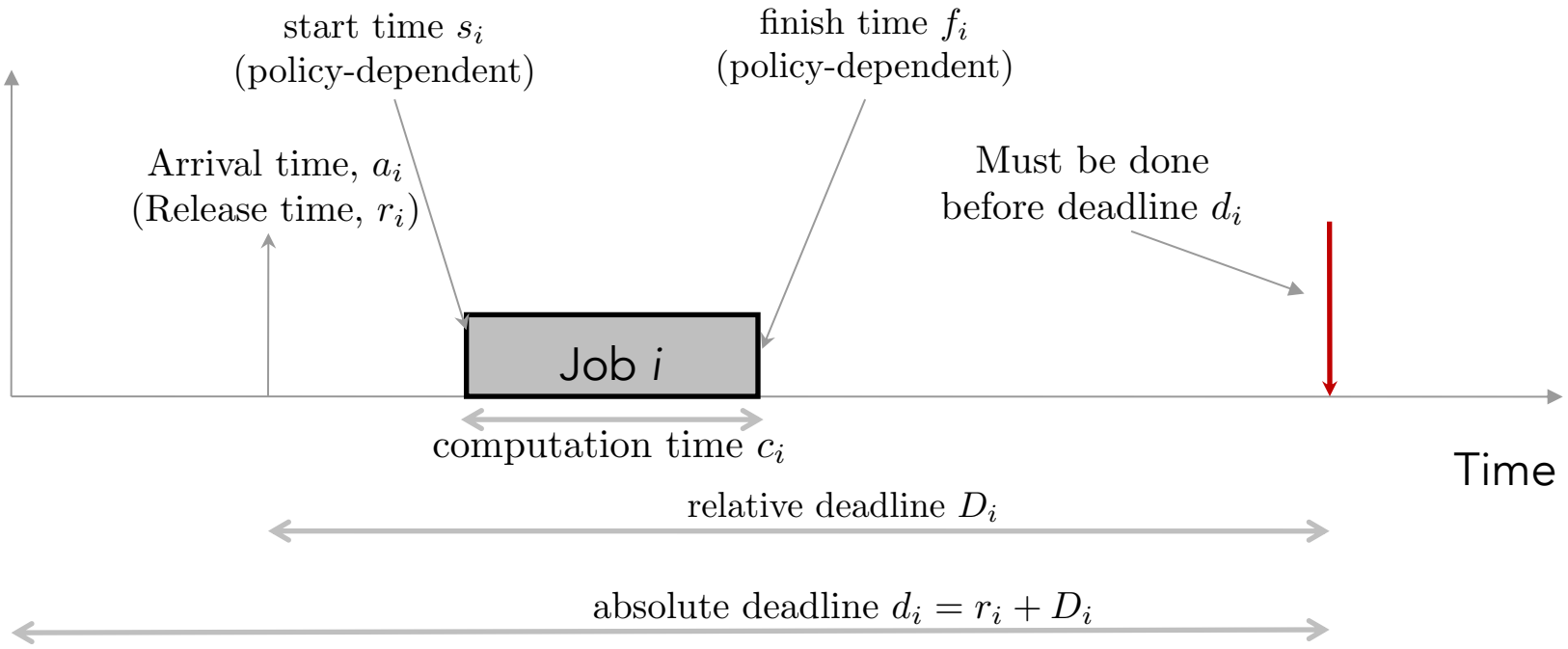
General Concepts

Introduction to real-time systems

Abstraction

- To reason about timeliness we need an abstraction that is:
  1. Expressive enough to capture realistic workload models and parameters
  2. Constrained enough to derive useful results (tractable)

Job model



Online vs. Offline scheduling

- What do we know about the jobs to be scheduled?
- **Offline**: All job parameters $r_i, c_i, D_i, \ldots$ and number of jobs n are known a priori
- **Online**: Jobs arrive as a *stream*
  - Might not know arrival times
  - We do not know number of jobs
  - We might not know job execution time requirements until job arrives or even until job finishes execution!

Job Scheduling: Policies

- A rule that specifies which job occupies which processor at every time instant
- May be
- Deterministic
- Randomized
- Preemptive or non-preemptive
- *Priority-driven* or *clock-driven* (or a hybrid of both, or none of which!)

Job Scheduling: Policies

- **Deterministic scheduling policy:** *n* jobs

  A function S: $\mathbb{R}_+ \rightarrow \{1, \ldots, n\}$ where $S(t)$ is the index of the job that occupies the processor at time t

- $S(t) = 0$ if the processor is kept idle
- Some authors impose the requirement that a policy be a *step function*

  → $\forall t > 0, \exists t_1, t_2 \geq 0$ with $t \in [t_1, t_2)$ such that $S(t') = S(t)$ for every $t' \in [t_1, t_2)$

Job Scheduling: Priority-driven Policies

- **Priority-driven policies**
  - Assign priorities to jobs so that the available job with the highest priority occupies the processor
  - Priority can be static or vary through time as job execution advances (dynamic priority)
  - **Q:** With n jobs, **how many *static* priority assignments exist?** (where each job has a unique priority)
  - **Q:** Can an optimal static priority assignment be found in time that is polynomial in the problem parameters?
  - What does **optimal** mean?
    - For now, minimizes some scheduling objective → will consider more sophisticated definitions of optimality shortly
- Priority-driven Policies: also called **work-conserving, list scheduling, greedy** → processor is never left idle on purpose, always schedule a job if one is available

Job Scheduling: Objectives (cost functions)

We have n jobs $J_1, \ldots, J_n$ with parameters r_i, c_i, D_i, etc

- Objective related to timing properties
  - minimize *total completion (finish) time:* $\sum_{i=1}^{n} f_i$
  - minimize *total* ***weighted*** *completion time:* $\sum_{i=1}^{n} w_i f_i$
  - Minimize **makespan** $\max_i f_i$
    - *Makespan:* finish time of last job to leave the system (length of schedule)
    - Is "machine owner oriented": A minimum makespan implies a good utilization of the machine(s).
    - Not suitable for interactive applications: Jobs released early might be delayed till end of an optimal-makespan schedule → not acceptable by "users"
  - Minimize total flow (*response) time:* $\sum_{i=1}^{n}(f_i - r_i)$
    - $(f_i - r_i)$*:* total time job i is in system = *waiting time + processing time*
    - Beneficial from "users" perspective
    - Also minimizes total completion time and total waiting time
    - Drawback: Can lead to *starvation*: Might still cause some jobs to be delayed unboundedly (no *fairness*)

Job Scheduling: Objectives (cost functions)

- Minimize max. *flow time:* $\max_i(f_i - r_i)$
  - Schedule responsive to *each* job
  - Starvation-free
- ***Lateness***: $L_i = f_i - d_i$ → Minimize maximum lateness $L_{\max} = \max_i L_i$
  - Worst case violation of deadlines. The earlier the better!
- ***Tardiness:*** $T_i = \max\{0, L_i\}$ → Minimize maximum tardiness $T_{\max} = \max_i T_i$
  - Doesn't care how early (before the deadline) jobs finish
- **# of tardy jobs** $\sum_{i=1}^{n} \mathbf{1}\{L_i > 0\} = \sum_{i=1}^{n} \mathbf{1}\{f_i > d_i\}$

Deadline-related

Job Scheduling: Stochastic Models

- What if we have only **statistical data** of job parameters? E.g.,
  - Jobs (events) *arrive* randomly
  - Job execution time (demand) is random
    - **Inventory** where product demands are not known exactly but we know their distribution
    - Machine might *fail* randomly, causing execution times to be "stretched" randomly by (down time + time to service machine)
- Questions of interest:
  - What is the *probability* that jobs miss their deadlines under some policy?
  - What is the *expected* max lateness $\mathbb{E}L_{\max}$ under some fixed policy?
  - What policy *minimizes* the expected max lateness?

Job Scheduling: Preemption

- Can we interrupt a job once it starts execution?
- Preemptible jobs/resources
  - **Example:** execution of programs on your computer may be interleaved to implement multitasking
- Non-preemptible jobs/resources:
  - You might be on an important phone call so you block all other incoming calls
  - Disable interrupts during the execution of an ISR
  - When using resources with complex or costly state associated with it → prohibitive context switching costs → *blocks on disk*
- No-preemption problems are usually computationally harder (**NP**-Hard)
- Allowing preemption gives more flexibility to designer → sometimes more tractable
  - **Example**: Minimizing maximum lateness with arbitrary release times is **NP**-hard when *no* preemptions allowed → becomes **poly-time** solvable when preemptions allowed
  - **Downside**: preemption (context switching) costs might be prohibitive

Example: Minimize total weighted completion time

- Given have n jobs $J_1, \ldots, J_n$
  - J_i has execution time $e_i > 0$ and weight (importance) $w_i \geq 0$
  - All jobs arrive at time 0 → $r_i = 0 \;\; \forall i \in \{1, \ldots, n\}$
  - One processor
  - No Preemption
- **For instance:** Job might be *message to be broadcasted*, all messages ready for transmission at time 0, message i has length $e_i > 0$, and a fraction $w_i \geq 0$ of receivers are interested in message i
  - Finish time f_i is the time at which message is fully transmitted

$$\textbf{minimize} \;\; \sum_{i=1}^{n} w_i f_i$$

Example: Minimize total weighted completion time

- Is there an optimal priority assignment?
- If so, how does the priority function look like?
- Think … greedy
- Key: Job interchange argument
- Will do proof of optimality in class
- Optimal policy: **weighted shortest processing time first (WSPT)**
- **Order jobs such that** $\frac{w_1}{c_1} \geq \frac{w_2}{c_2} \geq \cdots \geq \frac{w_n}{c_n}$ **:** ratio $\frac{w_i}{c_i}$ is job i's index under WSPT
- At any time instant schedule the available job with the highest index

Optimality of WSPT for total weighted completion time

- Let S be an optimal schedule that is **not** WSPT
  - Then there are two adjacent jobs i and k (assuming i is before k in S) such that $\frac{\boldsymbol{w_i}}{\boldsymbol{c_i}} < \frac{\boldsymbol{w_k}}{\boldsymbol{c_k}}$ → some pair of adjacent jobs violates the WSPT ordering
- Let t be the start time of J_i in S → under S:
  - $f_i = (t + c_i), \qquad f_k = (t + c_i + c_k)$
- Interchange the order of J_i and J_k to produce a new schedule S'
  - Interchanging those jobs doesn't affect $\sum_j w_j f_j$ of job sequence before and after J_i and J_k → difference in objectives of S and S' is solely due to J_i and J_k
- Under S':
  - $f_i' = (t + c_i + c_k), \qquad f_k' = (t + c_k)$

Optimality of WSPT for total weighted completion time

- But we know that $\frac{w_i}{c_i} < \frac{w_k}{c_k} \Rightarrow w_i c_k < w_k c_i$

$$
\begin{aligned}
w_i f_i' + w_k f_k' &= w_i(t + c_i + c_k) + w_k(t + c_k) \\
&= w_i t + w_i c_i + \underbrace{w_i c_k}_{< w_k c_i} + w_k t + w_k c_k \\
&< w_i(t + c_i) + w_k(t + c_i + c_k) \\
&= w_i f_i + w_k f_k
\end{aligned}
$$

- Objective under S' < Objective under S → contradicts optimality of S

Variations

Allow preemptions and arbitrary release times

Problem: $1|r_j, \text{prmt}|\sum w_j f_j$

1. Is the Weighted Shortest ***Remaining*** Processing Time rule optimal for this problem?
2. What if all weights are unity but we allow arbitrary release times and preemptions? $1\ |r_j, \text{prmt}|\ \sum f_j$
3. What if we disallow preemptions in 2.? $1\ |r_j|\ \sum f_j$

Optimality of SRPT for $1 \left| r_j, \text{prmt} \right| \sum f_j$

- SRPT is a **dynamic** priority rule
  - Priority of J_i at time $t = \frac{1}{c_i(t)}$ if $t \geq r_i$ and is undefined otherwise
    - $c_i(t)$ is the *remaining* execution time of J_i at time t
  - The job currently running can be preempted only by a job with higher priority that arrives during its execution
    - **Why?** A job's priority is a non-decreasing function of time → a job's priority increases as it is given more processor time
    - The job that is in possession of the processor cannot be preempted by a job that has arrived prior to its start time
    - This shows that (1) we need to check only arrival times to determine which job occupies the processor at every time instant, and (2) # of preemptions of SRPT is finite

Optimality of SRPT for $1 \left| r_j, \text{prmt} \right| \sum f_j$

- Let S be a policy that is optimal but not SRPT
- Then there is $t > 0$ and available job i with highest priority (least RPT) that is not scheduled at t but instead another available job $k \neq i$ is allocated the processor
- Let c_i' and c'_k be the remaining processing times of J_i and J_k
  - Then $c_i' < c_k'$ and a total of $c_i' + c_k'$ time is expended on J_i and J_k past time t
- Assume wlog that $f_i < f_k$ under S (the other case is identical)
- **Pairwise interchange**: more intricate because of preemptions
  1. Use the first c_i' time units spent on only i and k past time t to execute J_i
  2. Use the remaining c_i' time units spent on only i and k past time t to execute J_k

