ECE 455 – Module 3

Real-time Scheduling

Spring 2023

Murray Dunne – mdunne@uwaterloo.ca

Slides Acknowledgement

Some material in these slides is based on slides from:

- Prof. Sebastian Fischmeister
- Prof. Carlos Moreno

Other material used with citations

Buzzword: **Cyber-physical Systems**

- Real-world integrated systems require lots of different components and disciplines
  - Software, hardware, electrical, cloud, security, etc.
  - Physics, medicine, materials, chemistry
- **Cyber-physical Systems** combine all these details together
  - Hardware and software deeply intertwined

Categorizing Systems by Timing Requirements

- **Batch** – you don’t mind when results arrive
- **Interactive** (or **Online**) – want best-effort response time
  - Good enough is fine
- **Real-time** – Need bounded delays
  - Best effort not good enough

Categorizing Systems by Correctness and Timing

	On time	**Mostly on time**	**Too late**
Correct Value	**Real-time**	**Soft real-time**	Not Real-time
Incorrect Value	**Imprecise computation**	(nameless) “likes” “views”	“space heater” (aka useless)

Criticality

- **Soft** criticality
  - missing the deadline may occur with some very low probabiltiy
- **Hard** criticality
  - missing the deadline must not occur

	Time = Fast	Time = Slow
Criticality = Hard	Airbag Flight control	Missile defense system
Criticality = Soft	Keyboard Mouse	System cleanup

Tasks and **Jobs**

- A **task** is a distinct unit of work the system must do
- A task is composed of **jobs**

- Divided into:
  - **Periodic** tasks repeat jobs at a specific time
  - **Sporadic** tasks repeat jobs with some minimum inter-arrival time
  - **Aperiodic** tasks repeat jobs randomly (or not at all)

- Note: some people use “strictly periodic” to mean periodic and then “periodic” for sporadic
  - We will stick to the above bolded definitions in this class
  - But you may see these other definitions in some published works

Tasks Continued

- Tasks can be
  - **Synchronous** if they appear “predictably” relative to execution
    - Usually this is for tasks that are initiated by the code
  - **Asynchronous** if they appear unpredictably relative to execution
    - Such as interrupts

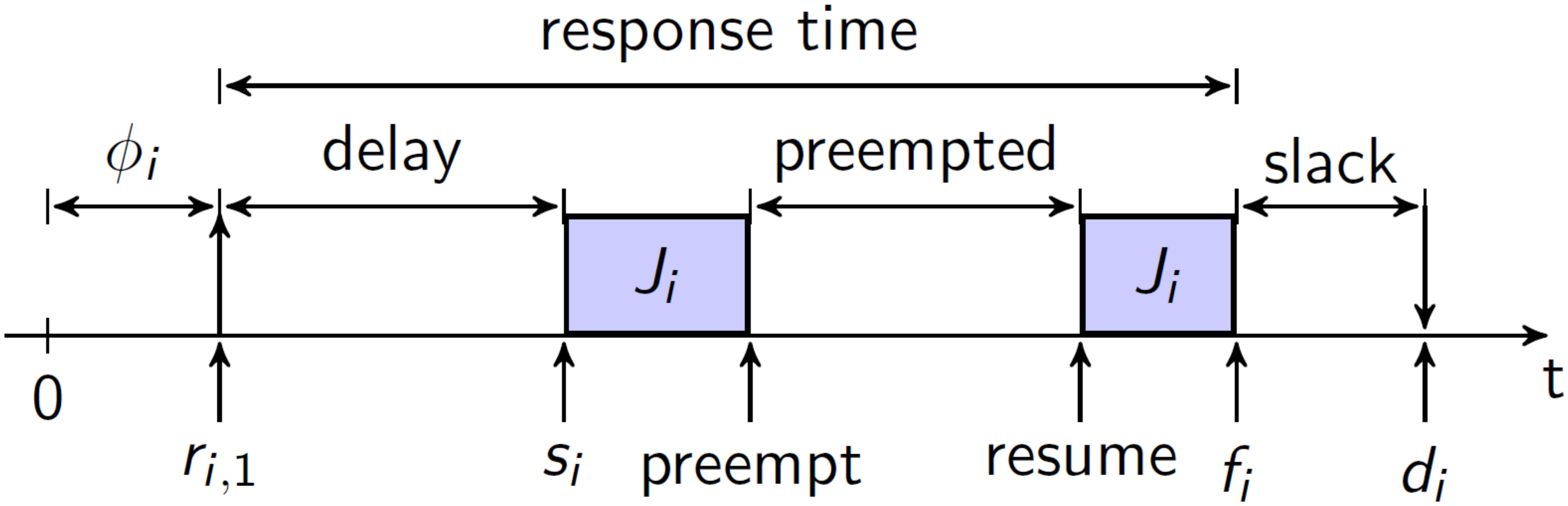
- Don’t get this confused with synchronous circuits or synchronous systems
  - These are completely different

Tasks Examples

	Periodic	Aperiodic	Sporadic
Synchronous	Polling loop	Garbage collection (don’t do this)	Exception handling
Asynchronous	Clock interrupt	Brownout interrupt	External interrupt

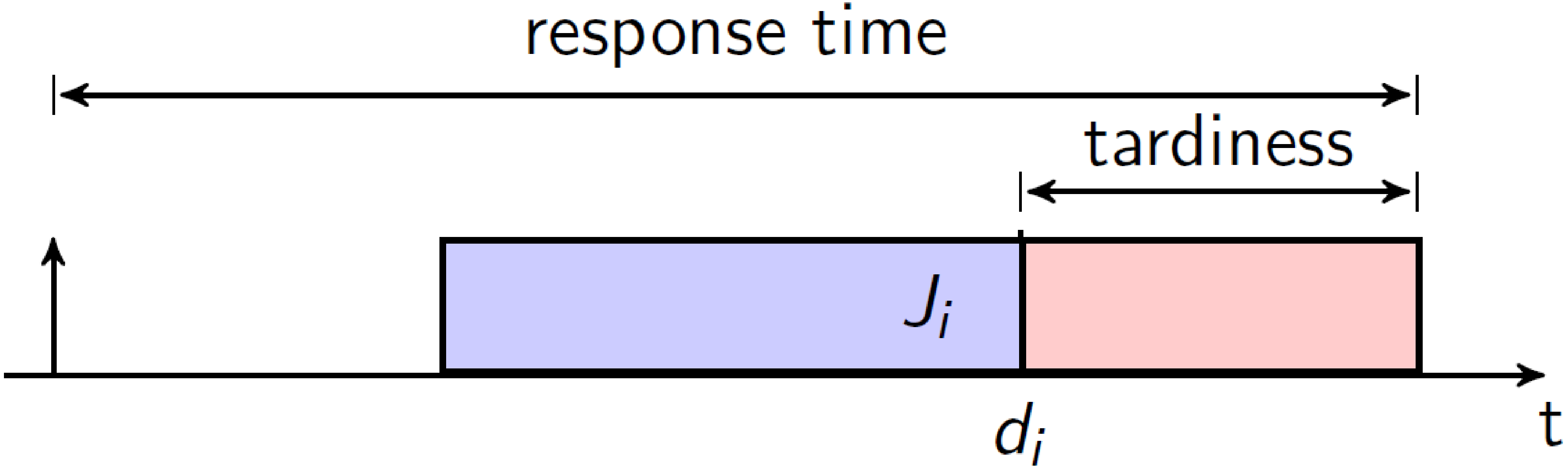
Disaster of the Day – AT&T Network Crash (1990)

Job Definitions



- r_i is the **release time** of job i
  - $r_{i,k}$ is the k^{th} release of job i etc.
- s_i is the **start time**
- f_i is the **finish time**
- d_i is the **deadline**
- $\emptyset_i$ is the **phase** (only one, no k)
- e_i is total **execution time**
- D_i is the **relative deadline** from r_i
  - $D_i = d_i - r_i$
- R_i is the **response time** $R_i = f_i - r_i$

Job Definitions Continued



- **Tardiness** is how far past the d_i a job is
  - Formally tardiness is max($f_i - d_i$, 0)
- **Lateness** is tardiness but allowing negative values
  - Formally lateness is $f_i - d_i$
- **Feasible interval** is the real-valued range $(r_i, d_i]$

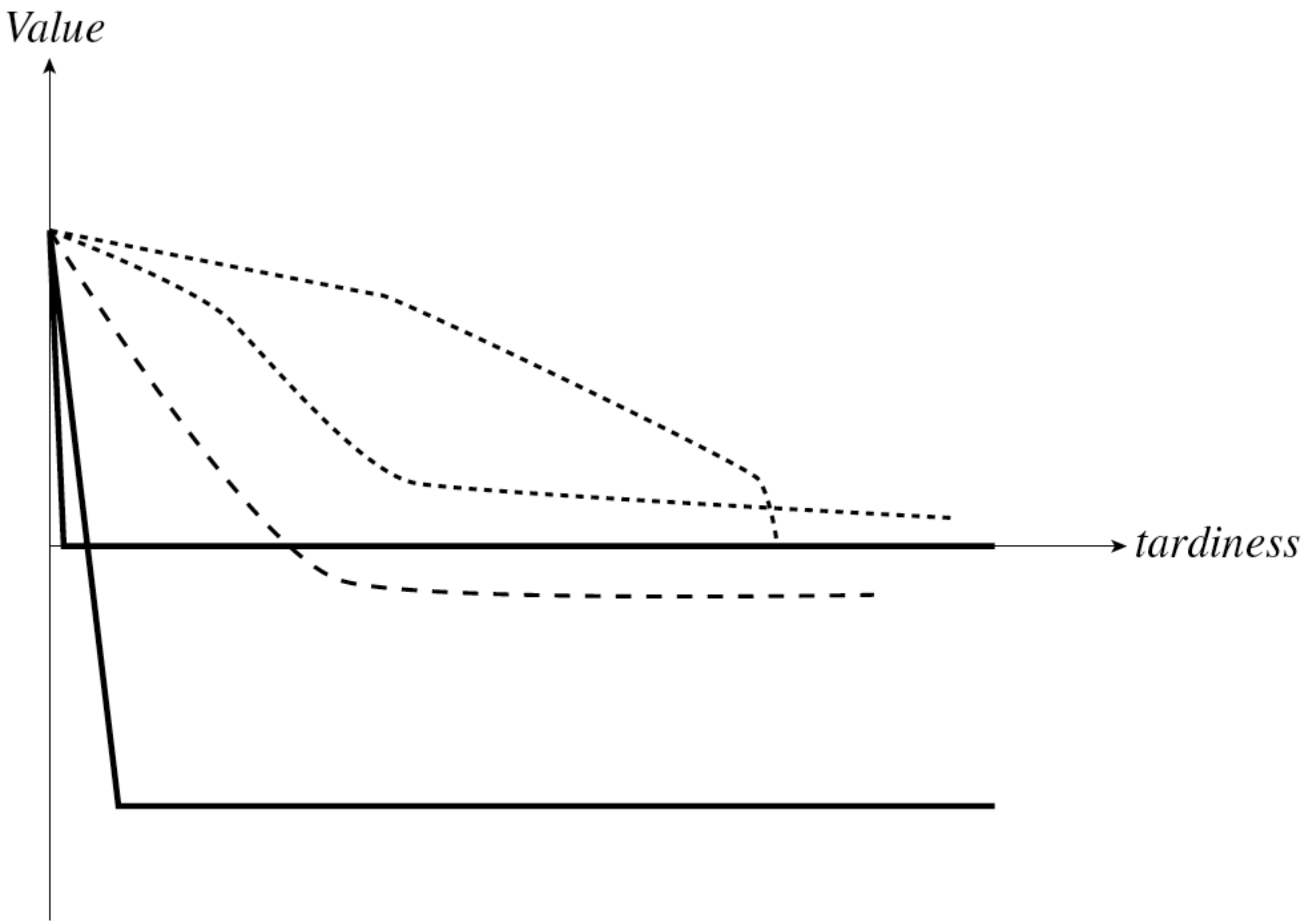
Hard Real-time

- Formally:
  - A **hard real-time** system is a system in which tardiness must always be zero!

