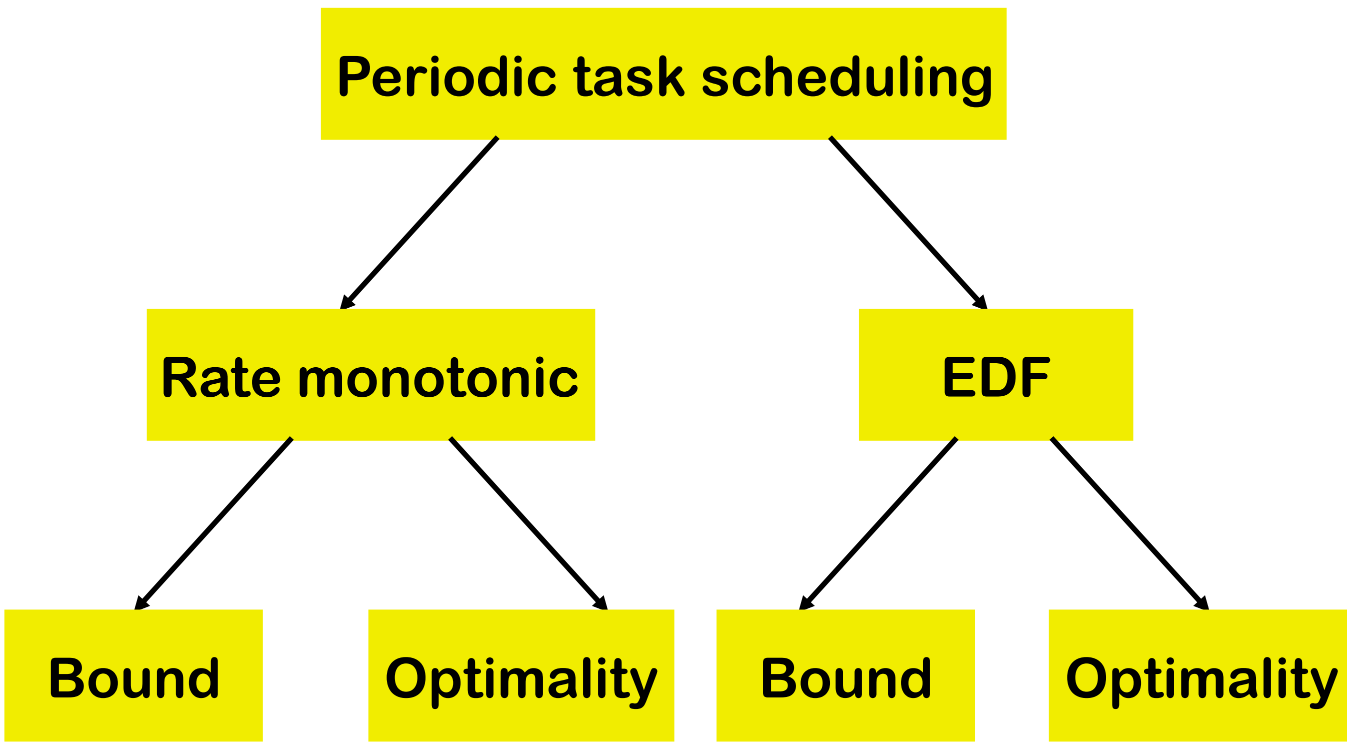
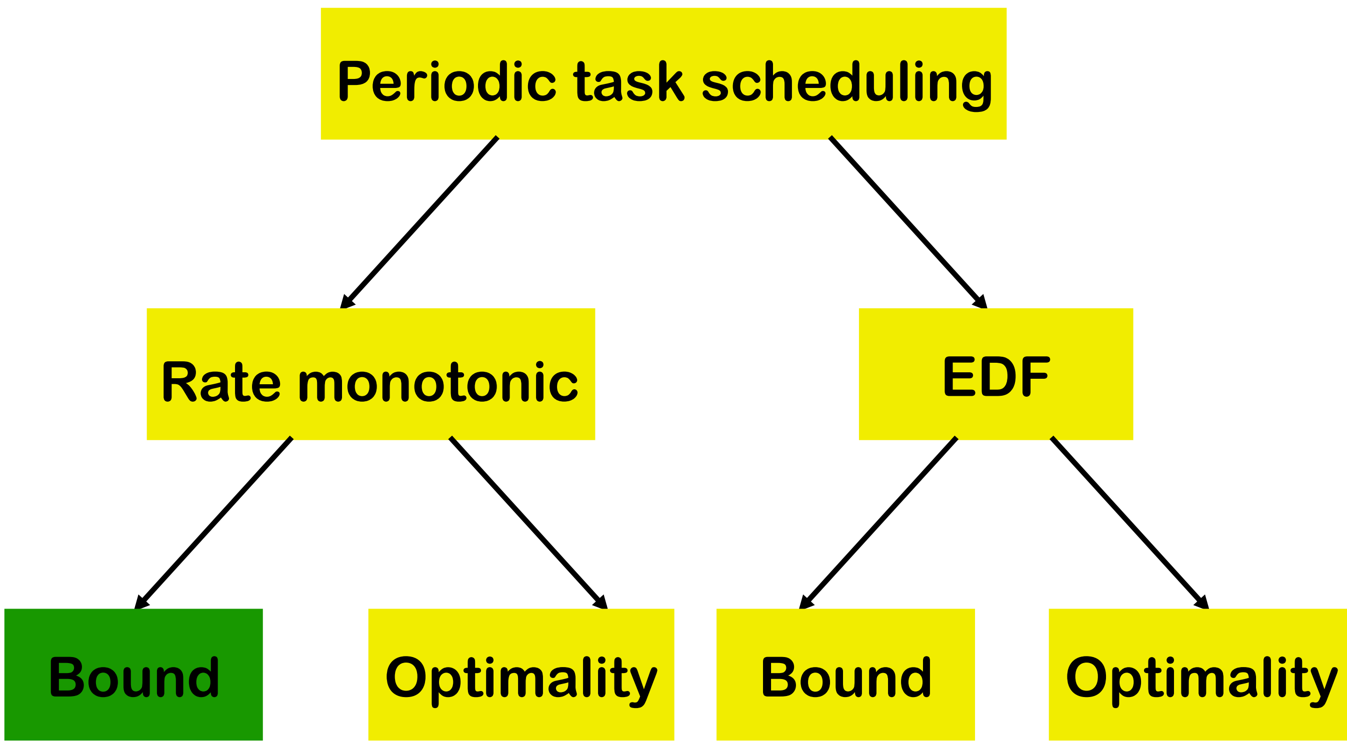
Periodic task scheduling

Optimality of rate monotonic scheduling (among static priority policies)
Utilization bound for EDF
Optimality of EDF (among dynamic priority policies)
Tick-driven scheduling (OS issues)

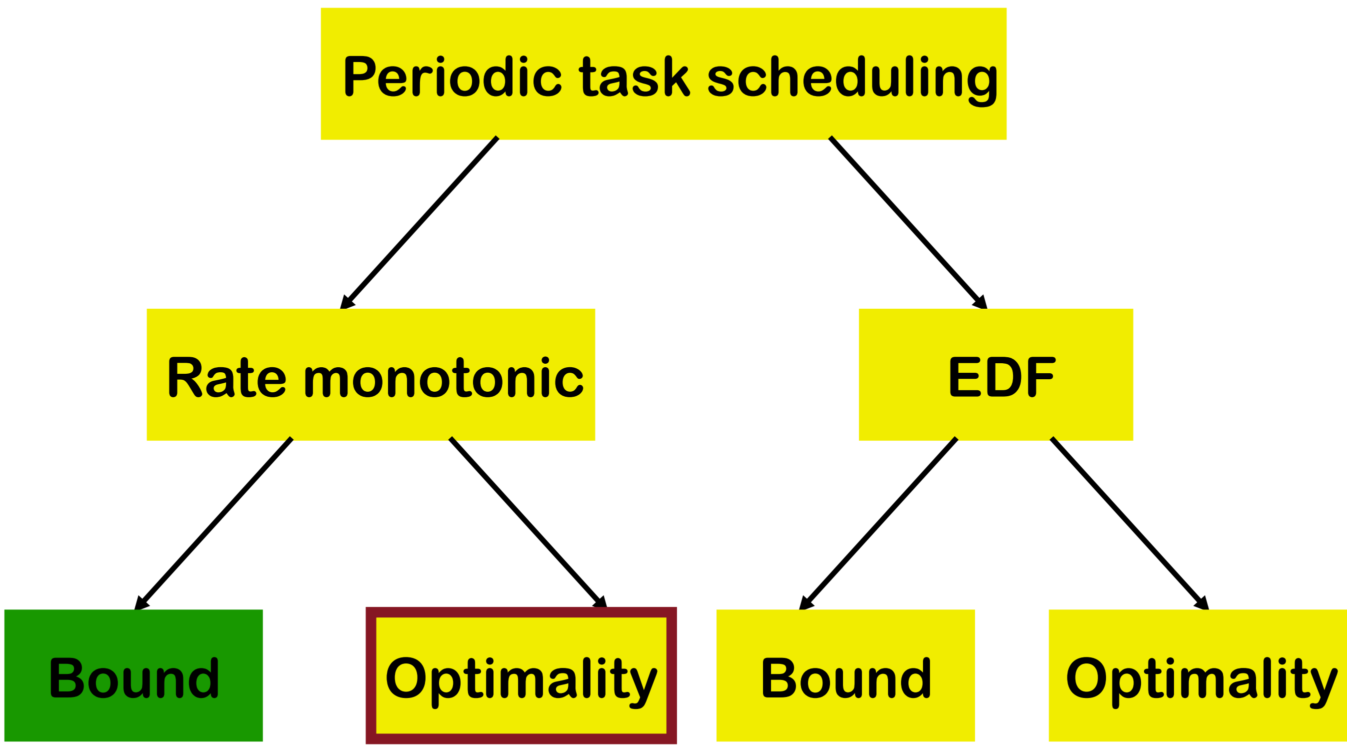
Lecture outline



Lecture outline



Lecture outline



Rate monotonic scheduling

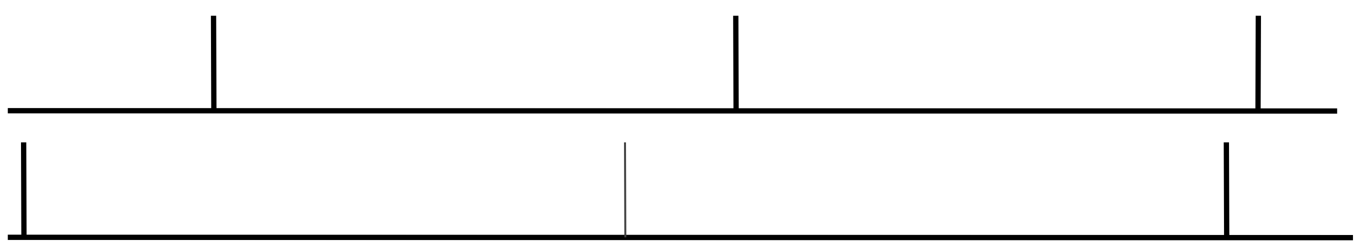
- Rate monotonic scheduling is the optimal fixed-priority (or static-priority) scheduling policy for periodic tasks.
  - Optimality (Trial #1):

Rate monotonic scheduling

- Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.
  - Optimality (Trial #1): If any other fixed-priority scheduling policy can meet deadlines, so can RM.

