

# Resource Access Control

## 1. Problems

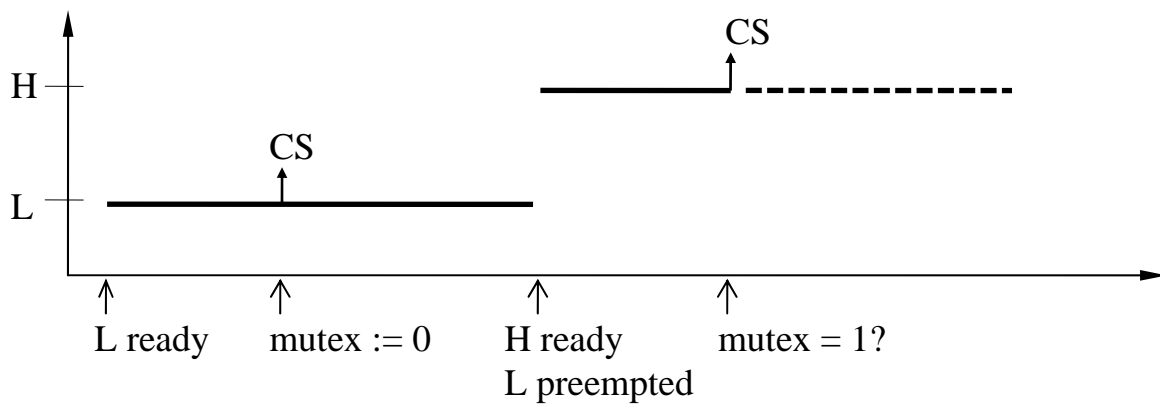
- **Priority Inversion**

*Assumptions:*

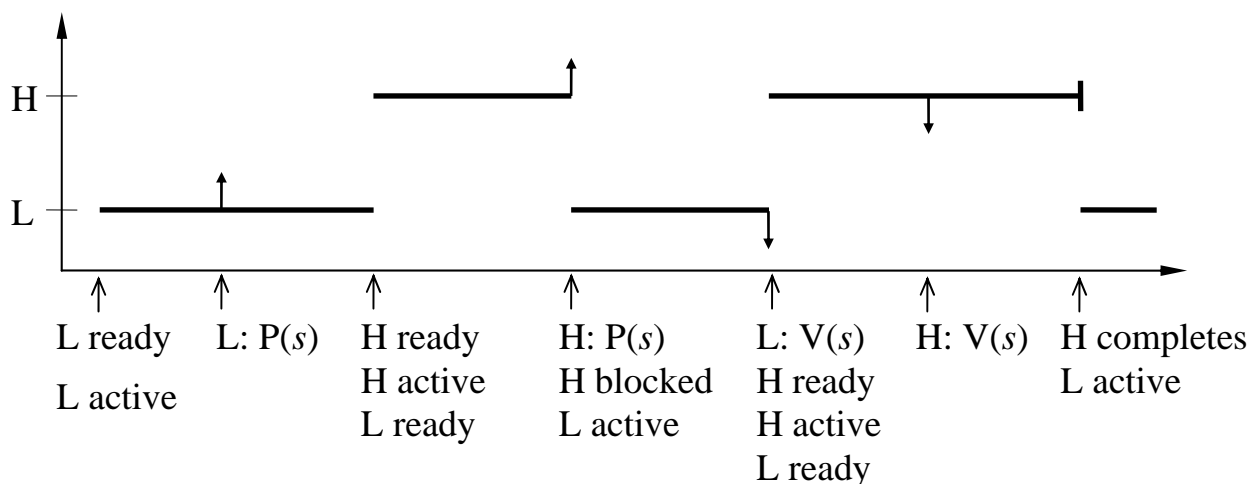
Jobs uses resources in a mutually exclusive manner (critical section CS).

Preemptive priority-driven scheduling. 1 processor.

