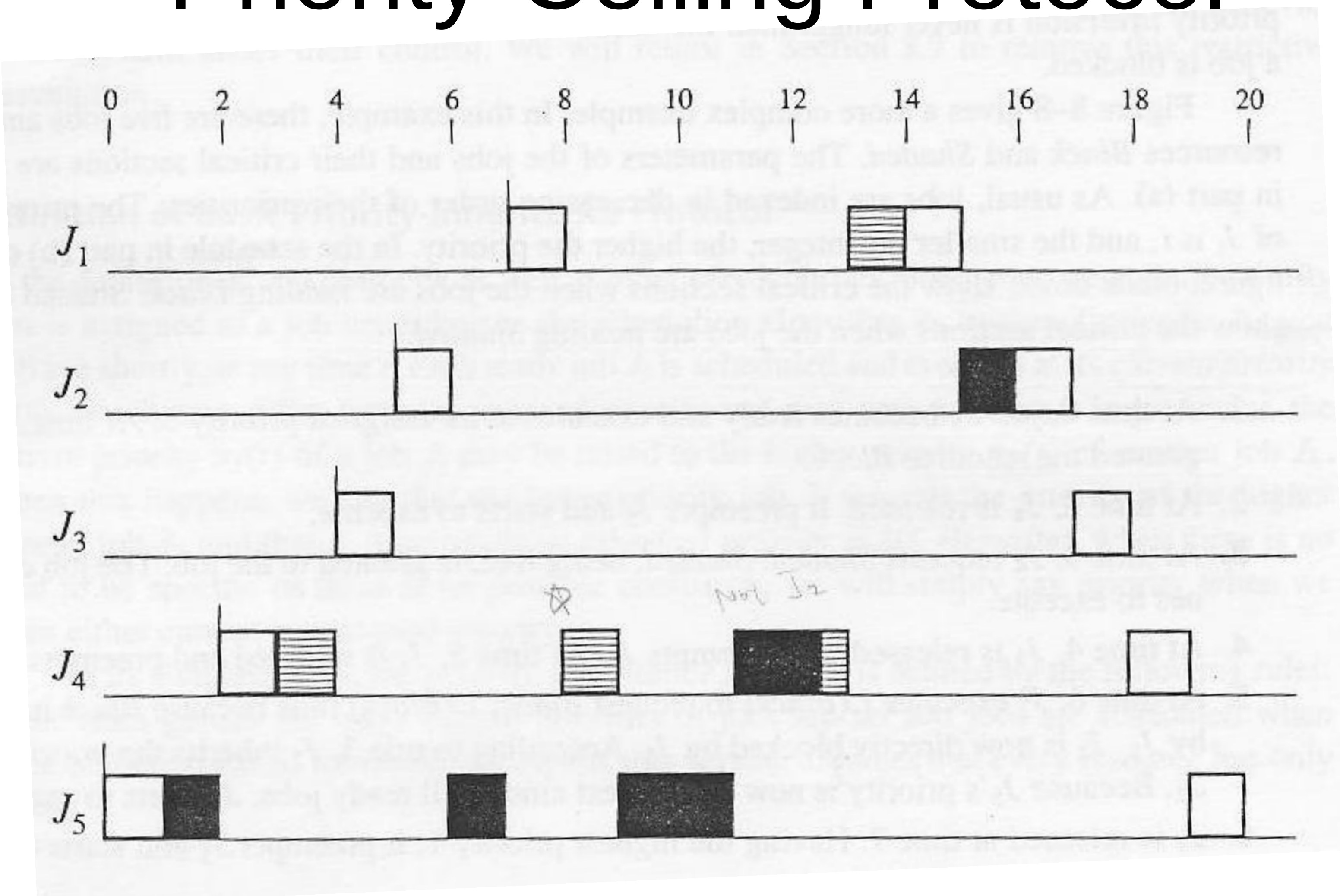
- Assumptions
  - All tasks have unique and fixed priorities
  - Resource usages of tasks are known *a priori*

每一個人只會被擋一次

- Terms
  - Π(R): priority ceiling of resource R 所有會用到這個資源的最高優先權
    - The highest priority of all tasks that require R
  - Π^(t): current system ceiling at time t (caution!)
    - The highest priority ceiling of all resources that are in use

