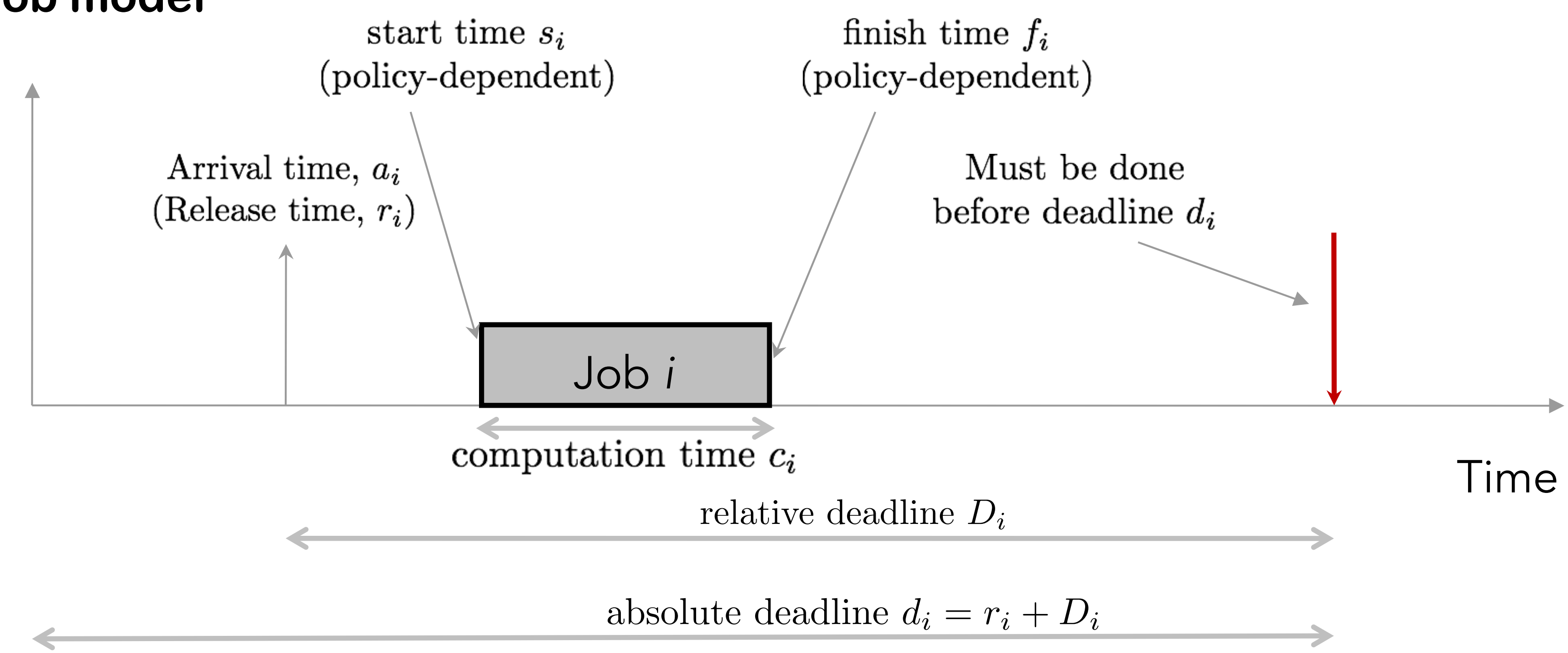
General Concepts

Introduction to real-time systems

Abstraction

- Need an abstraction to reason about timeliness that is:
  1. Expressive enough to include 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 know execution time either when job arrives or even after job finishes execution!
- In this course, mostly concerned with ***offline*** scheduling problems

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 $S: \mathbb{R}_+ \to \{1, \dots, n\}$ 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 additional 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 ready 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 may *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
- Also called **work-conserving, list scheduling, greedy** → processor is never left idle on purpose, always schedule a job if one is available

Job Scheduling: Randomized Policies

- Essentially a stochastic process $\pi = \{\pi_t : t\}$
- At $t > 0$, π_t is **a conditional probability distribution on jobs given past data**
- Past data includes **previous scheduling decisions, job allocations so far**, etc
- Intuitively, the policy throws a coin to aid in making allocation decisions
- Mostly used in stochastic job models where some job parameters are themselves random variables (later)

Job Scheduling: Objectives (cost functions)

We have n jobs $J_1, \dots, J_n$ 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

Job Scheduling: Objectives (cost functions)

- Minimize max. *flow time:* $\max_i (f_i - r_i)$
  - Schedule responsive to *each* job
  - Starvation-free
- Minimize *average* flow (*response*) time: $\frac{1}{n}\sum_{i=1}^{n}(f_i - r_i)$
  - Used in *soft* real time systems to improve QoS

$$\left.\begin{array}{l}
\text{• \textbf{Lateness}: } L_i = f_i - d_i \rightarrow \text{Minimize maximum lateness } L_{\max} = \max\limits_i L_i \\
\quad \text{• Worst case violation of deadlines. The earlier the better!} \\
\text{• \textbf{Tardiness}: } T_i = \max\{0, L_i\} \rightarrow \text{Minimize maximum tardiness } T_{\max} = \max\limits_i T_i \\
\quad \text{• Doesn't care how early (before the deadline) jobs finish} \\
\text{• \textbf{\# of tardy jobs} } \sum_{i=1}^{n} \mathbf{1}\{L_i > 0\} = \sum_{i=1}^{n} \mathbf{1}\{f_i > d_i\}
\end{array}\right\} \textbf{Deadline-related}$$

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 + how long it takes 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 scheduler → 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 $J_1, \dots, J_n$
  - J_i has execution time $e_i > 0$ and weight (importance) $w_i \geq 0$
  - All jobs arrive at time 0 → $r_i = 0 \;\; \forall i \in \{1, \dots, n\}$
  - One processor
- **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

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

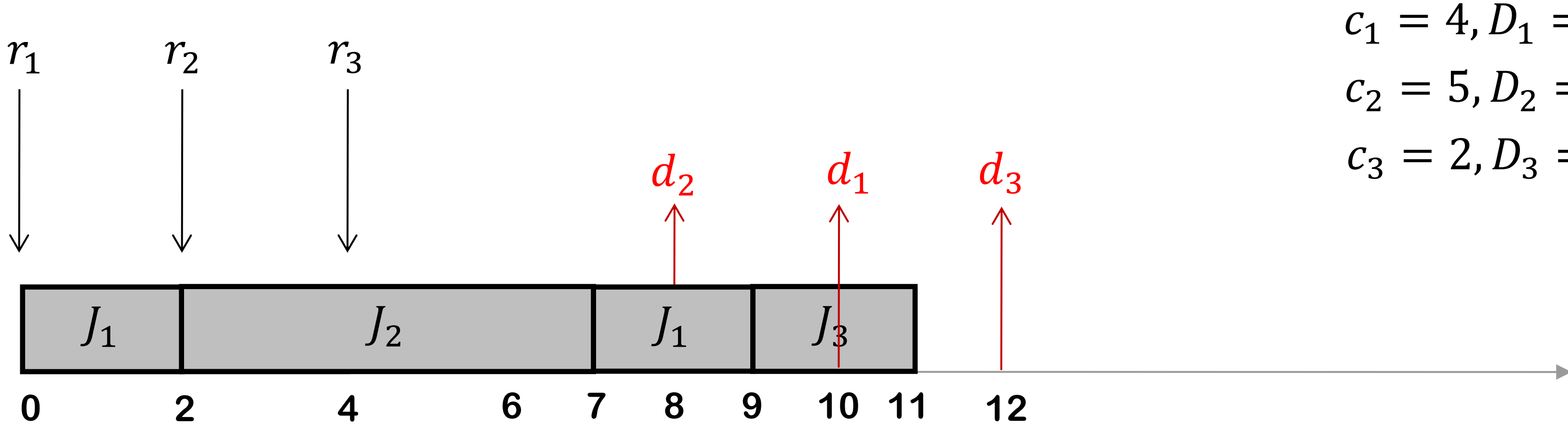
Example: Minimize total weighted completion time

- 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

Example: $1|\mathrm{prmt}, r_j|L_{\max}$

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

- $1|\mathrm{prmt}, r_j|L_{\max}$ is an example of the **Graham notation** for *job* scheduling problems
- **Earliest Deadline First (EDF)** is optimal → At every time instant, schedule the job whose *absolute deadline* $d_j = r_j + D_j$ is nearest if job j has already arrived ($t \geq r_j$)
- **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* $r_1, \dots, r_n$ and relative deadlines $D_1, \dots, D_n$ 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 $S \subset \{1, \dots, n\}$, 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 $L_{\max}(\text{EDF}) = \max_{S \subset \{1,\dots,n\}} r(S) + c(S) - d(S)$

  (achieves the min possible $L_{\max}$)

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 independent positive random variables on some probability space
- Which rule minimizes expected max lateness $\mathbb{E}L_{\max}$?
- Hint: very similar to deterministic case!

Schedulability problem for non-preemptive jobs with arbitrary release times

- We have n **non-preemptible** jobs $J_1, \dots, J_n$ → 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>.*
- *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 any other rule can***
  - *Equivalent: **If rule S cannot meet deadlines, then no other rule can***
- Is non-preemptive EDF optimal?

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

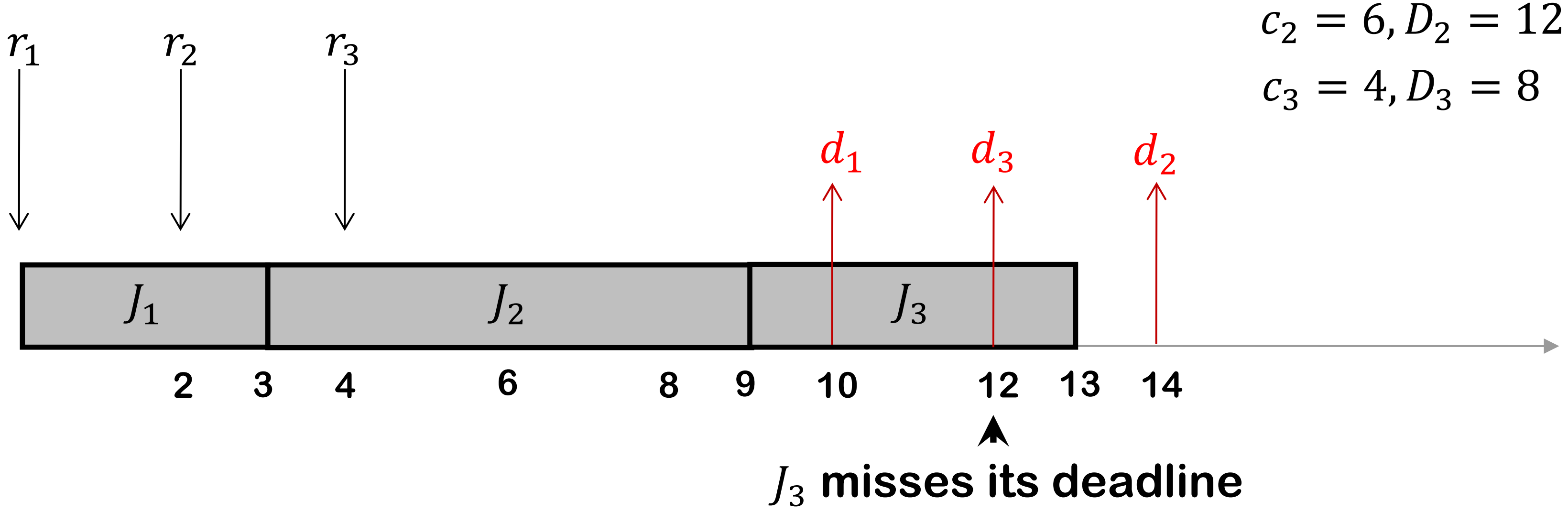
- **Problem:** *how to schedule jobs **non-preemptively** so that all jobs meet their deadlines <u>if this is possible</u>.*

- Is non-preemptive EDF optimal?

