3.b Task interactions and blocking effects

Where we allow tasks to cooperate by resource sharing, we look at how access control protocols prevent predictability hazards, and discuss their pros and cons

How can tasks cooperate?

- Tasks collaborate by exchanging data
  - Which has to happen in the face of preemptive scheduling
- Tasks have two ways of exchanging data
  - They may either send messages to one another *synchronously*
  - Or share memory and access it *asynchronously*
- Real-time tasks *cannot* synchronize with one another in the general case
  - The wait time of synchronization would be unbounded (infinite in the worst case), thus defeating timeliness
- Control means must be provided for resource sharing to be used predictably in the face of preemption
  - The incautious risks data races, deadlocks, starvation …

How does preemption happen?

- All the CPU does is to repeat a simple cycle of basic micro-operations forever (until stopped)
  - Fetch, Decode, Read, Execute, Write: one such cycle per processor instruction
- Pipelining that cycle increases throughput (# of instructions executed per unit of time)
  - Branching and jumping cause flushing of the pipeline, which incurs slowdown
- Such micro-operations *cannot* be interrupted
  - Electrons save no context: if you stop, you lose the whole pipeline!
- The *only* way to preempt program execution is to *prefix* a "check-for-interrupt" clause to the start of the cycle (Fetch stage)
  - The source of an interrupt is an event that needs attention (asynchronously from the running program): the execution must move to it, which *is* preemption
  - An interrupt request found asserted at Fetch *hijacks* the CPU
- Omitting that check or not allowing interrupt requests to register is tantamount to inhibiting (aka *disabling*) preemption

Inhibiting preemption /1

- Some real-world procedures do *not* tolerate preemption
  - Either because they use devices that require strictly timed commanding or because they use *non-reentrant* code
- Non-reentrant code only uses global memory
  - It does *not* have per-call local variables (no stack)
  - Preemption might allow multiple calls to it to occur, whose overlap of execution would disrupt its execution state
- Reentrancy needs a call stack and mechanisms to assure atomic use of shared data
  - Affording stack memory is costly: each task needs its own
- At its simplest, disabling preemption requires ignoring pending interrupts

Inhibiting preemption /2

- A higher-priority job J_h that, at its release time, finds a lower-priority job J_l executing with disabled preemption, gets ***blocked*** for a time duration that depends on J_l
  - Under FPS, this constitutes a case of ***priority inversion***
- The feasibility of J_h now depends on J_l!
  - Under FPS, this form of blocking for J_i is upper-bounded by $\boldsymbol{B_i(np) = \max_{k=i+1,..,n}(\theta_k)}$ where $\theta_k \le e_k$ is the longest span of J_k's non-preemptible execution
  - This cost is paid by of J_i only *once* per release because lower-priority jobs *cannot* preempt J_i

The drag of self suspension /1

- Some devices may take some time to respond to commanding
  - The tasks that control them may be tempted to self-suspend between command and response
- Task τ_i whose jobs self suspend (`sleep()`) suffers a degenerate form of blocking that worsens its response time, and can be bounded as

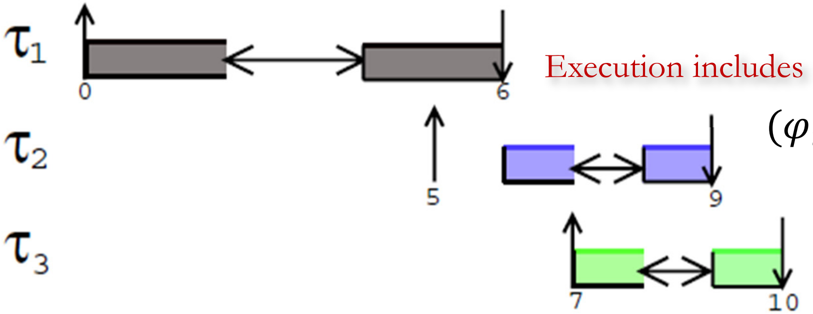
$$\boldsymbol{B_i(ss) = \max(\delta_i) + \sum_{k=1,..,i-1} \min(e_k, \max(\delta_k))}$$

  - $\max(\delta_i)$ is the longest duration of τ_i's self suspension
  - The $\sum$ term is the cumulative interference caused by self-suspending high-priority tasks that may become ready during the (shifted) busy period of τ_i
    - Every τ_k might resume from self-suspension exactly when τ_i does, and therefore interfere up to $max(\delta_k)$ but never more than e_k
- In general, a task τ_i that self suspends K times during execution incurs total blocking $\boldsymbol{B_i = B_i(ss) + (K+1)B_i(np)}$
  - As $B_i(np)$ is potentially incurred at at *every* resumption

The drag of self suspension /2

- Self suspension with independent tasks on single-core processors causes *scheduling anomalies*
  - Deadlines can be missed when task utilization or suspension delays are *decreased*

