# CIS 721 - Real-Time Systems
# **Lecture 2: Real-Time Systems Reference Model**

Mitch Neilsen
**neilsen@ksu.edu**

# Outline

- **Terms and Concepts (Ch. 1-2)**
- **Real-Time Systems Reference Model (Ch. 3)**
- **Commonly Used Approaches For Real-Time Scheduling  (Ch. 4)**
  - Clock-Driven Scheduling (Ch. 5)
  - Priority-Driven Scheduling (Ch. 6-7)

# Definitions

- A **job** is a unit of work that is scheduled and executed by the system (**$J_{i,k}$**).
- A **task** is a set of related jobs that provide some system function **$\tau_i = \{ J_{i,1}, J_{i,2}, \ldots , J_{i,n} \}$**; e.g., the reception of a data frame could be a job that is part of a task that provides time service.
- The **deadline** of a job is the time at which a job must be completed.

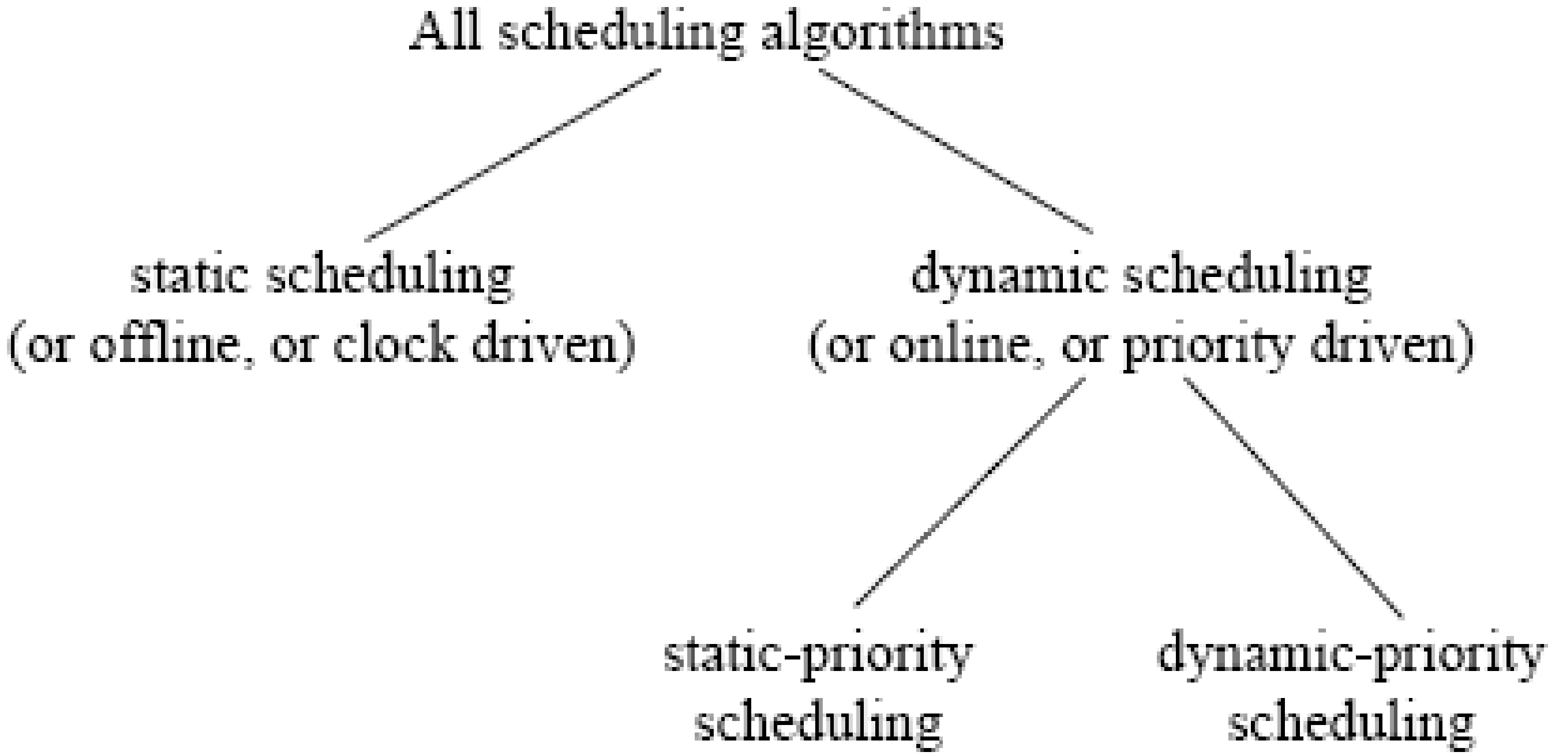
# Deadlines

- The **release time** (or **arrival time**) of a job is the time at which the job becomes available for execution ( $r_i$ or $R_i$ ).
- The **response time** of a job is the length of time between the release time of the job and the time instant when it completes.
- The **relative deadline** of a job is the maximum allowable response time of a job ( $D_i$ ).
- The **absolute deadline** of a job is the time at which a job must be completed ( $d_i = r_i + D_i$ ).

# Algorithms

- We’re interested in two types of algorithms:
  - **Scheduling algorithms** are used to generate to a schedule or priority assignment that can be used by a scheduler to schedule tasks at run time.
  - **Feasibility/schedulability analysis algorithms** are used to determine if a given task set is schedulable.
  - Normally, the second class of algorithms are much more complex than the first class.

# Classification of Scheduling Algorithms



# Static Scheduling Algorithms

- **Static scheduling algorithms** can be used if the scheduling algorithm has complete knowledge of the task set and all timing constraints such as deadlines, execution times, precedence, and **future** arrival times.
- The static algorithm operates on the set of tasks and constraints to generate a single, fixed schedule.