$c_1 = 3, D_1 = 10$

$c_2 = 6, D_2 = 12$

$c_3 = 4, D_3 = 8$

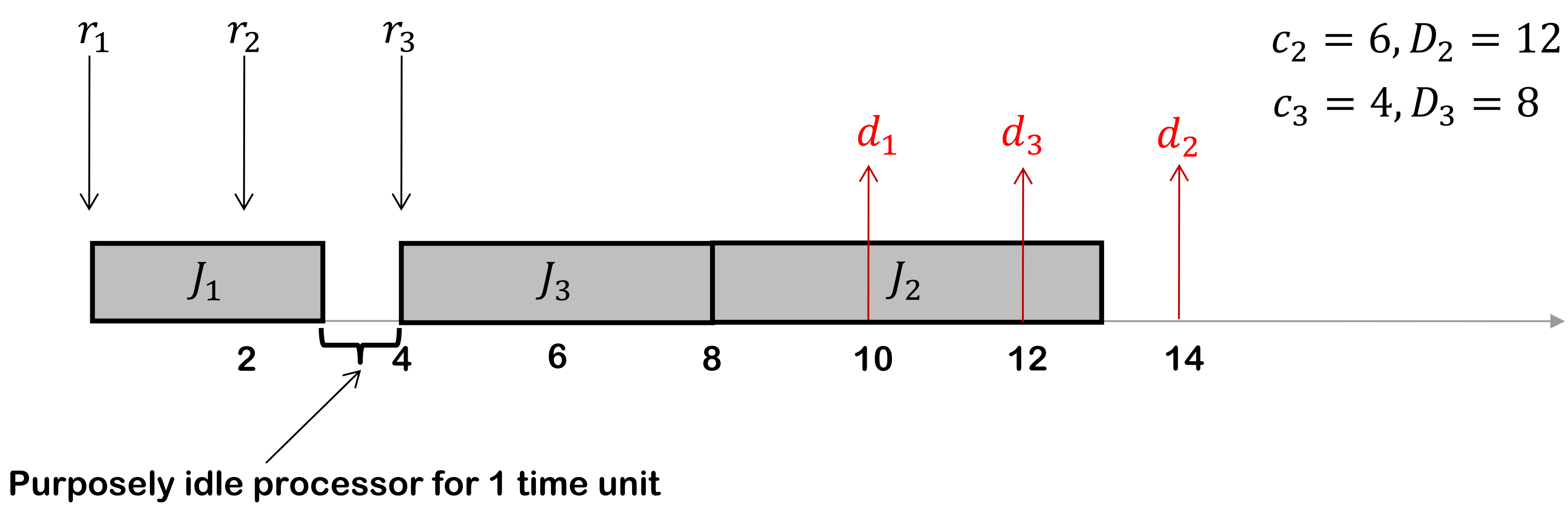


Non-preemptive scheduling with arbitrary release times

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

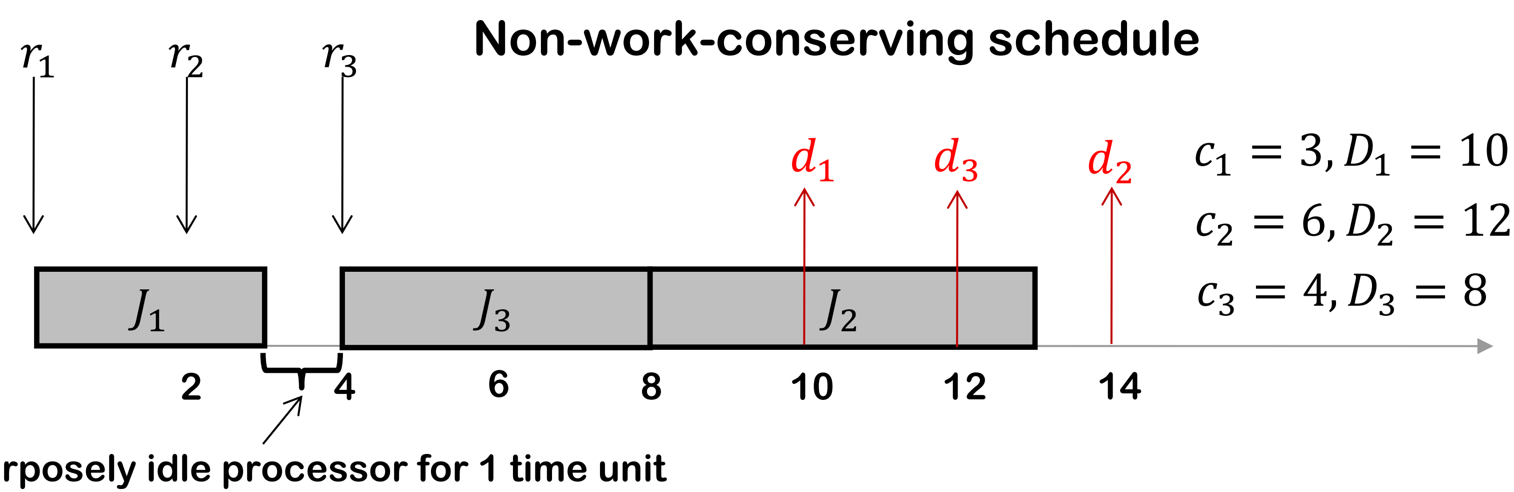
- **Problem:** *how to schedule jobs* ***non-preemptively*** *so that all jobs meet their deadlines* *<u>if this is possible</u>*.
- Is there a policy under which this job set can meet its deadlines?

$c_1 = 3, D_1 = 10$
$c_2 = 6, D_2 = 12$
$c_3 = 4, D_3 = 8$



Non-preemptive scheduling with arbitrary release times

- This schedule introduces idle time on purpose to meet deadlines
  - Schedule is ***non**-work-conserving*
- **Synonyms:** work-conserving, list schedule, greedy, priority-driven: Algorithms that *never leave the processor idle*
  - They dispatch a job to the processor as soon as one is available!
- **This example shows more:** *no priority-driven algorithm is optimal for the non-preemptive problem with arbitrary deadlines, execution times, and release times*

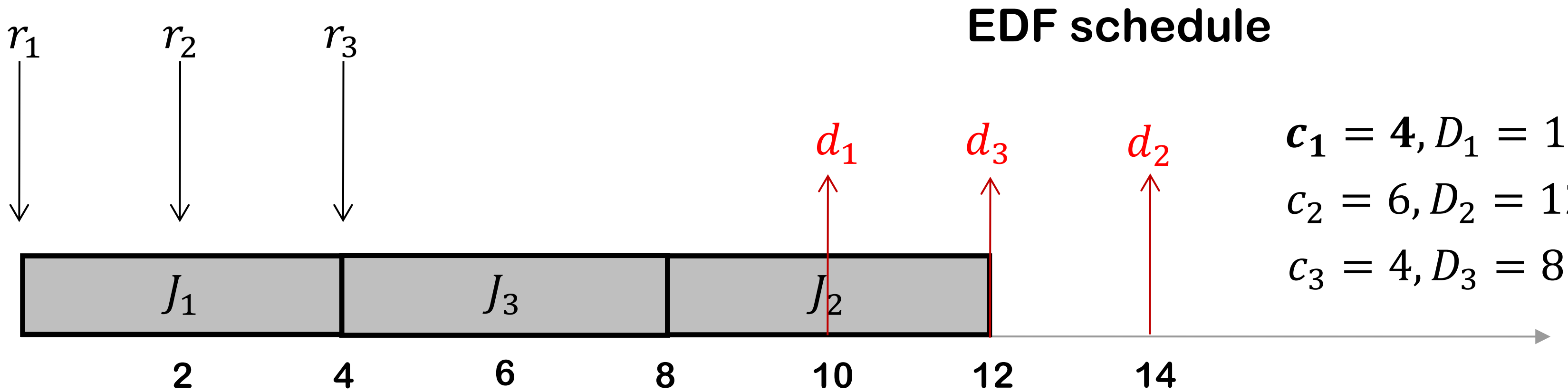


A scheduling anomaly

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****?*

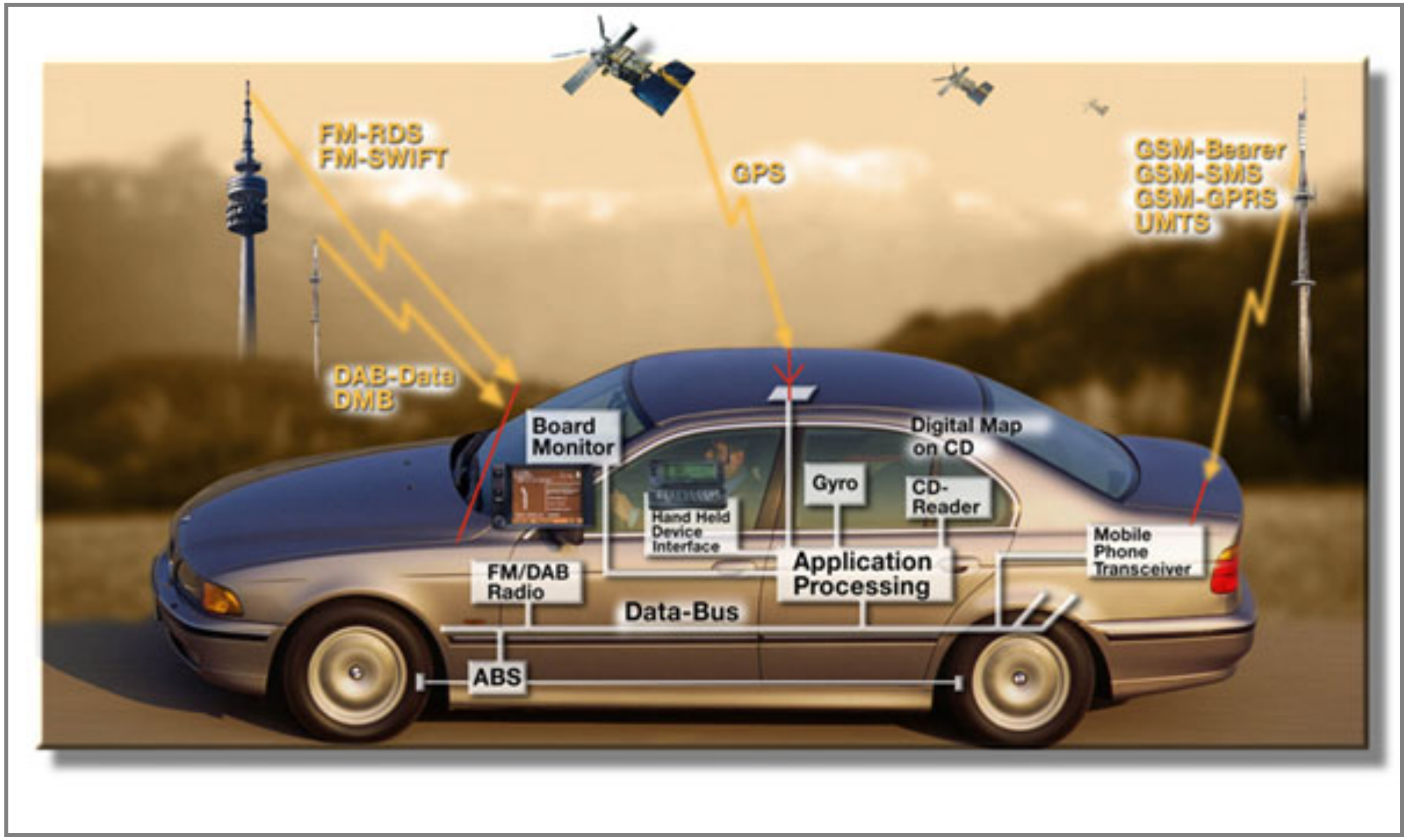
EDF schedule



$\boldsymbol{c_1 = 4}, D_1 = 10$
$c_2 = 6, D_2 = 12$
$c_3 = 4, D_3 = 8$

Review

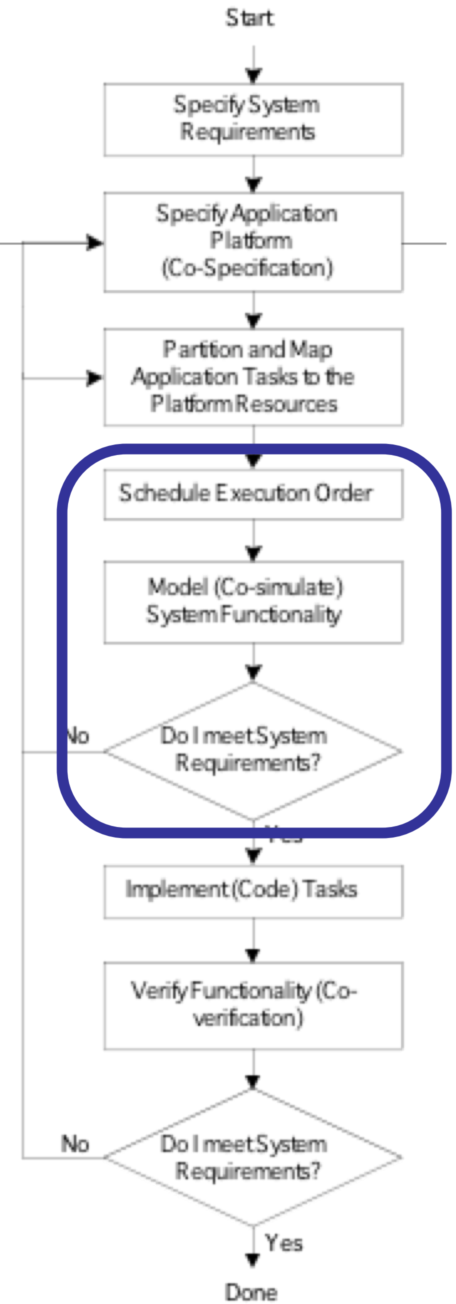
- What is a real-time system?
- What is an embedded system?
- What characteristic of a real-time system is probably the most important?



