CS 423
Operating System Design: Scheduling Periodic Tasks In Embedded Systems II

Professor Adam Bates

Spring 2017

Goals for Today

- Learning Objective:
  - Conclude discussion of real-time scheduling for embedded systems
- Announcements, etc:
  - MP1 is is out! Due Feb 20
  - Midterm Exam — Wednesday March 6th (in-class)
  - Updates to C4 reading lists; should be locked-in for the rest of the semester now.

Reminder: Please put away devices at the start of class

Feedback for C4 Students

- Detailed feedback on your Heisenbug summary went out today. This feedback sets the bar for future summary grades.

Feedback for C4 Students

- Area: Briefly, what is the general topic of this work?
- Problem: Whe is the problem that in the area that this work is considering?
- Solution: What is the HIGH-LEVEL idea that the authors have for addressing this problem? Note that an Implementation is not an idea
- Methodology: What are the central components of the system that define the author's design? Difficult to summarize succinctly; requires tough editing decisions on your part about what is important.
- Results: What are the quantitative (usually) findings described in the authors evaluative tests? (Also, implementation details if relevant)
- Takeaway: What is the broad lesson that this paper is communicating?

Feedback for C4 Students

An exemplar 5/5 summary (compiled from student submissions):

- Area: *Software Testing, Concurrent Programming, Operating Systems*
- Problem: *Building concurrent programs is a hard problem. Subtle interactions between threads may result in different program executions, and thus may generate bugs of different forms. Some bug rarely appears, i.e. once after four month full-loaded testing, and hard to reproduce and eliminate. It is important to design an efficient debugging tool for concurrent programs.*
- Solution: Create a software testing tool that monitors and controls the scheduling of threads and asynchronous events for the concurrent program under test. This would allow heisenbugs to be deterministically identified by enumerating all possible interleavings of execution scenarios of the program.
- Methodology: The authors introduce CHESS, a system that takes control of scheduling of thread and asynchronous events, so it can capture all the interleaving nondeterminism in the program. The control is obtained by CHESS scheduler. First, To deal with the input nondeterminism, CHESS uses some cleanup functions, which can make sure that every program runs from the same start state. In order to simplify the implementation of the scheduler, Lamport's happens-before graph is used to provide a execution order of threads in a concurrent execution…

Feedback for C4 Students

An exemplar 5/5 summary (compiled from student submissions):

- Methodology (continued): … CHESS uses wrappers for concurrent API calls with the synchronization operations. When the current programs contain data-races, CHESS scheduler enforces single-threaded execution by allowing only one thread access to a memory location at single time. To drive the test along different schedules, CHESS repeatedly executes in iteration, which has reply, record, and search phases. In the reply phase, the scheduler replays a sequence of sequence of scheduling choices from a trace file. Then, in the record phase, the scheduler schedules a thread until the thread yields the processor. In the search phase, the scheduler picks the next schedule based on the information it has. However, there are several challenges in the three phases, such as imperfect reply. Also, the scheduler is built for well-documented and standard APIs, so it can be reused in different programs.

- Results: CHESS has been tested in several large industry-scale programs such as PLINQ, CDS, and STM. Chess found 27 bugs in all the programs tested, 25 of which were previously unknown. Also Chess was able to reproduce all the failures that traditional stress tests reported.

- Takeaway: By carefully controlling the concurrency primitives, software tools can automate the process of reproducing Heisenbugs. This paper proves the hypothesis that errors in complex systems more likely occurs because of the complex interleaving of simple scenarios.

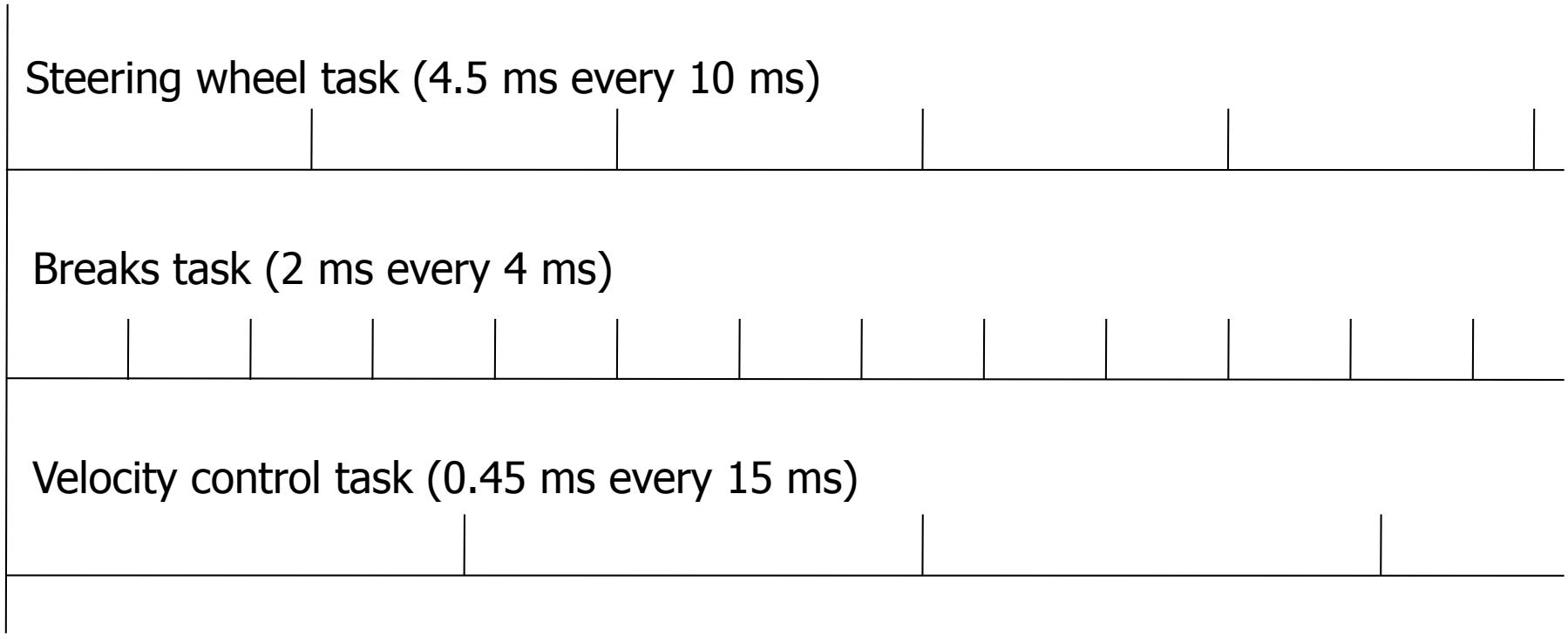
Feedback for C4 Students

Other Notes

- No figures; you need to describe results and design components in your own words.
- Don't plagiarize the paper?
  - You will not receive points for copy and pasting content from the paper into your summary
  - If you take content without acknowledging the source it will be considered academic dishonest
- Good summaries are about reflecting on the paper, deciding what are the most important ideas, and describing those ideas succinctly.
  - Great practice for writing paper abstracts, elevator pitches, grant proposals, etc.,…

