Technische Universität Dresden

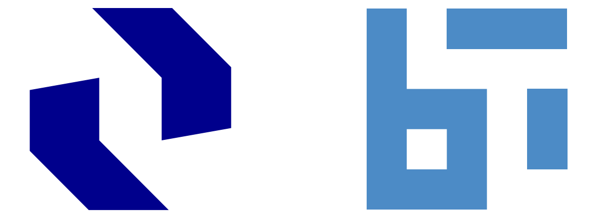
barkhausen institut

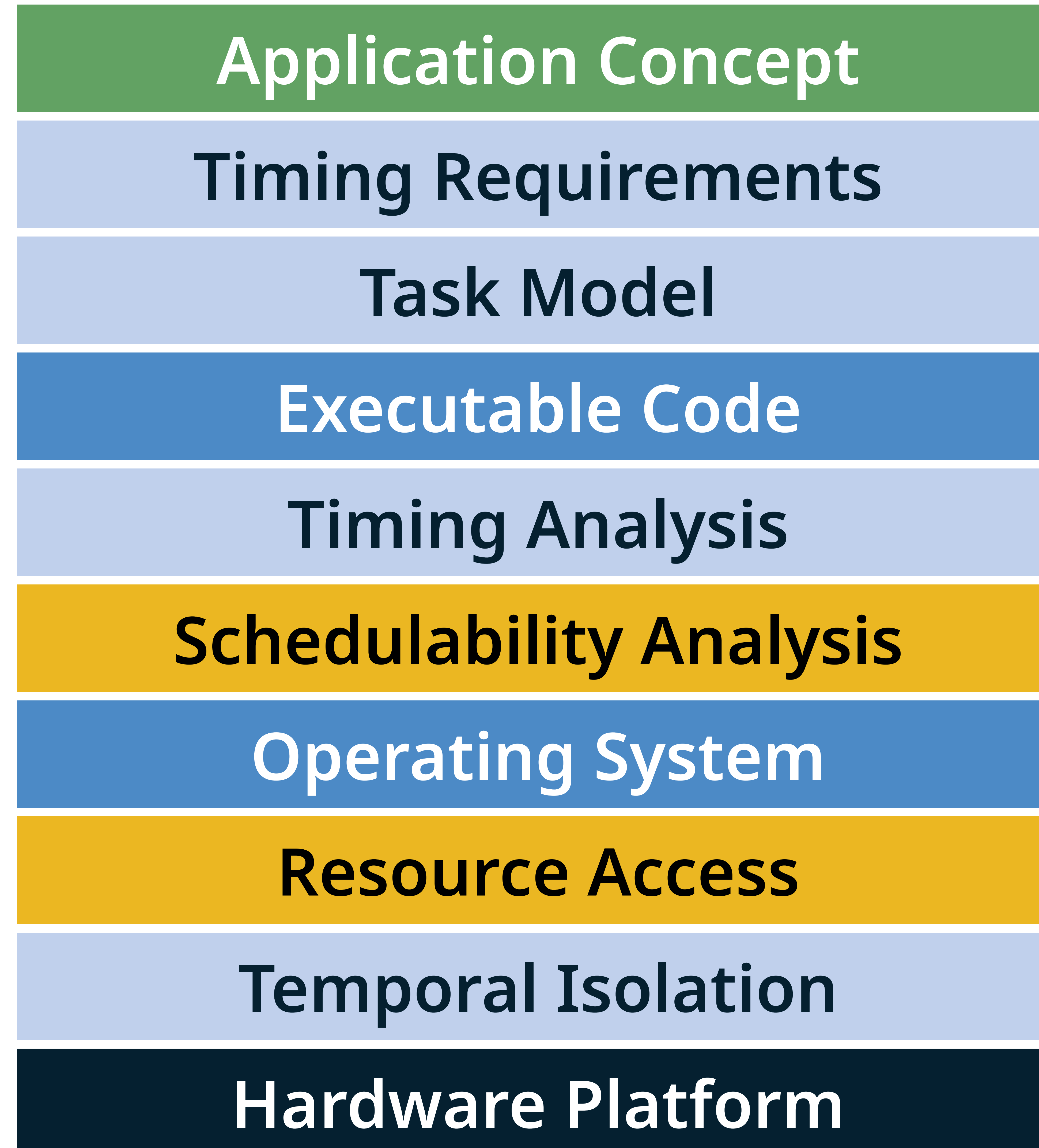
Resource Access Protocols

Real-Time Systems

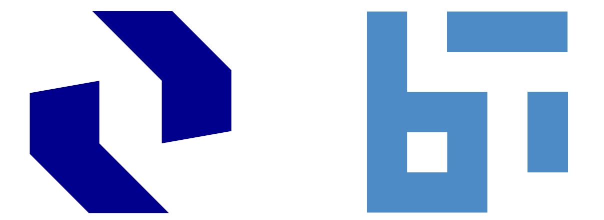
Michael Roitzsch

Real-Time Systems Technology Stack