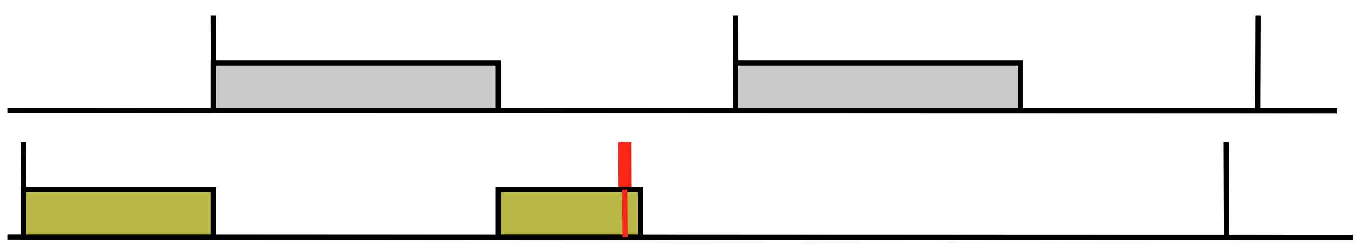
Rate monotonic scheduling

- Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.
  - Optimality (Trial #1): If any other fixed-priority scheduling policy can meet deadlines, so can RM



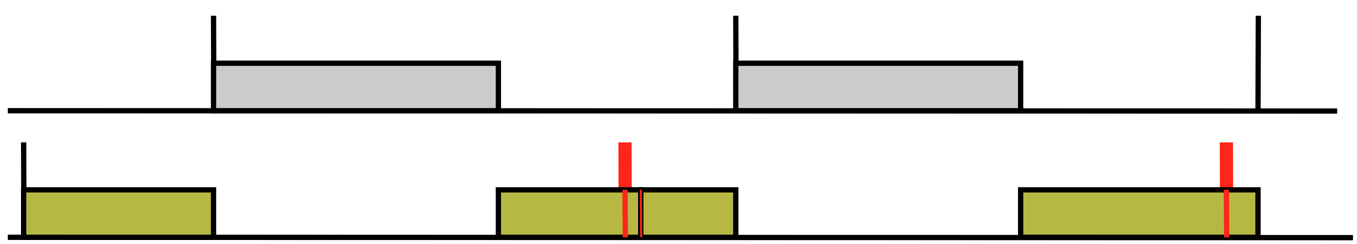
Rate monotonic scheduling

- **Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.**
  - **Optimality (Trial #1): If any other fixed-priority scheduling policy can meet deadlines, so can RM**



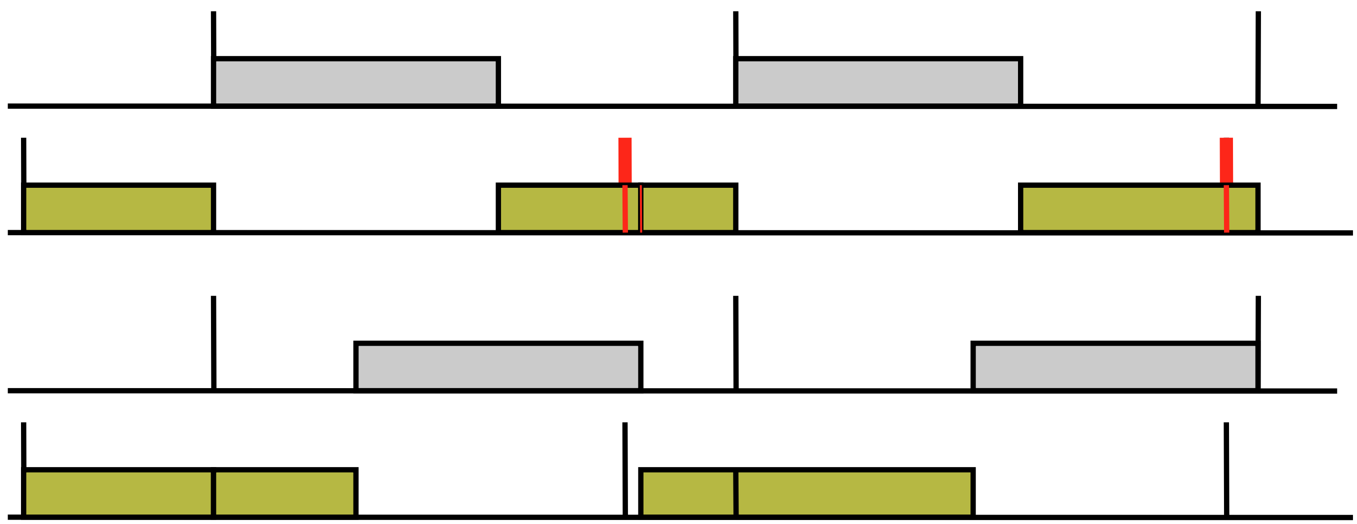
Rate monotonic scheduling

- Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.
  - Optimality (Trial #1): If any other fixed-priority scheduling policy can meet deadlines, so can RM



Rate monotonic scheduling

- Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.
  - Optimality (Trial #1): If any other fixed-priority scheduling policy can meet deadlines, so can RM



Rate monotonic scheduling

- **Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.**
  - **Optimality (Trial #2): If any other fixed-priority scheduling policy can meet deadlines in the worst case scenario, so can RM.**
- **How do we prove it?**

Rate monotonic scheduling

- Rate monotonic scheduling is the optimal fixed-priority scheduling policy for periodic tasks.
  - Optimality (Trial #2): If any other fixed-priority scheduling policy can meet deadlines in the worst case scenario, so can RM.
- How do we prove it?
  - Consider the worst case scenario
  - Show that if someone else can schedule then RM can

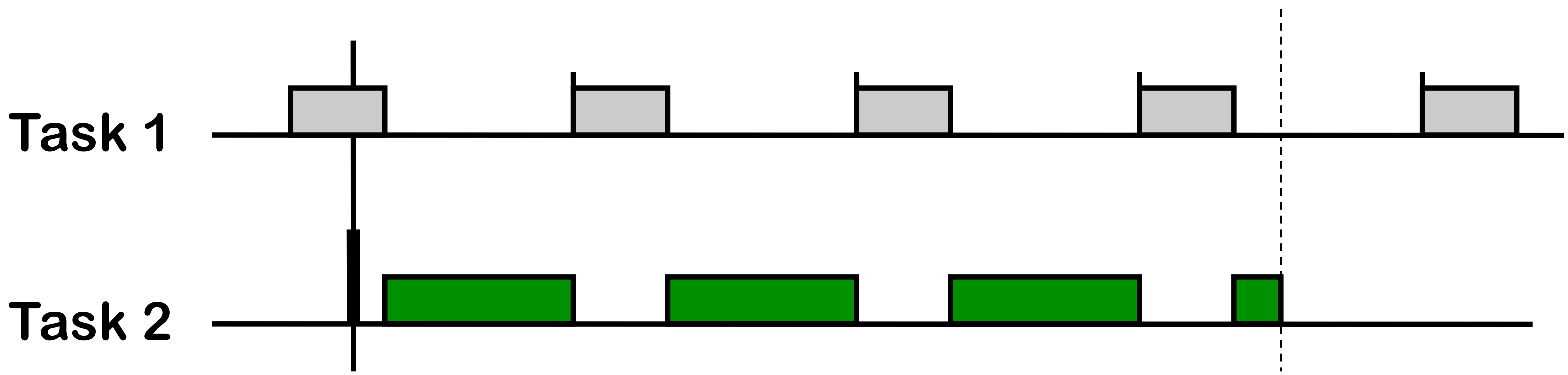
The worst-case scenario

- Q: When does a periodic task, *T*, experience the maximum delay?
- A: When it arrives together with all the higher-priority tasks (critical instance)

- Idea for the proof
  - If some higher-priority task does not arrive together with *T*, aligning the arrival times can only increase the completion time of *T.*

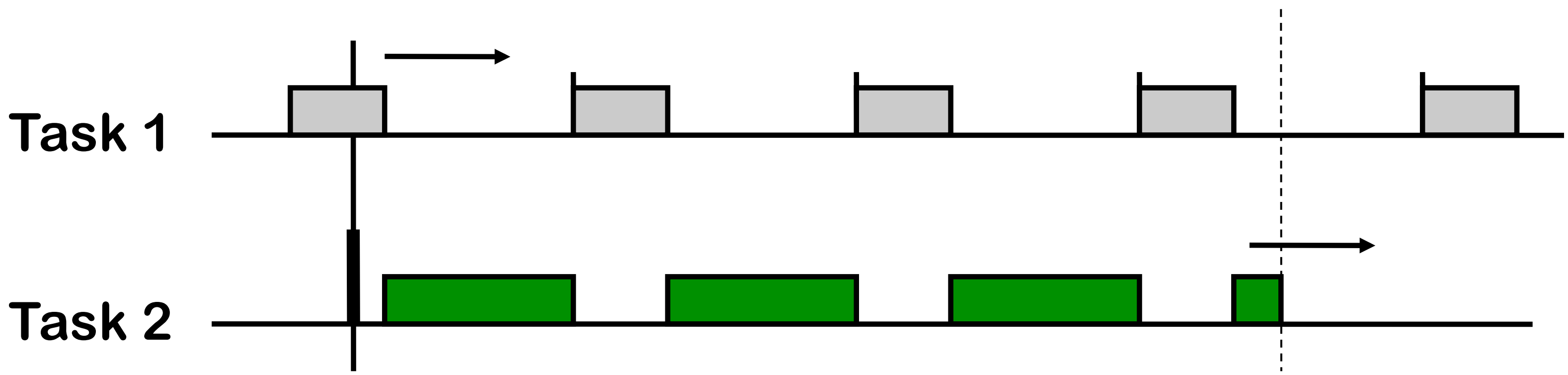
Critical instant theorem

Proof (Case 1)



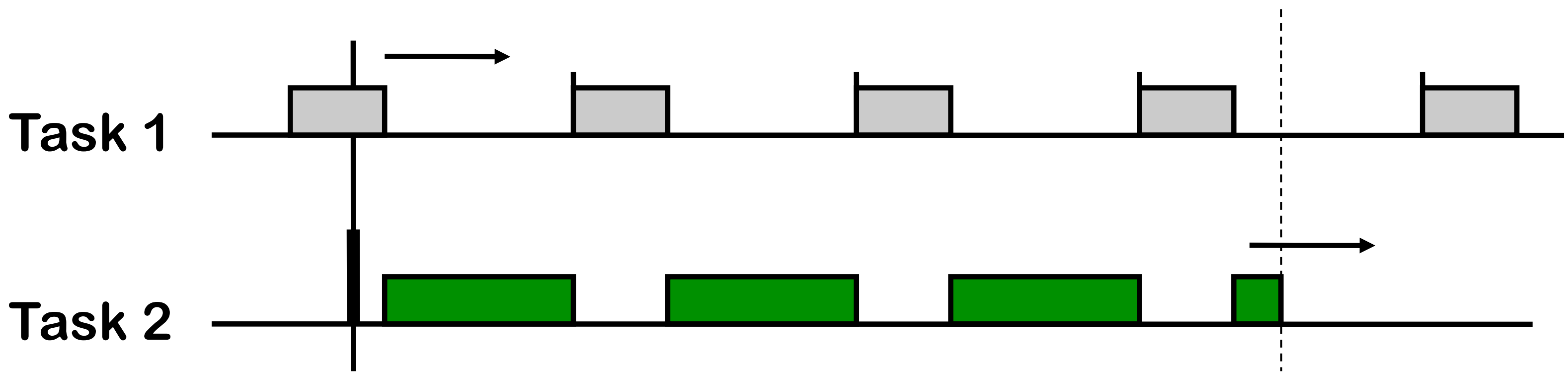
Case 1: Higher priority task 1 is running when task 2 arrives.