Task Scheduling

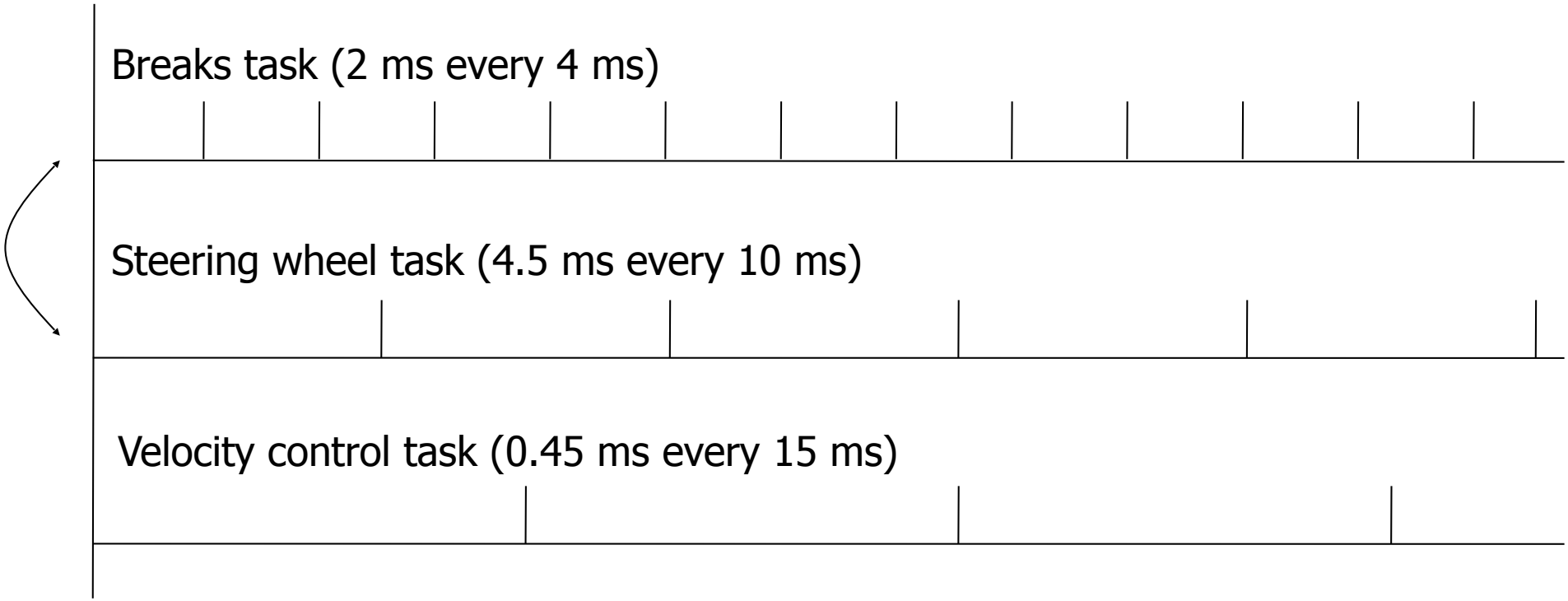
How do we assign task priorities? (SCHED_FIFO)



Task Scheduling

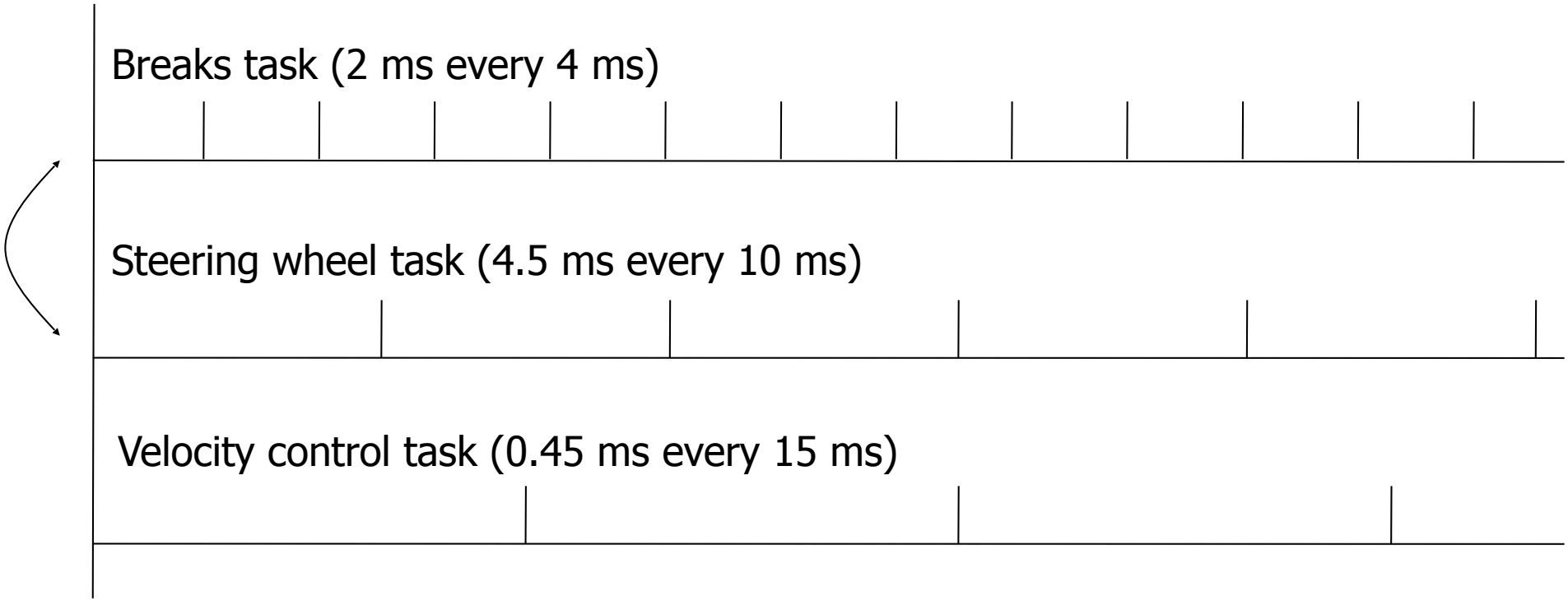
How do we assign task priorities? (SCHED_FIFO)

- ***Rate Monotonic*** (large rate = higher priority)