The system design process

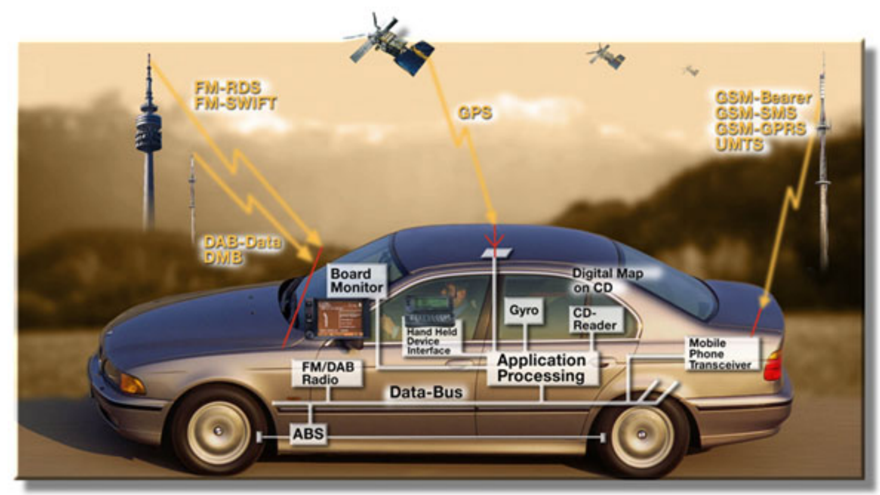
- Designing any computer system involves many steps.
- Some steps are common to many types of systems.
- A few steps are more important in a real-time system.
  - Scheduling is one such operation.
  - How do we know if a set of tasks can be scheduled in a predictable manner?

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



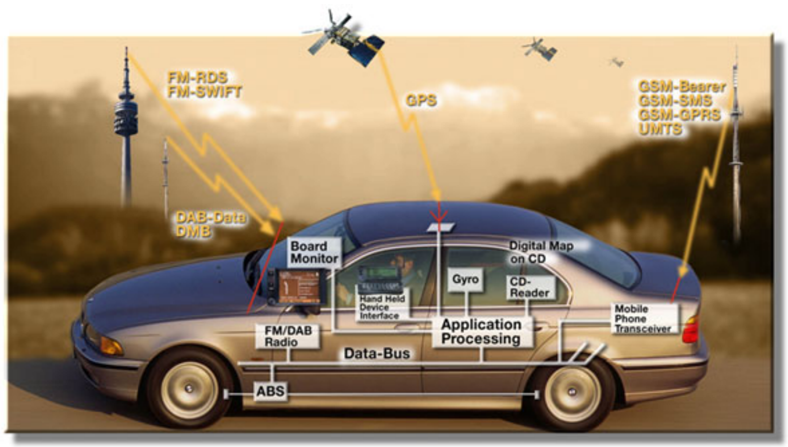
The schedulability question: Drive-by-Wire Example

- 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 – 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
- Is it possible to always compute all adjustments in time?
- The underlying computer system is a **uniprocessor** system.



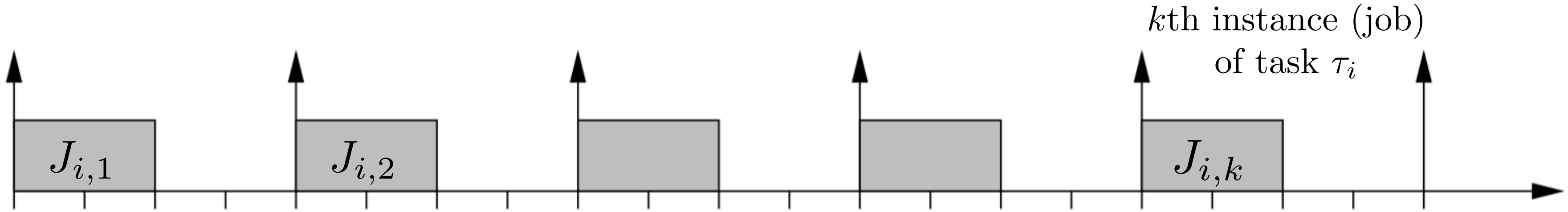
The schedulability question: Drive-by-Wire Example

- 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 – 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
- Is it possible to always compute all adjustments in time?
- The underlying computer system is a **uniprocessor** system.



