Resources & Resource Access Control

- Thus far…
  - We've assumed that tasks are independent of each other!
  - What if they share resources (other than processor itself)?
    - **Examples of resources**
      - Shared variables/data
      - Photo copier, printer, etc.
      - Remote server
  - So, what's different if tasks share resources
    - ***Mutual exclusion*** needs to be guaranteed
    - Scheduling needs to consider this aspect!

What is “mutual exclusion”?

- Assurance that only one task accesses resource at a time

- Simple example scenario: consider tasks T_1 and T_2
  - T_1 (higher-priority task)
    - Withdraws money (say $50) from bank account
  - T_2 (lower-priority task)
    - Deposits money (say $50) into same bank account

Mutual exclusion example

Account balance

100

Task T_1:

Load balance
Subtract withdrawal amount
Store updated balance

Task T_2:

Load balance
Add deposit amount
Store updated balance

Mutual exclusion example

Account balance

100

Task T_1:

Load balance
Subtract withdrawal amount
Store updated balance

Task T_2:

100

Load balance
Add deposit amount
Store updated balance

Mutual exclusion example

Account balance

100

Task T_1:

Load balance
Subtract withdrawal amount
Store updated balance

Task T_2:

150

Load balance
Add deposit amount
Store updated balance

Mutual exclusion example

Account balance

100

Task T_1:

100

Load balance
Subtract withdrawal amount
Store updated balance

Task T_2:

150

Load balance
Add deposit amount
Store updated balance

Mutual exclusion example

Account balance

100

Task T_1:

50

Load balance

Subtract withdrawal amount

Store updated balance

Task T_2:

150

Load balance

Add deposit amount

Store updated balance

Mutual exclusion example

Account balance

50

Task T_1:

50

Load balance
Subtract withdrawal amount
Store updated balance

Task T_2:

150

Load balance
Add deposit amount
Store updated balance

Mutual exclusion example

Account balance

150

Balance should have been 100!

Task T_1:

50

Load balance
Subtract withdrawal amount
Store updated balance

Task T_2:

150

Load balance
Add deposit amount
Store updated balance

Conclusion: there should ***no interleaving*** of execution when tasks are accessing shared resources!!!

So, to enforce mutual exclusion...

- Tasks must ***lock*** shared resources before using them
  - No other task must be granted resource until it is *unlocked*!

Shared Resources

- Add to model: set of ρ serially **reusable resources** R_1, R_2, ..., R_ρ, where there are v_i units of resource R_i.
  - **Examples of resources:**
    - Binary semaphore, for which there is one unit.
    - Counting semaphore, for which there may be many units.
    - Reader/writer locks.
    - Printer.
    - Remote server.

Locks

- If n units of resource R are required
  - Lock request → **L(R, n)**
  - Corresponding unlock request → **U(R, n)**
- A matching lock/unlock pair is a **critical section**
- Critical section denoted by **[R, n; e]**
  - R → resource requested
  - n → number of units
  - e → execution time of critical section

Locks (Continued)

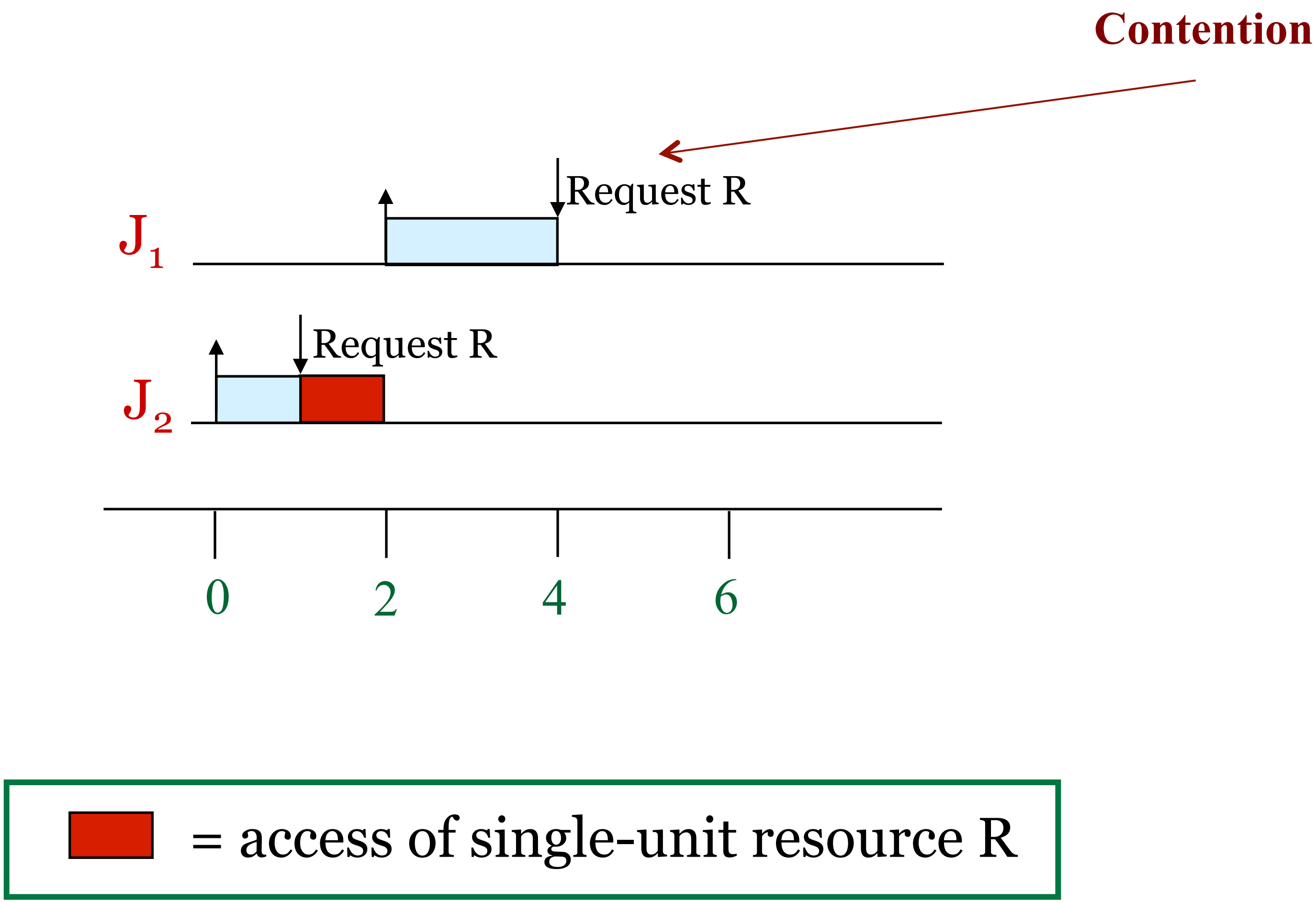
- Locks can be **nested**
- Notation:
  - $[R_1; 14\ [R_4, 3; 9\ [R_5, 4; 3]]]$
- Mostly interested in **outermost critical sections**
- **Note:** Assume there is only one kind of lock request
  - E.g., cannot distinguish reader locks/writer locks