所有最高被鎖住的資源

Priority-Ceiling Protocol

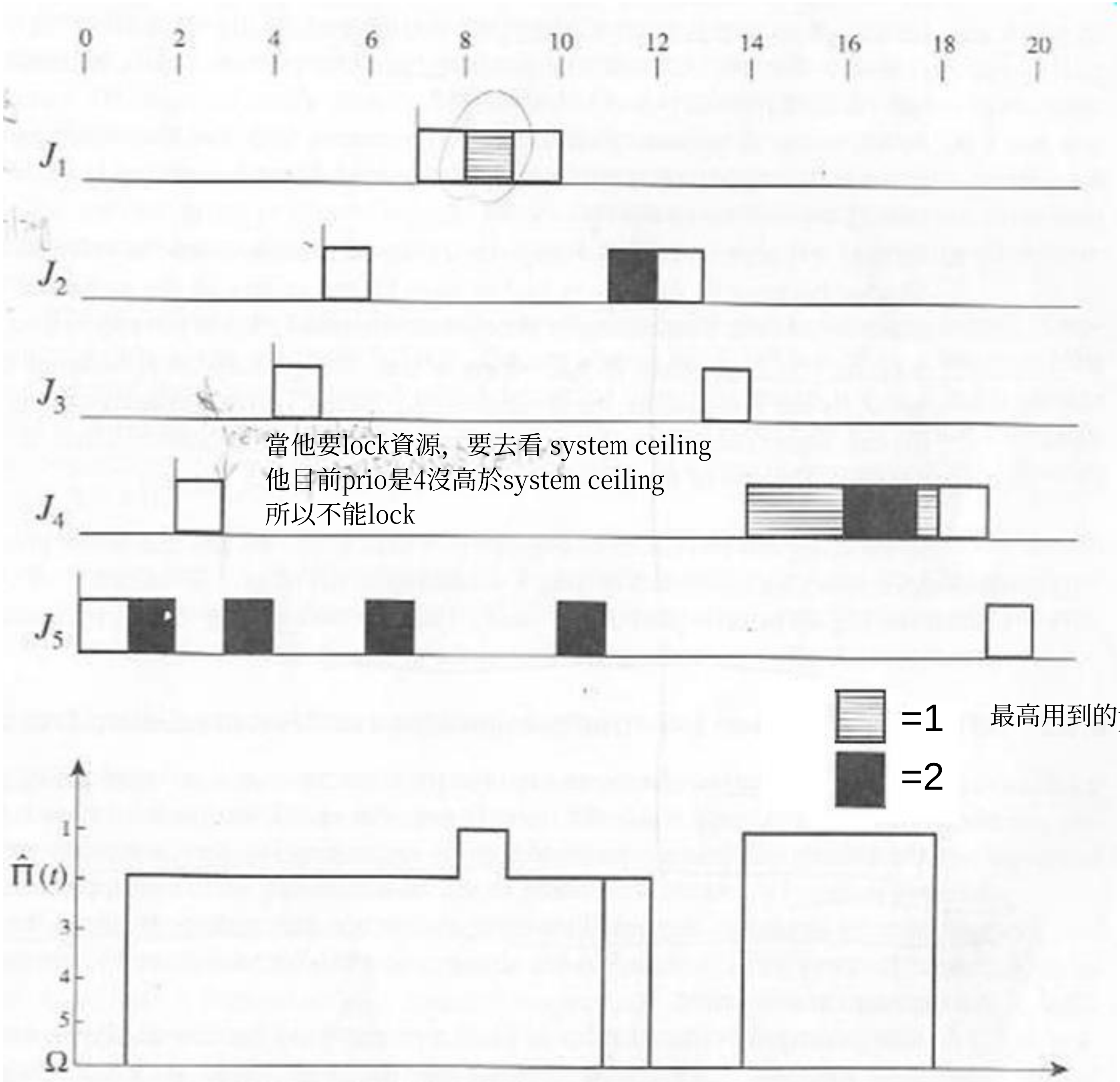


Ω 2 1 2 Ω

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).



當他要lock資源，要去看 system ceiling
他目前prio是4沒高於system ceiling
所以不能lock

prio 沒變

=1 最高用到的prio

=2

system ceiling

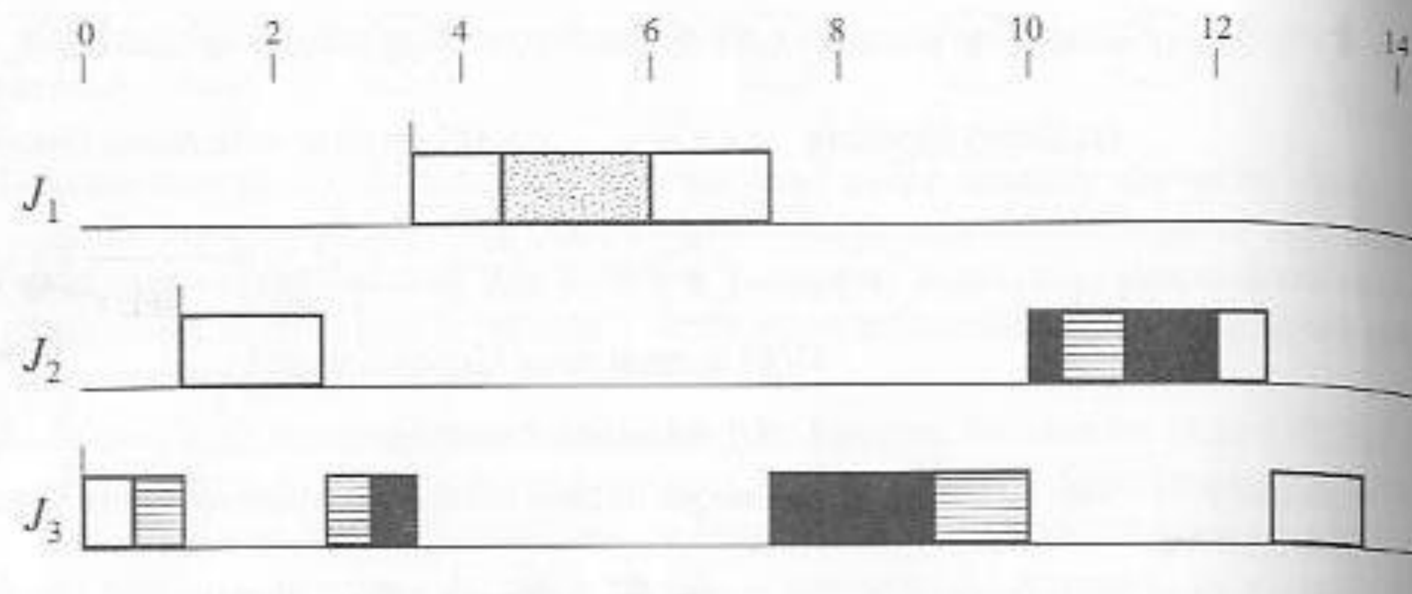
Priority-Ceiling Protocol

- PCP might deny a resource request even if the resource is free
  - PIP grants a resource request as the resource is free

- PCP imposes three types of blocking on jobs
  - Direct blocking
  - Priority-inheritance blocking
  - Avoidance blocking (new)
    - Priority-ceiling blocking
    - Actually, a kind of priority-inheritance blocking

Priority-Ceiling Protocol

- Lemma: PCP avoids uncontrolled blocking
- How about deadlocks?