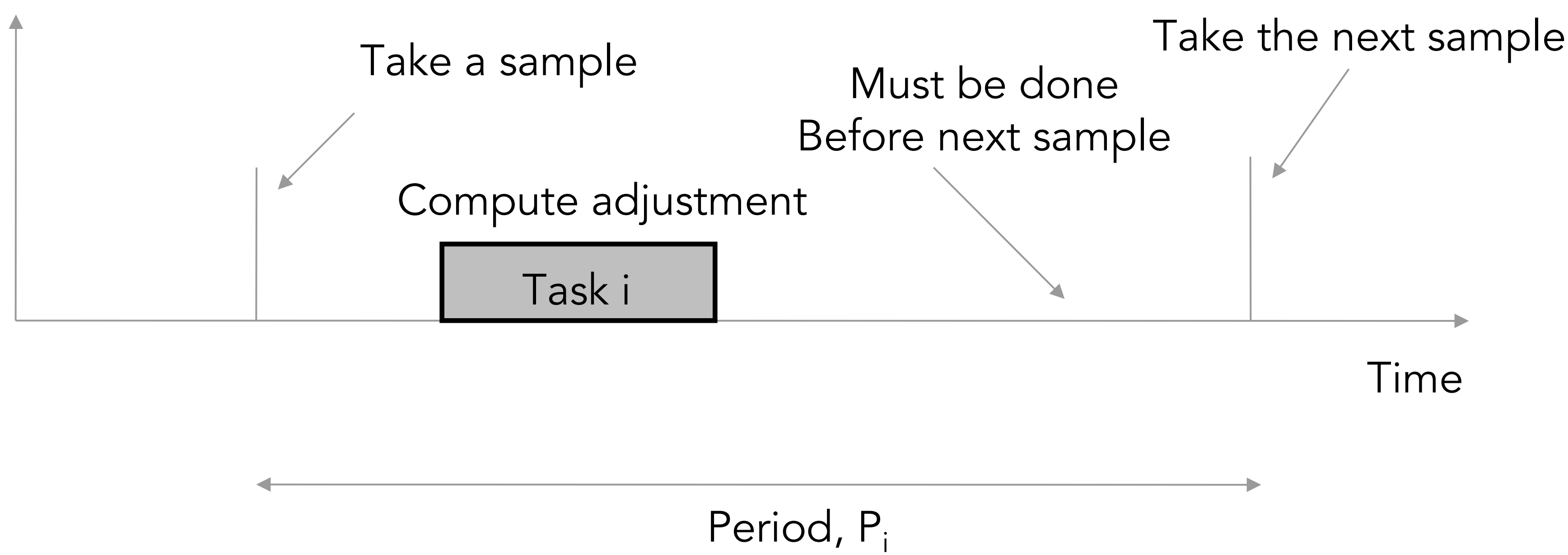
Tasks

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



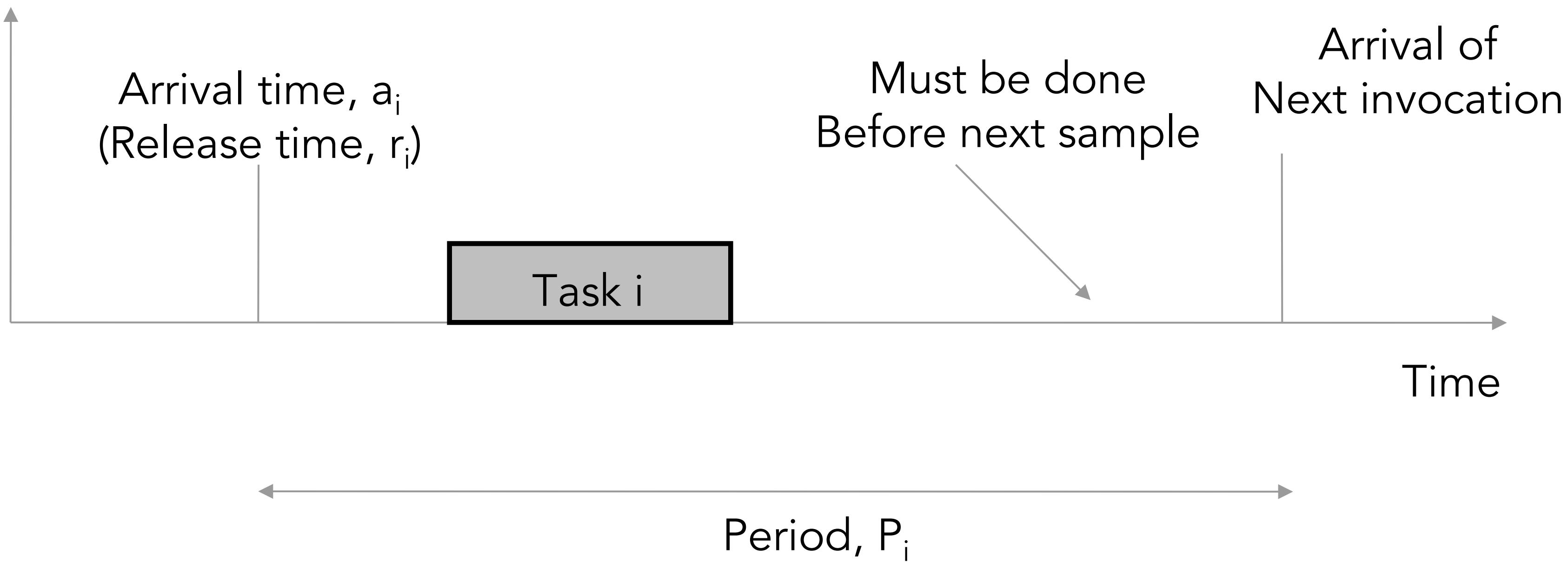
Some terminology

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



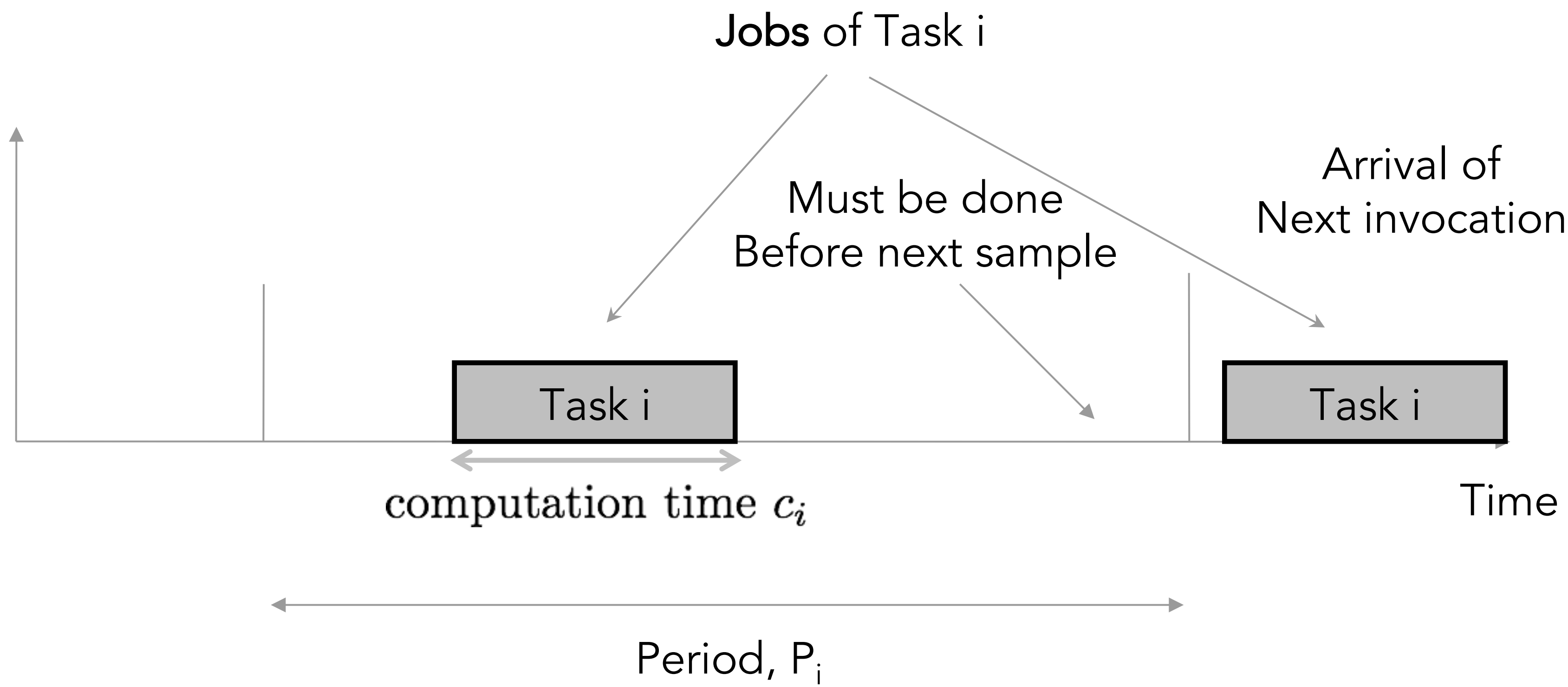
Some terminology

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



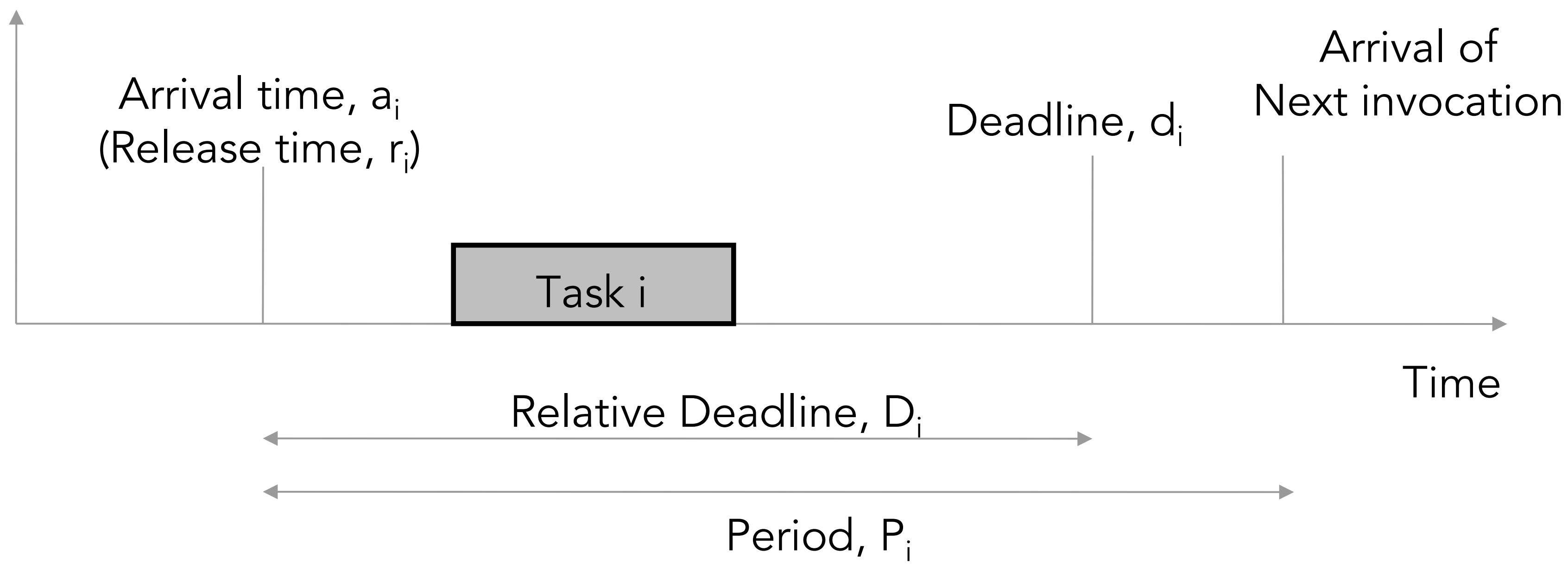
Some terminology

- 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

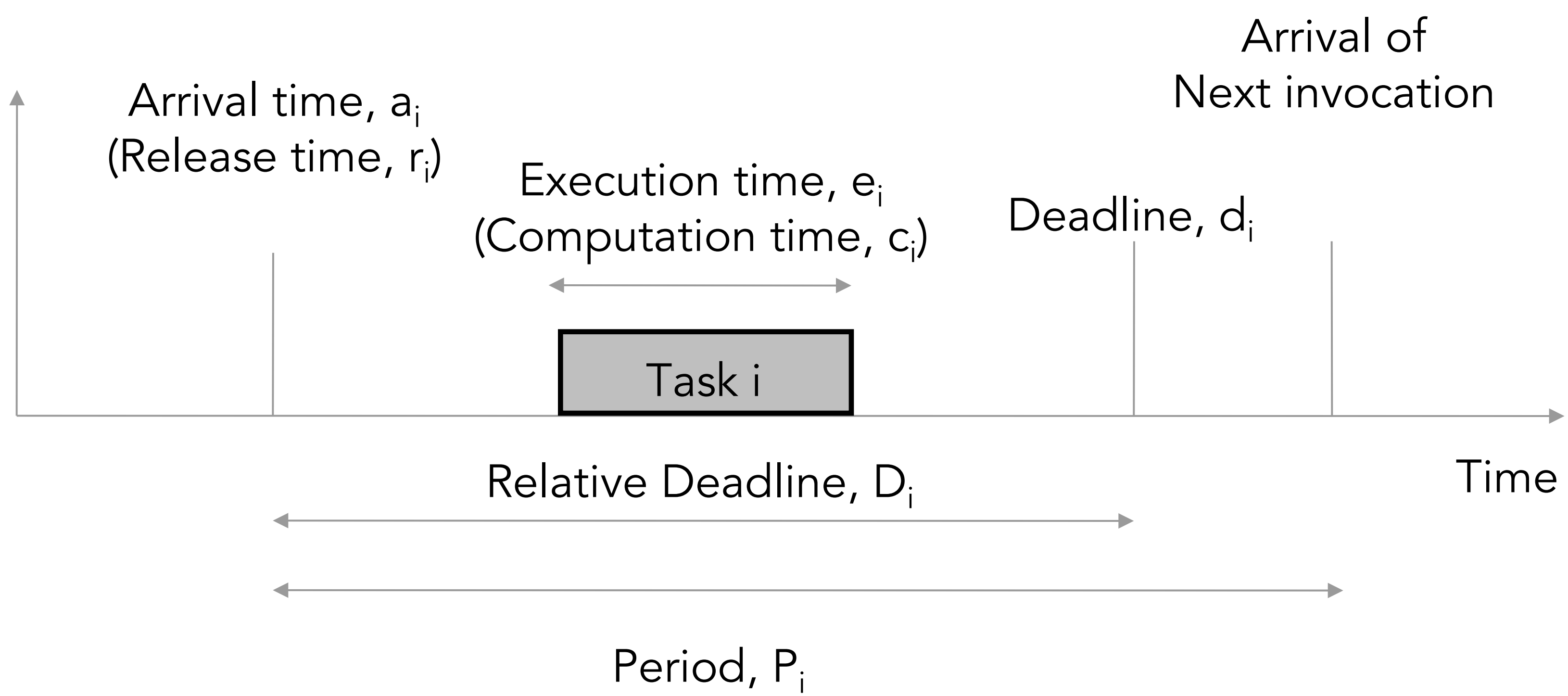
- Tasks, periods, arrival-time, deadline, execution time, etc.