Problem: Priority Inversion



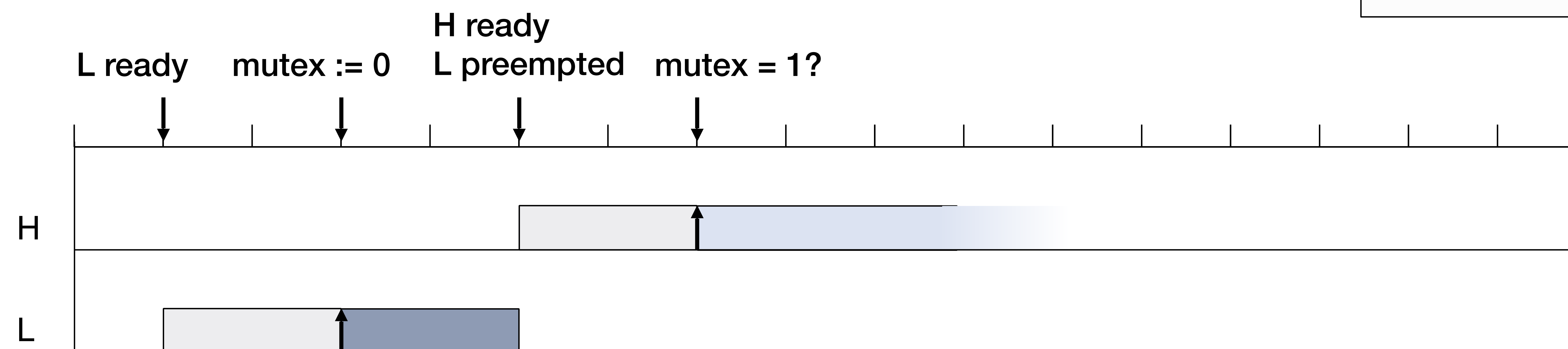
Assumptions

- jobs use resources in a mutually exclusive manner
- preemptive priority-driven scheduling
- fixed task priorities
- 1 processor

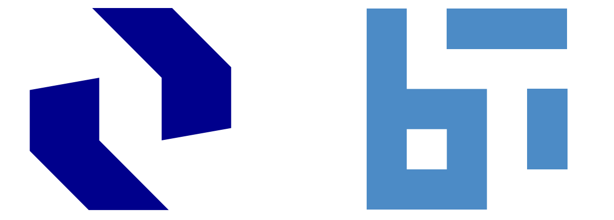
Busy waiting leading to priority inversion:

Declaration
for the following slides

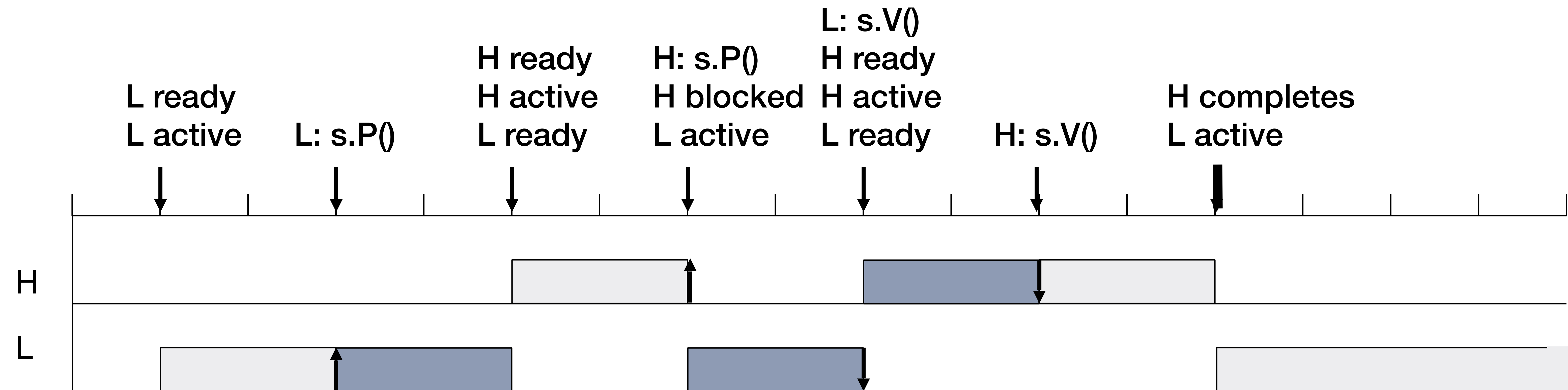
L(R) ↑ U(R) ↓



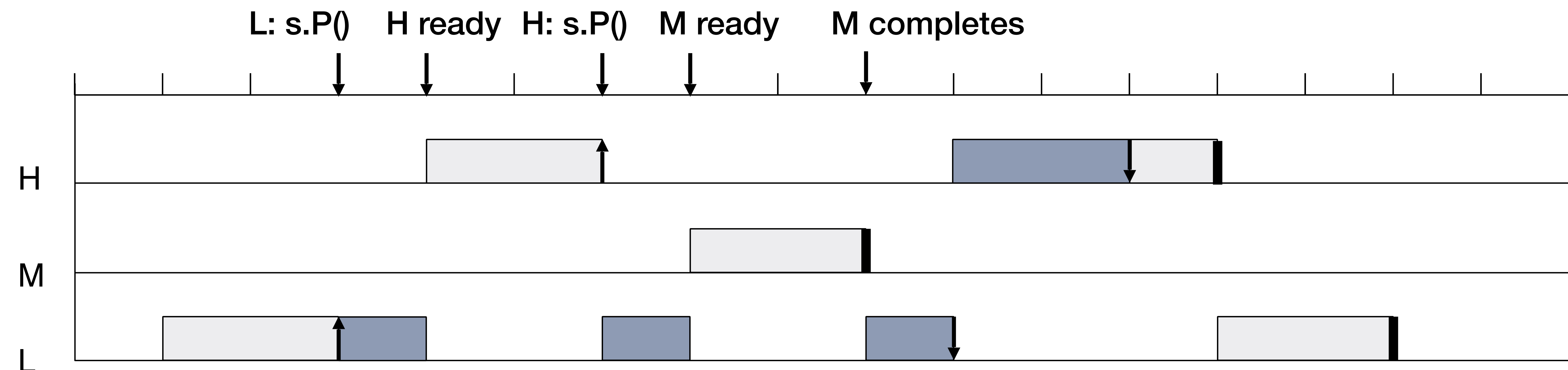
Problem: Priority Inversion



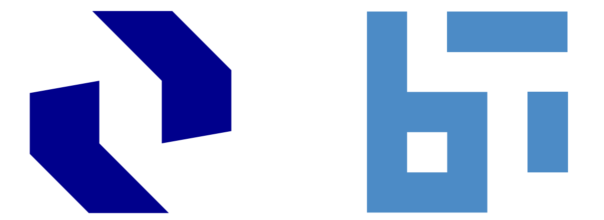
Semaphores leading to priority inversion:

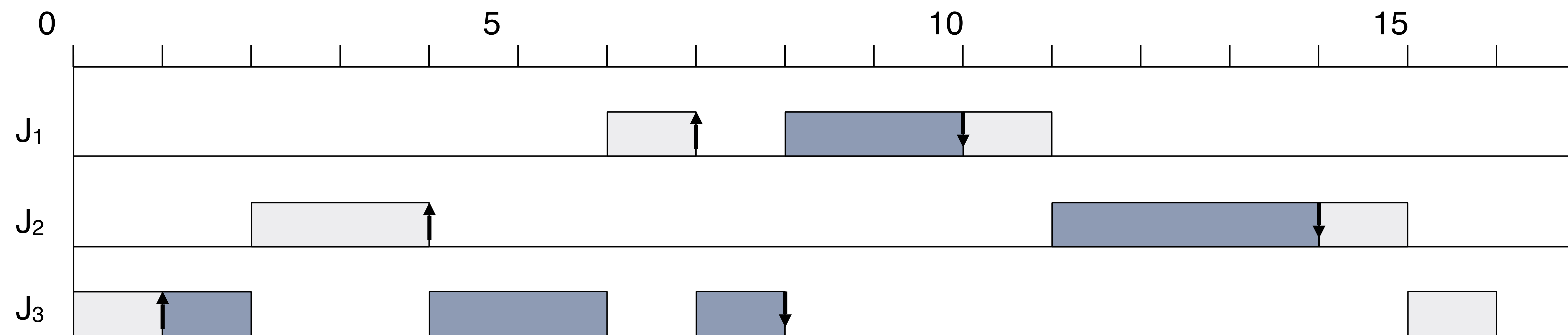


M: medium-prioritized job (not using s)

