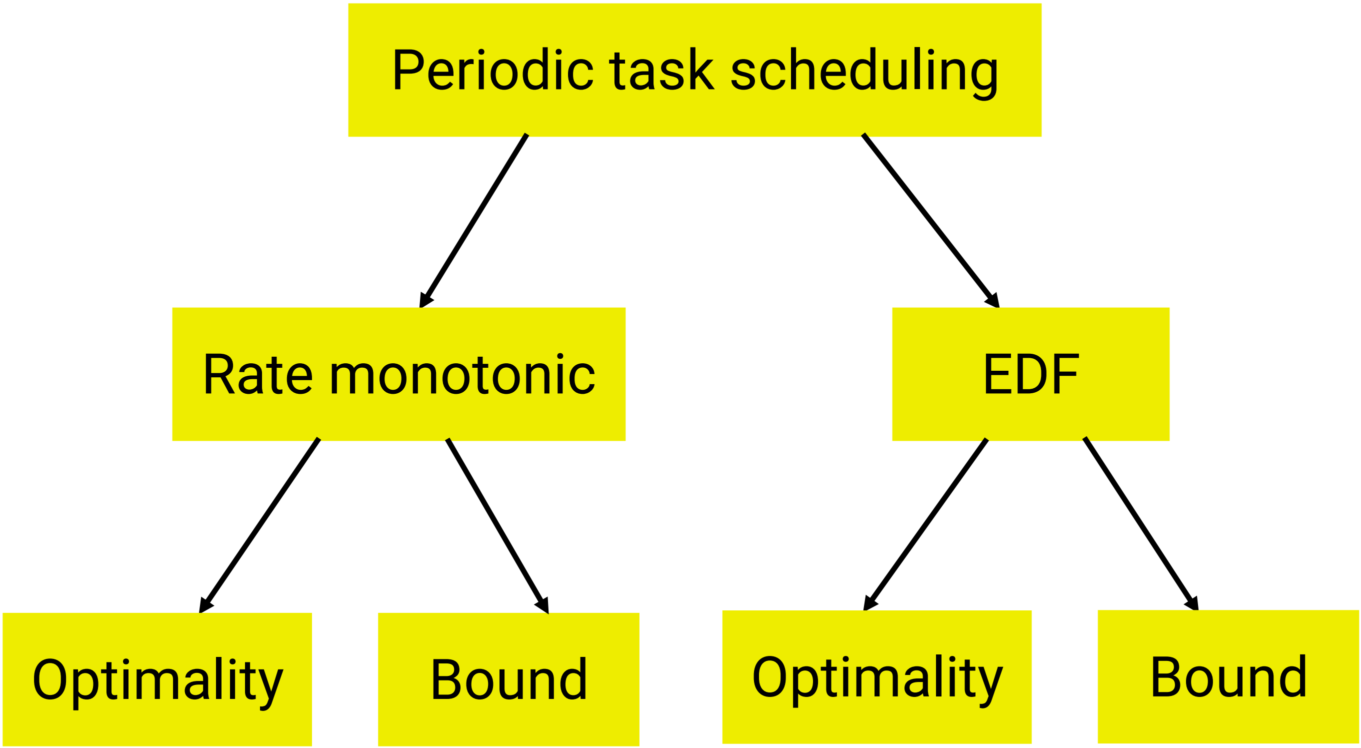
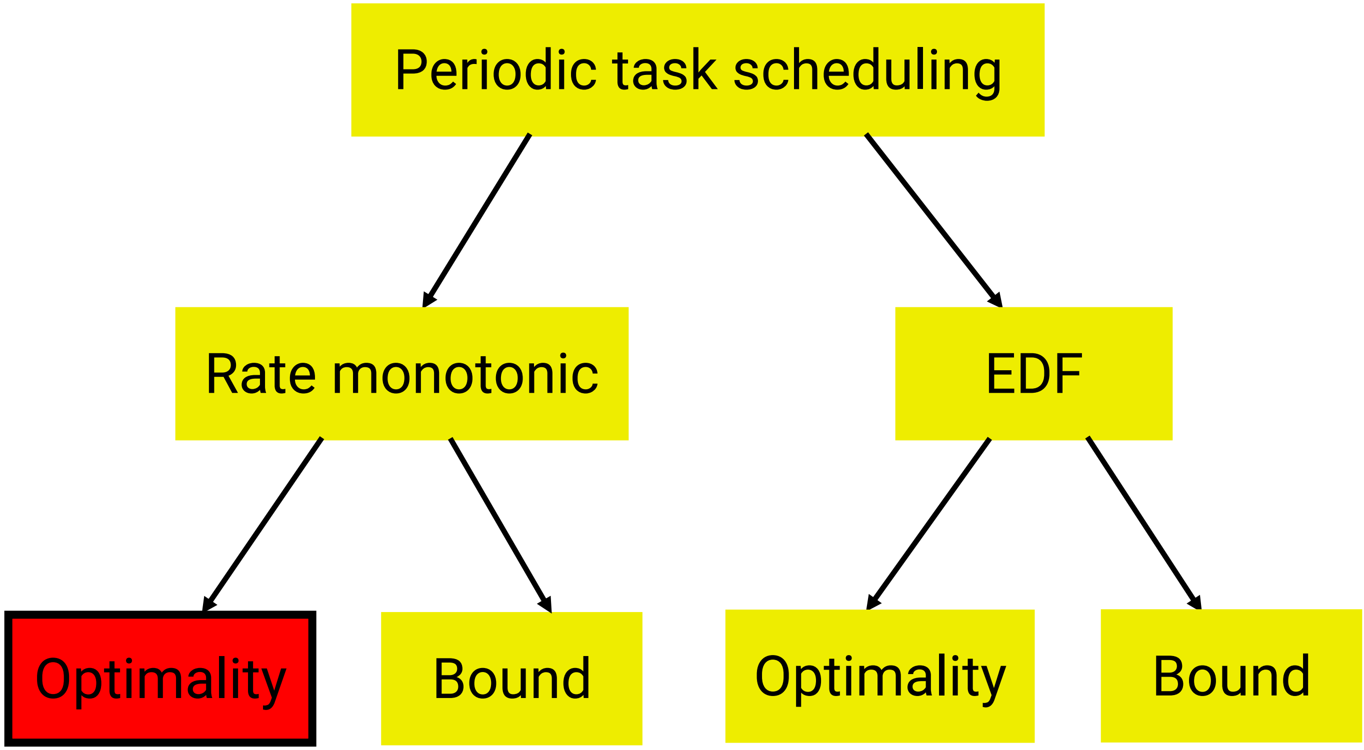
Periodic task scheduling

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)