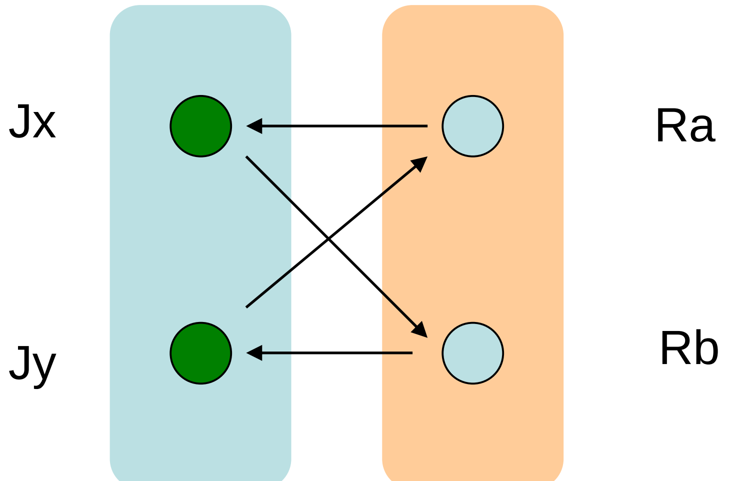
$\Pi(Dotted) = \pi_1$, $\Pi(Black) = \Pi(Shaded) = \pi_2$

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

Priority-Ceiling Protocol

- Theorem: PCP avoids deadlocks

  Proof:

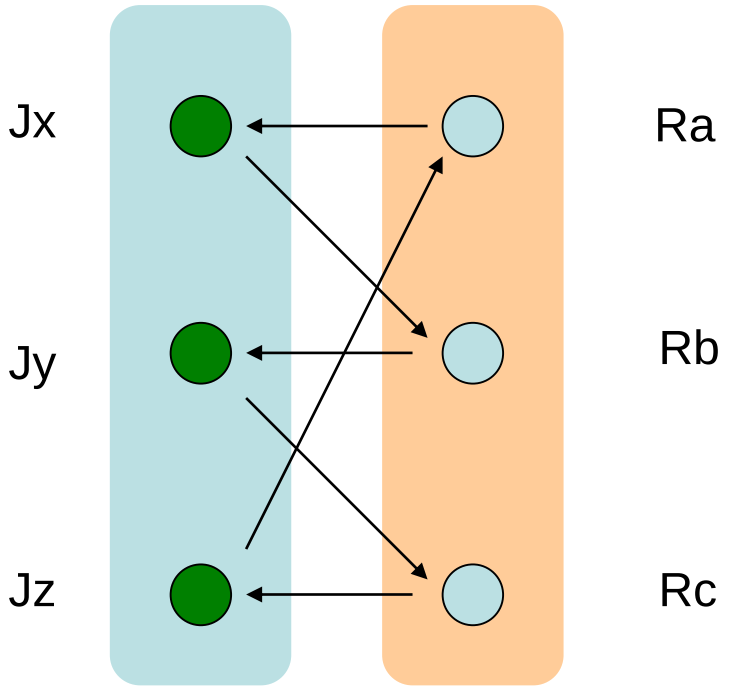


Jx and Jy can not simultaneously lock resources!!

Priority-Ceiling Protocol

- Theorem: PCP avoids deadlocks

  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 governed by PCP seem having longer response times
  - Because of priority-ceiling blocking (which does not happen in PIP)
- In the worst case, PCP does not impose longer blocking time on jobs than PIP does
  - Recall that jobs might suffer chain blocking and transitive blocking under PIP

Priority-Ceiling Protocol

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

  Proof:
  - A job can ever be blocked by only one job
  - No transitive blocking

Priority-Ceiling Protocol

- A job can ever be blocked by only one job
  - If job J acquires resource R after blocking, then the current system ceiling is no lower than J's priority and therefore no low-priority jobs can lock resources
  - If job J acquires resource R, then all resources required by J have been released by other jobs

$J_H \rightarrow J \rightarrow J_L$

Priority-Ceiling Protocol

- No transitive blocking
  - A job (J in the following statements) that holds some resource can not be blocked by any other jobs
    - If J_H is blocked by J when locking R, then (1) J has acquired R or (2) J has acquired R', which's priority ceiling is higher than that of R
    - Since J has acquired R or R', then the current system priority ceiling is no lower than J's priority. Therefore any J_L has released any resource needed by J