Optimality of SRPT for $1 \mid r_j, \text{prmt} \mid \sum f_j$

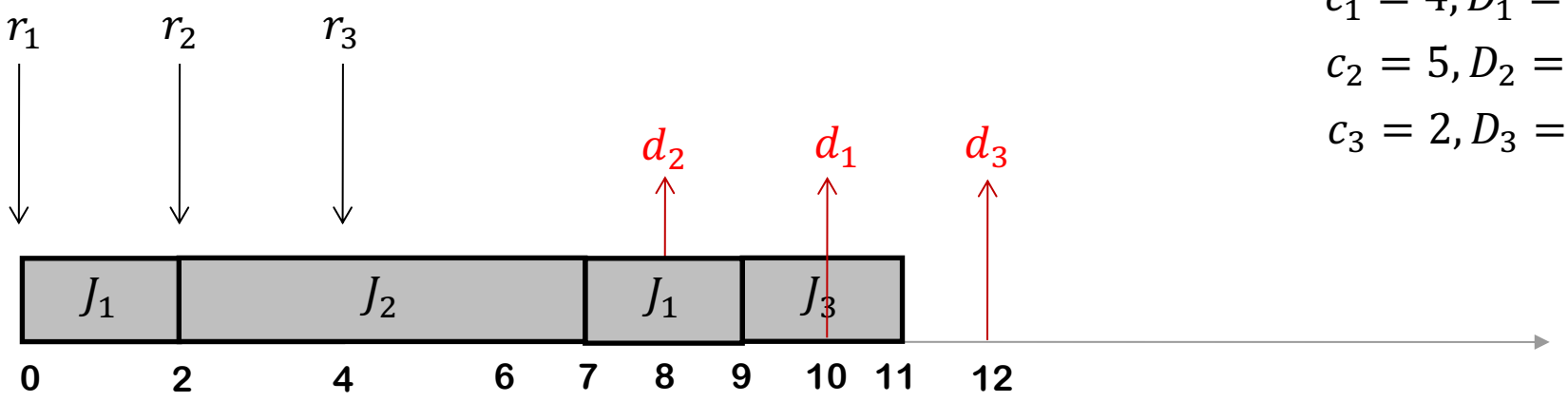
- Observe:
  - Because of preemptions, both J_i and J_k might be preempted after time t; however, exactly the total of their remaining processing times $c_i' + c_k'$ is spent on their combined execution after time t
  - The interchange of "execution time distribution" we did does *not* affect jobs other than J_i and J_k
  - In new schedule S': $f_i' < f_i$ and $f'_k = f_k$ → finish time of J_k doesn't change
  - Difference in objectives of S and S' is due only to J_i and J_k
  - $\sum_j f_j' - \sum_j f_j = (f_i' + f_k') - (f_i + f_k) = f_i' + f_k - f_i - f_k = f_i' - f_i < 0$
    - Value of objective of S' is strictly less than value of S → contradicts optimality of S

Deadline–driven Example: $1|\text{prmt}, r_j|L_{\max}$

Problem: Schedule n jobs with *arbitrary arrival times* $r_1, \ldots, r_n$ and *relative* deadlines $D_1, \ldots, D_n$ on a single processor with preemption allowed to *minimize* ***maximum lateness***

- $1|\text{prmt}, r_j|L_{\max}$ is an example of the **Graham notation** for *job* scheduling problems
- **Earliest Deadline First (EDF)** is optimal → At every time instant, schedule the available job whose *absolute deadline* $d_j = r_j + D_j$ is nearest
- **Preemptive:** if job i is running when job j arrives at time r_j and $d_j < d_i$, then remove job i from processor and start running job j, otherwise keep running job i

$$c_1 = 4, D_1 = 10$$
$$c_2 = 5, D_2 = 6$$
$$c_3 = 2, D_3 = 8$$



Optimality of preemptive EDF for $1|\text{prmt}, r_j|L_{\max}$

Problem: Schedule n jobs with *arbitrary arrival times* $r_1, \ldots, r_n$ and relative deadlines $D_1, \ldots, D_n$ on a single processor with preemption allowed to minimize **maximum lateness**

- Optimality of preemptive EDF follows by the two following claims:
- **Claim1:** $L_{\max} \geq r(S) + c(S) - d(S)$ for any $S \subset \{1, \ldots, n\}$, where $r(S) = \min_{j \in S} r_j$, $c(S) = \sum_{j \in S} c_j$, $d(S) = \max_{j \in S} d_j$ (regardless of scheduling algorithm)
- **Claim2:** preemptive EDF gives a schedule with $L_{\max}(\text{EDF}) = \max_{S \subset \{1,\ldots,n\}} \{ r(S) + c(S) - d(S) \}$

  (achieves the min possible $L_{\max}$)
- **Proof:** HW1
- Another optimal policy: Least **Laxity** First (LLF)
- **Laxity** (slack) at time t is $\ell_i(t) = d_i - t - c_i(t)$; $c_i(t)$: remaining execution-time at time t

Notes on preemptive EDF

- Is an *online* policy
- How many preemptions for n jobs?
  - equals the number of **distinct** *release times*
- Can be implemented in $O(n \log n)$

Stochastic $1||L_{\max}$

- Suppose job execution times are positive random variables on some probability space and deadlines are deterministic
- What rule minimizes the **expected max** lateness $\mathbb{E}(L_{\max}) = \mathbb{E}\big(\max(L_1, \ldots, L_n)\big)$?
- Hint: very similar to deterministic case!
- Does the same rule minimize the **max expected** lateness $\max\big(\mathbb{E}(L_1), \ldots, \mathbb{E}(L_n)\big)$?

Schedulability (feasibility) problems

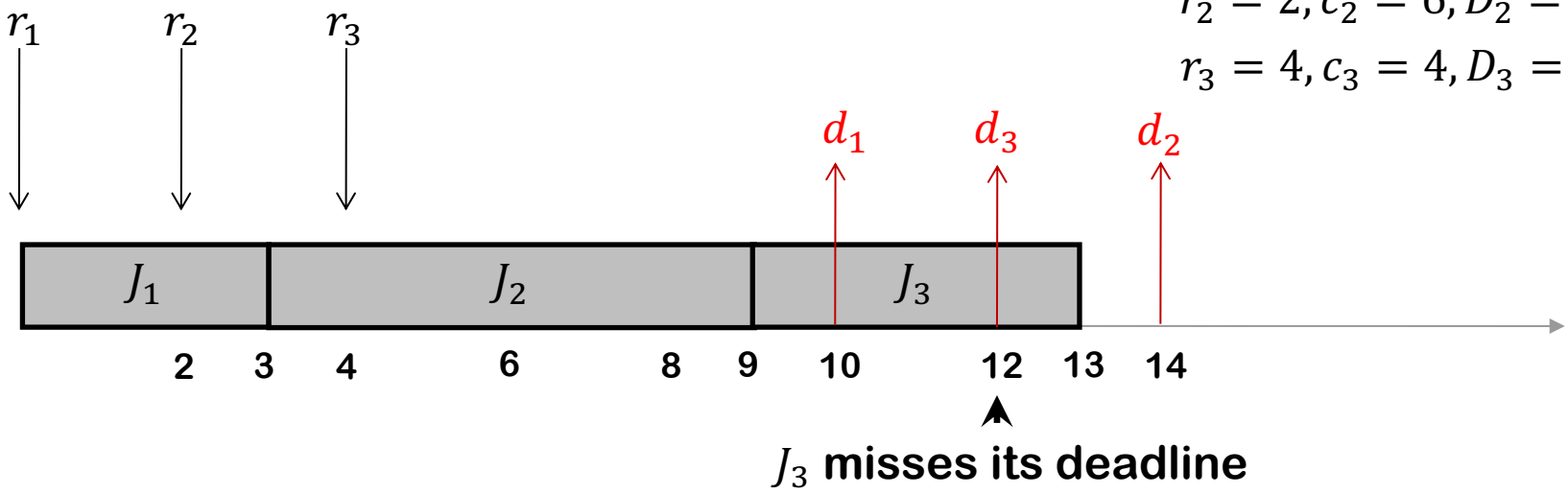
- We have n jobs $J_1, \ldots, J_n$ → J_i has parameters (c_i, r_i, D_i)
- **Problem:** *how to schedule jobs so that they meet their deadlines <u>if this is possible</u>.*
- *There is no cost function!*
  - *Such problems are called* ***feasibility*** *(or* ***schedulability****) problems*
- How do we define optimality then?
  - A rule *S* is **optimal** in the following sense: ***For every instance of the problem, it can always meet the deadlines if some other rule can***
  - *Equivalent:* ***If rule S cannot meet deadlines, then no other rule can***
  - **Formally**: Algorithm *S* is optimal for problem Π if
    $\forall S' \neq S \; \forall I \in \Pi \; (I \text{ schedulable under } S' \rightarrow I \text{ schdulable under } S)$

Non-preemptive scheduling with arbitrary release times

We have n **non-preemptible** jobs $J_1, \ldots, J_n$ → J_i has parameters c_i, r_i, D_i

- **Problem:** *how to schedule jobs **non-preemptively** so that all jobs meet their deadlines <u>if this is possible</u>.*
- Is non-preemptive EDF optimal?

$r_1 = 0, c_1 = 3, D_1 = 10$
$r_2 = 2, c_2 = 6, D_2 = 12$
$r_3 = 4, c_3 = 4, D_3 = 8$

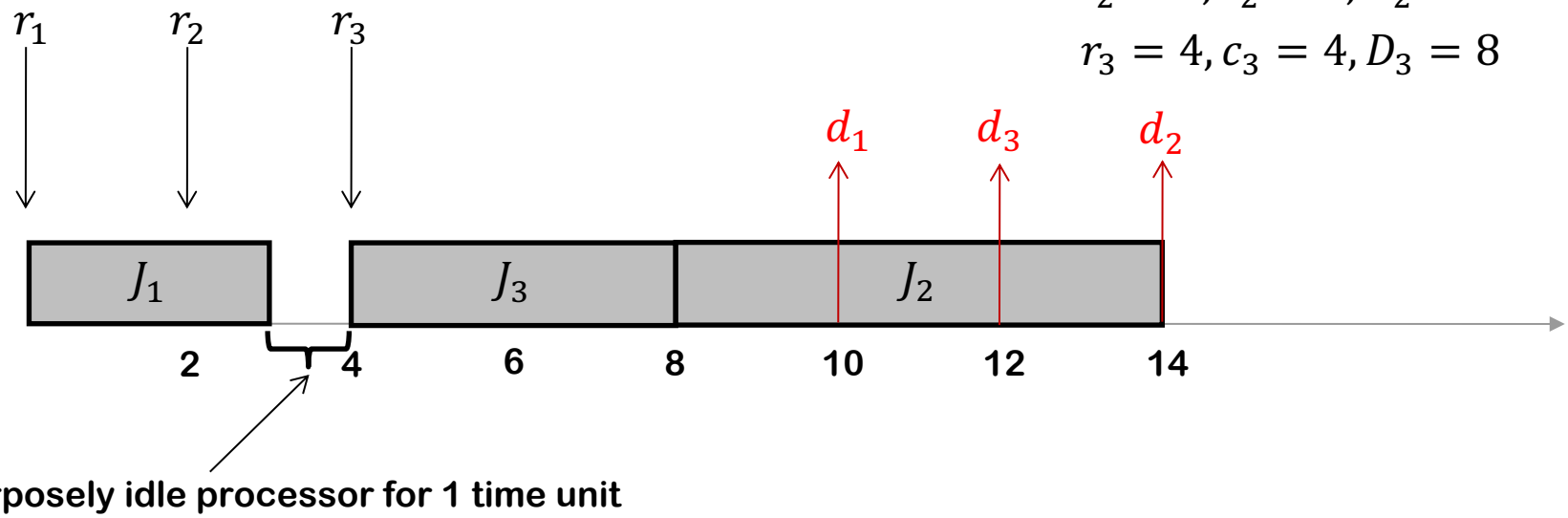


Non-preemptive scheduling with arbitrary release times

We have n **non-preemptible** jobs $J_1, \ldots, J_n$ → J_i has parameters c_i, r_i, D_i

- **Problem:** *how to schedule jobs* ***non-preemptively*** *so that all jobs meet their deadlines if this is possible.*
- Is there a policy under which this job set can meet its deadlines?