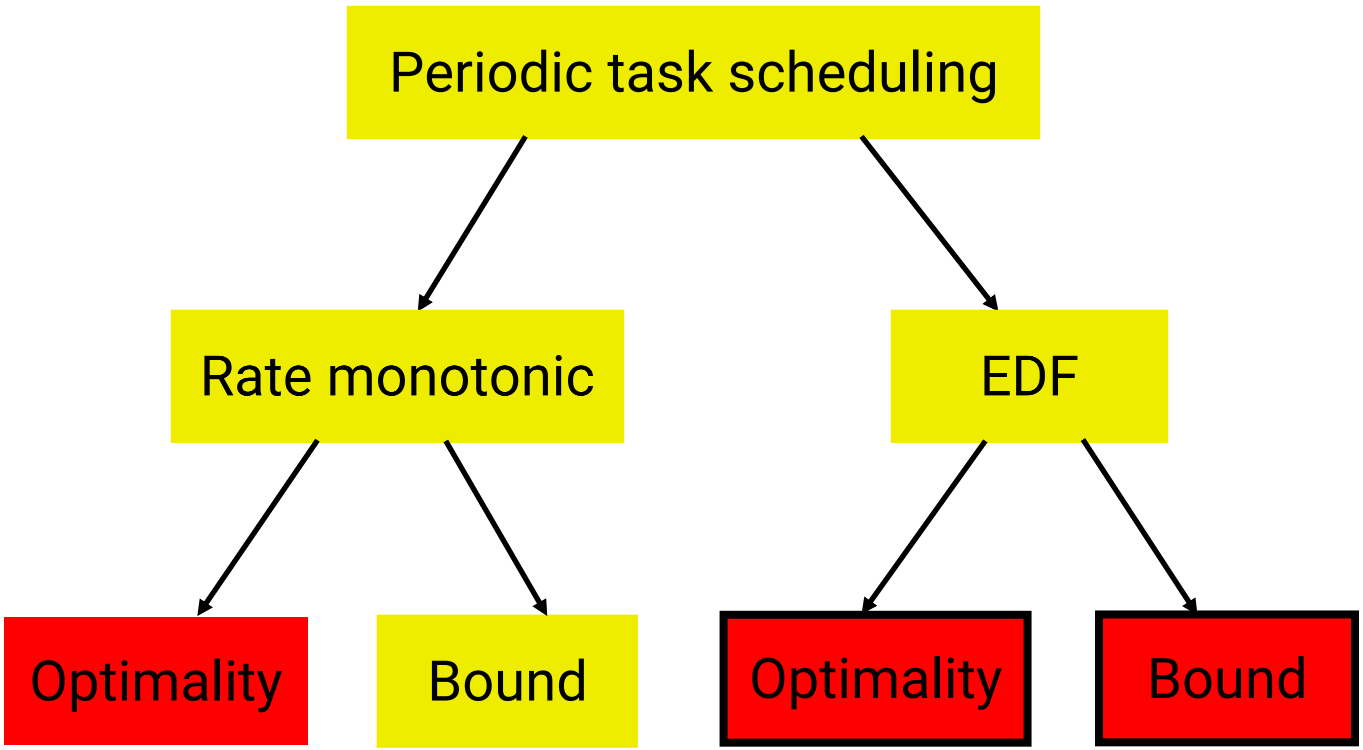
Lecture outline



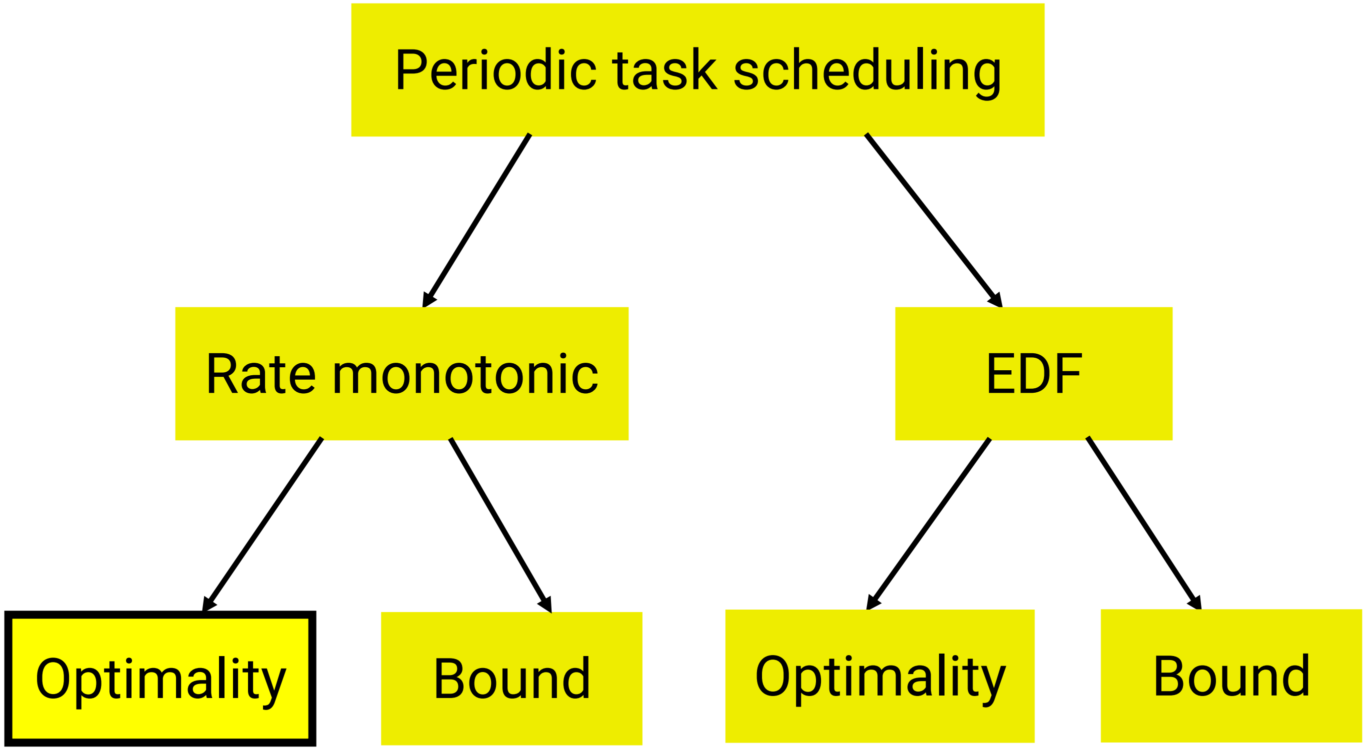
Lecture outline



Lecture outline



Next


Periodic task scheduling
Rate monotonic
EDF
Optimality
Bound
Optimality
Bound

Rate monotonic scheduling

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

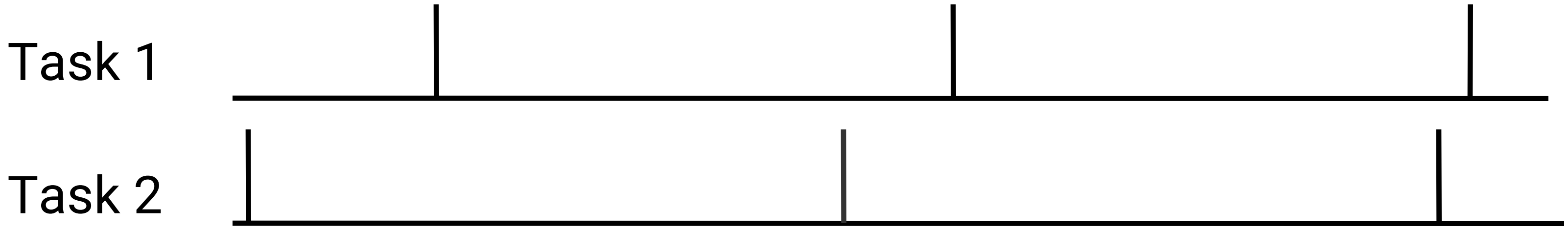
Rate monotonic scheduling

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

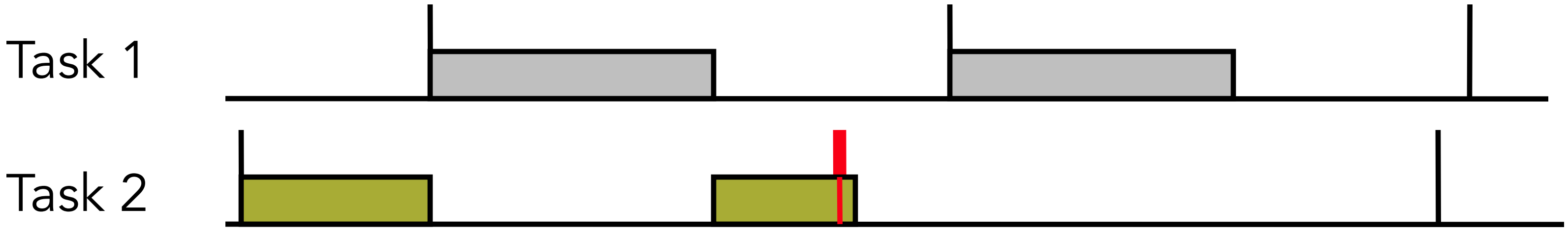
$$P_1 < P_2$$



Rate monotonic scheduling

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

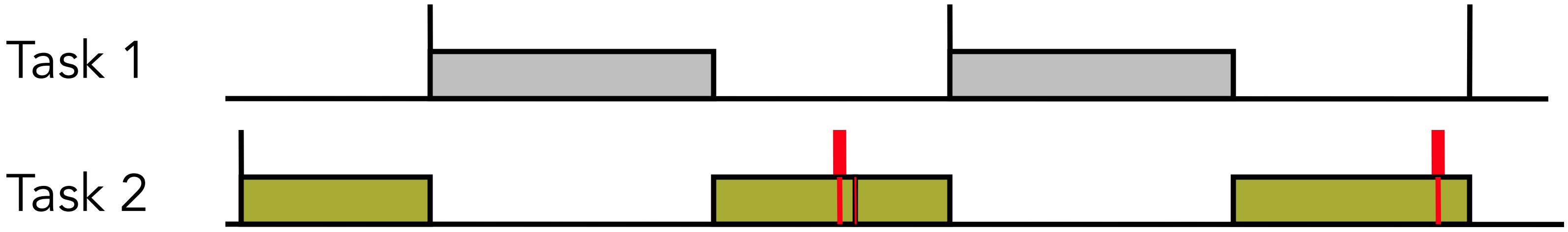
$$P_1 < P_2$$



Rate monotonic scheduling

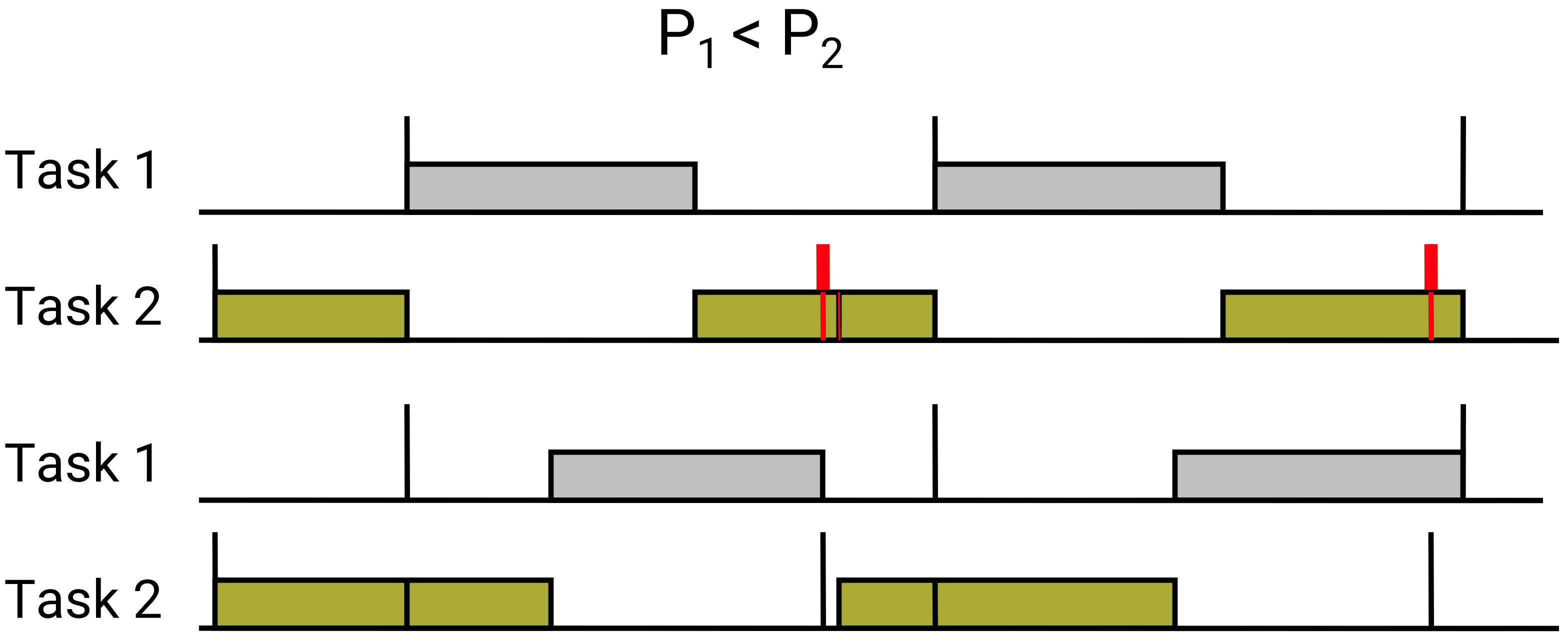
- Rate monotonic scheduling is an optimal fixed-priority (or static-priority) scheduling policy for periodic tasks.
  - Optimality (Trial #1): If any other fixed-priority scheduling policy can meet deadlines, so can RM

$$P_1 < P_2$$



Rate monotonic scheduling

- Rate monotonic scheduling is an optimal fixed-priority (or static-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 an optimal fixed-priority (or static-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

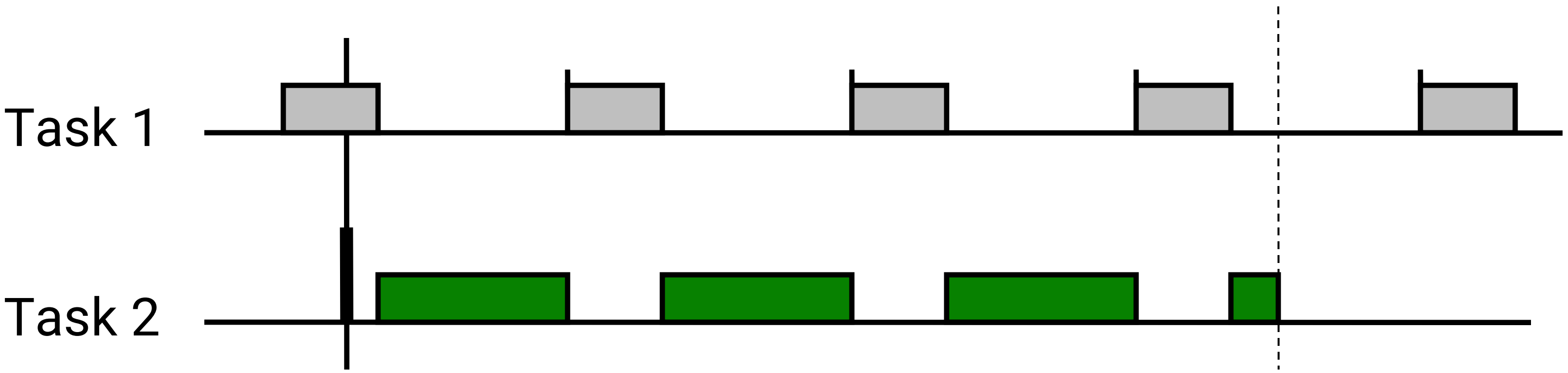
- Rate monotonic scheduling is the optimal fixed-priority (or static-priority) scheduling policy for periodic tasks.
  - Optimality (Trial #2): If any other fixed-priority scheduling policy can meet deadlines in the worst-case scenario, so can RM.
- How do we prove it?
  - Consider the worst-case scenario
  - 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?
  - Which arrival time produces the largest <u>response time</u> for *T*?
- **A:** When it arrives together with all the higher-priority tasks (critical instant)
  - Liu and Layland
- 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*.

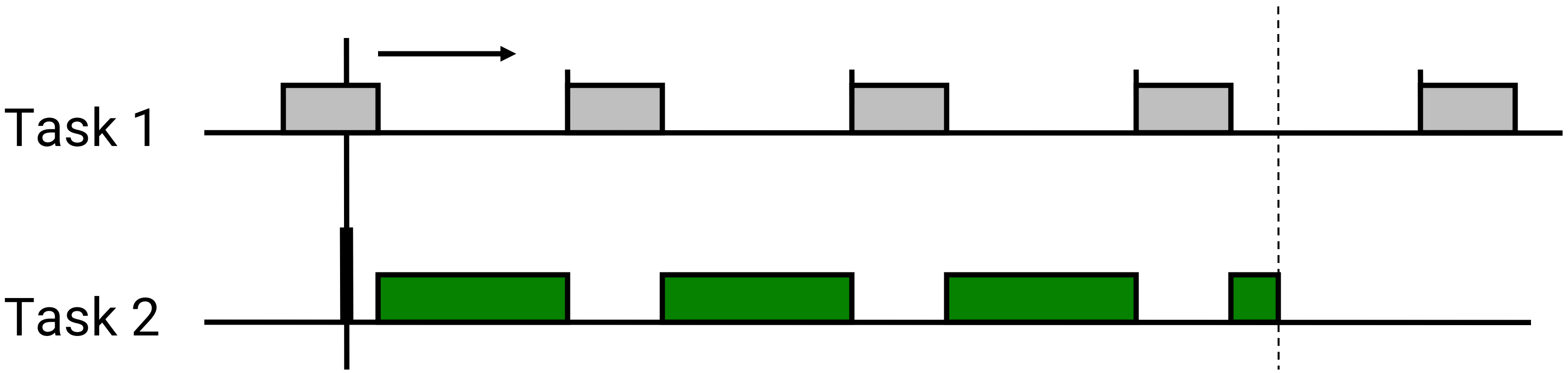
Critical instant theorem

Critical Instant: Proof (Case 1)



