The Linux Scheduler

What Are Scheduling Goals?

Goals of the Linux Scheduler

- Generate illusion of concurrency
- Maximize resource utilization (hint: mix CPU and I/O bound processes appropriately)
- Meet needs of both I/O-bound and CPU-bound processes
  - Give I/O-bound processes better interactive response
  - Do not starve CPU-bound processes
- Support Real-Time (RT) applications

Priority Structure

- Real-time processes have the top 99 priority levels.
- Non-real-time processes have levels 100-139

Two Fundamental Resource Sharing Mechanisms

- ?
- ?

Two Fundamental Resource Sharing Mechanisms

- Prioritization
- Resource partitioning

SCHED_FIFO

- Used for real-time processes
- Conventional preemptive fixed-priority scheduling
  - Current process continues to run until it ends or a higher-priority real-time process becomes runnable
- Same-priority processes are scheduled FIFO

SCHED_RR

- Used for real-time processes
- CPU “partitioning” among same priority processes
  - Current process continues to run until it ends or its time quantum expires
  - Quantum size determines the CPU share
- Processes of a lower priority run when no processes of a higher priority are present

SCHED_NORMAL

- Used for non-real-time processes
- Complex heuristic to balance the needs of I/O and CPU centric applications

Process Priority & Timeslice Recalculation

- Static priority
  - A “*nice*” value
  - Inherited from the parent process
  - Set up by user
- Dynamic priority
  - Based on static priority and applications characteristics (interactive or CPU-bound)
  - Favor interactive applications over CPU-bound ones
- Timeslice is mapped from priority

Heuristics

```
if (static priority < 120)
    Quantum = 20 (140 – static priority)
else
    Quantum = 5  (140 – static priority)
```

(in ms)

Heuristics

bonus = min (10, avg. sleep time / 100) (ms)

dynamic priority = max (100, min (static priority – bonus + 5, 139))

How & When to Preempt?

- Kernel sets the *need_resched* flag (per-process var) at various locations
  - scheduler_tick(), a process used up its timeslice
  - try_to_wake_up(), higher-priority process awaken
- Kernel checks *need_resched* at certain points, if safe, *schedule()* will be invoked
- User preemption
  - Return to user space from a system call or an interrupt handler
- Kernel preemption
  - A task in the kernel explicitly calls *schedule()*
  - A task in the kernel blocks (which results in a call to *schedule()*)

Other Scheduling Policies (... that you can implement)

- What if you want to maximize throughput?

Other Scheduling Policies
(... that you can implement)

- What if you want to maximize throughput?
  - Shortest job first!

Other Scheduling Policies
(... that you can implement)

- What if you want to meet all deadlines?

Other Scheduling Policies
(... that you can implement)

- What if you want to meet all deadlines?
  - Earliest deadline first!

Other Scheduling Policies
(... that you can implement)

- What if you want to meet all deadlines?
  - Earliest deadline first!
- Problem?

Other Scheduling Policies (... that you can implement)

- What if you want to meet all deadlines?
  - Earliest deadline first!
- Problem?
  - Works only if you are not “overloaded”. If the total amount of work is more than capacity, a domino effect occurs as you always choose the task with the nearest deadline (that you have the least chance of finishing by the deadline), so you may miss a lot of deadlines!

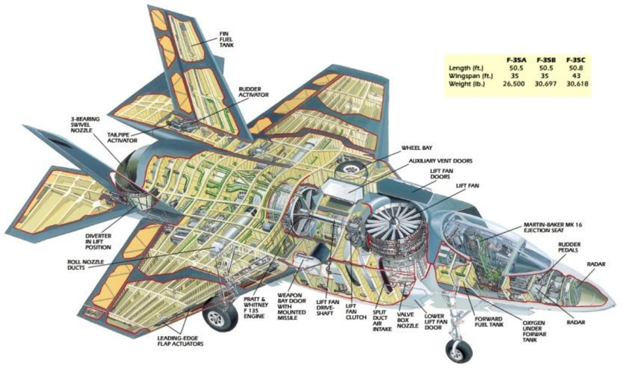
Example of EDF Domino Effect

- Problem:
  - You have a homework due tomorrow (Thursday), a homework due Friday, and a homework due Saturday
  - It takes on average 1.5 days to finish a homework.
- Question: What is your best (scheduling) policy?

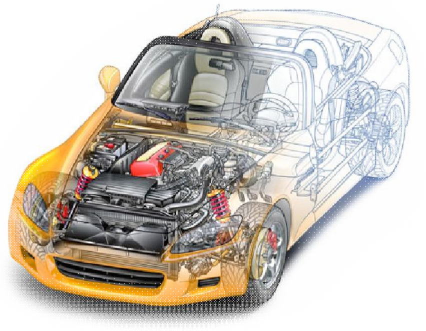
Example of EDF Domino Effect

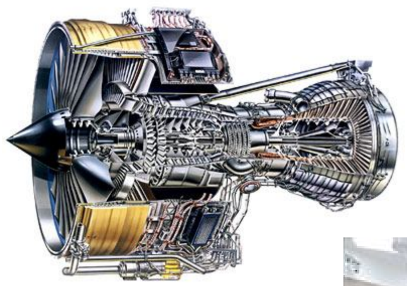
- Problem:
  - You have a homework due tomorrow (Thursday), a homework due Friday, and a homework due Saturday
  - It takes on average 1.5 days to finish a homework.
- Question: What is your best (scheduling) policy?
  - Note that EDF is bad: It always forces you to work on the next deadline, but you have only one day between deadlines which is not enough to finish a 1.5 day homework – you might not complete any of the three homeworks!
  - You could instead skip the Thursday homework and work on the next two, which you could then finish by their deadlines

Scheduling Periodic Tasks
(in Embedded Systems)



	F-35A	F-35B	F-35C
Length (ft.)	50.5	50.5	50.8
Wingspan (ft.)	35	35	43
Weight (lb.)	26,500	30,697	30,618





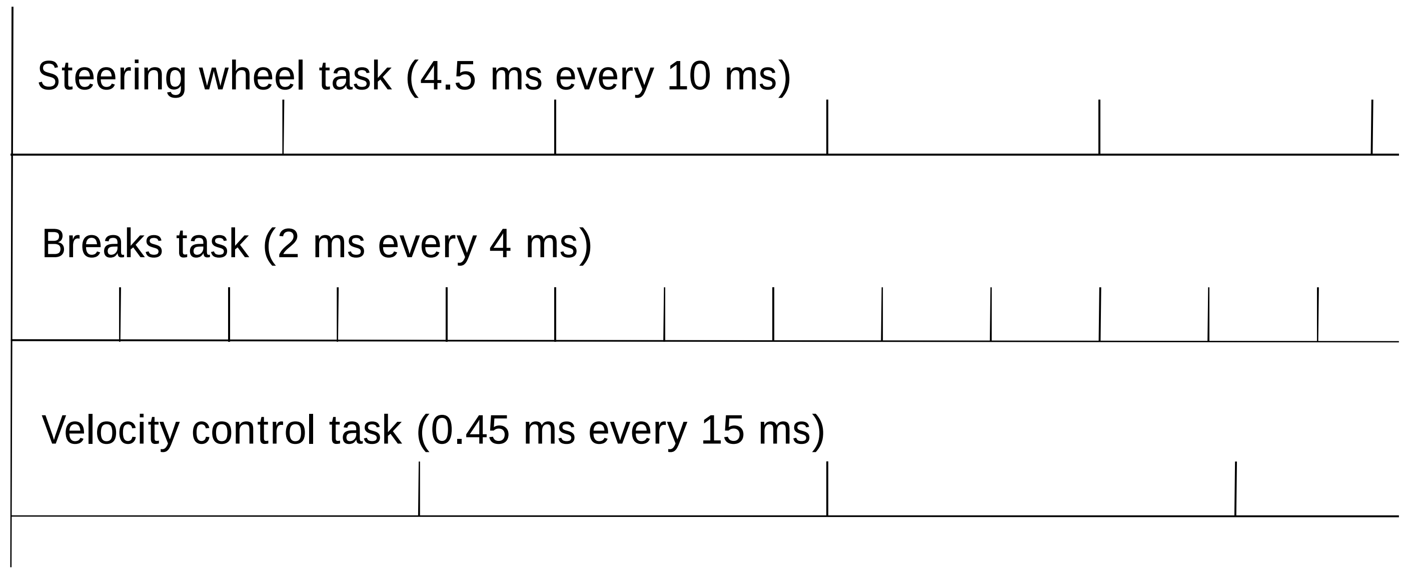
Embedded Computing Systems

Drive-by-Wire Example

- Consider a control system in a drive-by-wire vehicle
  - Steering wheel sampled every 10 ms – wheel positions adjusted accordingly (computing the adjustment takes 4.5 ms of CPU time)
  - Breaks sampled every 4 ms – break pads adjusted accordingly (computing the adjustment takes 2ms of CPU time)
  - Velocity is sampled every 15 ms – acceleration is adjusted accordingly (computing the adjustment takes 0.45 ms)
  - For safe operation, adjustments must always be computed before the next sample is taken
- What scheduling policy to use?