- Practical considerations: it is sometimes impossible to prove
  - Cosmic rays
  - Radiation probability
- Us a probably estimate instead
  - Often order 10^{-7} or lower
    - Some references will define hard-real time systems by specific probability numbers
      - This is wildly inconsistent
    - The developer must prove this

“Usefulness” functions

- How useful is the result if it’s late?
  - Solid line for hard real-time
    - It could just be useless
    - Or actively harmful
  - Dotted line for soft real-time



Processors and Resources

- A **processor** processes jobs
  - There may be different types of processors in a system
  - There are usually groups of interchangeable processors in multicore systems
  - Each processor in a multicore system is denoted $P_1 \ldots P_m$
    - Do not confuse this with period P_i coming soon

- A **resource** is something a processor requires to do a job
  - Memory, mutex, I/O device
  - May require a lock
  - Mostly hand-waved away in this class
    - “Plentiful” resource

Workloads

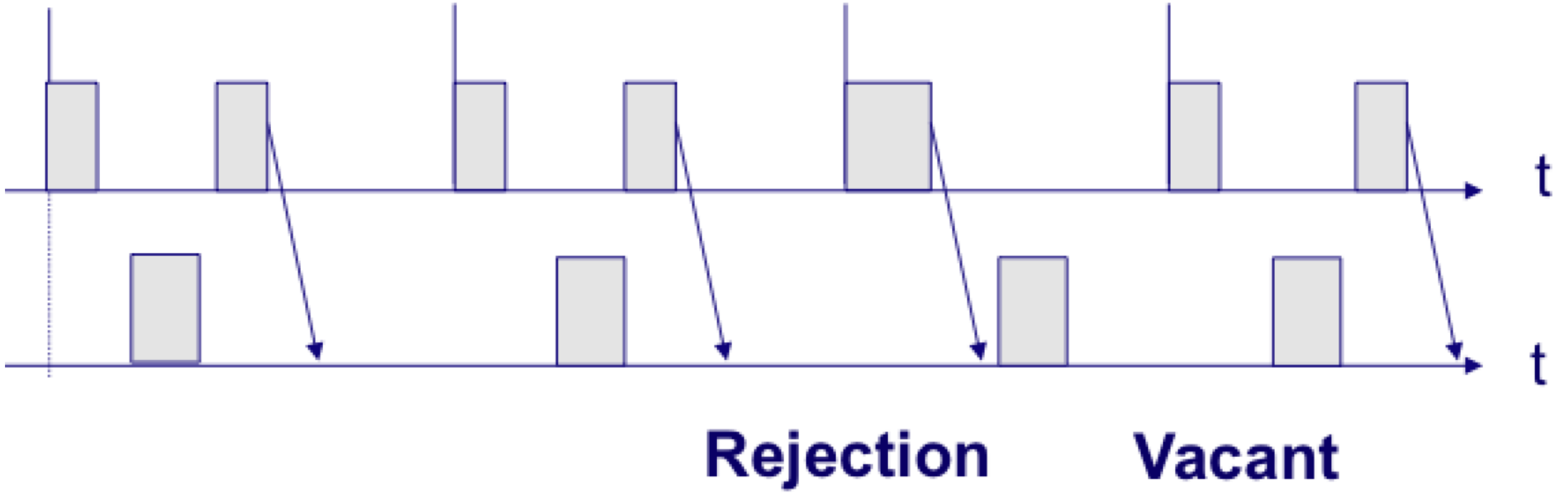
- A **workload** is a set of jobs to be processed
- Workloads can change at runtime
    - **A priori workloads** (also called **static** workloads) are all known ahead of time
    - **Dynamic workloads** may add jobs at runtime
        - Most workloads are dynamic in practice due to aperiodic tasks

Jitter

- **Jitter** is the fluctuation in occurrences of a repeated event
  - Assigned on a task level
- Types of jitter include:
  - **Relative release jitter** $RRJ_i = \max(|(s_{i,k} - r_{i,k}) - (s_{i,k-1} - r_{i,k-1})|)$
  - **Absolute release jitter** $ARJ_i = \max_k(s_{i,k} - r_{i,k}) - \min_k(s_{i,k} - r_{i,k})$
  - **Relative finish jitter** $RFJ_i = \max(|(f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1})|)$
  - **Absolute finish jitter** $ARJ_i = \max_k(s_{i,k} - r_{i,k}) - \min_k(s_{i,k} - r_{i,k})$
  - And similar for **execution jitter** $(f_{i,k} - s_{i,k})$

Rejected and Vacant Sampling

- Too much data?
  - **Rejected sampling**
  - Usually caused by jitter
- Reuse old data?
  - **Vacant sampling**
  - Usually caused by jitter



Execution Time

- The time a job takes to execute without any interference
  - Can differ between runs
  - The true execution time of job J_i isn't relevant
  - We care about the bounds $[e_i^-, e_i^+]$
  - Typically e_i just means e_i^+
    - However sometimes we get **underutilization**
      - We might overspecify the processor if we overestimate e_i^+
      - This is why having a good WCET estimate T' is so important!

Periodic Task Model

- A task T_i is characterized by a pair (P_i, e_i)
  - A job in this task is released every P_i time units
  - These jobs each have execution time e_i
  - The **utilization** of T_i is $u_i = e_i/P_i$ if $d_i = P_i$
    - This is the most common case
    - In what scenarios would $d_i \neq P_i$?
- The **hyperperiod** of a set of tasks $T_1 \ldots T_n$ is $H = \mathrm{lcm}(P_1 \ldots P_n)$

Job Precedence

- Job precedence constraints restrict the execution order of jobs
  - A partial ordering of jobs $J_i < J_j < J_k$
  - J_i is the **immediate predecessor** of J_j
  - J_i is a **predecessor** of J_k
  - A job is ready when all its constraints are satisfied
    - All its predecessors are finished
  - Usually represented with a precedence graph $G = (J, <)$

Additional Job Constraints

- Job can have other constraints
  - Any arbitrary evaluable condition could be used
    - Checking physical parameters
    - Check the response time of other jobs
    - Waiting for one of several jobs to finish

Job Preemption

- Jobs are either **preemptable** or **non-preemptable**
    - Sometimes only parts of a job are non-preemptable
        - Temporarily disable interrupts
- Preemption requires a context switch
    - This is the main overhead of most scheduling algorithms

Optional Jobs

- Jobs are either **mandatory** or **optional**
  - The system still works if optional jobs are dumped
  - Some examples:
    - Record keeping
    - Screen refreshes above some frequency

Schedules

- A **schedule** is an assignment of all jobs to available processors (or resources)
  - A schedule is **valid** iff
    - Every processor/resource is assigned to at most one job at any given time
    - Every job is assigned to at most one processor/resource at any given time
    - The entirety of a jobs WCET (e_i) is allocated
    - All precedence and resource use constraints are satisfied
  - If any of these properties is not met, the schedule is **invalid**
  - Note: there's no mention of deadlines here!

- A **scheduler** follows a **scheduling policy** (also called a **scheduling algorithm**) to allocate processors and resources to jobs

Feasible Schedules

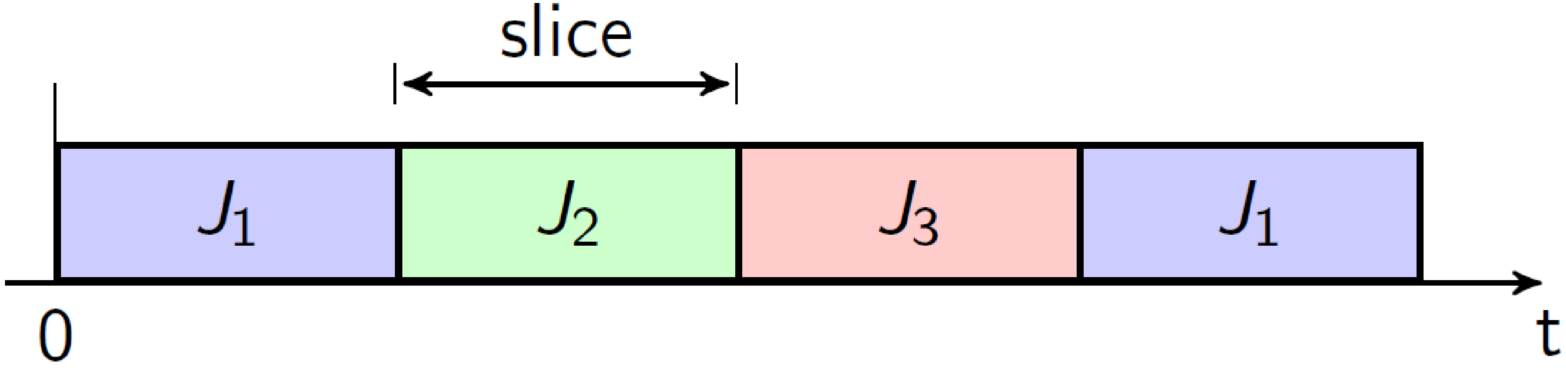
- A schedule is **feasible** if it is valid and meets all job timing constraints
- A workload is **schedulable** by a scheduling policy if that policy can find a feasible schedule for the workload

- How do we evaluate scheduling policies?

Starvation

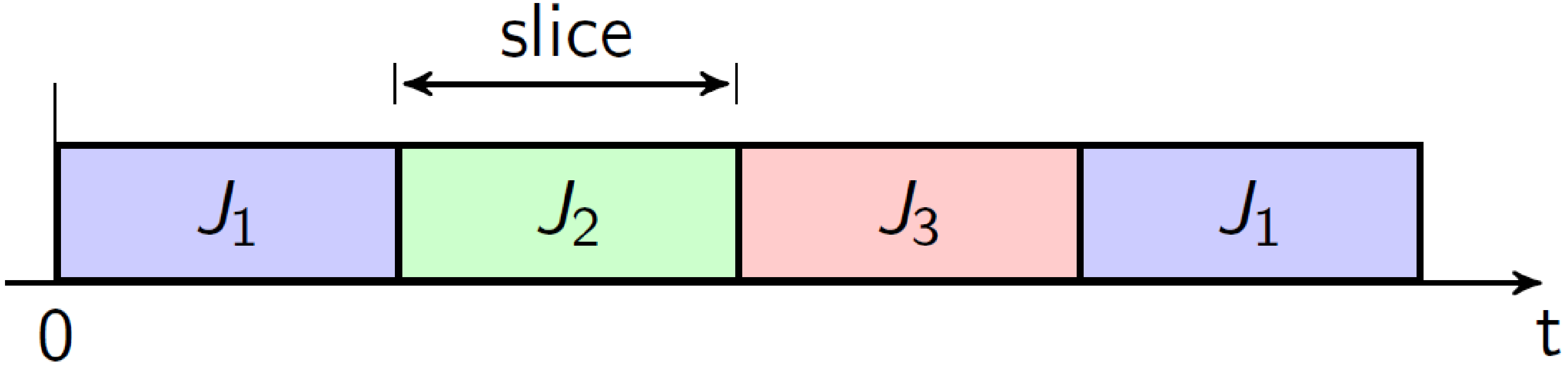
- A job is considered **starved** if
  - There existed a feasible schedule including that job
  - The scheduling policy did not pick that schedule because some other job had a higher priority
    - *Not* precedence

Policy 1: **Round Robin**



- Each job gets a fair **slice** (or **quantum**) on the processor
  - In repeating order
- A workload with n jobs gives each job $\frac{1}{n^{th}}$ of the processor
  - Or less if it needs less
- Very easy to implement
  - Use a queue, rejoin at end after slice is finished
- Requires preemptable jobs