$J_H \rightarrow J \rightarrow J_L$

Priority-Ceiling Protocol

- Blocking time

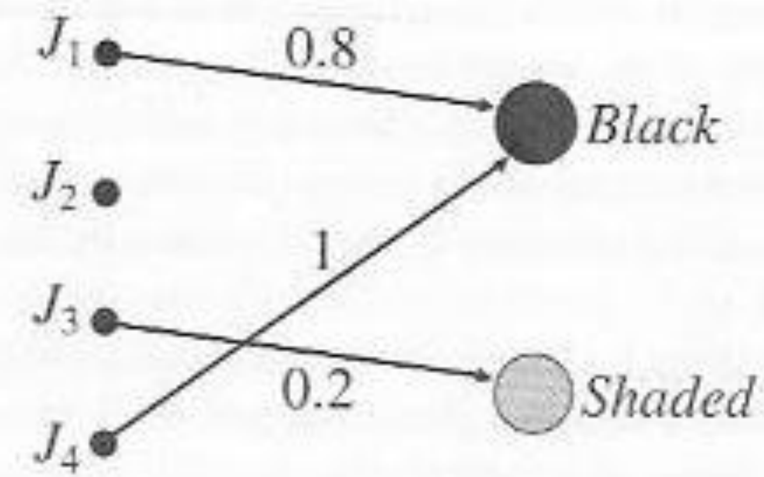
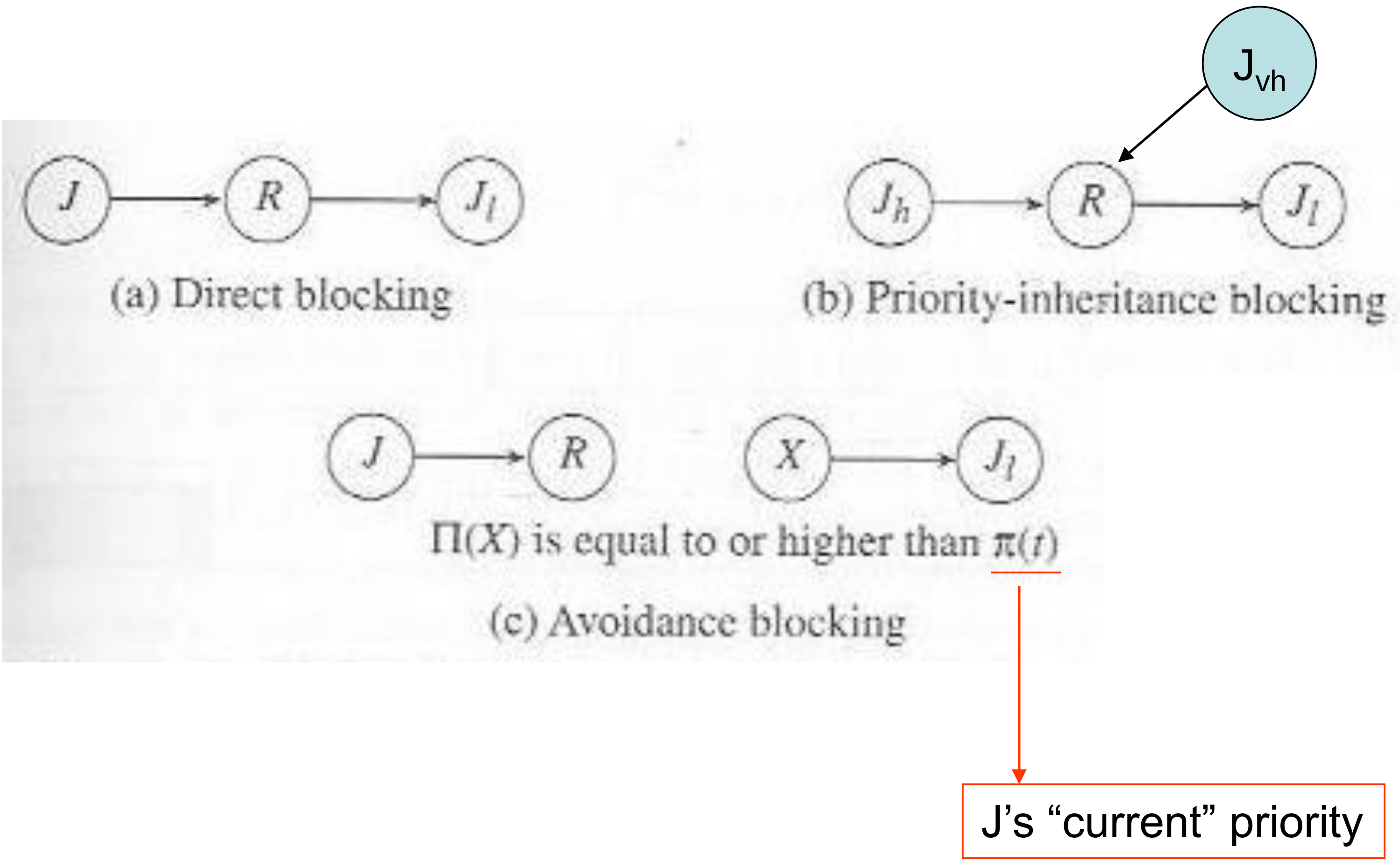


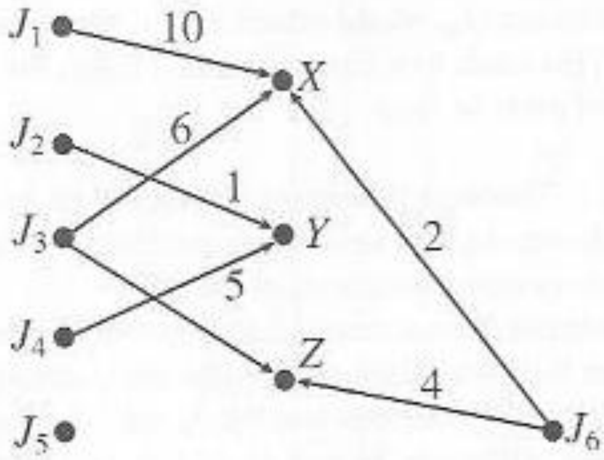
FIGURE 8–14 Example on duration of blocking.

J4 can block J2 by (1) priority ceiling of Black or (2) an priority inherited from J1

Priority-Ceiling Protocol



Priority-Ceiling Protocol



(a)

6,6,5,4,4

	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				*	

Priority-Ceiling Protocol

- For any job J that requires some resource $\{R_1,R_2,\ldots,R_n\}$, the blocking time due to priority inheritance and that due to priority ceiling are the same
  - Priority ceiling of each resource Ri >= any priority that J inherits from high-priority jobs
    - The highest priority that J inherits = priority ceiling of R