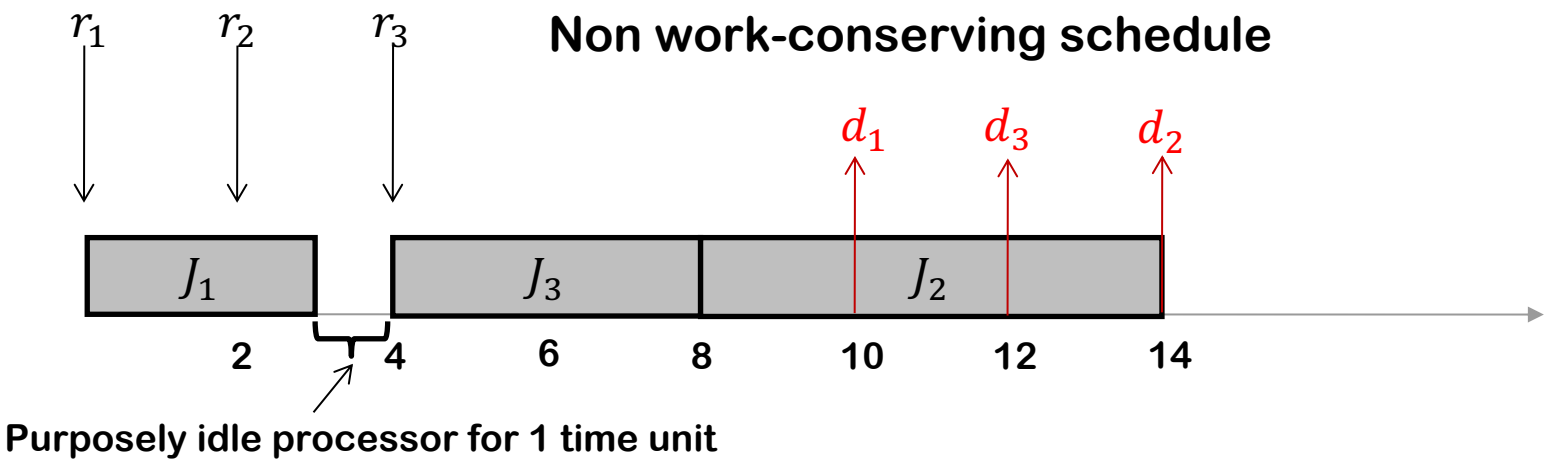
$r_1 = 0, c_1 = 3, D_1 = 10$
$r_2 = 2, c_2 = 6, D_2 = 12$
$r_3 = 4, c_3 = 4, D_3 = 8$



Non-preemptive scheduling with arbitrary release times

- This schedule introduces idle time on purpose to meet deadlines
  - Schedule is ***non****-work-conserving (not greedy, not priority-drive)*
- **This example shows more:** *no priority-driven algorithm is optimal for the non-preemptive problem with arbitrary deadlines, execution times, and release times*

$$r_1 = 0, c_1 = 3, D_1 = 10$$
$$r_2 = 2, c_2 = 6, D_2 = 12$$
$$r_3 = 4, c_3 = 4, D_3 = 8$$

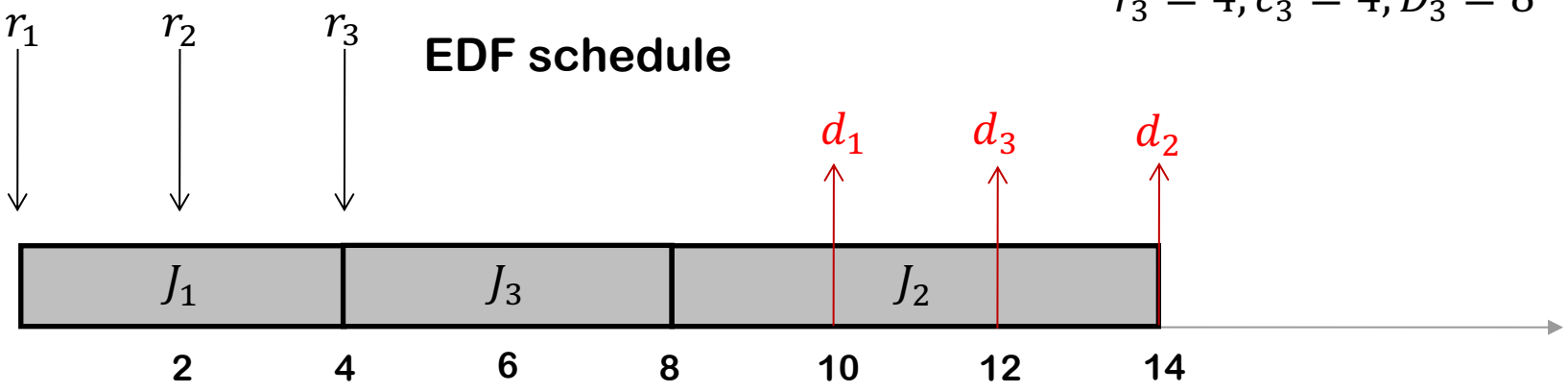


A scheduling anomaly

We have n **non-preemptible** jobs $J_1, \ldots, J_n$ → J_i has parameters c_i, r_i, D_i

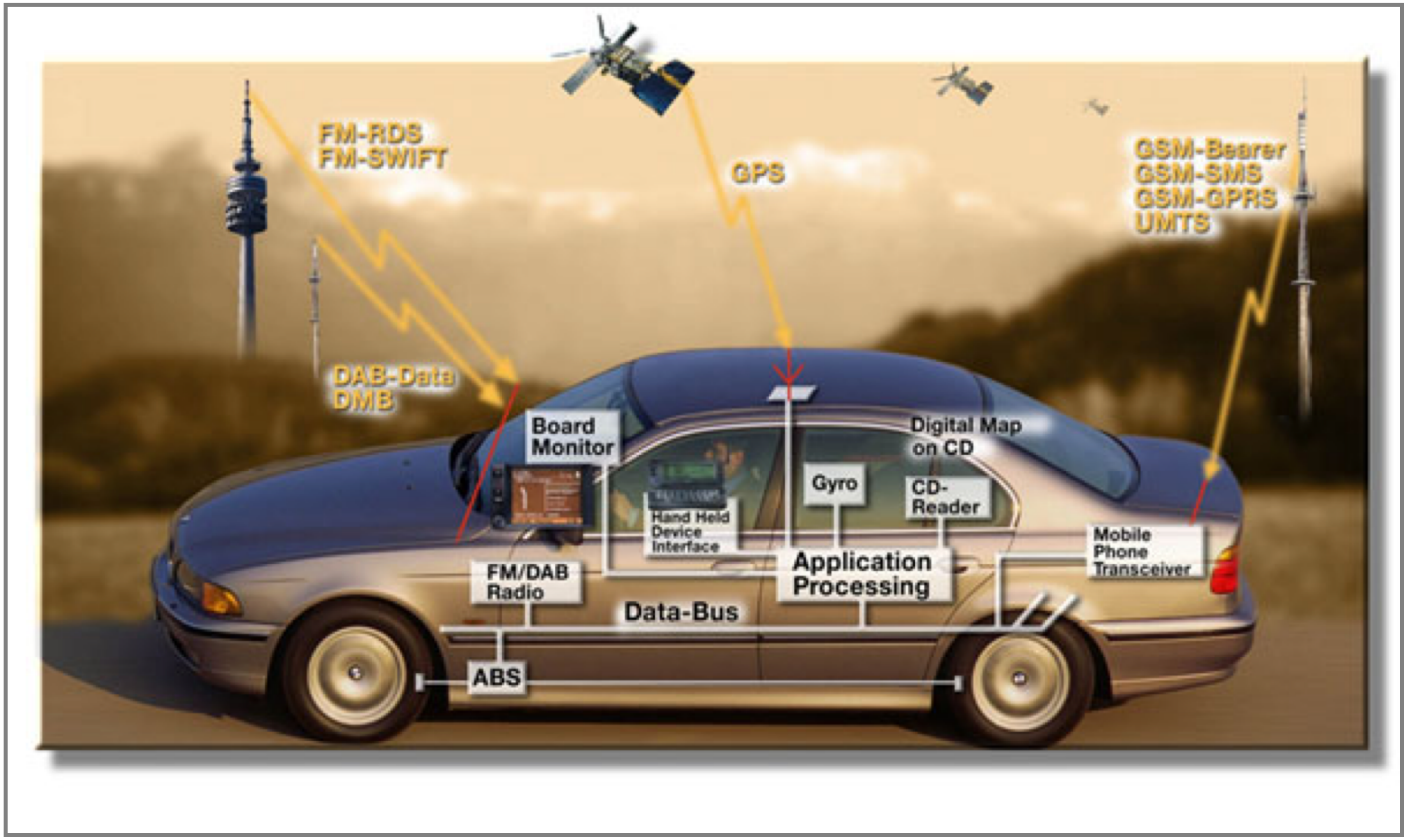
- **Problem:** *how to schedule jobs **non-preemptively** so that all jobs meet their deadlines <u>if this is possible</u>.*
- What happens under EDF if we increase c_1 from 3 to **4**?
  - ***Intuition:*** *More workload → more deadlines should be missed under same policy*
  - ***Reality:*** *All jobs meet their deadlines!*
  - ***Question:*** *Is this policy **predictable**?*

$r_1 = 0, \boldsymbol{c_1 = 4}, D_1 = 10$
$r_2 = 2, c_2 = 6, D_2 = 12$
$r_3 = 4, c_3 = 4, D_3 = 8$



Review

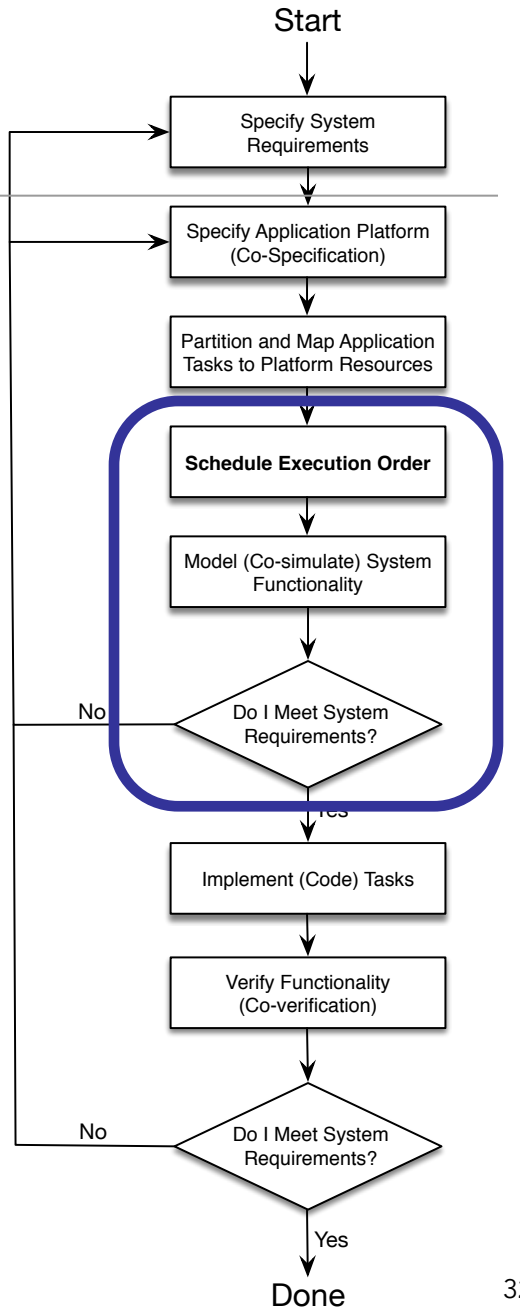
- What is a real-time system?
- What is an embedded system?
- What characteristic of a real-time system is probably the most important?



The system design process

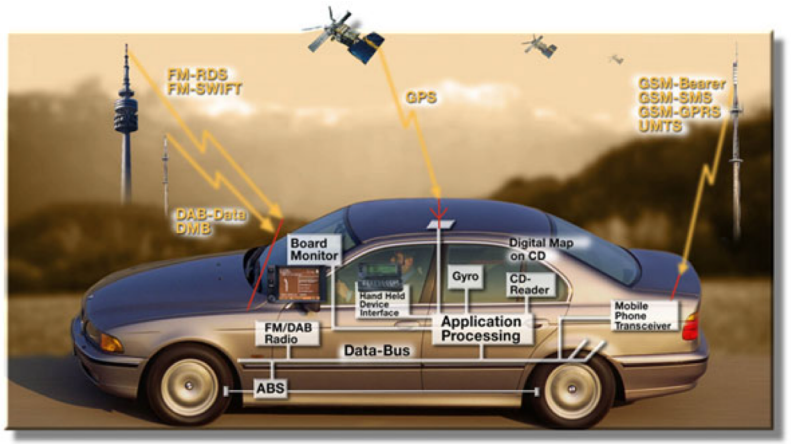
- Designing any computer system involves many steps.
- Some steps are common to many types of systems.
- A few steps are more important in a real-time system.
  - Scheduling is one such operation.
  - How do we know if a set of tasks can be scheduled in a predictable manner?

- We will touch upon other parts of the design process later in the course.



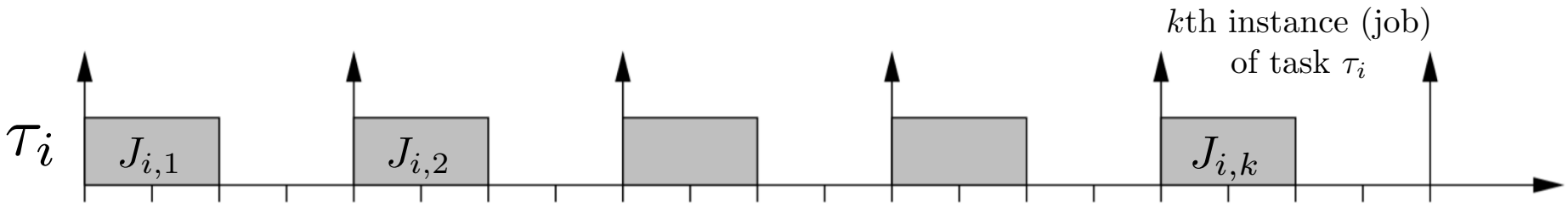
The schedulability question: Drive-by-Wire Example

- Consider a control system in a future vehicle
  - Steering wheel sampled every 10 ms – wheel positions adjusted accordingly (computing the adjustment takes 4.5 ms of CPU time)
  - Brakes sampled every 4 ms – brake pads adjusted accordingly (computing the adjustment takes 2 ms 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
- Is it possible to always compute all adjustments in time?
- The underlying computer system is a **uniprocessor** system.