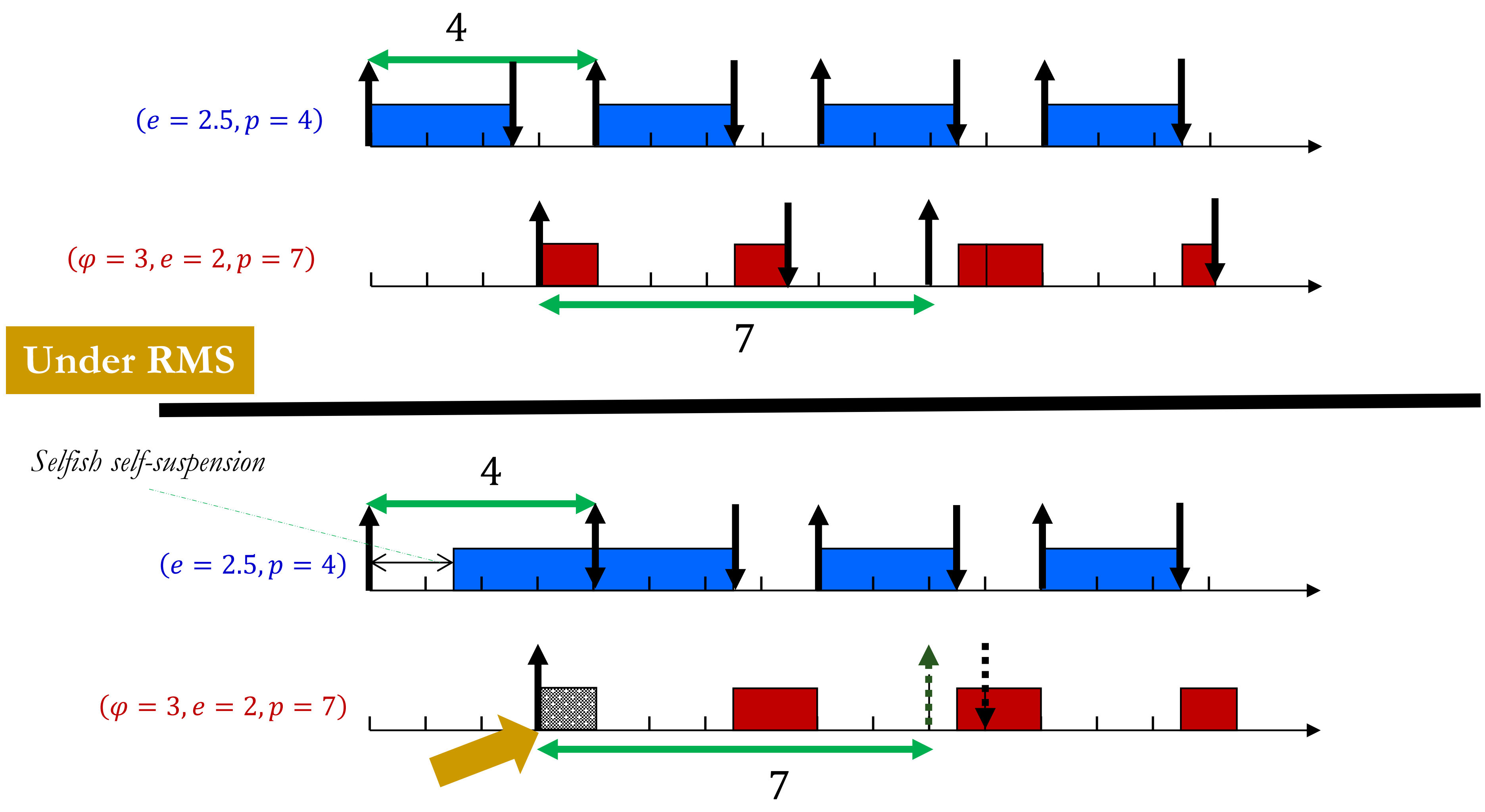
- **Example**: consider a feasible task set under EDF
  - $\tau_1 = \{0{,}10, (2{,}2{,}2), 6\}$
  - $\tau_2 = \{5{,}10, (1{,}1{,}1), 4\}$
  - $\tau_3 = \{7{,}10, (1{,}1{,}1), 3\}$
  - τ_3 would miss its deadline if τ_1's execution or suspension was 1 time unit shorter