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

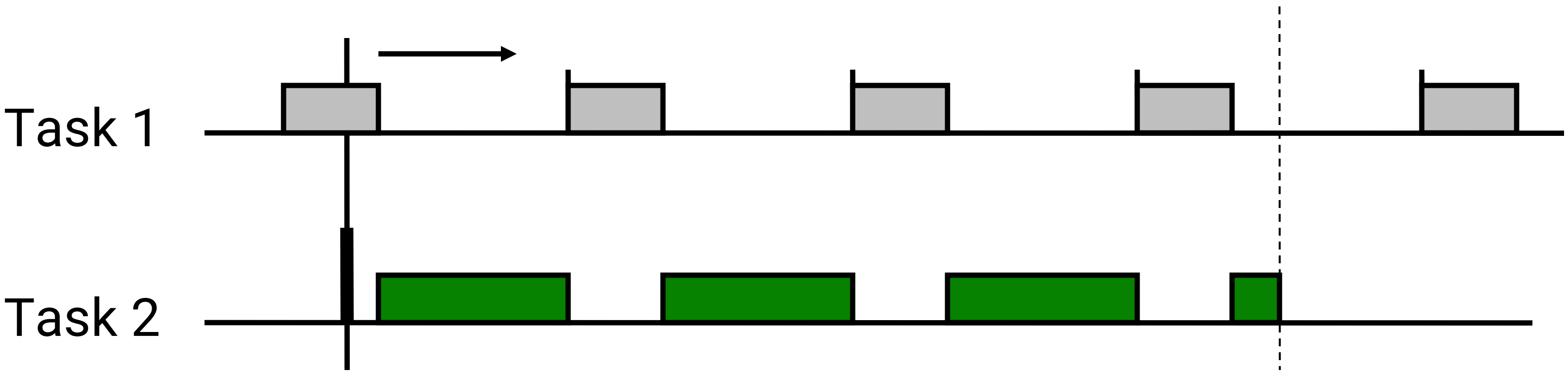
Critical Instant: Proof



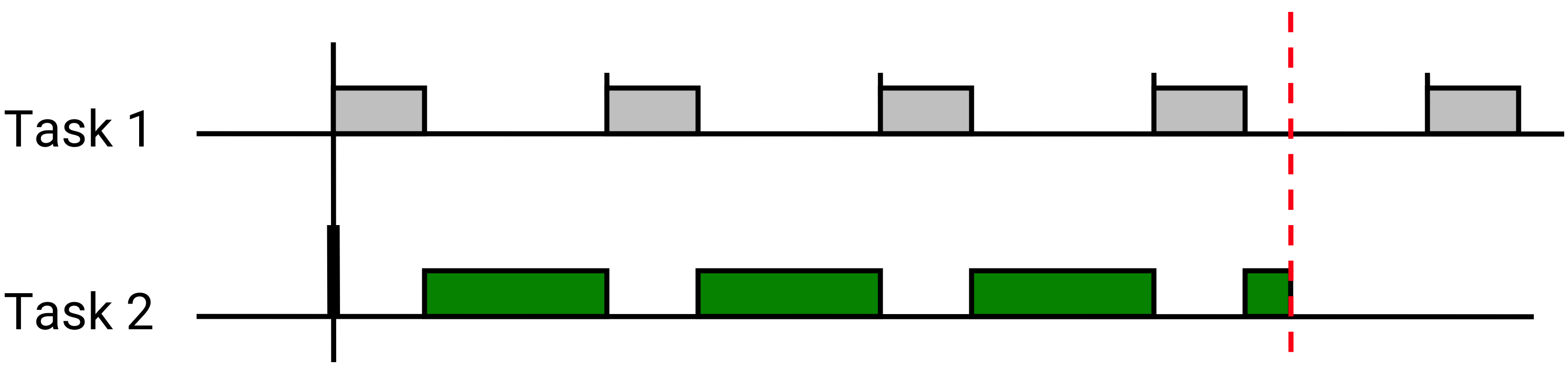
Case 1: higher priority task 1 is running when task 2 arrives

→ shifting task 1 right will increase completion time of 2

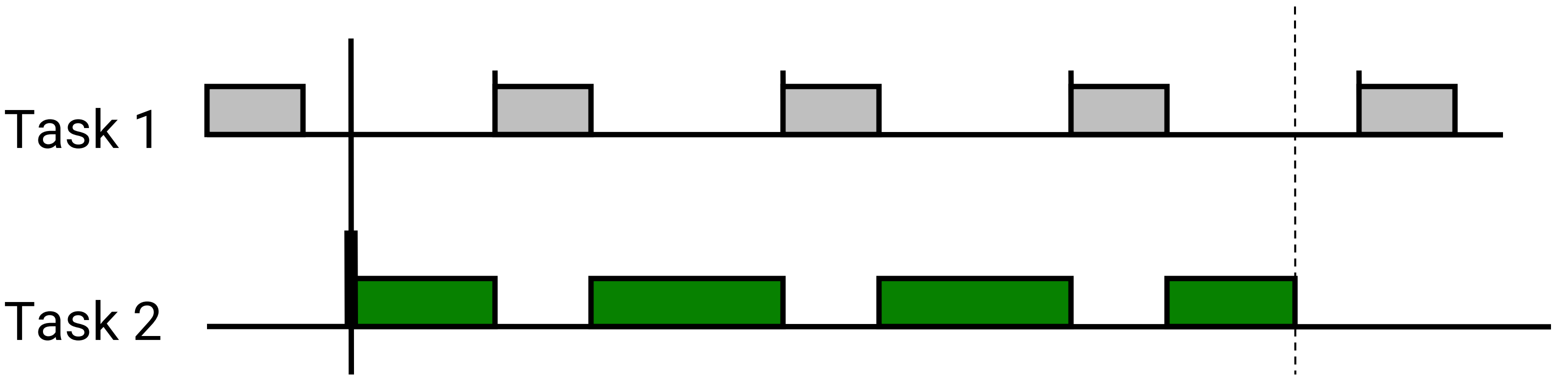
Critical Instant: Proof



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

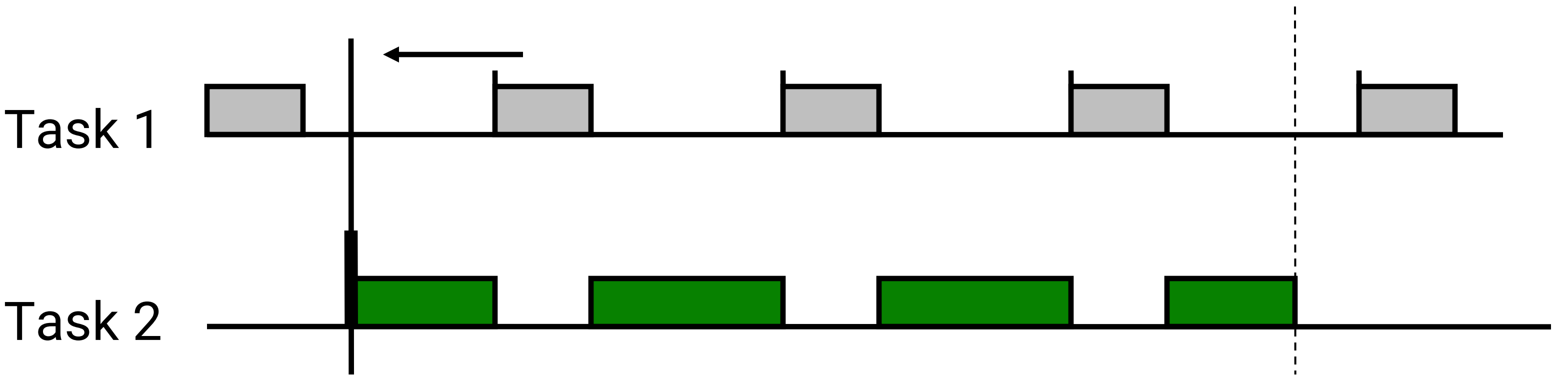


Critical Instant: Proof (Case 2)



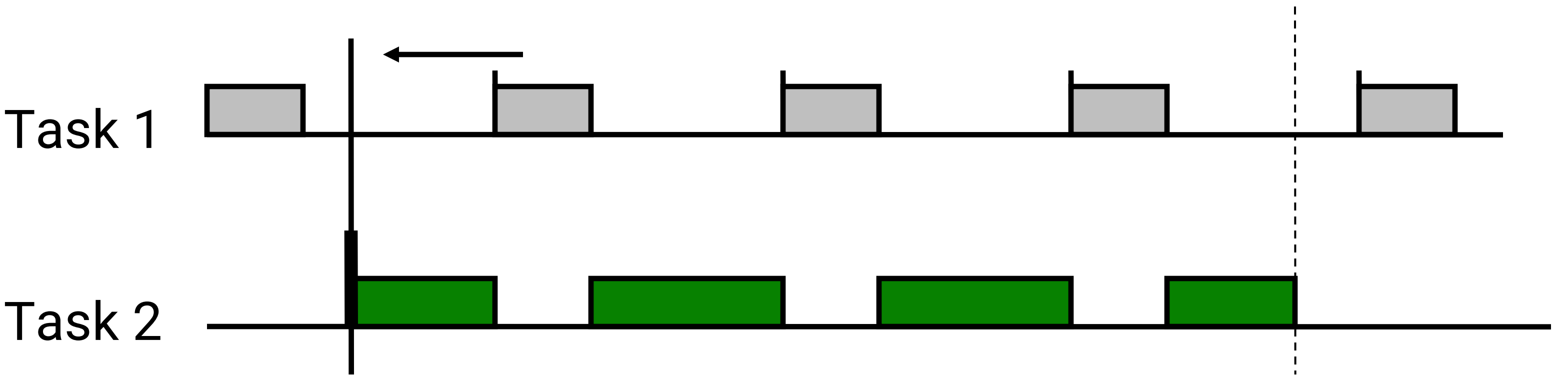
Case 2: processor is idle when task 2 arrives

Critical Instant: Proof (Case 2)

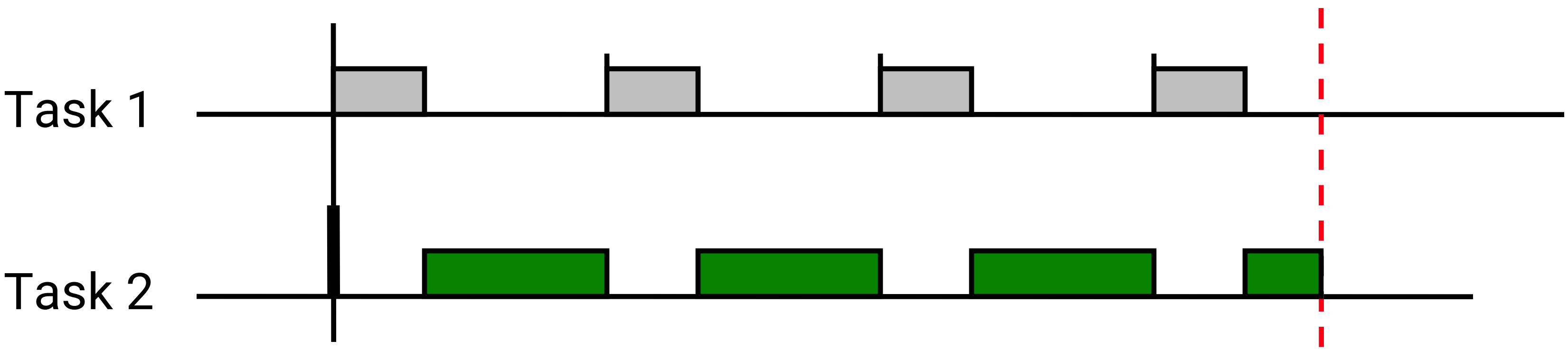


Case 2: processor is idle when task 2 arrives
shifting Task 1 left cannot decrease completion time of 2

Critical Instant: Proof (Case 2)



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



Critical Instant: Remarks

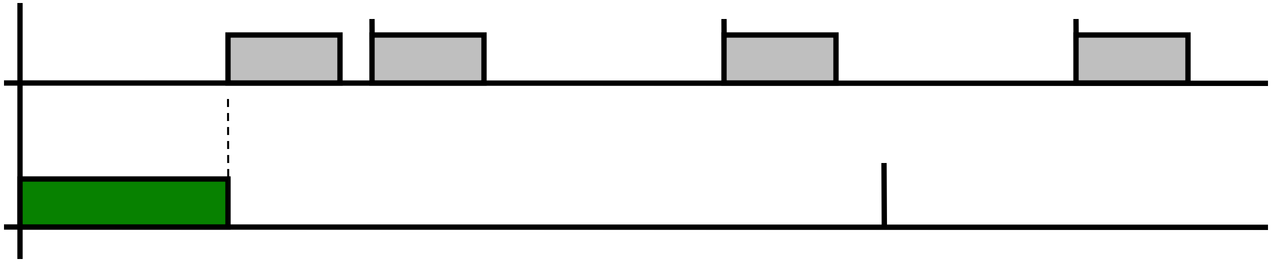
- **All analyses hereafter will assume the critical instant theorem in effect**
- Why is it important to identify the critical instant?
  - Characterizes the **worst-case** scenario when a task experiences the max delay (remember the pitfall in trial #1 in proving RM optimality earlier?)
  - For task schedulability, need to reason only about the feasibility of the job arriving at the critical instant only until its deadline

Assumptions

- All scheduling is preemptive
- A simultaneous release exists even if tasks have non-zero phases
  - Thus critical instant theorem assumed
- Implicit deadlines (deadline = period)
- A task does not suspend itself (on I/O, for instance)
- All tasks in a task set are <u>independent</u> (there are no precedence relations and no resource constraints.)
- All overheads in the kernel are assumed to be zero (context switching and others)

Optimality of the RM policy

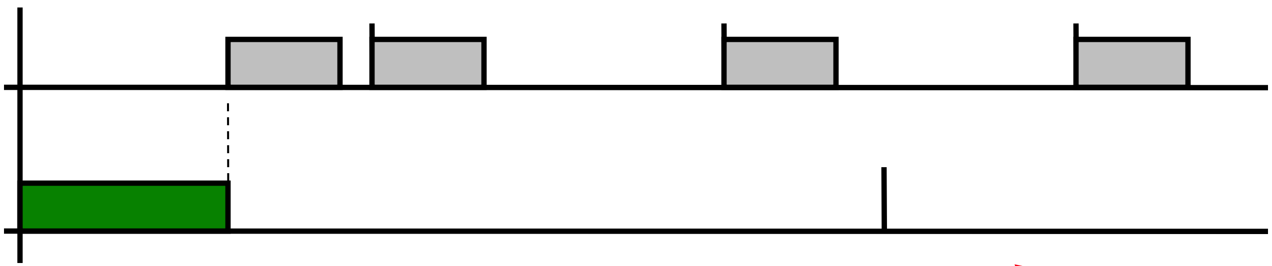
- If any other fixed-priority policy can meet deadlines so can RM



Policy X meets deadlines?