(absolute deadline) d_i = (release time) r_i + (relative deadline) D_i

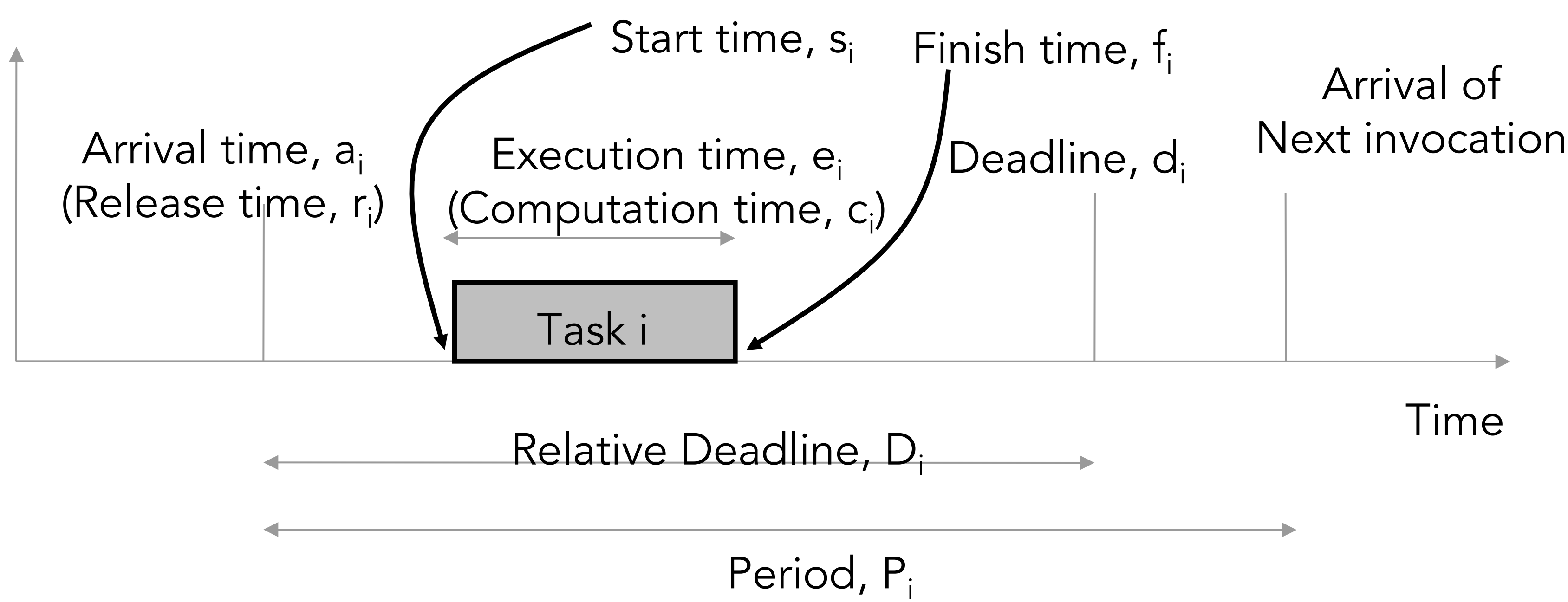
Some terminology

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



Some terminology

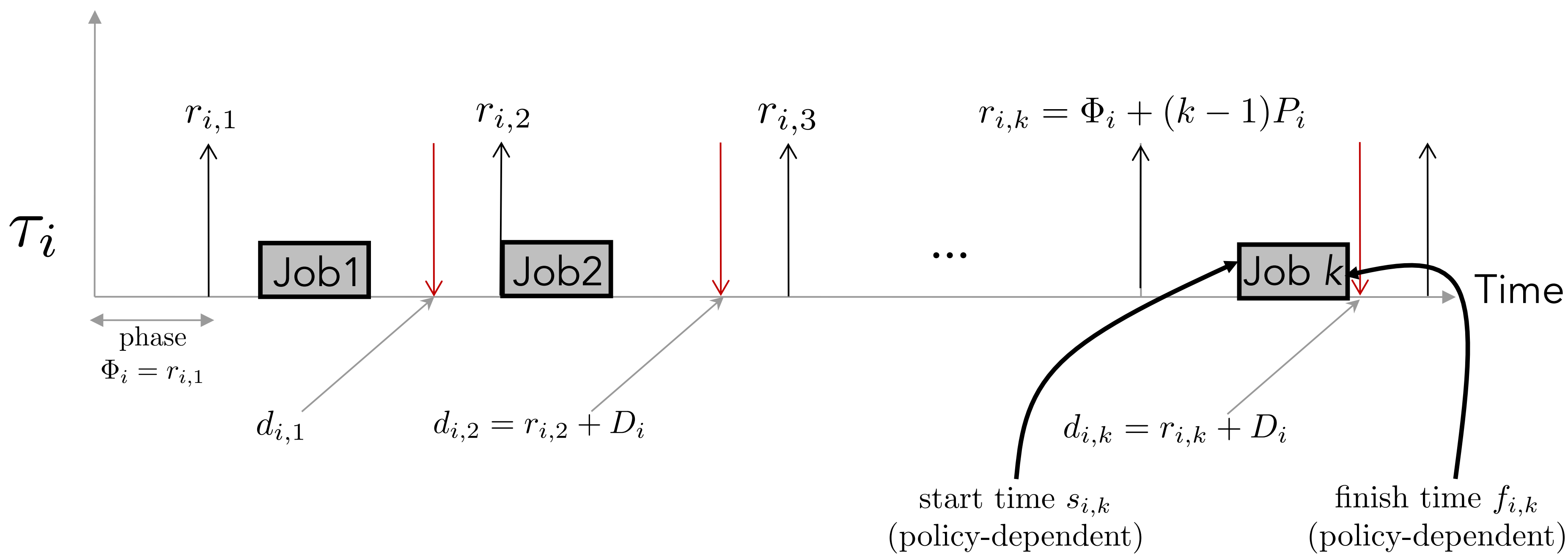
- Tasks, periods, arrival-time, deadline, execution time, etc.



Some terminology

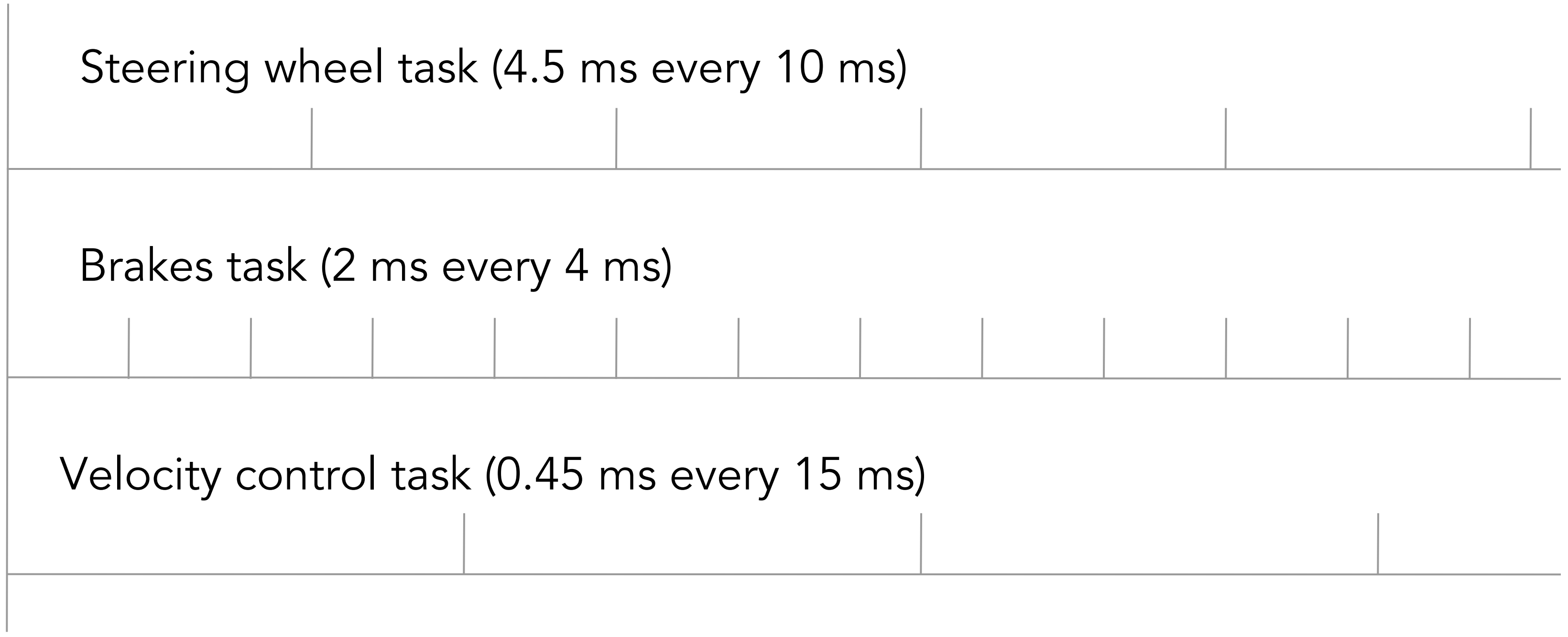
Tasks, periods, arrival-time, deadline, execution time, etc.

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



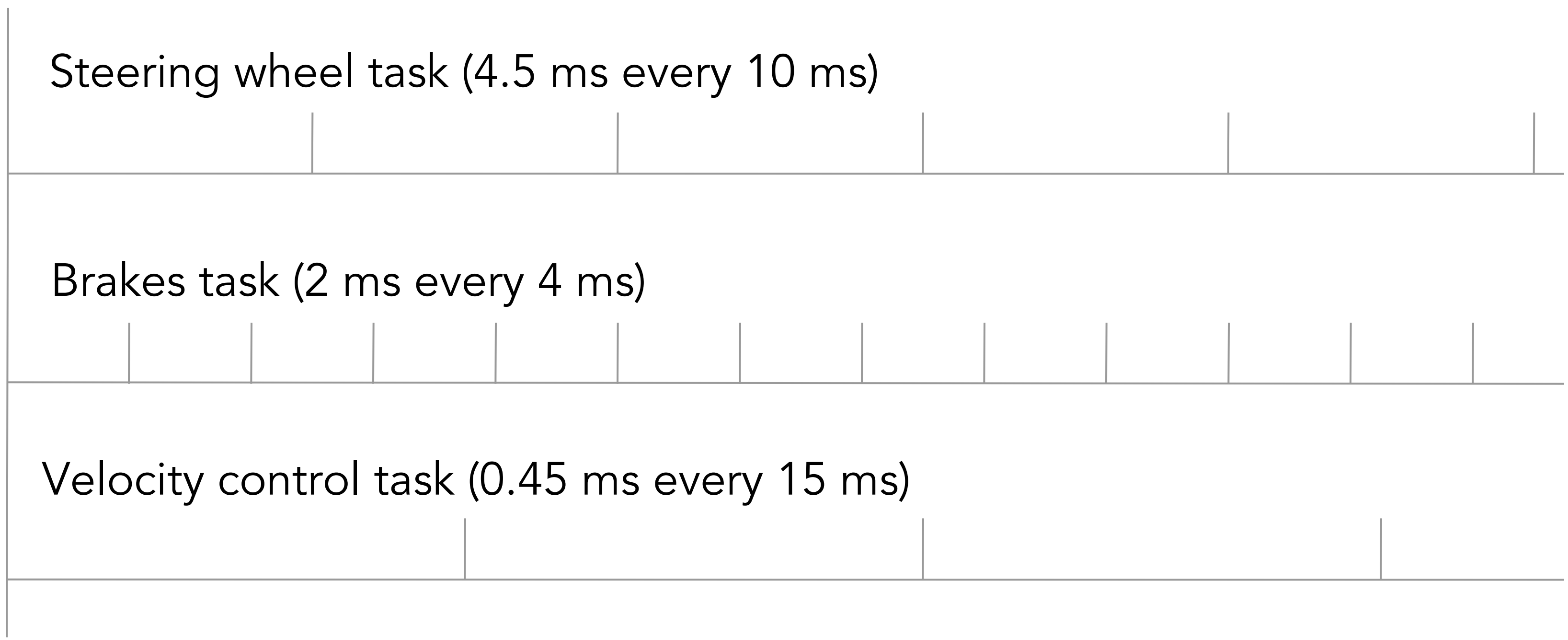
Back to the Drive-by-Wire example

- Find a schedule that makes sure all task invocations meet their deadlines
- Often, **relative deadlines are equal to the period lengths**



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%



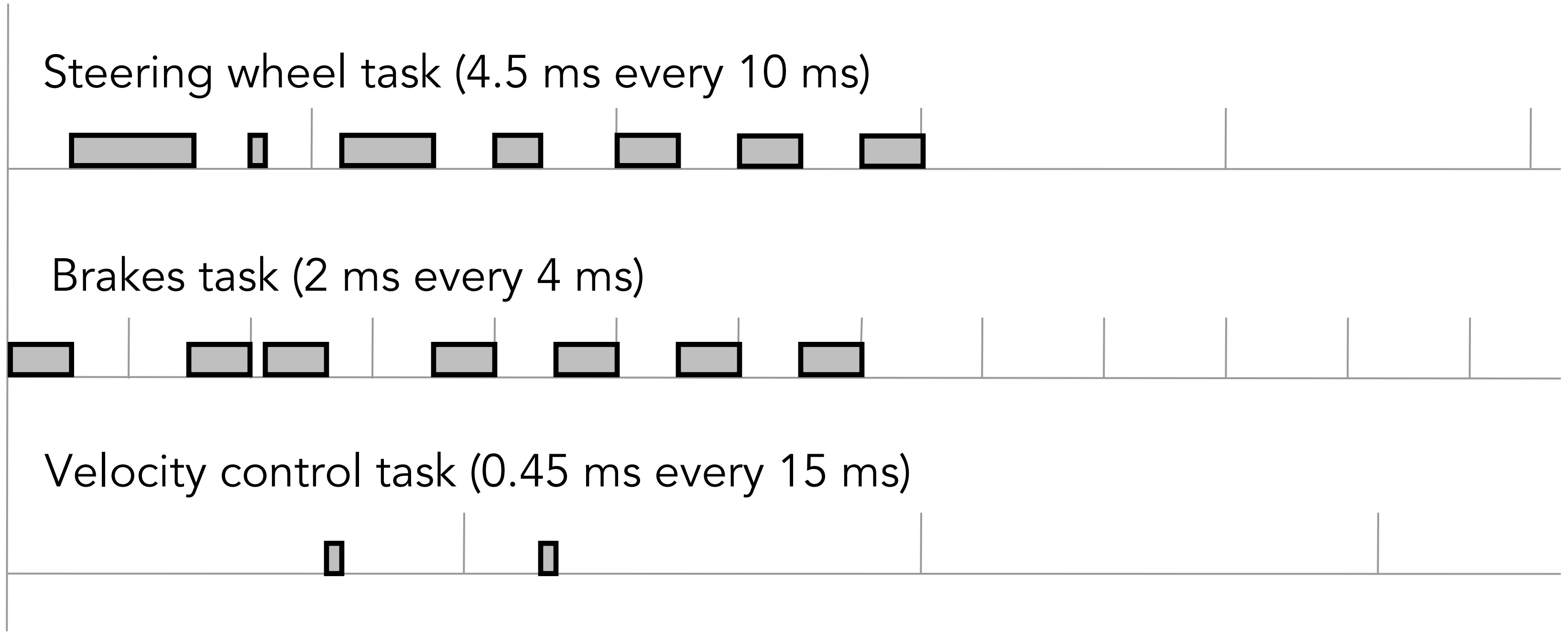
Utilization of a task set

- 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.