Execution includes self suspension

$(\varphi_i, p_i, e_i, D_i)$

The drag of self suspension /3

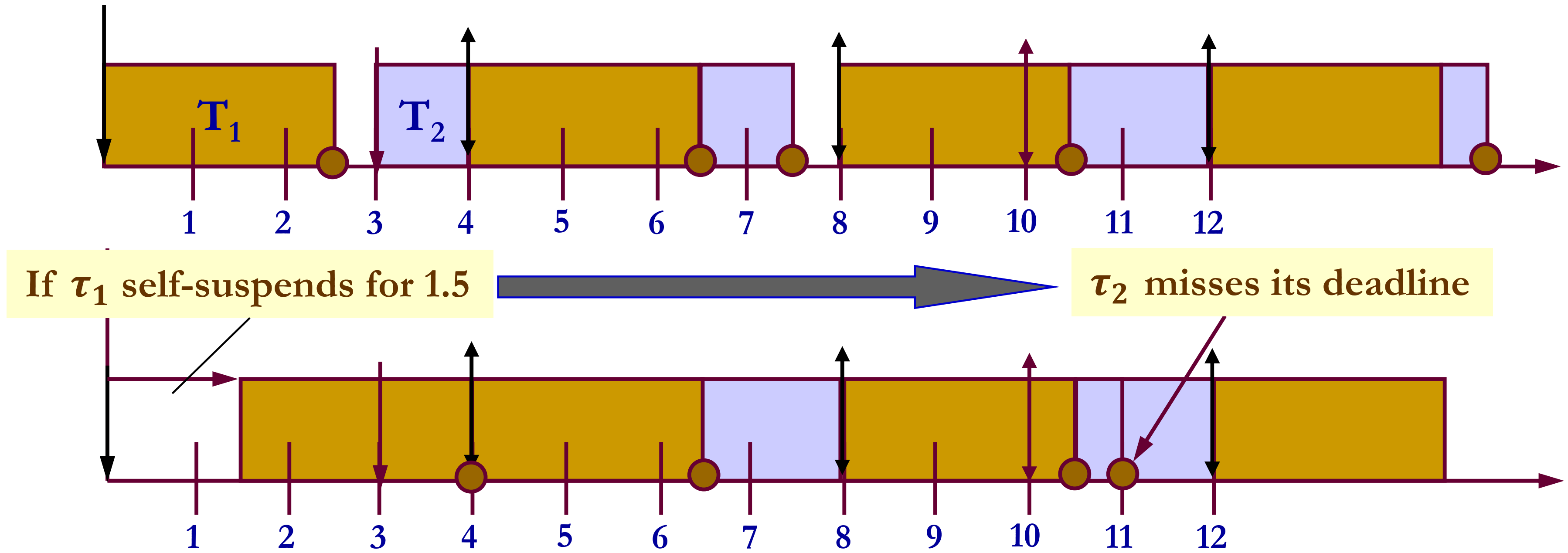


The drag of self suspension /4

$(\varphi_i, p_i, e_i, D_i)$

$\boldsymbol{\tau_1 = \{0, 4, 2.5, 4\}, \tau_2 = \{3, 10, 2, 10\}}$ $\boldsymbol{U = 0.875}$

Under RMS



τ_2's slack is: $\boldsymbol{\sigma_{2,1}(0) = D_{2,1} - \left\lceil \frac{D_{2,1}}{T_1} \right\rceil C_1 - C_2 = 10 - \left\lceil \frac{10}{4} \right\rceil 2.5 - 2 = 0.5}$

The blocking caused by τ_1's self-suspension on τ_2, is: $\boldsymbol{B_2(ss) = 0 + min(2.5, 1.5) = 1.5 > \sigma_{2,1}(0)}$

(This is a pessimistic upper bound: $\varphi_2 = 3$ reduces it to 1, but still $> \sigma_{2,1}(0)$)

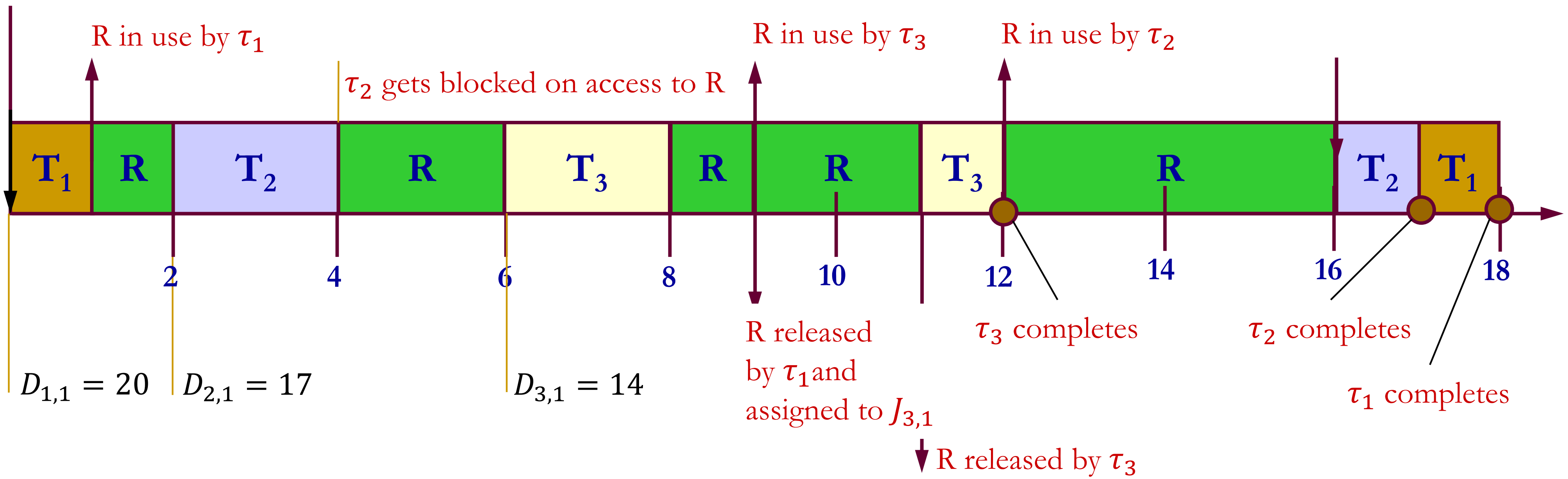
Effects of resource sharing /1

$(\varphi_i, p_i, e_i, D_i)$ — Max use of shared resource *R* per job

τ_1 = {-, -, 2, 20, R(4)}, τ_2 = {2, -, 3, 17, R(4)} , τ_3 = {6, -, 3, 14, R(2)}

under EDF (periods *not* specified: they do not matter here)

τ_1 :: e; R(4); e. τ_2 :: e; e; R(4); e. τ_3 :: e; e; R(2); e.