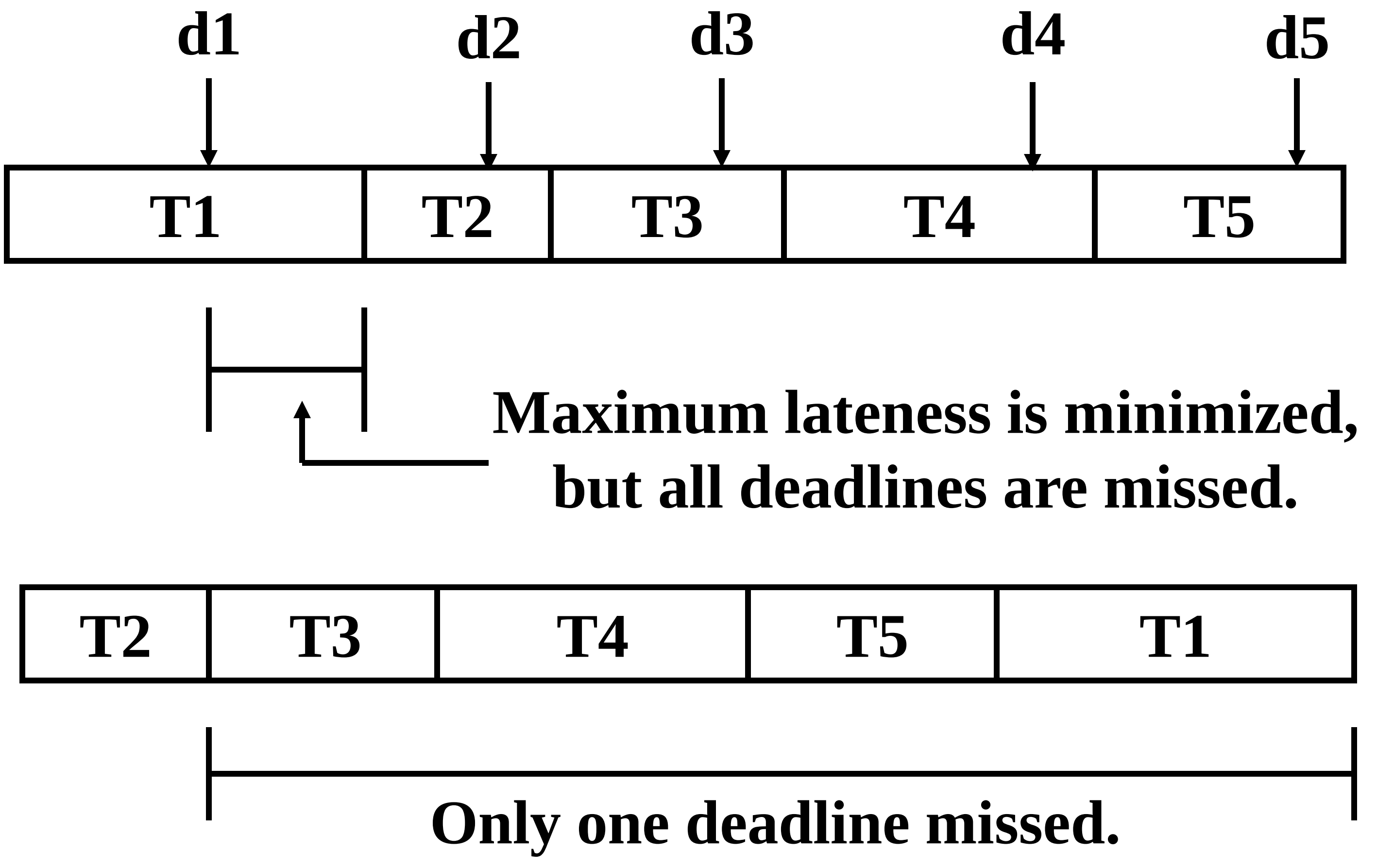
# Dynamic Scheduling Algorithms

- **Dynamic scheduling algorithms** have complete knowledge of **currently active** jobs, but new jobs may arrive at any time in the future.
- Dynamic scheduling is performed at run-time (online); however, offline analysis is usually performed to constrain the dynamic schedule; e.g., assign priorities, etc.

# Metrics used to evaluate scheduling algorithms

- processor utilization
- throughput
- weighted sum of task completion times
- schedule length
- number of processors required
- maximum lateness
- **missed deadlines**