Priority-Ceiling Protocol

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

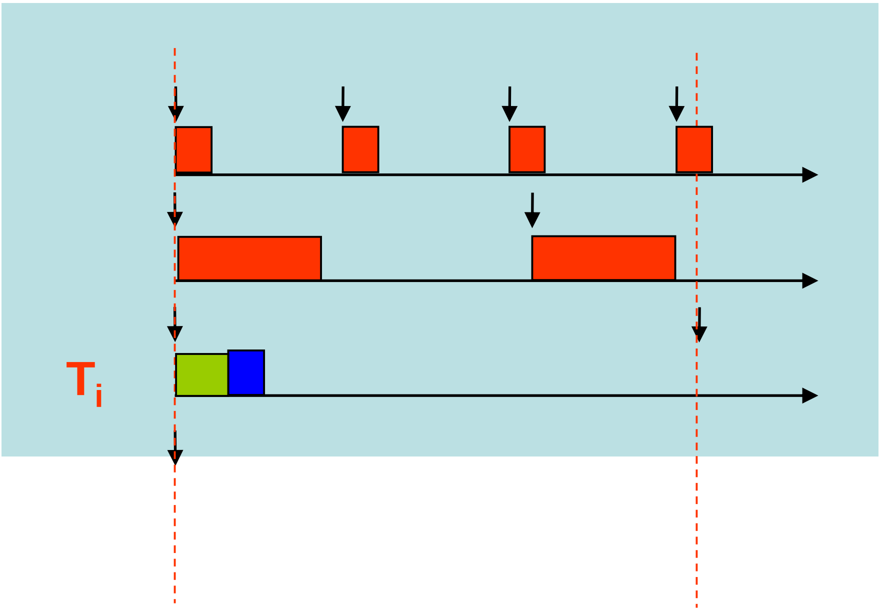
$$\forall i,\ 1 \le i \le n, \quad \frac{b_i}{p_i} + \sum_{j=1}^{i} \frac{c_j}{p_j} \le U(i)$$

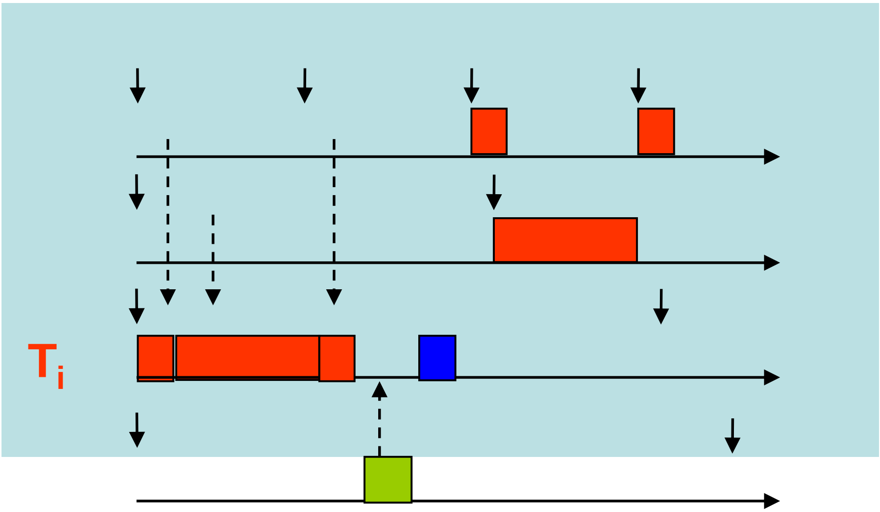
or $$\forall i,\ 1 \le i \le n, \quad \max\{\frac{b_i}{p_i}\} + \sum_{j=1}^{n} \frac{c_j}{p_j} \le U(n)$$

- Where b_i is the longest blocking time imposed on T_i

Priority-Ceiling Protocol

$$\forall i,\ 1 \le i \le n, \quad \max\{\frac{b_i}{p_i}\} + \sum_{j=1}^{n} \frac{c_j}{p_j} \le U(n)$$





The critical instant of T_i with or without blocking

Priority-Ceiling Protocol

- Fig 8-16

Another scenario: T1, T2 and T3 arrive at epsilon, T4 arrives at 0 and blocks T1.

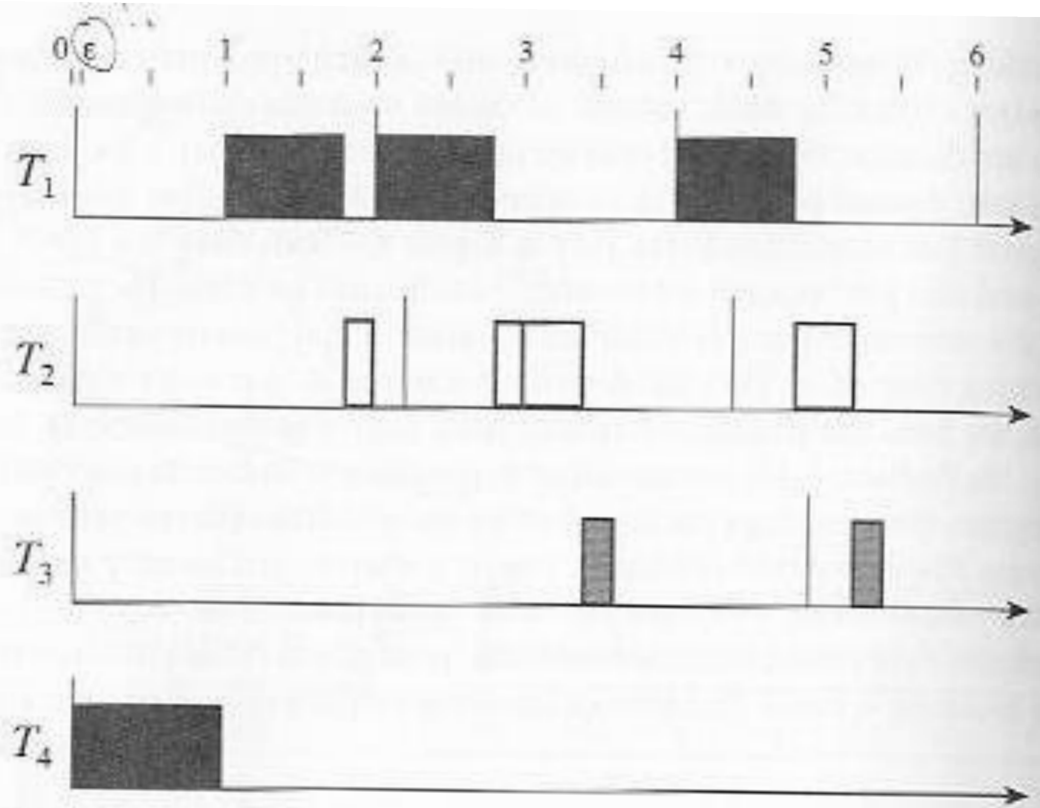


FIGURE 8–16 Example on fixed-priority scheduling and priority-ceiling protocol. ($T_1 = (\varepsilon, 2, 0.8; [Black; 0.8])$, $T_2 = (\varepsilon, 2.2, 0.4)$, $T_3 = (\varepsilon, 5, 0.2; [Shaded; 0.2])$, $T_4 = (10, 1; [Black; 1.0])$).