Conflicts and Contentions

- Resource **conflict**
  - Two jobs requesting common resources have conflict
  - If we had reader/writer locks, then the notion of a "conflict" would be a little more complicated
- Resource **contention**
  - Manifestation of conflict during system operation
  - One job requests for resource held by another job
  - Requesting job must wait…it is said to be ***blocked***

    → *need to calculate upper bound on waiting/blocking time*

Example of resource contention



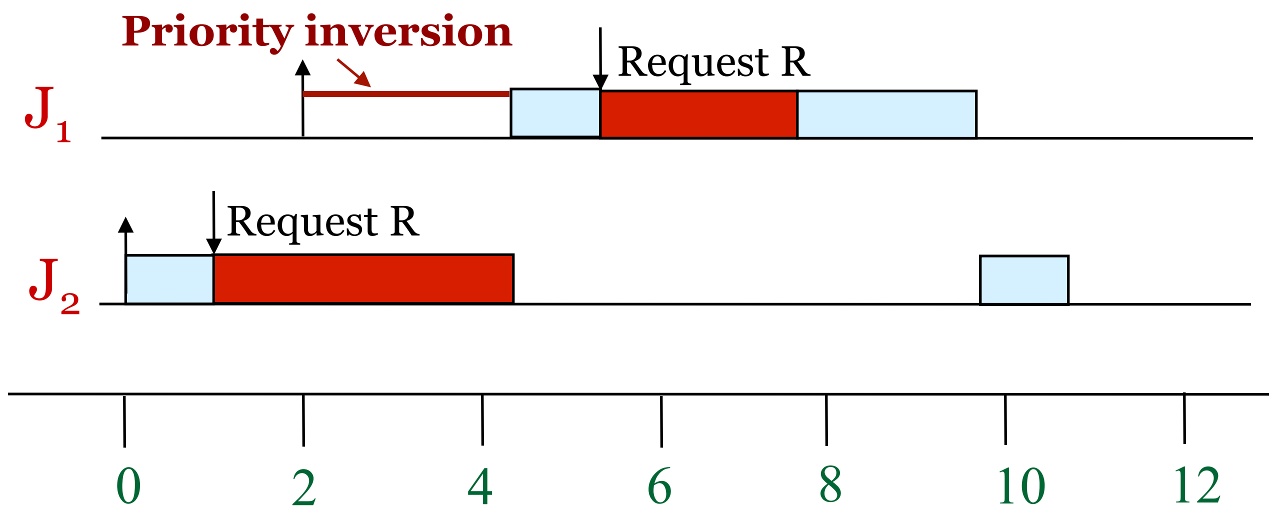
Dealing with resource contentions

- Execute all critical sections **non-preemptively**!
  - If tasks are indexed by priority, **blocking term** of T_i is