Intuition: Urgent tasks should be higher in priority

Task Scheduling

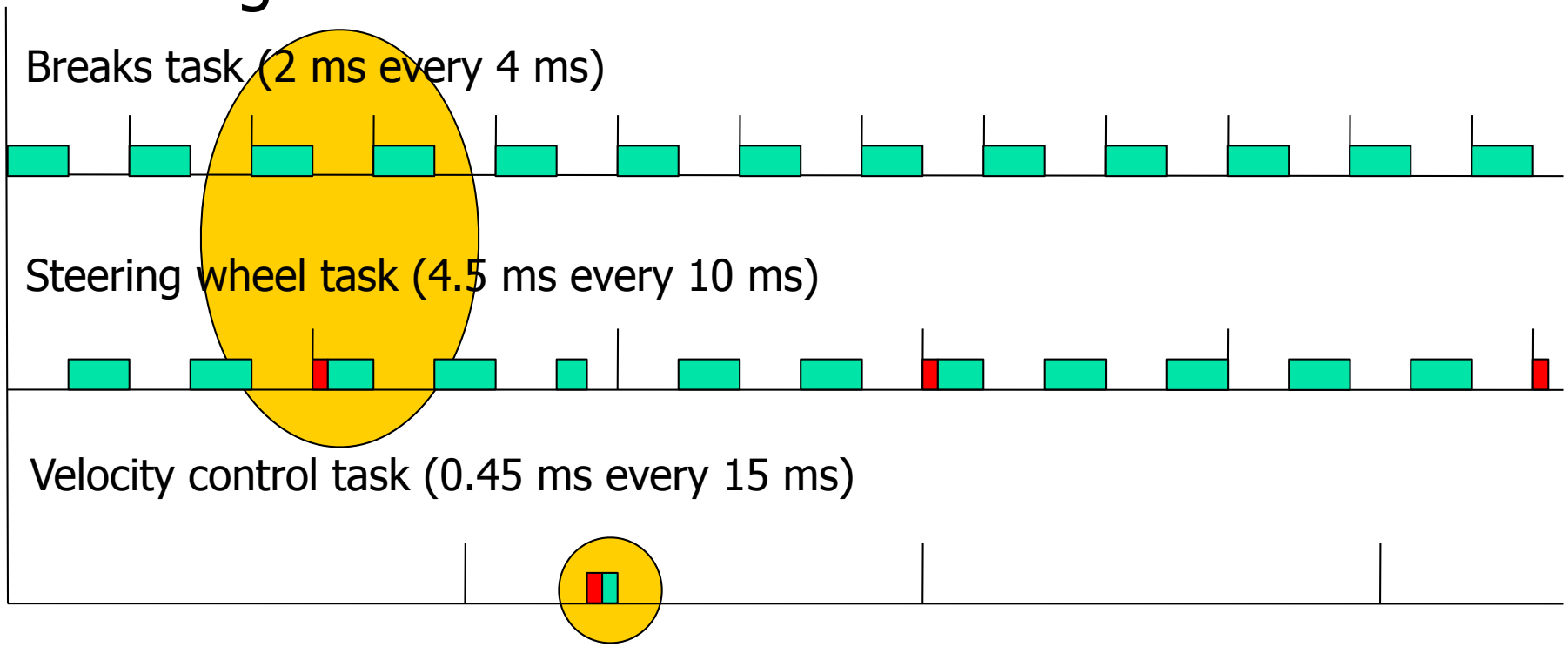


Intuition: Urgent tasks should be higher in priority

Is there a problem here??

Task Scheduling

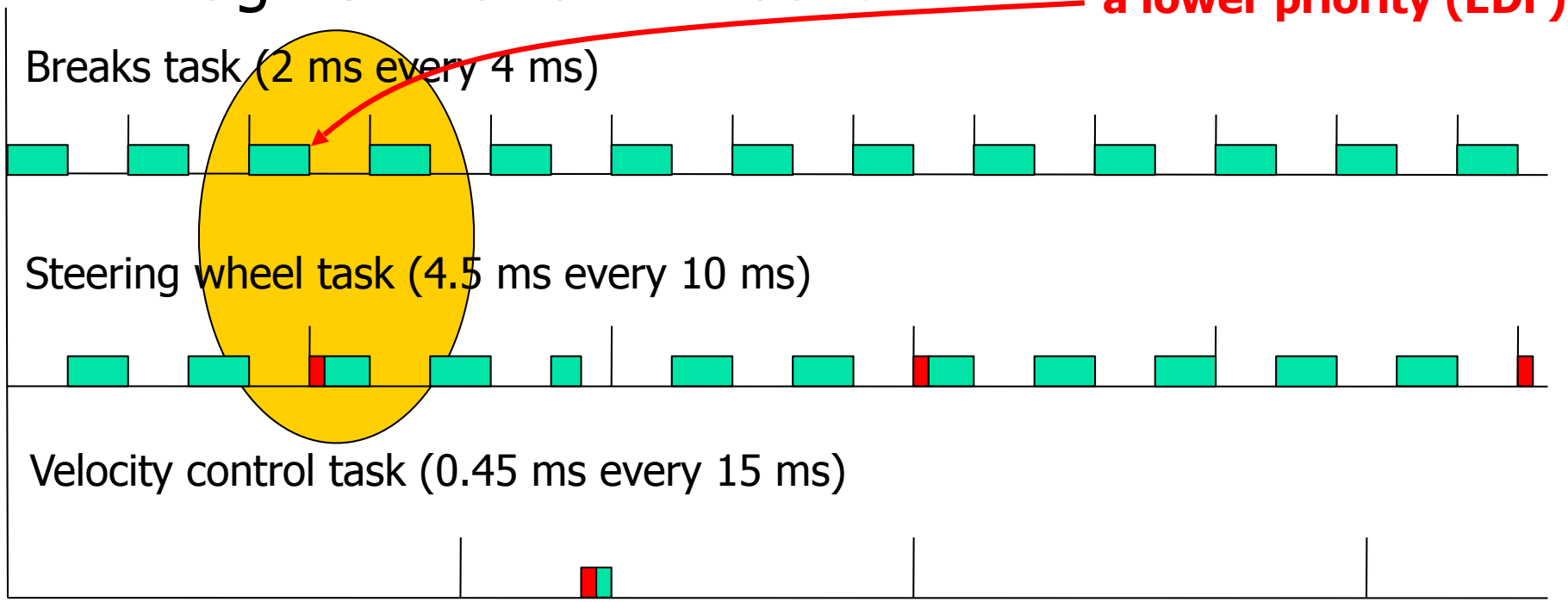
- Deadlines are missed!
- Average Utilization < 100%