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

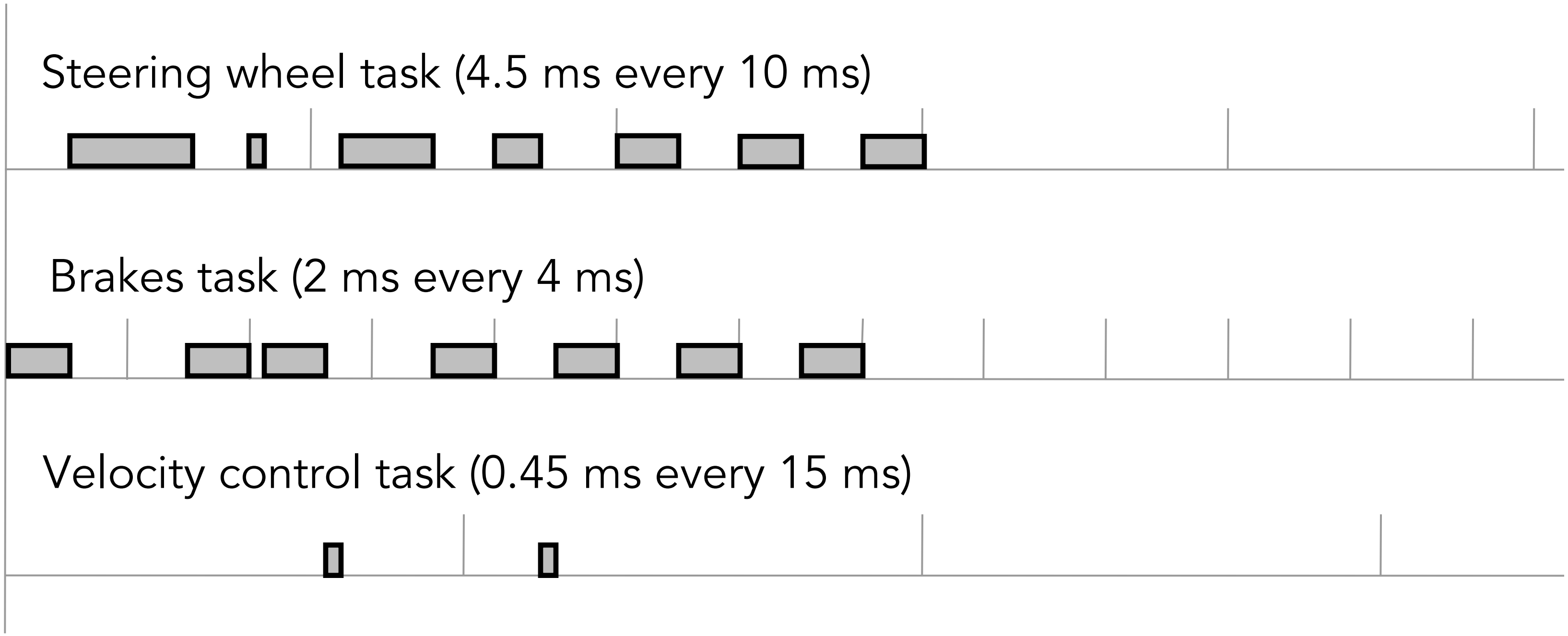
Task scheduling

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



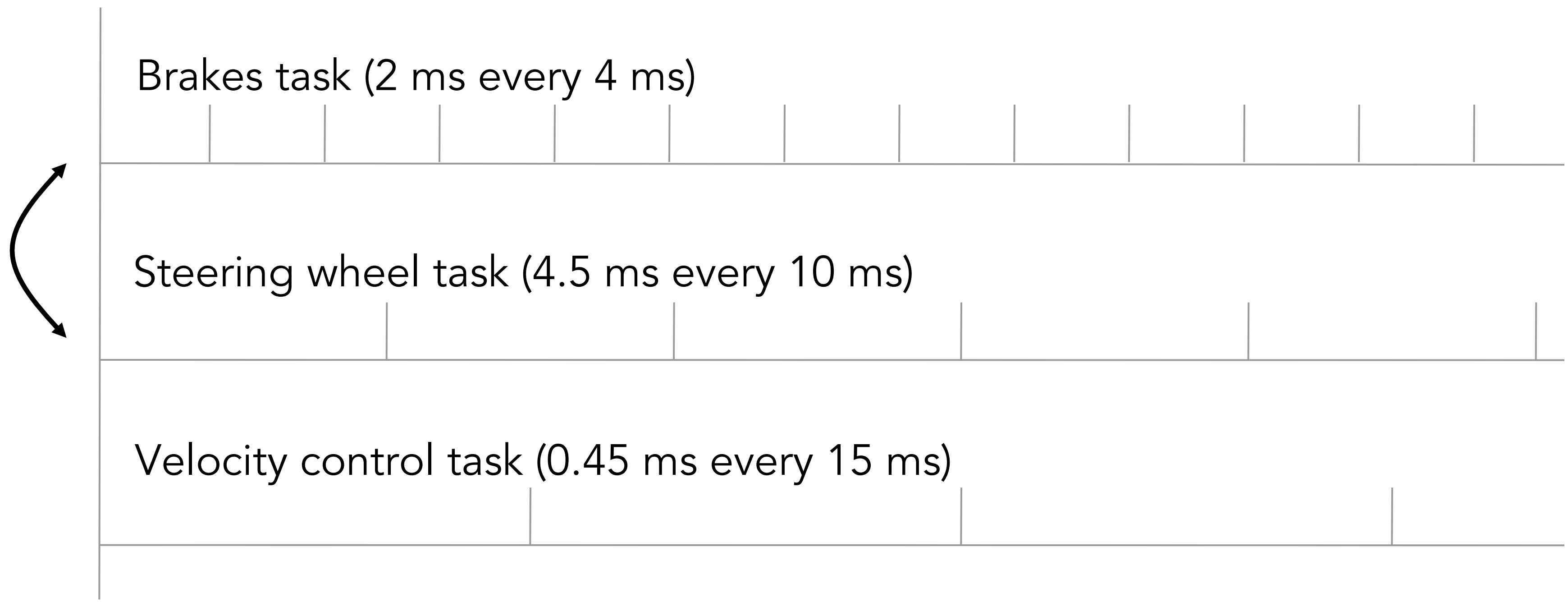
Task scheduling

- 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



Task scheduling

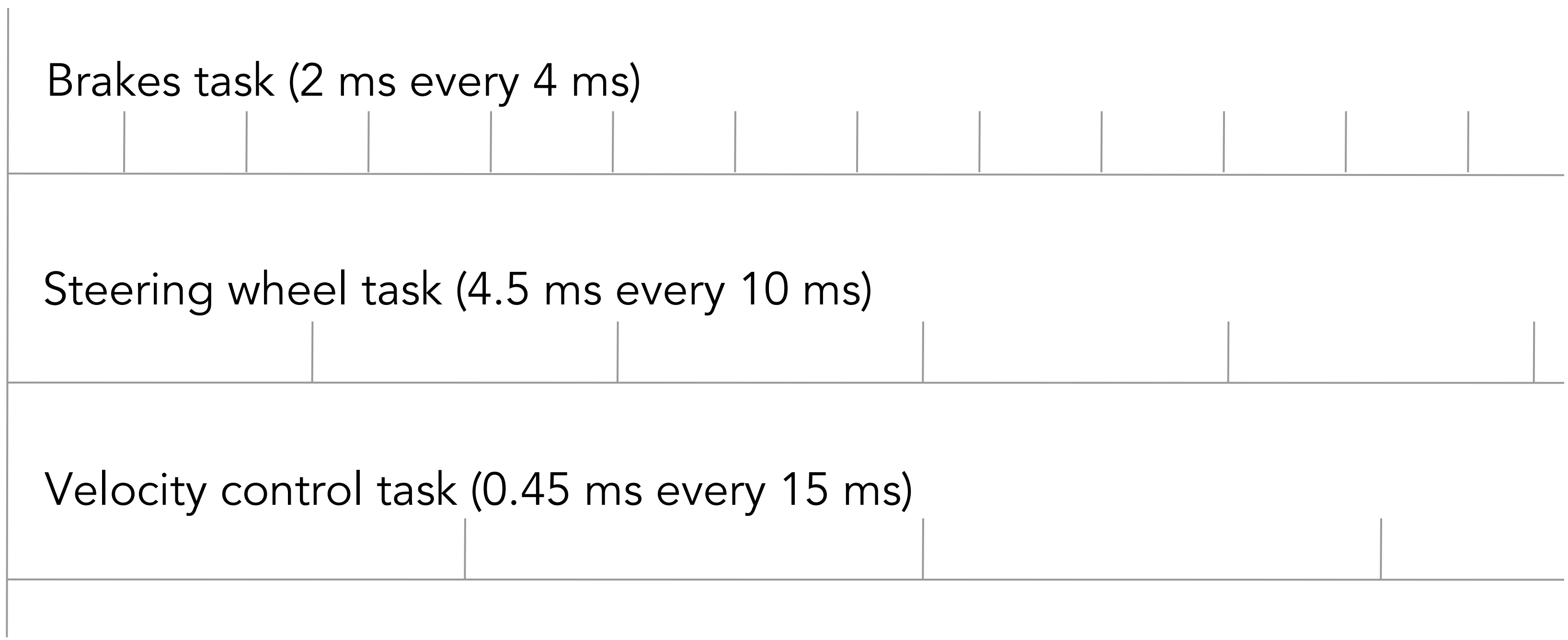
- In what order should tasks be executed?
  - Cyclic executive scheduling or
  - Priority based schedule (assign priorities; schedule is implied)



Intuition: Urgent tasks should be higher in priority

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

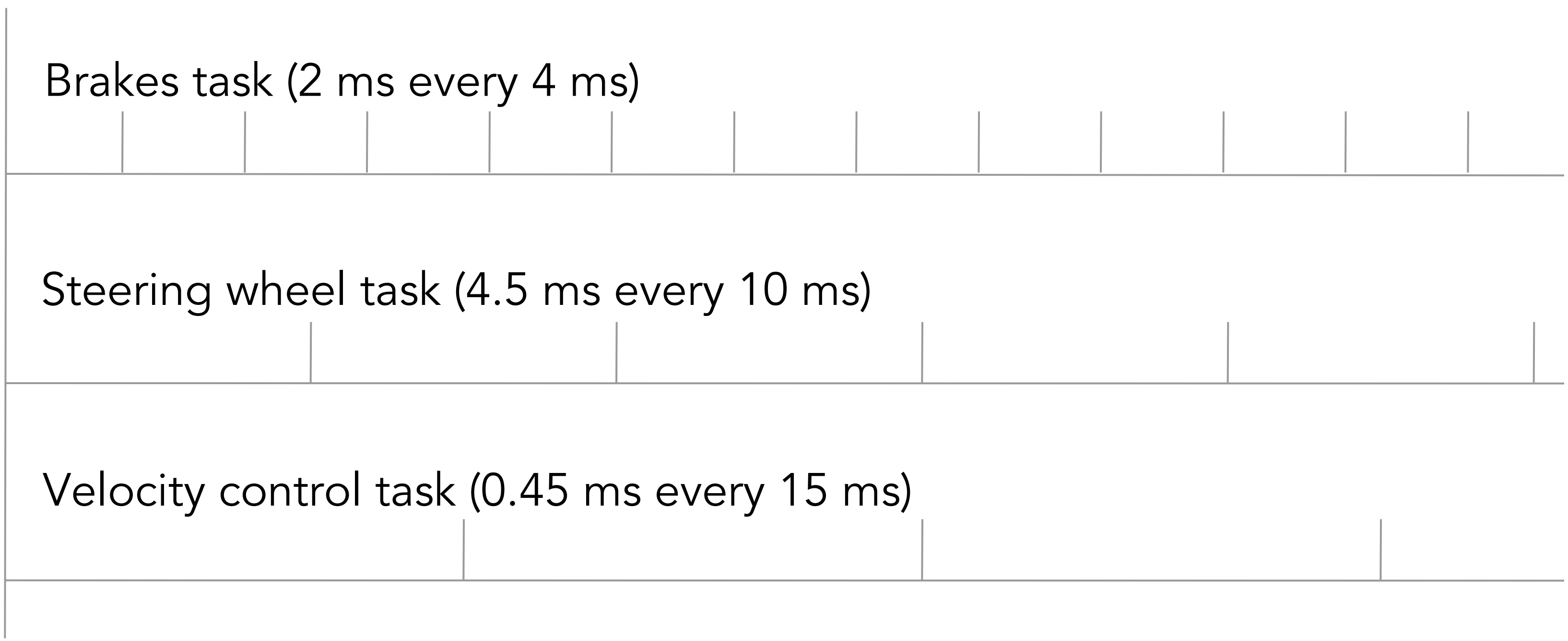
- Preemptive: Higher-priority tasks can interrupt lower-priority ones
- Non-preemptive: They can't



In this example, will non-preemptive scheduling work?