Priority-Ceiling Protocol

- Context-switch issue
  - As mentioned earlier, two context switches are accounted to the preempting job
    - Thus every job is accounted to 2*CS

  - Under PCP, a job can be blocked for at most once
    - Thus additional 2*CS is accounted to a job that potentially can "be blocked"

Priority-Ceiling Protocol

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

$$\forall i,\ 1 \leq i \leq n, \quad \frac{b_i}{p_i} + \sum_{j=1}^{i} \frac{f(j) * CS + c_j}{p_j} \leq U(i)$$

in which $f(j)$=4 if T_j 's blocking time is not zero, otherwise $f(j) = 2$.

Ceiling priority protocol

- If R is in use, T is blocked.
- If R is free, R is allocated to T. T's execution priority is raised to the priority ceiling of R if that is higher. At any given time, T's execution priority equals the highest priority ceiling of all its held resources.

Ceiling priority protocol

- T's priority is assigned the next-highest priority ceiling of another resource when the resource with the highest priority ceiling is released.
- The task returns to its assigned priority after it has released all resources.

Stack-Resource Policy

- Motivation
  - PCP is originally designed for RMS, but so far no resource-synchronization protocols are there for EDF
  - PCP shows high concurrency with the price that introduces extra context switches

Stack-Resource Policy

- In EDF, no task is assigned to a fixed priority thus PCP is not directly applicable
  - EDF is a job-level fixed priority scheduling algorithm
  - Dynamic in task level

- Consider job J_i is released at time r_i and its relative deadline is d_i
  - Job J_i could be preempted by job J_j only if $d_j - r_j < d_i - r_i$
    - I.e., Job J_j could be blocked by J_i

Stack-Resource Policy

- In periodic systems, the difference between deadline and release time of jobs with respect to a task is a constant
  - In other words, jobs of a task can only be blocked by jobs of another task which has a longer period
- Because the notation “priority” does not apply to tasks in EDF, tasks are associated with “preemption level” instead
  - The shorter task periods are, the higher their preemption levels are

Stack-Resource Policy

- Because jobs under PCP may be blocked after they start executing, job execution do not comply with LIFO discipline

- Job execution complies with LIFO discipline only if jobs won’t voluntarily give up CPU on the way they execute
  - When a job starts executing, it executes continuously until it finishes

- If LIFO discipline is complied with, one single run-time stack is enough for tasks
  - Saves a lot of memory usage

uC/OS-II: RAM requirement = application requirement + kernel requirement + SUM(task stacks + MAX(ISR nesting))

Stack-Resource Policy

- Basic idea of SRP
  - Task preempting / blocking in EDF is nothing different from that in RMS
    - The definition and controlling of priority inversion should be revised
  - Once a job starts executing, all resource it requires have been released by other tasks
    - To comply with LIFO discipline and to prevent deadlocks

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 $\Psi(t)$ 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 "urgency" or "absolute deadline"!!
And preemption levels reflect task periods!!

- PCP: check system ceiling when jobs lock resources
- SRP: check system ceiling when jobs are scheduled to run