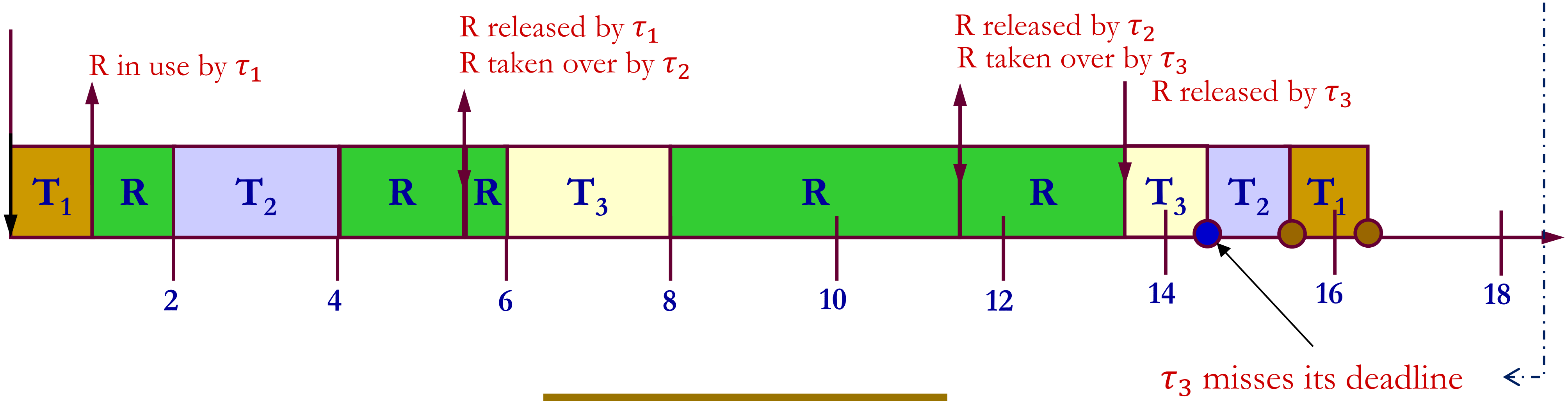
Effects of resource sharing /2

$(\varphi_i, p_i, e_i, D_i)$

τ_1 = {-, -, 2, 20, R(2.5)}, τ_2 = {2, -, 3, 17, R(4)} , τ_3 = {6, -, 3, 14, R(2)}

under EDF

Same as before, except with *shorter* use of R by τ_1



Scheduling anomaly!

Access contention /1

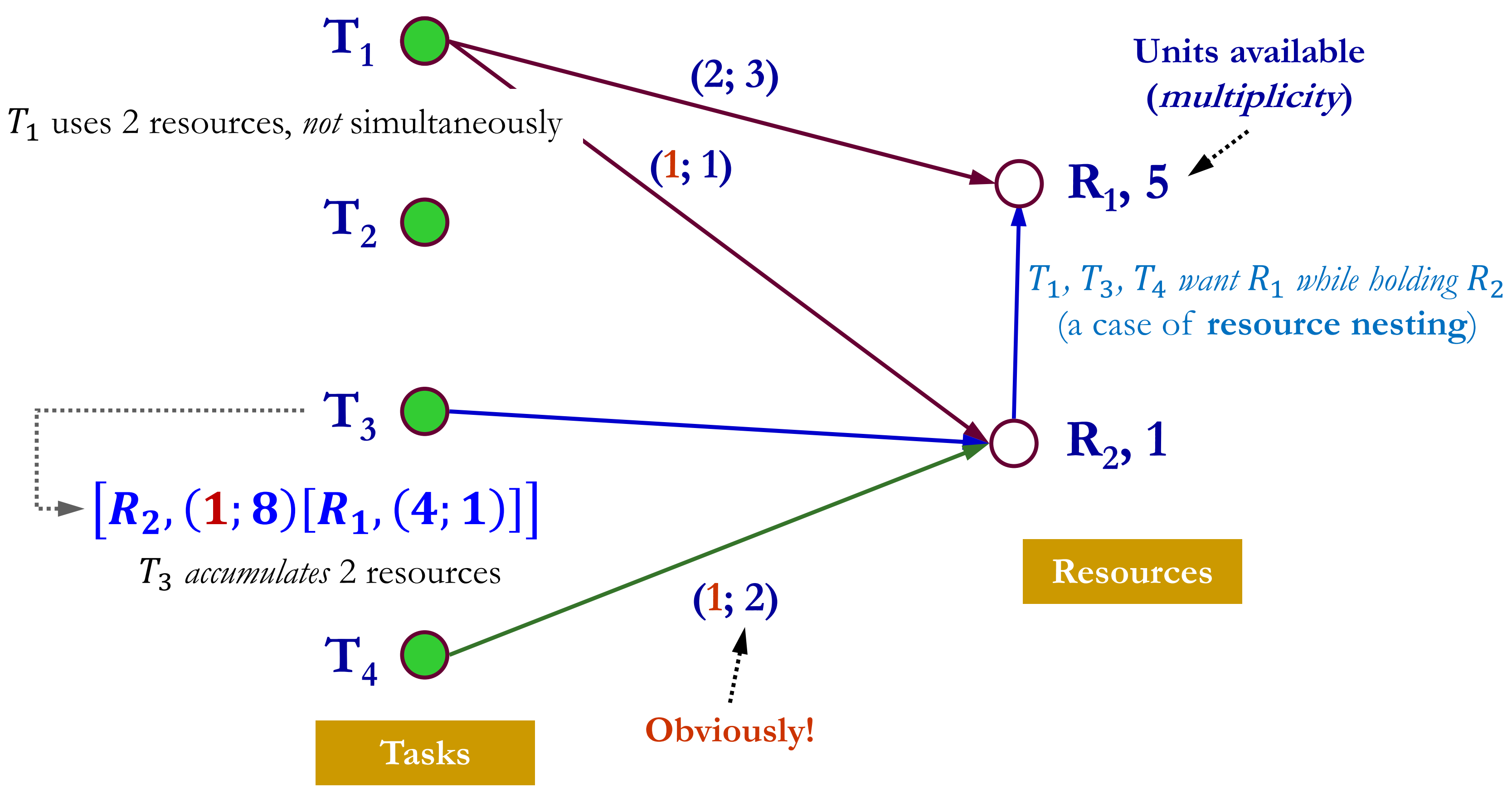
- Concurrent access to shared resources causes potential for contention that needs specialized control
  - A ***resource access control protocol***
- Such a protocol specifies (1) when, (2) on what conditions, (3) in which order, a resource access request may be granted
  - Access contention situations may cause *priority inversion* to arise (see following examples)