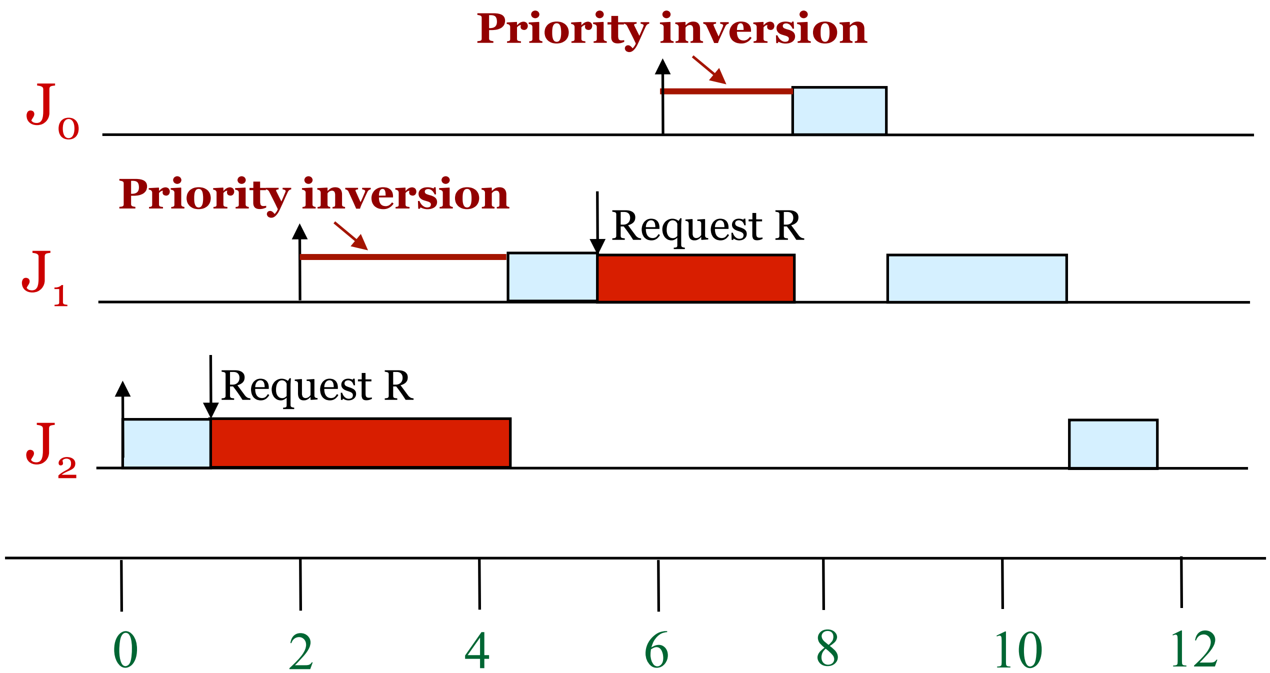
    $\max_{i+1 \leq k \leq n} e_k$

    - e_k is the execution cost of the longest critical section of T_k
  - Note: Critical section ➔ **outermost critical section** unless otherwise specified

Example



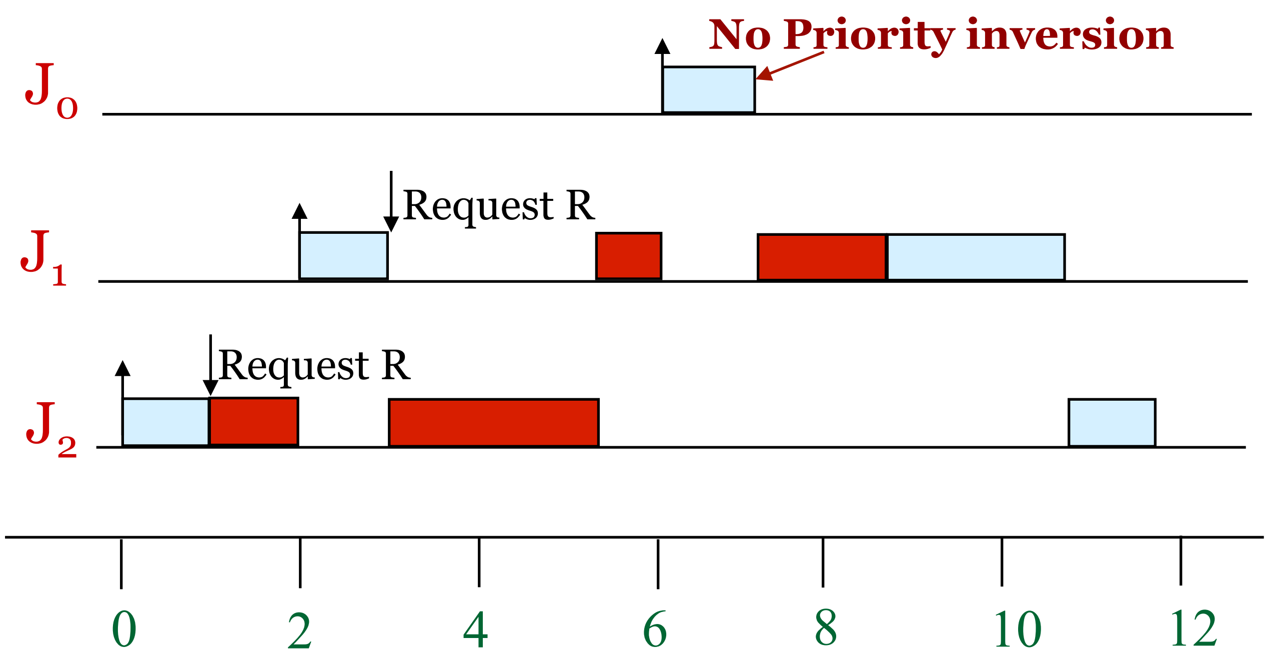
Let’s add another job into the mix…



What are the **disadvantages** here?

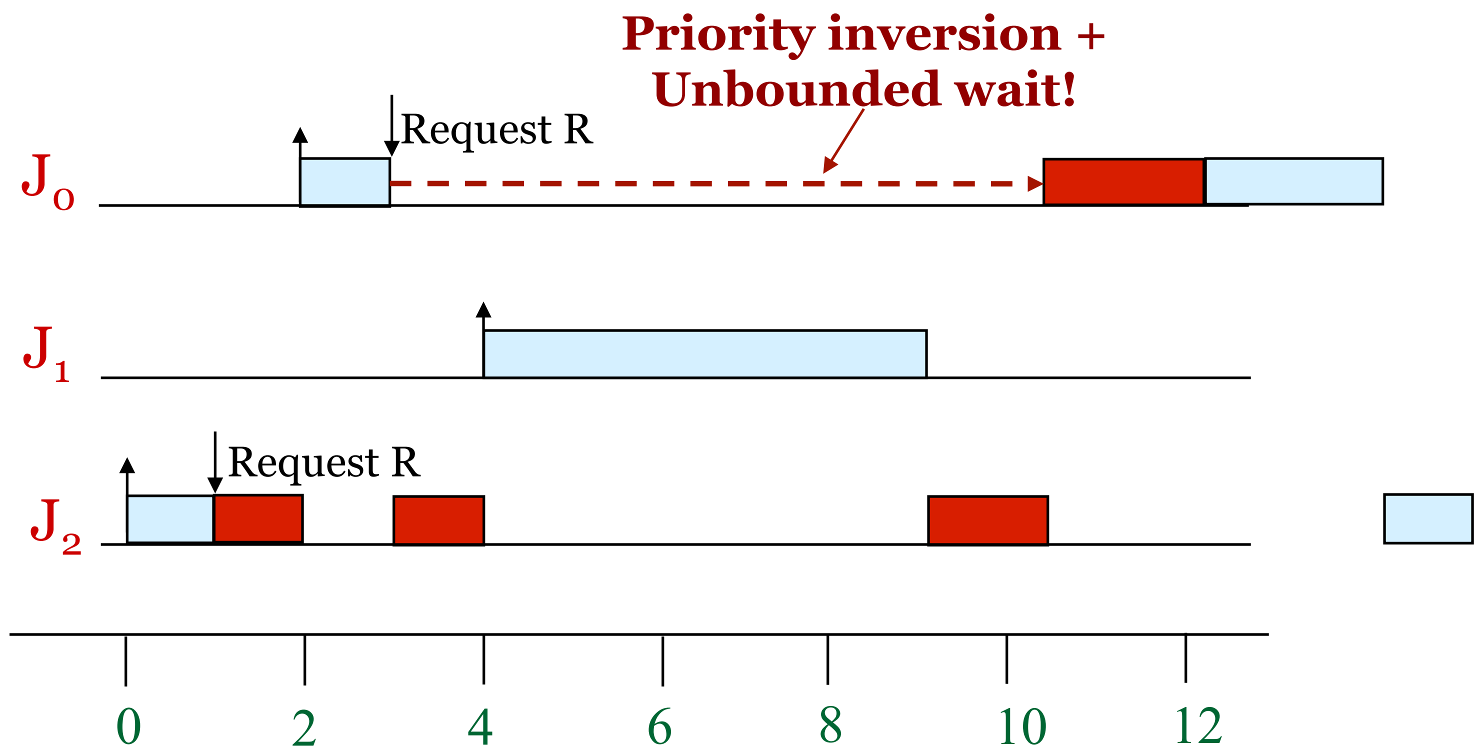
Making only resources non-preemptive...