# Minimize maximum lateness



# Missed Deadlines

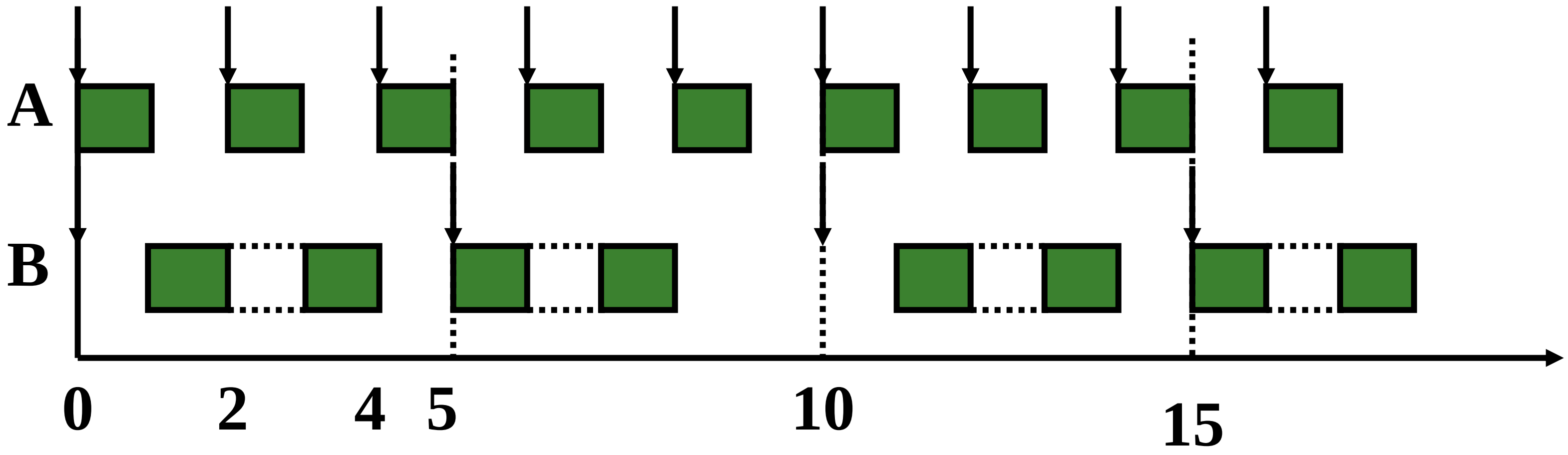
- Much real-time work is only concerned with missed deadlines; e.g., for **hard real-time systems all deadlines must be met**.
- In which case, an **optimal** scheduling algorithm is one that will fail to meet a deadline for any given task set only if no other scheduling algorithm can meet the deadlines.

# Periodic Task

- A periodic task $\boldsymbol{\tau_i}$ **= {** $\mathbf{J_{i,1}, J_{i,2}, \ldots , J_{i,n}}$ **}** is a sequence of jobs with identical parameters with:
  - a **period** ( $p_i$ or $T_i$ ) equal to the length of time between the release times of consecutive jobs,
  - an **execution time** ( $e_i$ or $C_i$ ) equal to the maximum execution time of any job in the task, and
  - a **phase** ( $\varphi_i$ ) equal to the release time of the first job in $\boldsymbol{\tau_i}$ .

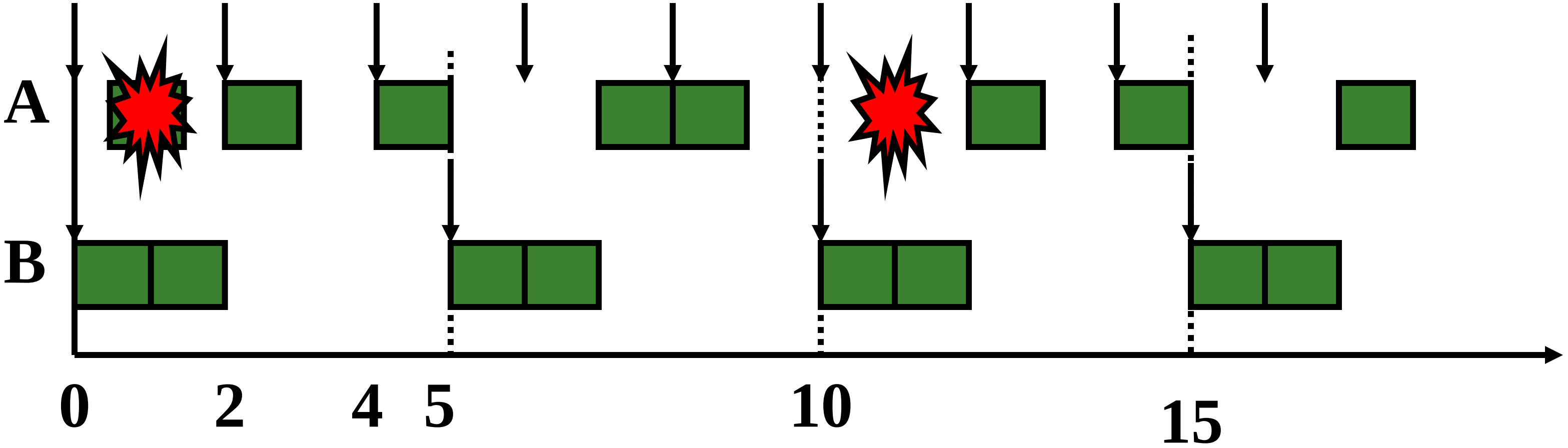
# Example #1 - Priority-Driven Scheduler

| Task $\tau_i$ | | Period $T_i$ | Deadline $D_i$ | Run-Time $C_i$ | Phase $\varphi_i$ |
|---|---|---|---|---|---|
| A | (High Priority) | 2 | 2 | 1 | 0 |
| B | (Low Priority) | 5 | 5 | 2 | 0 |



# Example #2

| Task $\tau_i$ | | Period $T_i$ | Deadline $D_i$ | Run-Time $C_i$ | Phase $\varphi_i$ |
|---|---|---|---|---|---|
| A | (Low Priority) | 2 | 2 | 1 | 0 |
| B | (High Priority) | 5 | 5 | 2 | 0 |