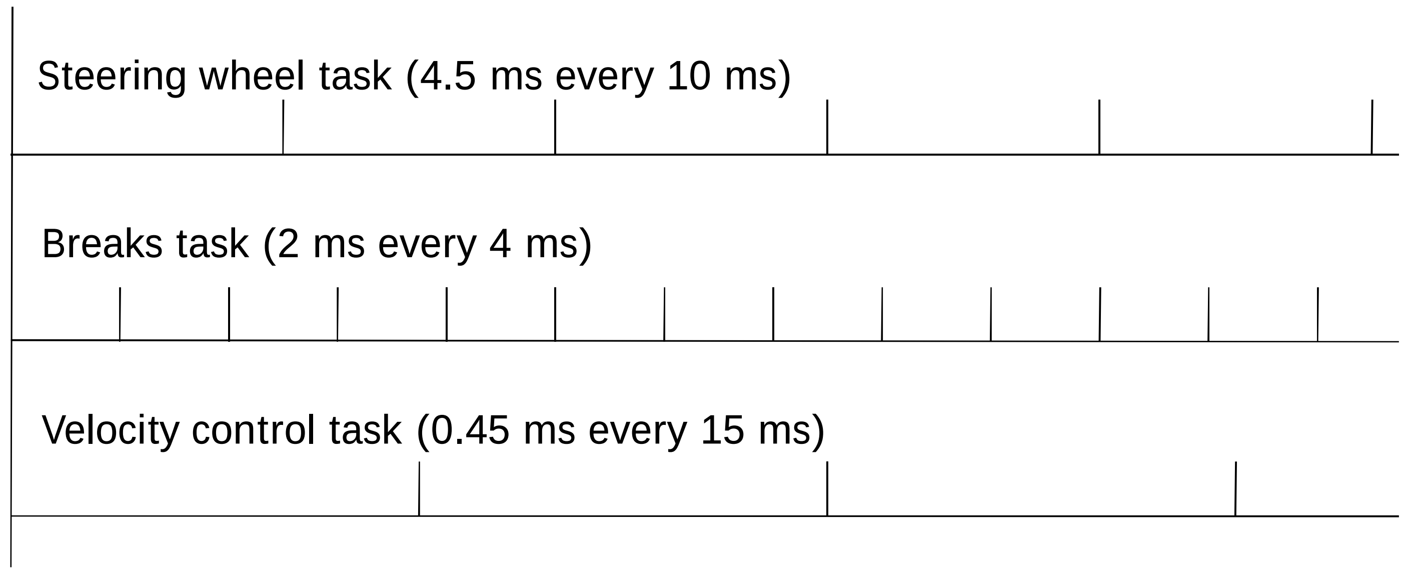
Drive-by-Wire Example

- Find a schedule that makes sure all task invocations meet their deadlines


Steering wheel task (4.5 ms every 10 ms)
Breaks task (2 ms every 4 ms)
Velocity control task (0.45 ms every 15 ms)

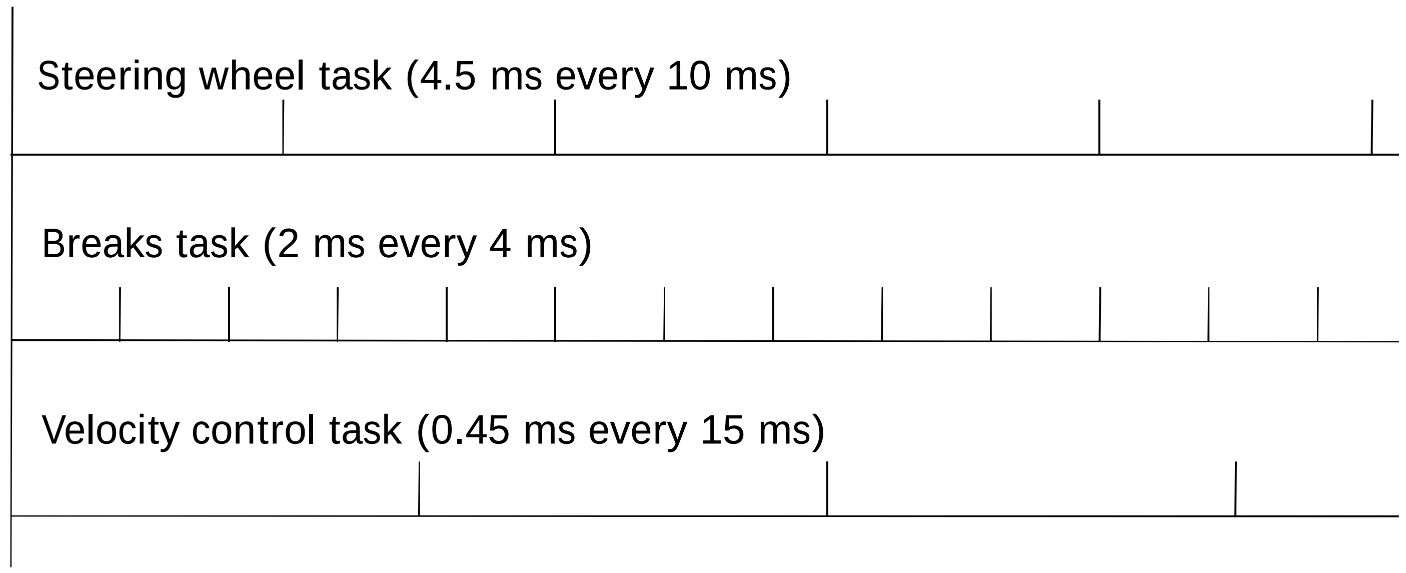
Drive-by-Wire Example

- Sanity check: Is the processor over-utilized? (e.g., if you have 5 homeworks due this time tomorrow, each takes 6 hours, then 5x6 = 30 > 24 → you are overutilized)


Steering wheel task (4.5 ms every 10 ms)
Breaks task (2 ms every 4 ms)
Velocity control task (0.45 ms every 15 ms)