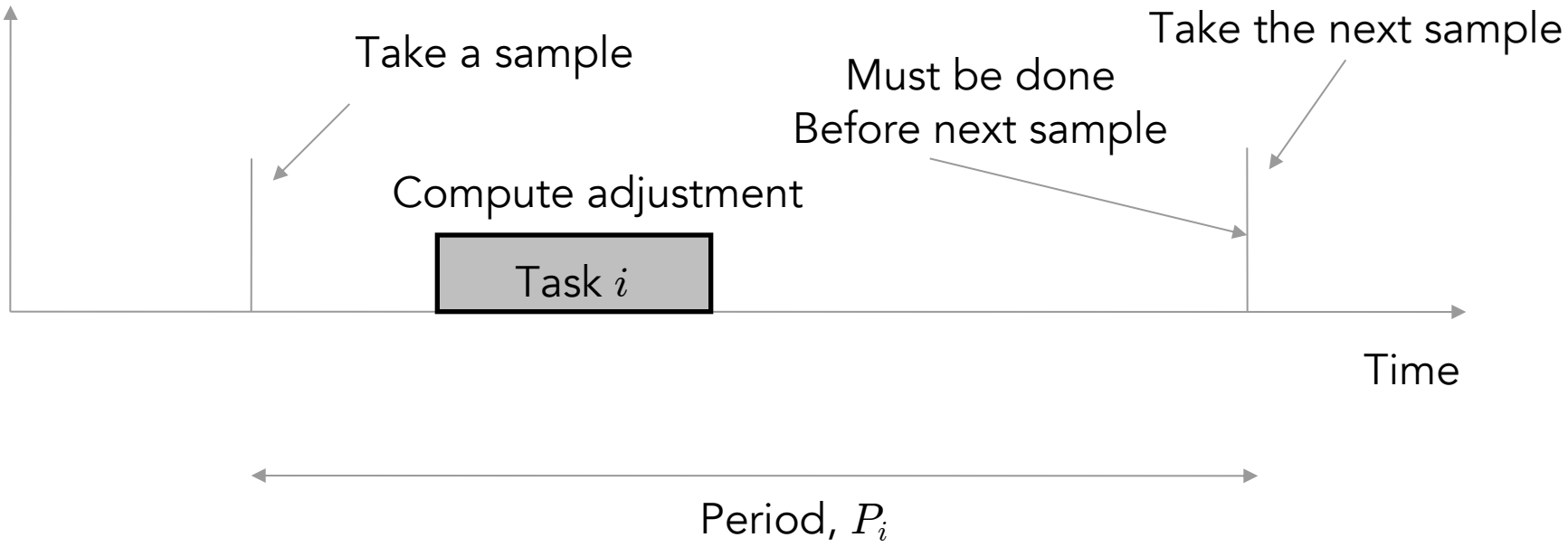
Tasks

- Basic RT model: Periodic **Task model**
  - A **task** releases an infinite (indefinitely long) succession of **jobs**



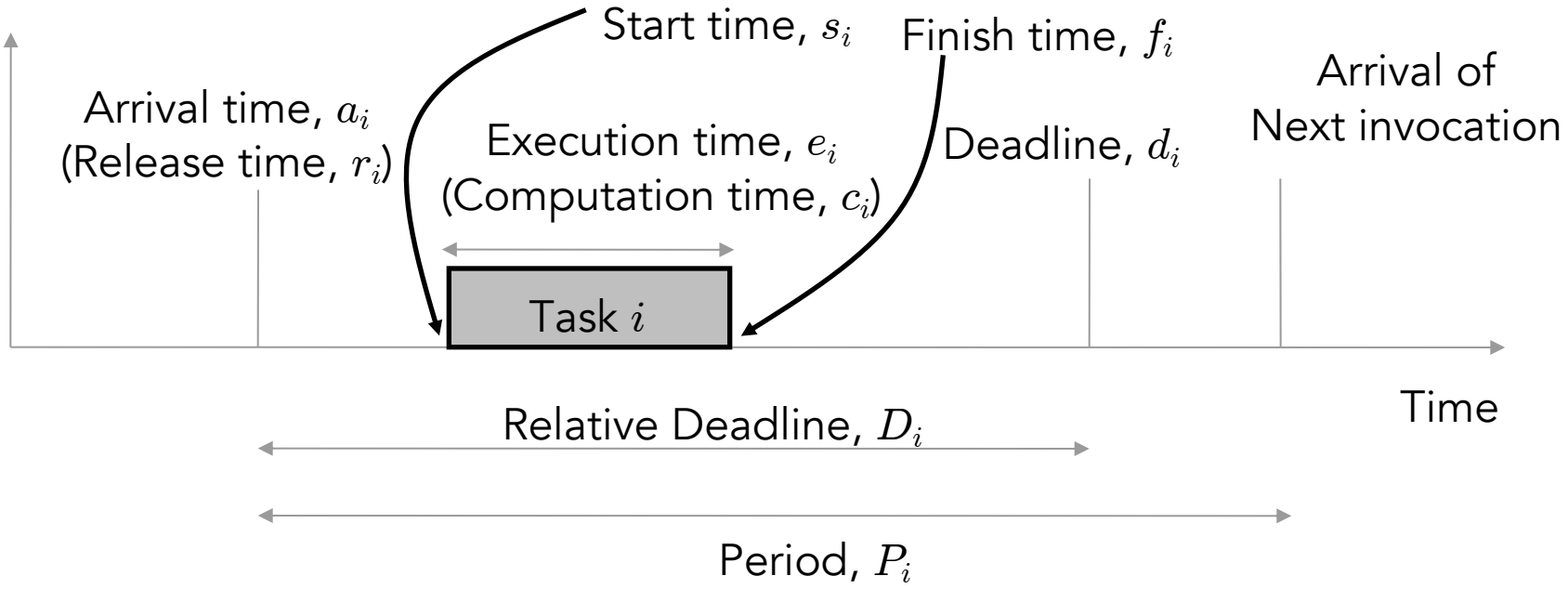
Some terminology

- Tasks, periods, arrival-time, deadline, execution time, etc.



Some terminology

- Tasks, periods, arrival-time, deadline, execution time, etc.

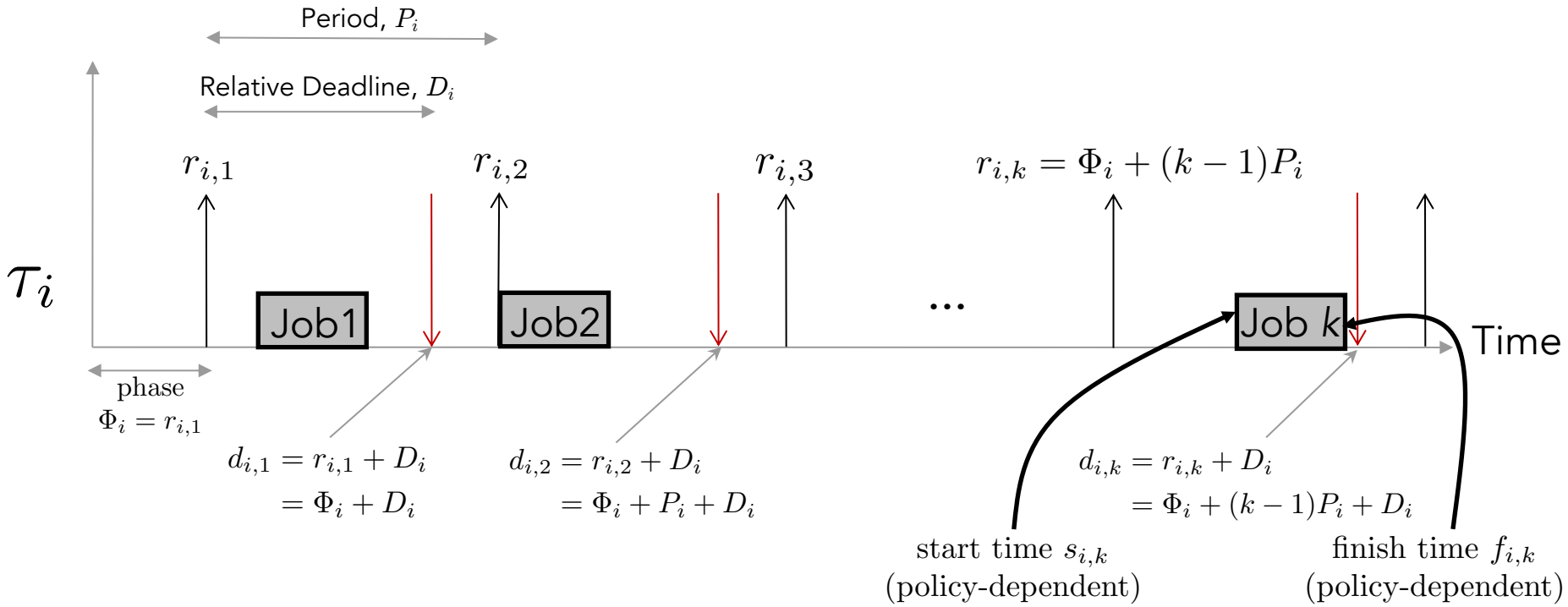


Some terminology

Tasks, periods, arrival-time, deadline, execution time, etc.

Each invocation of a task is called a "job."

A common assumption is that arrival times for the first job of all tasks is 0.

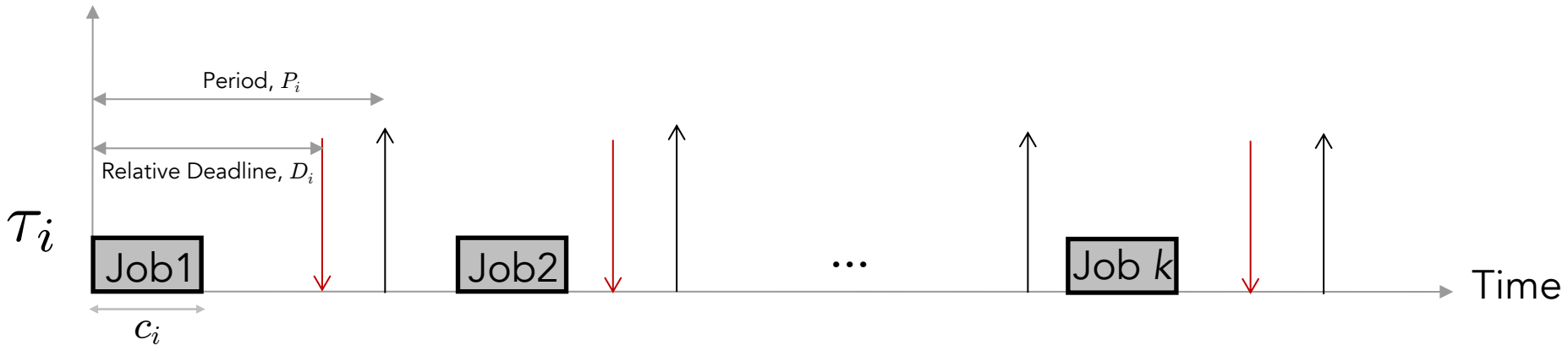


Some terminology

Tasks, periods, arrival-time, deadline, execution time, etc.

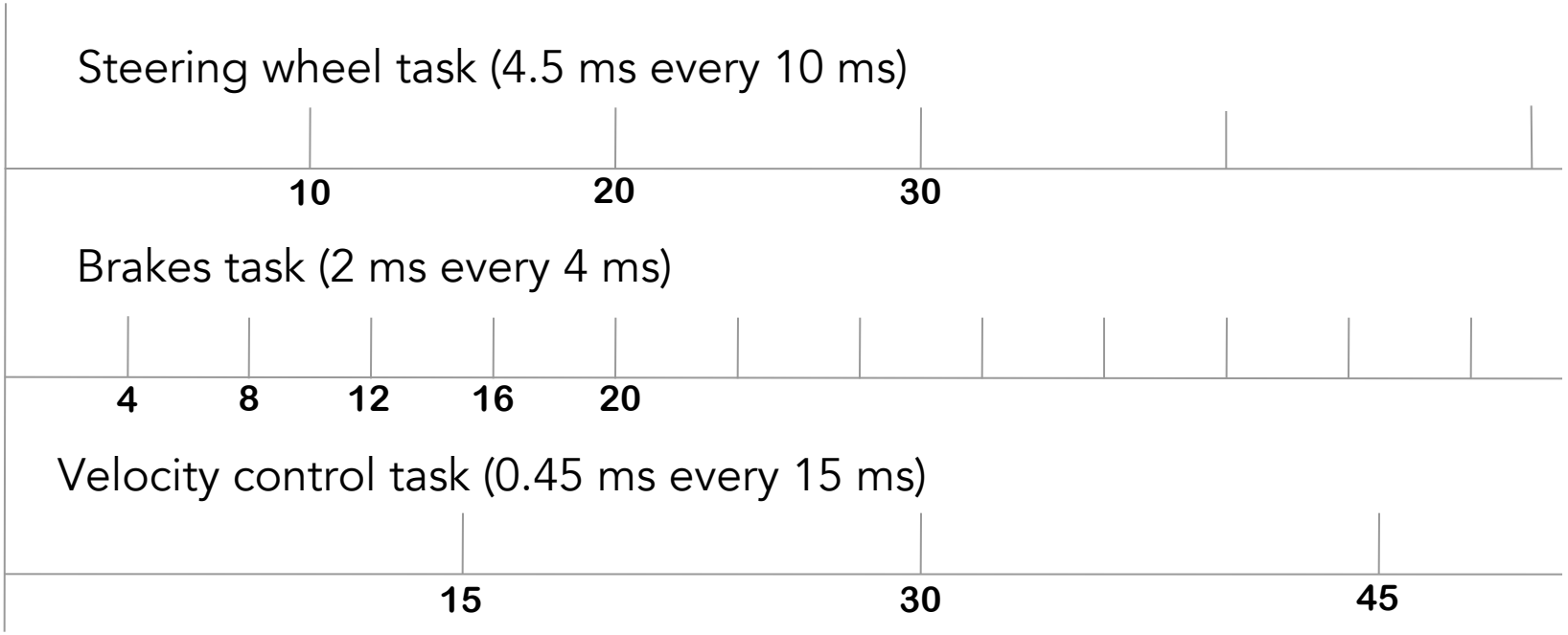
Each invocation of a task is called a "job."