# Example #3

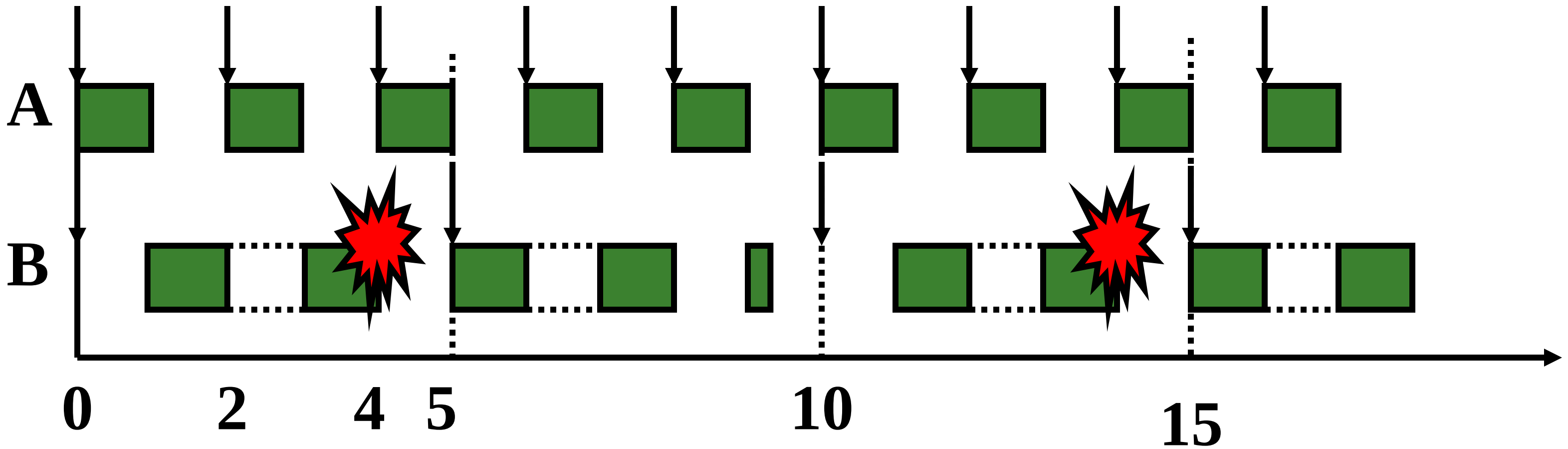
| Task $\tau_i$ | Period $T_i$ | Deadline $D_i$ | Run-Time $C_i$ | Phase $\varphi_i$ |
|---|---|---|---|---|
| A | 2 | 2 | 1 | 0 |
| B | 5 | 5 | 2.1 | 0 |

$U = C_1 / T_1 + C_2 / T_2 = 1 / 2 + 2.1 / 5 = 0.92$

Even if $U < 1$, a task set **may not be schedulable** using fixed priority scheduling.

# Example #3

| Task $\tau_i$ | | Period $T_i$ | Deadline $D_i$ | Run-Time $C_i$ | Phase $\varphi_i$ |
|---|---|---|---|---|---|
| A | (High Priority) | 2 | 2 | 1 | 0 |
| B | (Low Priority) | 5 | 5 | 2.1 | 0 |



# Observations

- The schedulability of a task set depends on priority assignment (Example 1 is schedulable, but Example 2 is not).
- Even if the utilization of a task set is less than one, it may not be schedulable by any fixed priority assignment (Example 3 is not).

# Priority-Driven Scheduling Algorithms for Periodic Tasks

- **Fixed-Priority** - assigns the same priority to all jobs in a task.
- **Dynamic-Priority** - assigns different priorities to the individual jobs in each task.

- After looking at Static Scheduling Algorithms (next time), we will start our investigation of Dynamic Scheduling Algorithms by considering **fixed-priority algorithms**.

# Issues in Fixed Priority Assignment

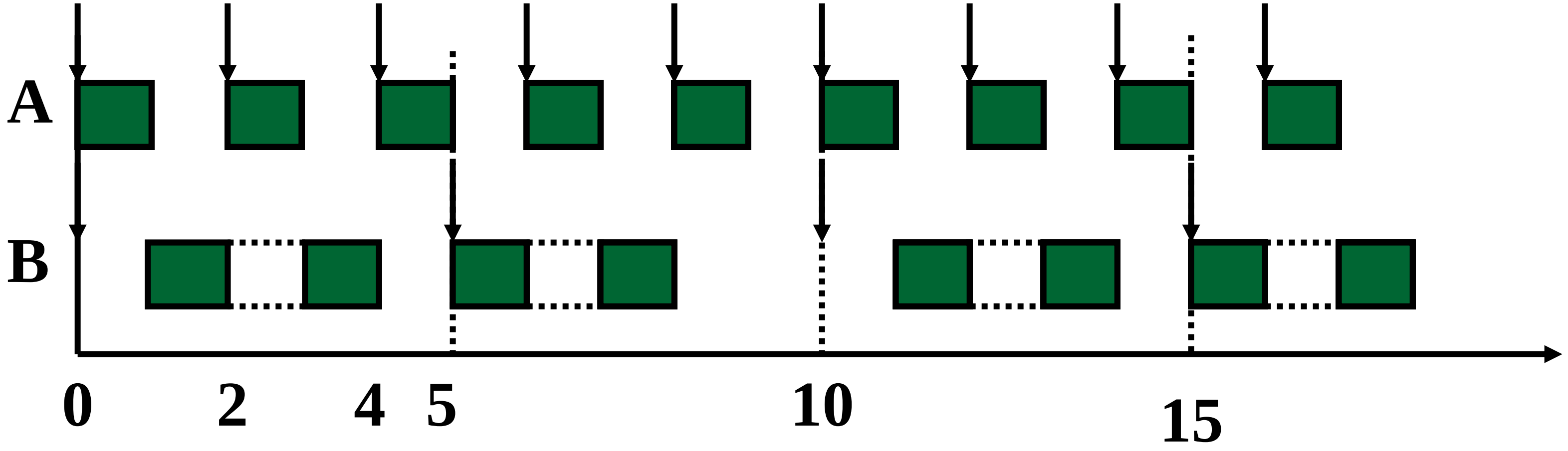
- How to assign priorities?
- How to determine which assignment is the best; e.g., how to evaluate a priority assignment algorithm (method)?
- How to compare different priority assignment algorithms?