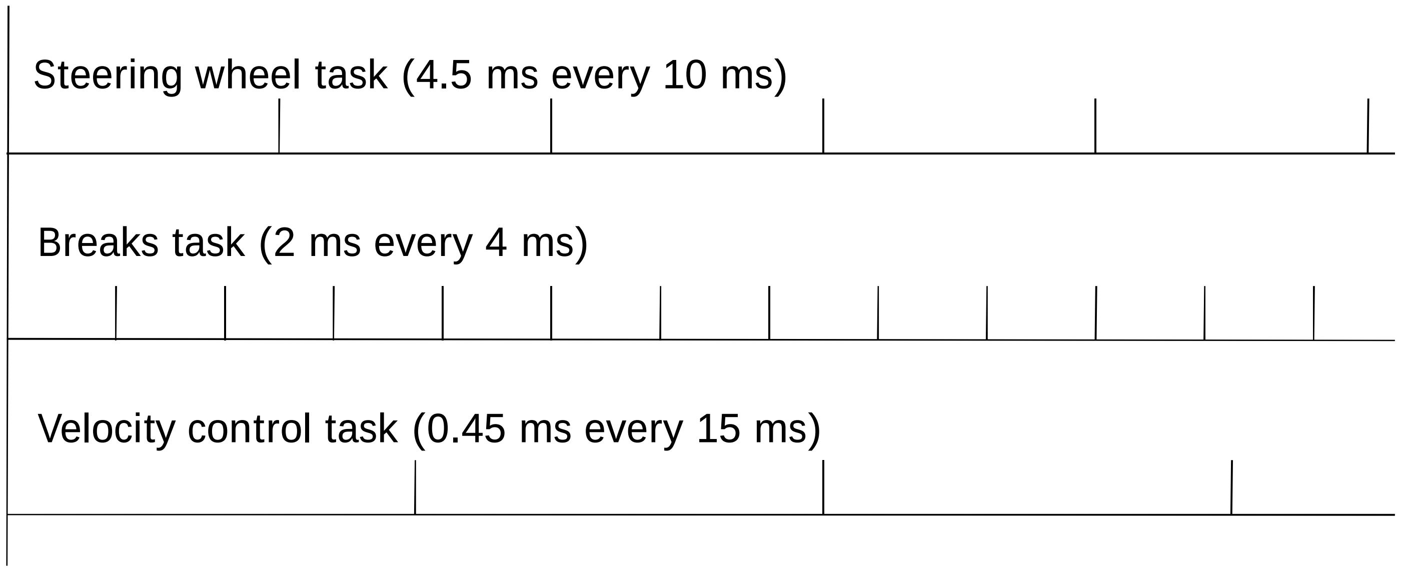
Drive-by-Wire Example

- Sanity check: Is the processor over-utilized? (e.g., if you have 5 homeworks due this time tomorrow, each takes 6 hours, then 5x6 = 30 > 24 → you are overutilized)
  - Hint: Check if processor utilization > 100%


Steering wheel task (4.5 ms every 10 ms)
Breaks task (2 ms every 4 ms)
Velocity control task (0.45 ms every 15 ms)

Task Scheduling

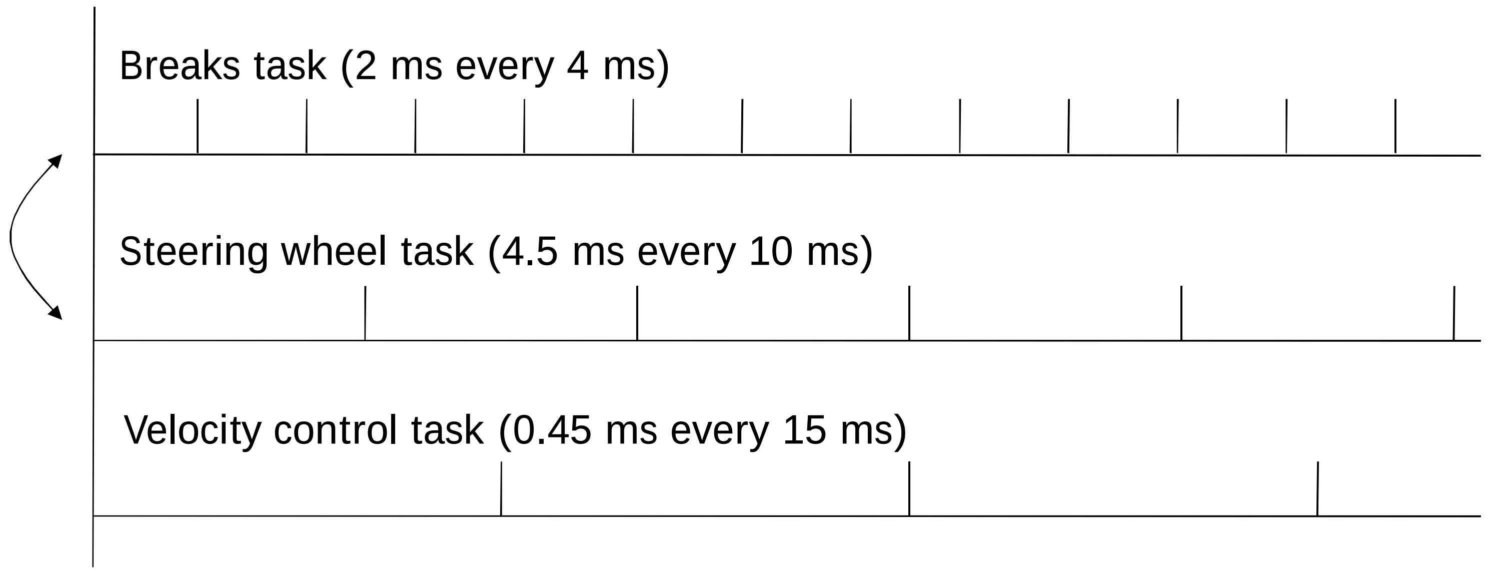
- How to assign task priorities?


Steering wheel task (4.5 ms every 10 ms)
Breaks task (2 ms every 4 ms)
Velocity control task (0.45 ms every 15 ms)

How to assign priorities to tasks?

Task Scheduling

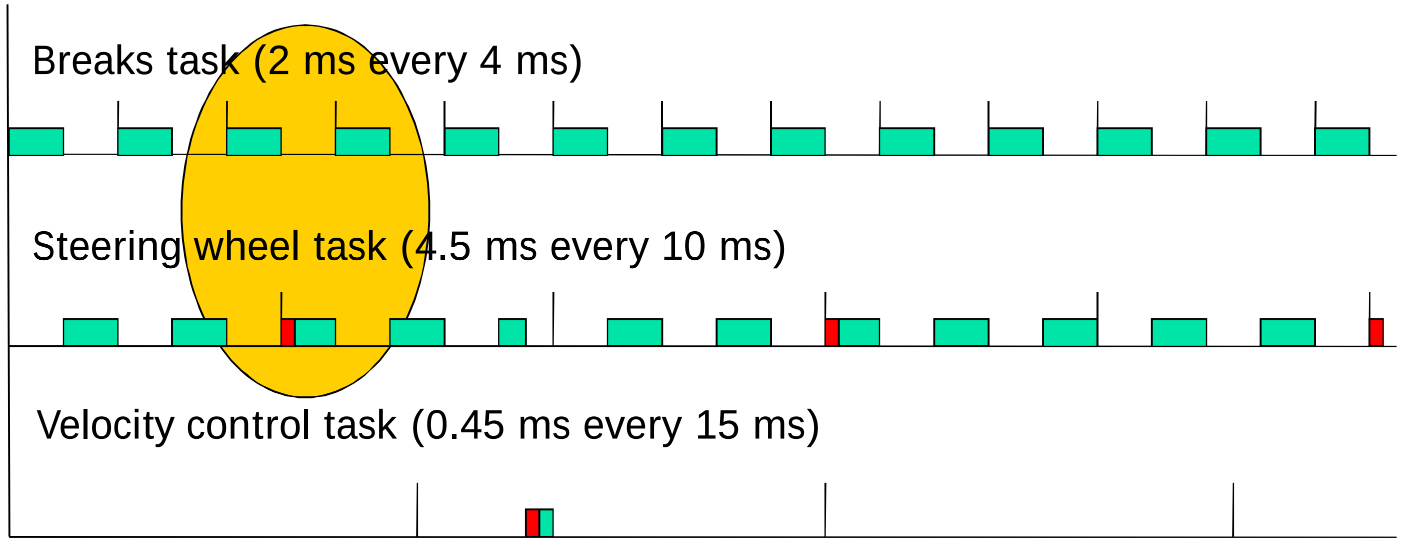
- How to assign task priorities?
  - Rate Monotonic (large rate = higher priority)


Breaks task (2 ms every 4 ms)
Steering wheel task (4.5 ms every 10 ms)
Velocity control task (0.45 ms every 15 ms)

Intuition: Urgent tasks should be higher in priority

Problem?