Task Scheduling

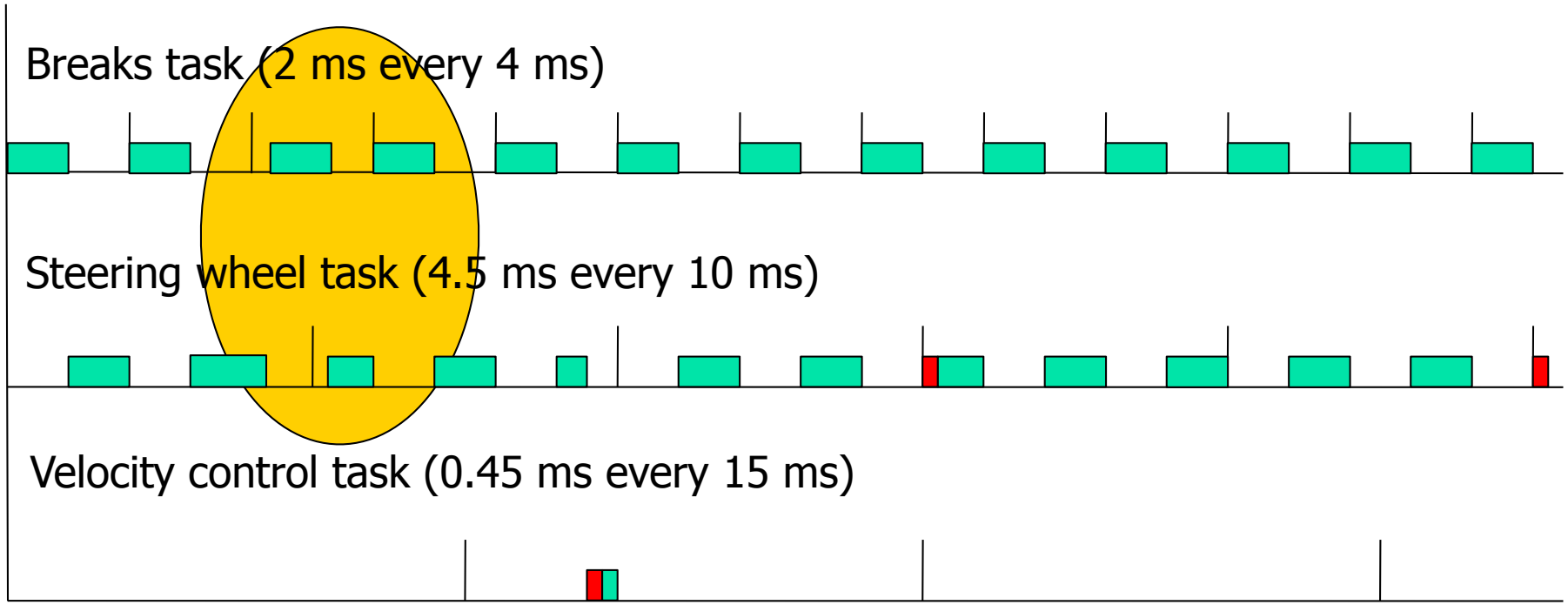
- Deadlines are missed!
- Average Utilization < 100%

Fix:
Give this task invocation a lower priority (EDF)



Task Scheduling

- Static versus Dynamic priorities?
  - Static: Instances of the same task have the same priority
  - Dynamic: Instances of same task may have different priorities



Intuition: Dynamic priorities offer the designer more flexibility and hence are more capable to meet deadlines

Task Scheduling

Re: Real Time Scheduling of Periodic Tasks…

- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)
- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent periodic tasks (meets the most deadlines of all static priority scheduling policies)

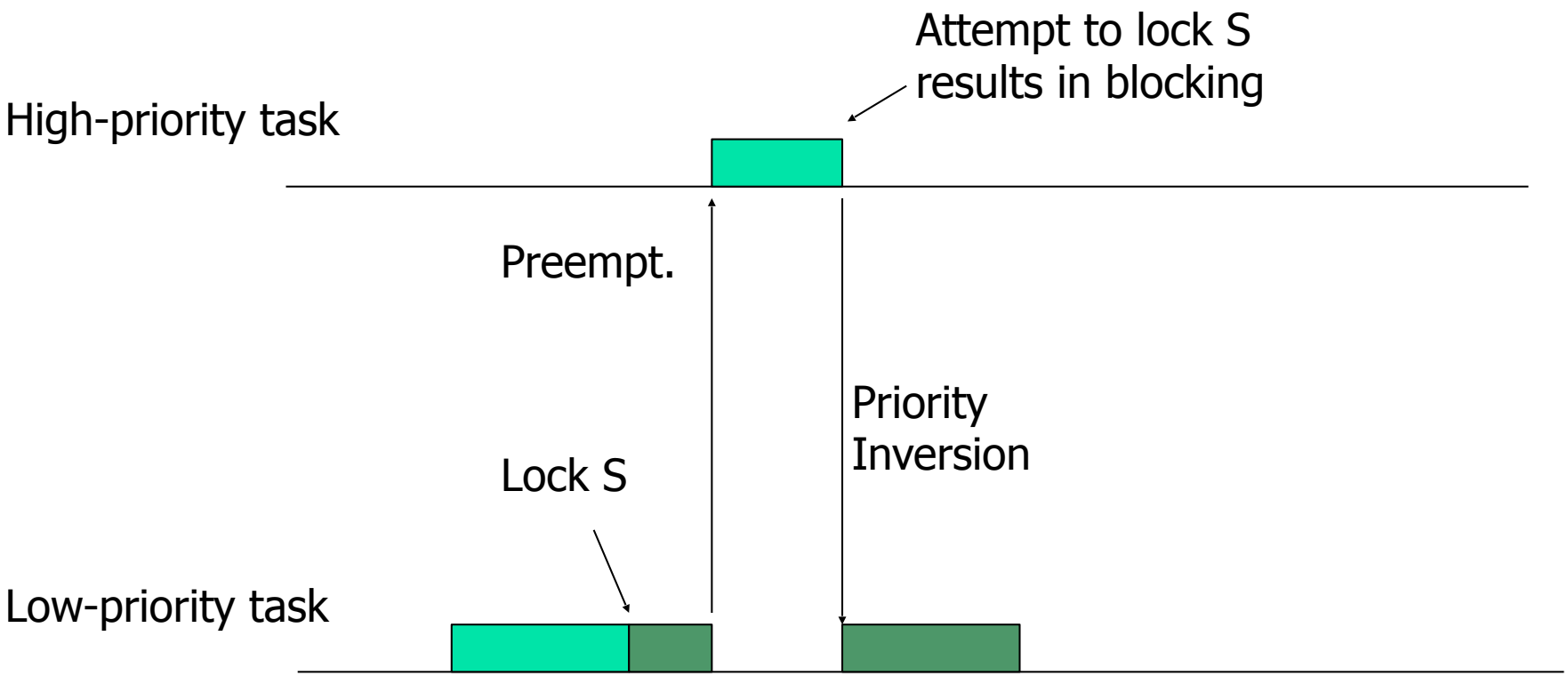
Task Scheduling

Re: Real Time Scheduling of Periodic Tasks…

- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for <u>independent</u> periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)
- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for <u>independent</u> periodic tasks (meets the most deadlines of all static priority scheduling policies)