Access contention /2

- In order to help contain response time
  - Jobs should *not* self suspend (directly or indirectly)
  - Jobs can be *preempted*
  - Priority inversion situations should be minimized
- We say that job J_h is ***directly blocked*** by a lower-priority job J_l when
  - J_l is granted exclusive access to a shared resource R
  - J_h has requested R and its request has *not* been granted
- To study the problem we may want to use a ***wait-for graph***

Wait-for graph

(i; j) denotes i units of resource required, for j units of time



Resource access control [**option a**]

- ***Inhibiting preemption*** in critical sections
  - A job that requires access to a resource is *always* granted it
  - A job that has been assigned a resource runs at a priority higher than any other job
    - These two clauses imply each other (why?)
    - They jointly prevent deadlock situations from occurring (why?)
- This protocol causes ***bounded*** priority inversion
  - At most *once* per job (we already know why)
  - For a maximum duration of $\boldsymbol{B_i(rc) = max_{k=i+1,\ldots,n}(C_k)}$
    - For job indices in monotonically non-increasing order and C_k denoting the worst-case duration of critical section for job J_k

Critique of [**option a**]

- This strategy causes ***distributed overhead***
  - *All* jobs – including those that do *not* compete for resource access – incur some time penalty
  - Very unfair: undesirable
- It should be preferable that time overhead be *solely* (or at least mostly) incurred by the jobs that *do* compete for resource access
  - The priority of the job that is granted the resource should be *no less* than that of its *competitor* jobs (but of no other)
    - This principle has two possible realizations
    - One is called ***priority inheritance***, the other is called ***priority ceiling***
    - We shall now examine how each of them operates

Resource access control [**option b**]

- ***Basic priority inheritance protocol*** (BPIP)
  - The job's priority may vary over time
  - The variation follows inheritance principles
- **Protocol rules**
  - Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time, they assume their *assigned priority*
  - Allocation: when job J requires access to resource R at time t
    - If R is free, R is assigned to J until release
    - If R is busy, the request is denied and J becomes *blocked*
  - Priority inheritance: when job J becomes blocked, job J_l that blocks it takes on J's current priority as its *inherited priority* and retains it until R is released; at that point J_l reverts to its previous priority

Critique of [**option b**]

- BPIP suffers two forms of blocking
  - ***Direct blocking***, owing to resource contention
  - ***Inheritance blocking***, owing to priority raising
- Priority inheritance is *transitive*
  - Direct blocking *is* transitive as jobs may need to accumulate resources
- BPIP does *not* prevent deadlock
  - Cyclic blocking proceeds from transitive direct blocking
- BPIP incurs *reducible* distributed overhead
  - Under BPIP, a job may become blocked every time it competes for a shared resource, hence multiple times in the same run
- BPIP needs *no* prior knowledge on which resources are shared
  - It is inherently dynamic, hence usable for open (non real-time) systems

Resource access control [**option c**]