- Deadlines are missed!
- Average Utilization < 100%

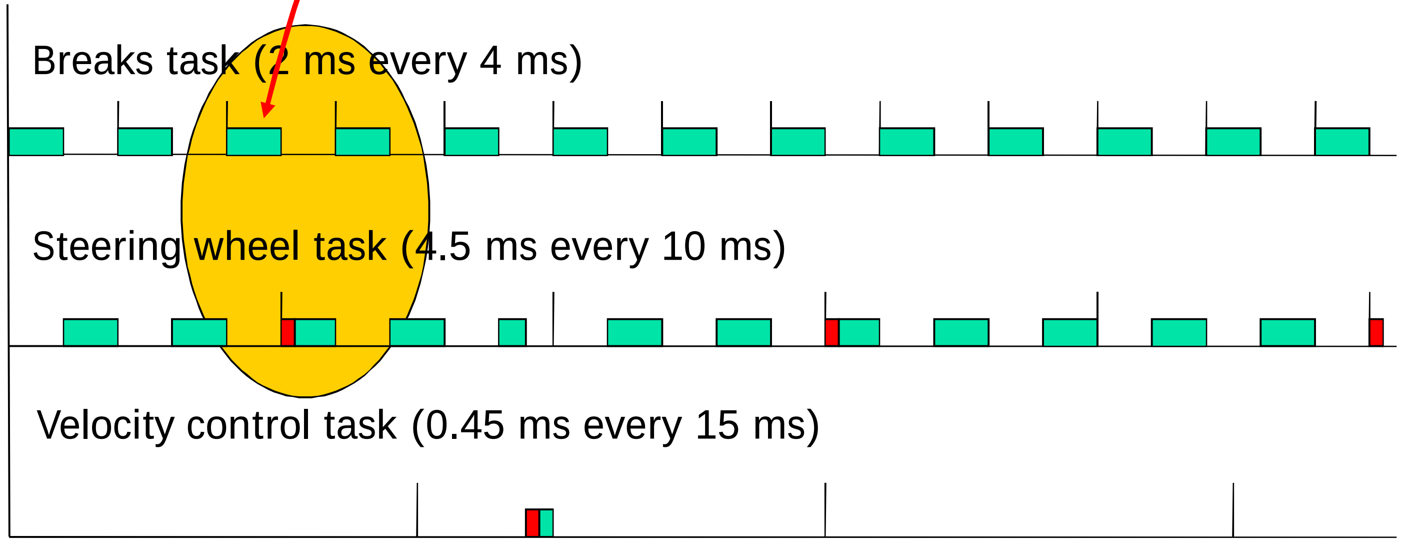


Breaks task (2 ms every 4 ms)
Steering wheel task (4.5 ms every 10 ms)
Velocity control task (0.45 ms every 15 ms)

Problem?

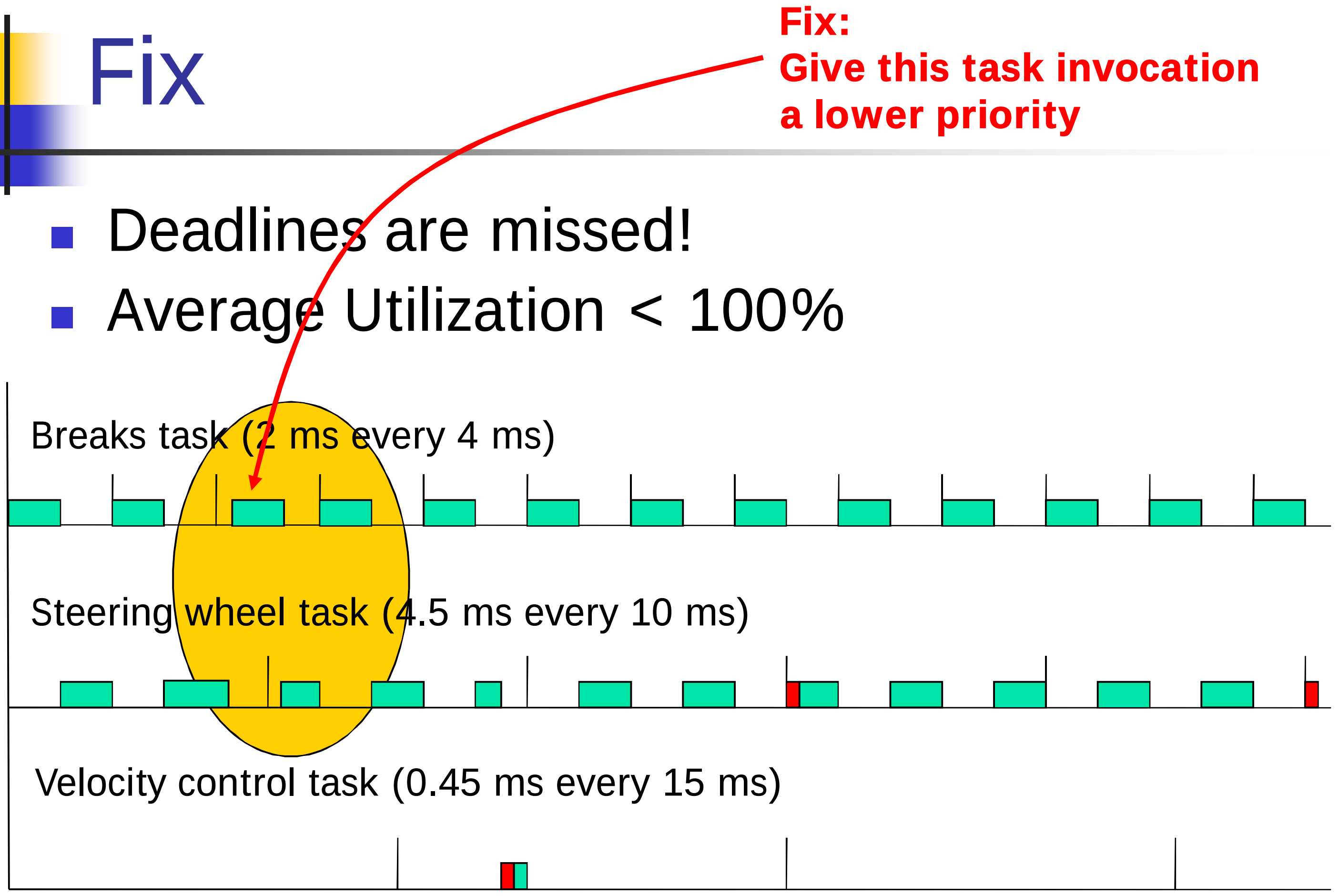
Fix:
Give this task invocation a lower priority