Policy 1: Round Robin Continued

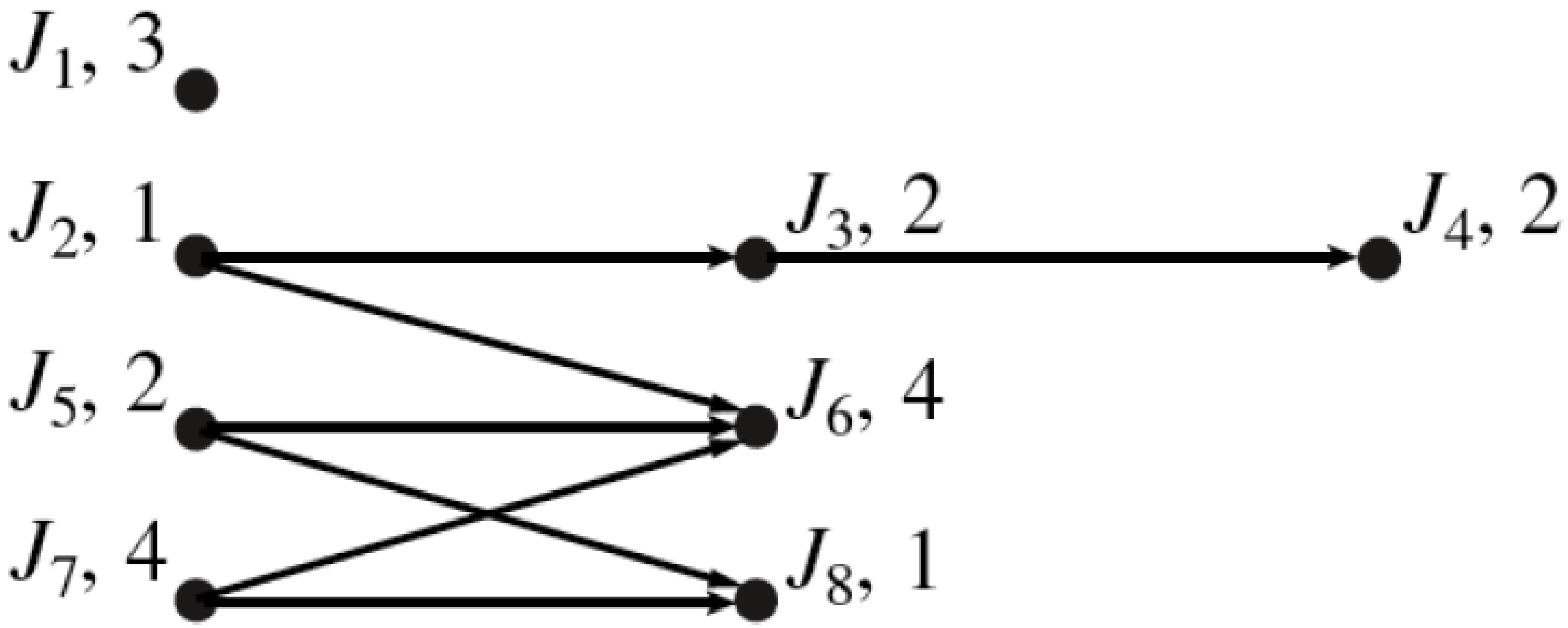


- Can handle sporadic and aperiodic tasks
  - Just add them to the queue
- Does *not* explicitly respect deadlines
  - Cannot always find a feasible schedule even though one might exist
- Variants:
  - **Weighted round robin:** the order of jobs within a round assigned on priority
  - **Dynamic** vs. **pre-allocated:** a priori workloads can by pre-allocated, dynamic workloads also work!

Policy 2: **Simple Priority**

- Run jobs in precedence order
  - If there's no precedence between to jobs, fall back to a **priority**
    - Each job is assigned a priority
      - Static integers assignment (this is **simple priority**)
      - Dynamic runtime priorities based on:
        - Job parameters
        - Physical variables
        - Other
- Easy to implement
  - Use a priority queue
- Does not require preemptable jobs
- **Locally optimal**
  - Will never leave the processor idle if there is a task that could run
- May respect deadlines depending on priority assignment scheme

Simple Priority Example



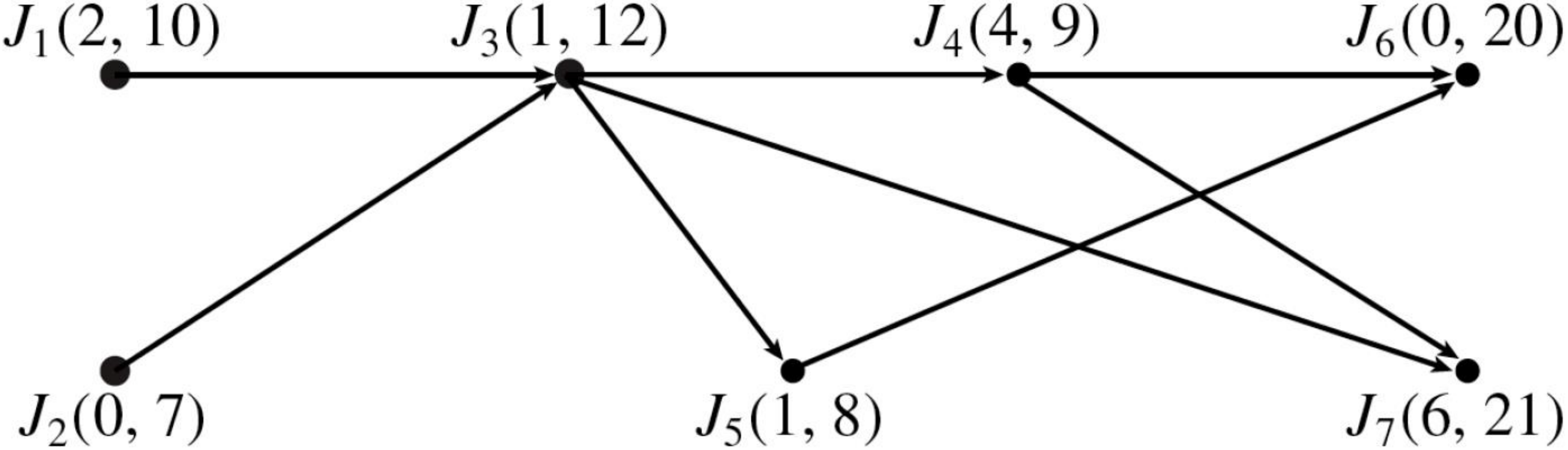
Try a two processor system. Try with and without preemption

Effective Release/Deadline

- r_i^* is the **effective release time** of job i
  - If no precedence $r_i^* = r_i$
  - Otherwise r_i^* is the latest release time between all jobs that job i depends on
  - Fixes the issue where a job depends on another job with a later release time
- d_i^* is the **effective deadline**
  - If no precedence $d_i^* = d_i$
  - Otherwise d is the earliest deadline between all jobs that depend on job i
  - Fixes an issue where a job depends on another job with an earlier deadline

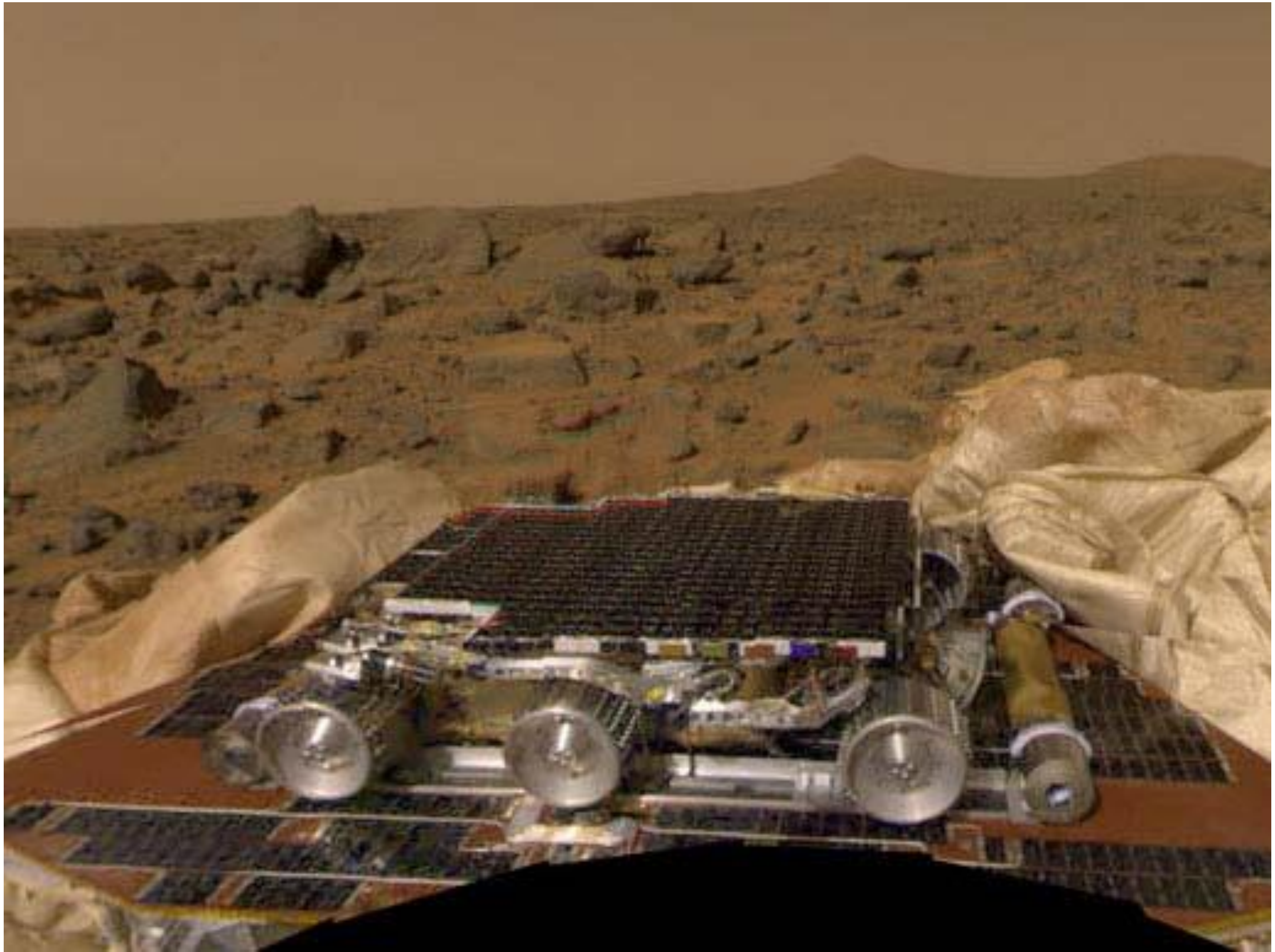
- Can calculate in $O(n^2)$ for a workload of n jobs
- From here on most examples use r_i^* and d_i^* implicitly as r_i and d_i
  - It’s very easy to compute

Effective Release/Deadline Example



Format is $J_i(r_i, d_i)$

Disaster of the Day – Pathfinder (1997)



Online vs. **Offline** Scheduling

- In **Offline** scheduling we know the workload ahead of time
- In **Online** scheduling we do not know the workload ahead of time
- In practice there is nuance here
  - Know some things (like periodic tasks) ahead of time
  - Know which tasks and execution times, but not when
  - Blurry line!