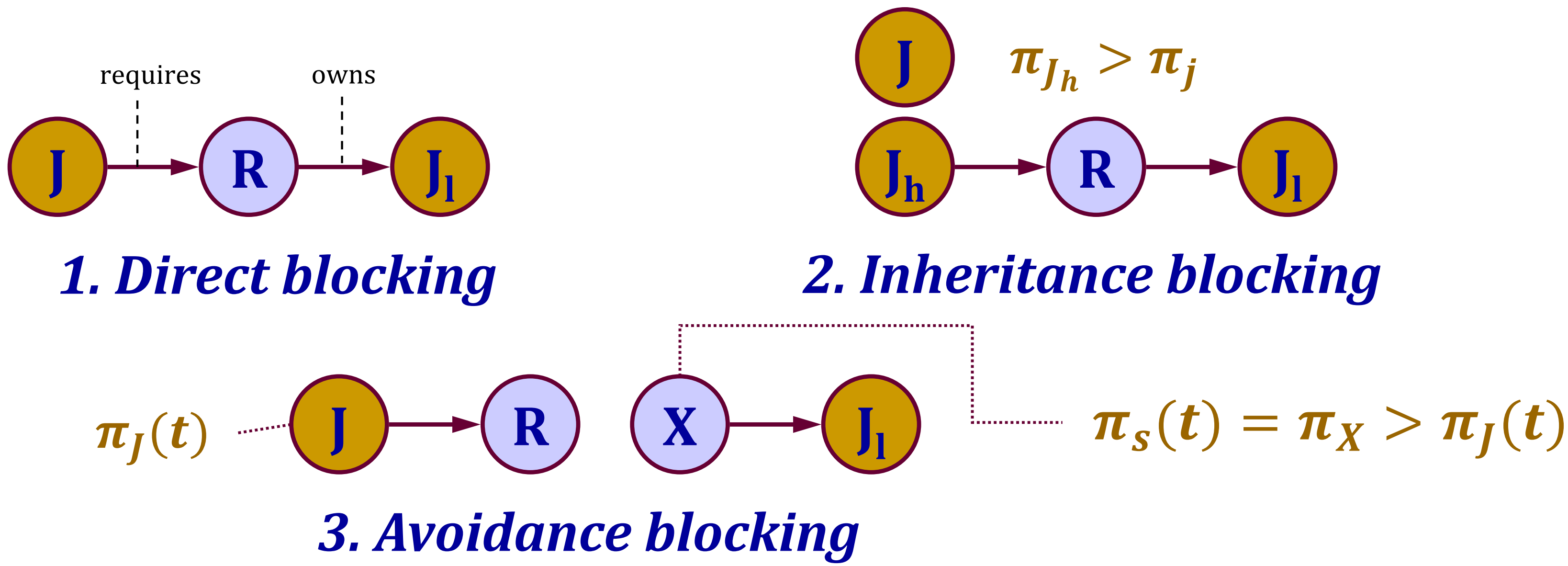
- ***Basic priority ceiling protocol*** (BPCP)
  - Similar to BPIP, except that it needs *all* resource requirements to be *statically known*
  - Every resource R is assigned a *priority ceiling attribute* set statically to the highest priority of the jobs that require R
    - At time t, the system has a ceiling $\pi_s(t)$ attribute set to the highest priority ceiling of all resources currently in use
    - If no resource is currently in use at t, $\pi_s(t)$ defaults to $\Omega <$ the lowest priority of all jobs

BPCP protocol rules

- Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time they assume their assigned priority
- Allocation: when job J requests access to resource R at time t
  - If R is already assigned, the request is denied and J becomes blocked
  - If R is free and J's priority $\pi_J(t) > \pi_s(t)$, the request is granted
  - If J currently owns the resource whose priority ceiling $= \pi_s(t)$, the request is granted
  - Otherwise the request is denied and J becomes blocked — Avoidance blocking
- Priority inheritance: when job J becomes blocked by job J_l, J_l takes on J's current priority $\pi_J(t)$ until J_l releases all resources with priority ceiling $> \pi_J(t)$; at that point J_l's priority reverts to the level that preceded access to those resources

Critique of [**option c**] /1

- BPCP is *not* greedy (BPIP is!)
  - Under BPCP, a request for a free resource may be denied
- Hence, BPCP causes each job J to incur **three** distinct forms of blocking caused by lower-priority job J_l