Proof

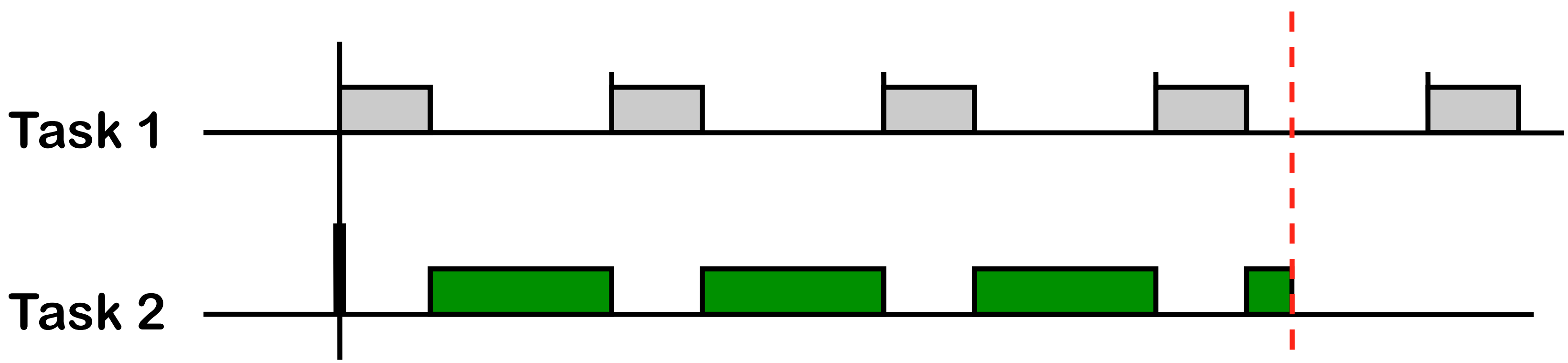


Case 1: higher priority task 1 is running when task 2 arrives
→ shifting task 1 right will increase completion time of 2

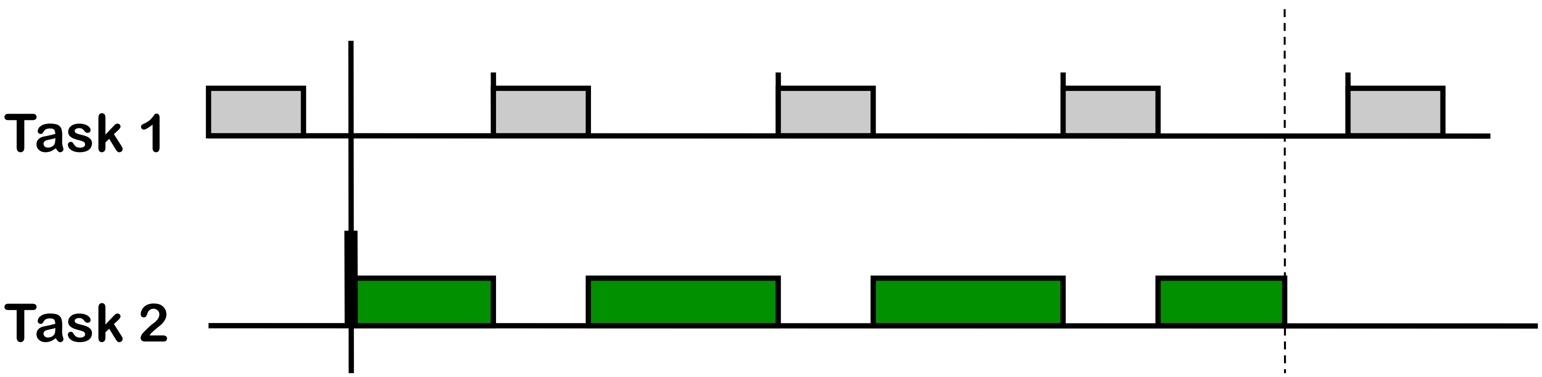
Proof



Case 1: higher priority task 1 is running when task 2 arrives
→ shifting task 1 right will increase completion time of 2

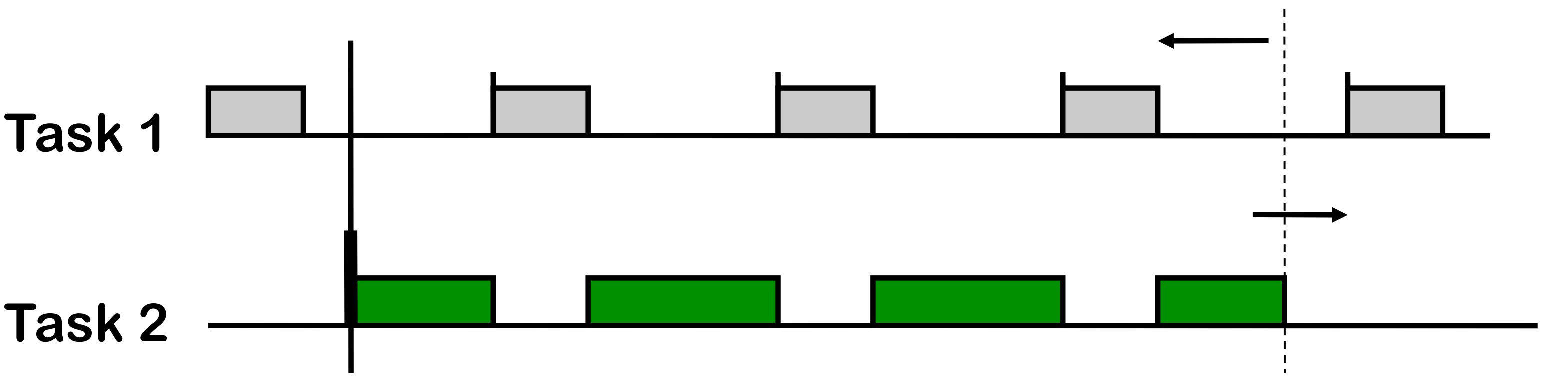


Proof (Case 2)



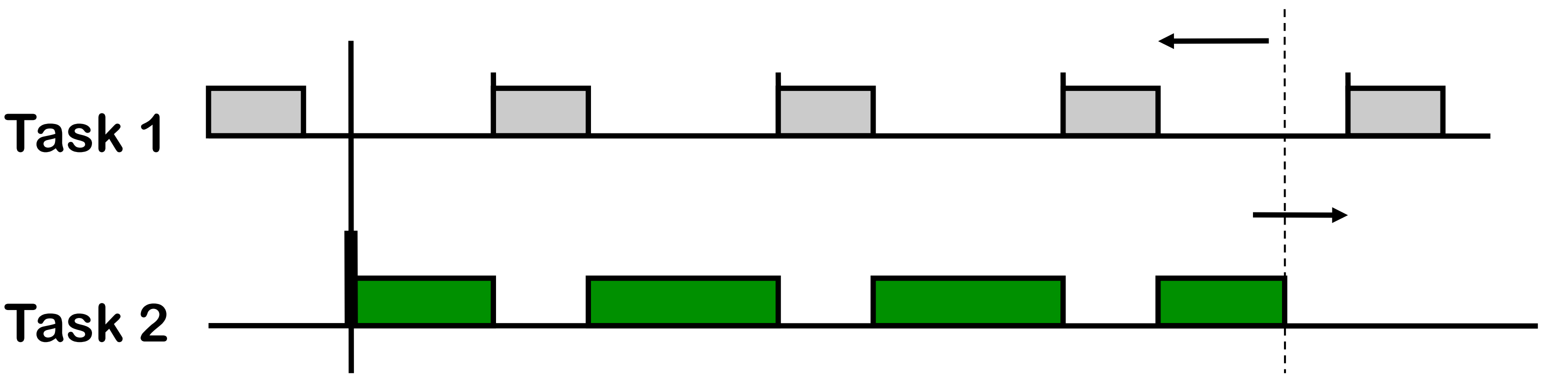
Case 2: processor is idle when task 2 arrives

Proof (Case 2)

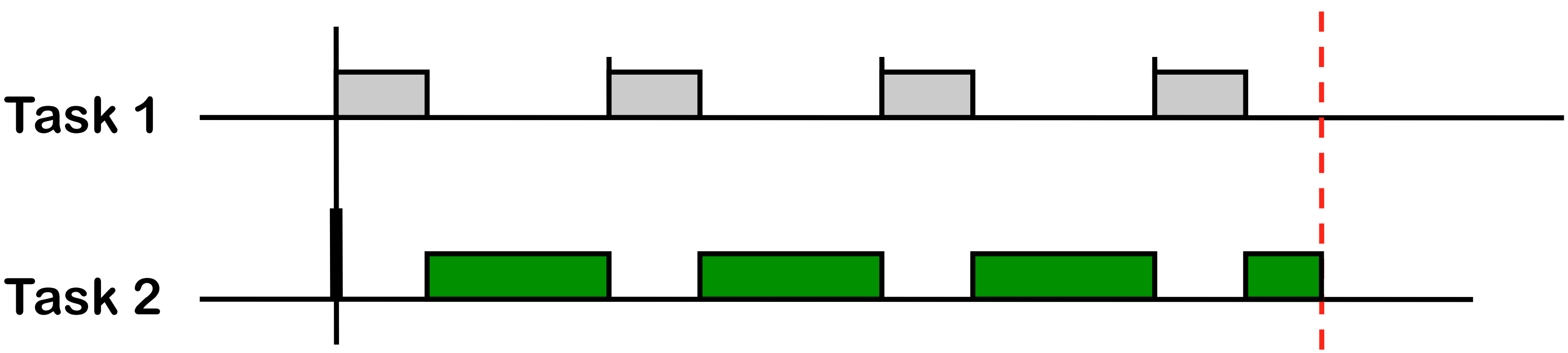


Case 2: processor is idle when task 2 arrives
→ shifting task 1 left cannot decrease completion time of 2

Proof (Case 2)

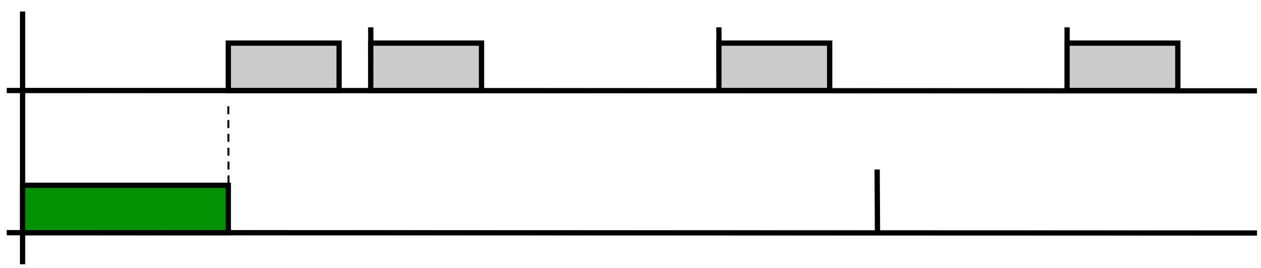


Case 2: processor is idle when task 2 arrives
→ shifting task 1 left cannot decrease completion time of task 2



Optimality of the RM policy

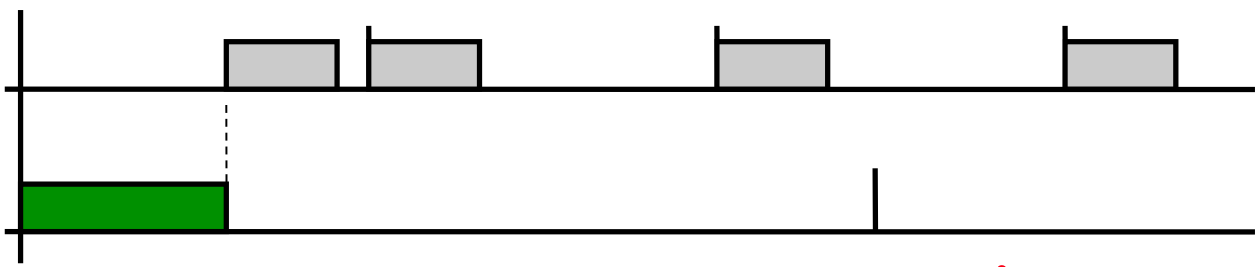
- If any other fixed-priority policy can meet deadlines so can RM



Policy X meets deadlines?