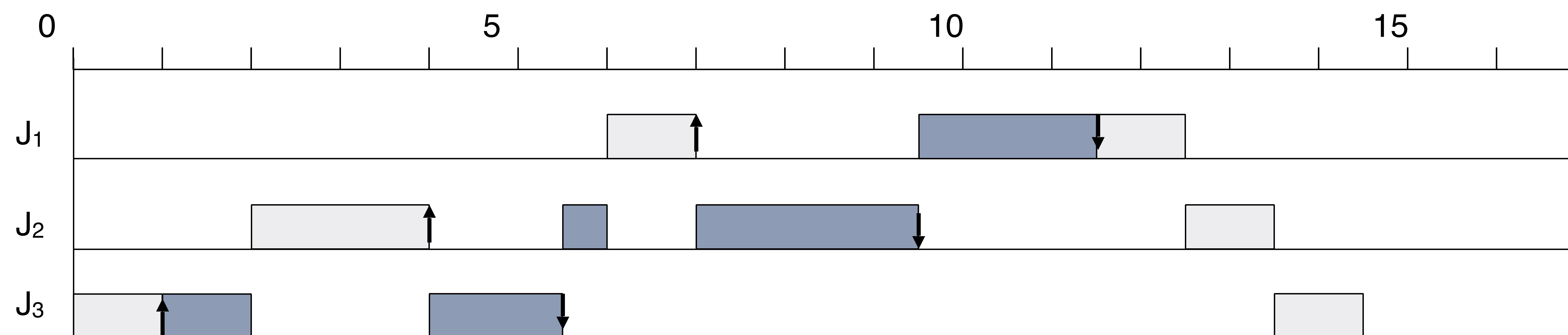


Problem: Timing Anomalies

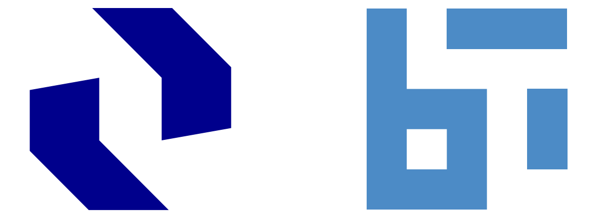




Reduction of resource usage of J_3 by 1.5:

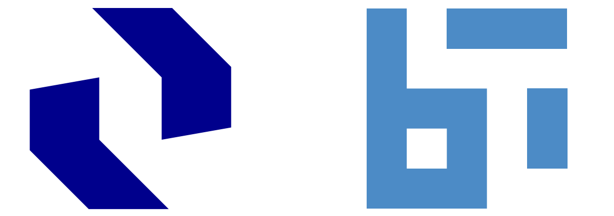


Problem: Deadlocks



- exclusive resources
- non-preemptive resources
- sequential acquire
- cyclic wait-condition

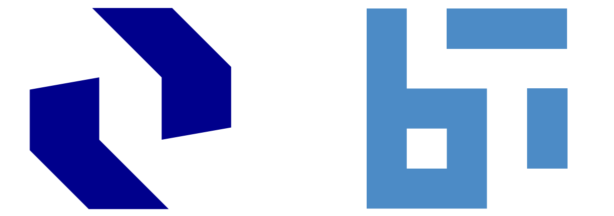
Assumptions and Notations



1 processor, preemptive priority-driven scheduling, no self-suspension

$R_1, \ldots, R_r$	resources; nonpreemptable, exclusive
$L(R_k), U(R_k)$	acquire/release of R_k; release: LIFO
$\uparrow R_k \quad \downarrow R_k$	
$J_1, \ldots, J_n$	jobs
J_h, J_l	job of high/low priority
$p_1, \ldots, p_n$	assigned priorities (highest priority: 1) J_i ordered according to priorities (w.l.o.g.)
$p_i(t)$	current priority of J_i at time t

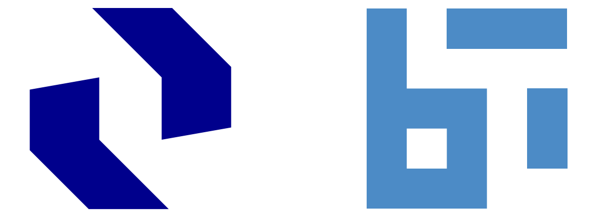
Assumptions and Notations



- **Jobs conflict with one another**
  operate with a common resource
- **Jobs contend for a resource**
  one job requests the resource that another job already owns
- **Blocked job**
  scheduler does not grant the requested resource
- **Priority inversion**
  J_l executes while J_h is released, but blocked

Priority Inheritance

Basic Priority Inheritance Protocol (Sha et al., 1990)



(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.

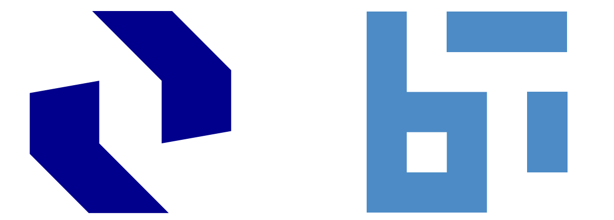
(b) R not free: request is denied, J is blocked.

(3) **Priority-Inheritance Rule**

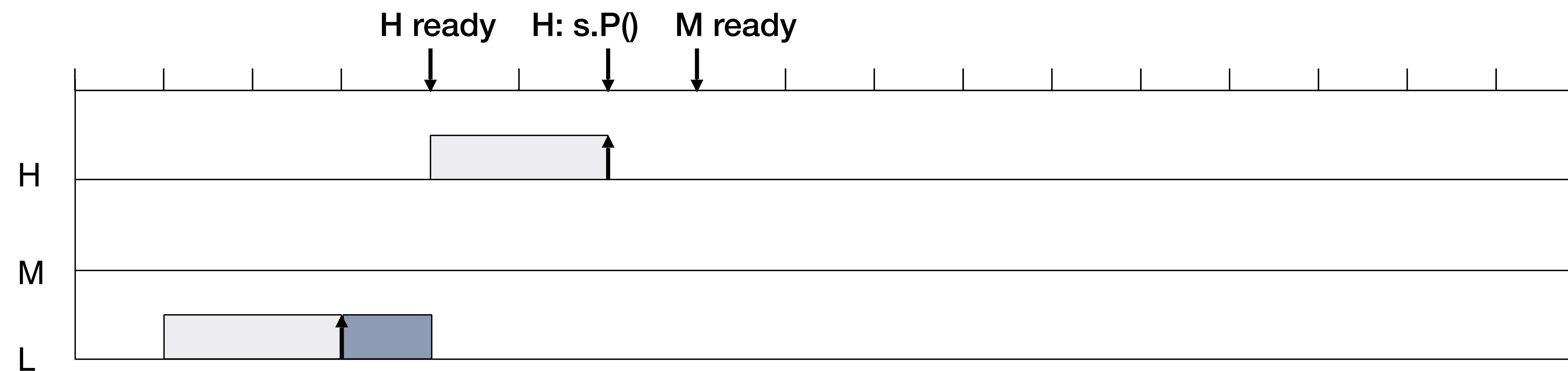
When J becomes blocked by J_l, then J_l inherits the current priority of J,
i.e. $p_l(t) := p(t)$.