- Allow jobs with resources to be preempted, handle contention dynamically
  - i.e., start higher-priority jobs as they arrive...handle contention for R if and when it happens



Have we solved all issues?

No, not quite! Consider another scenario…

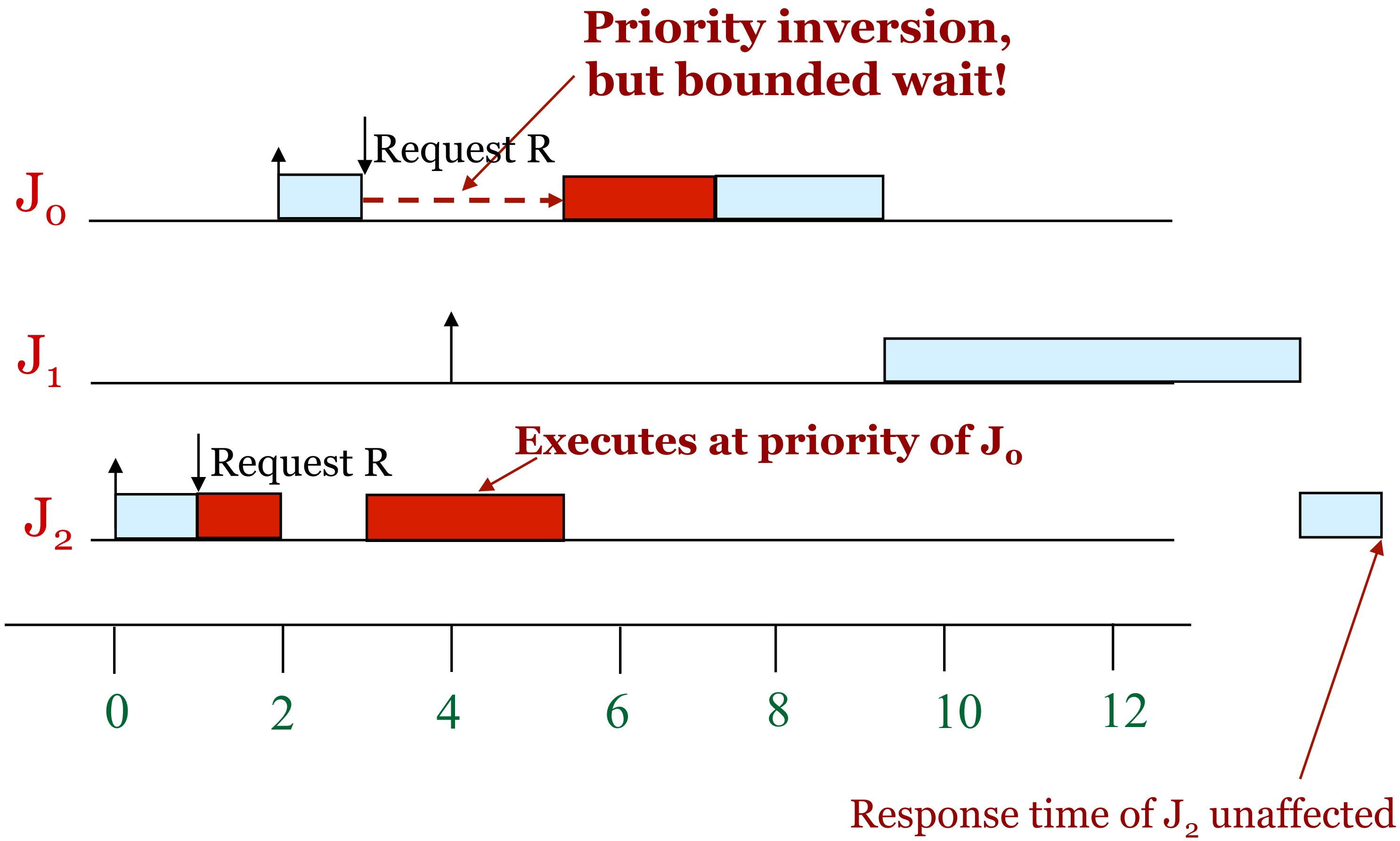


The problem here is the medium priority job $\mathbf{J_1}$!!!