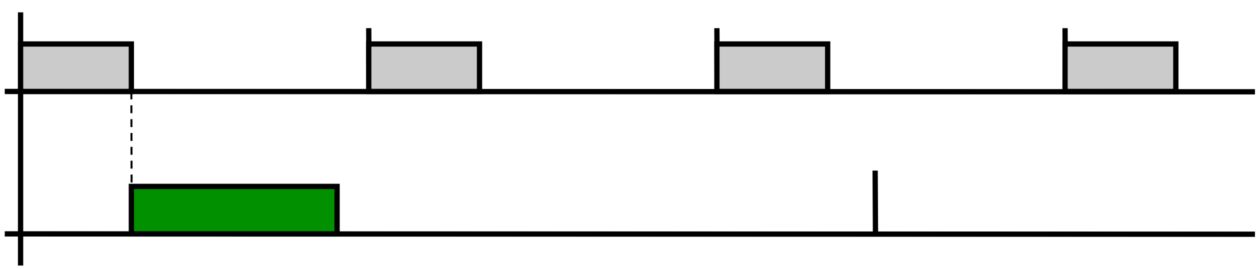
Optimality of the RM policy

- If any other policy can meet deadlines so can RM

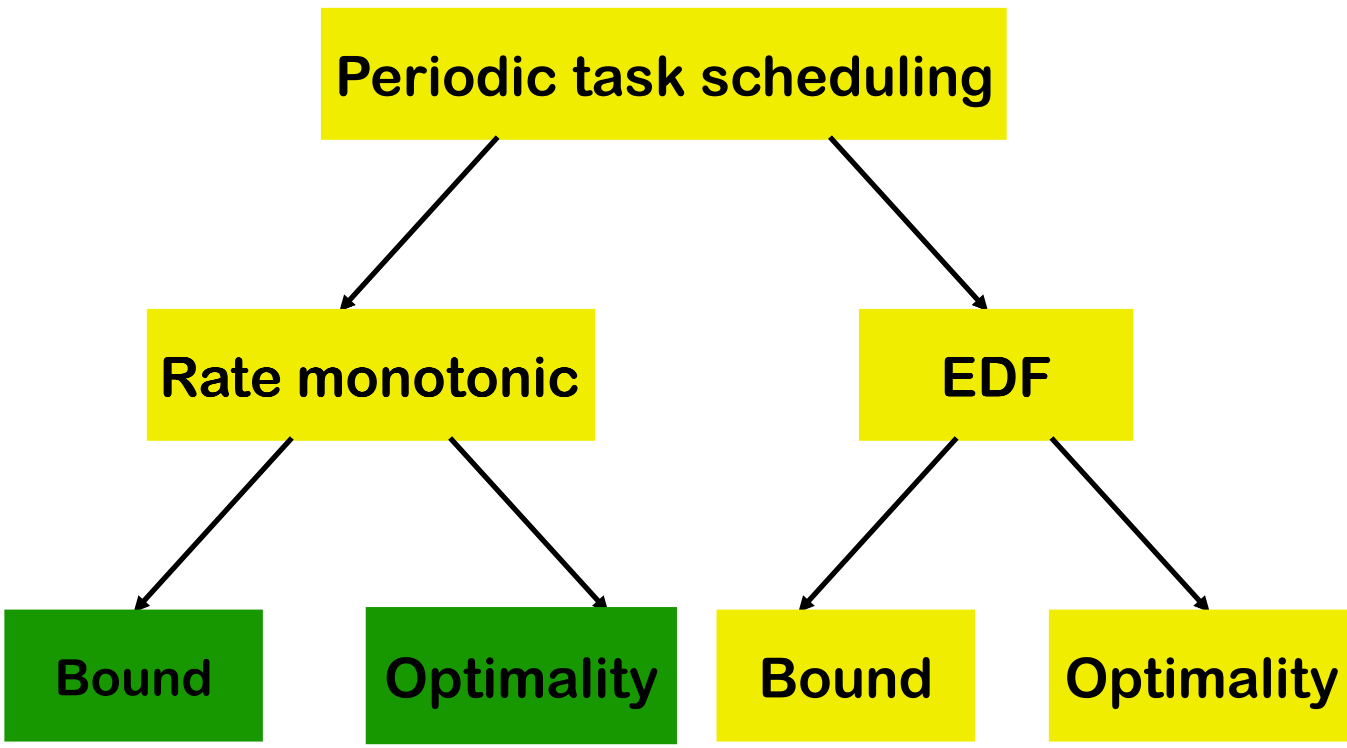


Policy X meets deadlines? **YES**

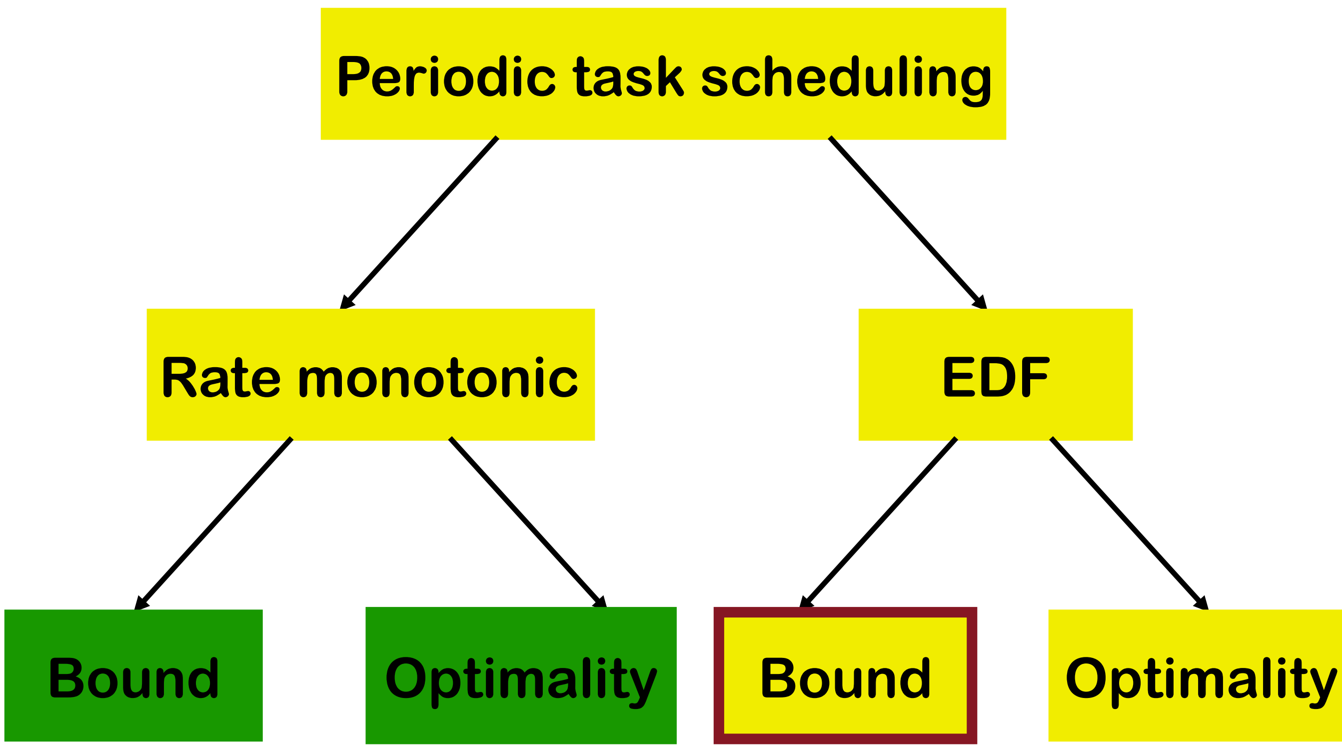
→ RM meets deadlines



Lecture outline



Lecture outline

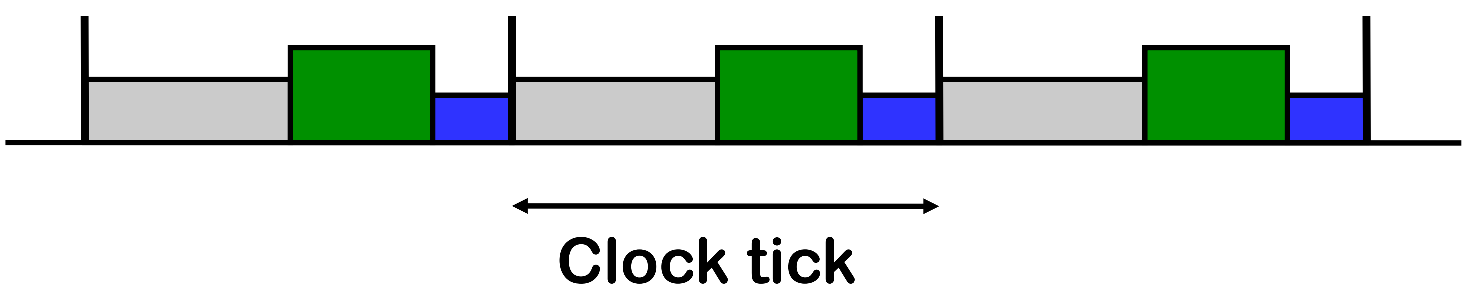


Utilization bound for EDF

- **Why is it 100%?**
- **Consider a task set where:**

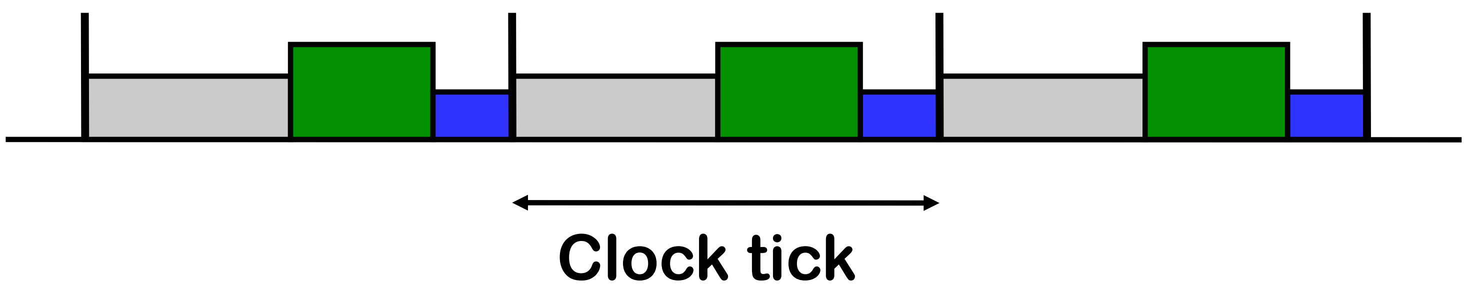
$$\sum_i \frac{C_i}{P_i} = 1$$

- **Imagine a policy that reserves for each task *i* a fraction f_i of each clock tick, where $f_i = C_i/P_i$**



Utilization bound for EDF

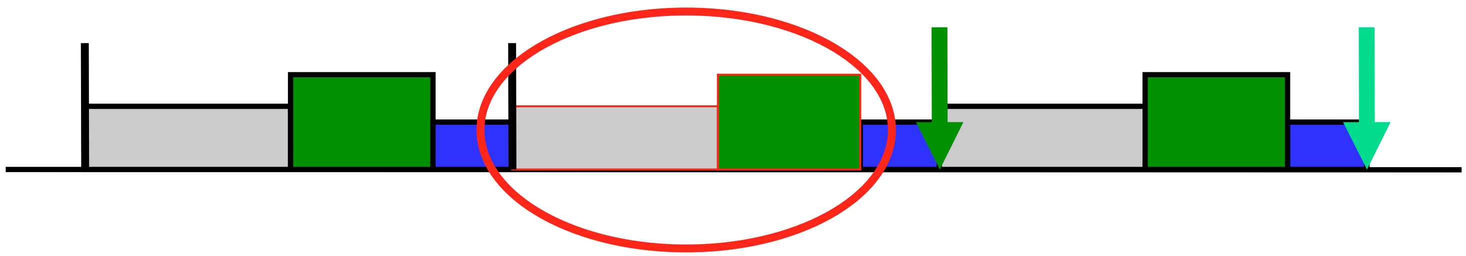
- Imagine a policy that reserves for each task *i* a fraction f_i of each time unit, where $f_i = C_i / P_i$



- This policy meets all deadlines, because within each period P_i it reserves for task *i* a total time
  - Time = $f_i \, P_i = (C_i / P_i) \, P_i = C_i$ (i.e., enough to finish)