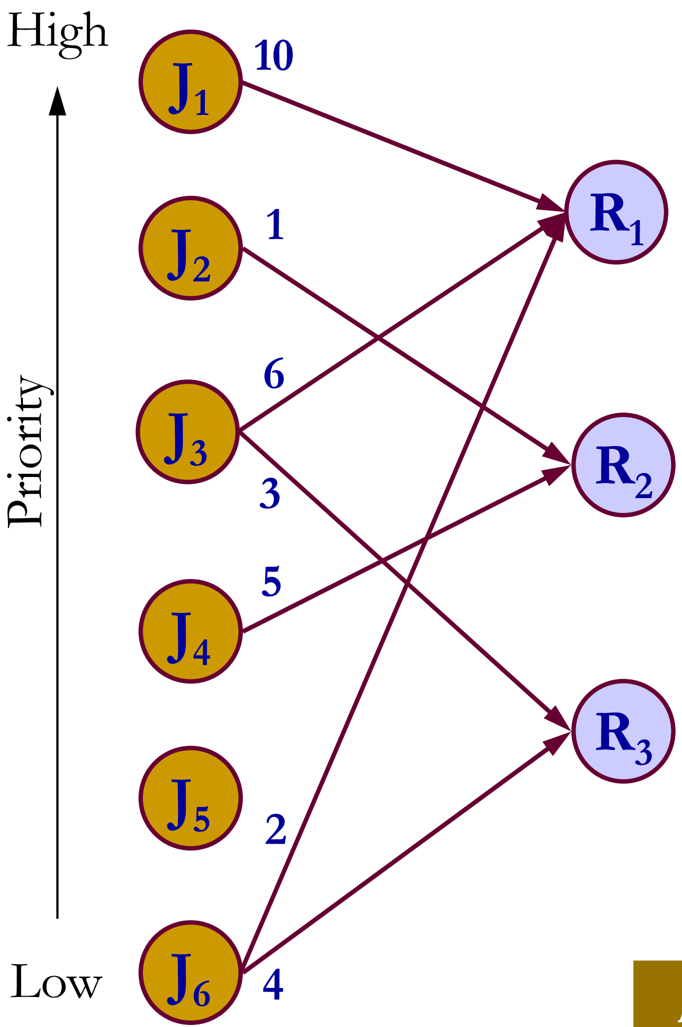
Critique of [**option c**] /2

- ***Avoidance blocking*** is what makes BPCP not greedy and also prevents deadlock from occurring
  - If, at time t, job J has $\pi_J(t) > \pi_s(t)$ then it must be so that
    - J will never use any of the resources in use at time t
    - So won't all jobs with higher priority than J
- The system ceiling $\pi_s(t)$ determines which jobs can be assigned a resource free at time t without risking deadlock
  - All jobs with priority higher than the system ceiling $\pi_s(t)$
- **Caveat**
  - To stop job J from blocking itself when attempting to accumulate resources, BPCP must grant its request in case $\pi_J(t) \leq \pi_s(t)$, but J at t holds the resources $\{X\}$ whose priority ceiling is $= \pi_s\ (t)$

Critique of [**option c**] /3

- BPCP does *not* incur reducible distributed overhead as it does *not* permit transitive blocking
- **Theorem** [Sha & Rajkumar & Lehoczky, 1990]
  Under BPCP a job may become blocked for *at most* the duration of *one* critical section
  - Under BPCP, when a job becomes blocked, its blocking can *only* be caused by a single ready job
  - The job that causes others to block cannot itself be blocked
    - Hence BPCP does not permit transitive blocking
  - Demonstration: **By exercise**
- The maximum possible value of that duration for job J_i is termed the *blocking time* $B_i(rc)$ due to resource contention
  - $B_i(rc)$ must be accounted for in the schedulability test for J_i