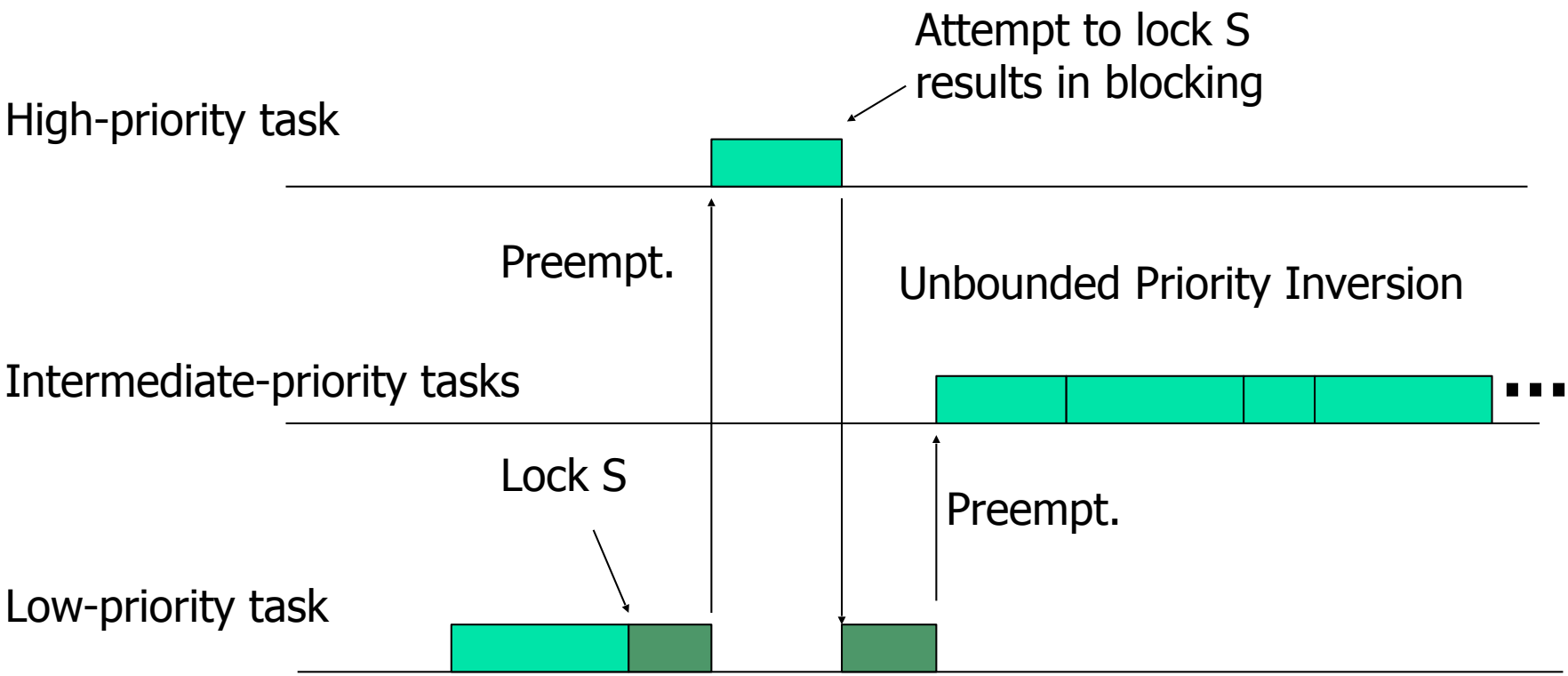
Priority Inversion

Locks and priorities may be at odds. Locking results in priority inversion



Unbounded Priority Inversion

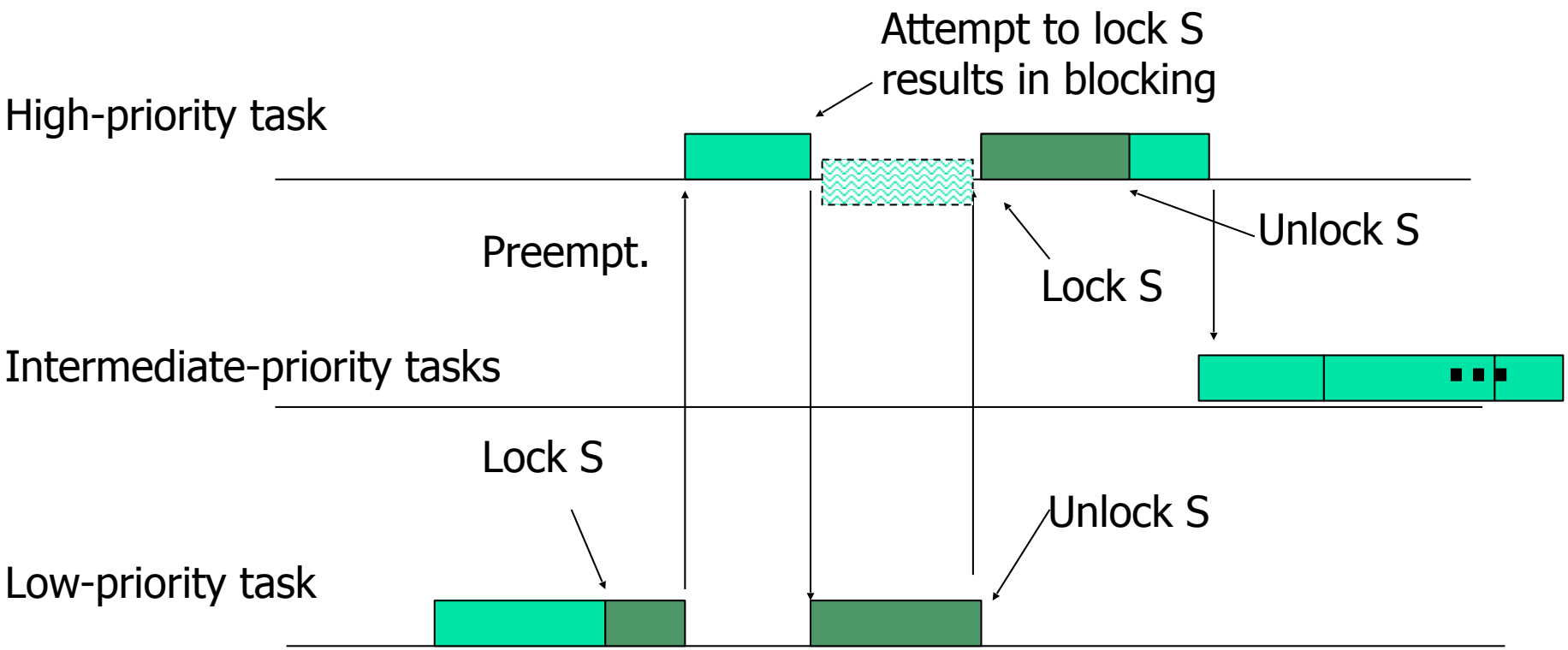
Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one



How can we prevent unbounded priority inversion?