- Deadlines are missed!
- Average Utilization < 100%



Fix

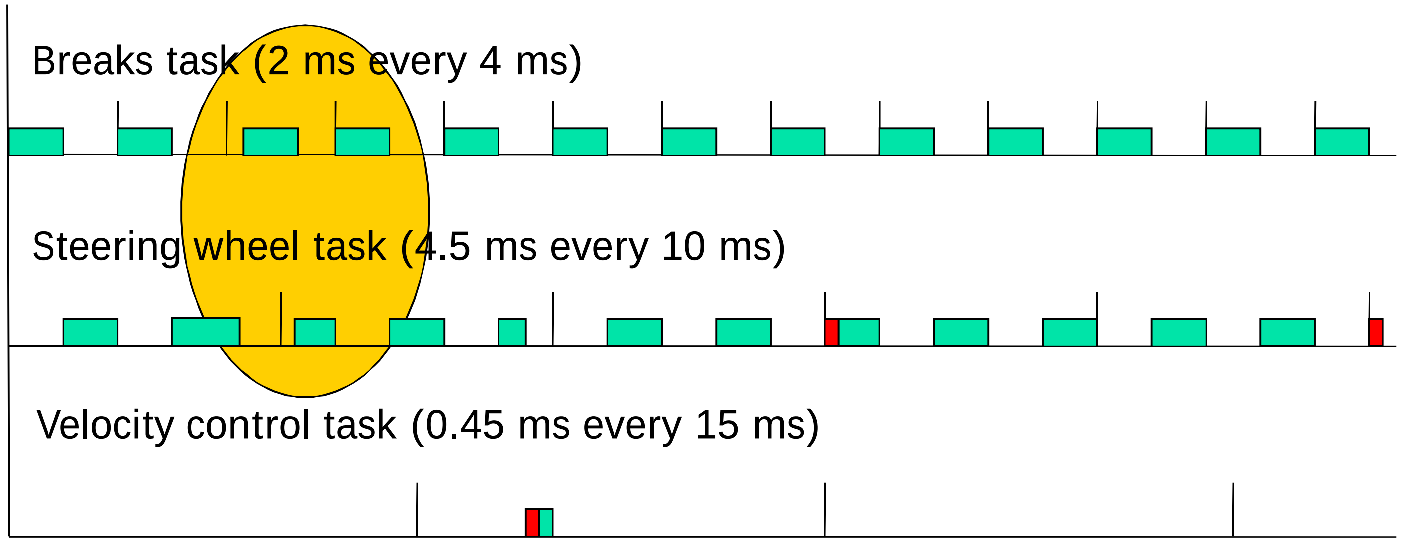
- Deadlines are missed!
- Average Utilization < 100%



Fix:
Give this task invocation
a lower priority
Breaks task (2 ms every 4 ms)
Steering wheel task (4.5 ms every 10 ms)
Velocity control task (0.45 ms every 15 ms)

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


Breaks task (2 ms every 4 ms)
Steering wheel task (4.5 ms every 10 ms)
Velocity control task (0.45 ms every 15 ms)

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

Real-time Scheduling of Periodic Tasks

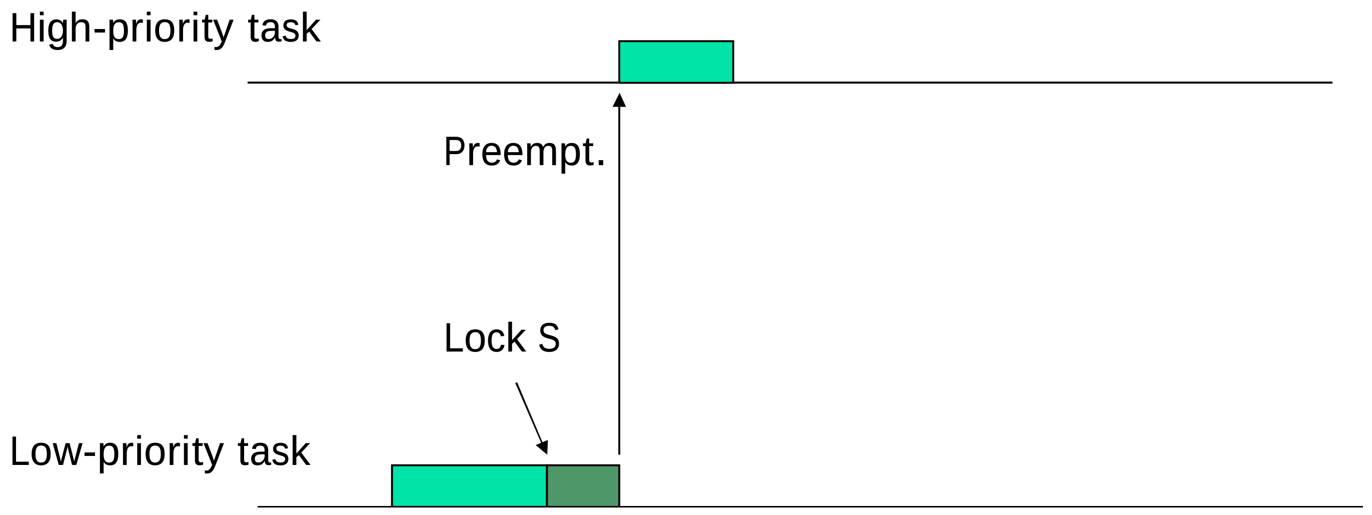
- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)
- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent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

Advanced Topic: Locking and Priority Inversion

- What if a higher-priority process needs a resource locked by a lower-priority process?
  - How long will the higher priority process have to wait for lower-priority execution?