A common assumption is that arrival times for the first job of all tasks is 0.



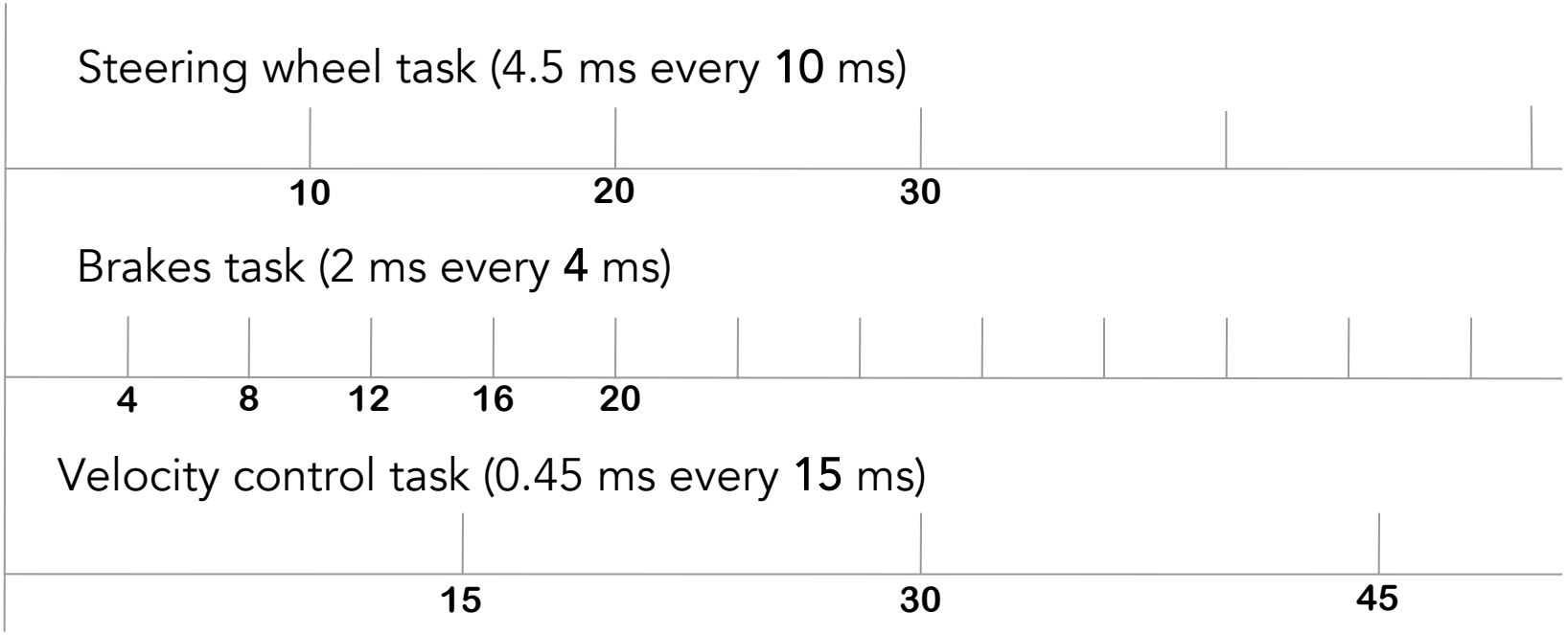
Back to the Drive-by-Wire example

- Find a schedule that makes sure all task invocations meet their deadlines
- Often, **relative deadlines are equal to the period lengths (implicit-deadline)**
- **Here each task's phase is 0**



Back to the Drive-by-Wire example

- Sanity check #1: Is the processor over-utilized? (e.g., if you have 5 assignments due this time tomorrow and each takes 6 hours, then 5x6 = 30 > 24 → you are overutilized)
  - Hint: Check if processor utilization > 100%

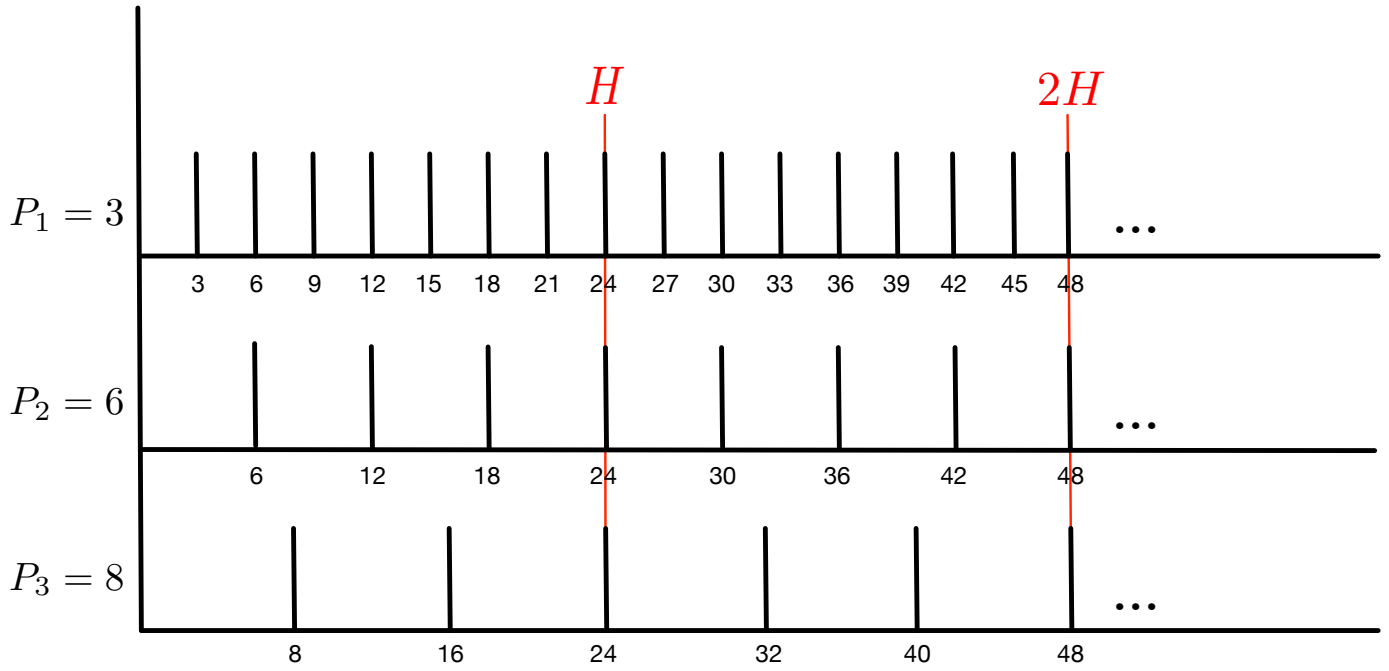


Utilization of a task set

- For a set of tasks $\{T_i\}$ with execution times $\{e_i\}$ and periods $\{P_i\}$, the utilization, U, is the fraction of time, in the long run, for which the task set will use the system.

$$U = \sum_i \frac{e_i}{P_i}$$

Why care about task utilization factor $U = \sum_i \frac{e_i}{P_i}$?



When does the **collective (system-wide)** pattern of arrivals repeat itself?

When a task arrives it *requests* c_i: $\mathbf{request_i}([\mathbf{0}, \boldsymbol{t}]) = \left\lceil \frac{\boldsymbol{t}}{\boldsymbol{P_i}} \right\rceil \boldsymbol{c_i}$

[# of requests increases by 1 at the *right limit* of kP_i]

Requests become *demands* that must be completed at absolute deadline boundaries

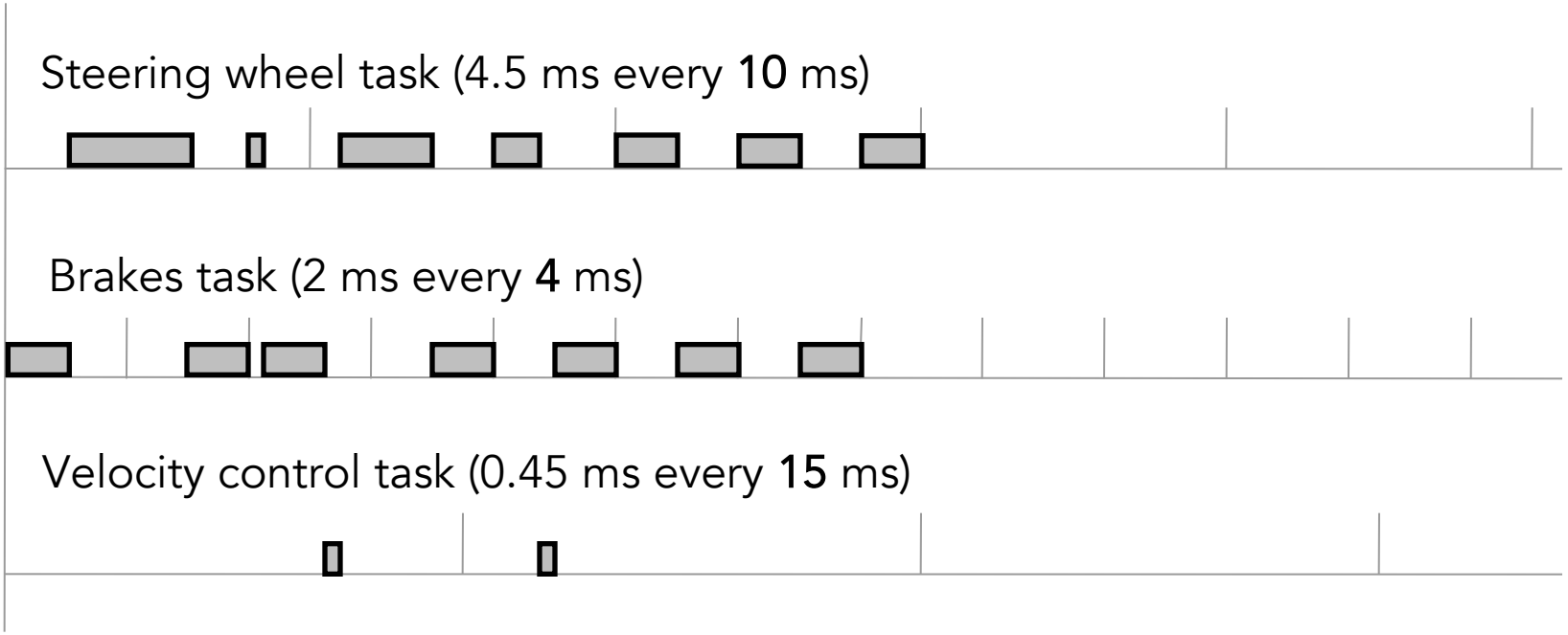
[actually at the left limit of kP_i, just before the next invocation arrives]

$$\mathbf{demand_i}([\mathbf{0}, \boldsymbol{t}]) = \left\lfloor \frac{\boldsymbol{t}}{\boldsymbol{P_i}} \right\rfloor \boldsymbol{c_i}$$

At hyperperiod boundaries: $\text{demand}_i([0,t]) = \text{request}_i([0,t]) = H/P_i$

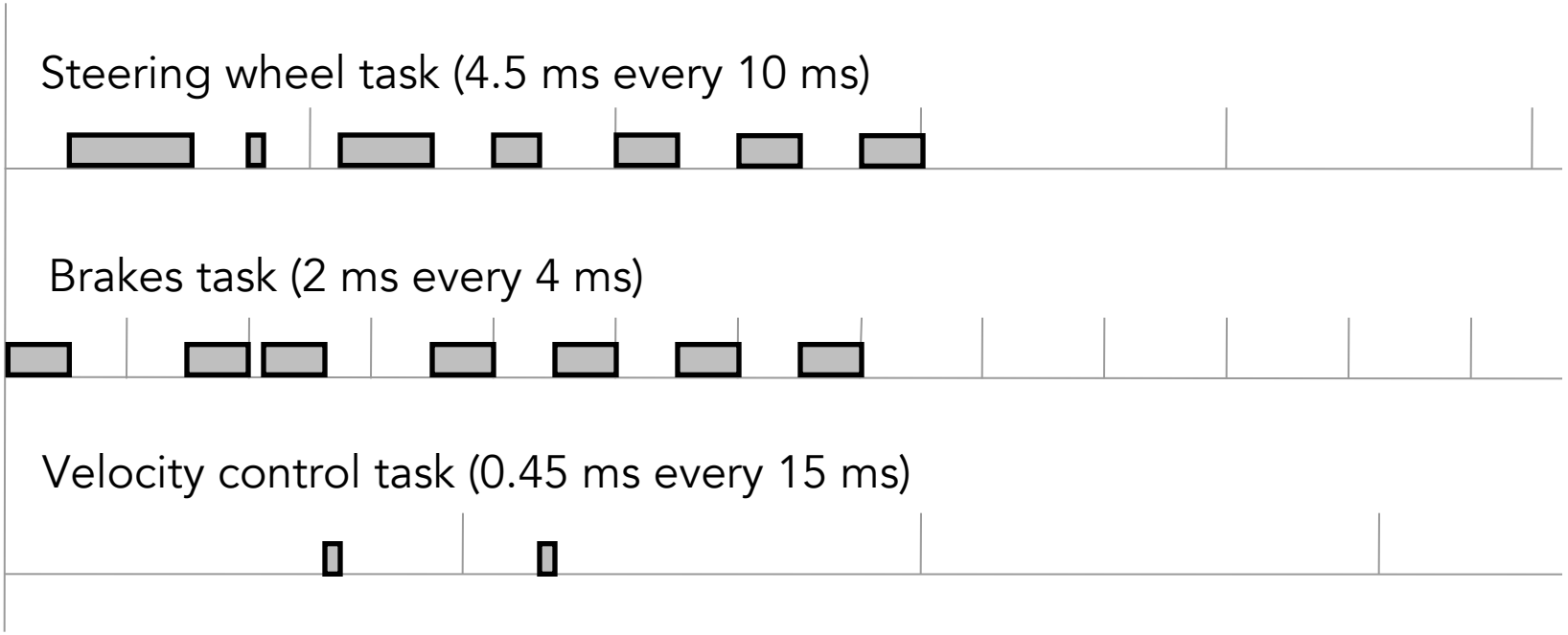
Task scheduling: Clock-driven scheduling