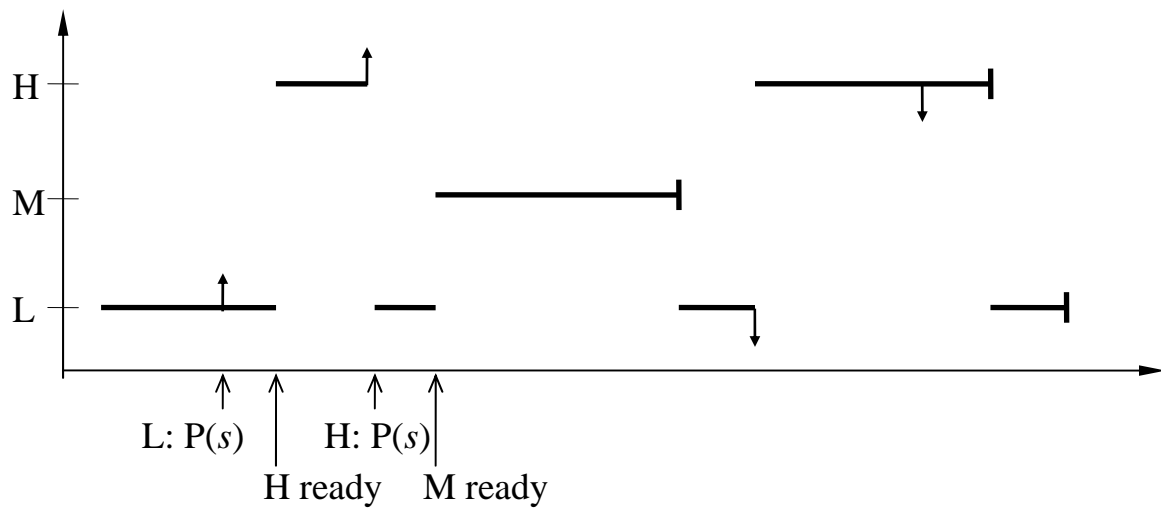
- *Busy waiting and priority inversion*



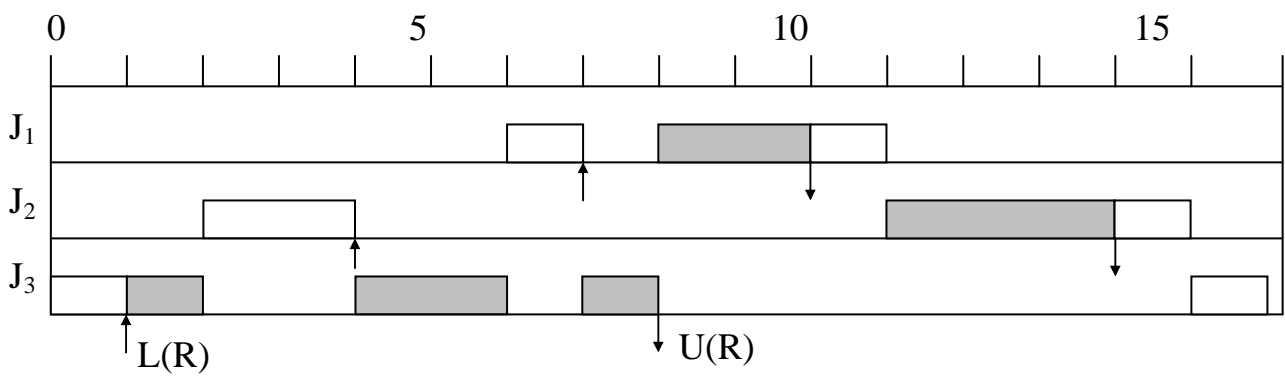
- *Semaphores and priority inversion*



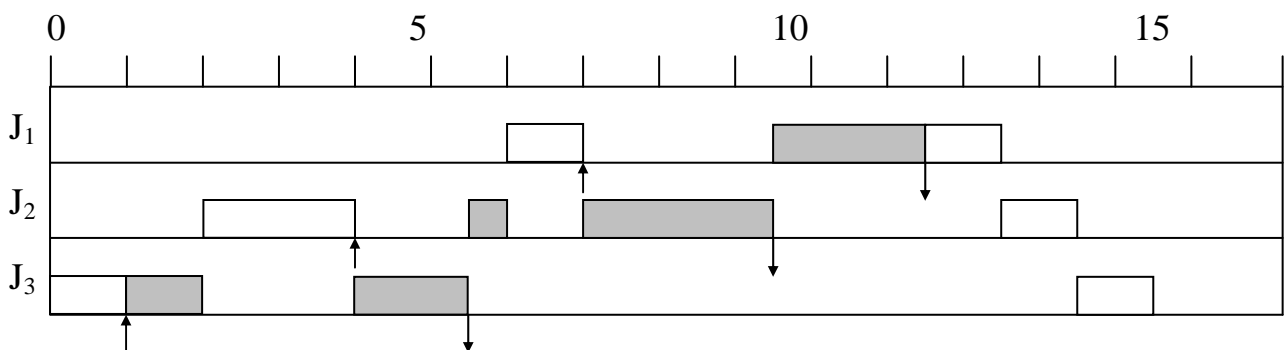
M: medium-prioritized job (not using  $s$ )



### • Timing Anomalies



Reduction of resource usage of  $J_3$  by 1.5:



### • Deadlocks

## 2. Assumptions and Notations

- 1 processor, preemptive priority-driven scheduling  
jobs are not self-suspending
- $R_1, \dots, R_r$  resources; nonpreemptable, exclusive
- $L(R_k), U(R_k)$  require/release of  $R_k$ ; release: LIFO  
 $\uparrow R_k \quad \downarrow R_k$
- $J_1, \dots, J_n$  jobs;  
 $J_h, J_l$ : job of high/low priority  
 $p_1, \dots, p_n$  “assigned” priorities (highest priority: 1);  
w.l.o.g.  $J_i$  ordered according to priorities  
 $p_i(t)$  current priority of  $J_i$
- ***Jobs conflict with one another***  
require the same resource  
***Jobs contend for a resource***  
one job requests the resource that another job already has  