Task scheduling

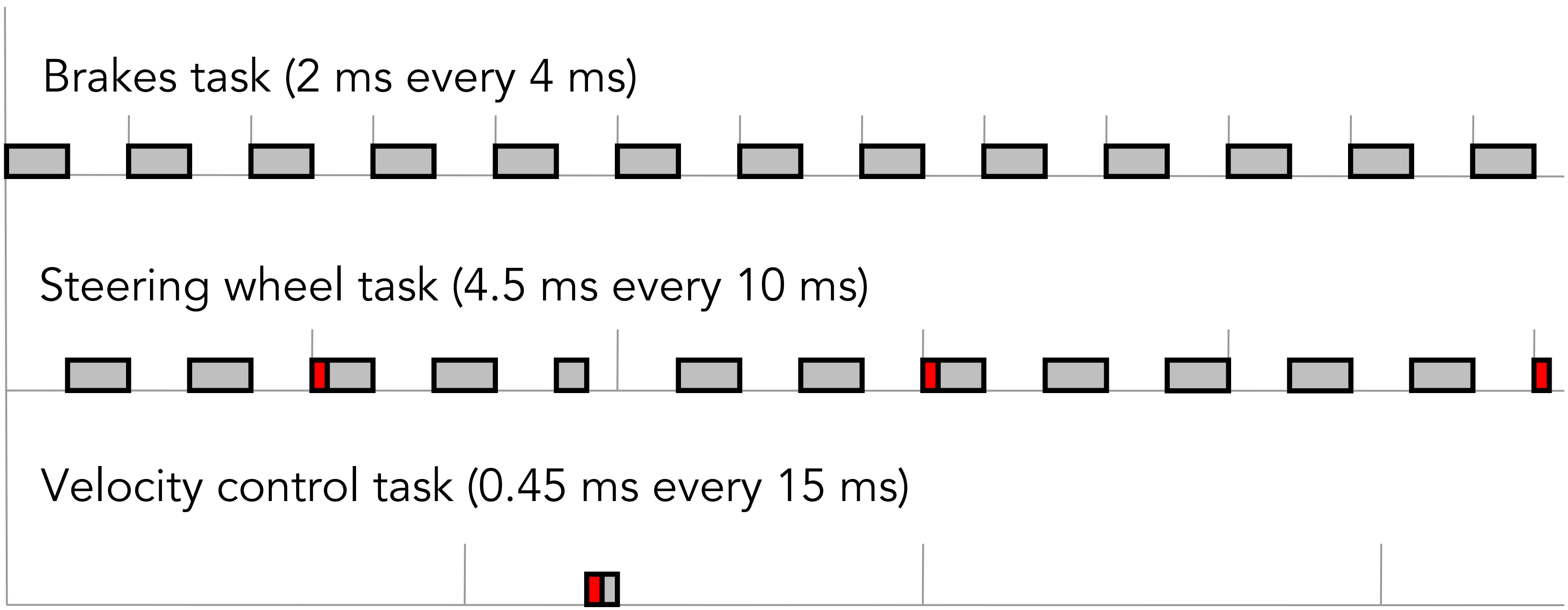
- Preemptive versus non-preemptive
  - Preemptive: Higher-priority tasks can interrupt lower-priority ones
  - Non-preemptive: They can't