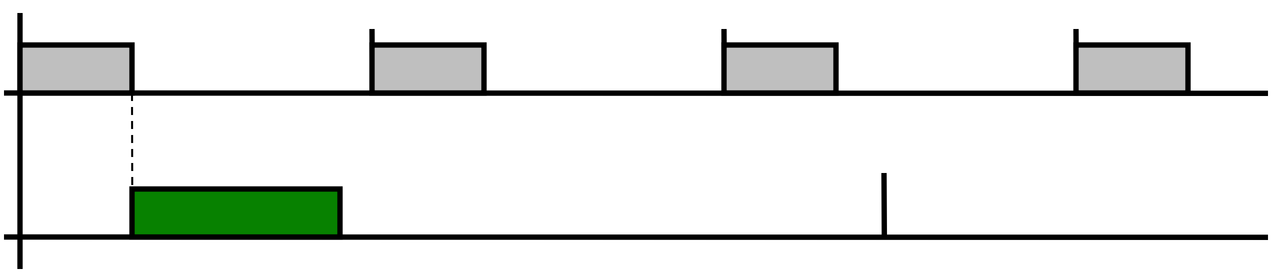
Optimality of the RM policy

- If any other policy can meet deadlines so can RM

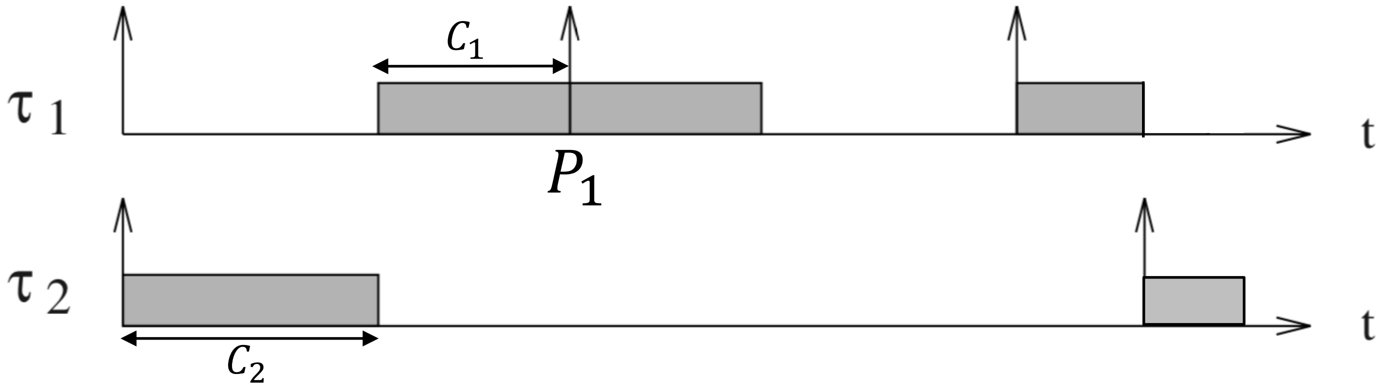


Policy X meets deadlines?
→ RM meets deadlines

YES



Optimality of the RM policy: Proof



Two tasks scheduled **not** according to RM

For feasibility in a non-RM policy, we need $C_1 + C_2 \leq P_1$ to hold at critical instant

Why?

Optimality of the RM policy: Proof

For feasibility in a non-RM policy, we need $C_1 + C_2 \leq P_1$ to hold at critical instant

- Now **exchange priorities** of tasks to make it into an RM assignment
- Plan:
  - identify all possible cases
  - In each case derive feasibility condition
  - Show that if $C_1 + C_2 \leq P_1$ then derived feasibility condition in RM holds

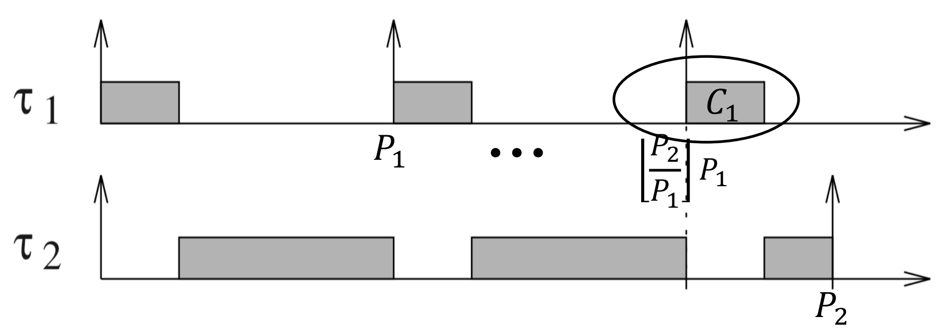
Optimality of the RM policy: Case 1

For feasibility in a non-RM policy, we need $C_1 + C_2 \leq P_1$ to hold at critical instant

- **Case 1:** computation time of τ_1 is short enough that all its *requests* are completed before the second request of τ_2
  - Number of periods P_1 **entirely contained** in P_2 is $\left\lfloor \frac{P_2}{P_1} \right\rfloor$ → Let $F = \left\lfloor \frac{P_2}{P_1} \right\rfloor$
  - **Case 1** translates to $\boldsymbol{C_1 + FP_1 \leq P_2}$
  - **Feasibility:** All computation *requested* by τ_1 during P_2, in addition to C_2, should be completed by P_2

$$(F+1)C_1 + C_2 \leq P_2 \qquad (*)$$

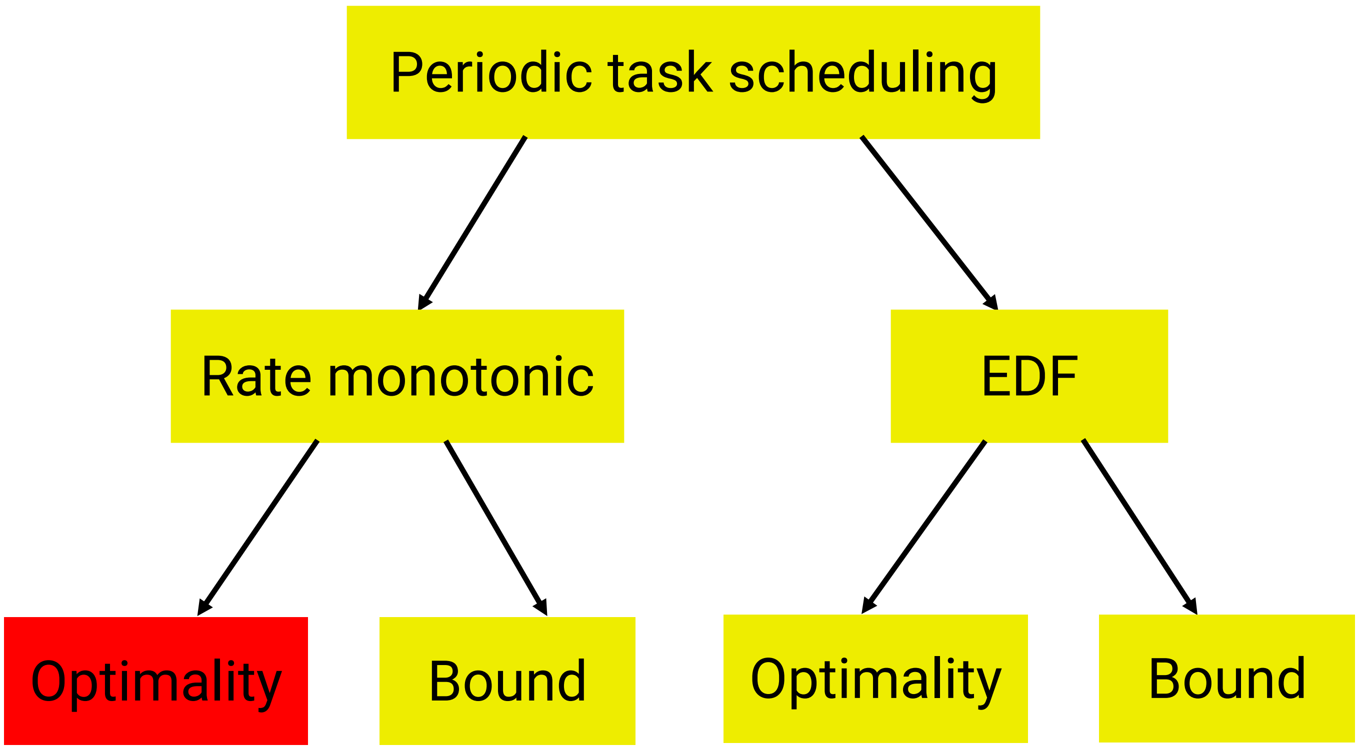
  - Need to show that $C_1 + C_2 \leq P_1$ implies $(*)$



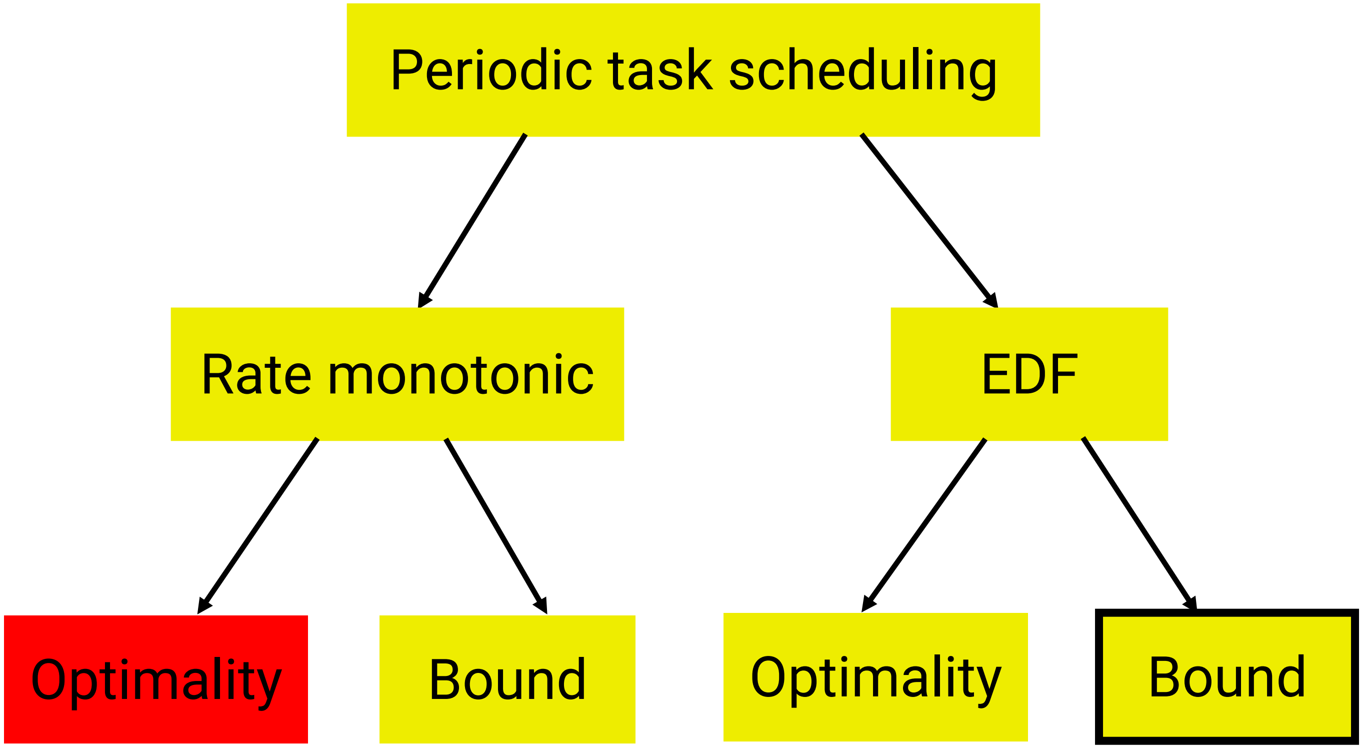
Optimality of the RM policy: Case 2

- homework

What have we achieved?