The problem, once again…

- Problems in previous example
  - Priority inversion
    - This can't be eliminated altogether when sharing resources
  - Unbounded wait time for J_0
    - Caused due to medium-priority task J_1
    - Fact that J_0 is waiting is not considered by scheduler
      - Scheduler only knows about priorities…
    - Solution?
      - Fact that J_0 is waiting needs to be reflected in priorities!
      - Make J_2 "inherit" J_0's priority while using resource
      - "Priority Inheritance Protocol"

Example revisted…



Priority Inheritance Protocol Definition

Each job J has **assigned priority** and **current priority**. At its release time, current priority of every job is equal to its assigned priority.

1. **Scheduling Rule:** Ready jobs scheduled preemptively in a priority-driven manner according to their *current* priorities.

2. **Allocation Rule:** When job J requests for resource R at time t
   a. If R is free, R is allocated to J until J releases it
   b. If R is not free, the request is denied and J is *blocked*

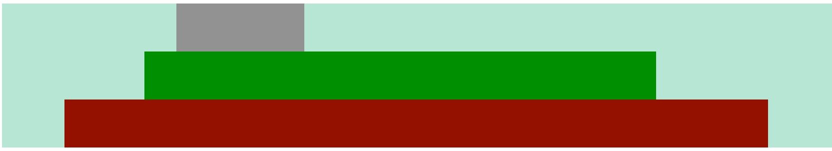
3. **Priority-Inheritance Rule**
   a. When requesting job J becomes *blocked*, job J_l that blocks J *inherits* the *current priority* of J.
   b. Job J_l executes at its inherited priority until it releases R (or until it inherits an even higher priority)
   c. Upon release of R, priority of J_l returns to priority it had when it acquired resource R

Formal terminology…

- Multiple serially **reusable resources** $R_1, R_2, \ldots, R_\rho$, where there are v_i units of resource R_i
- If n units of resource R are required
  - Lock request → **L(R, n)**
  - Corresponding unlock request → **U(R, n)**
- A matching lock/unlock pair is a **critical section**
- Critical section denoted by **[R, n; e]**
  - R → resource requested
  - n → number of units
  - e → execution time of critical section

Nested resource usage

- Locks can be **nested**
- Notation:
  - [$\mathbf{R_1}$; 14 [$\mathbf{R_4}$, 3; 9 [$\mathbf{R_5}$, 4; 3]]]

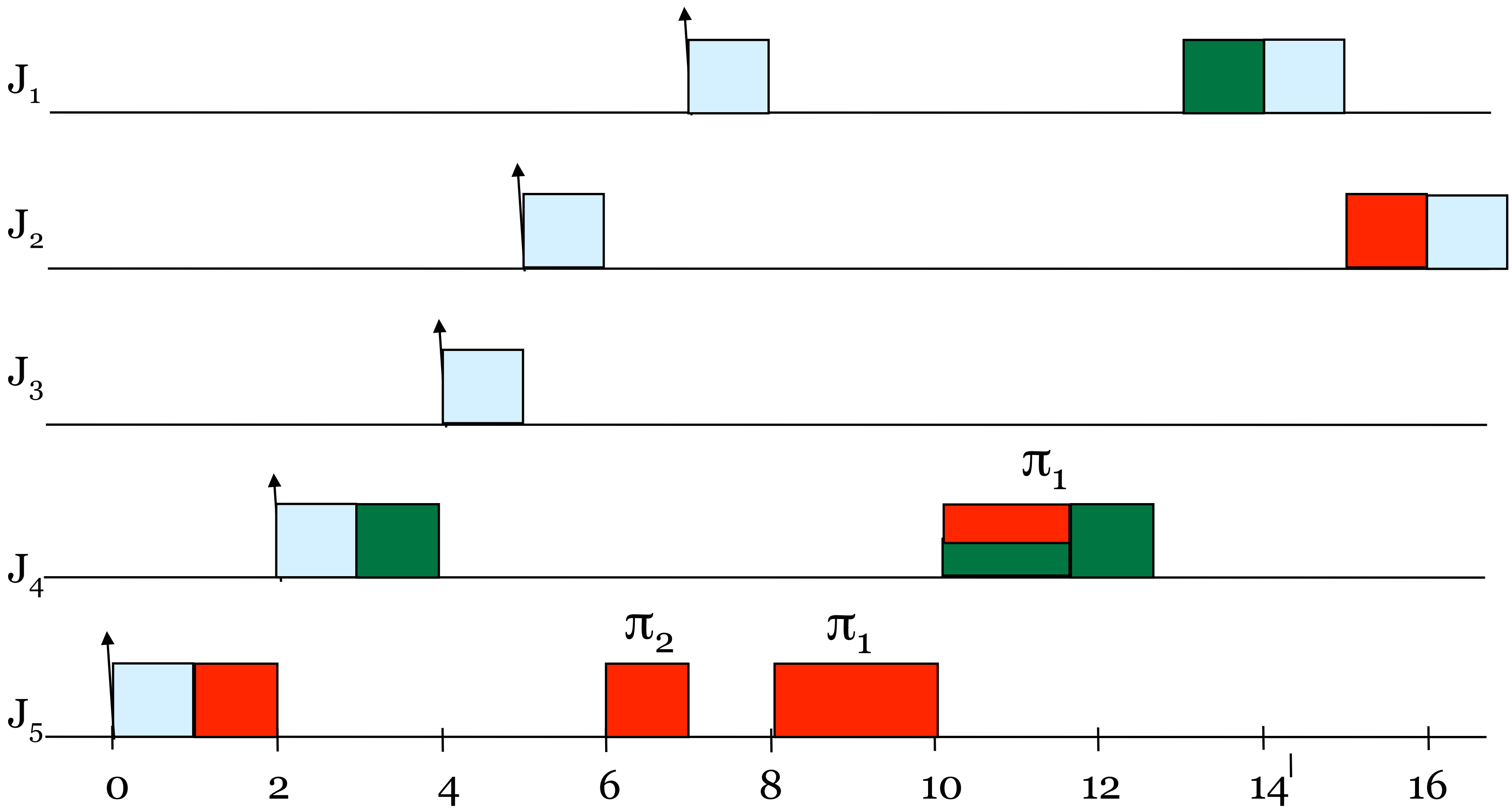


By allowing transitive priority inheritance, the protocol we have discussed extends to nested resource usage as well