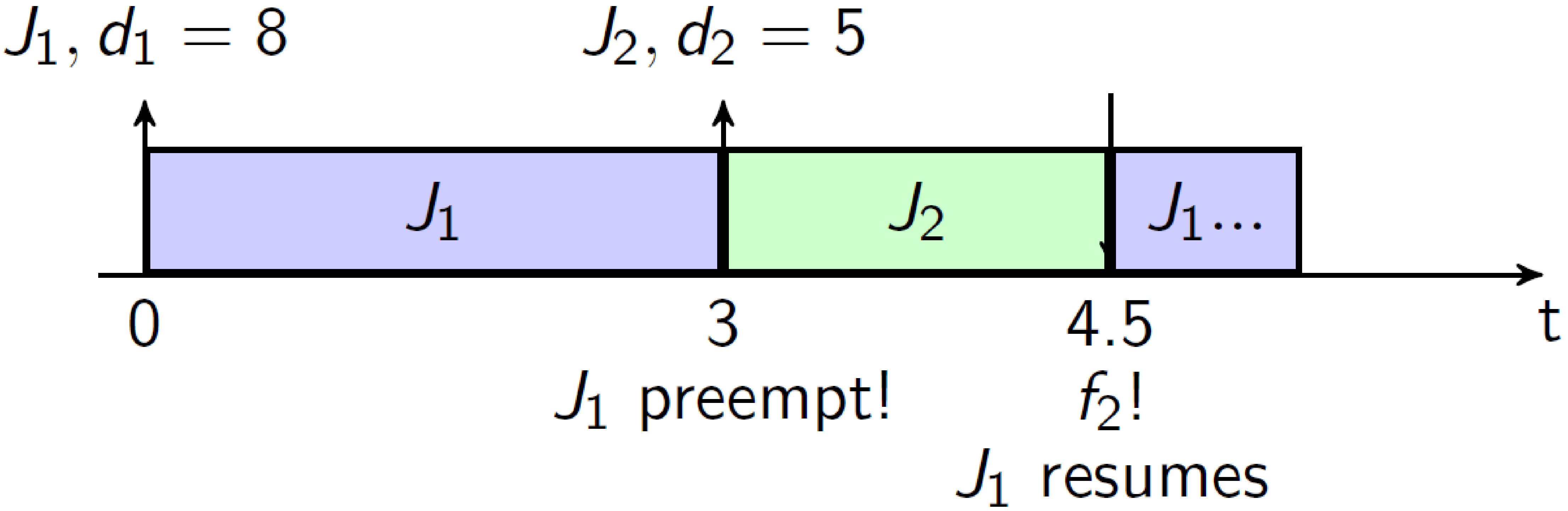
- A **clairvoyant** scheduler knows all the information ahead of time
  - All tasks/jobs, all runtimes, all deadlines, all release times, etc. etc.
  - Usually infeasible

Policy 3: **Earliest Deadline First (EDF)**

- Simple priority, where priority is assigned as
  - $J_i > J_j$ iff $d_i < d_j$
- EDF is **optimal** (both globally and locally) if
  - Preemption is enabled and
  - There is only one processor
- **Optimal** means it will always find a feasible schedule if one exists

EDF Example

- Consider jobs in the format $J_i(r_i, d_i, e_i)$
  - $J_1(0,8,4)$
  - $J_2(3,5,1.5)$



EDF Proof of Optimality

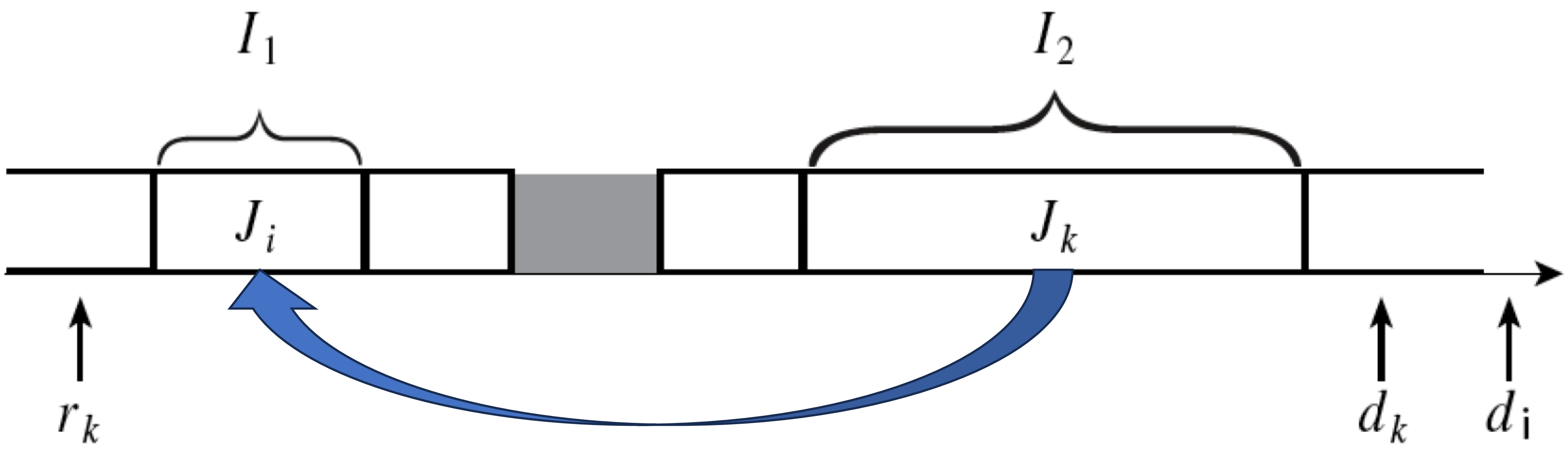
- Proof idea: any feasible schedule can be transformed into an EDF schedule

- Proof basics
  - Some parts of jobs J_i and J_k are scheduled in intervals I_1 and I_2
  - $d_i > d_k$ but I_1 earlier than I_2
    - If I_1 was later than I_2 the schedule is already EDF

EDF Proof of Optimality Continued

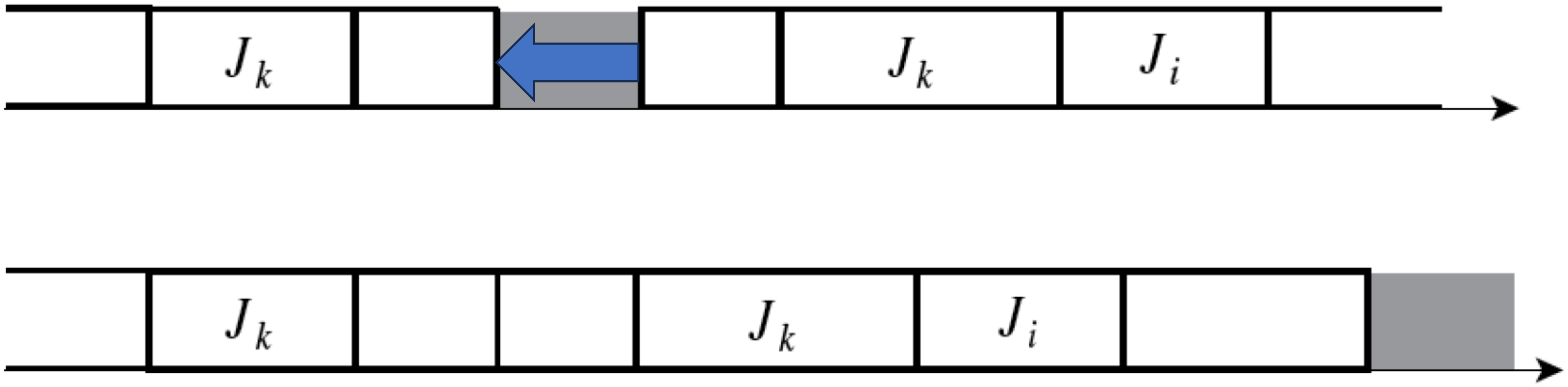
- Case 1: $r_k >$ end of I_1
  - This is an EDF schedule, since J_k couldn't run before J_i was finished

- Case 2: $r_k <$ end of I_1
  - Swap J_i and J_k to fit parts into I_1 and I_2



EDF Proof of Optimality Continued 2

- May need to fill gaps in case 2



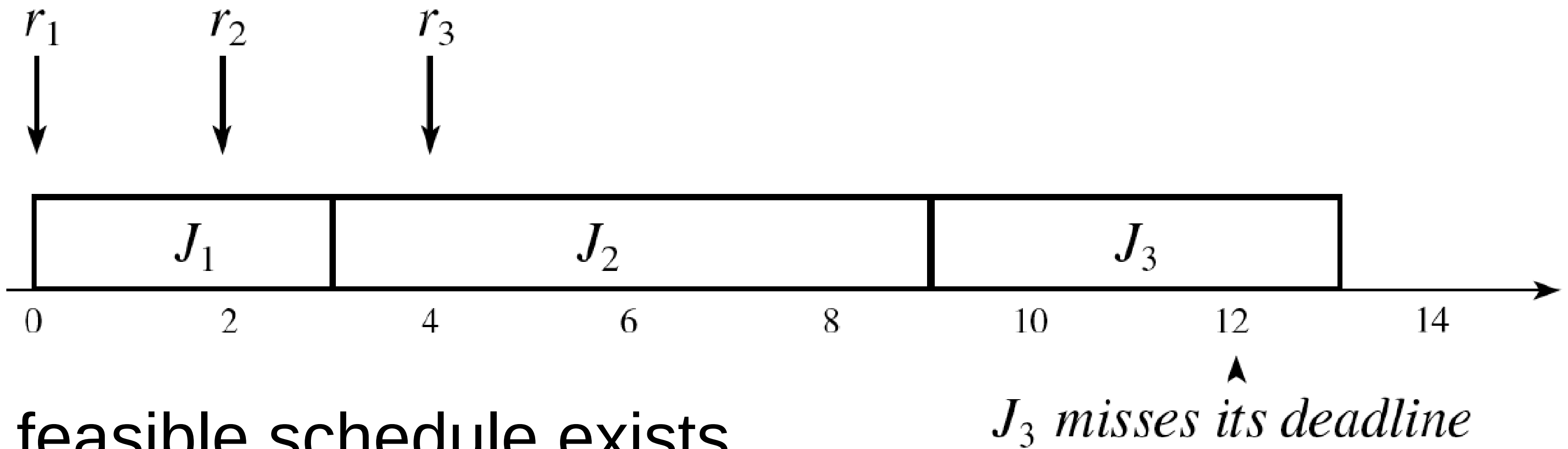
EDF Non-Preemptive Optimality?

- Can we apply this proof if preemption is not available?
  - No. Why?

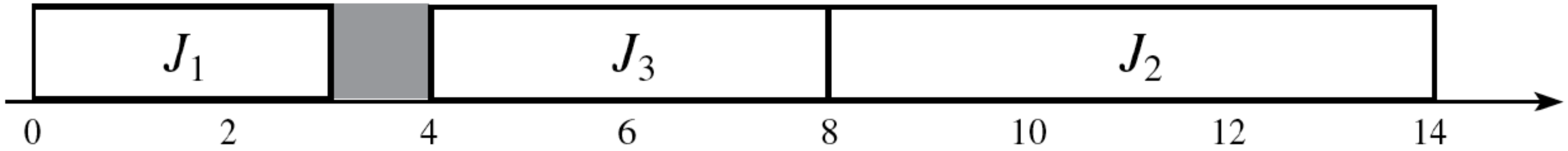
EDF Counterexamples – No Preemption

- Consider jobs in the format $J_i(r_i, d_i, e_i)$
  - $J_1(0{,}10{,}3)$ $J_2(2{,}14{,}6)$ $J_3(4{,}12{,}4)$

- EDF fails!