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

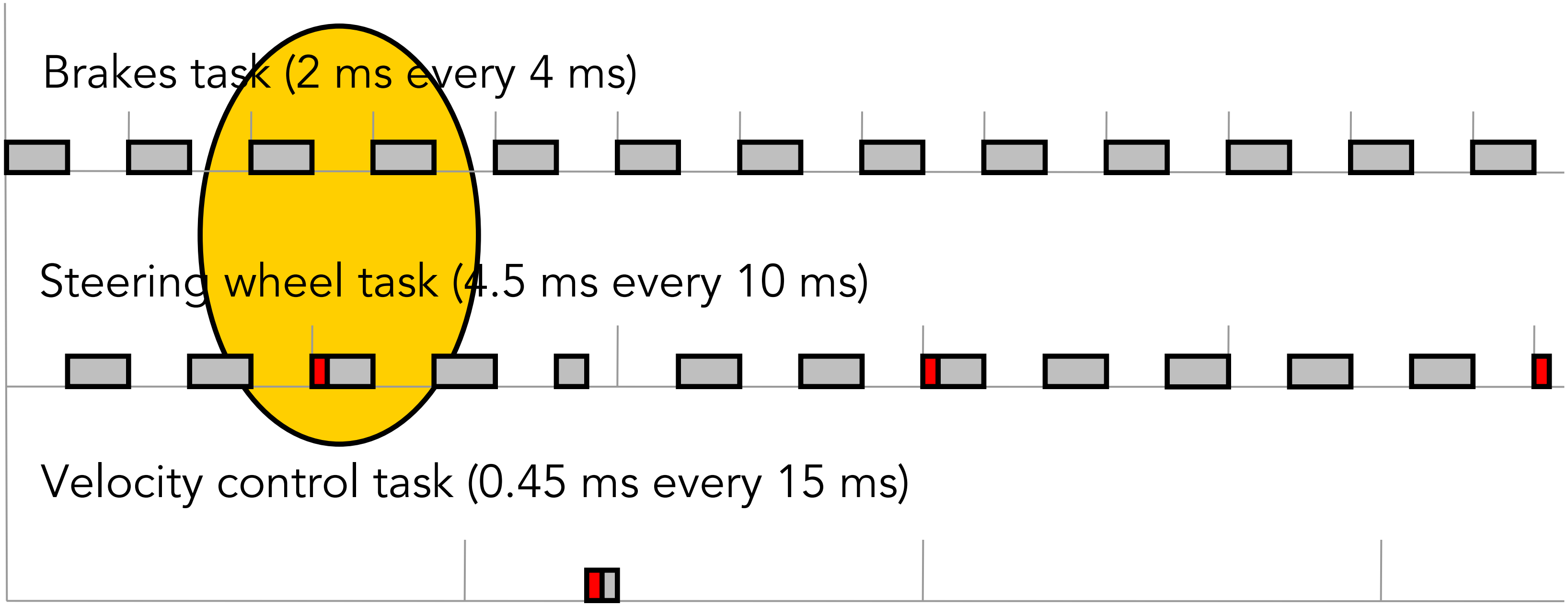
Timeline

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



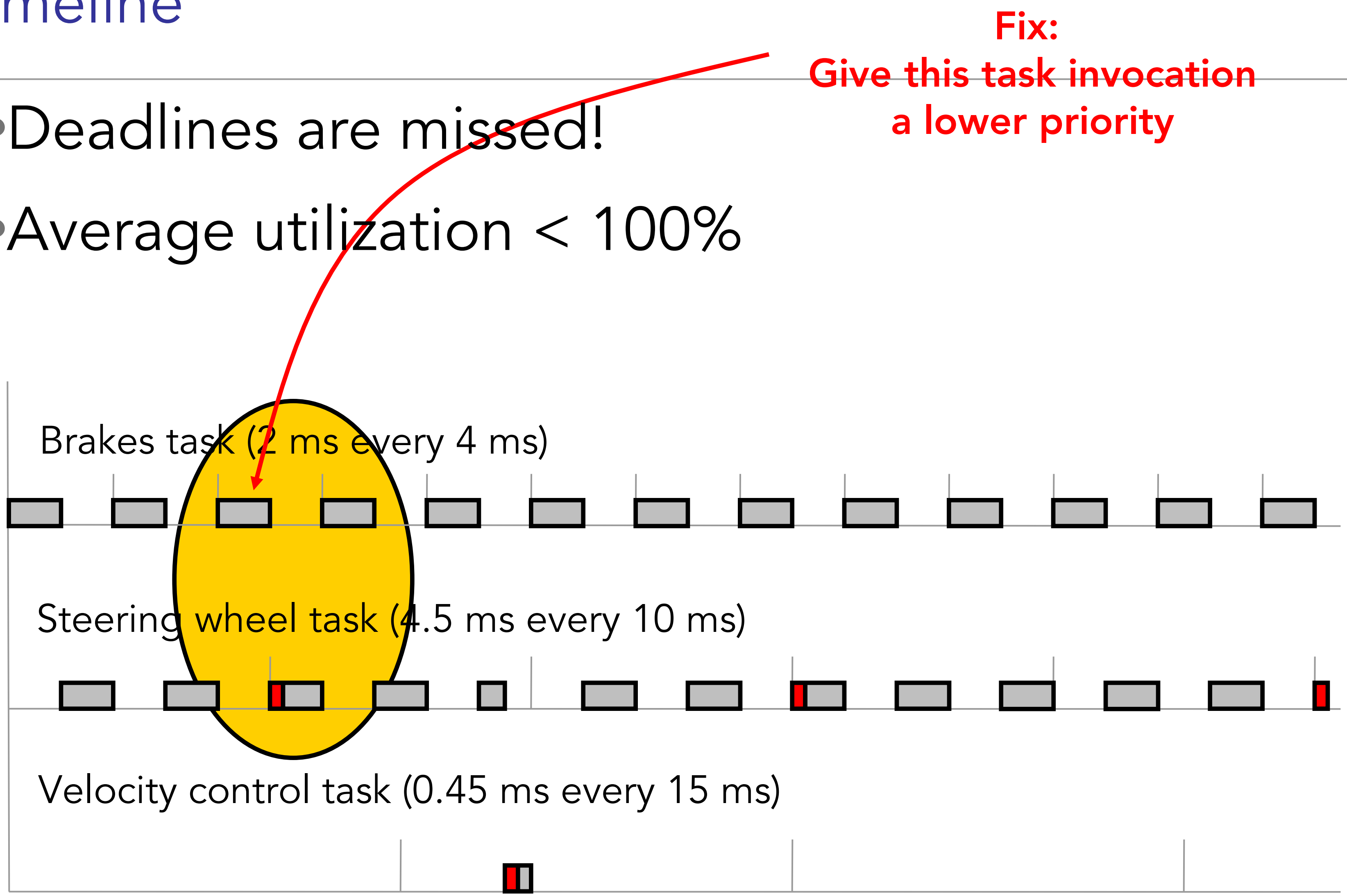
Timeline

- Deadlines are missed!
- Average utilization < 100%



Timeline

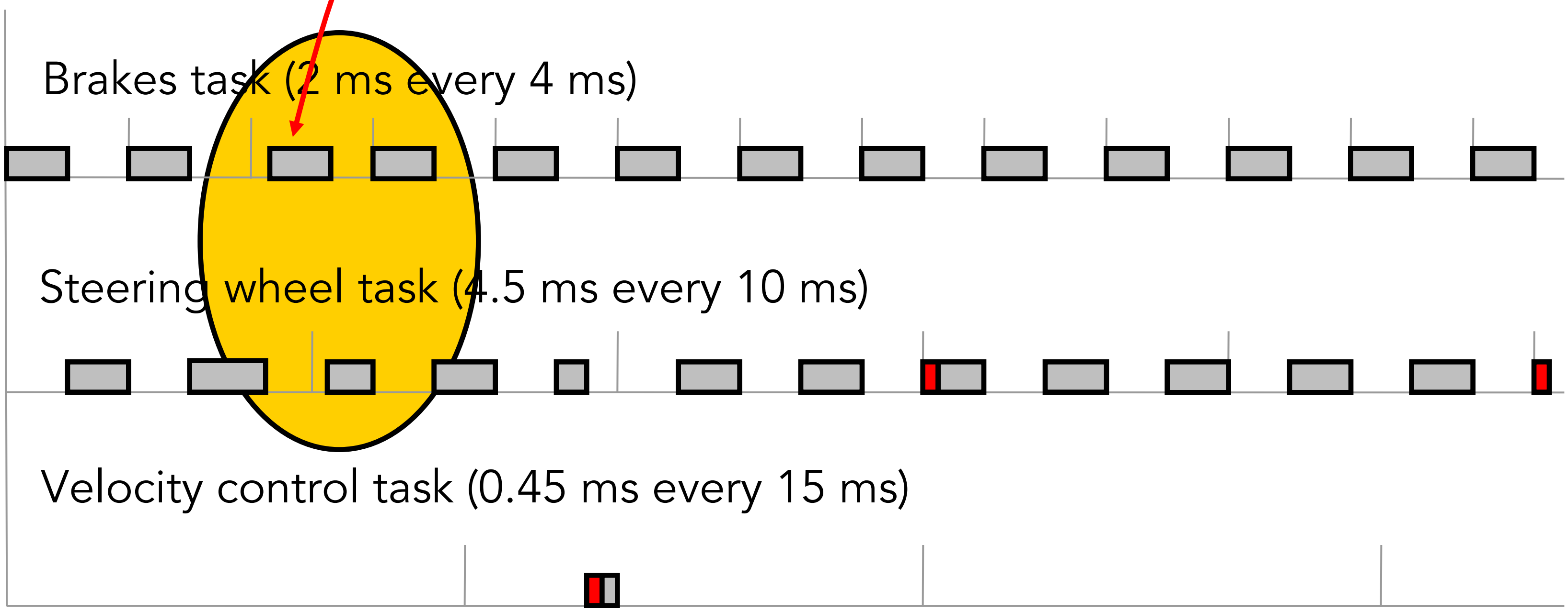
- Deadlines are missed!
- Average utilization < 100%



Timeline

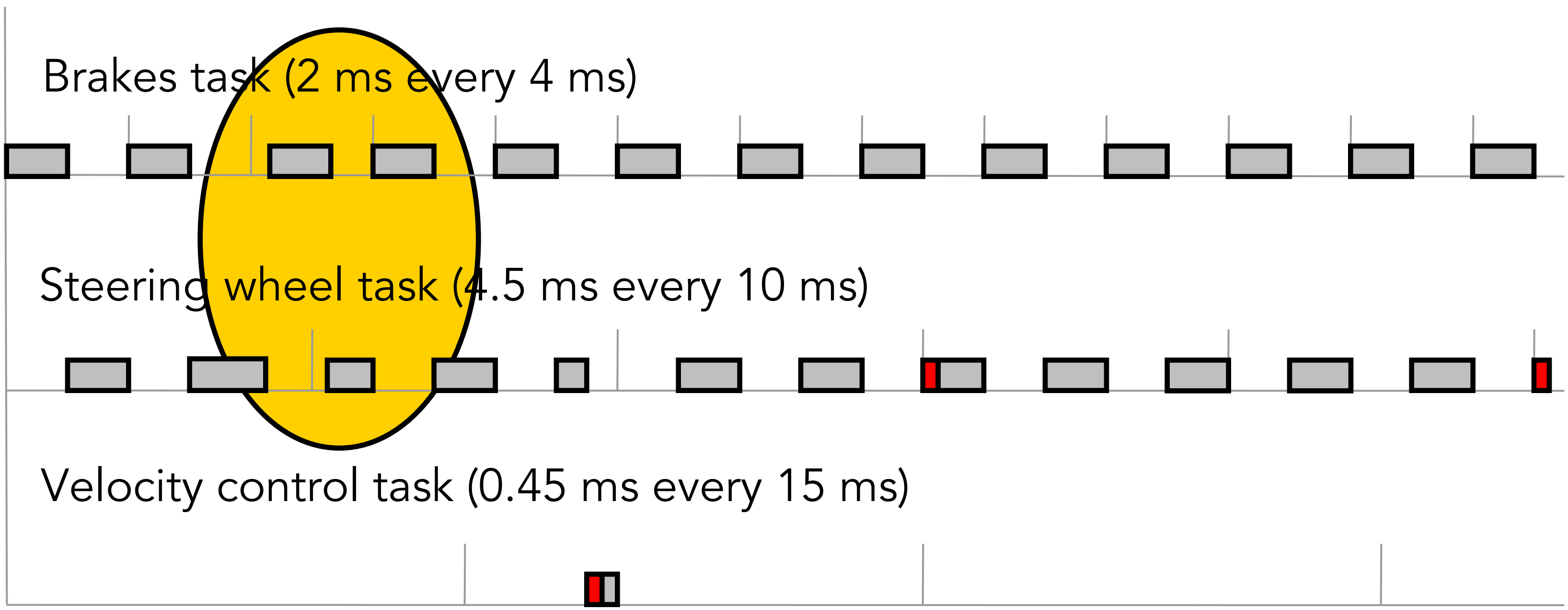
- Deadlines are missed!
- Average utilization < 100%

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 priority policies
  - **Rate monotonic priority**: tasks with shorter periods get higher priority
  - **Deadline monotonic priority**: tasks with shorter deadlines get higher priority
  - Rate monotonic priorities and deadline monotonic priorities are identical if relative deadlines are equal to the periods
  - **Shortest job first policy**
- Dynamic 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

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 <= 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 <= 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?

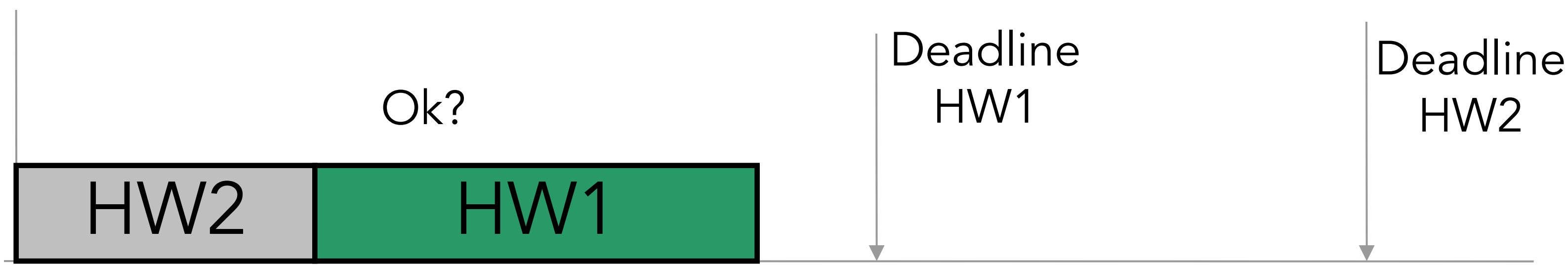
Utilization Bounds

Utilization bounds for schedulability

- U_S is called a **utilization bound** for a given scheduling policy S if and only if
  1. All task sets with utilization $\leq U_S$ can be scheduled using the policy S and
  2. *There exists at least one* task set with utilization $(U + \epsilon)$ for some $\epsilon > 0$ that **cannot** be scheduled using policy S.
- Of course, the maximum value that U_S can attain for any S is 1. Why? In class
- U_S is a function of task utilizations $u_1, \ldots, u_n$

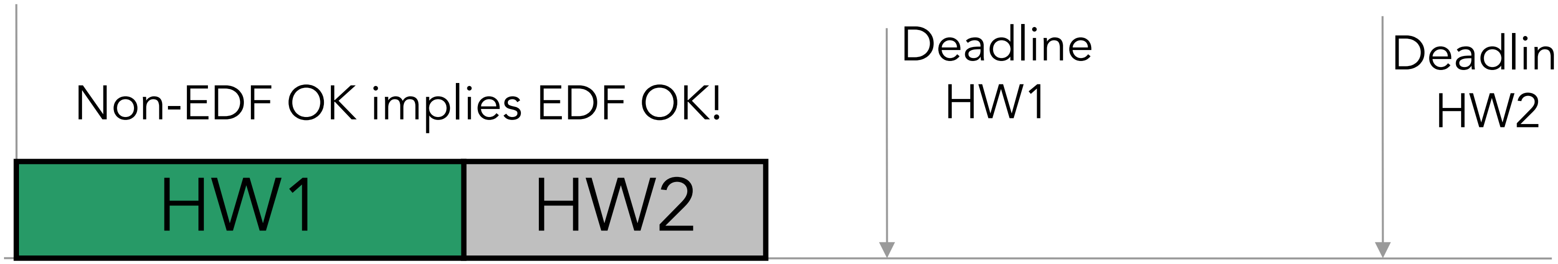
Optimality result for EDF scheduling

- 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

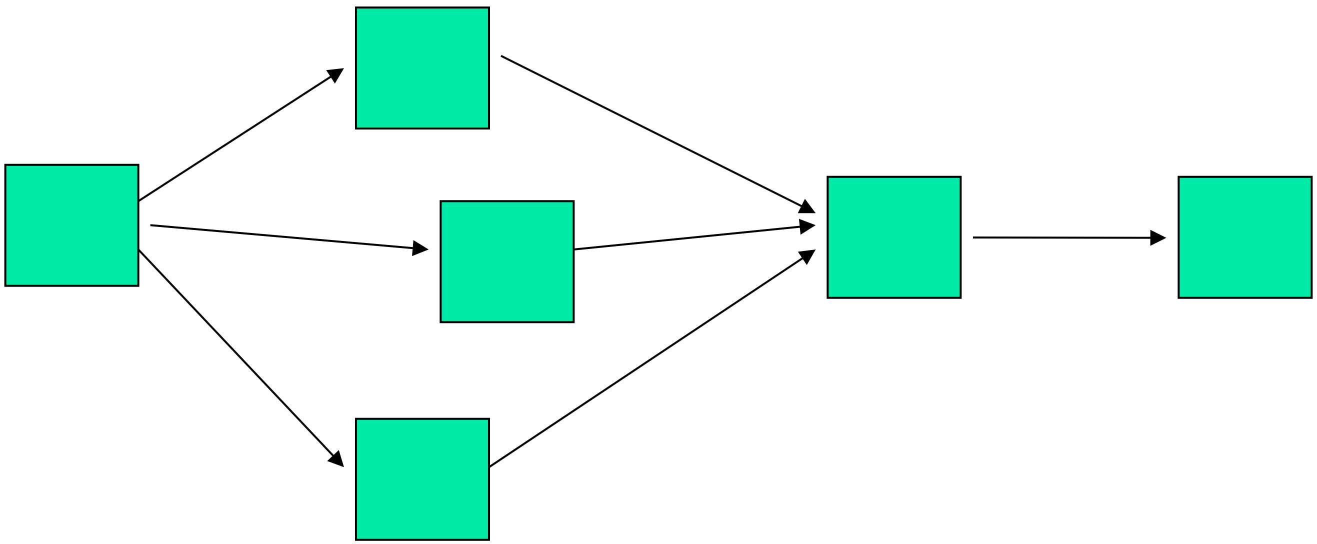
- You are the system administrator of Ameritrade.com (online stock trading site)
- You offer the following guarantee to your premium customers:
    - Stock trades of less than a $100,000 value go through in 8 seconds or you charge no commission.
    - Stock trades of more than a $100,000 value go through 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?