# Rate-Monotonic Algorithm (RM)

- The **rate** of a task is the inverse of its period.
- Task with **higher rates** are assigned **higher priorities**.
- C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, JACM, Vol. 20, No. 1, pages 46-61, 1973.

# Example #1 - Rate Monotonic Assignment

| Task $\tau_i$ | | Period $T_i$ | Deadline $D_i$ | Run-Time $C_i$ | Phase $\varphi_i$ |
|---|---|---|---|---|---|
| A | (High Priority) | 2 | 2 | 1 | 0 |
| B | (Low Priority) | 5 | 5 | 2 | 0 |



# Real-Time Reference Model (Ch. 3)

- Idea: Abstract away functional characteristics and focus on **timing requirements and resource requirements**.
- **Reference Model Components**
  - **Resource Graph** – identify available system resources, resource types, and dependencies
  - **Task Graph** – identify task dependencies
  - **Scheduling and Resource Management** – identify algorthithms for scheduling and resource management

# Processors and Resources

- **Processors ($P_i$)** are **active** system resources, such as computers, transmission (tx) links, and database servers
- **Resources ($R_i$)** are **passive** system resources, such as memory, mutexes, semaphores, and database locks
- **Example: Sliding Window Protocol**
  - Job = transmit a message
  - Processor = data link
  - Resource = valid sequence number

# Types of Resources

- **Reusable** – most resources are reusable; e.g., they can be reused by subsequent jobs after being released.
  - Ex: a mutex is a serially reusable resource
- **Plentiful** – a resource is plentiful if no job is ever prevented from executing due to a lack of this resource.
  - Ex: a read-only (immutable) configuration file
  - Plentiful resources are typically removed from the model.

# Resource or Processor?

- For some problems, it is hard to classify system resources as processors or resources.
- This is where experience and the "art" of modeling comes in to play.
- **Example: I/O Bus**
  - In many cases the I/O Bus is viewed as a plentiful resource and ignored in the model
  - However, if we want to study how I/O activities impact real-time performance of an I/O arbitration scheme, then the bus must be modeled as a resource or processor.

# Temporal Parameters

- $J_i$ : **job** – a unit of work
- $T_i$ (or $\tau_i$ ): **task** - a set of related jobs
- A **periodic task** is sequence of invocations of jobs with identical parameters.
- $r_i$: **release time** of job $J_i$
- $d_i$: **absolute deadline** of job $J_i$
- $D_i$: **relative deadline** (or just **deadline**) of job $J_i$
- $e_i$: (Maximum) **execution time** of job $J_i$

# Periodic Task Model

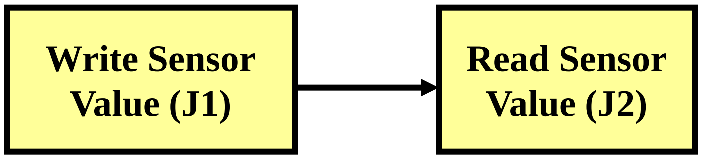
- **Tasks:** $T_1$, ….., $T_n$
- Each consists of a set of **jobs**: $T_i = \{J_{i1}, J_{i2}, \ldots\}$
- $\varphi_i$ : p**hase** of task $T_i$ = time when its first job is released
- $p_i$: p**eriod** of $T_i$ = inter-release time
- H: h**yperperiod** $H = \mathrm{lcm}(p_1, \ldots, p_n)$
- $e_i$: e**xecution time** of $T_i$
- $u_i$: **utilization** of task $T_i$ is given by $u_i = e_i / p_i$
- $D_i$: (relative) **deadline** of $T_i$, typically $D_i = p_i$

# Types of Release Times

- **Fixed** – release times are known values (**periodic**)
- **Jittered:** $r_i \in [r_i^-, r_i^+]$ : release time of job $J_i$ falls within a known interval
- **Sporadic or aperiodic** – release times are unknown
  - A(x) = interarrival time (time between two consecutive jobs) probability distribution
  - B(x) = execution time distribution
- Definitions
  - **Sporadic tasks** have jobs with hard relative deadlines, but **aperiodic tasks** have either soft or no deadlines

# Precedence Constraints/Graphs

- A **precedence graph** reflects data/control dependencies
- **Example: Sensor/actuator (producer/consumer)**

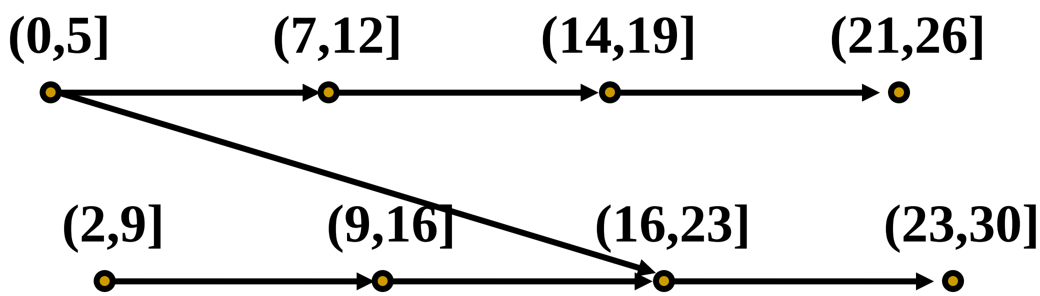



**J1 < J2**

- A precedence relation, denoted < , defines a partial order on the set of jobs.
- $J_i < J_k$ if $J_i$ is a predecesor of $J_k$
- Precedence graph: $G = (\mathbf{J}, <)$, $\mathbf{J} = \{J_1 , J_2 , \ldots \}$
- Precedence constraints can include AND/OR constraints.
- Some dependencies **cannot** be captured by task graphs
  - Example: access to shared data

# Task Graph

- A **task graph** is an extended precedence graph:
  - Vertices denote jobs
  - Edges denote dependencies
  - The label in brackets above each job give its feasible interval ( $r_i$, $d_i$ ] = ( release time, absolute deadline ].