Next



Periodic task scheduling
Rate monotonic
EDF
Optimality
Bound
Optimality
Bound

Recall: Utilization bounds for schedulability

- U_S is called a **utilization bound** for a given scheduling policy S if

  All task sets with utilization factor $\leq U_S$ can be scheduled using policy S

- U_S is **tight** if, in addition, for a given scheduling policy S the following holds:

  For every $\epsilon > 0$ *, there exists at least one* task set with utilization $(U_S + \epsilon)$ that **cannot** be scheduled using policy S

- A tight bound is the best possible utilization bound: If U_S is tight, then no other $U > U_S$ can be a utilization bound for scheduling policy S

- Of course, the maximum value that U_S can attain for any S is 1. Why? In class

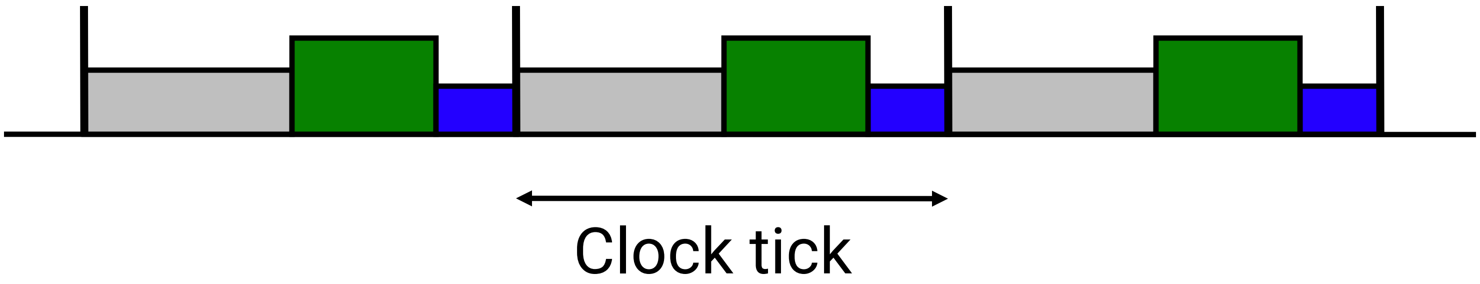
- U_S is also called the ***schedulable utilization*** of algorithm S

Utilization bound for EDF

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

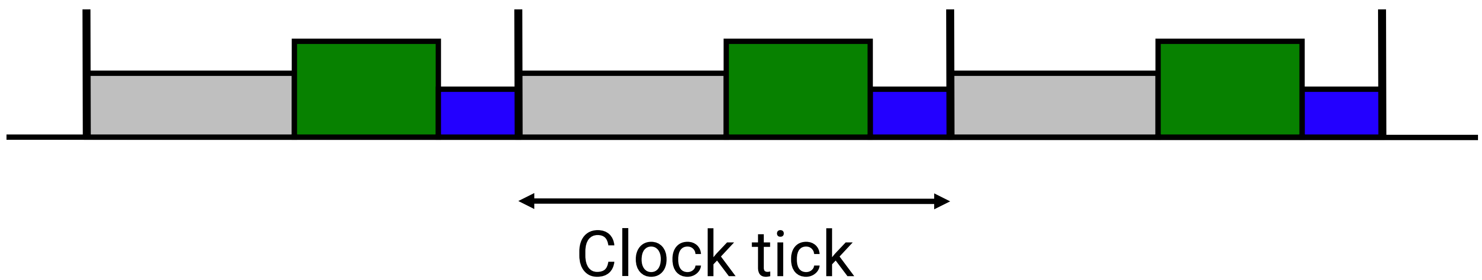
$$\sum_i \frac{c_i}{P_i} = 1$$

- **Plan:** construct an EDF schedule for this taskset
- Imagine a policy that reserves for each task i a fraction u_i of each clock tick, where $u_i = C_i / P_i$



Utilization bound for EDF

- Imagine a policy that reserves for each T_i a fraction u_i of each time unit, where $u_i = C_i / P_i$
- Assume wlog that the periods are integers (perhaps after proper scaling)
- Divide time into, say, integer L-length ticks (e.g., $L = GCD(P_1, \ldots, P_n)$)



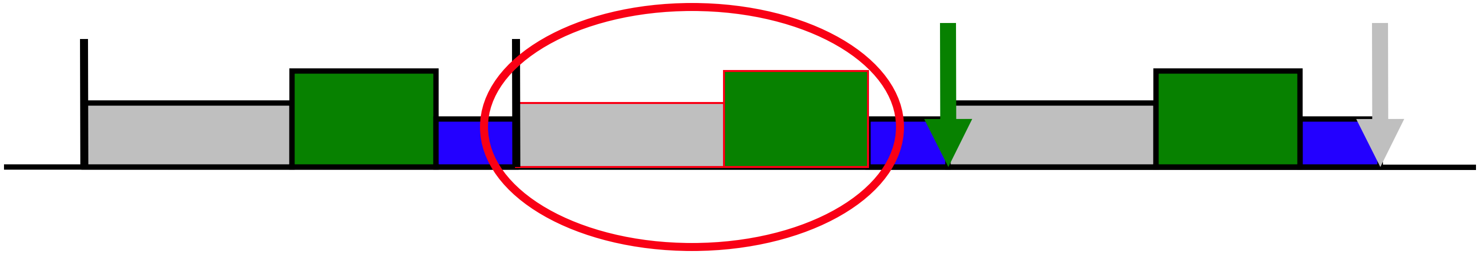
- This policy meets all deadlines, because:
  - Time given to T_i in its period = $u_i \times$ tick length [time/tick] $\times$ (# ticks/period)

    $= u_i L (P_i/L) = (C_i / P_i) P_i = C_i$ [time/period]

    (i.e., enough to finish)

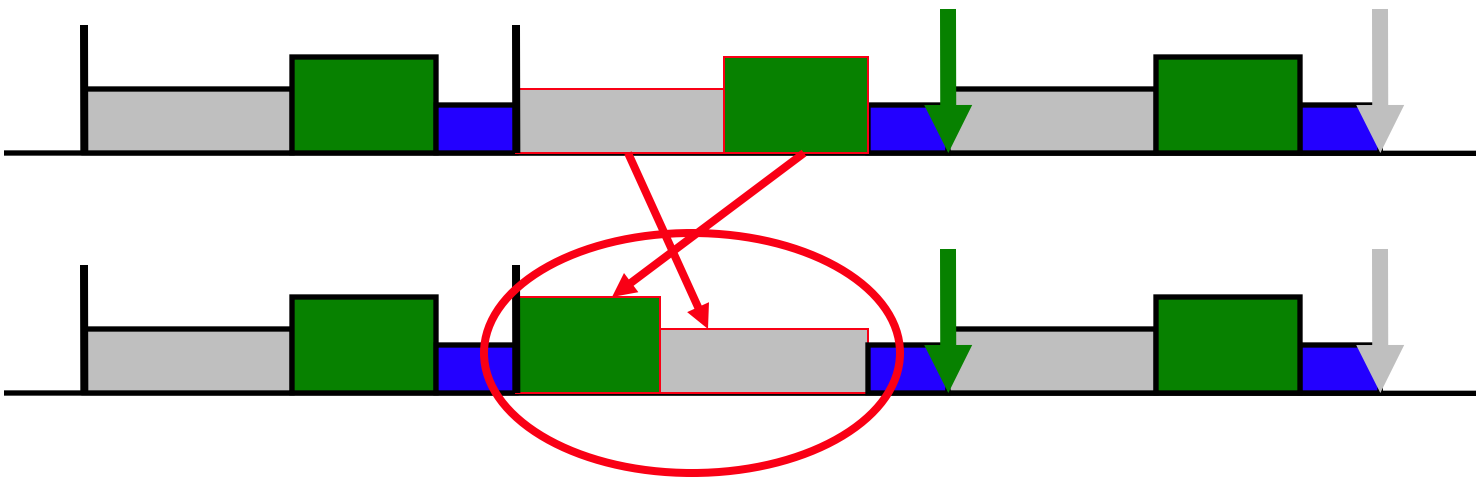
Utilization bound for EDF

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



Utilization bound for EDF

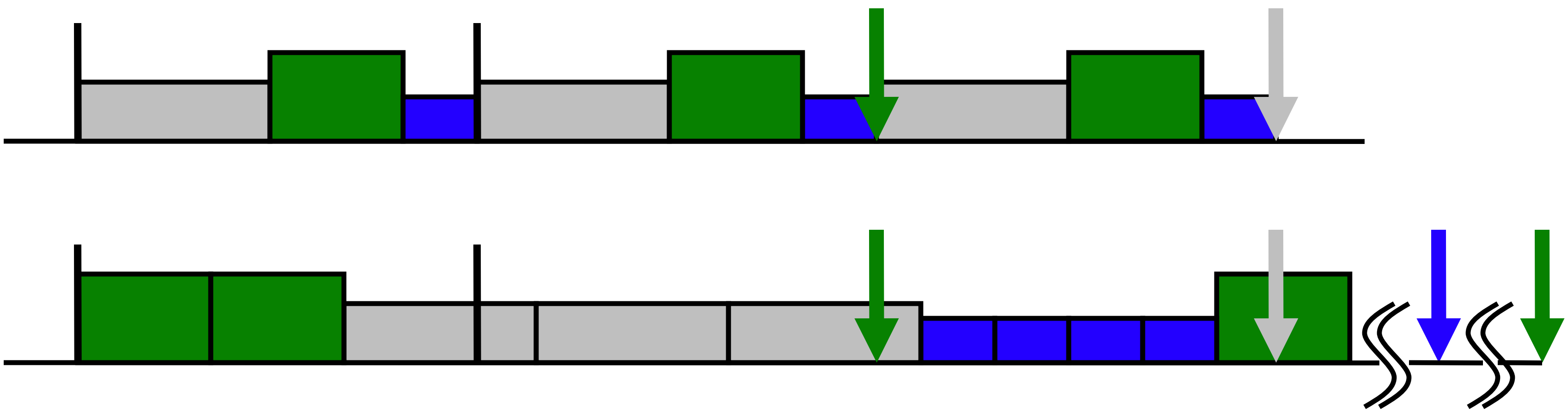
- Pick any two execution chunks that are not in EDF order and swap them



- Still meets deadlines! Why?

Utilization bound for EDF

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

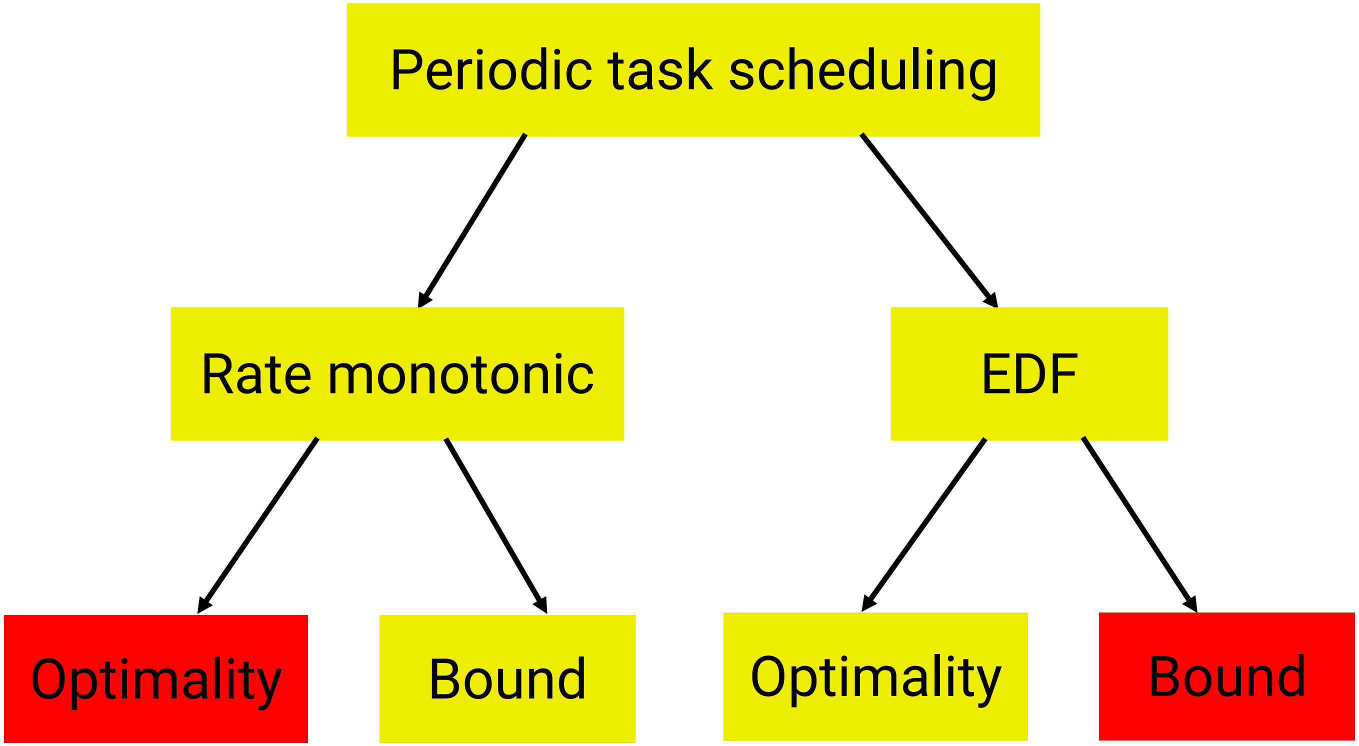


- Still meets deadlines!
- Repeat swap until all in EDF order
    - → EDF meets deadlines