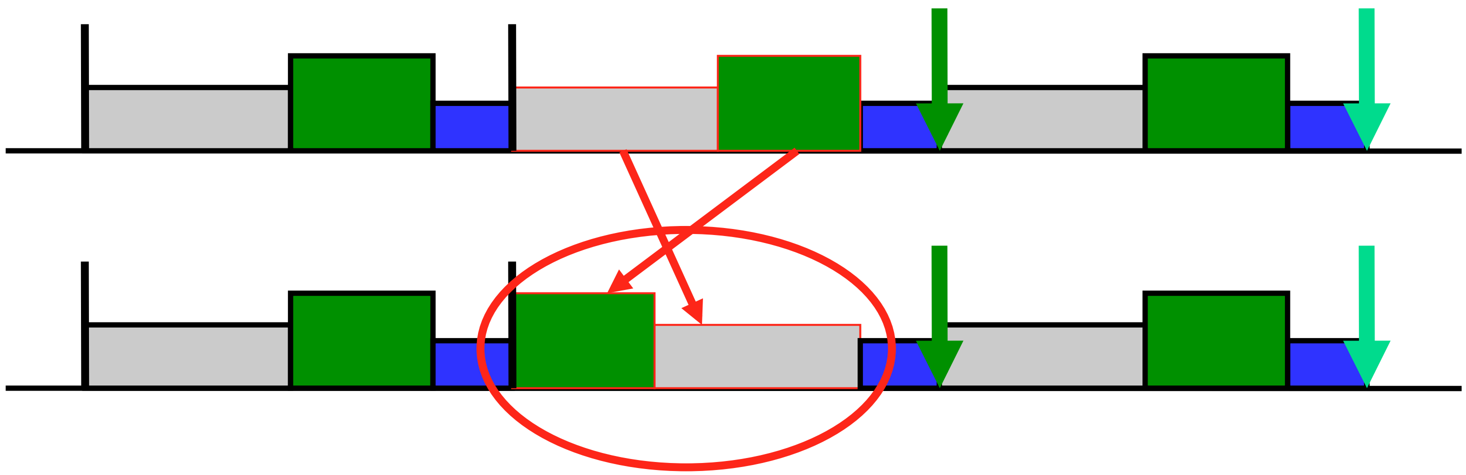
Utilization bound for EDF

- Pick any two execution chunks that are not in EDF order and swap them



Utilization bound for EDF

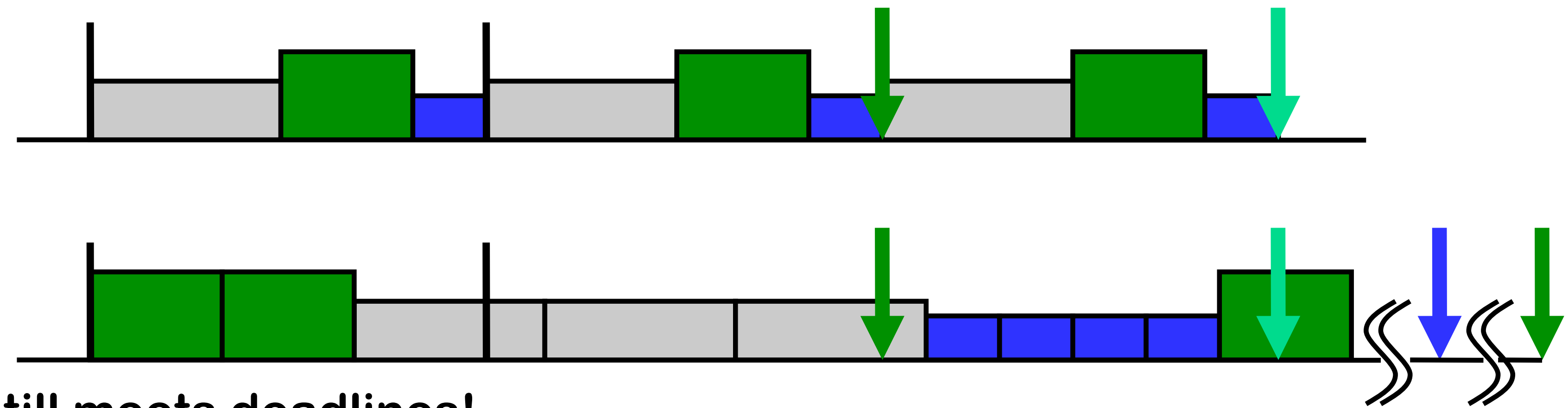
- Pick any two execution chunks that are not in EDF order and swap them



- Still meets deadlines!

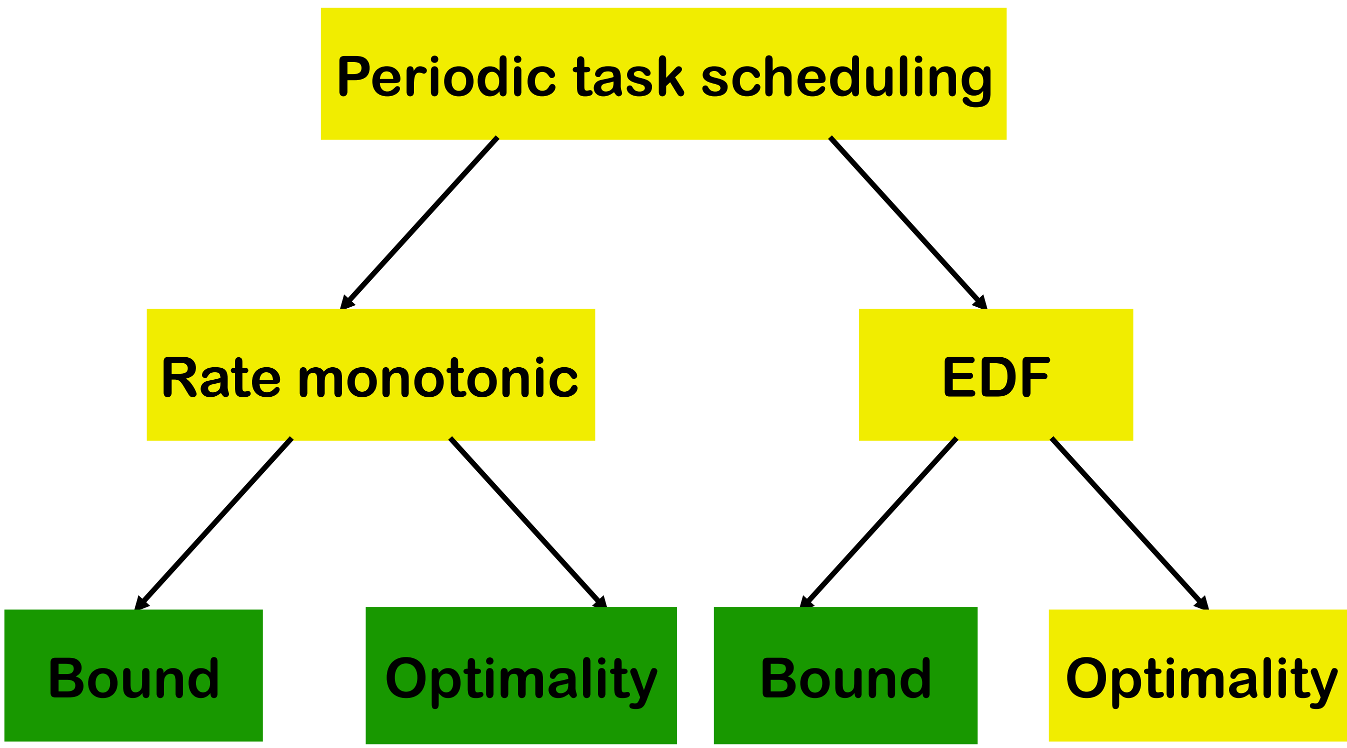
Utilization bound for EDF

- Pick any two execution chunks that are not in EDF order and swap them

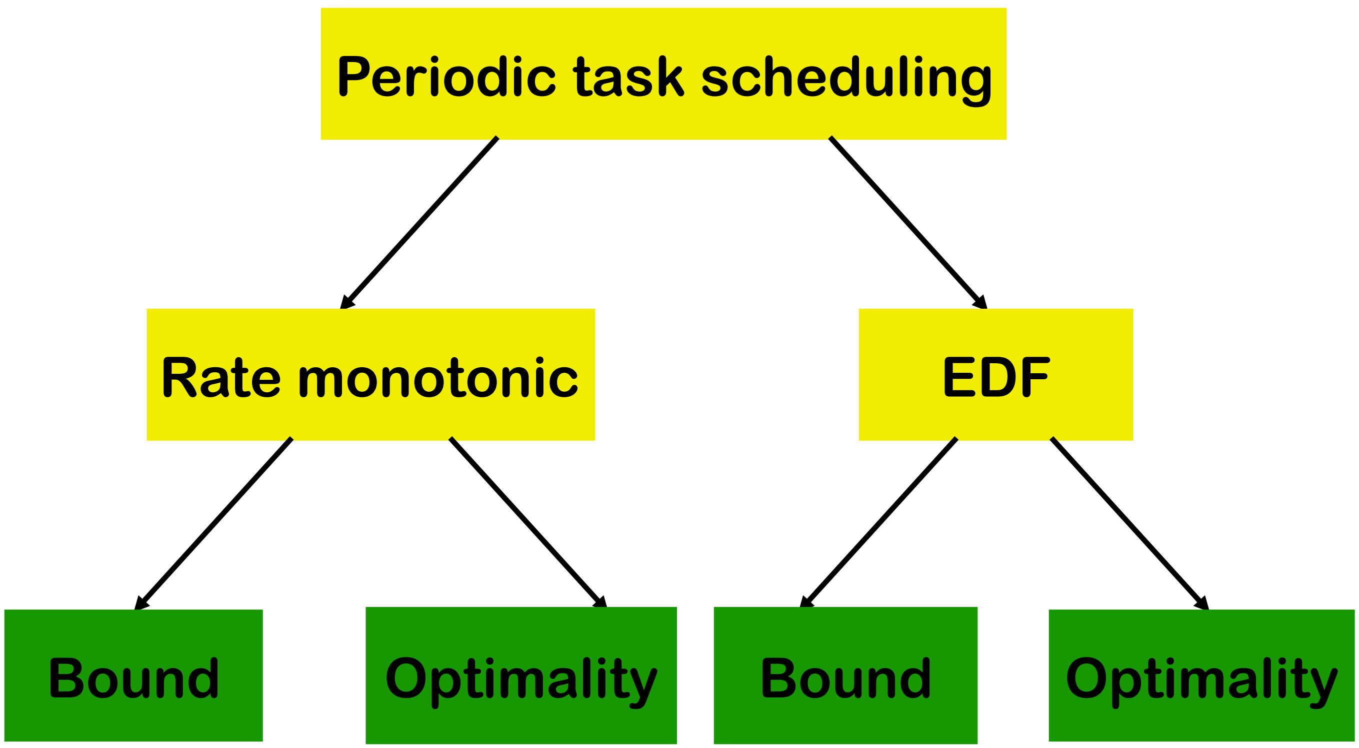


- Still meets deadlines!
- Repeat swap until all in EDF order
  - → EDF meets deadlines