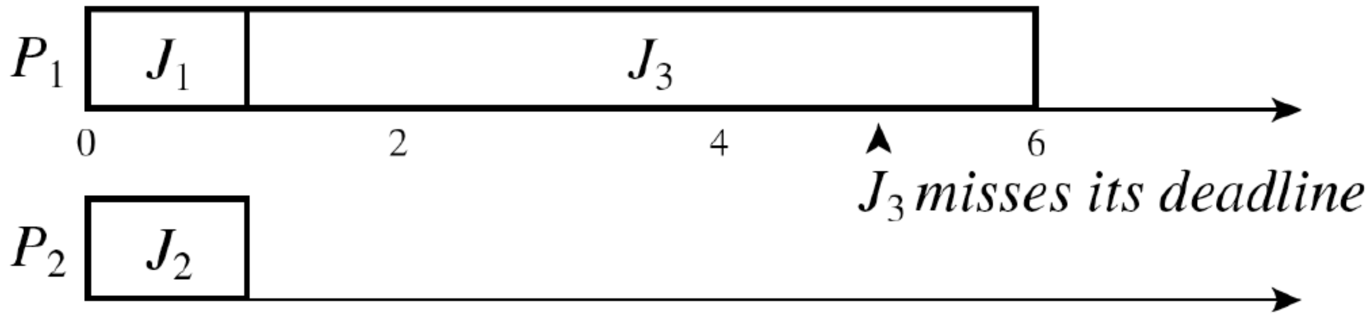
- But a feasible schedule exists



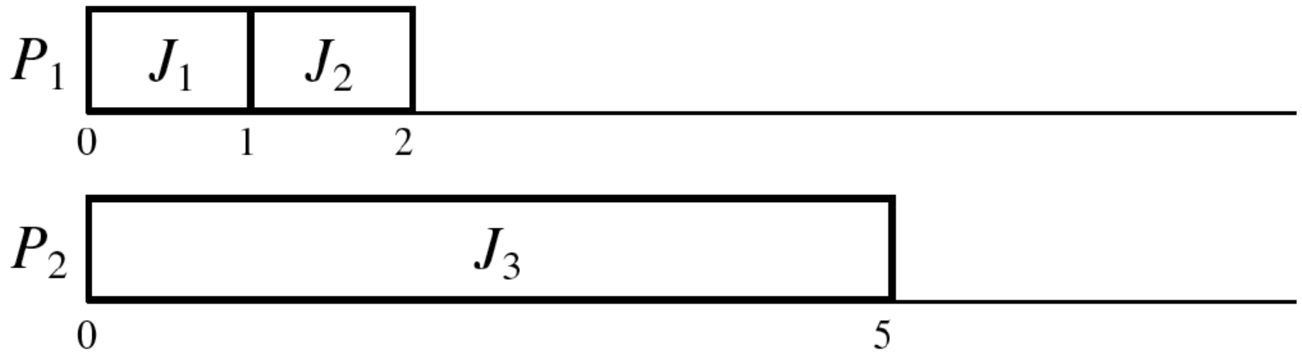
EDF Counterexamples – Two Processors

- Consider jobs in the format $J_i(r_i, d_i, e_i)$
  - $J_1(0,1,1)$ $J_2(0,2,1)$ $J_3(0,5,5)$

- EDF fails!



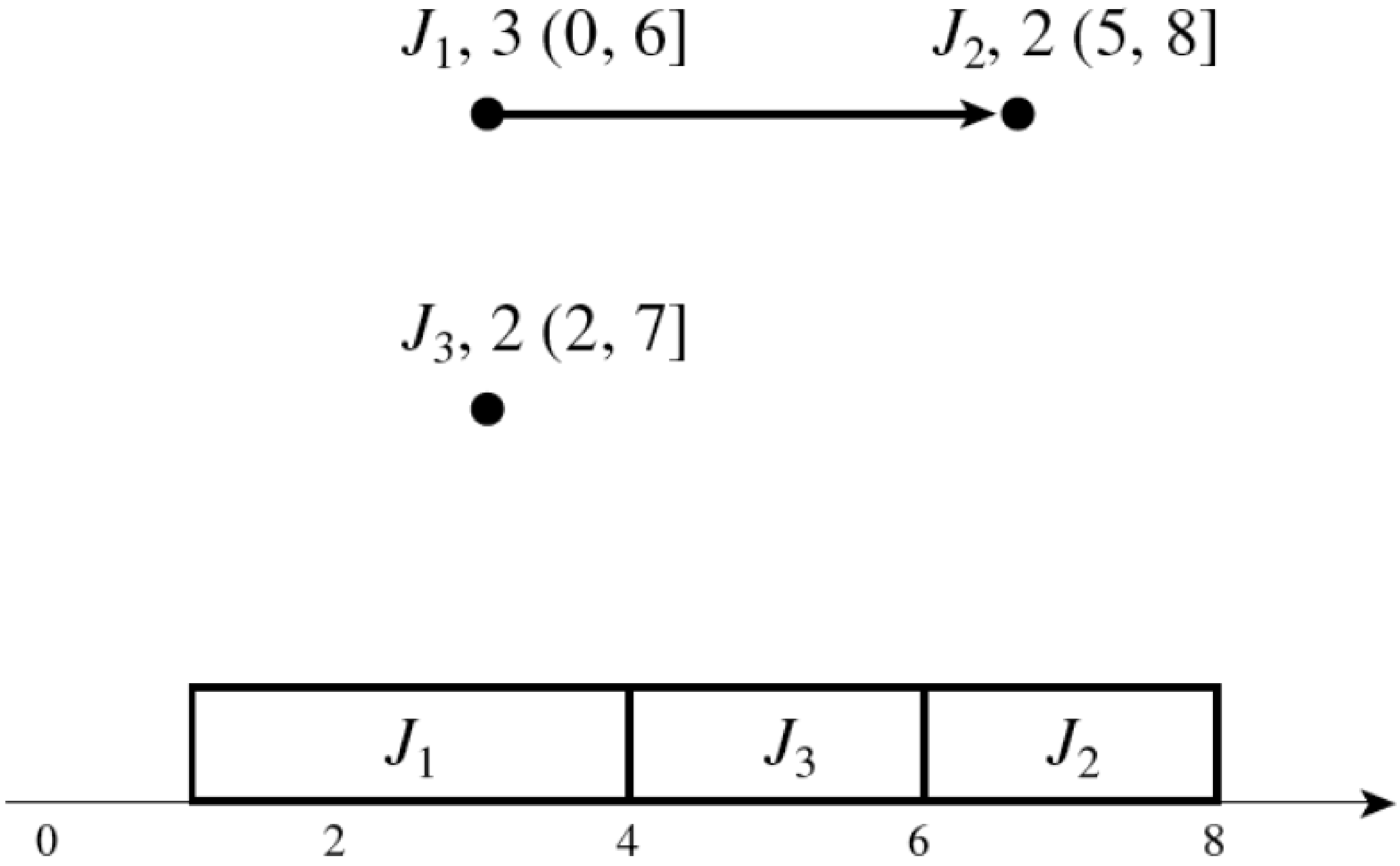
- But a feasible schedule exists



Policy 4: **Latest Release Time (LRT)**

- Almost like simple priority, where priority is assigned as
  - $J_i > J_j$ iff $r_i > r_j$
  - Except that we start at the deadline and work backwards!
- LRT is **optimal** (both globally and locally) if
  - Preemption is enabled and
  - There is only one processor
  - You know the release times ahead of time!
    - Offline only algorithm
- Not a true priority algorithm!
  - Why?

LRT Example



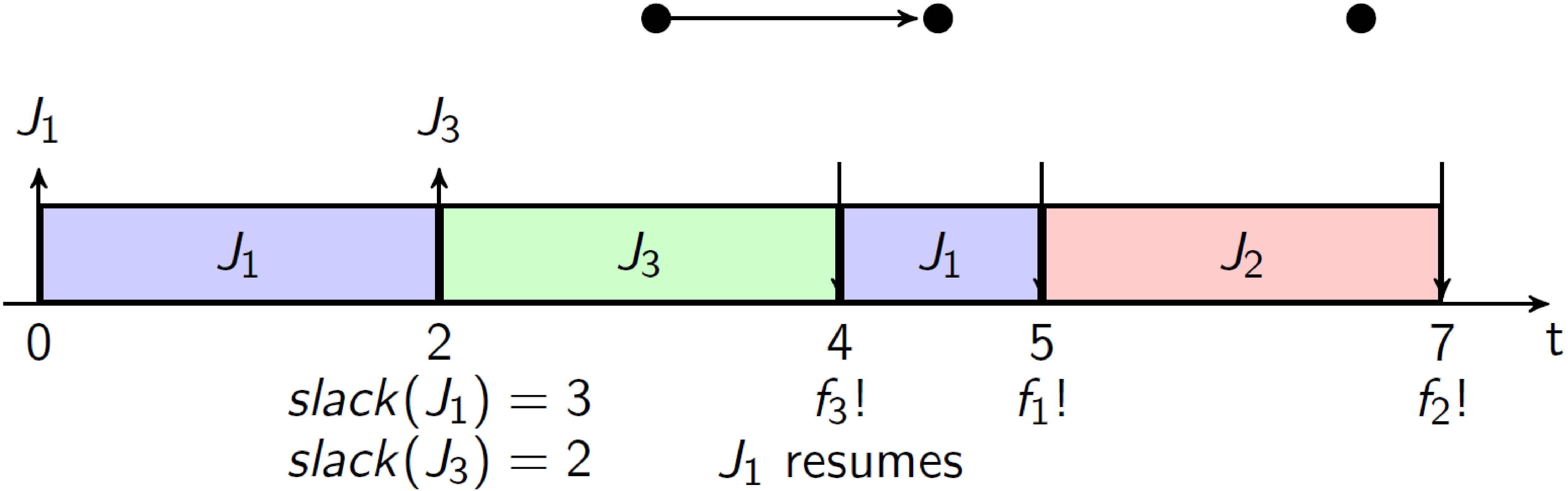
Policy 5: **Least Slack Time First (LST)**

- **Slack time** is the amount of time remaining before the deadline of a task, minus the remaining execution time of that task
- Simple priority, where priority is assigned as
  - $J_i > J_j$ iff $slack_i < slack_j$
- LST is **optimal** (both globally and locally) if
  - Preemption is enabled and
  - There is only one processor
  - Same conditions as EDF

- **Strict** vs. **Relaxed**
  - If two jobs have the same slack, strict would swap back and forth between them repeatedly, relaxed would select one and run it to completion

LST Example

$J_i, e_i\,(r_i, d_i]$ $J_1, 3\,(0, 6]$ $J_2, 2\,(5, 9]$ $J_3, 2\,(2, 6]$



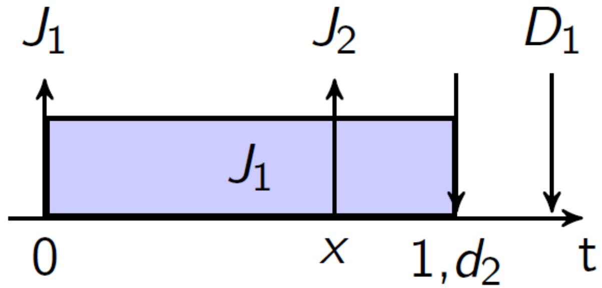
Disaster of the Day – Airbus A400M (2015)

Comparison of "Priority" Approaches

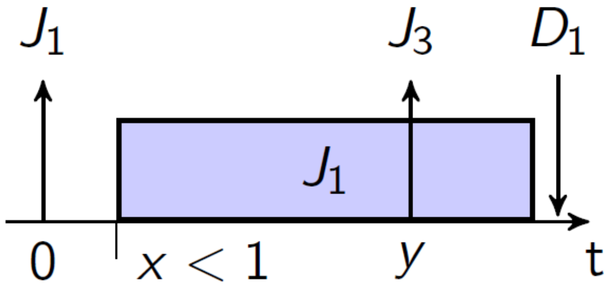
- EDF
  - Well known, robust, simple to implement and use
  - Does not require knowing e_i
- LRT
  - Good fit for soft real-time
  - Handles aperiodic tasks well
  - Requires *precise* WCET, tardiness cascades!
  - Offline
- LST
  - Requires knowing e_i
  - Otherwise just as good as EDF

No optimal online policy without preemption

- Assume a task $J_i(r_i, d_i, e_i)$ with $J_1(0,2,1)$ arrives.
  - It is currently time 0
  - Do we schedule it now, or wait?
- If we do it now:
  - Unlucky! A job $J_2(x, 2, 1 - x)$ arrives at time x. A clairvoyant scheduler would have delayed J_1



- If we delay it to start at time $x < 1$:
  - Unlucky! A job $J_2(y, 2, 1)$ with $y > \mathrm{x}$ arrives at time y. A clairvoyant scheduler would have run J_1 immediately.