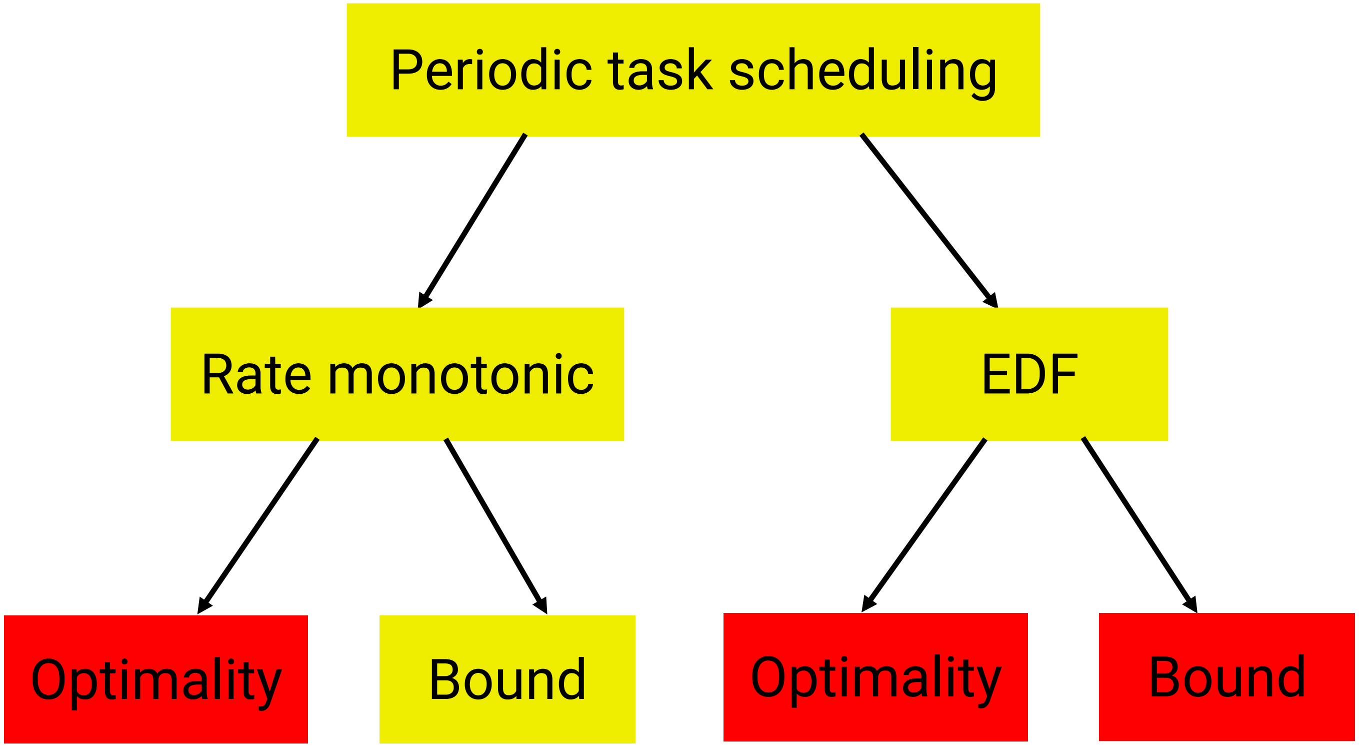
Utilization bound for EDF

- Why does this prove that the utilization bound of EDF is 1?
  - We showed that every taskset with $U = 1$ is feasible under EDF
  - Must also show that every taskset with $U < 1$ is also feasible under EDF
  - This is not needed! Previous argument follows for any $U \leq 1$
  - Consequences:
    - EDF is optimal!
    - EDF is able to schedule every task set who utilization is 1 or less

Next


Periodic task scheduling
Rate monotonic
EDF
Optimality
Bound
Optimality
Bound

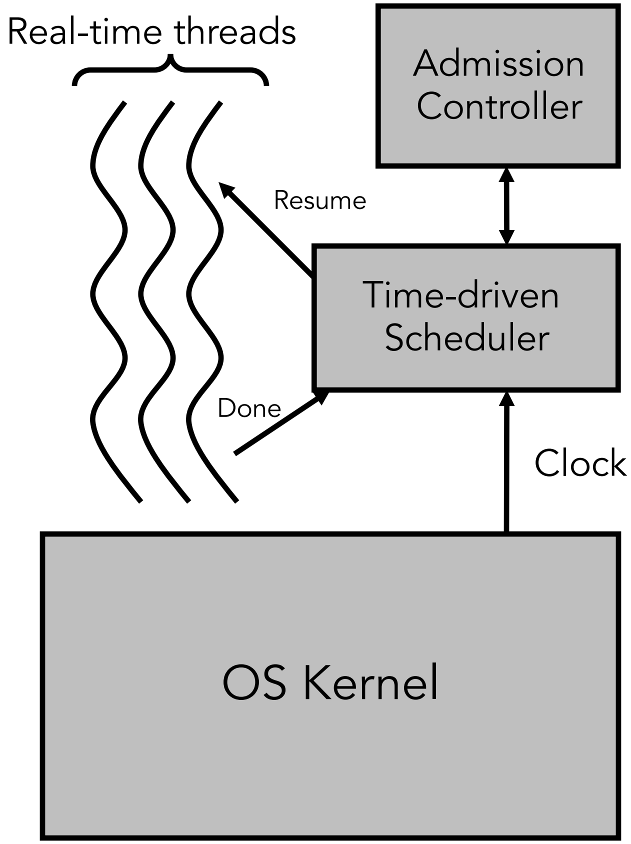
Next



Periodic task scheduling
Rate monotonic
EDF
Optimality
Bound
Optimality
Bound

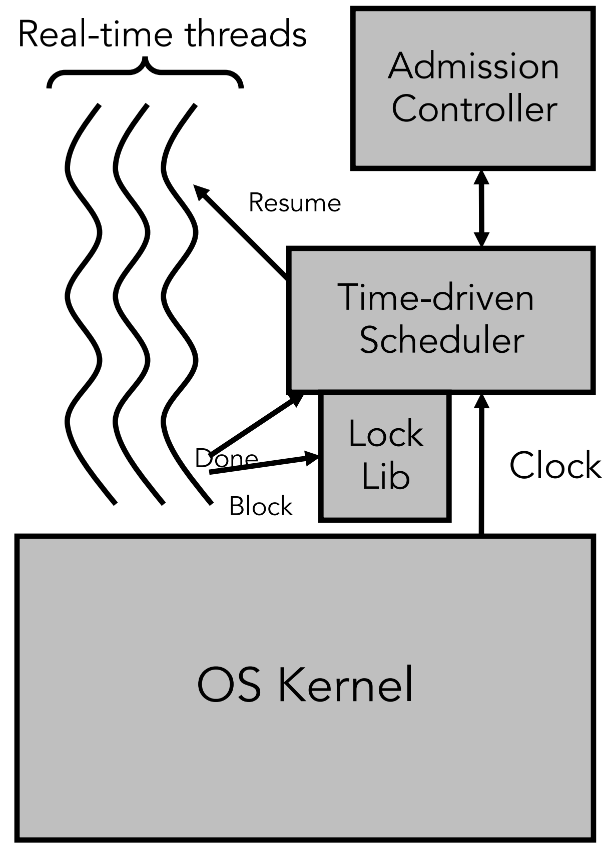
Tick-based scheduling within an OS

- 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

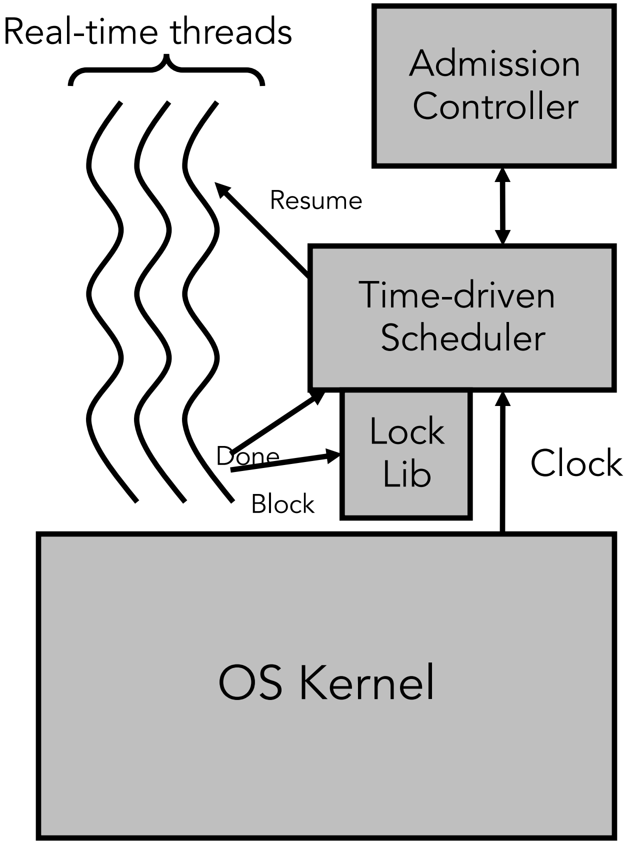


The time-driven scheduler

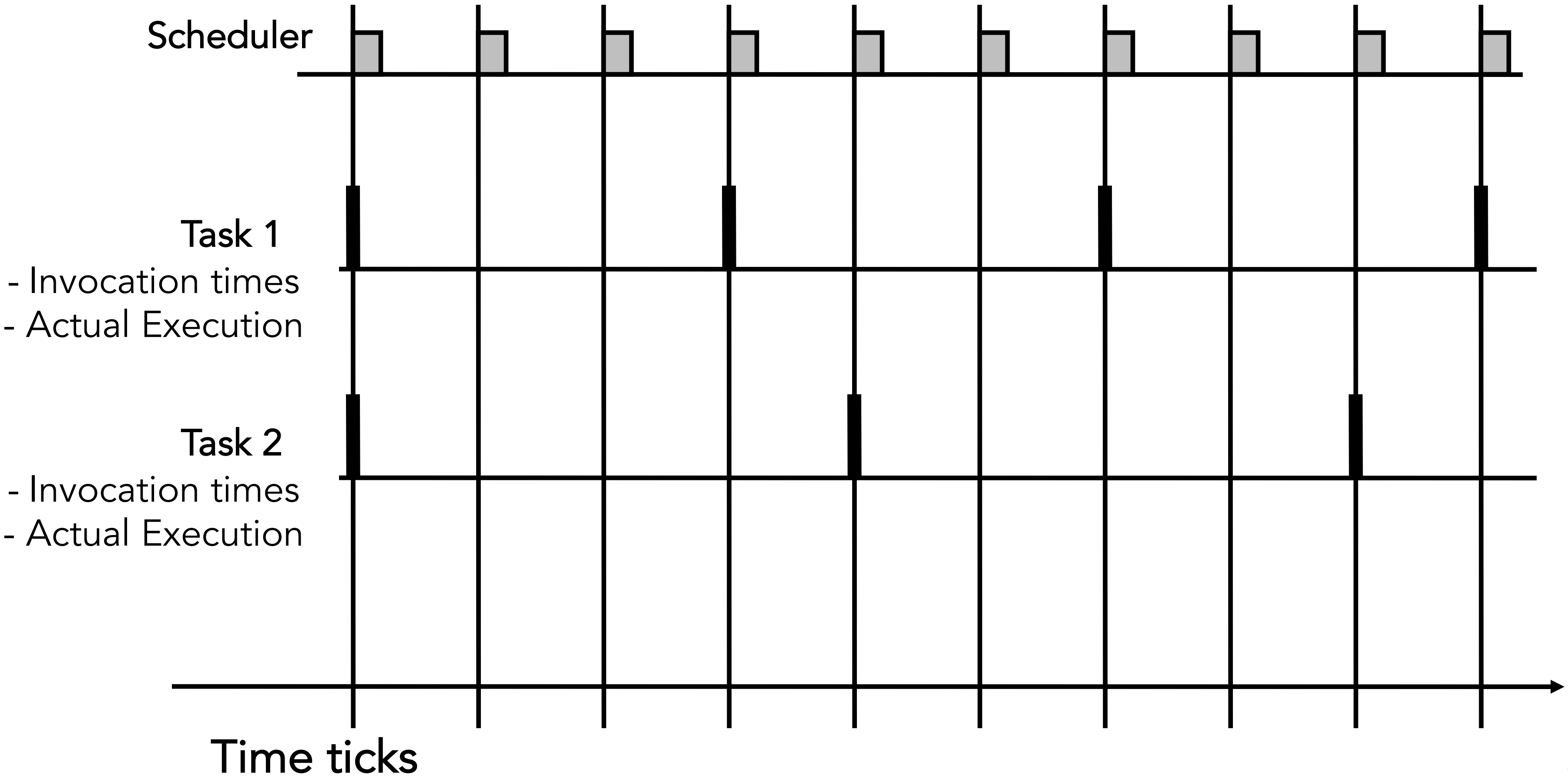
- /* N is the number of periodic tasks */
- For i=1 to N
-   if (current_time = next_arrival_time of task i)
-     put task i in ready_queue

- /* ready_queue is a priority queue that implements
-   the desired scheduling policy. */