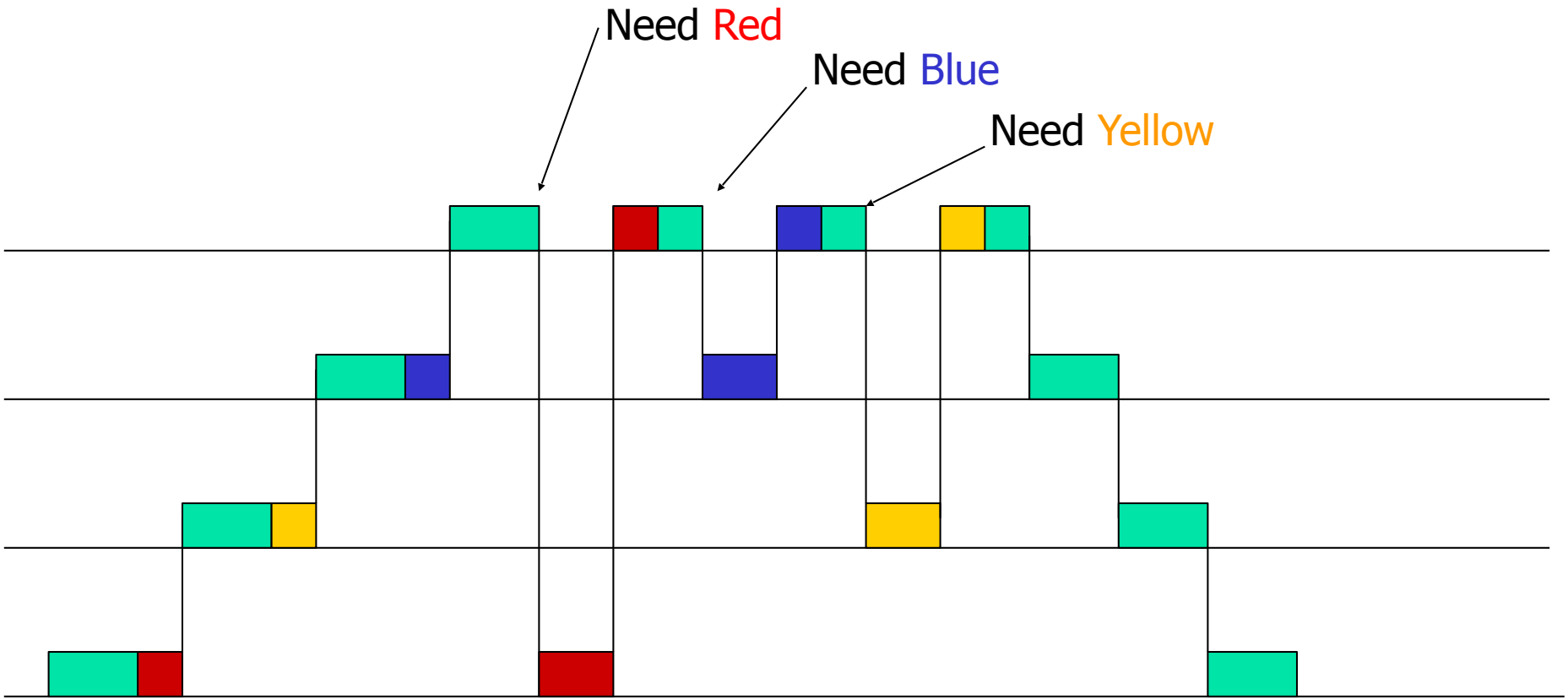
Priority Inheritance Protocol

Solution: Let a task inherit the priority of any higher-priority task it is blocking



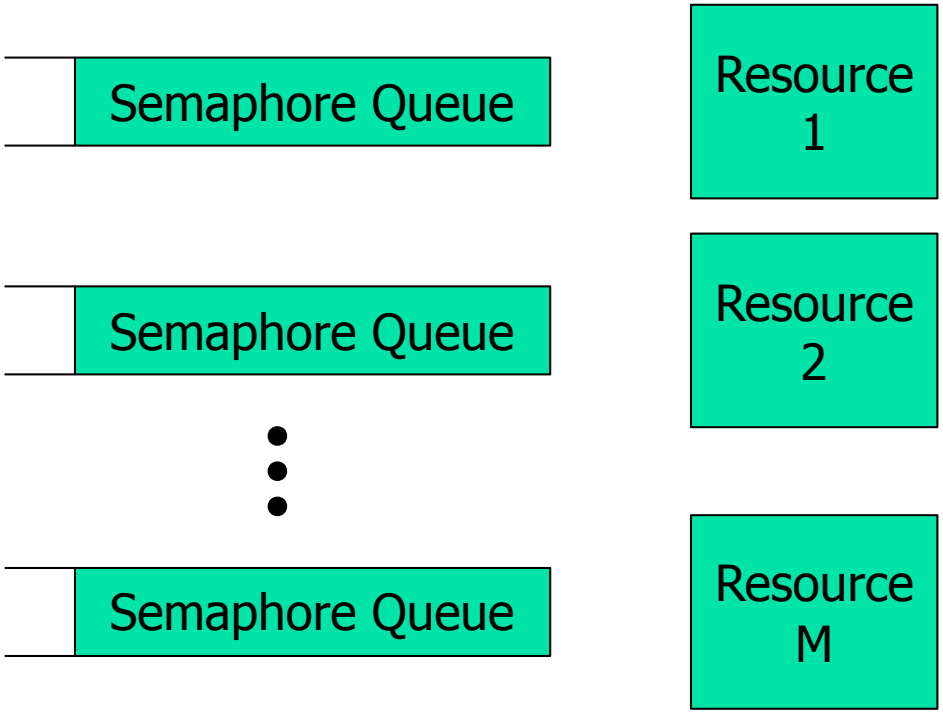
Priority Inheritance Protocol

... but what is the maximum blocking time (for high priority tasks)?



Max Priority Inversion Time

- What is the longest period of time a high priority task will wait on a resource?



Two priority inversion scenarios to consider:

(a) Lower priority task holds a resource I need (direct blocking)

(b) Lower priority task inherits a higher priority than me because it holds a resource the higher-priority task needs (push-through blocking)