Policy 6: **Cyclic Executive**

- Determine a schedule for known periodic tasks offline
  - Leave spaces for sporadic and aperiodic tasks
  - Keep a queue of sporadic and aperiodic tasks at runtime
  - Run schedule and fill spaces with aperiodic and sporadic

- Divide offline schedule into **frames**
  - Fixed length interval in which running jobs are fixed
  - Shorter than hyperperiod

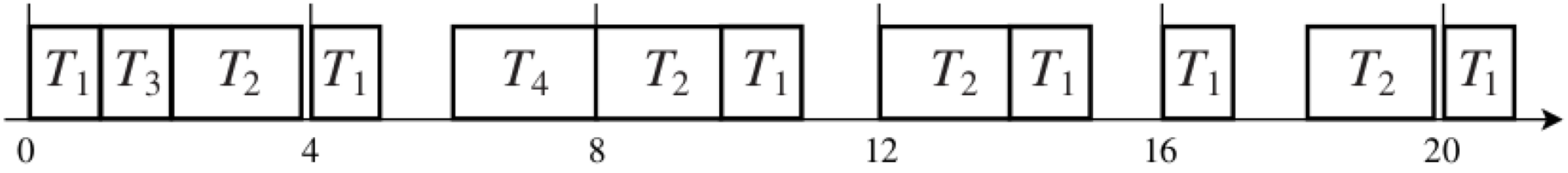
Cyclic Executive Algorithm

- Let $t =$ current global frame number, $F =$ num frames in hyperperiod
- In a timer interrupt at intervals of frame time f
  - Let $k = t \bmod F$
  - Check for tardiness in last job
    - Take appropriate action (see module 4)
  - Look up schedule $L(k)$ in table L for frame k
  - Execute the slices in frame k
  - If there's slack time between/before/after slices
    - Take tasks from aperiodic/sporadic queue
    - Set a timer for start of next slice => run aperiodic/sporadic task
      - If it finishes, stop timer and remove from queue
      - Otherwise preempt, requeue, and run next slice from frame

Cyclic Executive Example Frames

- Consider periodic tasks in the format $J_i(P_i, e_i)$
  - $T_1(4,1)$ $T_2(5,1.8)$ $T_3(20,1)$ $T_4(20,2)$
  - Hyperperiod is 20

- Frames might be:



Cyclic Executive Properties

- No preemption within a frame
  - Guaranteed continuous execution
  - Can make preemption free offline schedule
  - Aperiodic/sporadic tasks cannot be non-preemptable
    - Restricts frame size (must fit longest non-preemptable task)

- Enforce actions at each frame
  - Tardiness
  - Overflowing aperiodic/sporadic queue

Cyclic Executive Frame Time Conditions

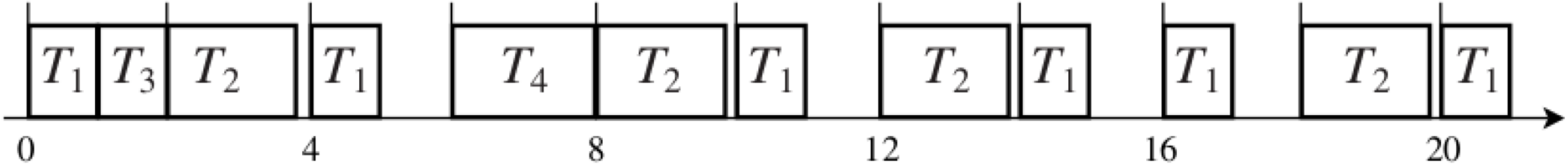
- Frames should be sufficiently long no job needs to be preempted
  - $f \geq \max(e_i)$
- Frame size should evenly divide hyperperiod
  - $\lfloor H/f \rfloor - H/f = 0$
- Frame should be sufficiently small such that between the release time and deadline of every job there is at least one frame
  - $2f - \gcd(P_i, f) \leq D_i$
  - For periodic model this is $2f - \gcd(P_i, f) \leq P_i$

Cyclic Executive Frame Time Example 1

- Consider periodic tasks in the format $J_i(P_i, e_i)$
  - $T_1(4,1)$ $T_2(5,1.8)$ $T_3(20,1)$ $T_4(20,2)$

Time	Tasks released
0	T_1, T_3
2	T_2
4	T_1
6	T_4
8	T_2
10	T_1
12	T_2
14	T_1
16	T_1
18	T_2

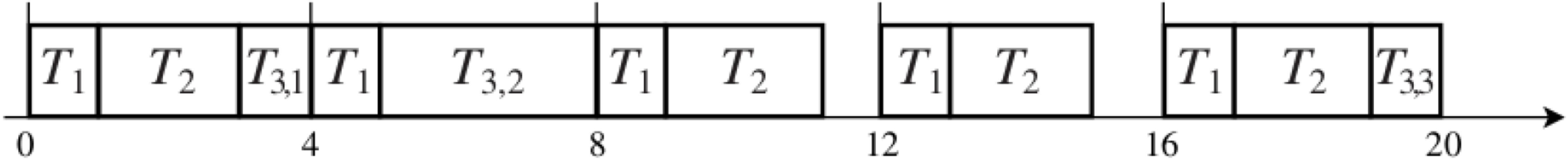
- Condition 1: $f \geq 2$
- Condition 2: $f \in \{ 2, 4, 5, 10, 20 \}$
- Condition 3: $f = 2$



Cyclic Executive Frame Time Example 2

- Consider periodic tasks in the format $J_i(P_i, e_i)$
  - $T_1(4,1)$ $T_2(7,2)$ $T_3(20,5)$

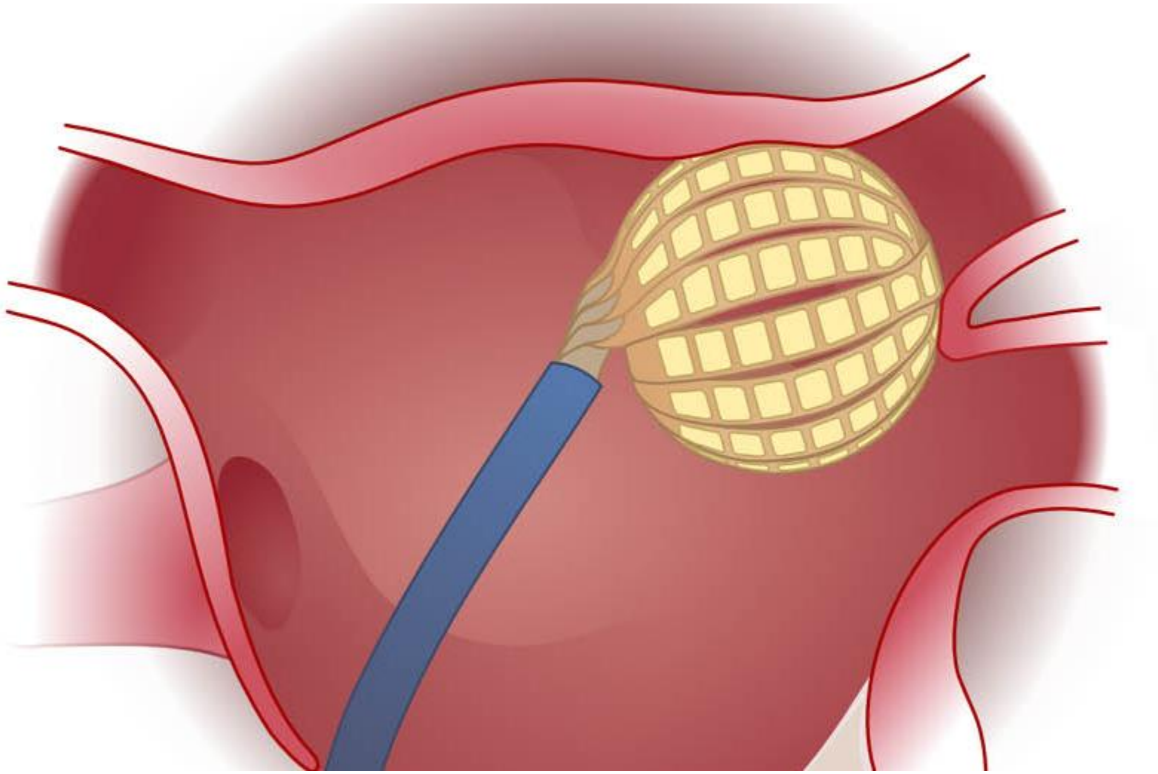
- Condition 1: $f \geq 5$
- Condition 2: $f \in \{1, 2, 4, 5, 7, 10, 14, 20, 28, 35, 70, 140\}$
- Condition 3: ???
- **Can't find f $\rightarrow$ need to slice jobs**



Cyclic Executive **Slack Stealing**

- Move slack time to beginning of frame
  - Run aperiodic/sporadic jobs first
    - Dramatically improve response time
    - Save on preemption if there was multiple slack periods
  - “free” with good offline schedule
  - More vulnerable to bad WCET estimates

Not-a-Disaster of the Day – Globe (20XX-present)



Cyclic Executive **Sporadic Acceptance**

- Instead, have a separate queue for sporadic tasks
  - We know minimum interarrival time
    - Can check if will fit in next frame’s slack time or not
    - Only schedule if we know it will fit

Policy 7: **Rate Monotonic (RM)**

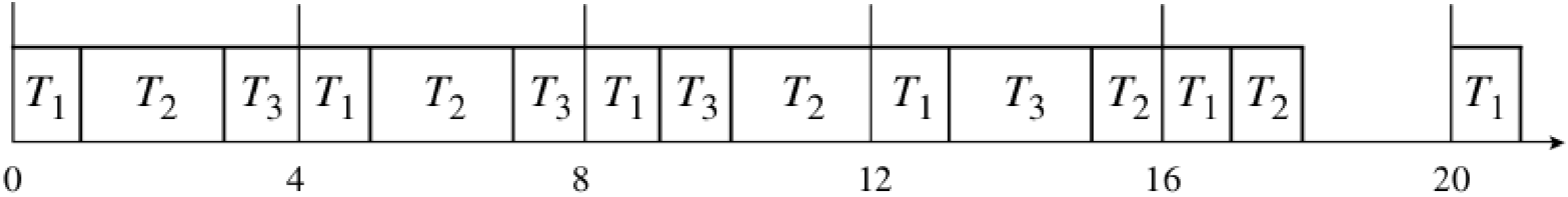
- Simple priority, where priority is assigned as
  - $J_i > J_j$ iff $P_i < P_j$

- RM is **optimal** (both globally and locally) if
  - Preemption is enabled and
  - There is only one processor and
  - Tasks are **harmonic**
    - $\forall T_i, T_j : kP_i = P_j$
    - This is rare in practice

- Very popular!
  - Easy to implement
  - Predictable
    - Longest period jobs are tardy first