- J_l executes at this priority until it releases R at time t''.
- Now the priority of Jl returns to its previous priority:
  $pl(t'') := pl(t')$ t': time when J_l acquires R.

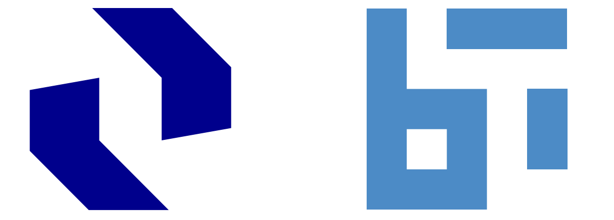
Priority Inheritance: Example



- 2 jobs: no effect!
- 3 jobs:



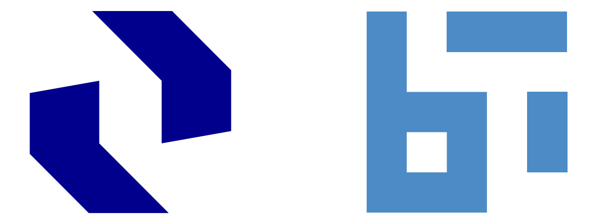
Priority Inheritance



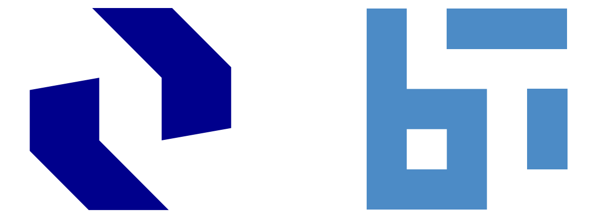
Properties

- priority inheritance is transitive
- no unbounded uncontrolled priority inversion
- priority inheritance does *not* reduce blocking times as much as possible

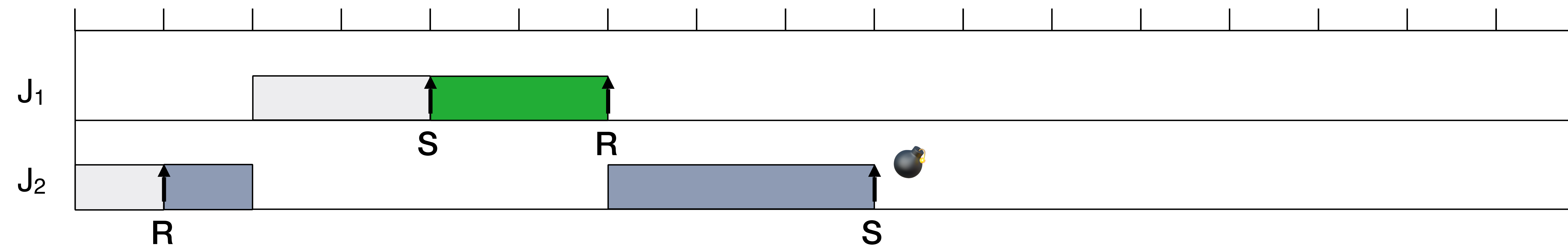
Priority Inheritance



Priority Inheritance



Priority inheritance does not prevent deadlocks.

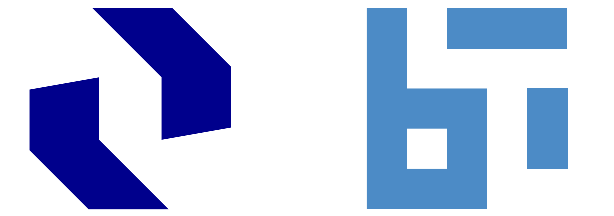


Technische Universität Dresden

barkhausen institut

Priority Ceiling

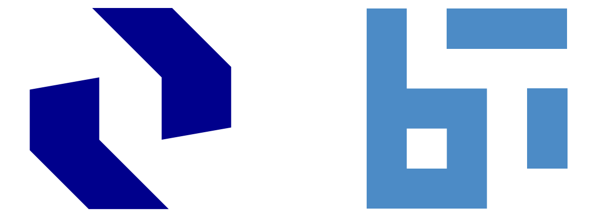
Priority Ceilings (Sha/Rajkumar/Lehoczky, 1990)



Assumptions and Notations

- 1 processor, preemptive priority-driven scheduling, no self-suspension
- assigned priorities p_i
  priorities: natural numbers, 1 highest, Ω lowest priority
- resources required by all jobs are known a priori
- $\mathbf{P(R)}$ priority ceiling of *R*
  highest priority of all jobs that require *R*
- $\mathbf{\hat{P}(t)}$ 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* of *J*: $p(t) := p$

(2) **Allocation Rule**

J requests *R* at time *t*.