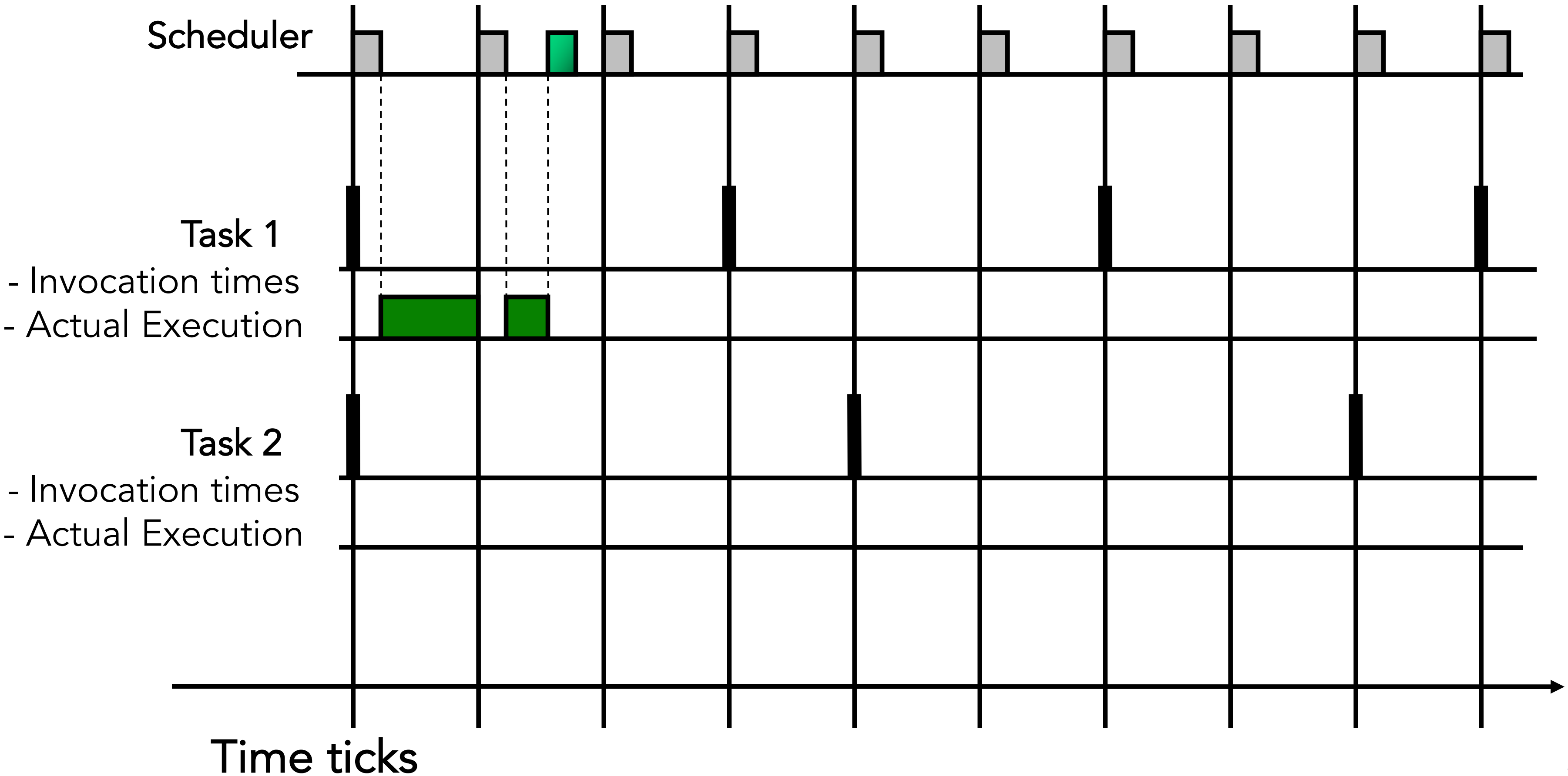
- Inspect top task from ready queue, call it j
- If (a task is running and its priority is higher than priority of j) return
- Else resume task j (and put the running task into the ready queue if applicable); return