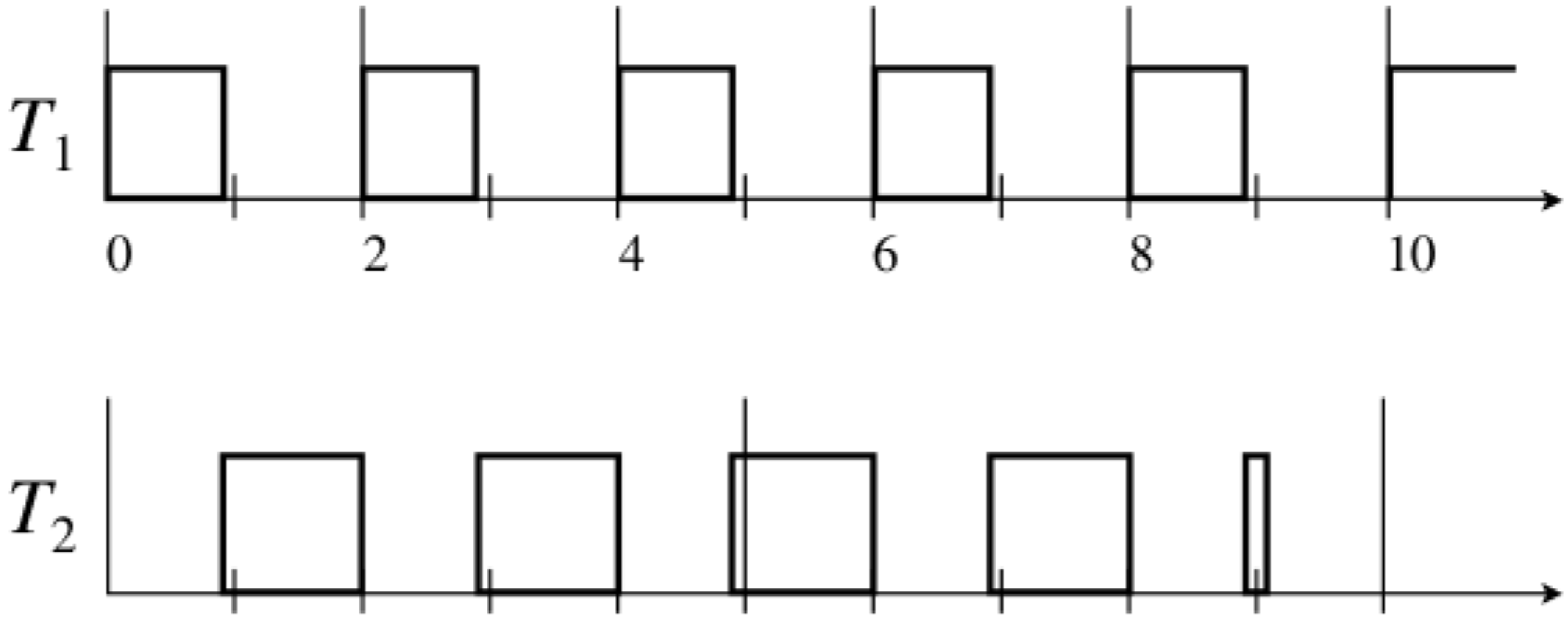
RM Example

- Consider the periodic model with tasks in the format $T_i(P_i, e_i)$
  - $T_1(4,1)$ $T_2(5,2)$ $T_3(20,5)$



RM Example 2

- Consider the periodic model with tasks in the format $T_i(P_i, e_i)$
  - $T_1(2,0.9)$ $T_2(5,2.3)$



Response Time of RM

- The response time of a job J_i in RM is bounded by

$$R_i^n = e_i + \sum_{j \in hp(i)} e_j \left\lceil \frac{R_i^{n-1}}{P_j} \right\rceil$$

  - $R_i^0 = e_i$
  - Repeat until $R_i^n = R_i^{n-1}$
  - Where $hp(i)$ are the tasks with higher priority (lower period) than T_i

- This is the same as interrupt interference time!

Response Time of RM Example

- Consider the periodic model with tasks in the format $T_i(P_i, e_i)$
  - $T_1(9,3)$ $T_2(12,4)$ $T_3(18,3)$

Policy 8: **Deadline Monotonic (DM)**

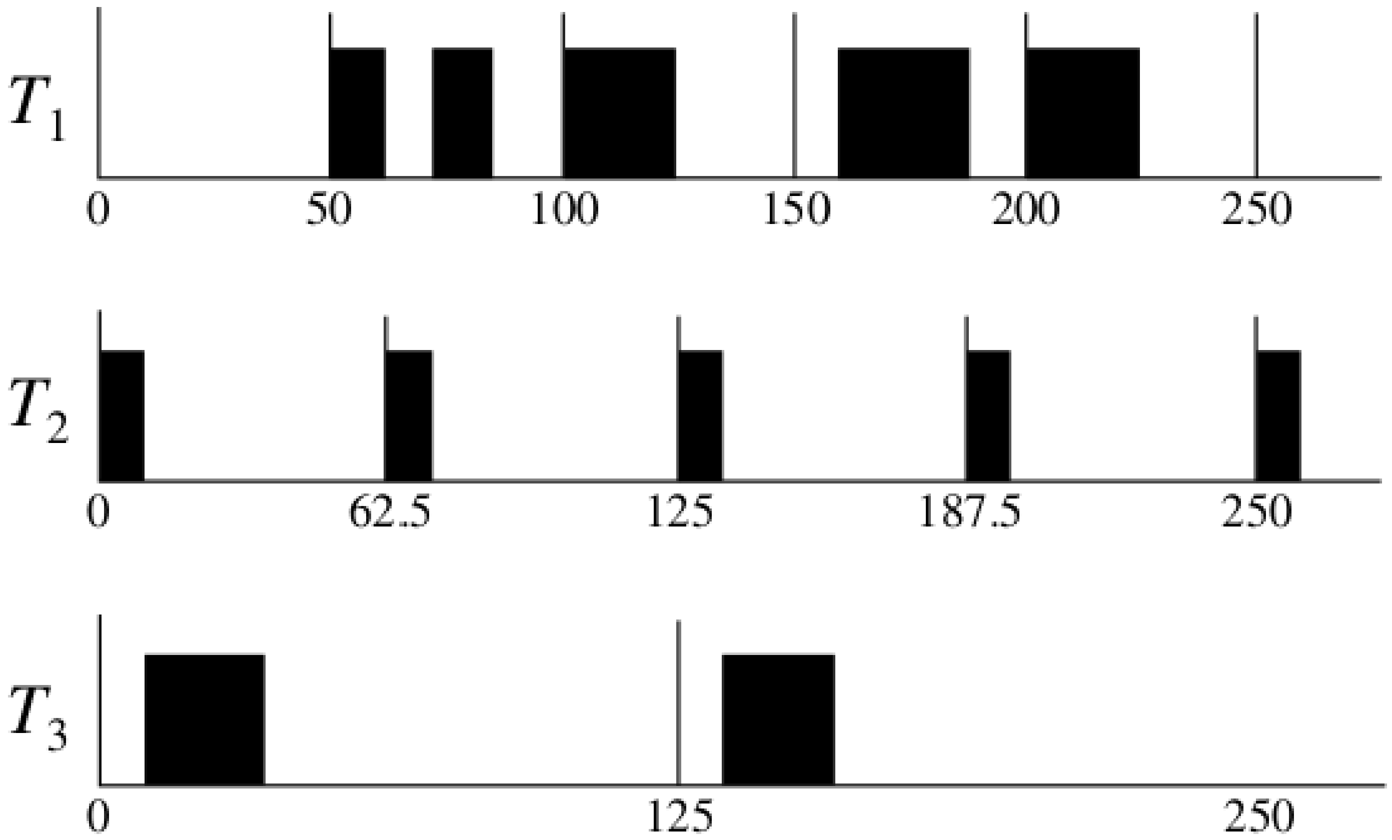
- Simple priority, where priority is assigned as
  - $J_i > J_j$ iff $D_i < D_j$
  - Same as RM when $P_i = D_j$ as in simple periodic model

- DM is **optimal** (both globally and locally) if
  - Same as RM

- But if tasks are non-simply periodic (phase, or $P_i \neq D_j$) DM may find a feasible schedule where RM cannot

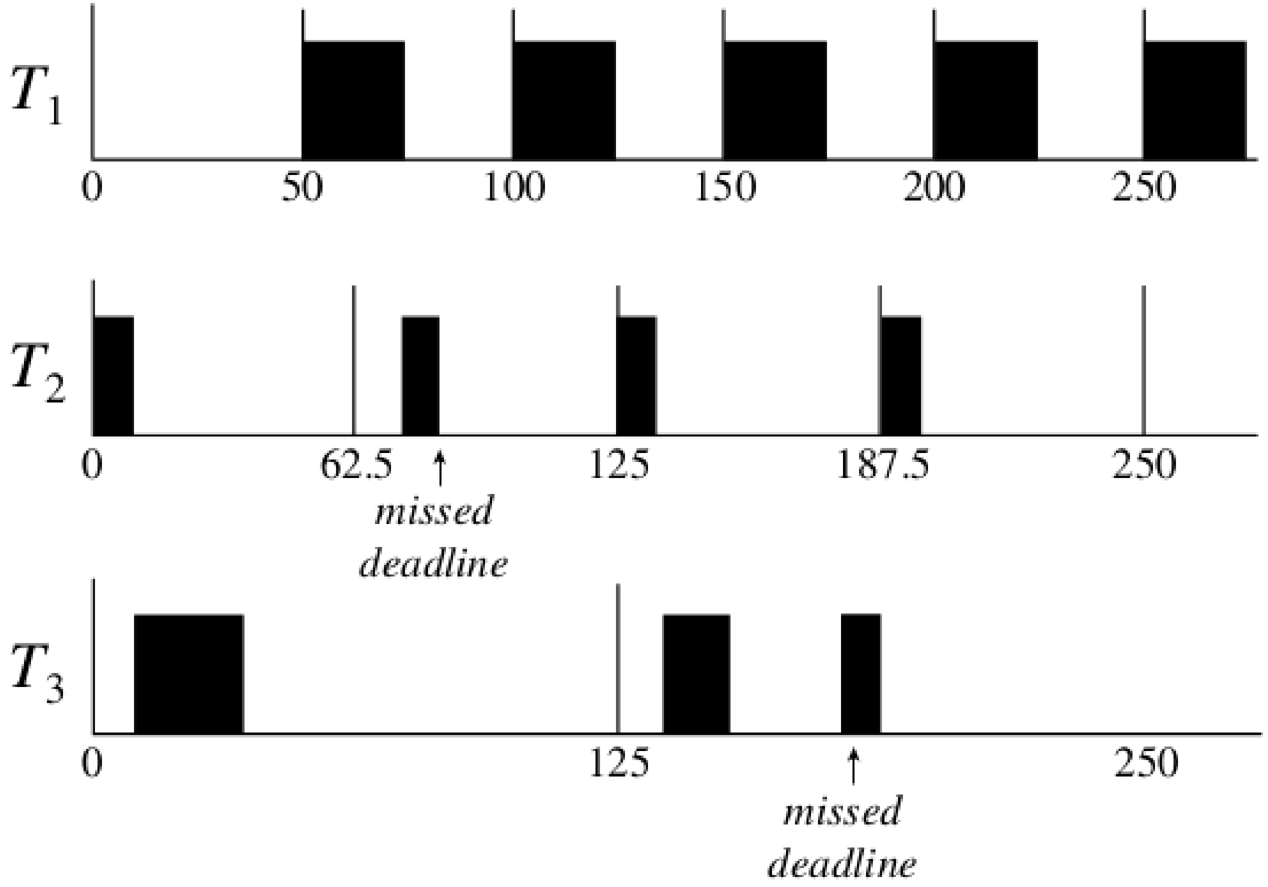
DM Example

- Consider a model with tasks in the format $T_i(\emptyset_i, P_i, e_i, D_i)$
  - $T_1(50{,}50{,}25{,}100)$ $T_2(0{,}62.5{,}10{,}20)$ $T_3(0{,}125{,}25{,}50)$