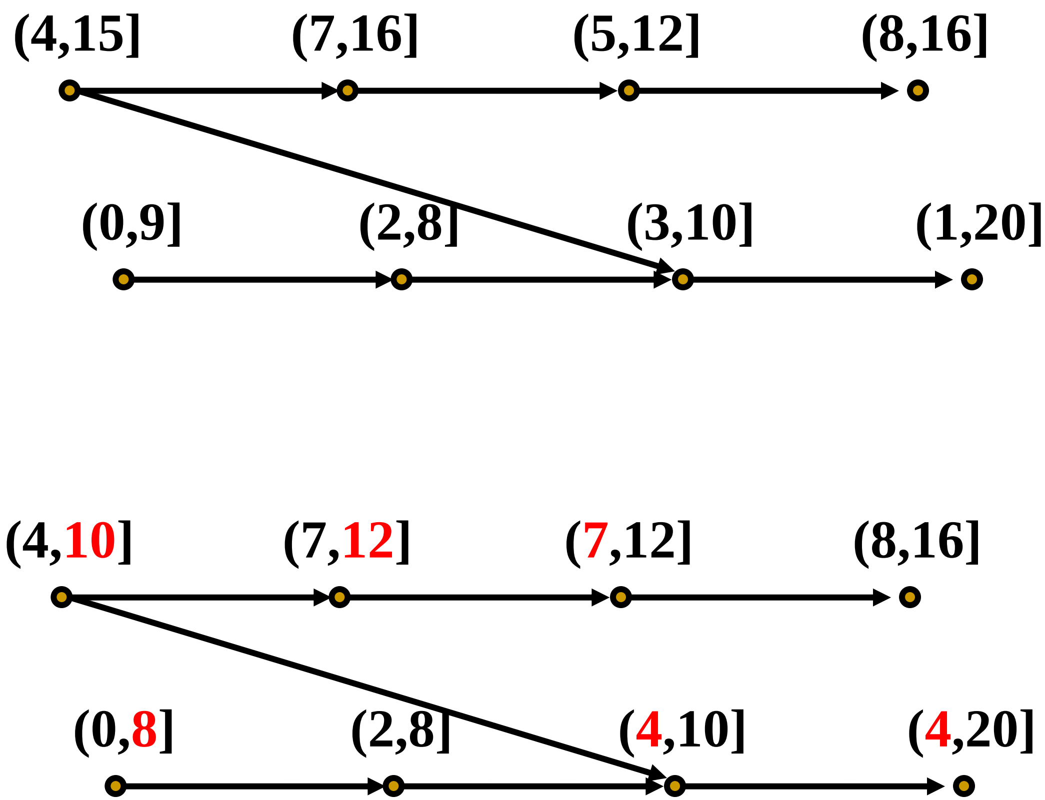
# Effective Timing Constraints

- Timing constraints are often inconsistent with precedence constraints; e.g., $d_1 > d_2$ , but $J_1 < J_2$
- Effective timing constraints on a single processor:
  - **Effective release time**:

    $r_i^{eff} = \mathbf{max}( r_i , \{ r_k^{eff} \mid J_k < J_i \} )$
  - **Effective deadline**:

    $d_i^{eff} = \mathbf{min}( d_i , \{ d_k^{eff} \mid J_i < J_k \} )$
- **Theorem:** A set of jobs **J** can be feasibly scheduled on a processor iff it can be feasibly scheduled to meet all effective release times and effective deadlines.

# Effective Release Times and Deadlines



# Note

- Unless otherwise specified, we will use the terms release time and effective release time interchangeably; likewise, we will use the terms deadline and effective deadline interchangeably.

# System Characterization

- **Preemptivity** - are the jobs preemptable; e.g., can the current task be suspended to assign the processor to a more urgent task?
- **Context-switching time** - is the time required to switch between tasks negligible?
- **Laxity type** - are deadlines hard or soft?
- **Resource requirements** - are any resources required by the job to execute, and for what time interval are these resources required (e.g., critical sections).
- **Criticalness –** can jobs be assigned weights to indicate their importance relative to other jobs? If so, algorithms can be used to optimize weighted performance metrics.

# Schedules

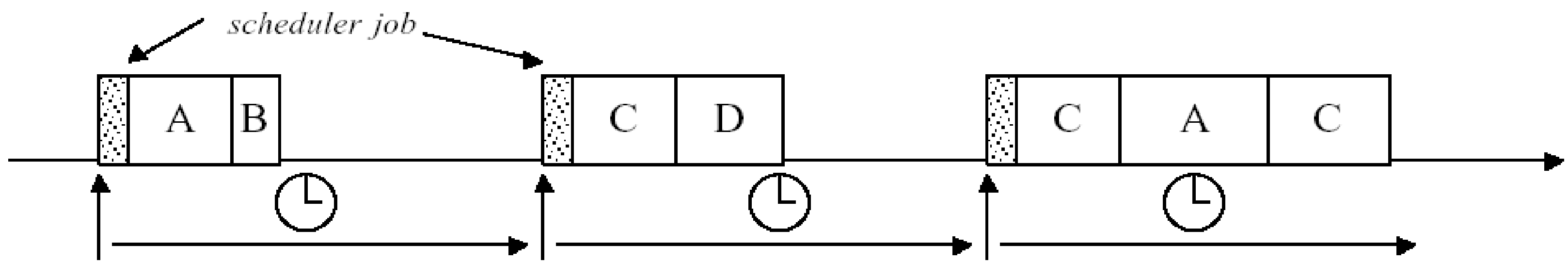
- A **schedule** is an assignment of jobs to available processors. In a **feasible schedule**, every job starts at or after its release time and completes by its deadline.
- In a hard real-time system, a scheduling algorithm is **optimal** if it always produces a feasible schedule if such a schedule exists.
- In a soft real-time system, we can consider different **performance metrics:**
  - Number of **missed deadlines** (tardy jobs).
  - Maximum (or average) **tardiness or lateness**.
  - Maximum (or average) **response time.**