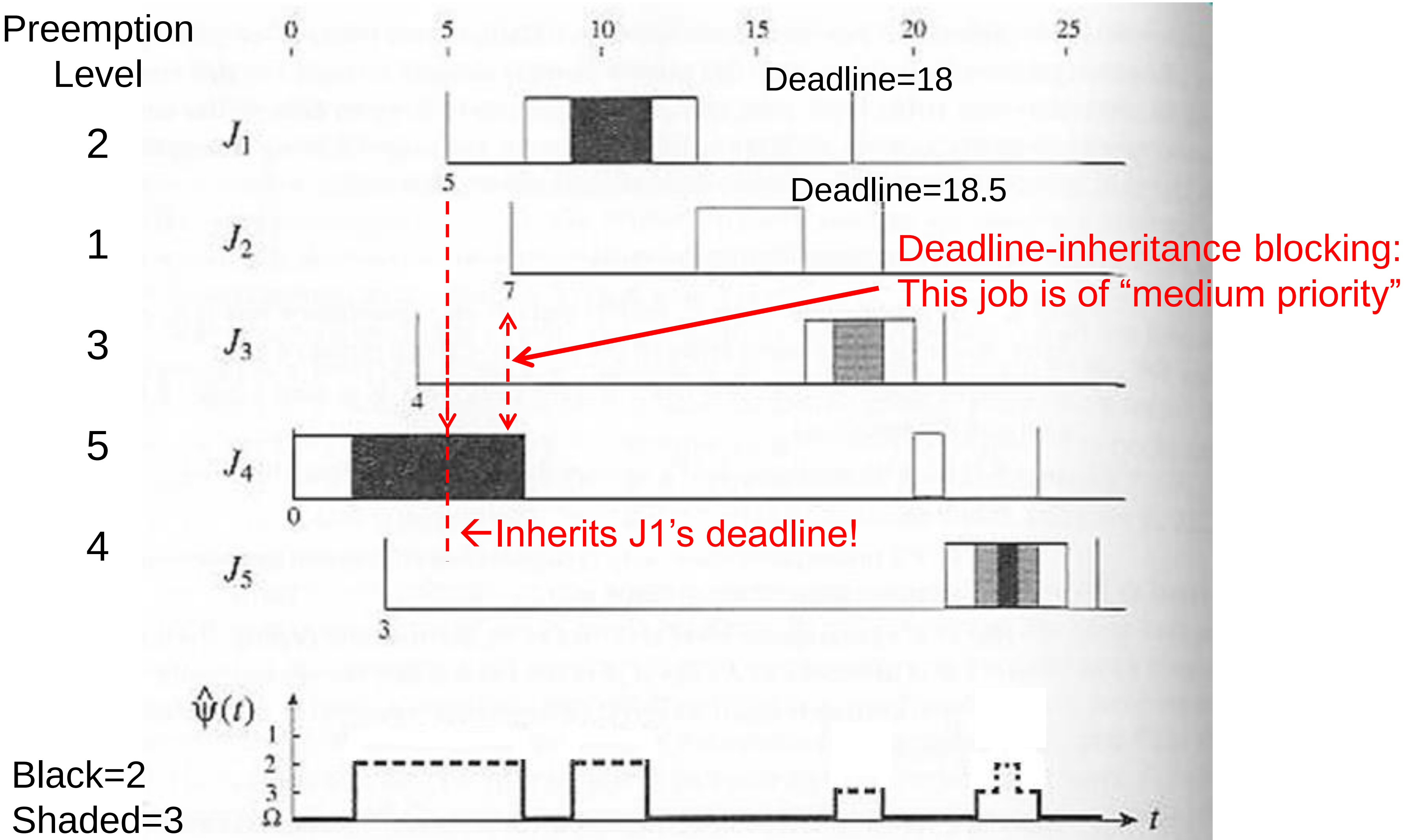


FIGURE 8–20 Example of priority ceilings of multiple-unit resources.

** jobs of higher task preemption level do not always preempt jobs of lower preemption levels

Stack-Resource Policy

- SRP avoids uncontrolled priority inversion
  - Protected by “priority”-inheritance
- SRP avoids transitive blocking and chain blocking
  - When a job start executing, all that resources it needs have been released, and these resources will not be locked until the job completes
- SRP prevent deadlocks from happening
  - A job is never be blocked once it start executing

Stack-Resource Policy

- Blocking time
  - Direct blocking (?)
  - Deadline-inheritance blocking
  - Preemption-ceiling blocking

- Blocking time imposed on a job under SRP+EDF is the same as that of PCP+RMS

Stack-Resource Policy

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

$$\forall i, \frac{b_i}{p_i} + \sum_{j=1}^{n} \frac{2*CS+c_j}{p_j} \leq 1$$

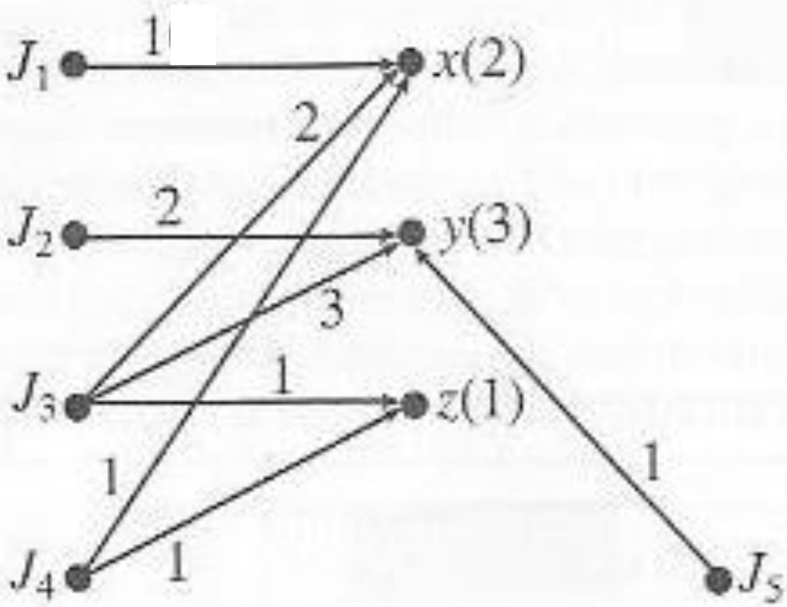
- No 2 extra context switches as that in PCP!

Stack-Resource Policy

- Pros:
  - Tasks can share one single stack
  - Easy to implement
  - Saves up 2 context switches
- Cons:
  - Relatively low concurrency, compared to PCP

Stack-Resource Policy

- Priority ceilings for resources of multiple units
- Let k be the number of free units of resource R after some units of R is locked
- Let Π(R,k) be the priority ceiling (preemption ceiling) of resource R there are k units available
  - Π(R,k) equals to the highest priority (preemption level) of all tasks that require more than k units (>) of resource R
    - These task jobs may be directly blocked when trying to lock resource R



Resources		$x(2)$	$y(3)$	$z(1)$
Units Required by	J_1	1	0	0
	J_2	0	2	0
	J_3	2	3	1
	J_4	1	0	1
	J_5	0	1	0
$\Pi(*, 0)$		π_1	π_2	π_3
$\Pi(*, 1)$		π_3	π_2	Ω
$\Pi(*, 2)$		Ω	π_3	Ω
$\Pi(*, 3)$		Ω	Ω	Ω

FIGURE 8–21 Example of priority ceilings of multiple-unit resources.