***Blocked job***  
scheduler cannot grant the requested resource
- ***Priority inversion***  
 $J_l$  executes while  $J_h$  is blocked

### 3. Priority Inheritance

for preemptive priority-driven scheduling

SHA et al., 1990

- **Basic Priority-Inheritance Protocol**

- (1) *Scheduling Rule*

- A ready job  $J$  is scheduled according to its current priority  $p(t)$ ;  
at release time  $t$ :  $p(t) := p$ .

- (2) *Allocation Rule*

- $J$  requests  $R$  at time  $t$ .

- (a)  $R$  free:  $R$  is allocated to  $J$  until  $J$  releases  $R$ .

- (b)  $R$  not free: request is denied,  $J$  becomes blocked.

- (3) *Priority-Inheritance Rule*

- When  $J$  becomes blocked by  $J_l$ , so  $J_l$  inherits the current priority of  $J$ , i.e.  $p_l(t) := p(t)$ .

- $J_l$  executes at this priority until it releases  $R$  at time  $\tilde{t}$ .

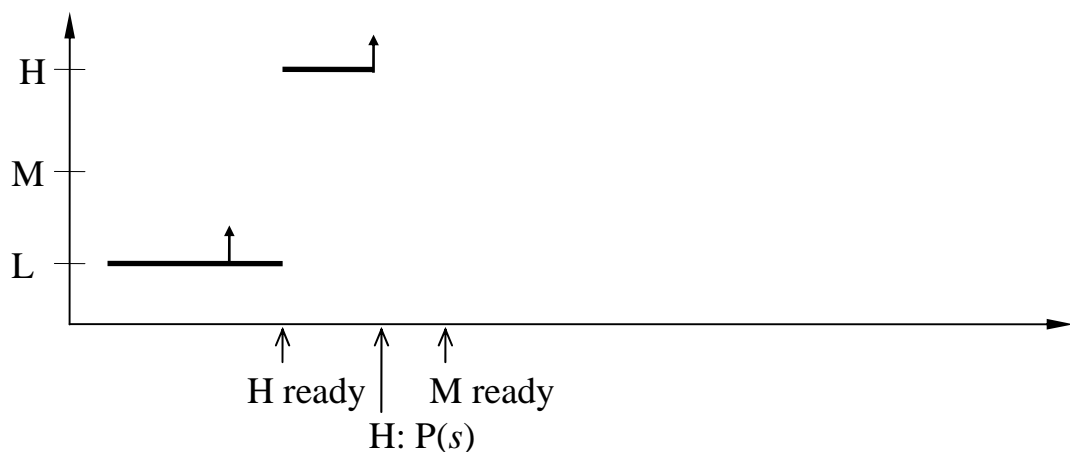
- Now the priority of  $J_l$  returns to its previous priority:

- $$p_l(\tilde{t}) := p_l(t') \quad t': \text{time when } J_l \text{ acquires } R.$$

- **Example**

- 2 jobs: no effect!

- 3 jobs:





## 4. Priority Ceilings – Basic Priority-Ceiling Protocol

SHA/RAJKUMAR/LEHOCZKY, 1990

- **Assumptions and Notations**

- 1 processor, preemptive priority-driven scheduling  
no self-suspension
- Assigned priorities  $p_i$  are fixed.  
priorities: natural numbers, 1 highest,  $\Omega$  lowest priority.
- The resources required by all jobs are known a priori.
- $P(R)$  *priority ceiling of R*  
highest priority of all jobs that require  $R$ .
- $\hat{P}(t)$  *priority ceiling of the system at time t*  
highest priority ceiling of all resources that are in use at time  $t$ .

- **Basic Priority-Ceiling Protocol**

- (1) **Scheduling Rule**

At release time  $t^{rel}$  of  $J$ :  $p(t^{rel}) := p$ .

- (2) **Allocation Rule**

$J$  requests  $R$  at time  $t$ .

- (a)  $R$  held by another job: request denied,  $J$  blocked (“on  $R$ ”).

- (b)  $R$  free:

- ( $\alpha$ )  $p(t) \succ \hat{P}(t)$ :  $R$  is allocated to  $J$ .

- ( $\beta$ ) otherwise:  $R$  is allocated to  $J$  only if  $J$  is the job holding the resource(s)  $R'$  with  $P(R') = \hat{P}(t)$ .

Otherwise the job becomes blocked.

- (3) **Priority-Inheritance Rule**

When  $J$  becomes blocked by  $J_i$ ,  $J_i$  inherits  $J$ 's current priority  $p(t)$ .

$J_i$  (preemptive) executes at this priority until it releases every resource whose priority ceiling is at least  $p_i(t)$ .

At that time,  $J_i$ 's priority returns to  $p_i(t')$

( $t'$ : time when it was granted the resource).

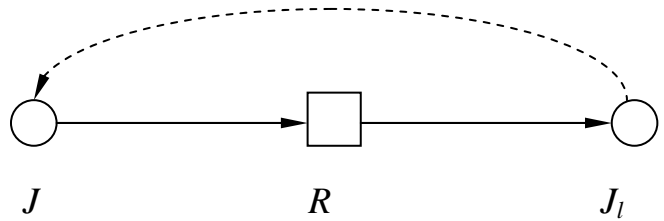
- **Example**



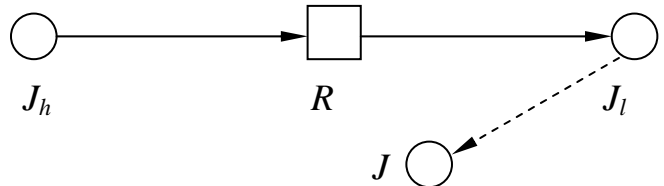
• **Properties**

- Difference to priority inheritance: *three* ways to blocking

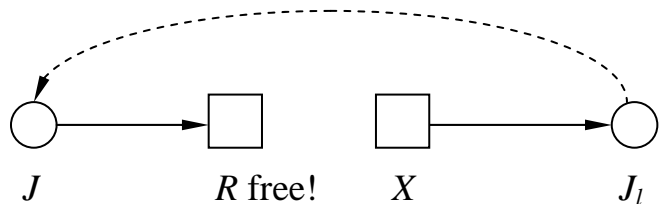
direct:



inheritance:

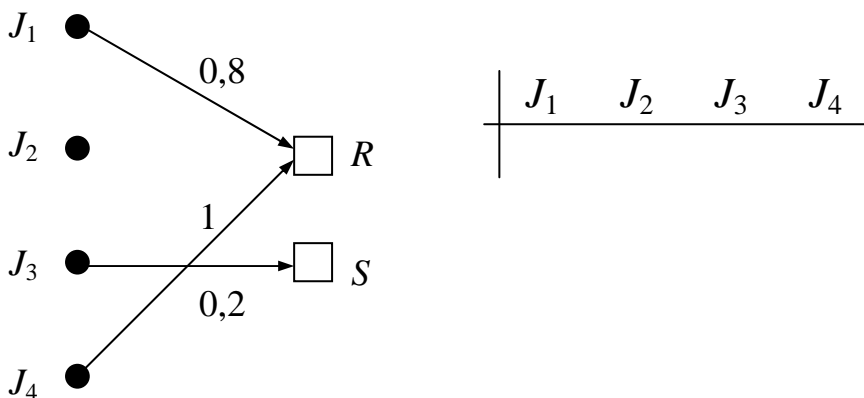


ceilings:



- Deadlocks can never occur.
- A job can be blocked for at most *one* resource request.

Computation of blocking time – Example:



- There can be no transitive blocking.

## 5. Stack-Based, Priority-Ceiling Protocol

- **Further Assumptions**

- Common run-time stack for all jobs (no self-suspending!).
- Stack space of an active job is on the top of the stack (preemption!).
- Stack space is freed when the job completes.

- **Protocol**

(0)  $\hat{P}(t) = \Omega$ , when all  $R$  are free.

$\hat{P}(t)$  is updated whenever a resource is allocated or freed.

(1) ***Scheduling Rule***

After  $J$  is released, it is blocked until  $p \succ \hat{P}(t)$ .

Priority-driven scheduling based on assigned priorities. (!)

(2) ***Allocation Rule***

Whenever a job requests a resource, it is allocated the resource. (!!!)

- **Properties**

- When a job begins to execute, all the resources it will ever need are free.
- Both protocols result in the same longest blocking time of a job.
- Deadlocks can never occur.

• **Exercise 1.**

5 jobs  $J_i$ ;  $p_i = i$ ,  $r_i$  release time,  $e_i$  execution time

2 resources A, B;  $a_i$  begin of usage (relative to  $r_i$ )

$b_i$  duration of usage

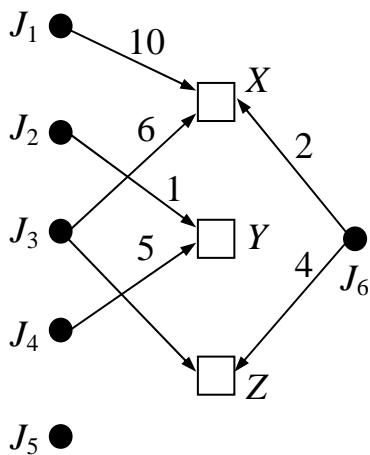
$i$	$r_i$	$e_i$	A: $a_i$	$b_i$	B: $a_i$	$b_i$
1	7	3	1	1	-	-
2	5	3	-	-	1	1
3	4	2	-	-	-	-
4	2	6	1	4	3	1.5
5	0	6	-	-	1	4

To find: schedule and blocking times

- without priority inheritance
  - using priority-inheritance protocol
  - using priority-ceiling protocol
  - using stack-based priority-ceiling protocol
- and progress of  $\hat{P}(t)$  for c) and d).

• **Exercise 2.**

Compute the blocking times (distincted by the reason of blocking) in the case of the given system of jobs!



	direct					pr. inheritance					pr. ceiling				
	$J_2$	$J_3$	$J_4$	$J_5$	$J_6$	$J_2$	$J_3$	$J_4$	$J_5$	$J_6$	$J_2$	$J_3$	$J_4$	$J_5$	$J_6$
$J_1$															
$J_2$															
$J_3$															
$J_4$															
$J_5$															