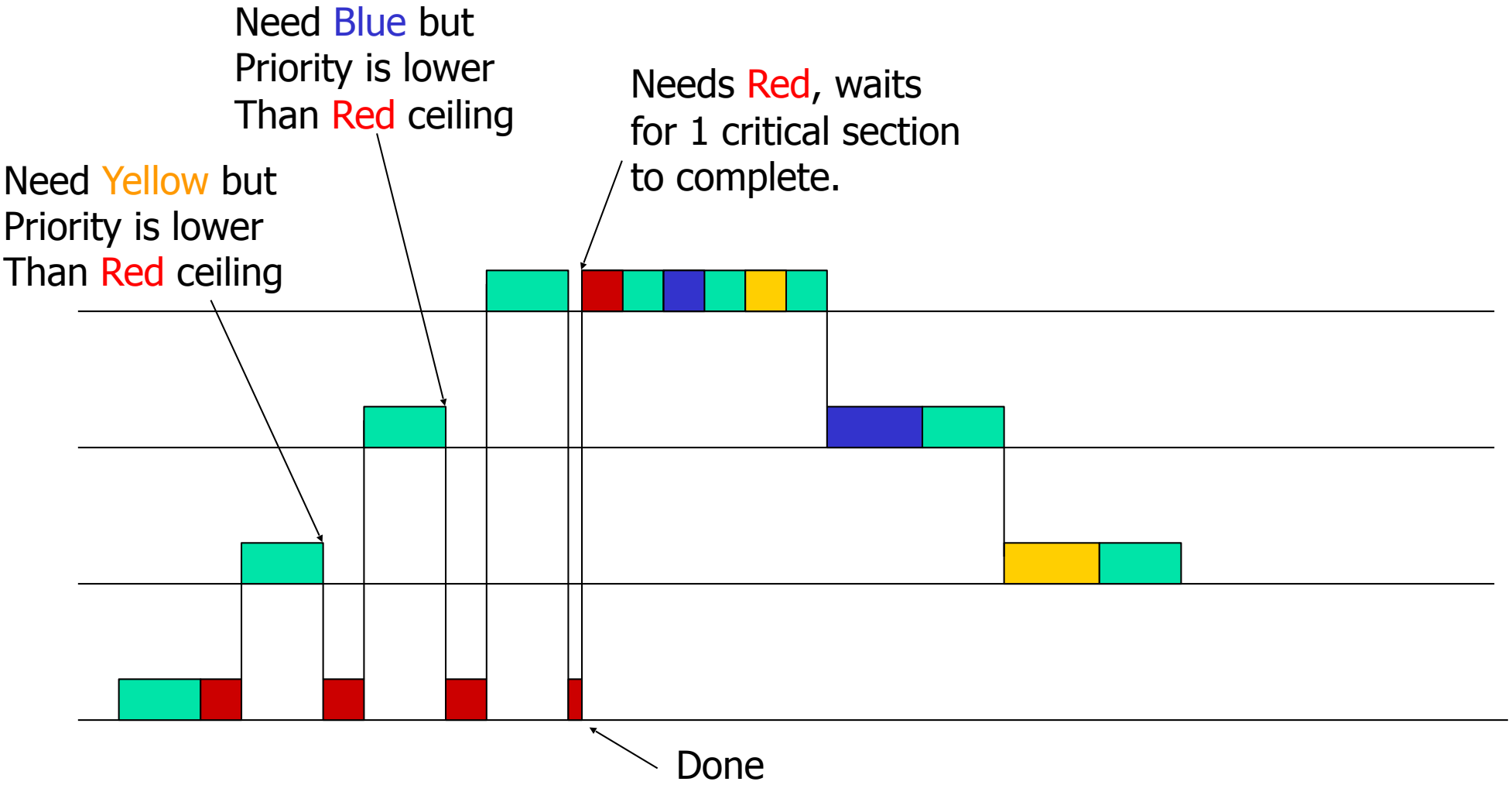
Priority Ceiling Protocol*

- The Priority Ceiling (PC) of Semaphore s:
  - PC(s) = highest priority of all processes that may lock s
- A process P is allowed to start a new critical section if and only if:
  - For all s currently locked, Priority(P) > PC(s)
- If P is suspended such that Process Q is blocking P:
  - Q inherits P's Priority if Priority(Q) < Priority(P)
  - Note: This step is the Priority Inheritance Protocol
- Property: A process can be blocked by a lower-priority process for at most the duration of One (1) critical section.
- Assumptions?