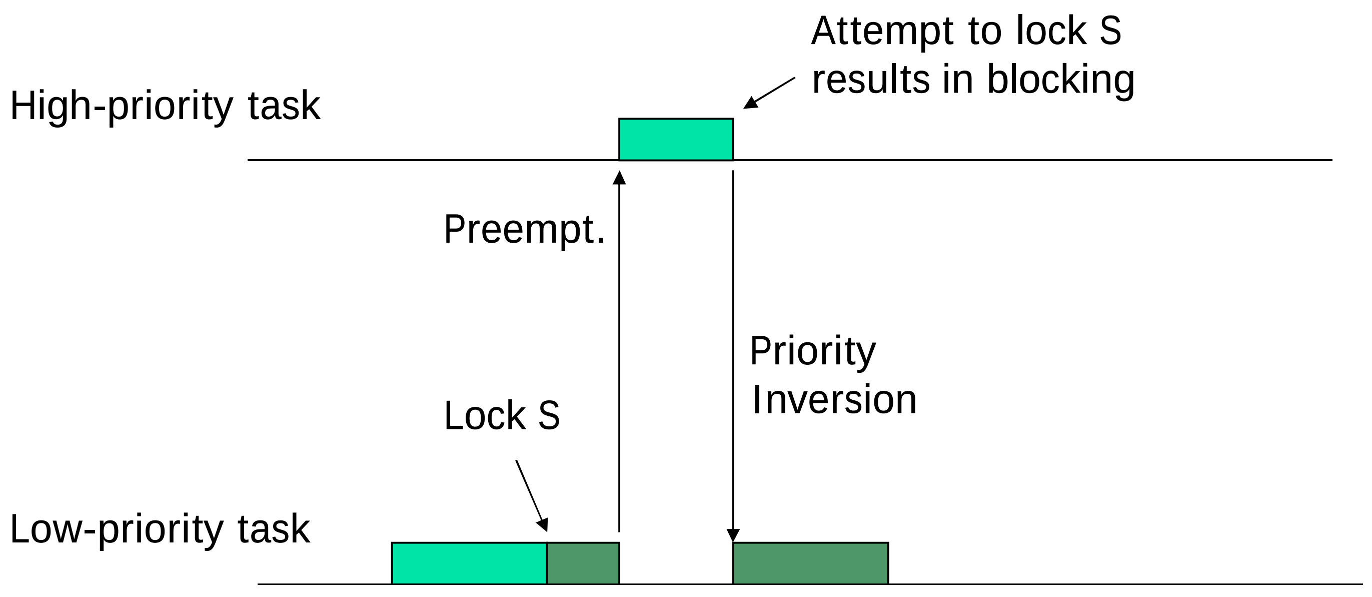
Priority Inversion

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



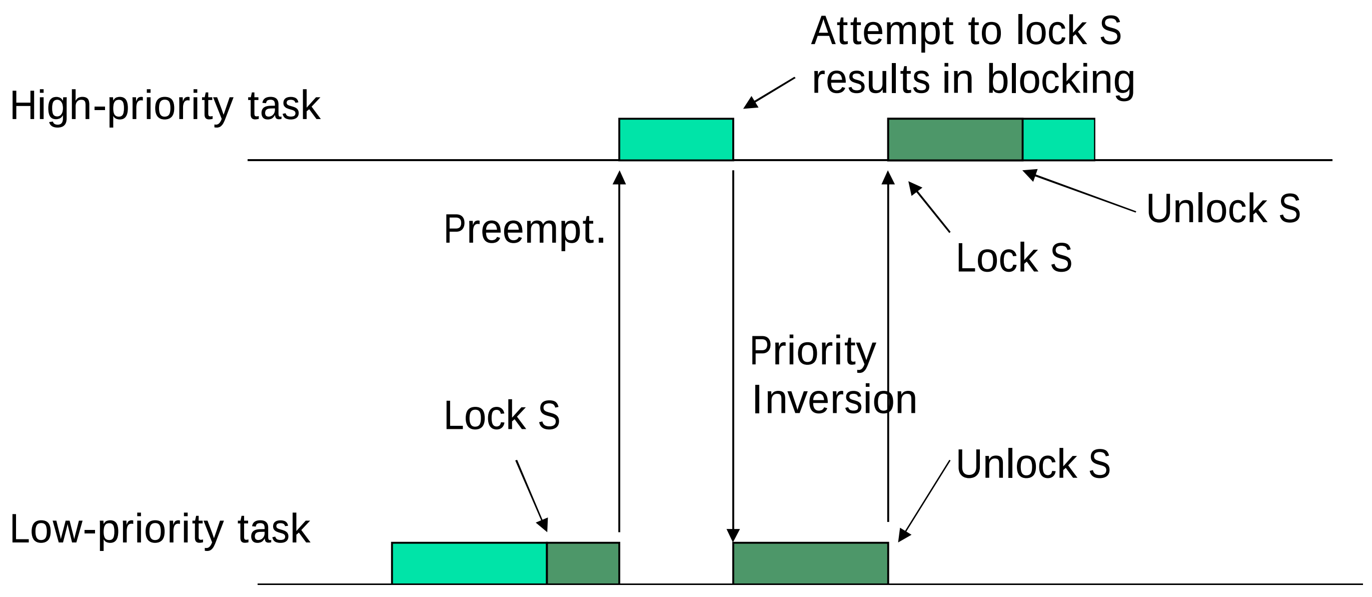
Priority Inversion

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



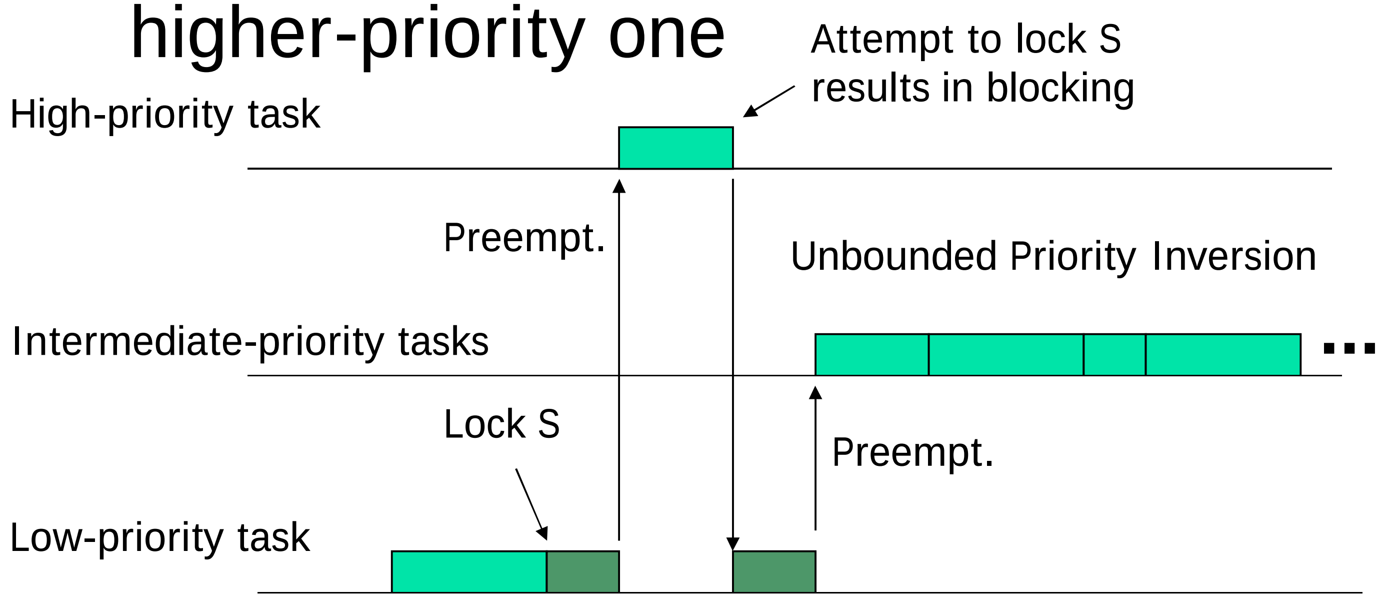
Priority Inversion

- How to account for priority inversion?