Computing the BPCP blocking time /1



Directly blocked by

	J2	J3	J4	J5	J6
J1		6			2
J2			5		
J3					4
J4					
J5					

Priority-inheritance blocked by

	J2	J3	J4	J5	J6
J1					
J2		6			2
J3			5		2
J4					4
J5					4

Avoidance blocked by

	J2	J3	J4	J5	J6
J1					
J2		6			2
J3			5		2
J4					4
J5					

$B_i(rc)$ = max value in row J_i across all tables

Computing the BPCP blocking time /2

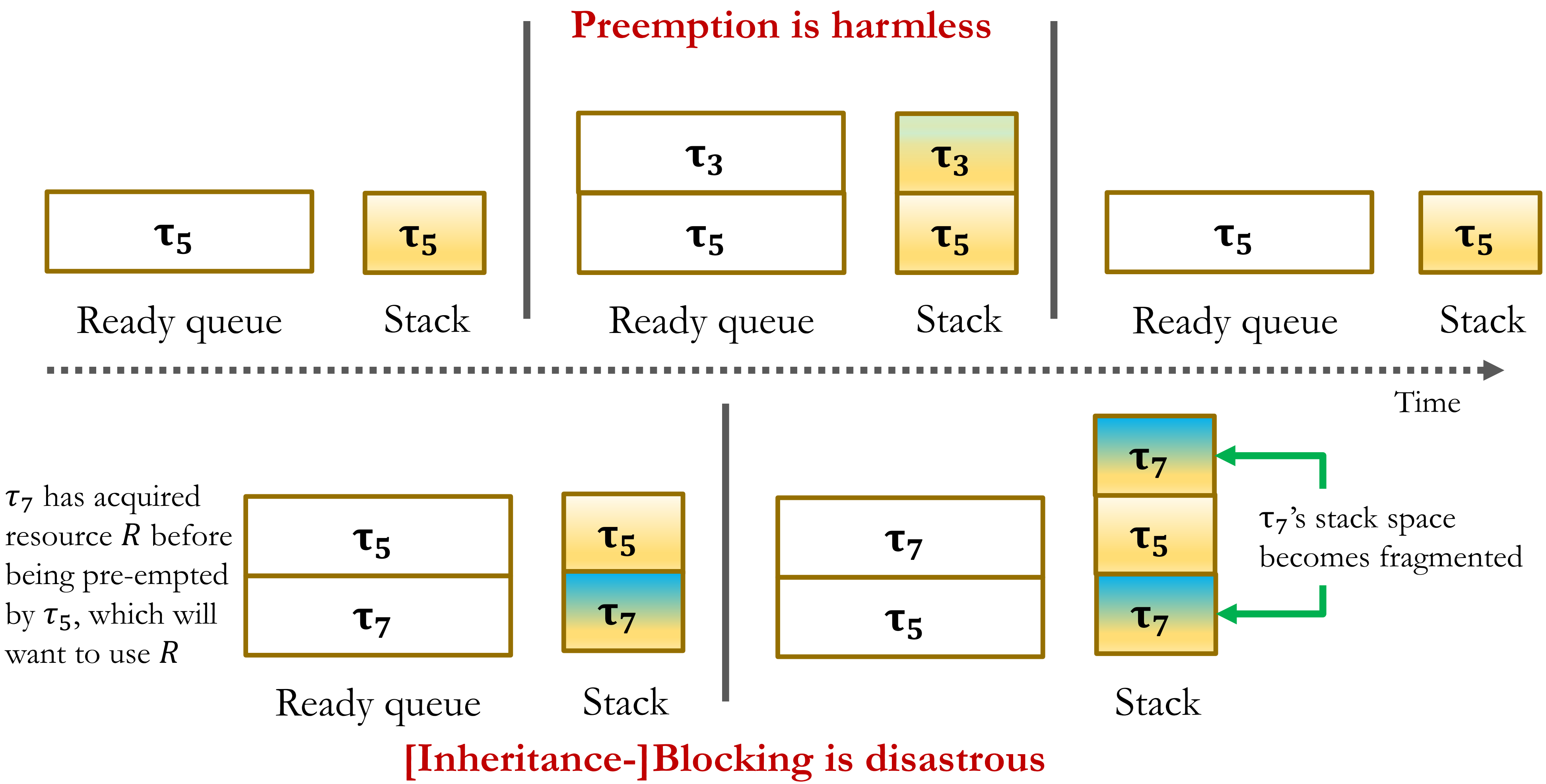
- Table rows are sorted by priority
  - Jobs are assigned distinct priorities (i.e., no overlap)
- Table "*directly blocked by*" is easy to understand …
- Table "*priority-inheritance blocked by*"
  - Job J_{i+1} causes direct blocking inherits the blocked job's priority: all jobs with priority lower than the inherited one but higher than J_{i+1}'s suffer blocking
  - The value in cell $[i, k]$ is max across (rows $1, \dots, i-1$; column k) in Table "*directly blocked by*"
- Table "*avoidance blocked by*"
  - The resource is free but another resource with priority ceiling higher than your current priority is being used by a job with assigned priority lower than yours
  - The cells here are as in Table "*priority-inheritance blocked by*" except for the jobs that do *not* request resources (e.g., J_5), which are exempt from this blocking

Resource access control [**option d**]

- ***Stack-based ceiling priority protocol***
  - SB-CPP uses the system ceiling same as BPCP, but it allows jobs to *share* stack space, saving precious memory
    - Try and see why!
    - The other protocols seen so far don't
  - SB-CPP does it by ensuring that *no* request for resources will *ever* be denied to a running job

    - This prevents jobs' stack space from fragmenting
  - Blocking causes stack fragmentation
    - Preemption does not!
    - One more reason to discourage self-suspension …

What blocking and preemption do to the stack



SB-CPP protocol rules [Baker, 1991]

- Computation of and updates to ceiling $\pi_s(t)$:
  - When all resources are free, $\pi_s(t) = \Omega$
  - $\pi_s(t)$ is updated any time t a resource is assigned or released
- Scheduling: on release at time t, job J stays *blocked* until its *assigned priority* $\pi_J(t) > \pi_s(t)$
  - Jobs that are not blocked are dispatched to execution by preemptive priority-driven scheduling
- Allocation: whenever a job issues a request for a resource, the request is granted

Critique of [**option d**]

- Under SB-CPP, a job J can only begin execution when the resources it may need are free
  - Otherwise $\pi_J(t) > \pi_s(t)$ cannot hold
- Under SB-CPP, a job J that may get preempted does *not* become blocked on resumption
  - The preempting job *cannot* contend resources with J
- SB-CPP prevents deadlock from occurring
- Under SB-CPP, $B_i(rc)$ for any job J_i is the same as BPCP's
- SB-CPP has lower algorithmic complexity in time and space than BPCP, as it needs *less* checks against $\pi_s(t)$

Resource access control [**option e**]

- ***Ceiling priority protocol***
  - CPP does *not* use the system ceiling $\pi_s(t)$
  - Resources continue to have a ceiling priority attribute
- Scheduling: jobs are scheduled with FPS with "*FIFO within priorities*" ruling
  - A job that does not hold any resource, runs with its *assigned priority*
  - A job that acquires a resource has its *current priority* set to the highest value among the ceiling priority of the resources that it holds
- Allocation: whenever a job issues a request for a resource, the request is granted 

Summary

- Issues arising from task contention of shared resources under preemptive priority-based scheduling
- Survey of resource access control protocols
- Critique of the surveyed protocols

Selected readings

- L. Sha, R. Rajkumar, J.P. Lehoczky (**1990**)
  Priority inheritance protocols: an approach to real-time synchronization
  DOI: 10.1109/12.57058
- T. Baker (**1990**)
  A Stack-Based Resource Allocation Policy for Real-time Processes
  DOI: 10.1109/REAL.1990.128747