Same Example, but with Rate Monotonic

- Consider a model with tasks in the format $T_i(\emptyset_i, P_i, e_i, D_i)$
  - $T_1(50{,}50{,}25{,}100)$ $T_2(0{,}62.5{,}10{,}20)$ $T_3(0{,}125{,}25{,}50)$



Disaster of the Day – Zenit SL3

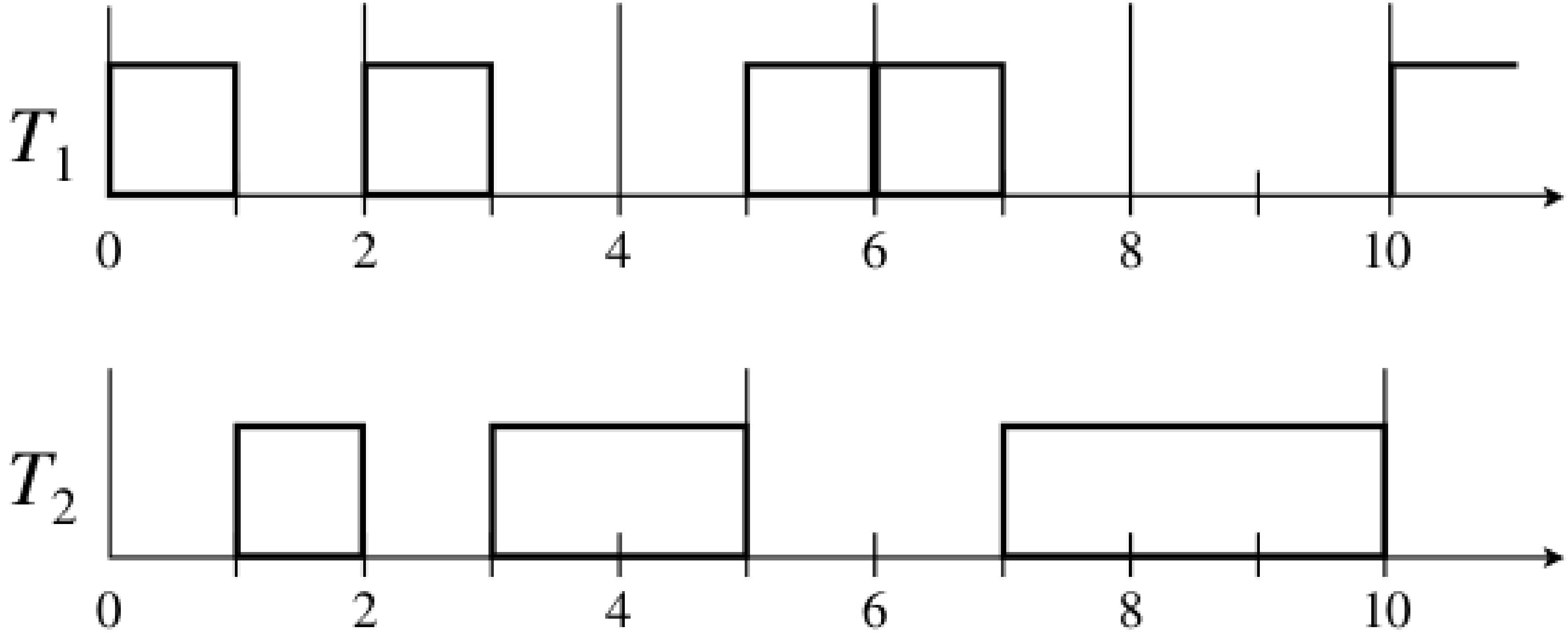
Schedulable Utilization

- A scheduling policy can schedule any periodic workload of its total utilization is equal or less than the **schedulable utilization $\boldsymbol{U}$** of that policy
  - A higher U is better
  - The maximum possible U is 1

- Optimal online priority algorithms have higher U than fixed priority
  - But what about predictability?
    - Fixed priority algorithms are more predictable when they fail

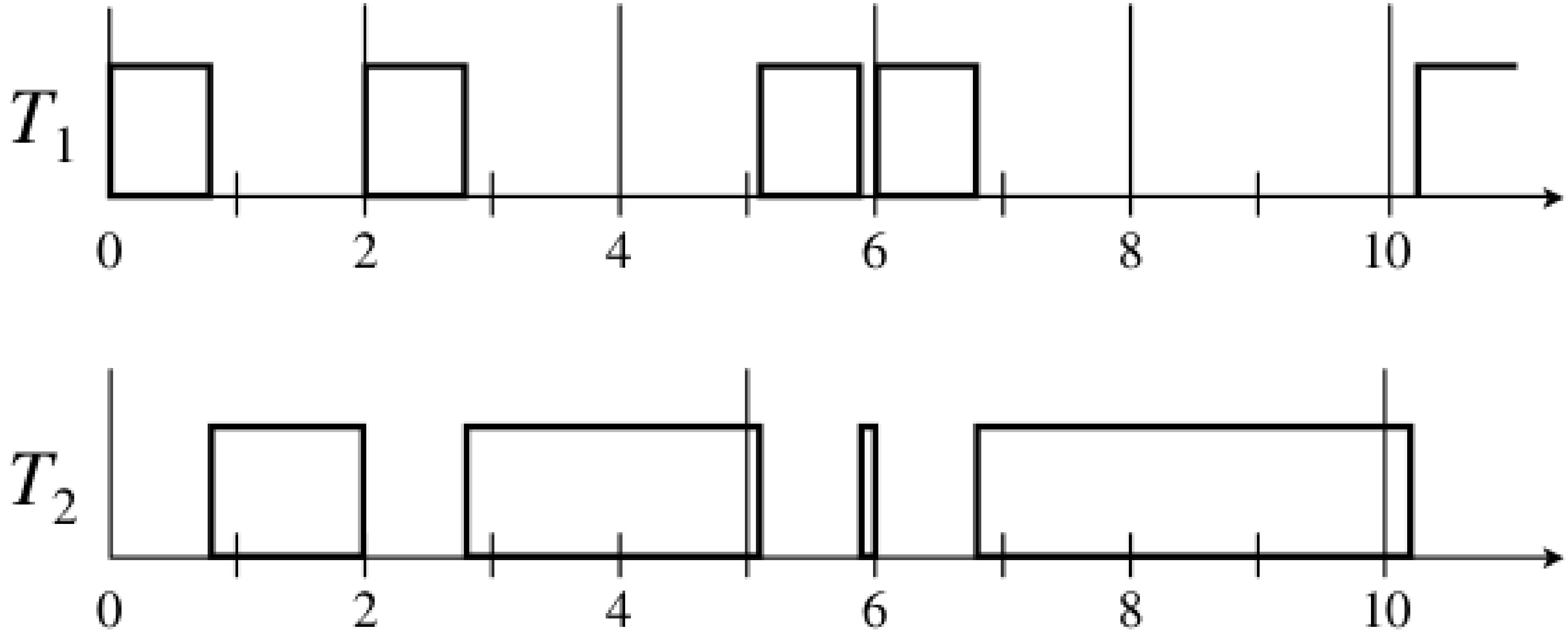
The failures of EDF – Part 1

- Consider the periodic model with tasks in the format $T_i(P_i, e_i)$
  - $T_1(2,1)$ $T_2(5,3)$
  - $U = 1.1$



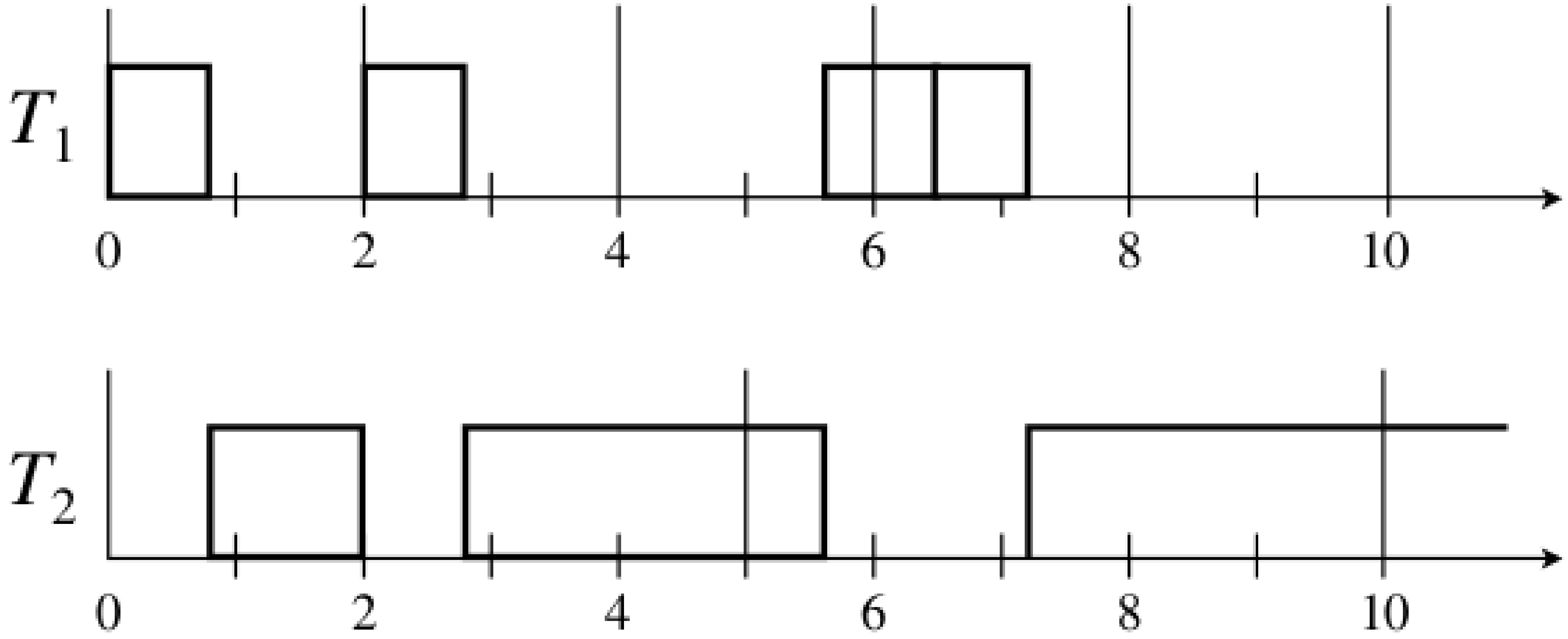
The failures of EDF – Part 2

- Consider the periodic model with tasks in the format $T_i(P_i, e_i)$
  - $T_1(2,0.8) \quad T_2(5,3.5)$
  - $U = 1.1$



The failures of EDF – Part 3

- Consider the periodic model with tasks in the format $T_i(P_i, e_i)$
  - $T_1(2{,}0.8) \quad T_2(5{,}4)$
  - $U = 1.2$



Schedulable Utilization of EDF

- Consequence of EDF optimality
  - *A system of T of independent, preemptable jobs with relative deadlines equal to their respective periods can be feasibly scheduled on one processor if and only if its total utilization is equal to or less than 1.*

- EDF has $U = 1$
- Works for $D_i > P_i$ so long as total utilization still 1 or less

- This gives the schedulability test for EDF as (given EDF assumptions):

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, P_k)} \leq 1$$

Schedulable Utilization of RM

- RM has $U = 1$ iff tasks are harmonic

- If tasks are not harmonic $U = n(2^{1/n} - 1)$
  - For tasks T_1 ... T_n with unique periods
  - Still in the periodic model

- This maintains all the predictability advantages!