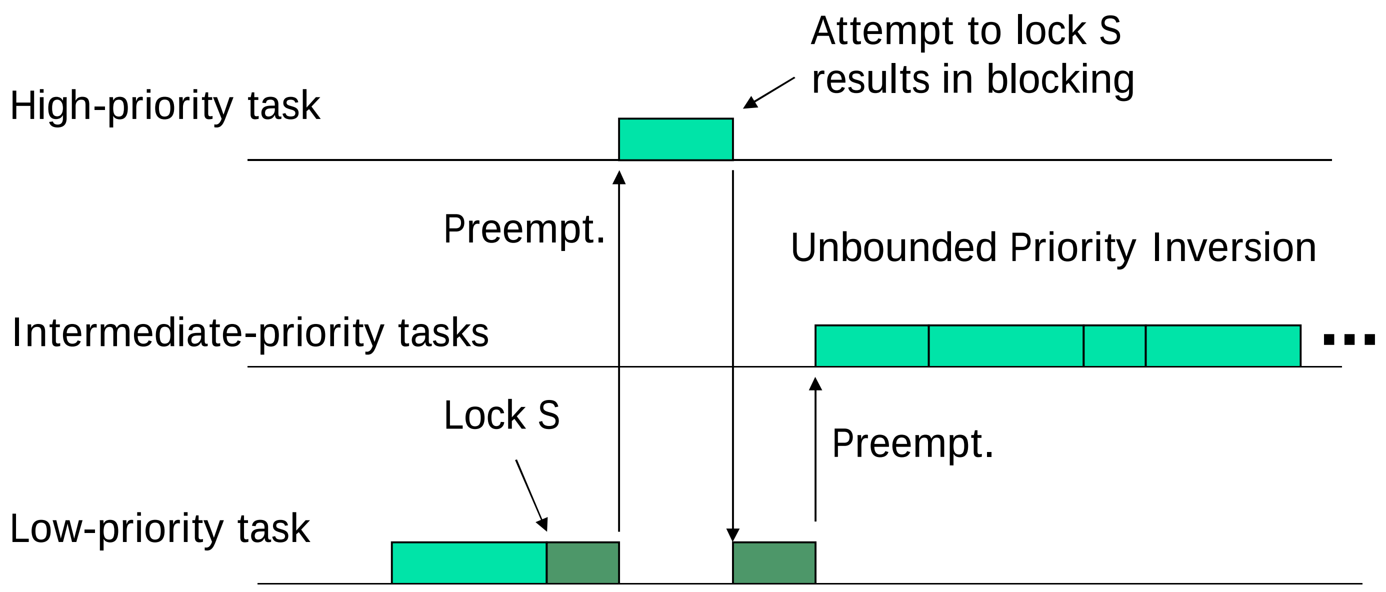
Unbounded Priority Inversion

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



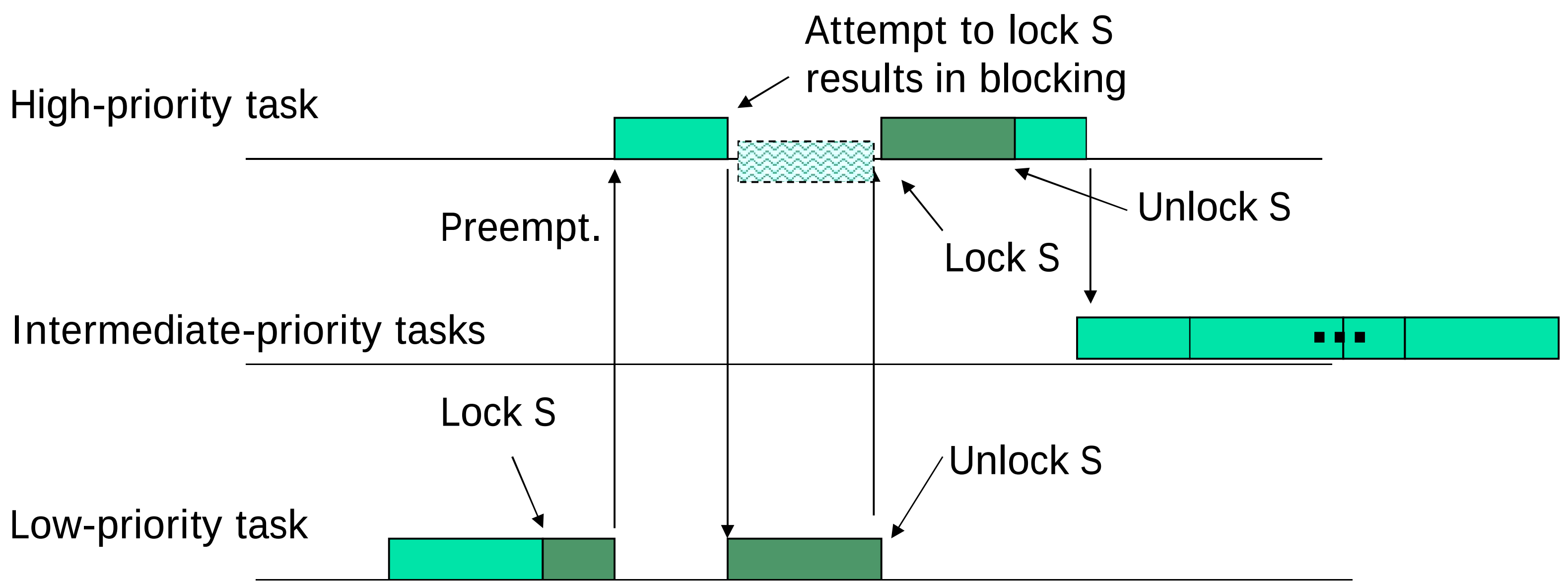
Unbounded Priority Inversion

- How to prevent unbounded priority inversion?


Attempt to lock S results in blocking
High-priority task
Preempt.
Unbounded Priority Inversion
Intermediate-priority tasks
...
Lock S
Preempt.
Low-priority task

Priority Inheritance Protocol

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

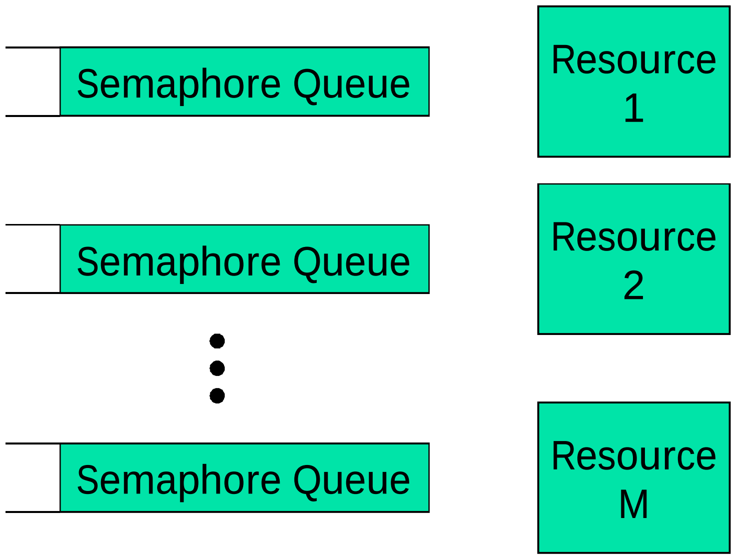


Priority Inheritance Protocol

- Question: What is the longest time a task can wait for lower-priority tasks?
  - Let there be N tasks and M locks
  - Let the largest critical section of task i be of length B_i
- Answer: ?

Computing the Maximum Priority Inversion Time

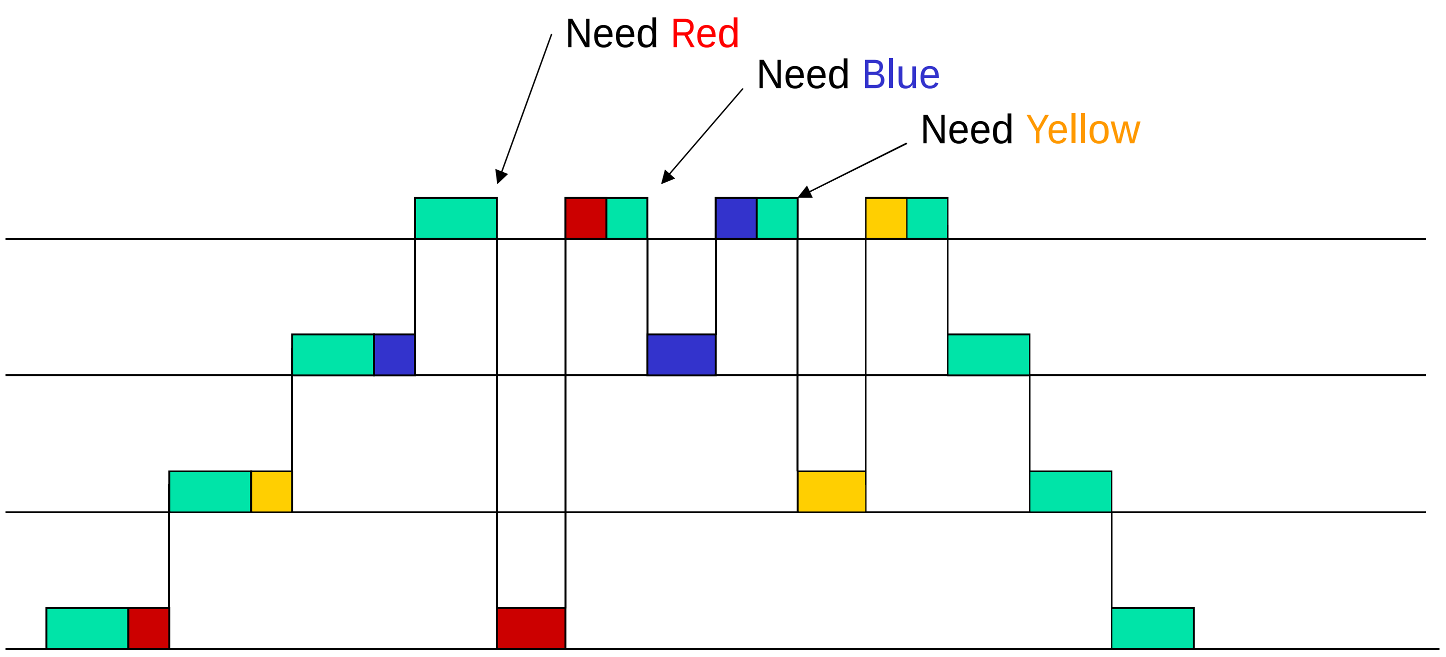
- Consider the instant when a high-priority task that arrives.
  - What is the most it can wait for lower priority ones?