A more complicated example

Job	Release time	Execution time	Critical sections
J1	7	3	[**Green**; 1] (from 1)
J2	5	3	[**Red**;1] (from 1)
J3	4	2	
J4	2	6	[**Green**; 4 [**Red**; 1.5]] (from 1, 2)
J5	0	6	[**Red**; 4] (from 1)

Here's the schedule according to PIP



Properties of PIP

- We have two kinds of blocking with the PIP: **direct blocking** and **inheritance blocking**
  - J_2 directly blocked by J_5 during interval (6,10]
  - J_2 inheritance blocked by J_4 during interval (10,13]
  - J_3 inheritance blocked by J_5 during interval (6,7]
- Jobs can **transitively** block each other
  - @ 8, J_5 blocks J_4 and J_4 blocks J_1

PIP Schedulability Analysis

Lemma 1: Job J_k waits for at most one completion of a critical section of job that blocks it (directly or indirectly), **regardless of how many times J_k is using each lock.**

Proof sketch: since k has higher priority, once J_i gets out, J_k will not let it begin something else.

Lemma 2: If job k uses lock s, then it can be blocked for the duration of at most one critical section guarded by s, regardless how many times s is used by how many jobs.

Proof sketch: k has higher priority (by definition) than all jobs that try to block it. Hence, the first time k tries to get a lock s, it will be blocked for at most the time it takes for the lower-priority job inside to get out. The following times, no lower-priority job may have gotten inside a section guarded by s.

A slight modification...

Job	Release time	Execution time	Critical sections
J1	7	3	[**Green**; 1] (from 1)
J2	5	3	[**Red**;1] (from 1)
J3	4	2	
J4	2	6	[**Green**; 4 [**Red**; 1.5]] (from 1, 2)
J5	0	6	[**Red**; 4 **[Green; 1]**] (from 1, 2)

Locks and Deadlocks

```
J5:
do {
  lock (Red)
  lock (Green)
  use (Red, Green)
  unlock (Green)
  unlock (Red)
}
```

```
J4:
do {
  lock (Green)
  lock (Red)
  use (Red, Green)
  unlock (Red)
  unlock (Green)
}
```

Possible result: J5 gets **Red**, J4 gets **Green** → deadlock!

How can we fix the problem?

Need Partial/Total Order of Locks

- Choose a total order of all locks: $L_1 < L_2 < \ldots < L_n$
  - Or a partial order for subsets of locks used by same tasks, e.g., $L_1 < L_3 < L_4$ and $L_2 < L_5$
- When you need both L_i and L_j, get that lock first that comes first in the order

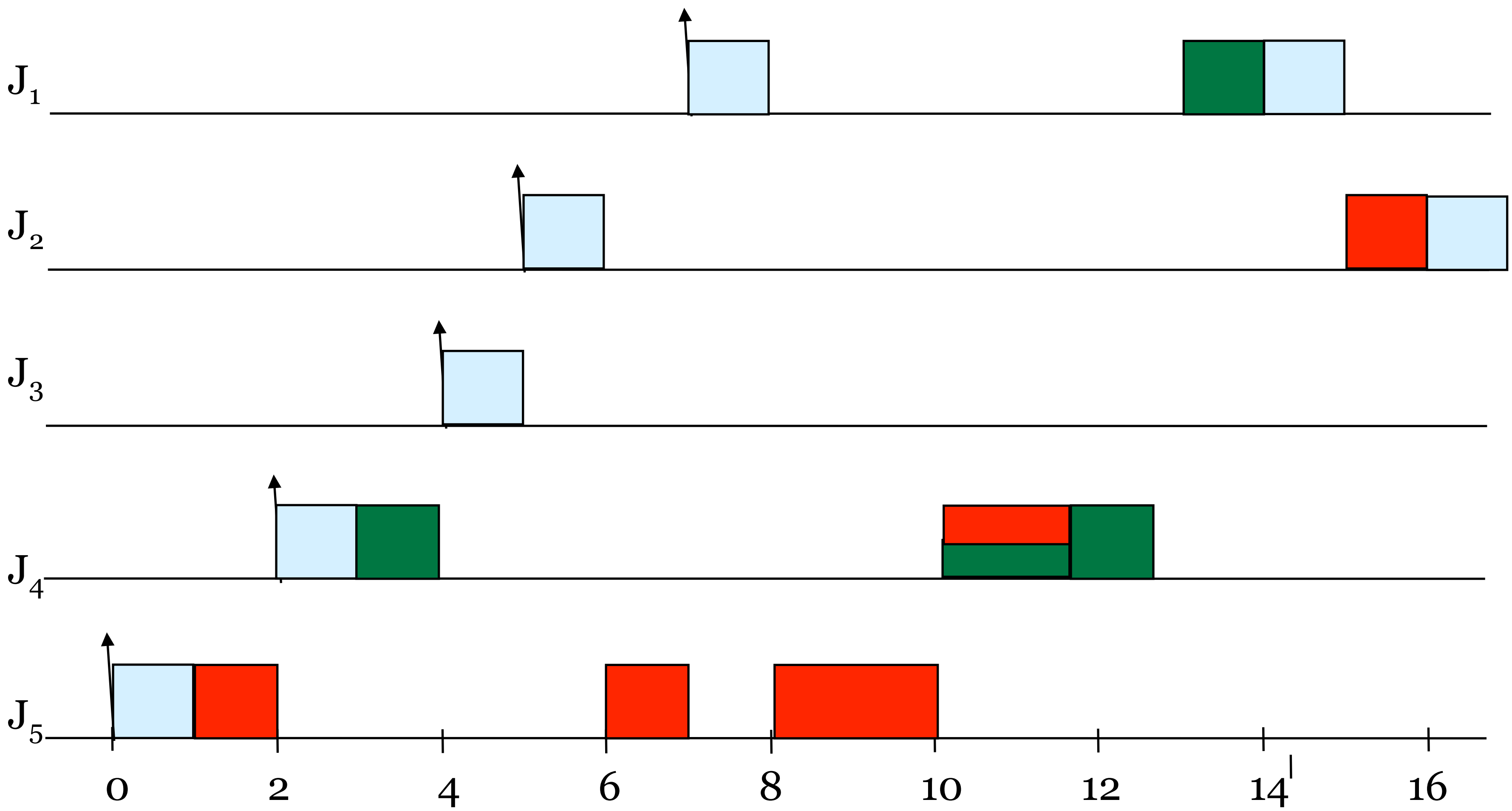
Is this the best protocol for high-priority tasks?