(a) R held by another job: request denied, J blocks ('on R')

(b) R free:

i) $p(t) > \hat{P}(t)$: *R* is allocated to *J*

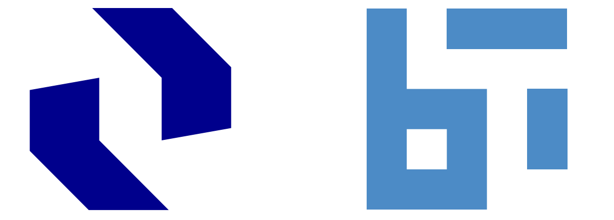
ii) otherwise: *R* is allocated to *J* only if *J* is the job holding the resource(s) *R′* with $P(R') = \hat{P}(t)$, otherwise *J* blocks

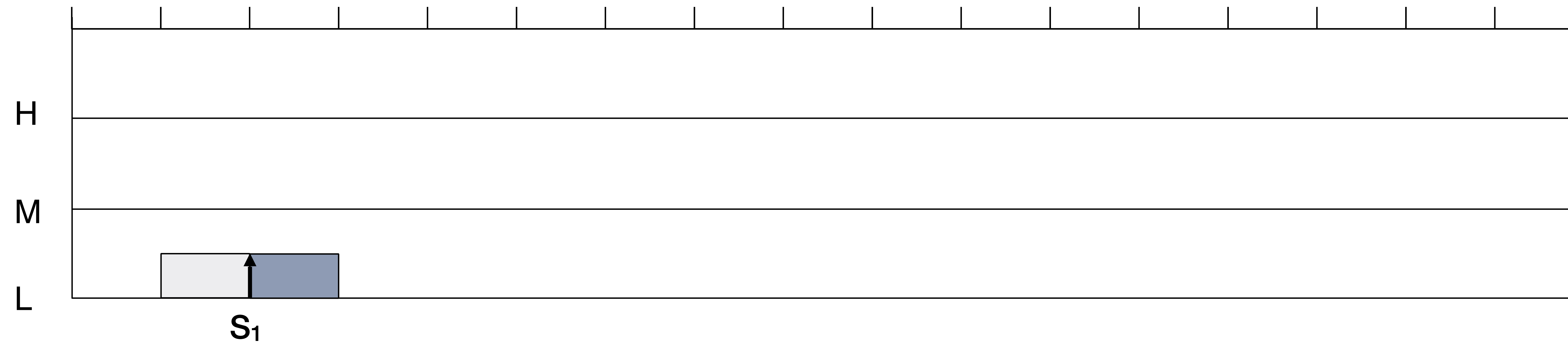
(3) **Priority-Inheritance Rule**

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

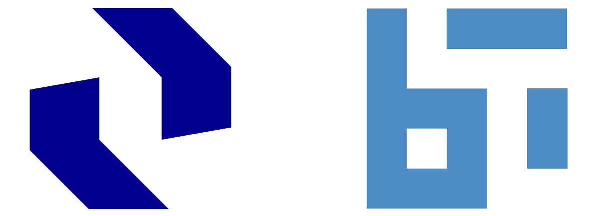
- J_l (preemptively) executes at this priority until it releases every resource whose priority ceiling is at least $p(t)$
- At that time, J_l's priority returns to $p_l(t')$ (t': time when it was granted the resource)