If I am a task, priority inversion occurs when
(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)

Maximum Blocking Time

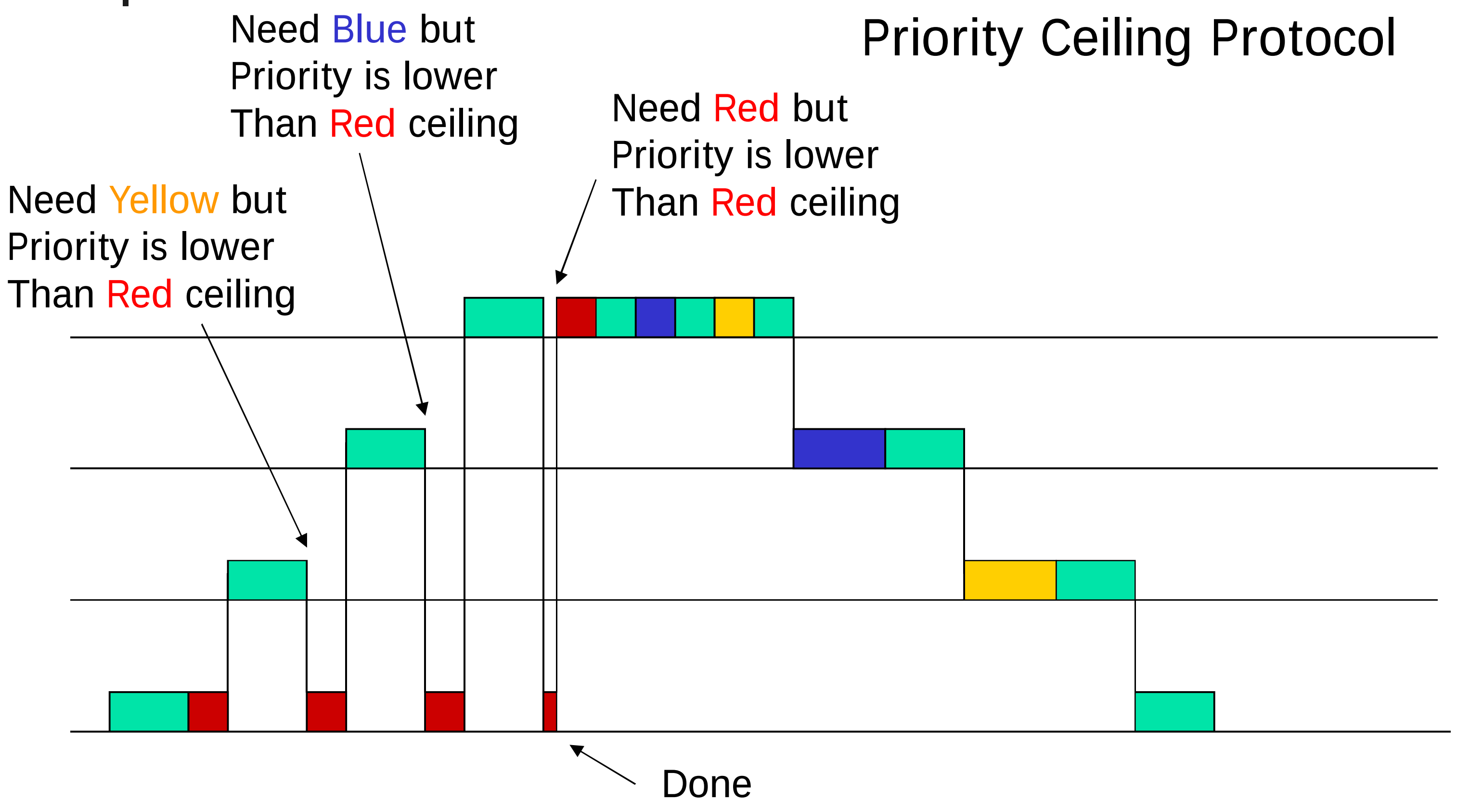
Priority Inheritance Protocol



Priority Ceiling Protocol

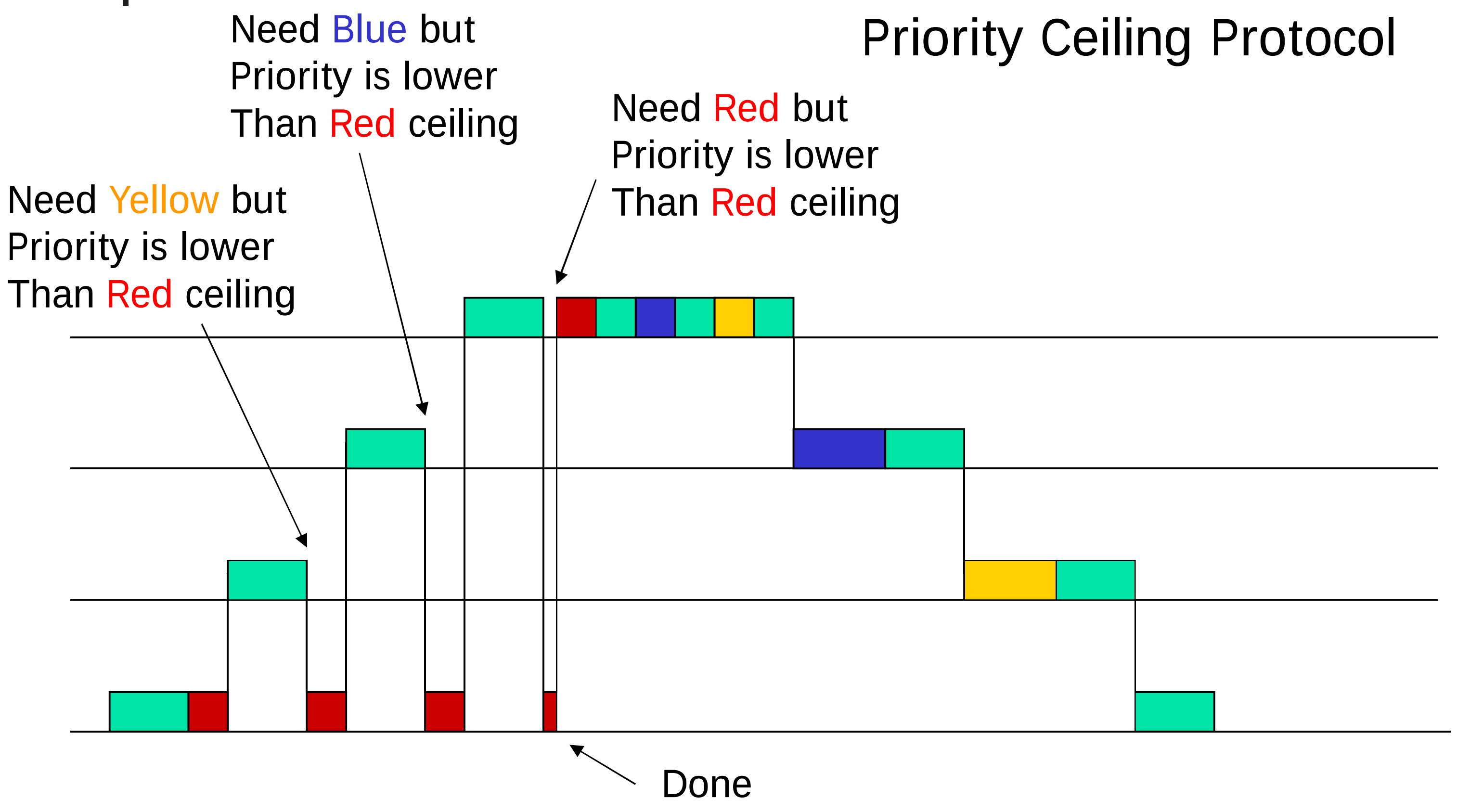
- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it
- A task that requests a lock R_k is denied if its priority is not higher than the highest priority ceiling of all currently locked semaphores (say it belongs to semaphore R_h)
  - The task is said to be blocked by the task holding lock R_h
- A task inherits the priority of the top higher-priority task it is blocking

Maximum Blocking Time



Maximum Blocking Time

Priority Ceiling Protocol



Questions

- 1) In Linux, you want to schedule a group of processes strictly in priority order (no time slices). Which policy should you choose?
- 2) In Linux, you want to give a set of 3 processes (that never block) the same priority but apportion the CPU among them by the ratios: 30%, 20% and 50%. Which policy should you choose?
- 3) In Linux, if a process (that is scheduled by SCHED_NORMAL) never blocks, at most how many priority levels its dynamic priority can drop compared to its static priority?