* Same definition as last class, just a little easier to parse

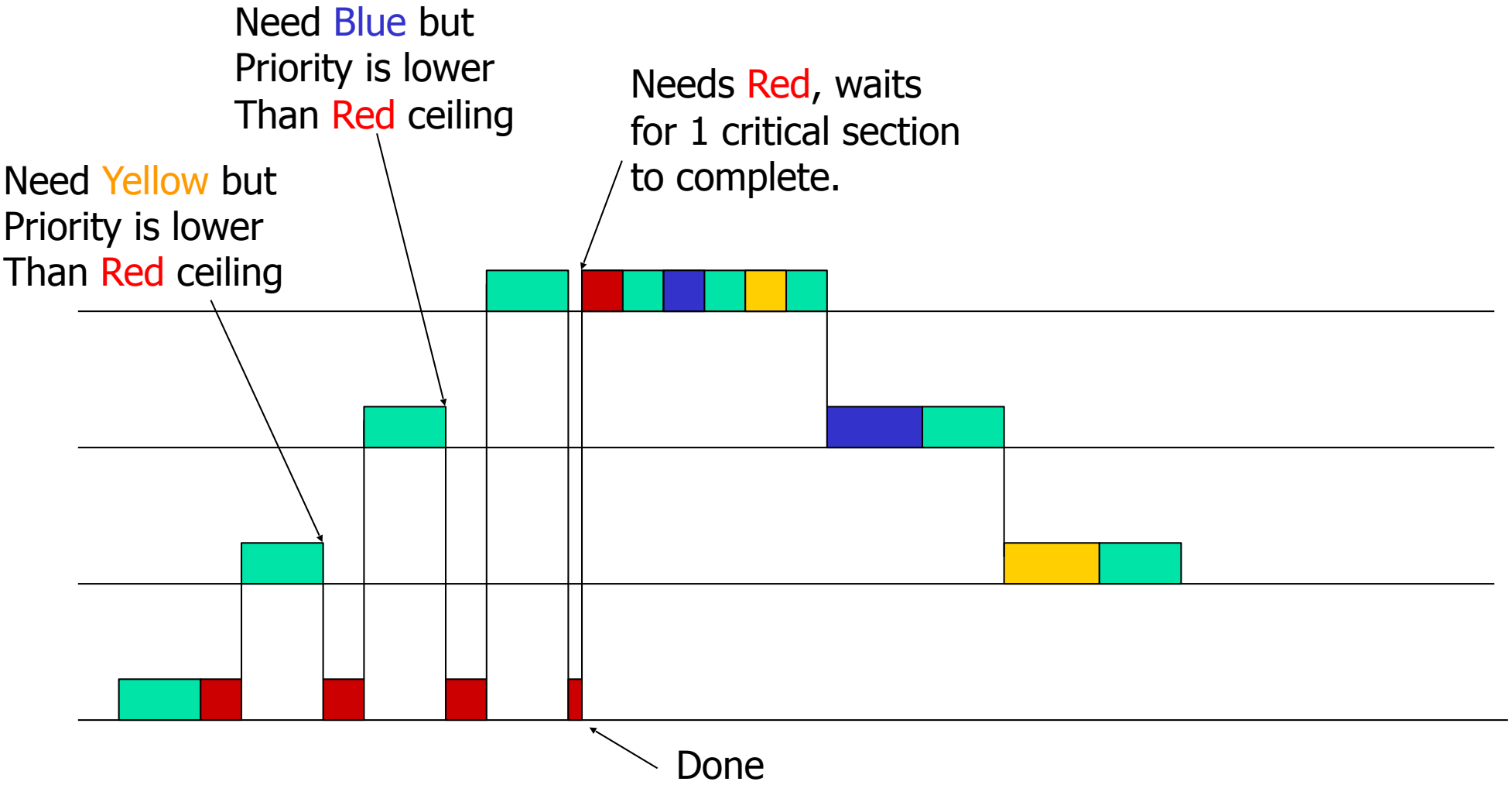
Priority Ceiling Protocol

Observe maximum blocking time for high priority process…



Priority Ceiling Protocol

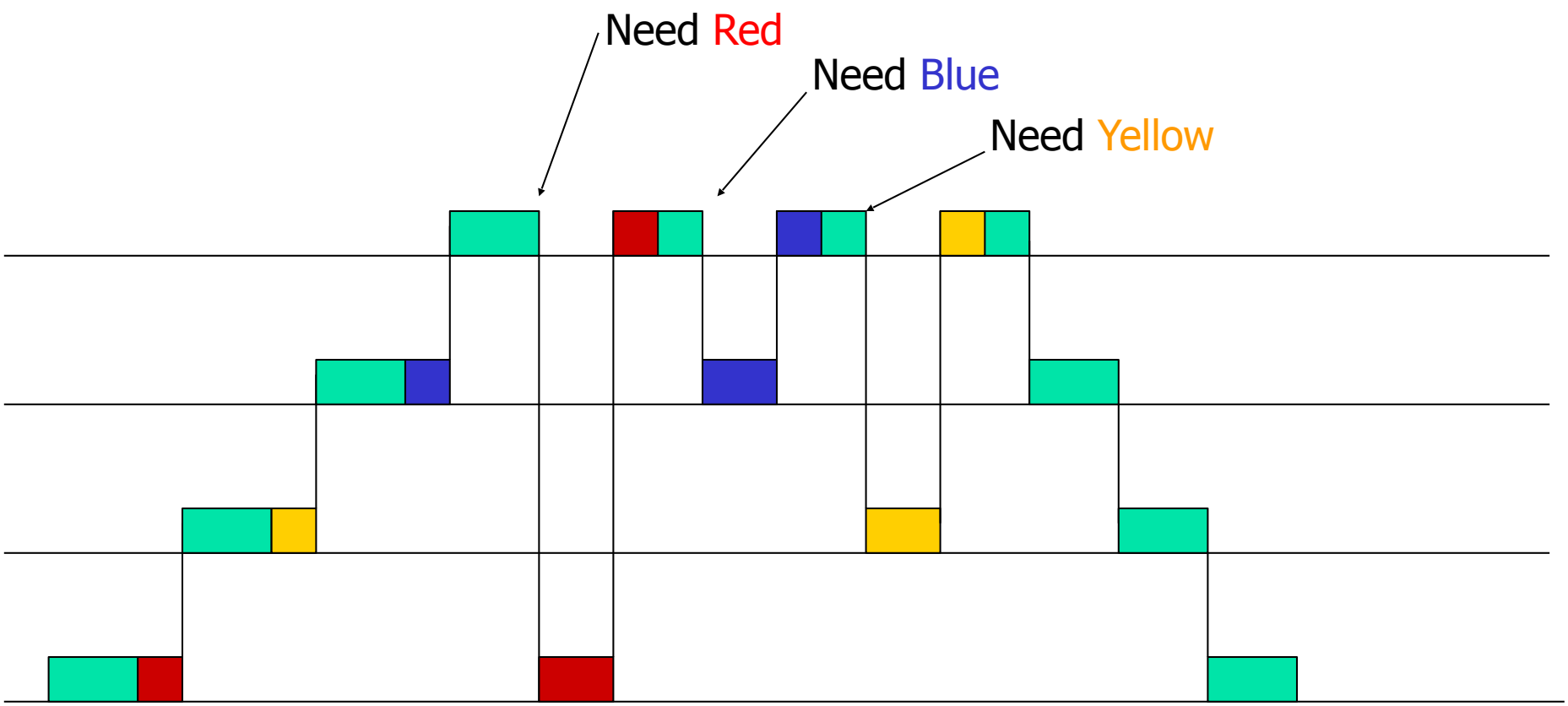
Observe maximum blocking time for high priority process…



Is the maximum blocking time property violated for intermediate priority processes?

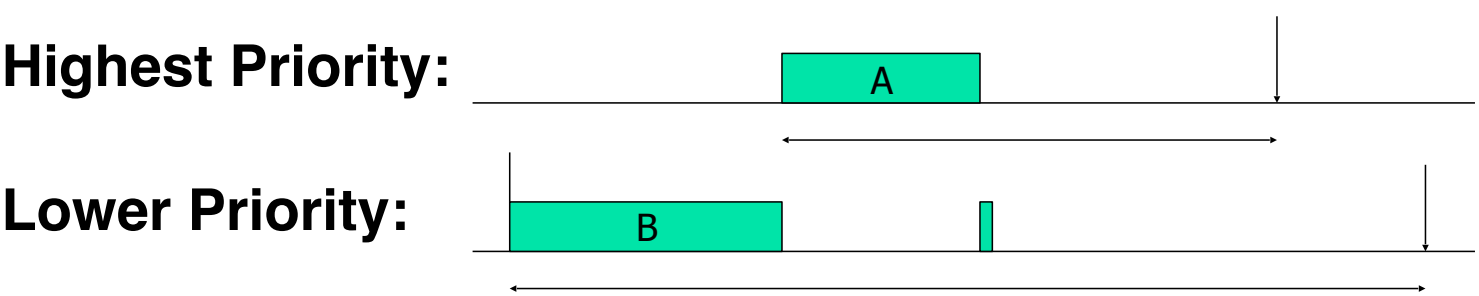
Maximum Blocking Time

Compare to the Priority Inheritance Protocol!!



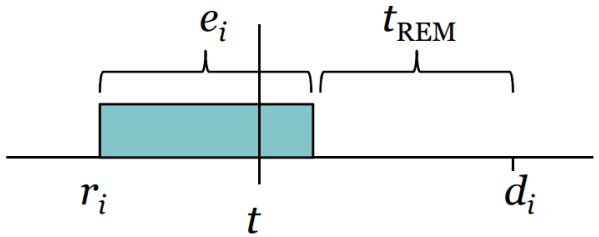
EDF Recap

- Recall that with Earliest Deadline First (EDF) dynamic priority system…
  - Assign priority to jobs based on their deadline
  - The Earlier the deadline, the higher the priority
  - Only knowledge of deadlines is known
- Priority is proportional to the absolute deadline
- "Preemption level" is proportional to the relative deadline.
- Observe that:
  - If A arrives after B and Priority (A) > Priority (B) then PreemptionLevel (A) > PreemptionLevel (B)
- **Any lingering source of inefficiency here?**



Least Slack Time (LST)

- **Idea: There's no point in completing a task earlier than its deadline. Other tasks many be executed first**
  - Assign priority to jobs based on slack time!
- A job J has deadline D, execution time E, and was released at time R
- At time t < D…
  - Remaining_Execution_Time = E - (t - R)
  - Slack Time = D - t - Remaining_Execution_Time



- <u>Both EDF and LST are optimal</u> for realtime deadlines if a valid scheduling exists.
- Since EDF gets the job done too…
  - Pros of LST?
  - Cons of LST?