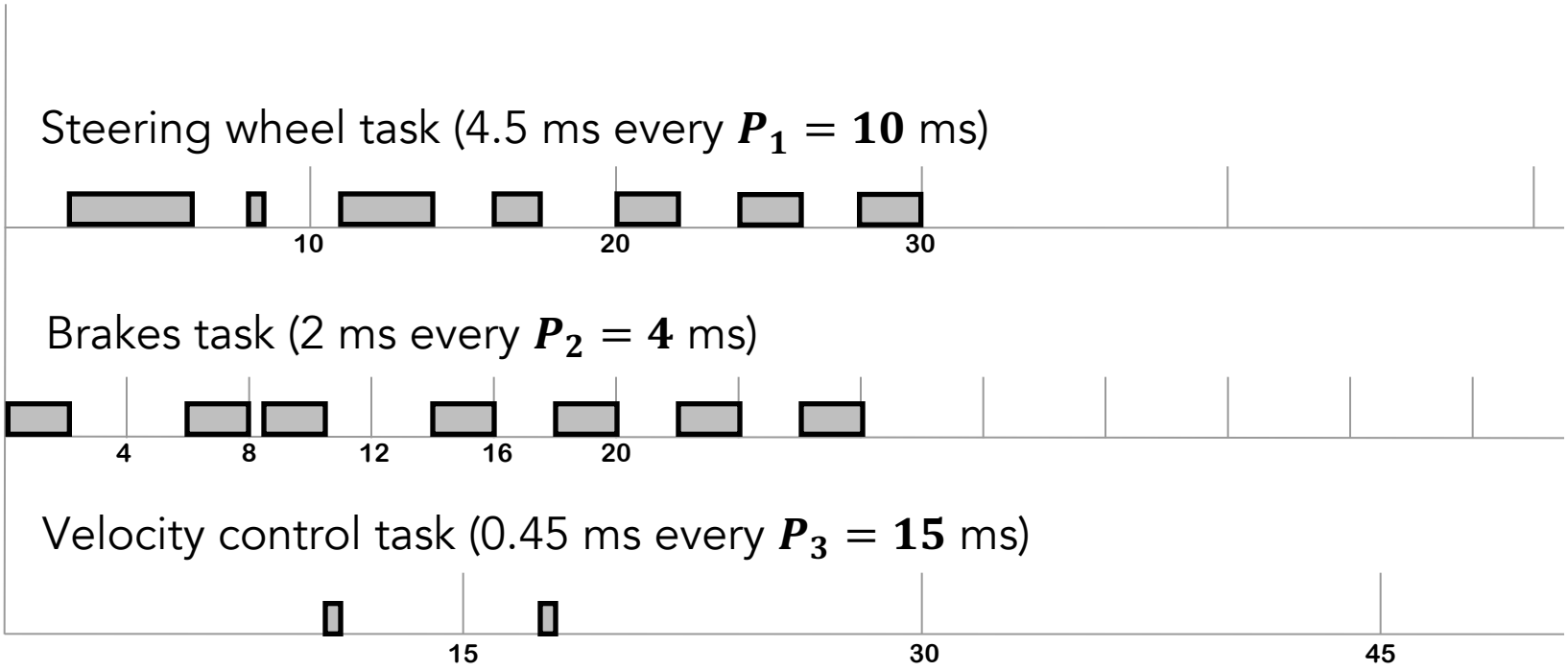
- In what order should tasks be executed?
  - Hand-crafted schedule (fill timeline by hand)
  - Cyclic executive scheduling

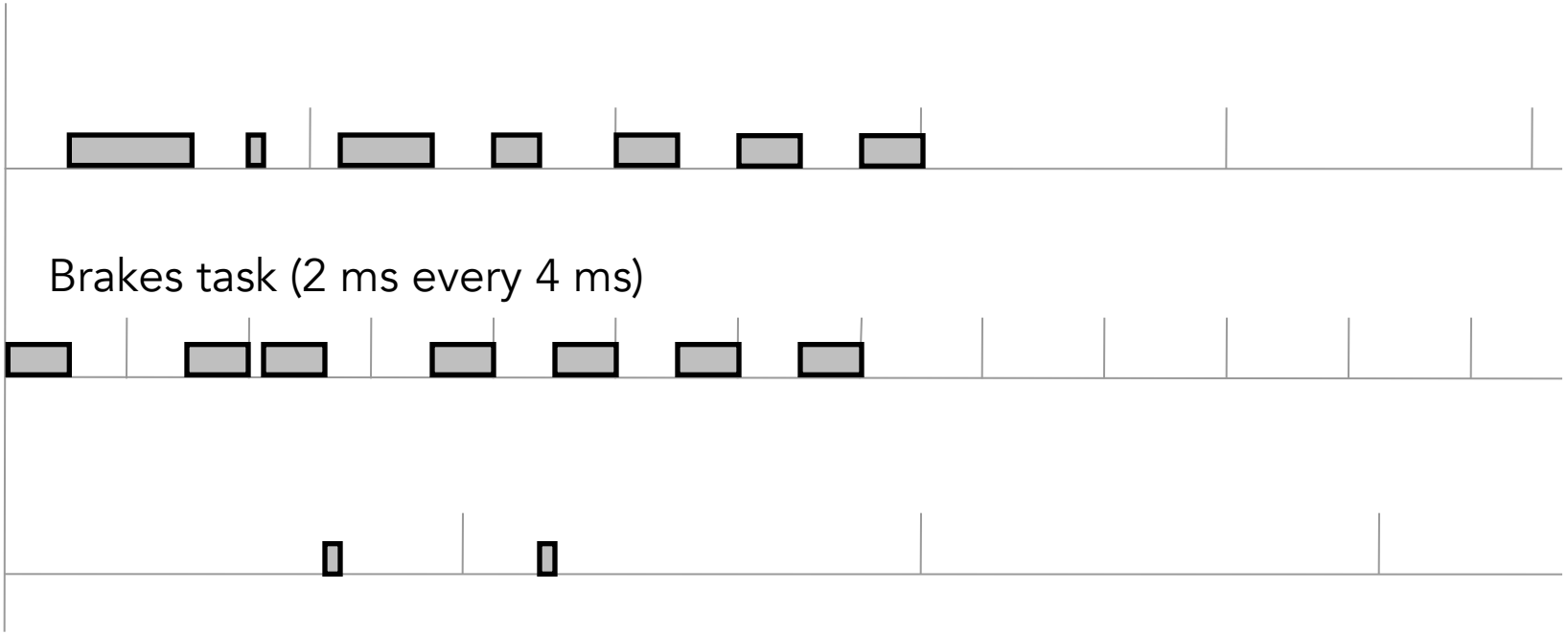


Task scheduling: Clock-driven scheduling

- Cyclic executive scheduling
  - Why is it called a "cyclic" executive?
  - What are the problems with cyclic executive scheduling?
    - Hard to adjust the schedule if tasks change
    - Difficult to specify

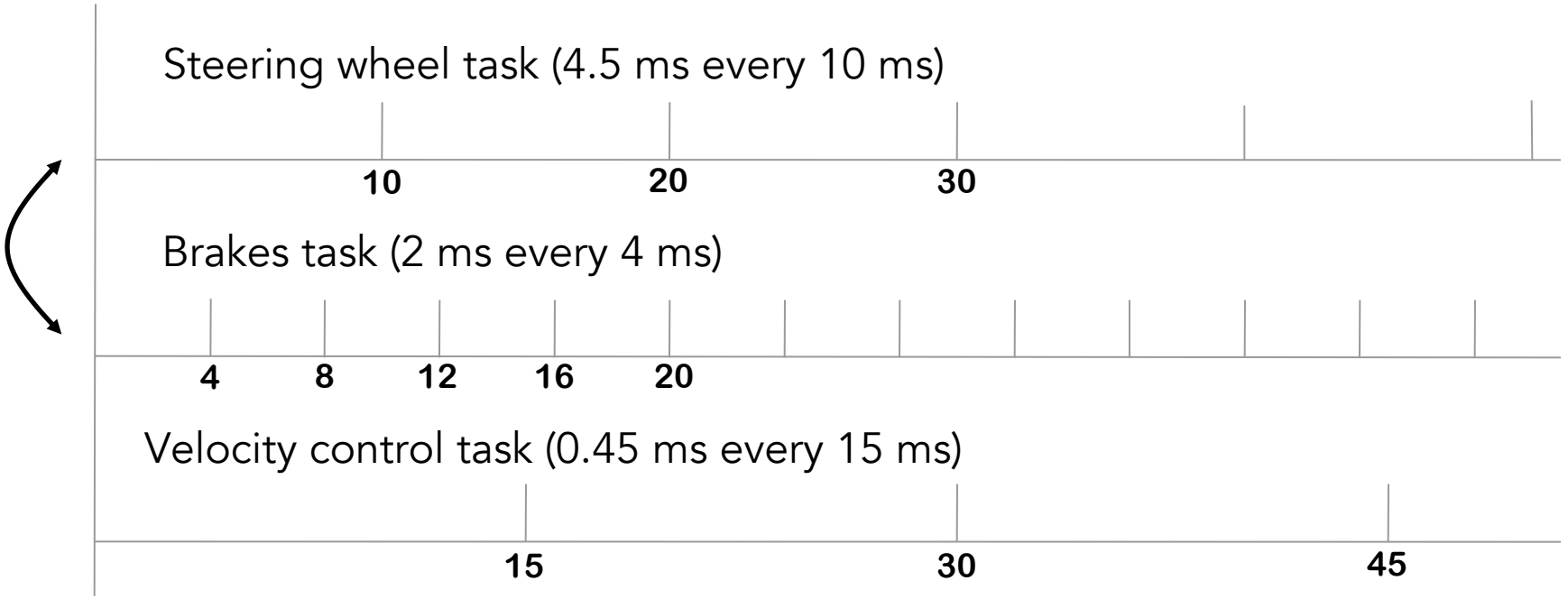



Steering wheel task (4.5 ms every $P_1 = 10$ ms)
10
20
30
Brakes task (2 ms every $P_2 = 4$ ms)
4
8
12
16
20
Velocity control task (0.45 ms every $P_3 = 15$ ms)
15
30
45


Brakes task (2 ms every 4 ms)

Task scheduling

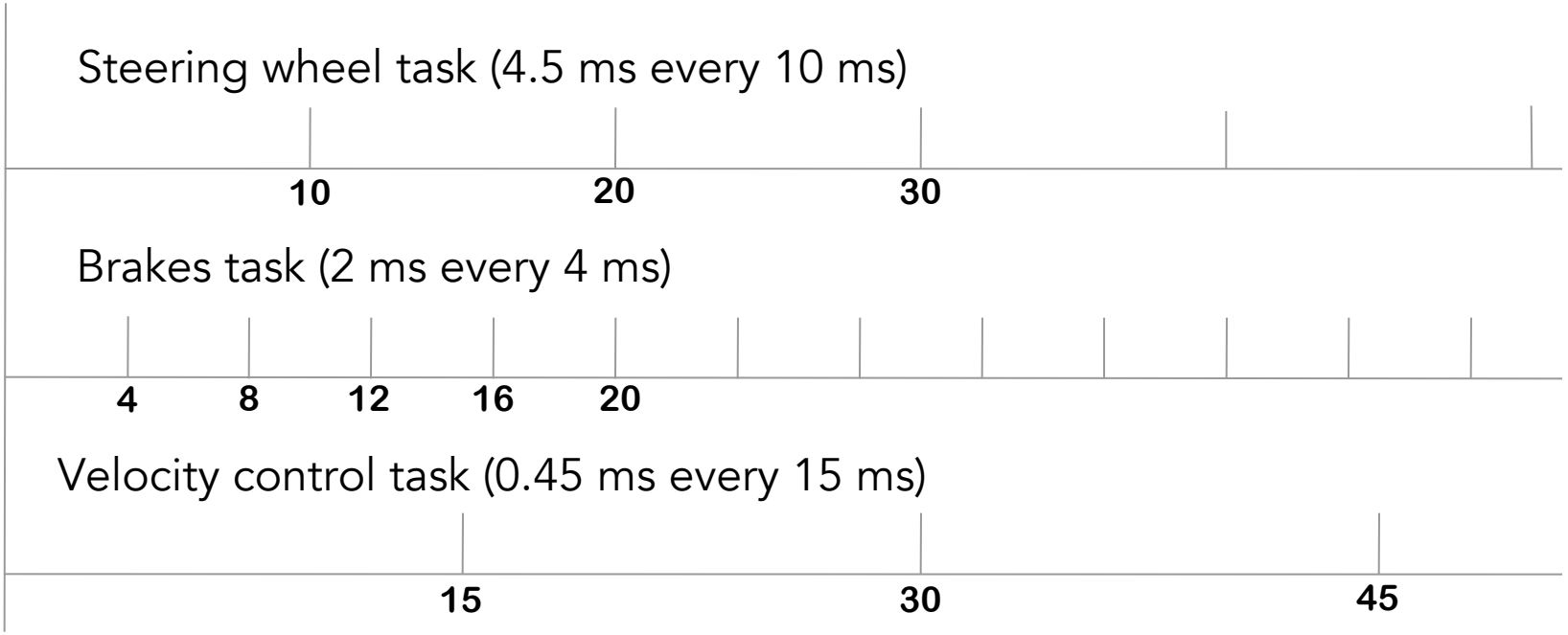
- In what order should tasks be executed?
  - Cyclic executive scheduling or
  - Priority based schedule (assign priorities; schedule is implied)