- Assume we know what tasks request what resources
  - Can this info. be used to the advantage of high-prio tasks?
  - If yes, how?

Let’s revisit our PIP example…

Job	Release time	Execution time	Critical sections
J1	7	3	[**Green**; 1] (from 1)
J2	5	3	[**Red**;1] (from 1)
J3	4	2	
J4	2	6	[**Green**; 4 [**Red**; 1.5]] (from 1, 2)
J5	0	6	[**Red**; 4] (from 1)

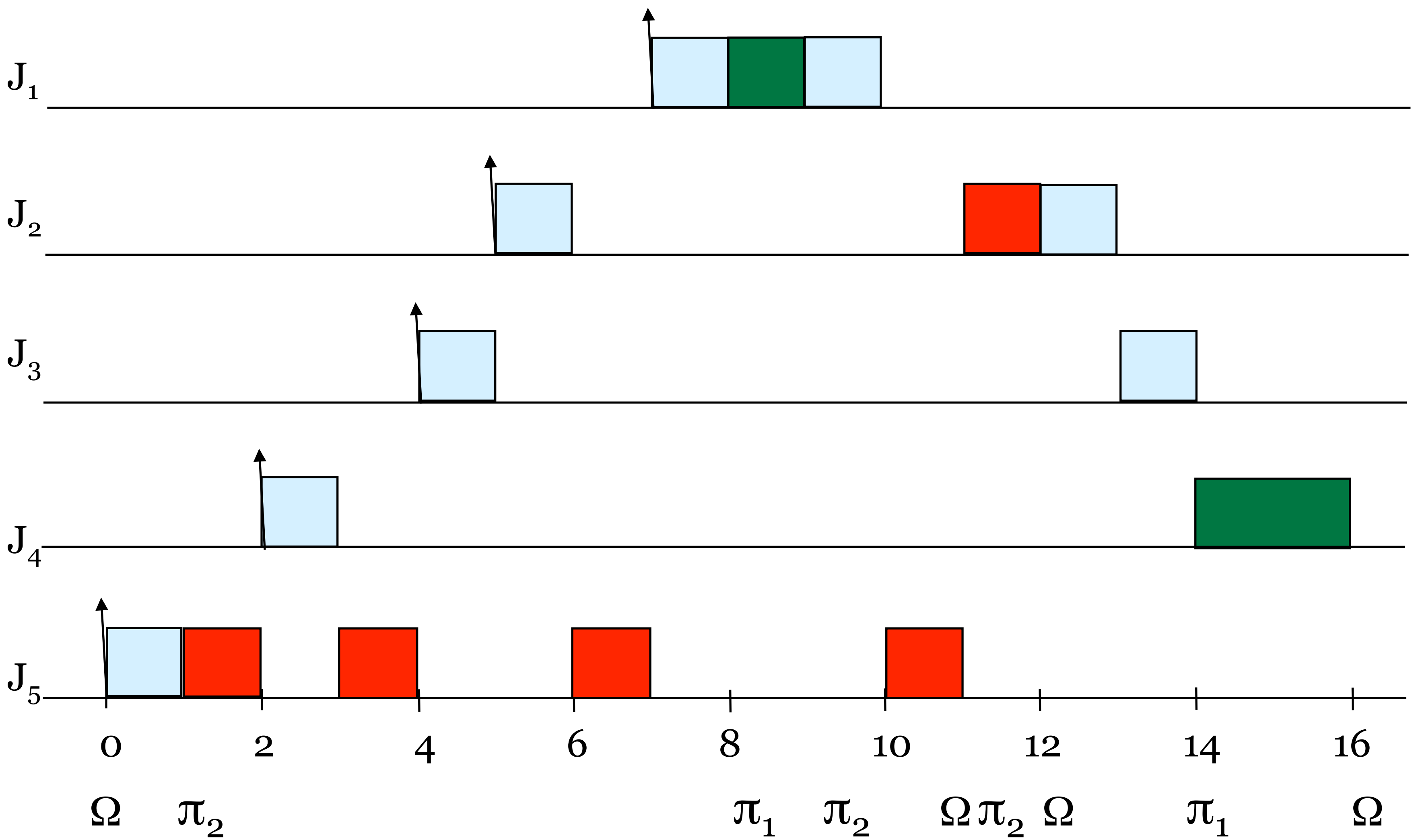
Here's what the timeline looked like



Is this the best we can do for high-prio tasks?