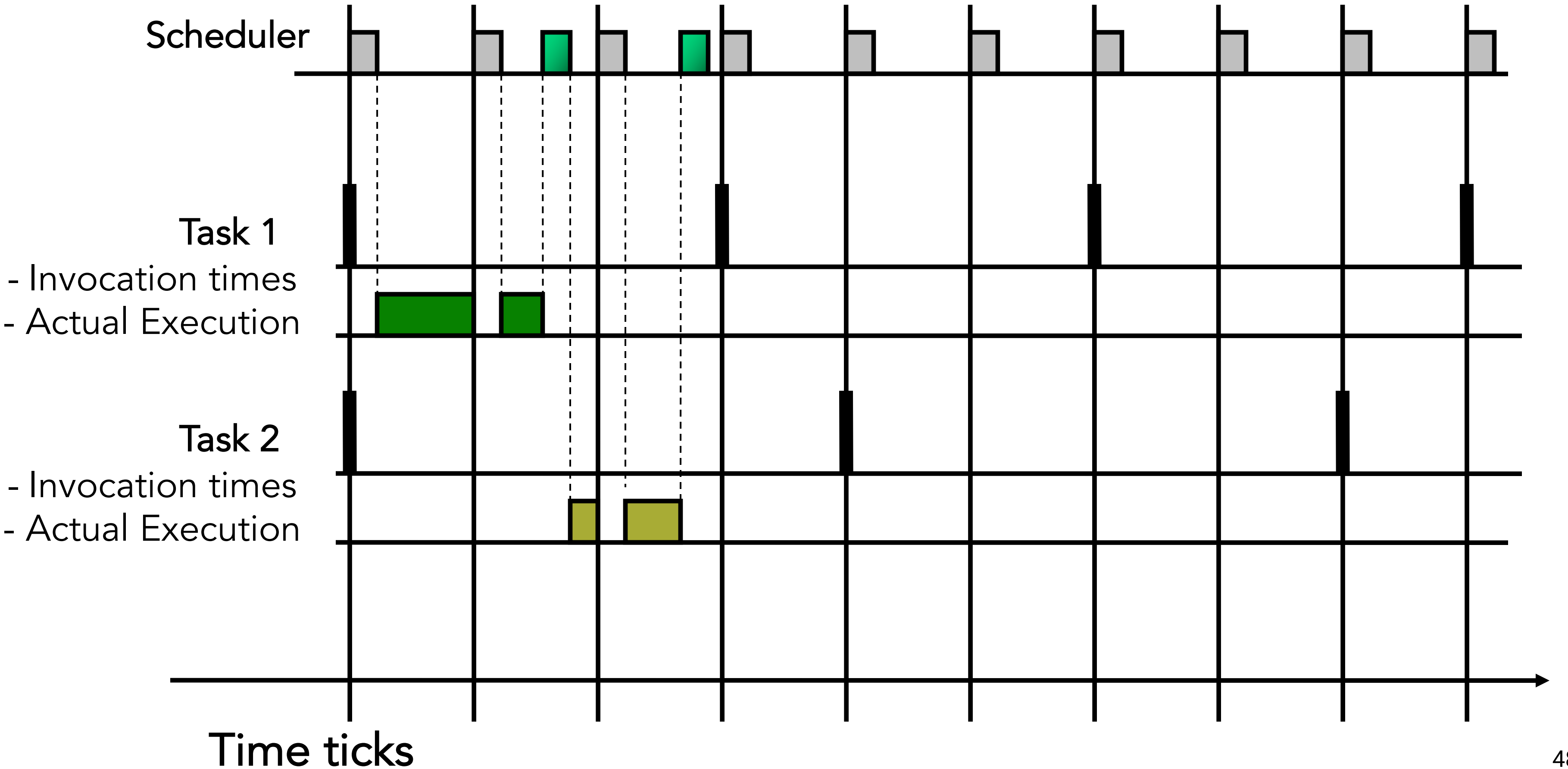
An example schedule



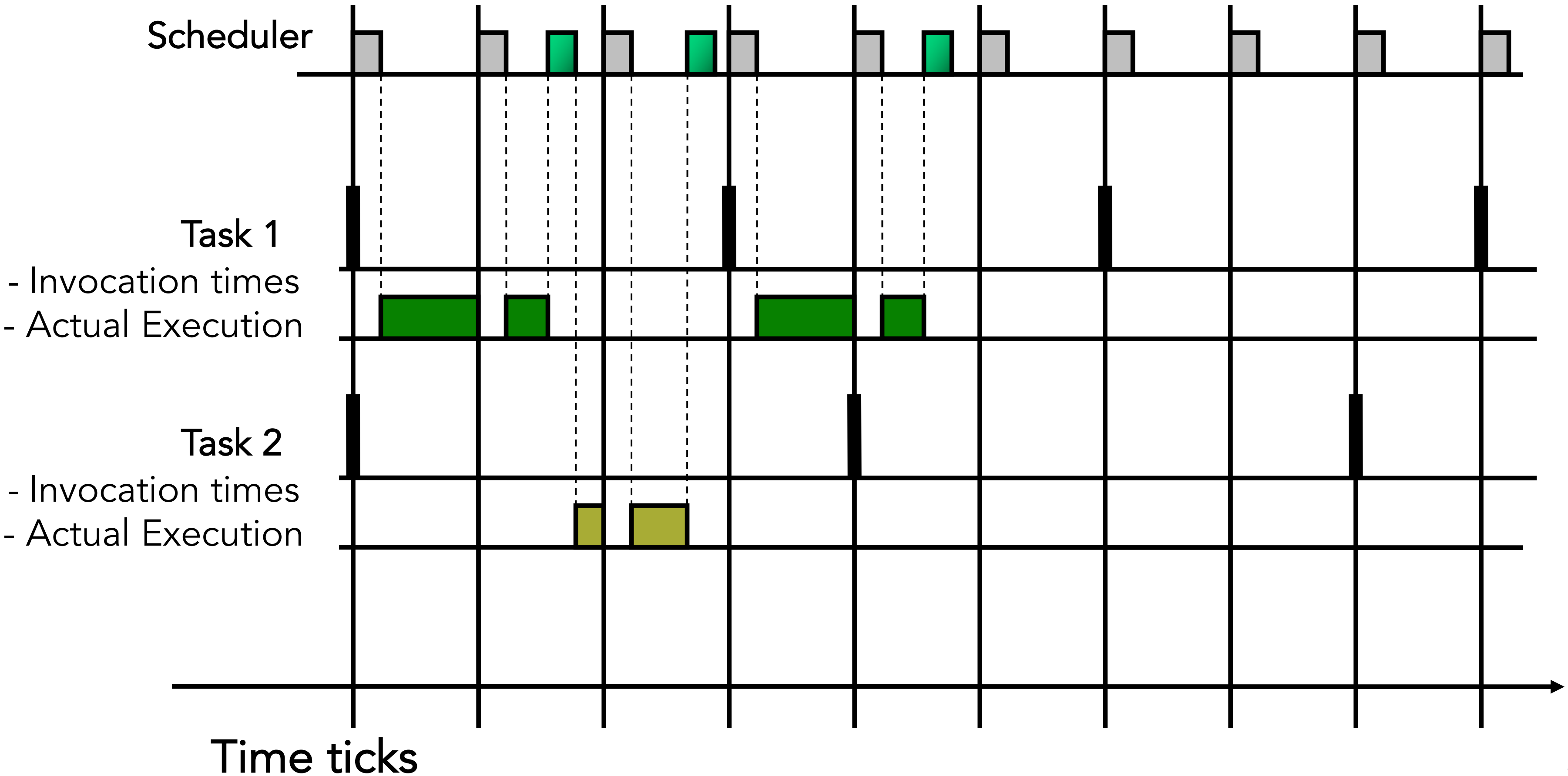
An example schedule



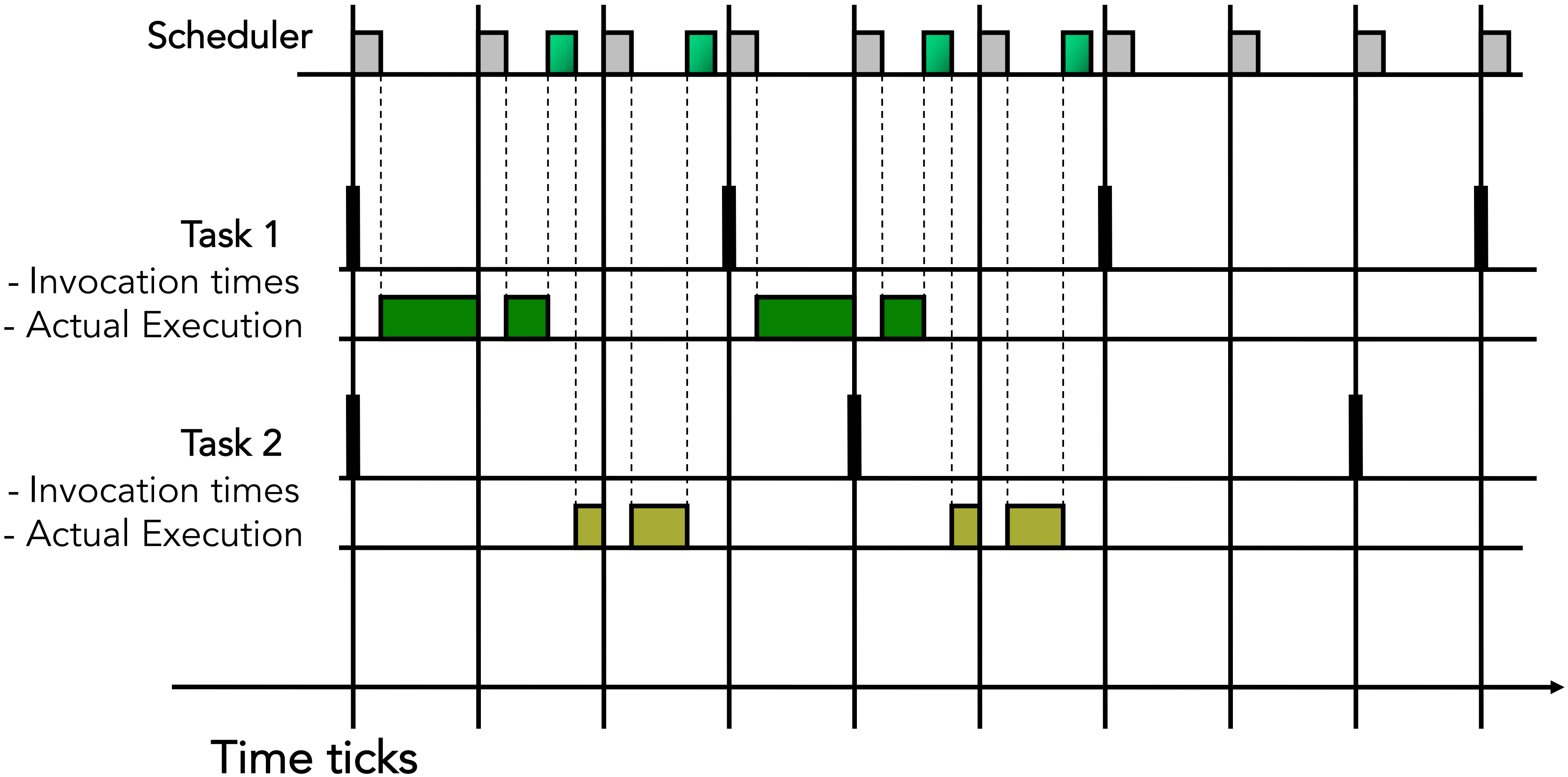
An example schedule



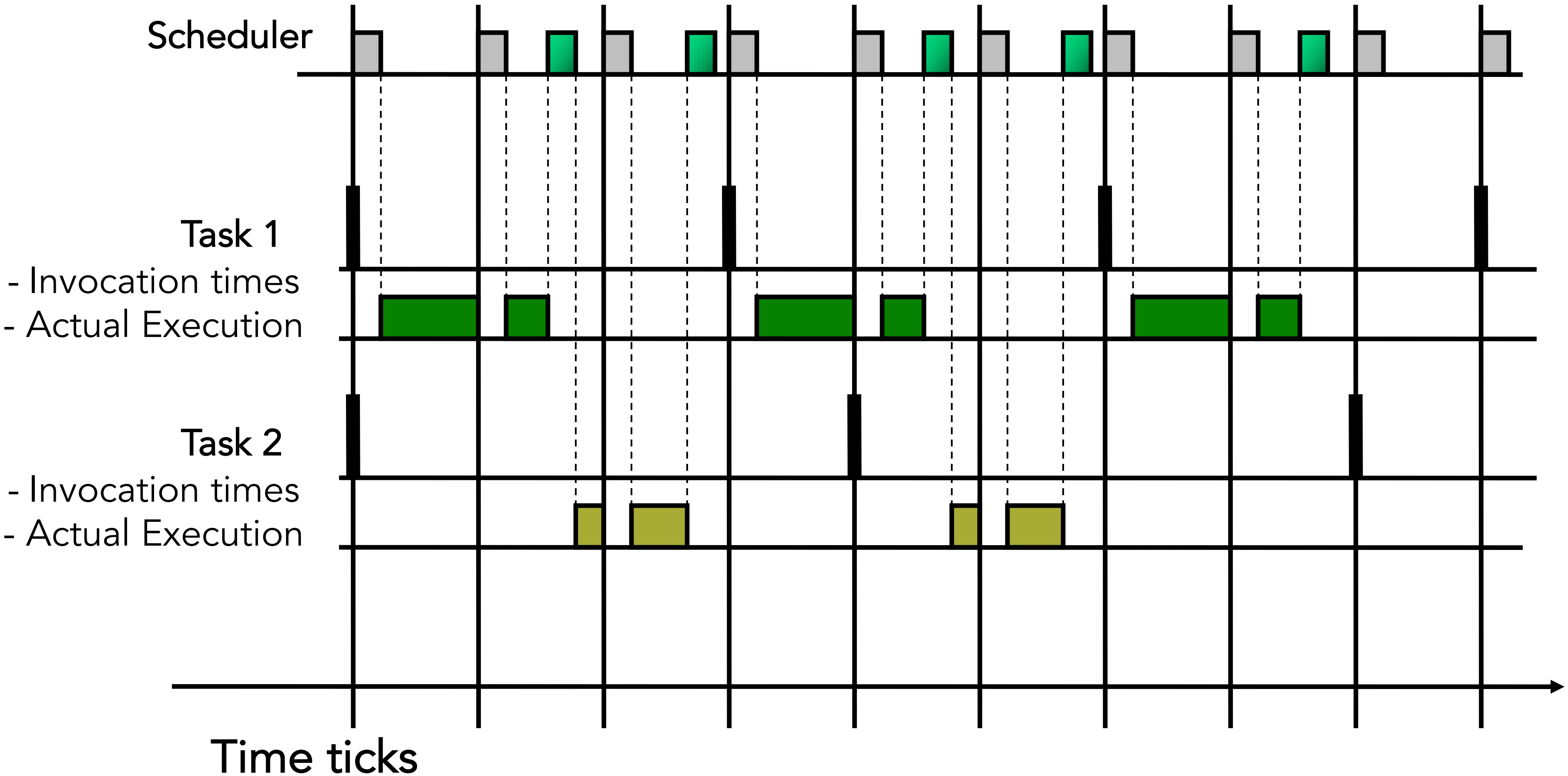
An example schedule



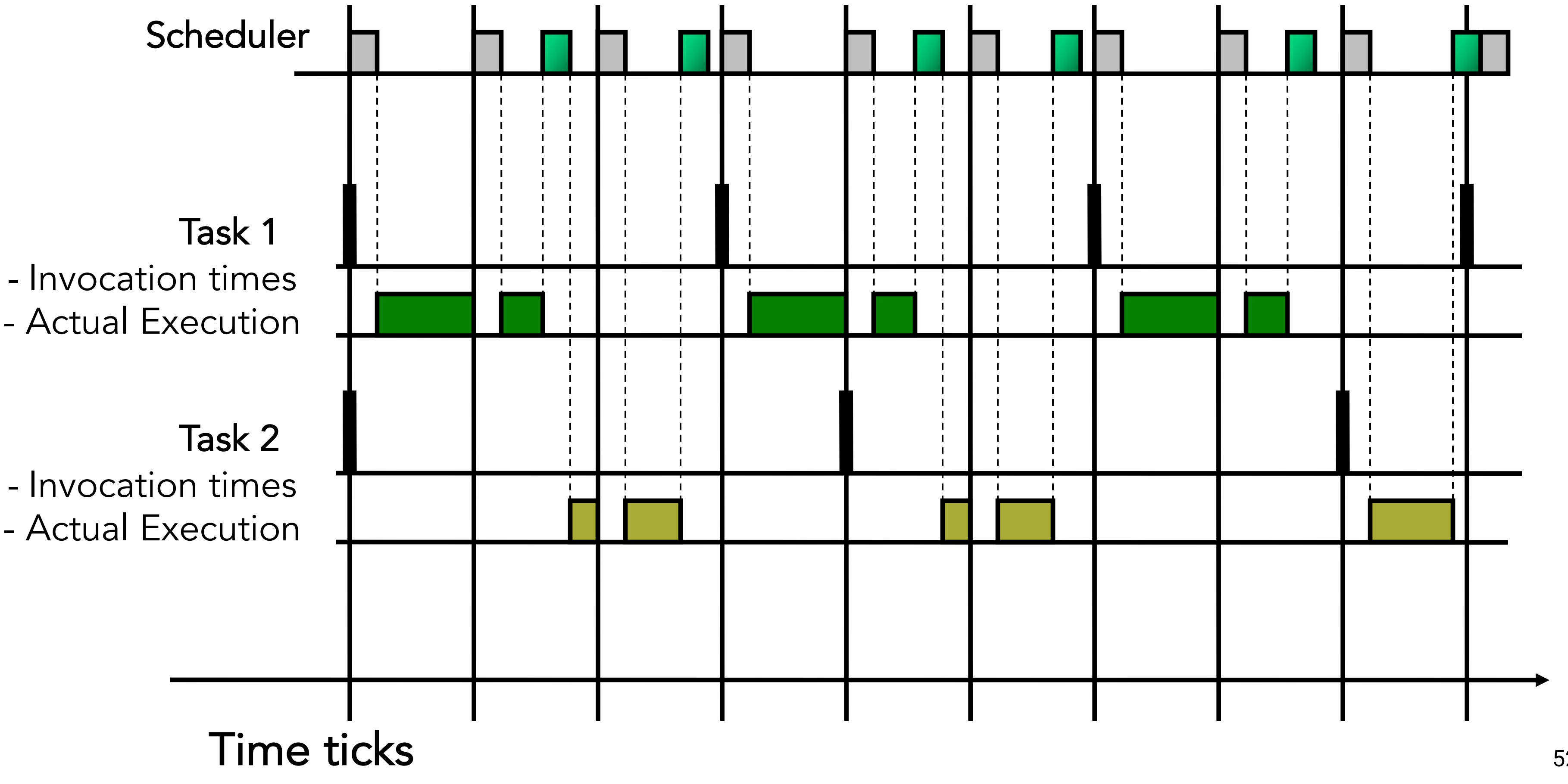
An example schedule



An example schedule

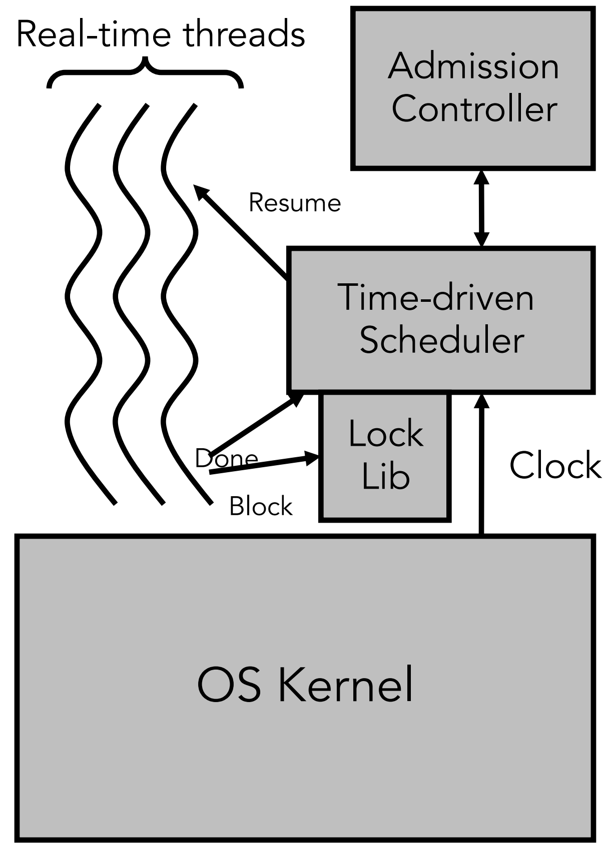


An example schedule



Admission controller

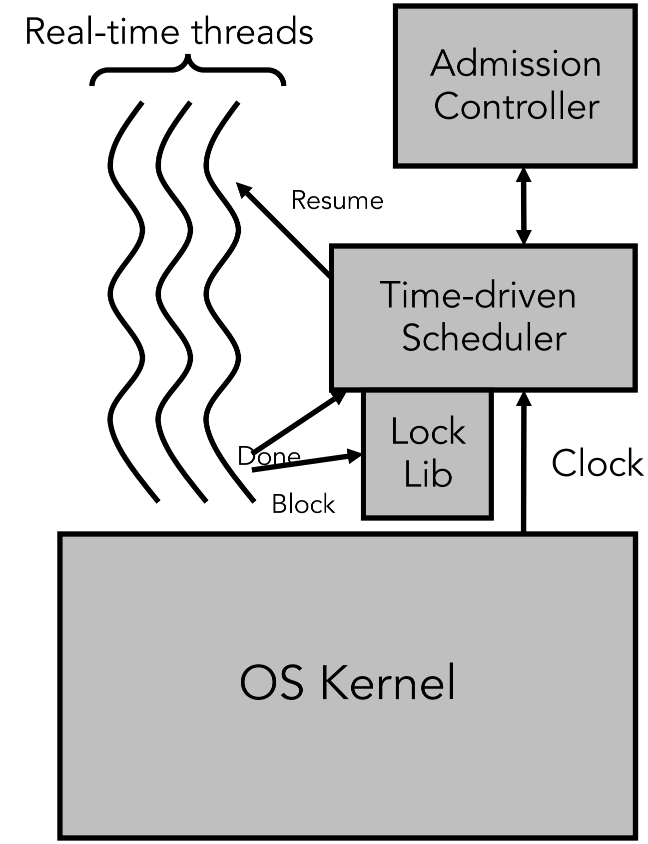
- Implements schedulability analysis
  - If $U+C_{new}/P_{new} < U_{bound}$ admit task
  - Must account for various practical overheads. How?
  - Examples of overhead:
    - How to account for the overhead of running the time-driven scheduler on every time-tick?
    - How to account for the overhead 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!)