Intuition: Urgent tasks should be higher in priority

Task scheduling: **Preemptive** versus **non-preemptive**

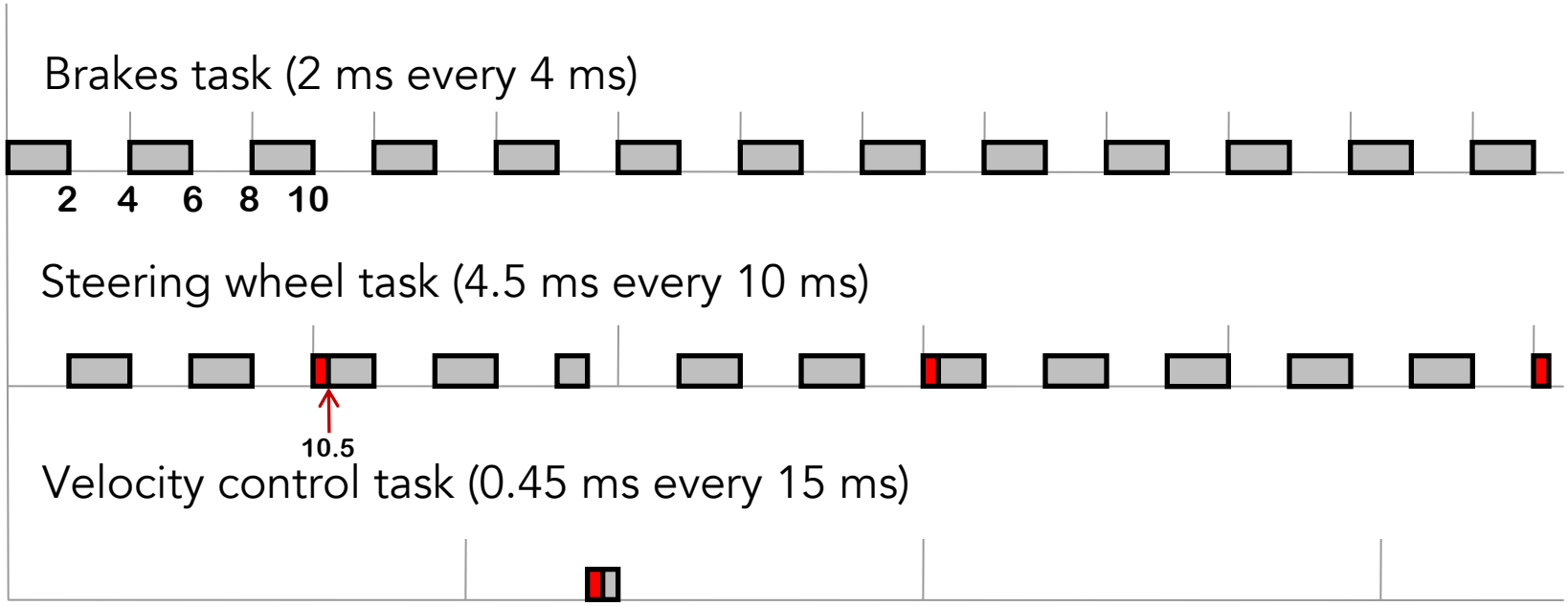
- Preemptive: Higher-priority tasks can interrupt lower-priority ones
- Non-preemptive: Can the tasks meet deadlines?



In this example, will non-preemptive scheduling work?
Hint: Compare relative deadlines of tasks to execution times of others

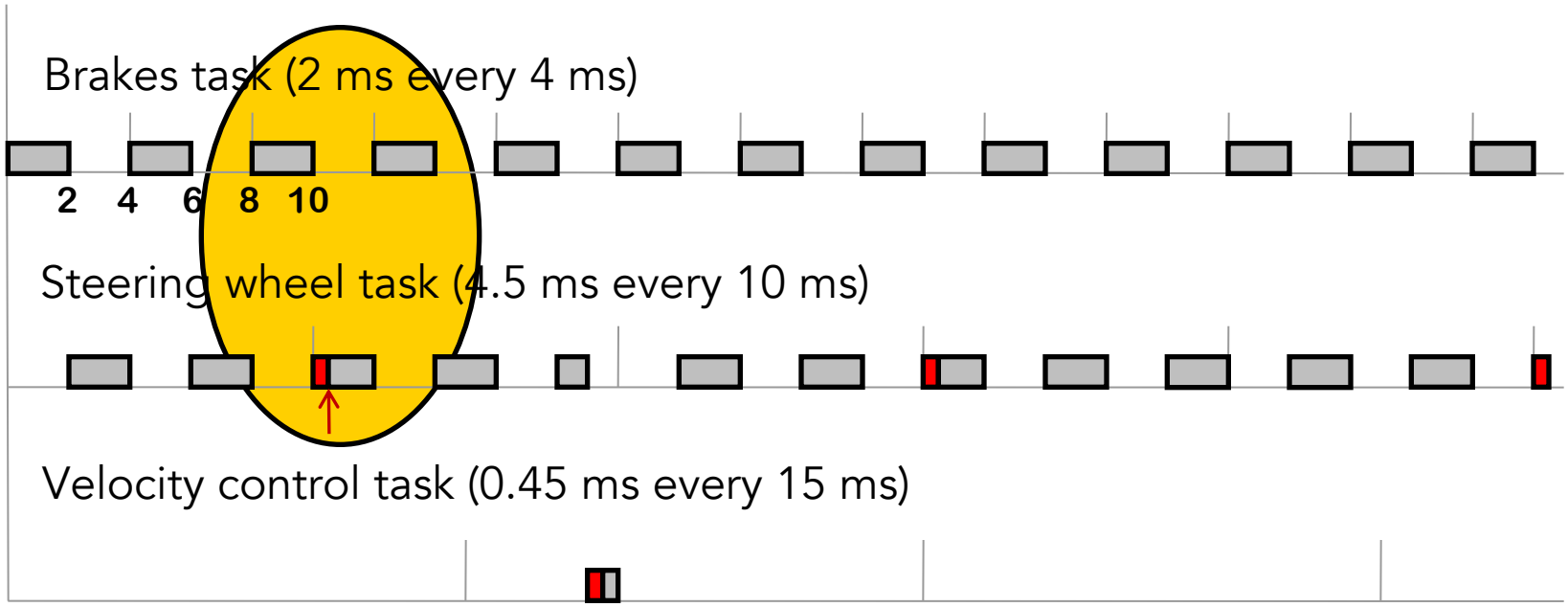
Timeline

- Even with preemption, deadlines are missed!
- Average utilization < 100%



Timeline

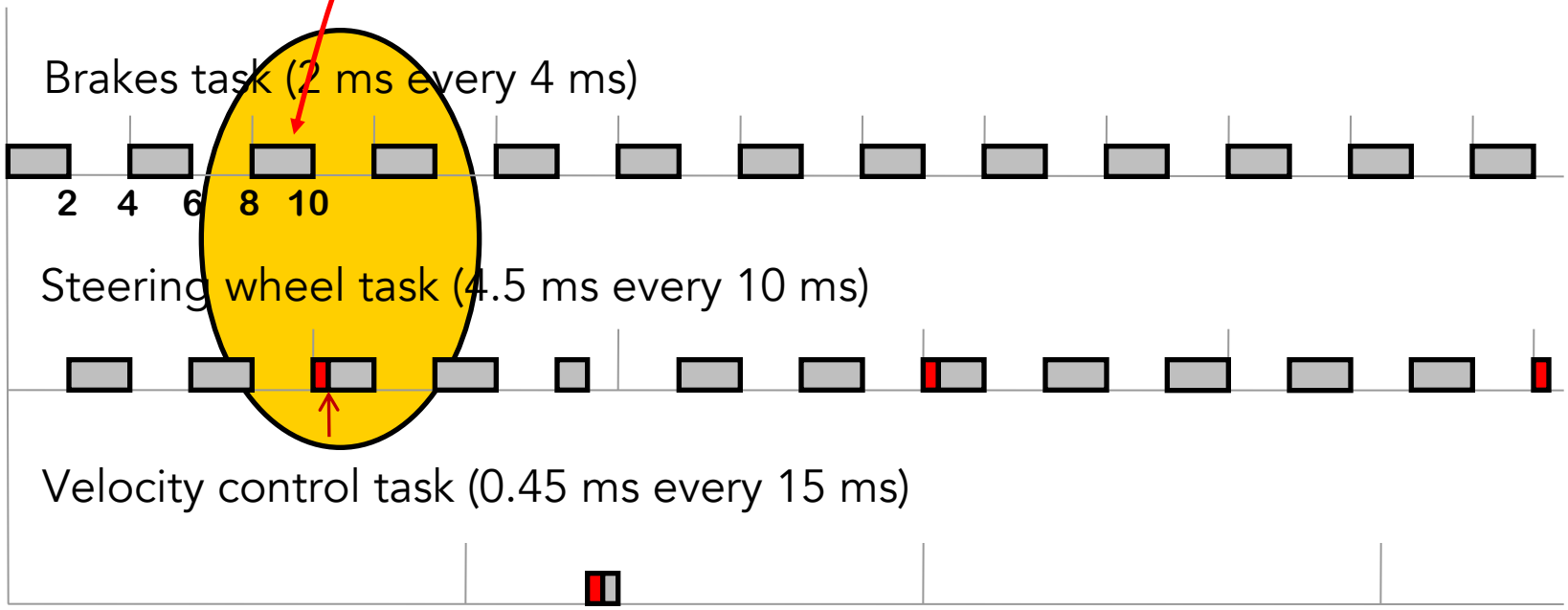
- Deadlines are missed!
- Average utilization < 100%



Timeline

- Deadlines are missed!
- Average utilization < 100%

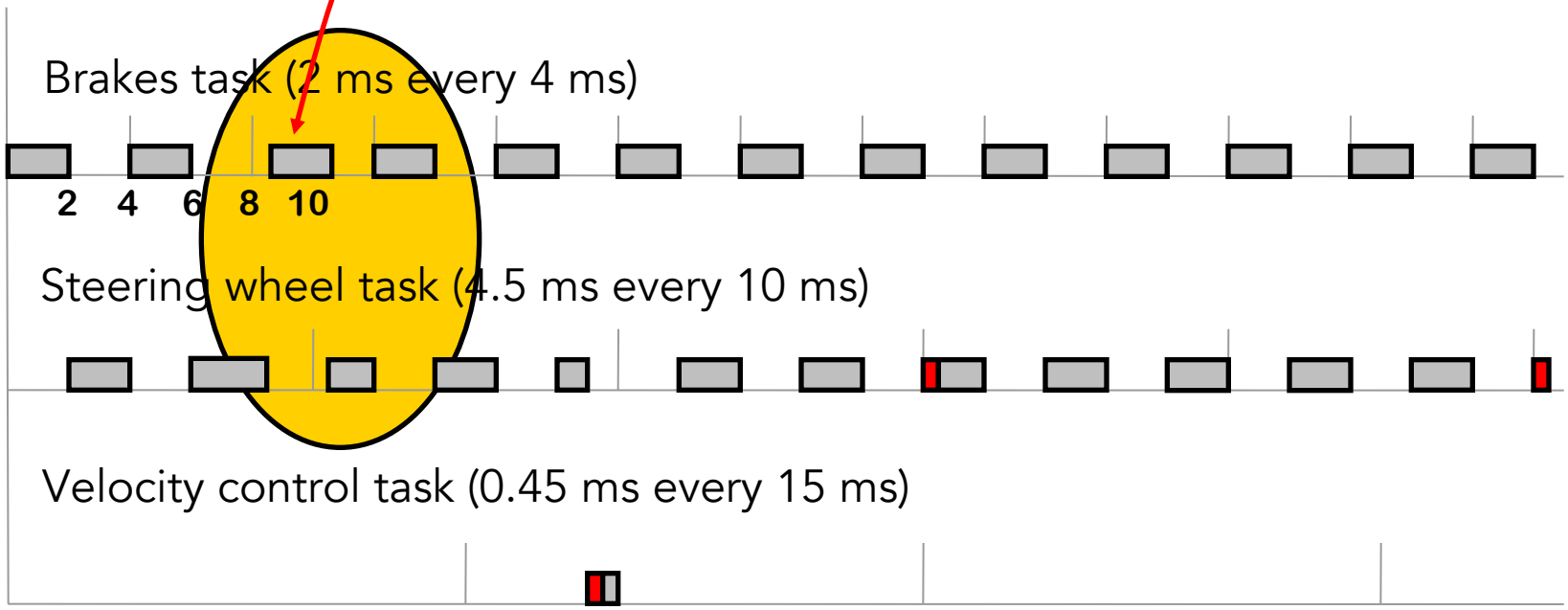
Fix:
Give this task invocation a lower priority