- Assume we know when and what resources tasks request
  - Can this info. be used to reduce blocking of high-prio tasks?
  - If yes, how?
  - Basic idea
    - Identify highest priority task using every resource
      - Resource ceiling
    - Keep track of resources currently in use
      - System ceiling
    - Prevent locking of resource by task that has lower priority than system ceiling

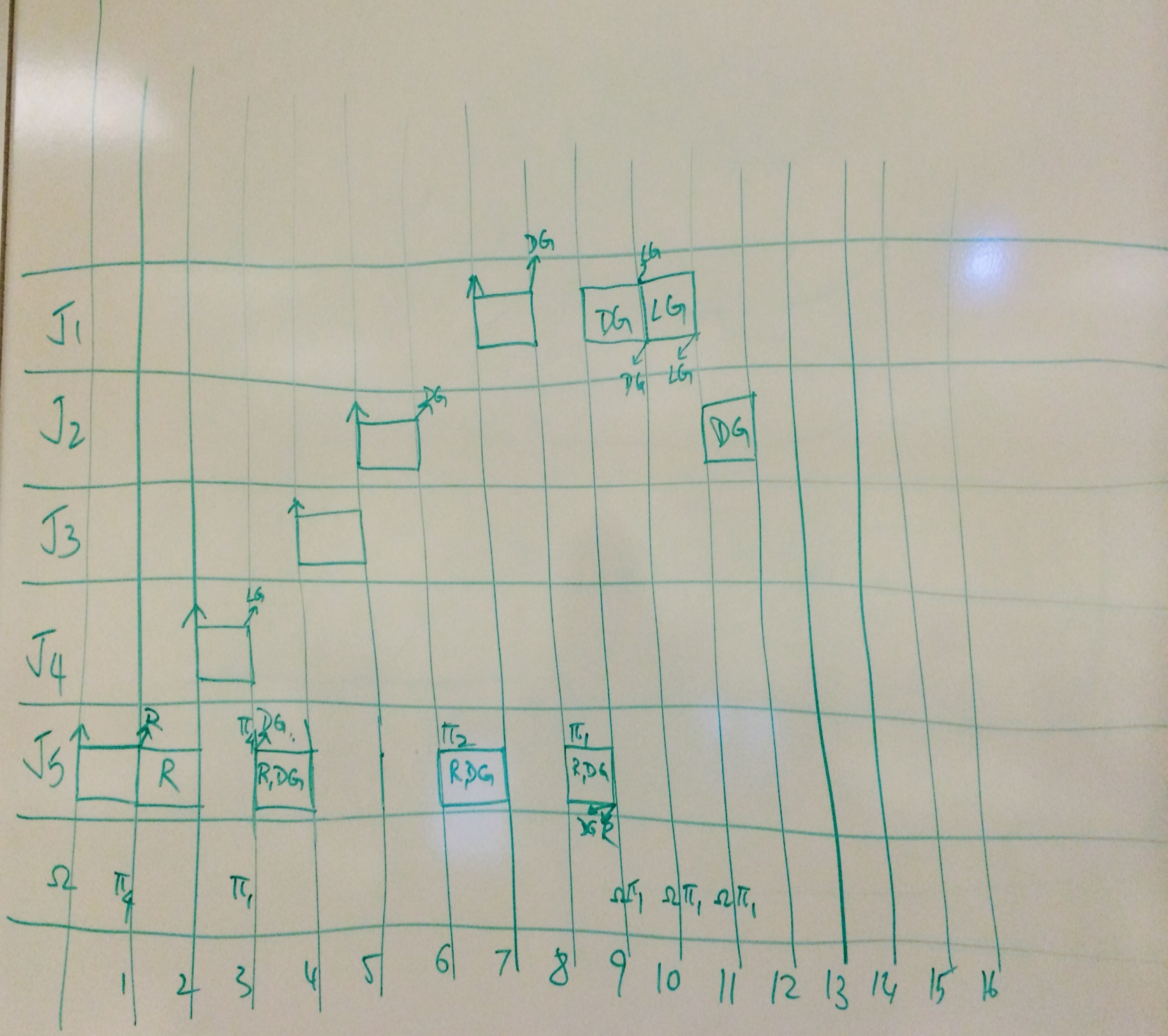
How does the schedule change?



Yet another "complicated" example

Job	Release time	Execution time	Critical sections
J1	7	3	[**DarkGreen**; 1] (from 1), [**LightGreen**; 1] (from 2)
J2	5	3	[**DarkGreen**;1] (from 1)
J3	4	2	
J4	2	6	[**LightGreen**; 4 [**Red**; 1]] (from 1, 2)
J5	0	6	[**Red**; 4 [**DarkGreen**; 3]] (from 1, 2)

$DG \rightarrow \pi_1 \quad LG \rightarrow \pi_1 \quad R \rightarrow \pi_4$



The Priority Ceiling Protocol (Sha, Rajkumar, Lehoczky)

- **Two key assumptions:**
  - The assigned priorities of all jobs are fixed (as before).
  - The resources required by all jobs are know *a priori* before the execution of any job begins

Definition: The **priority ceiling** of any resource R is the highest priority of all the jobs that require R, and is denoted by $\Pi(R)$

Definition: The **current priority ceiling** $\Pi'(t)$ of the system is equal to the highest priority ceiling of the resources currently in use, or Ω if no resources are currently in use (Ω is a priority lower than any real priority)

PCP Definition

1. Scheduling Rule:

(a) At release time t, current priority $\pi(t)$ of every job J = assigned priority. Job remains at this priority except under conditions of rule 3.

(b) Jobs J scheduled preemptively + strict priority at its current priority $\pi(t)$.

2. Allocation Rule: Whenever a job J requests a resource R at time t, either:

(a) R is held by another job -- J's request fails + J becomes blocked -- or

(b) R is free.

(i) If J's prio $\pi(t)$ > current prio ceiling $\Pi'(t)$, R is allocated to J.

(ii) If J's prio $\pi(t) \leq$ ceiling $\Pi'(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling = $\Pi'(t)$; otherwise, J's request is denied and J becomes blocked.

3. Priority-Inheritance Rule: When J becomes blocked, the job J_l that blocks J inherits the current priority $\pi(t)$ of J. J_l executes at its inherited priority until it releases every resource whose priority ceiling is $\geq \pi(t)$ (or until it inherits an even higher priority); at that time, the priority of J_l returns to the priority $\pi_l(t')$ it had at the time t' when it was granted the resources.