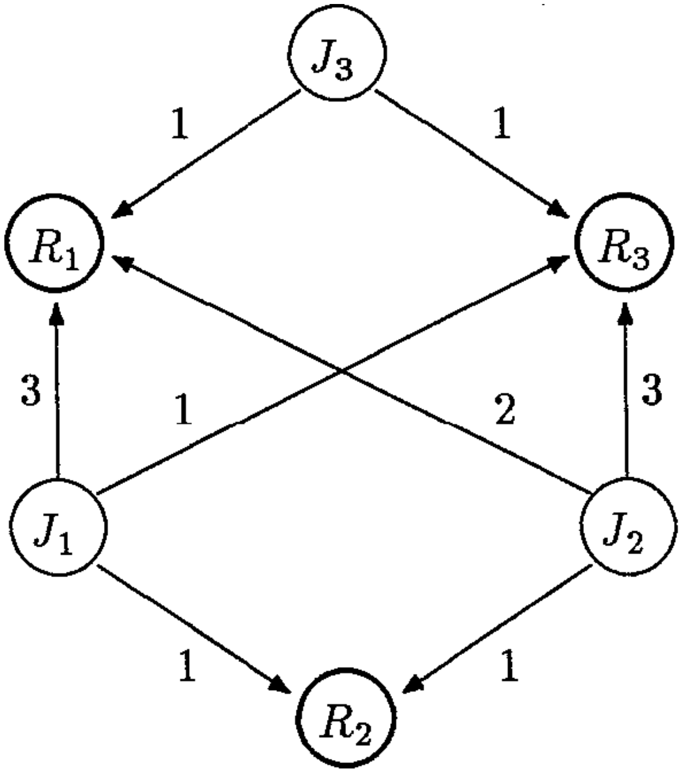
- **Note that, the ceiling of resource R has effects to resource locking (PCP) or job scheduling (SRP) only if some units of resource R are (is) successfully locked!!**

- **$\Pi(R,0)$ will be used as the ceiling of R when a resource of one unit is locked!**

R	N_R	$\mu_R(1)$	$\mu_R(2)$	$\mu_R(3)$	$\lceil R \rceil_0$	$\lceil R \rceil_1$	$\lceil R \rceil_2$	$\lceil R \rceil_3$
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

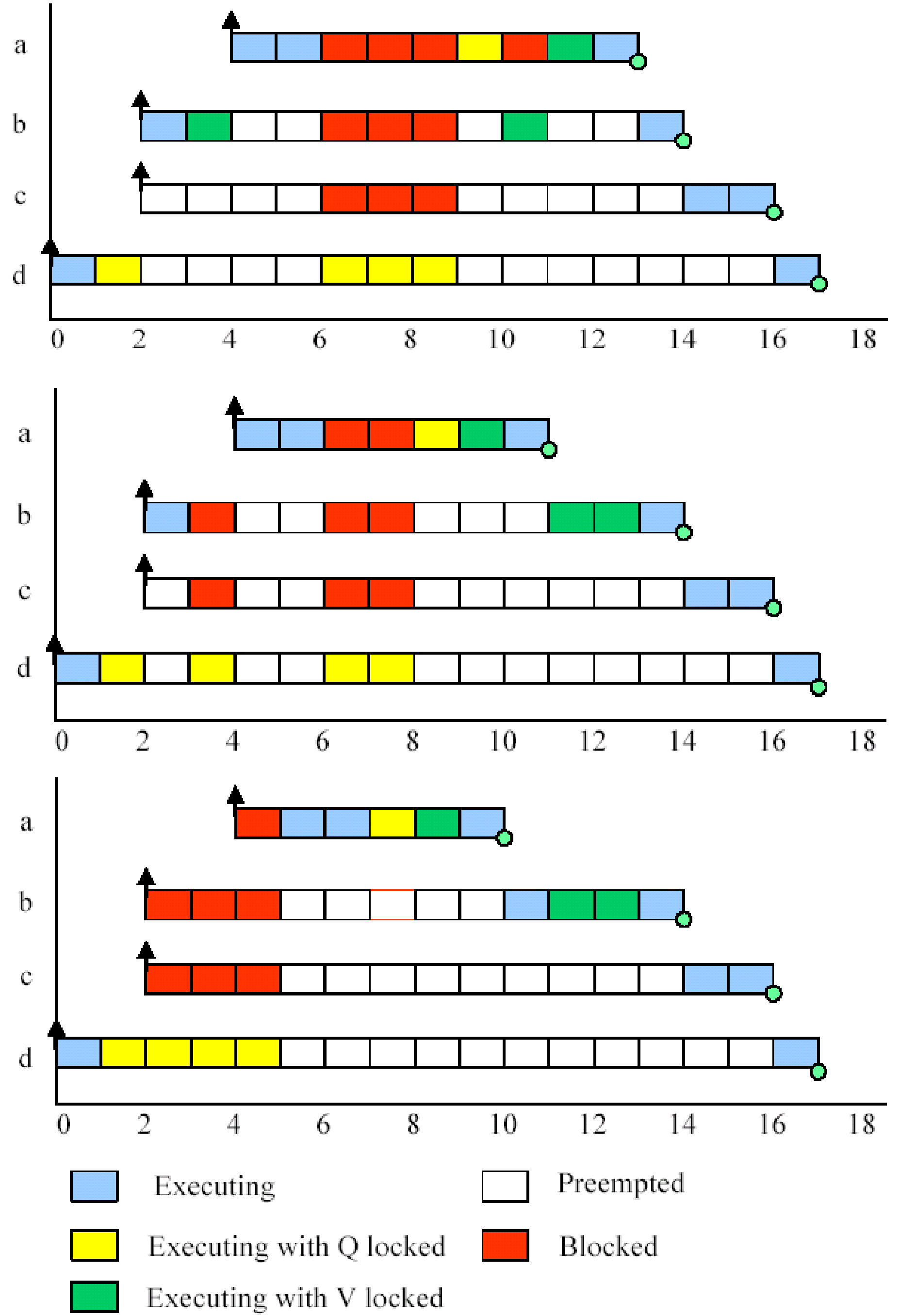


Preemption levels: J3>J2>J1

Ungoverned

PCP+RMS

SRP+EDF



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

Summary

- Check list
  - Priority inversion
  - Blocking time
  - Blocking duration
  - Deadlock
  - Concurrency
  - Multiple-unit resources
- The tradeoff between concurrency and simplicity is always the primary challenge