Timeline

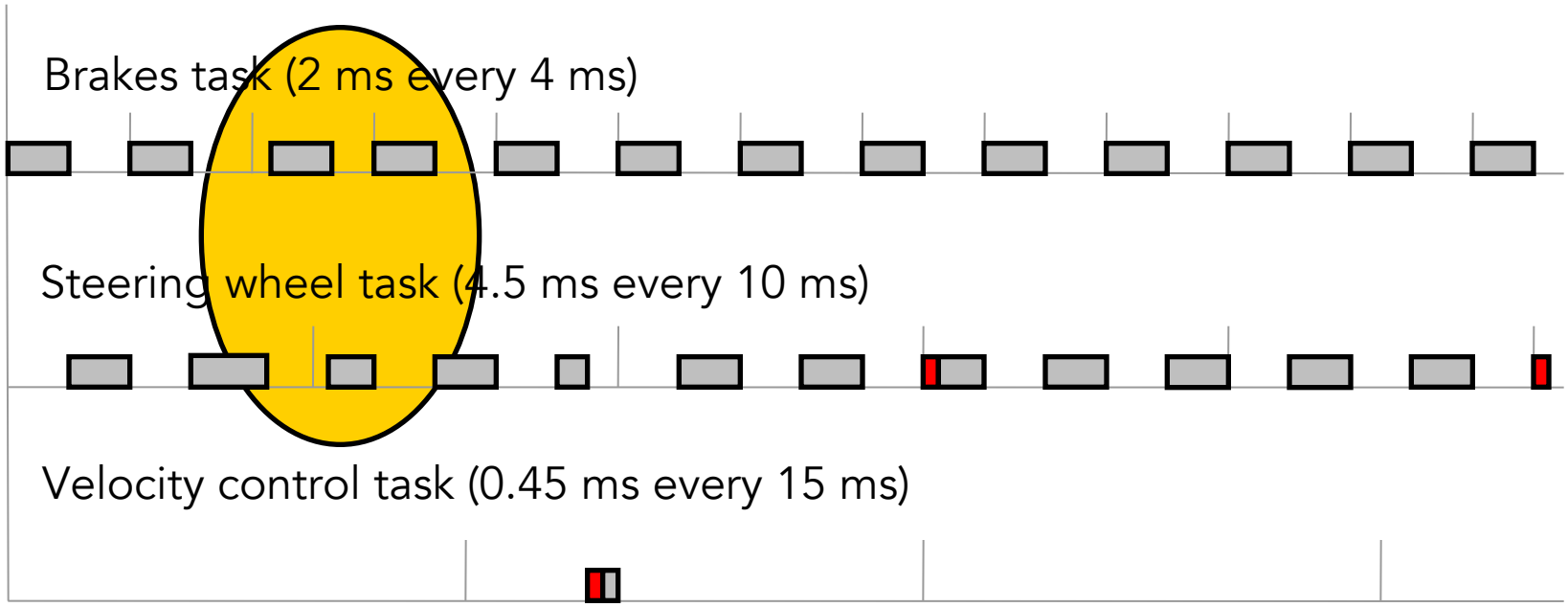
- Deadlines are missed!
- Average utilization < 100%

Fix:
Give this task invocation a lower priority



Task scheduling

- Static versus Dynamic priorities?
  - Static: All jobs (instances) of the same task have the same priority
  - Dynamic: Jobs (instances) of same task may have different priorities



Intuition: Dynamic priorities offer the designer more flexibility and hence are more capable of meeting deadlines

Examples of policies

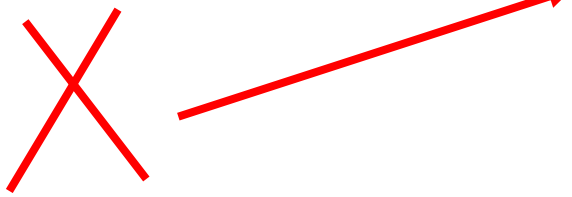
- Static-task priority policies
  - **Rate monotonic priority:** tasks with shorter periods (higher frequency or **rate**) get higher priority
  - **Deadline monotonic priority:** tasks with shorter relative deadlines get higher priority
  - Rate monotonic priorities and deadline monotonic priorities are identical if relative deadlines are equal to the periods
- Dynamic-**task** (static-**job**) priority policies
  - **Earliest deadline first:** jobs with the earliest absolute deadline take highest priority
  - **First In, First Out:** jobs with earliest arrival time take highest priority
- Dynamic-**job** priority policies: priority of ***job*** might change through time
  - **Shortest Remaining Processing Time (SRPT), Least Laxity First (LLF)**

Interesting questions

- What is the optimal dynamic priority scheduling policy? (Optimal: meets all deadlines as long as any other policy in its class can)
  - Can it meet all deadlines as long as the processor is not over-utilized? [$U \leq 1$]
- What is the optimal static priority scheduling policy?
  - When can it meet all deadlines?
  - Can it meet all deadline as long as the processor is not over-utilized?

Interesting questions

- What is the optimal dynamic priority scheduling policy? (Optimal: meets all deadlines as long as any other policy in its class can)
  - Can it meet all deadlines as long as the processor is not over-utilized? [$U \leq 1$]
- What is the optimal static priority scheduling policy?
  - When can it meet all deadlines?
  - Can it meet all deadline as long as the pr Utilization Bounds s not over-utilized?



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:

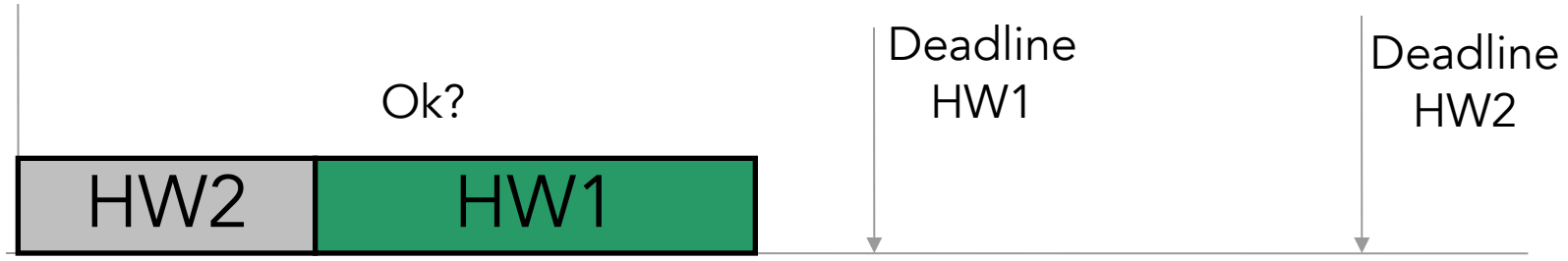
  For every $\epsilon > 0$, *there exists at least one* task set with utilization $(U_S + \epsilon)$ that **cannot** be scheduled using policy S

- A tight bound is the best (largest) 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

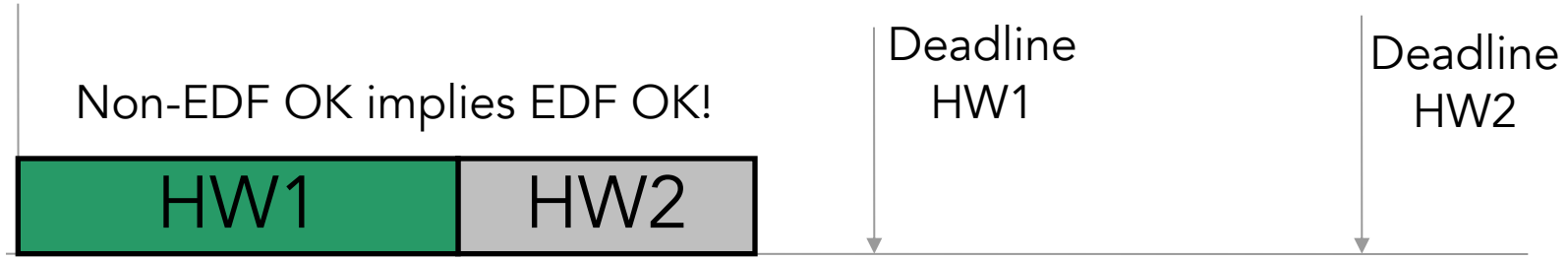
Optimality result for EDF scheduling

- EDF is the optimal dynamic priority scheduling policy
  - Priorities correspond to absolute deadlines
  - It can meet all deadlines whenever the processor utilization is less than 100%
  - Intuition:
    - You have HW1 due tomorrow and HW2 due the day after, which one do you do first?
    - If you started with HW2 and met both deadlines you could have started with HW1 (in EDF order) and still met both deadlines
    - EDF can meet deadlines whenever anyone else can



Optimality result for EDF scheduling

- EDF is the optimal dynamic priority scheduling policy
  - It can meet all deadlines whenever the processor utilization is less than 100%
  - Intuition:
    - You have HW1 due tomorrow and HW2 due the day after, which one do you do first?
    - If you started with HW2 and met both deadlines you could have started with HW1 (in EDF order) and still met both deadlines
    - EDF can meet deadlines whenever anyone else can



Why study static-priority policies?

- EDF is the optimal dynamic scheduling policy and has a utilization bound of 1.
  - The utilization bound is 1 (or 100%) when tasks have periods equal to their relative deadlines.
- EDF, however, is hard to implement in most systems.
  - Complexity is high.
  - Job queues need to be reordered often (high overhead!).
  - Most hardware subsystems allow only static priorities.

What you should know

- Definitions
  - Tasks
  - Task invocations
  - Release/arrival time,
  - Absolute deadline and relative deadline
  - Period
  - Start time and finish time
- Preemptive versus non-preemptive scheduling
- Priority-based scheduling
- Static versus dynamic priorities
- Utilization and Schedulability
- Optimality of EDF

Another example

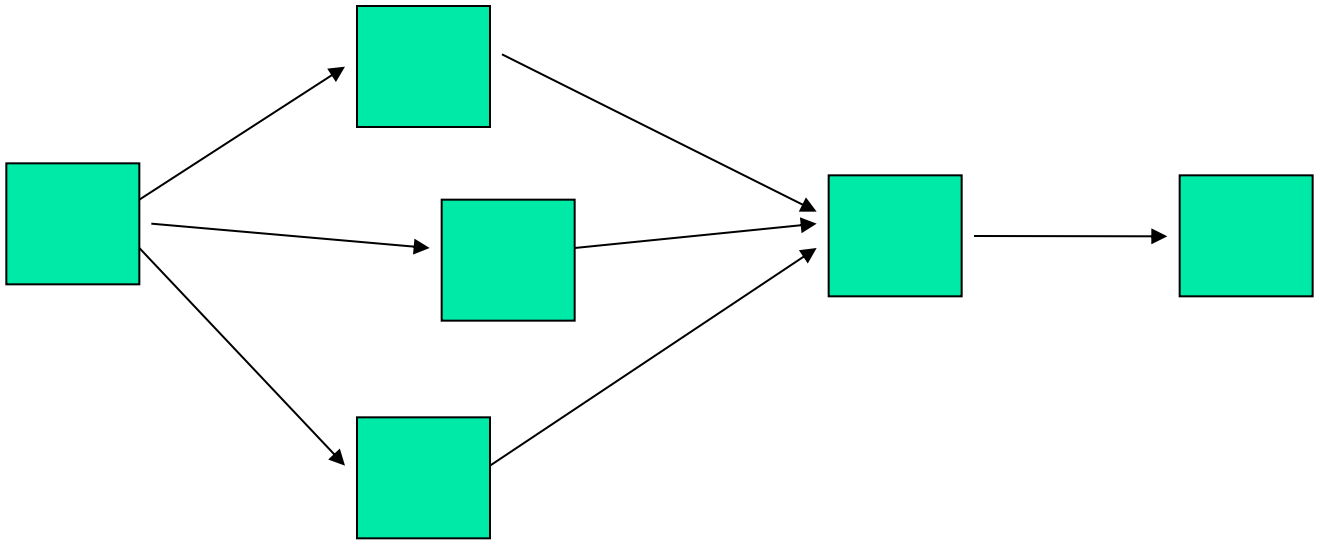
- You are the system administrator of Ameritrade.com (online stock trading site)
- You offer the following guarantee to your premium customers:
  - Stock trades of less than a $100,000 value go through in 8 seconds or you charge no commission.
  - Stock trades of more than a $100,000 value go through in 3 seconds or you charge no commission.
- Non-premium customers do not enjoy these guarantees
- Your job is to ensure that the premium customers are always served within their agreed-upon maximum latencies. What needs to be done?

For later: Aperiodic tasks

- Periodic tasks vs. aperiodic tasks
  - Aperiodic tasks may arrive at any time
  - Periodic tasks arrive at regular intervals [strictly P_i]
- **Sporadic** tasks
  - Successive arrivals have a *minimum separation distance* [greater than or equal to P_i].
- How does the lack of periodicity affect scheduling, and schedulability analysis?

For later: Precedence constraints

- In the discussion thus far, we focused on tasks that have no dependencies.
- What if tasks have precedence constraints?
  - Tasks can execute only if their predecessors have finished execution.



For later: Resource constraints

- In addition to the CPU, tasks may need resources
  - Memory
  - Disk
  - Shared data structures
- Types of resources
  - Space multiplexed: An example is the memory system. Different tasks may use different portions of the resource.
  - Time multiplexed: Only one task can access the resource at a time. An example is a data structure protected by a lock.
- How do resource constraints affect scheduling?