Lecture outline

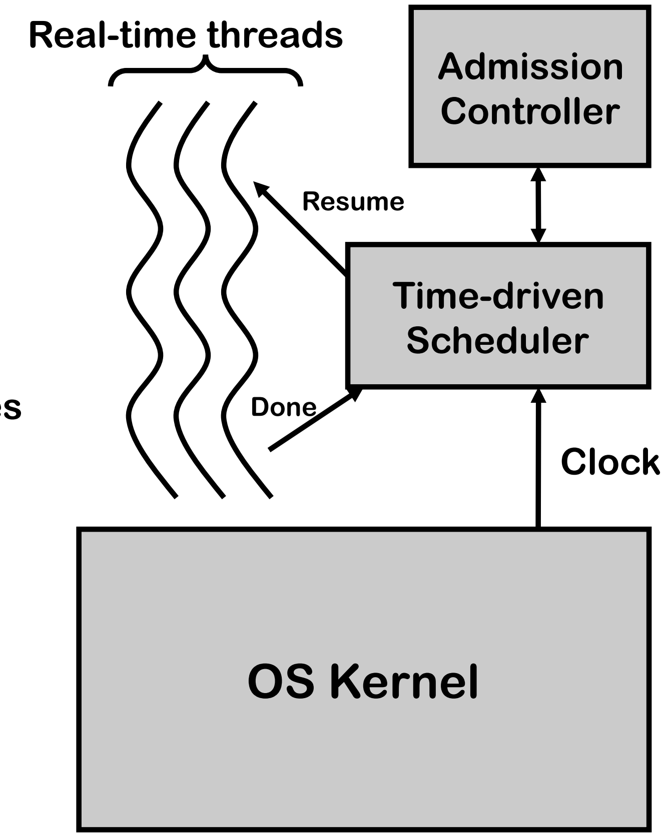


Lecture outline



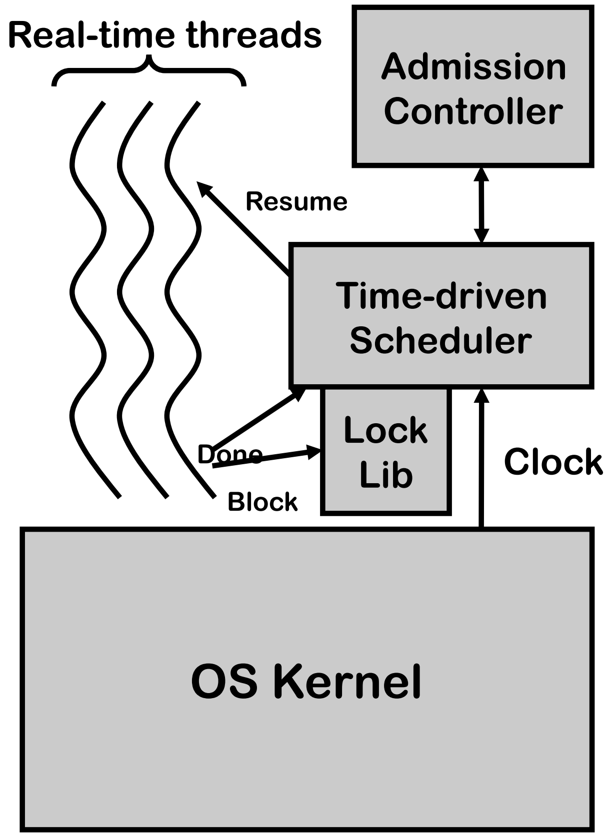
Tick-based scheduling within an OS

- A real-time library for periodic tasks on Linux or Windows
  - There is need to provide approximate real-time guarantees on common operating systems (as opposed to specialized real-time OSes)
  - A high-priority "real-time" thread pool is created and maintained
  - A higher-priority scheduler is invoked periodically by timer-ticks to check for periodic invocation times of real-time threads. The scheduler resumes threads whose arrival times have come.
  - Resumed threads execute one invocation then block.
  - Scheduling is preemptive
  - The scheduler can implement arbitrary scheduling policies including EDF, RM, etc.
  - An admission controller is responsible for spawning new periodic threads if the new task set can meet its deadlines.



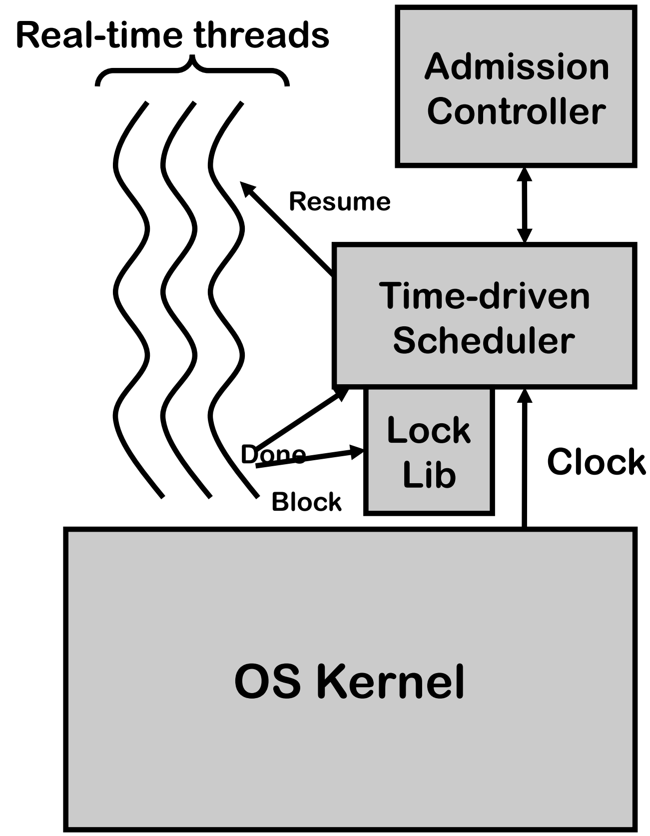
Tick-based scheduling within an OS

- A real-time library for periodic tasks on Linux or Windows
  - There is need to provide approximate real-time guarantees on common operating systems (as opposed to specialized real-time OSes)
  - A high-priority “real-time” thread pool is created and maintained
  - A higher-priority scheduler is invoked periodically by timer-ticks to check for periodic invocation times of real-time threads. The scheduler resumes threads whose arrival times have come.
  - Resumed threads execute one invocation then block.
  - Scheduling is preemptive
  - The scheduler can implement arbitrary scheduling policies including EDF, RM, etc.
  - An admission controller is responsible for spawning new periodic threads if the new task set can meet its deadlines.
  - Scheduler implements wrappers for blocking primitives

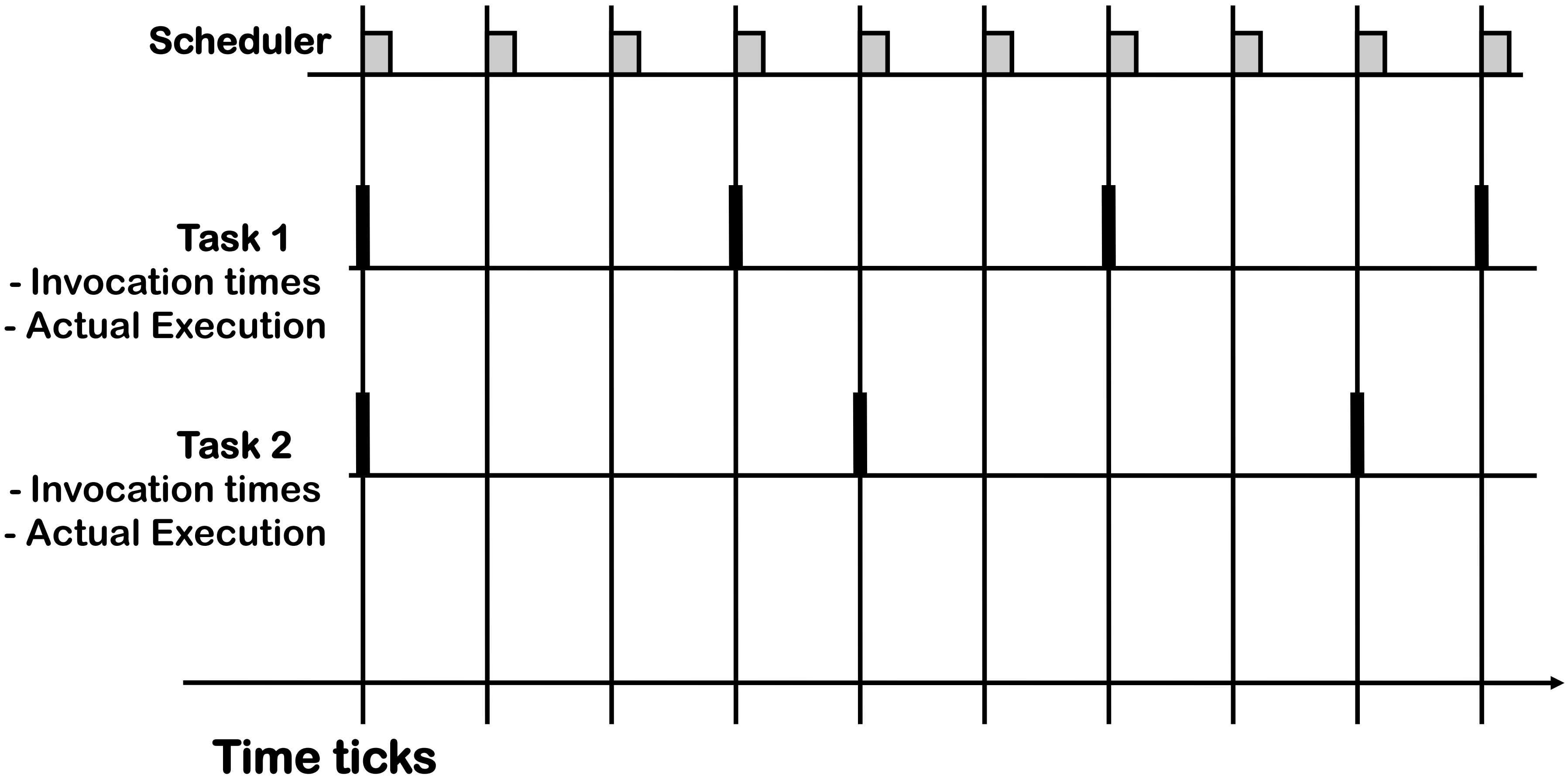


The time-driven scheduler

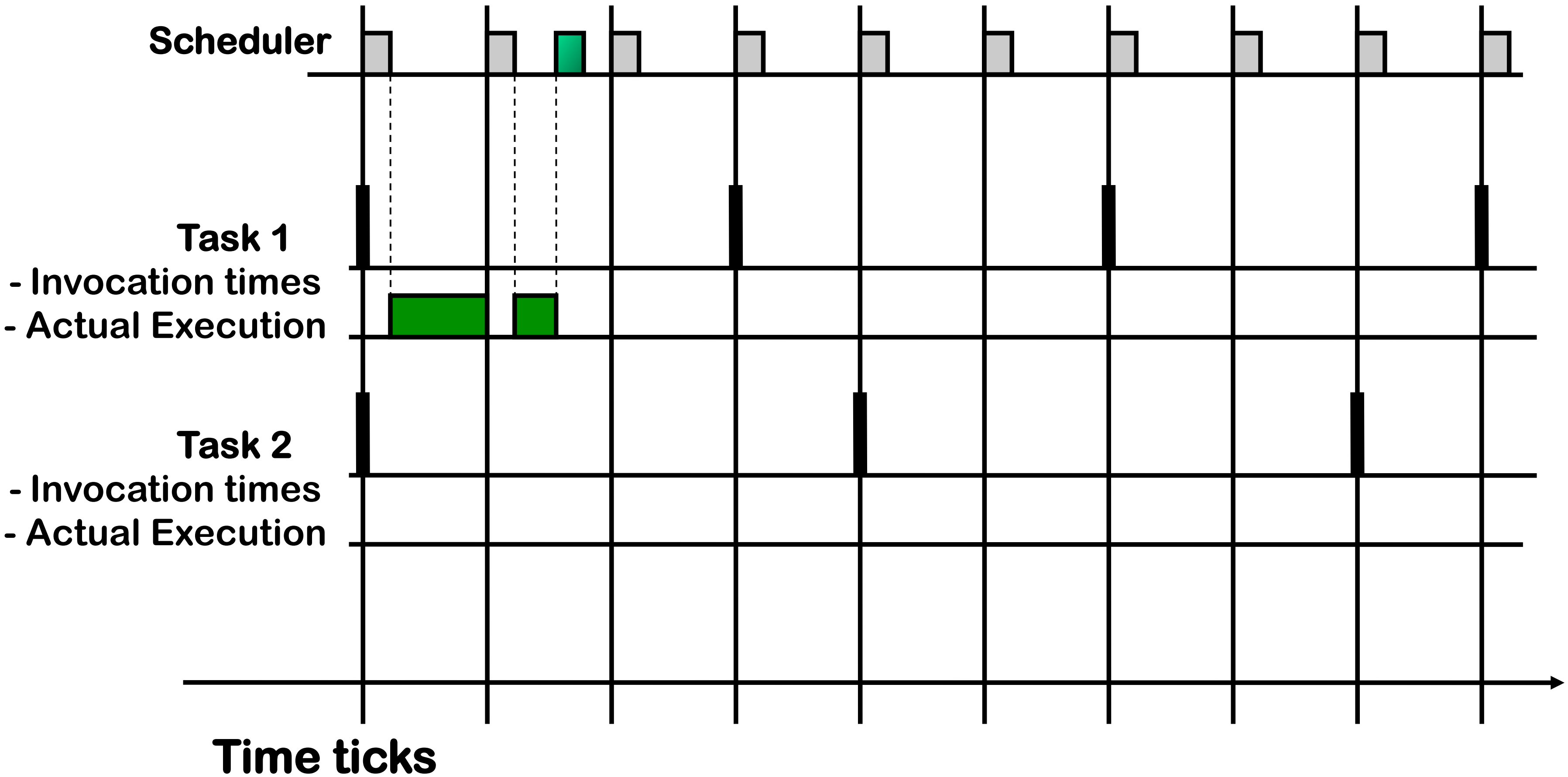
- /* N is the number of periodic tasks */
- For i=1 to N
-   if (current_time = next_arrival_time of task i)
-     put task i in ready_queue
- /* ready_queue is a priority queue that implements
-   the desired scheduling policy. */
- Inspect top task from ready queue, call it j
- If (a task is running and its priority is higher than priority of j) return
- Else resume task j (and put the running task into the ready queue if applicable); return