Basic Priority-Ceiling Protocol



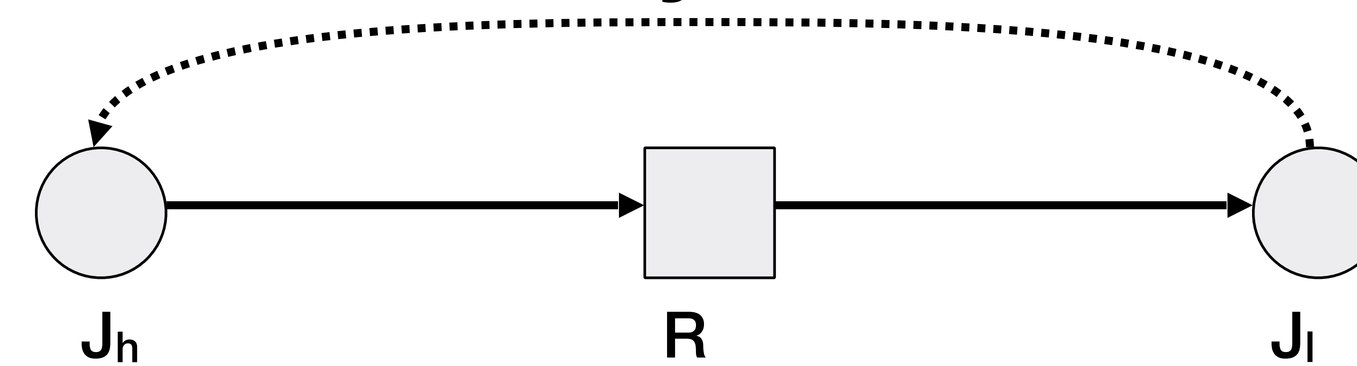


Basic Priority-Ceiling Protocol

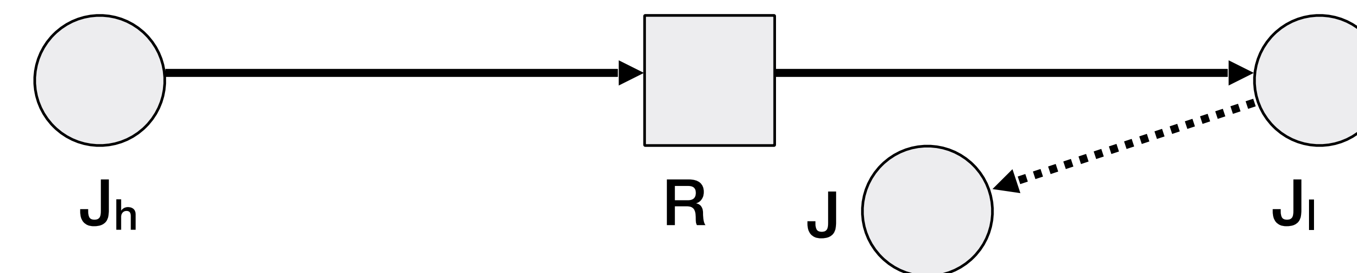


Properties

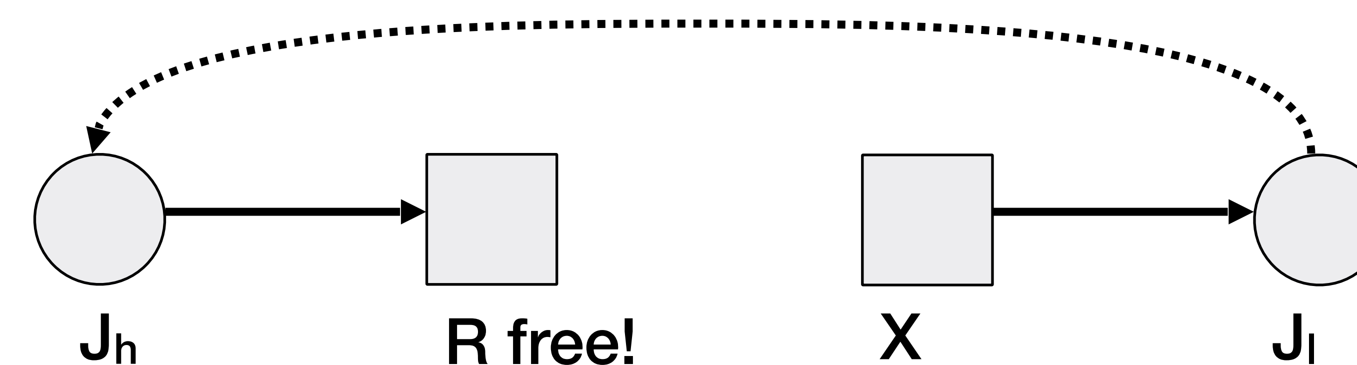
- Difference to priority inheritance: three ways to blocking
  - direct:



  - inheritance:

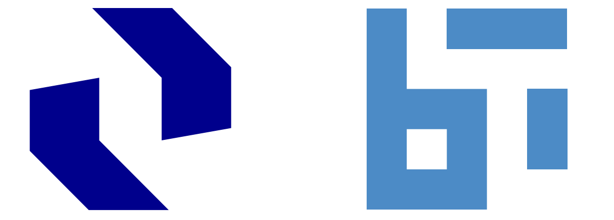


  - ceilings:



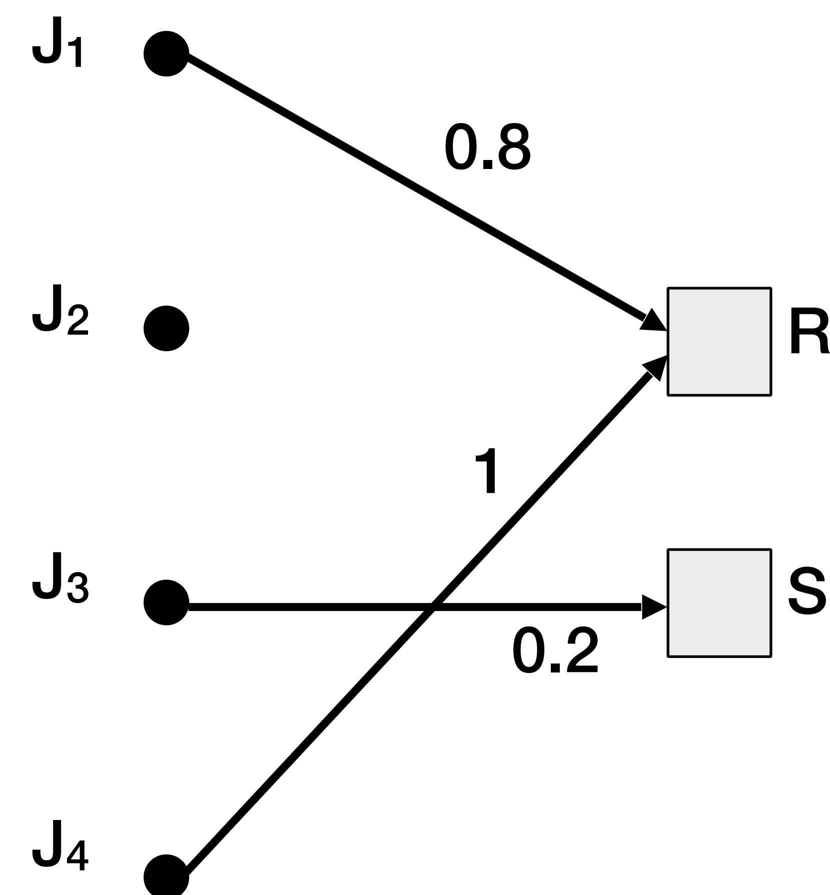
- Deadlocks can never occur
- There can be no transitive blocking

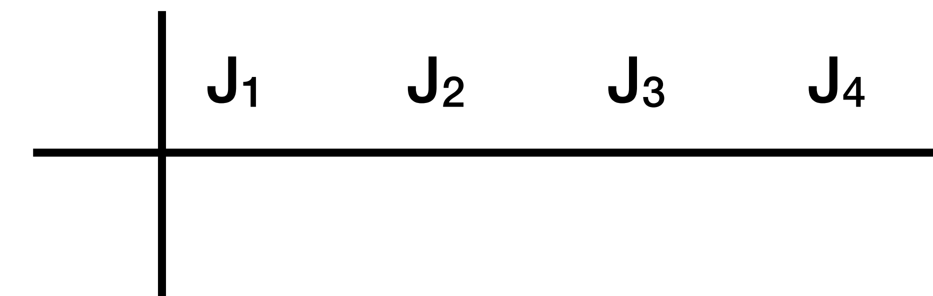
Basic Priority-Ceiling Protocol



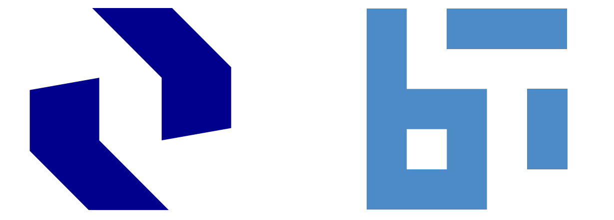
Computation of Blocking Time

- a job can be blocked for at most one resource request
- example:





Stack-Based Priority-Ceiling Protocol



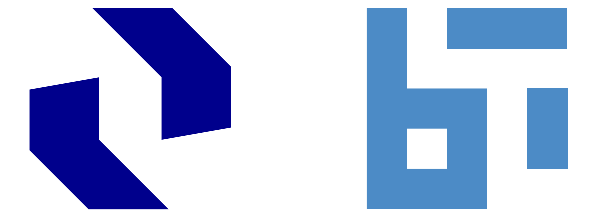
Assumptions on Execution Environment

- common run-time stack for all jobs
- stack space of the active job is on the top of the stack
- stack space is freed when the job completes
- example: embedded system with no kernel, user-level threads

Requirement

- properly nested execution of jobs:
  preemption must return where we came from

Stack-Based Priority-Ceiling Protocol



(1) $\hat{P}(t) = \Omega$, when all R are free,
$\hat{P}(t)$ is updated whenever a resource is allocated or freed

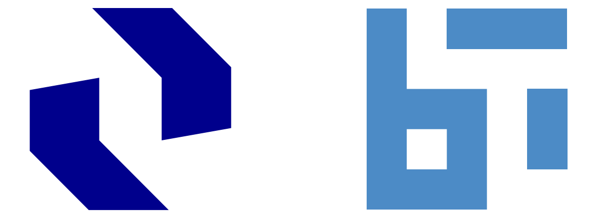
(2) **Scheduling Rule**

- after J is released, it is blocked until $p > \hat{P}(t)$
- priority-driven scheduling based on assigned priorities (!)

(3) **Allocation Rule**

- whenever a job requests a resource, it is granted the resource (!)

Stack-Based Priority-Ceiling Protocol



Properties

- When a job begins execution, all resources it will ever need are free
- both ceiling protocols result in the same longest blocking time of a job
- deadlocks cannot occur