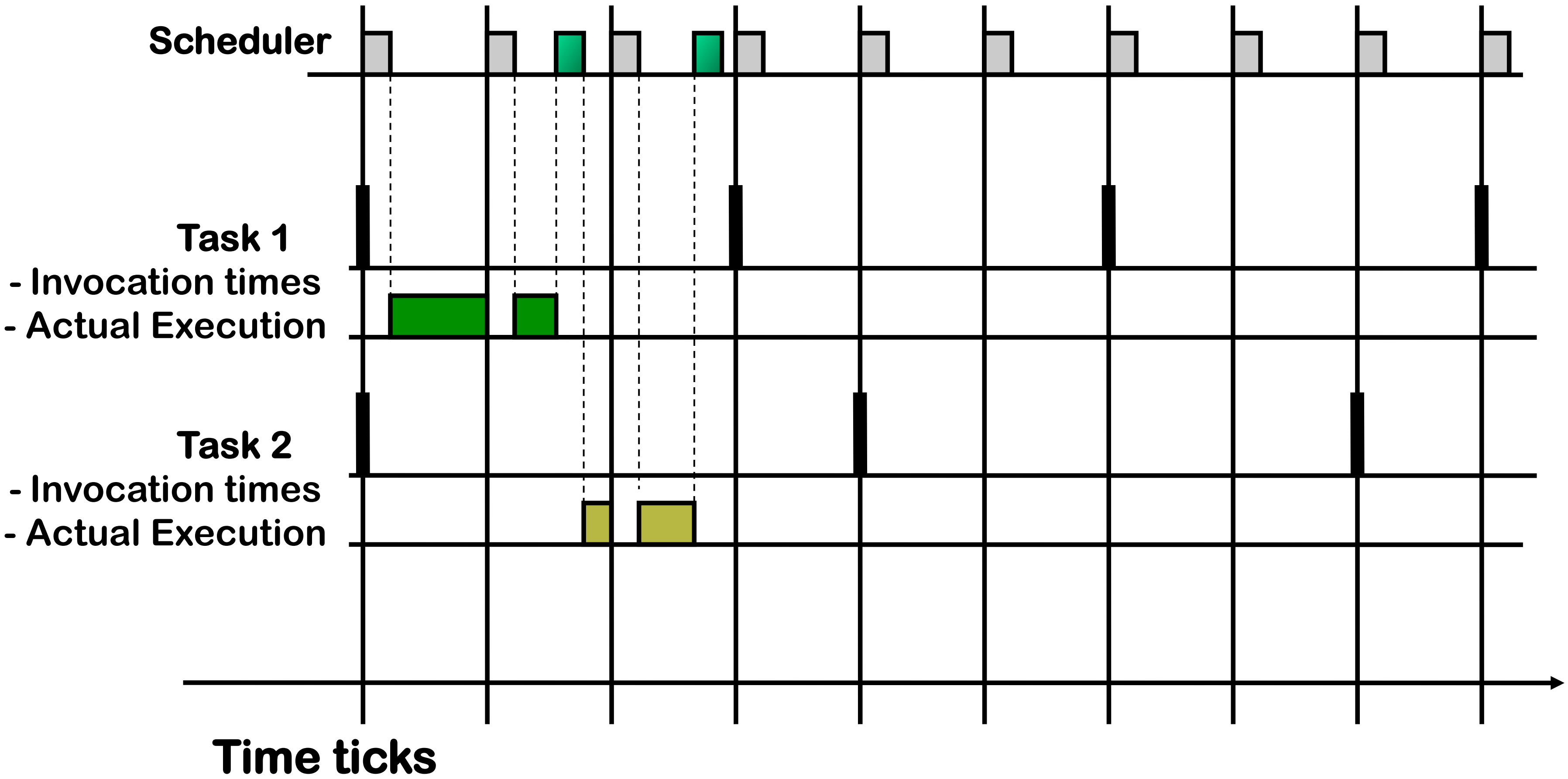
An example schedule



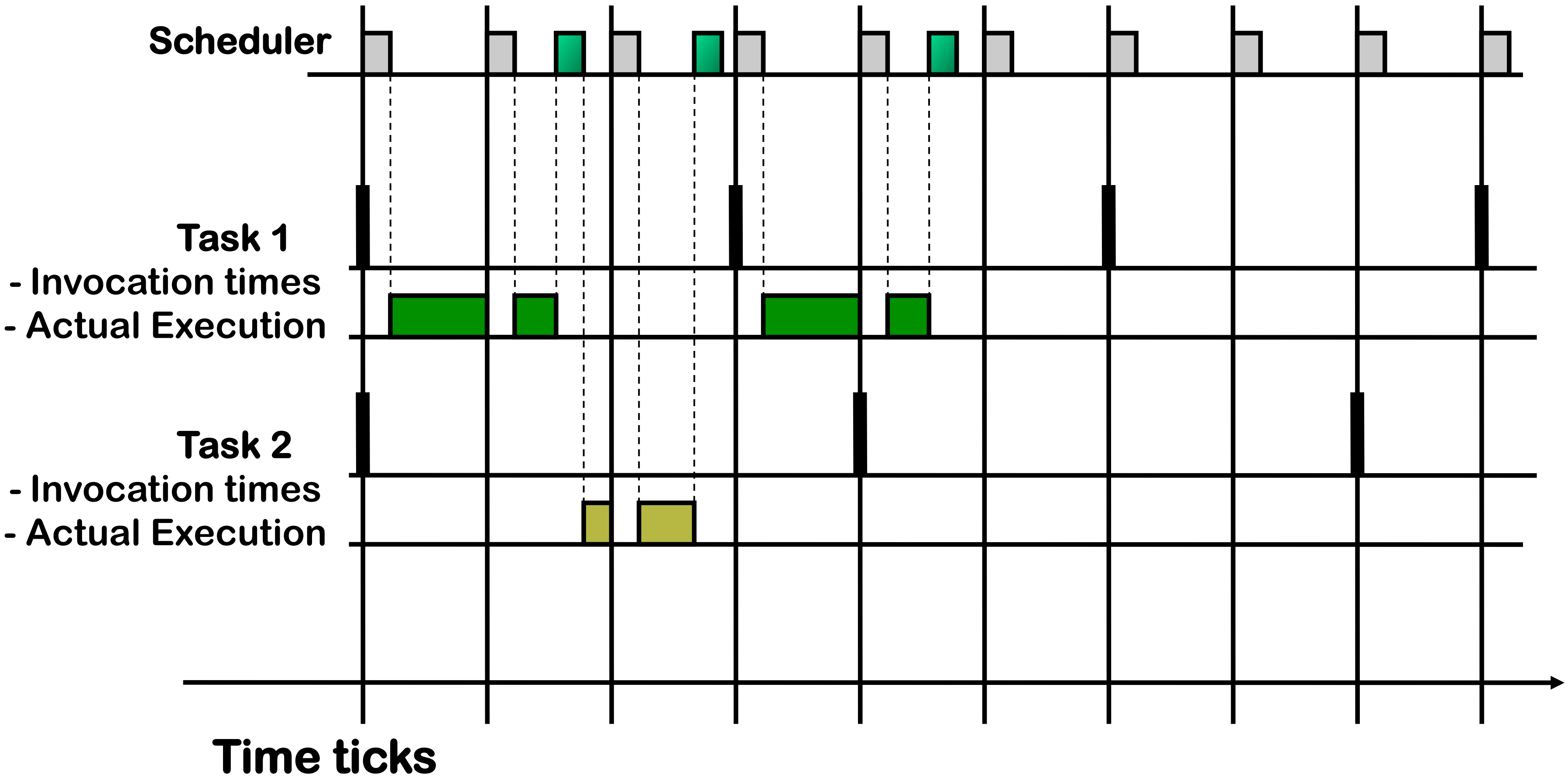
An example schedule



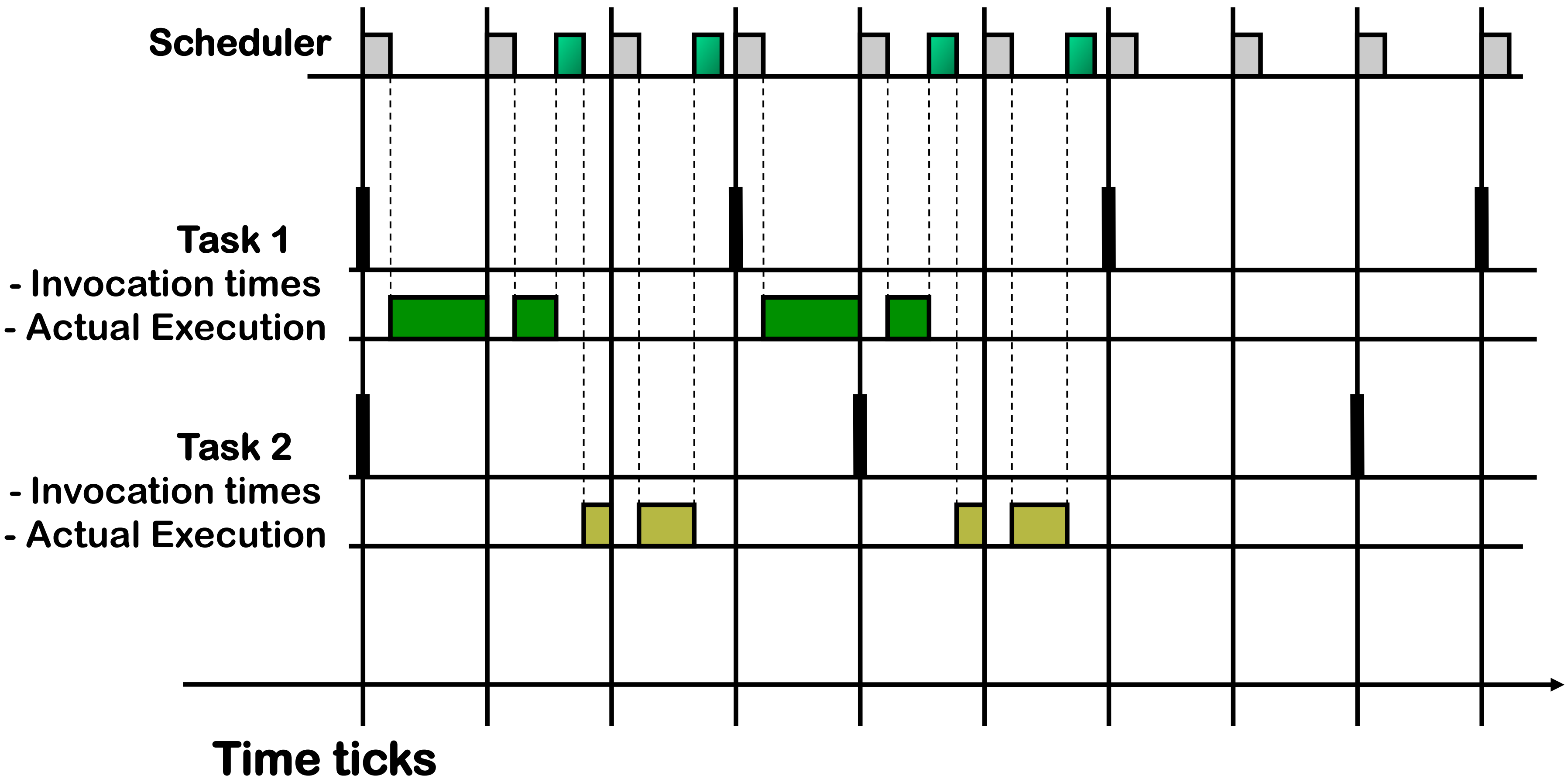
An example schedule



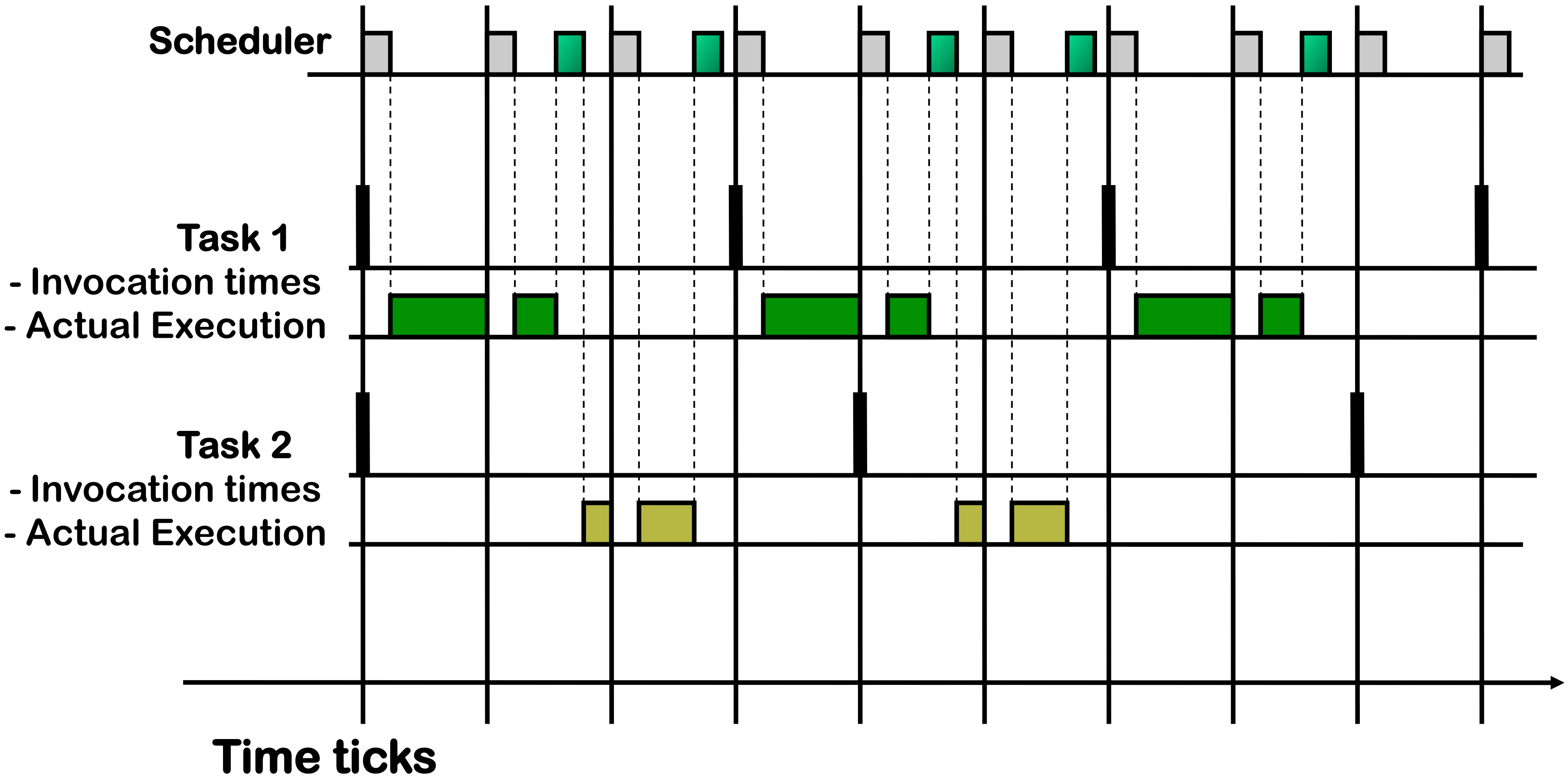
An example schedule



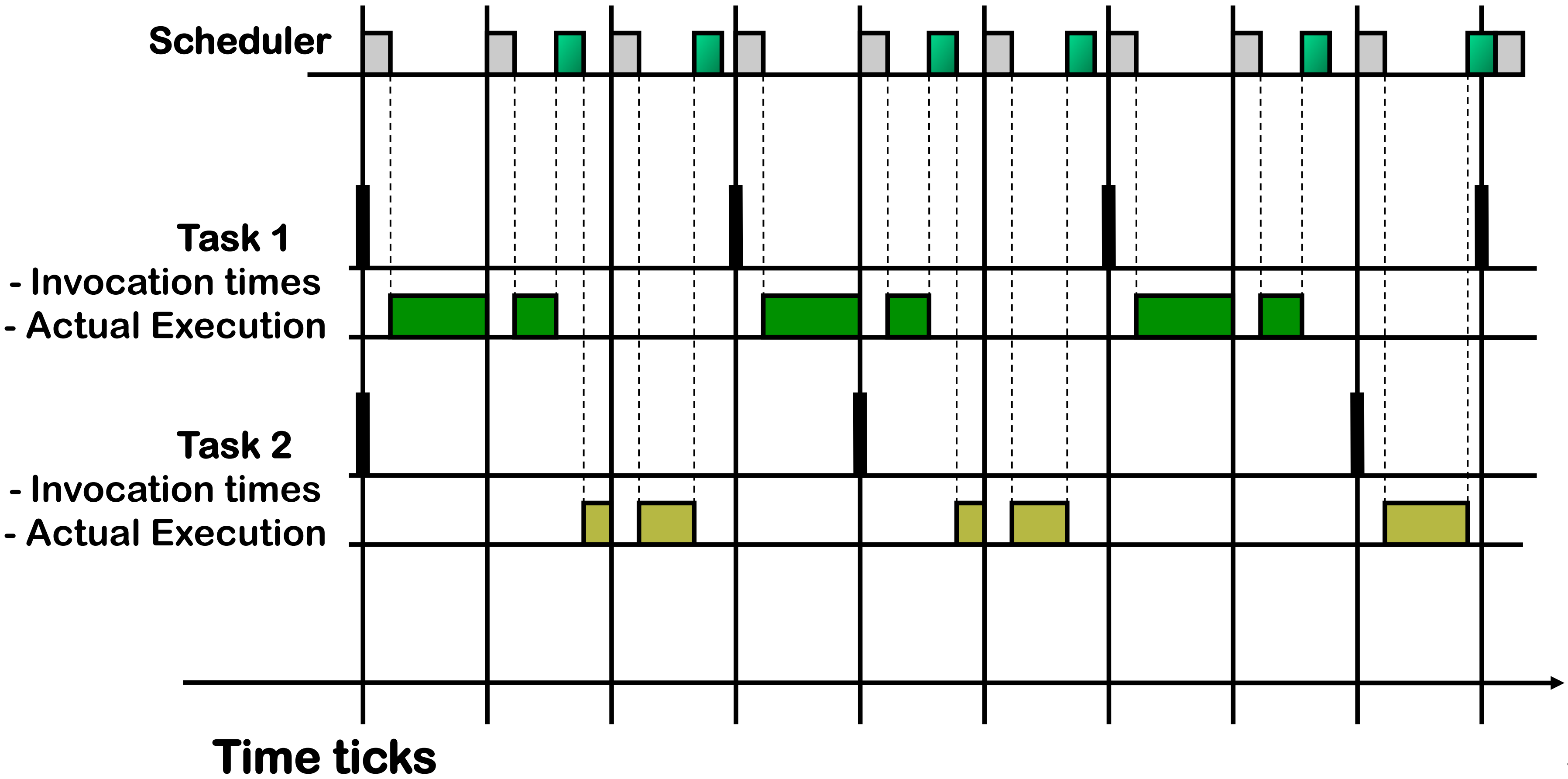
An example schedule



An example schedule

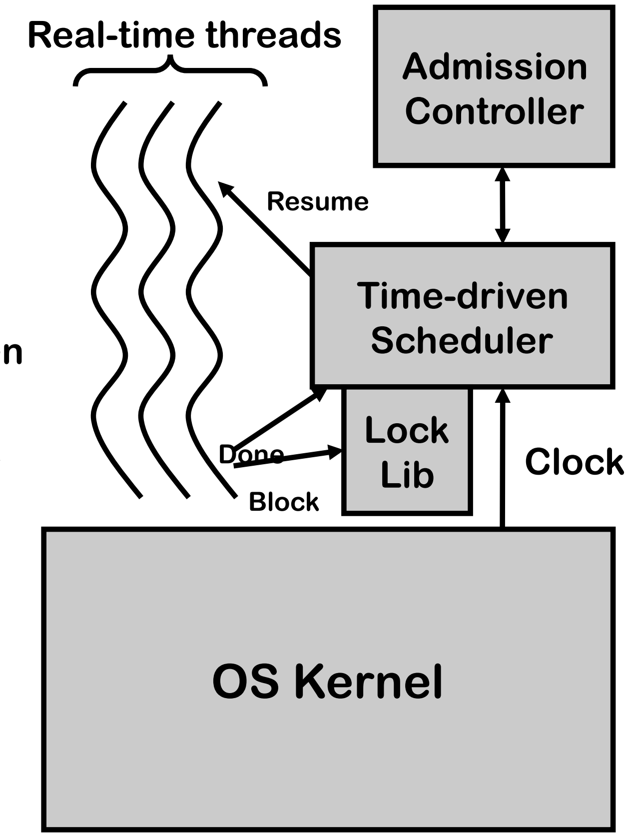


An example schedule



Admission controller

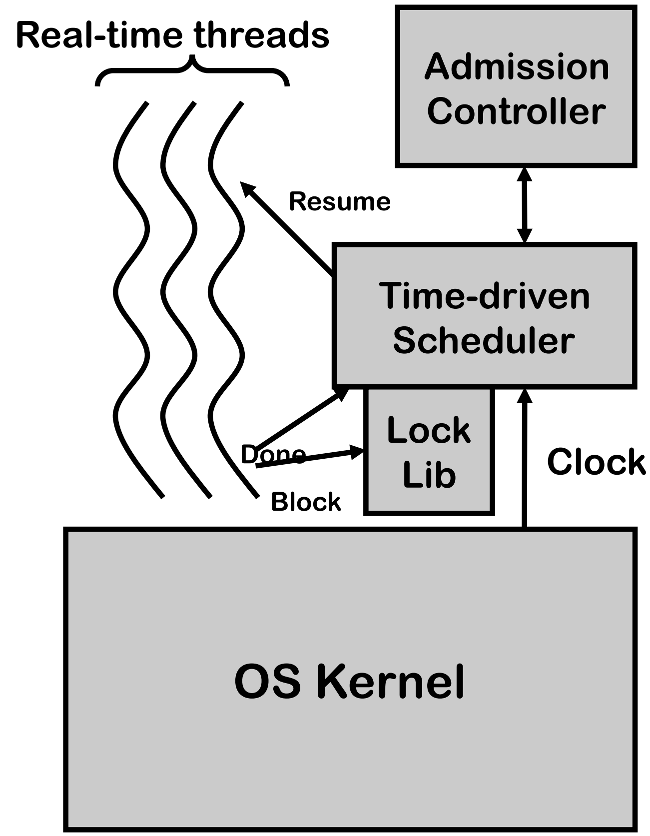
- Implements schedulability analysis
  - If $U+C_{new}/P_{new} < U_{bound}$ admit task
  - Must account for various practical overheads. How?
  - Examples of overhead:
    - How to account for the overhead of running the time-driven scheduler on every time-tick?
    - How to account for the overhead of running the scheduler after task termination?
- If new task admitted
  - $U = U + C_{new}/P_{new}$
  - Create a new thread
  - Register it with the scheduler



Library with lock primitives

- Lock (S) {
-   Check if semaphore S = locked
-   If locked
-     enqueue running tasks in semaphore queue
-   Else
-     let semaphore = locked
- }

- Unlock (S) {
-   If semaphore queue empty then
-     semaphore = unlocked
-   Else
-     Resume highest-priority waiting task
- }



Problem: some threads may execute blocking OS calls (e.g., disk or network read/write and block without calling your lock/unlock!)