# Common Approaches For Real-Time Scheduling

- **Clock-Driven (Time-Driven) Approach –** scheduling decisions are made at specific time instants.
- **Weighted Round-Robin Approach -** every job joins a FIFO queue; when a job reaches the front of the queue, its weight refers to the fraction of processor time (number of time slices) allocated to the job.
- **Priority-Driven (Event-Driven) Approach -** ready jobs with highest priorities are scheduled for execution first.
  - Scheduling decisions are made when particular events occur; e.g., a job is released or a processor becomes idle. A **work-conserving** processor is busy whenever there is work to be done.

# Clock-Driven Scheduling (Ch. 5)

- **Scheduling decision time**: point in time when scheduler decides which job to execute next.
- Scheduling decision time in clock-driven system is defined *a priori*; e.g., the scheduler periodically wakes up and generates a portion of the schedule.




- When job parameters are known *a priori*, the schedule can be precomputed off-line, and stored as a table (called a **table-driven scheduler**).

# Priority-Driven Scheduling (Ch. 4)

- **Work-conserving** schedulers never leave the processor idle when there is work to be done; e.g., **priority-driven schedulers** typically apply a work-conserving, list-driven, greedy approach for scheduling.
- **Examples:** FIFO, LIFO, SETF (Shortest Execution Time First), LETF, EDF (Earliest Deadline First).
- Possible **implementation** of preemptive, priority-driven scheduling:
  - Assign priorities to jobs.
  - Scheduling decisions are made when: (a) a job becomes ready, (b) a processor becomes idle, or (c) priorities of jobs change
- At each scheduling decision time, choose ready task with highest priority.

# Preemptive vs. Nonpreemptive

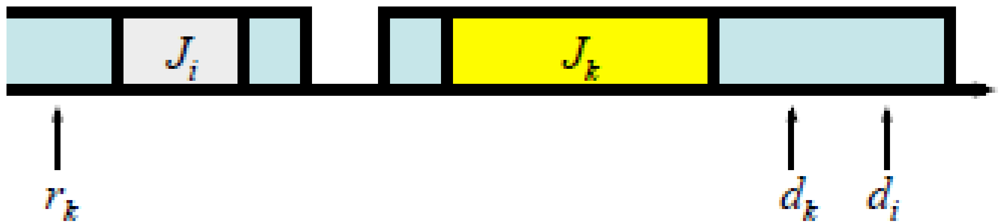
- In the **non-preemptive** case, scheduling decisions are only made when the processor becomes idle (not when a job becomes ready or when priorities change).

# EDF Algorithm

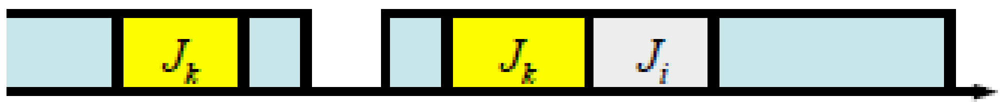
- **Earliest-Deadline-First (EDF) algorithm:**
  - At any time, execute the available job with the **earliest deadline**.
- **Theorem: (Optimality of EDF):** In a system with **one processor** and **preemption** allowed, EDF is optimal; that is, EDF can produce a feasible schedule for a given job set J with arbitrary release times and deadlines, **if** a feasible schedule exists.
- **Proof:** Suppose that a feasible schedule S exists, then apply schedule transformations to S to generate an EDF schedule that is also feasible.

# EDF proof (schedule transformations)

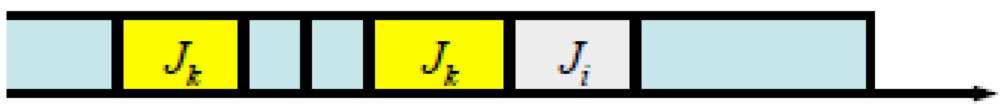
1. Any feasible schedule can be transformed into an EDF schedule
   - If $J_i$ is scheduled to execute before $J_k$, but $J_i$'s deadline is later than $J_k$'s either:
     - The release time of $J_k$ is after the $J_i$ completes $\Rightarrow$ they're already in EDF order
     - The release time of $J_k$ is before the end of the interval in which $J_i$ executes:




       - Swap $J_i$ and $J_k$ (this is always possible, since $J_i$'s deadline is later than $J_k$'s)




       - Move any jobs following idle periods forward into the idle period




     $\Rightarrow$ the result is an EDF schedule

2. So, if EDF fails to produce a feasible schedule, no feasible schedule exists
   - If a feasible schedule existed it could be transformed into an EDF schedule, contradicting the statement that EDF failed to produce a feasible schedule

# EDF may not be optimal

- When preemption is not allowed:

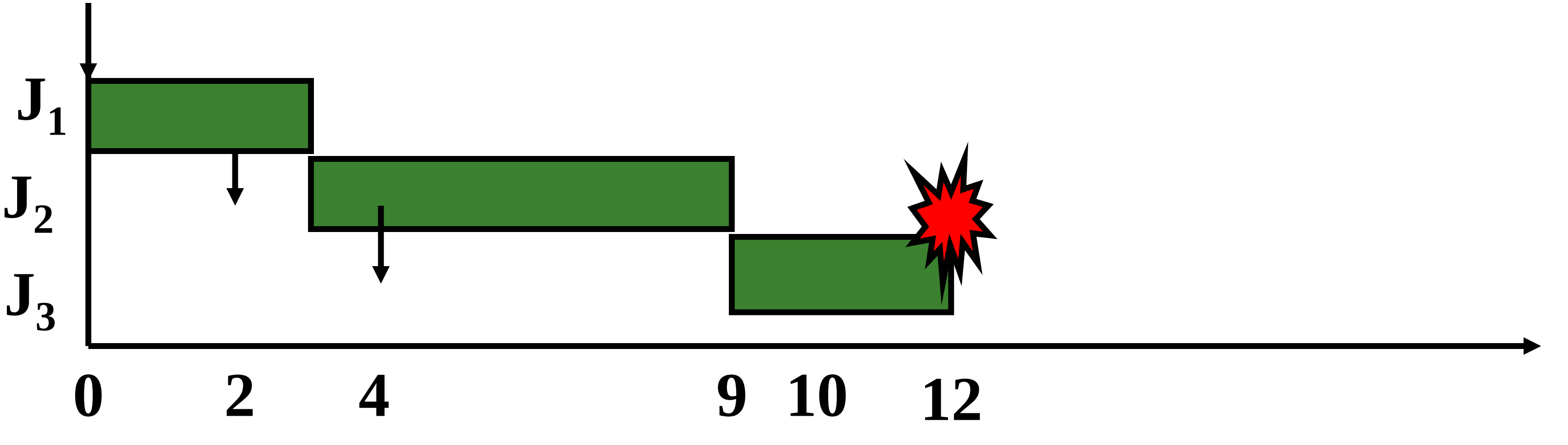
$$
\begin{array}{lcccccc}
 & & & r_i & d_i & e_i & \\
J_1 & = & ( & 0, & 10, & 3 & ) \\
J_2 & = & ( & 2, & 14, & 6 & ) \\
J_3 & = & ( & 4, & 12, & 4 & )
\end{array}
$$

- When more than one processor is used:

$$
\begin{array}{lcccccc}
 & & & r_i & d_i & e_i & \\
J_1 & = & ( & 0, & 4, & 1 & ) \\
J_2 & = & ( & 0, & 4, & 1 & ) \\
J_3 & = & ( & 0, & 5, & 5 & )
\end{array}
$$

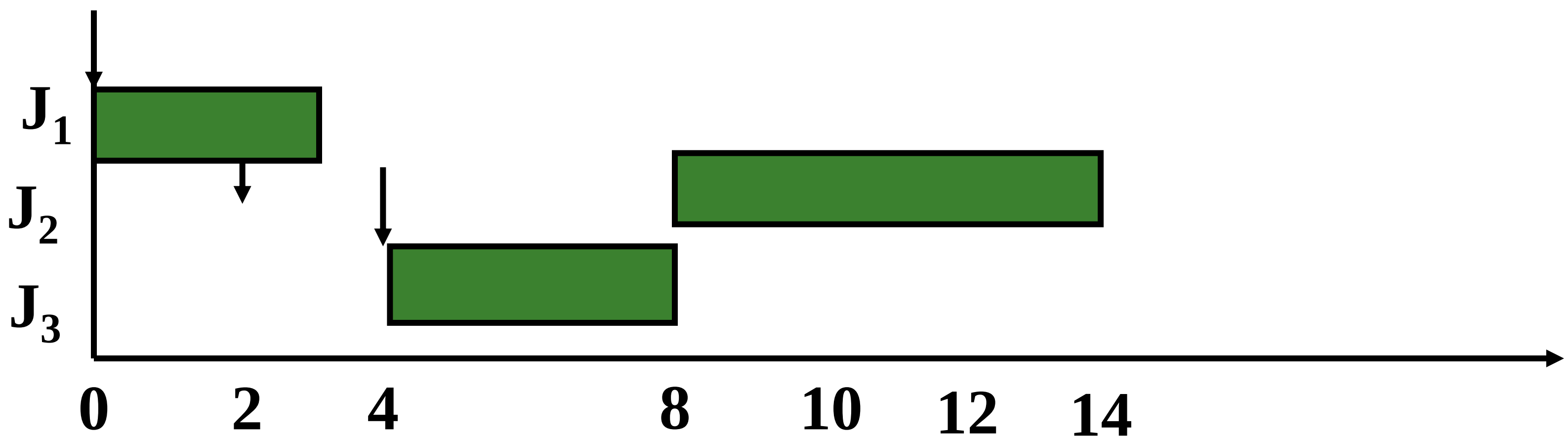
# EDF + Preemption is not allowed

$$
\begin{array}{lcl}
 & & \quad r_i \quad d_i \quad e_i \\
J_1 & = & (\ 0,\ 10,\ 3\ ) \\
J_2 & = & (\ 2,\ 14,\ 6\ ) \\
J_3 & = & (\ 4,\ 12,\ 4\ )
\end{array}
$$



# Optimal + Preemption is not allowed

| | $r_i$ | $d_i$ | $e_i$ |
|---|---|---|---|
| $J_1$ = ( | 0, | 10, | 3 ) |
| $J_2$ = ( | 2, | 14, | 6 ) |
| $J_3$ = ( | 4, | 12, | 4 ) |



# EDF may not be optimal

- When preemption is not allowed:

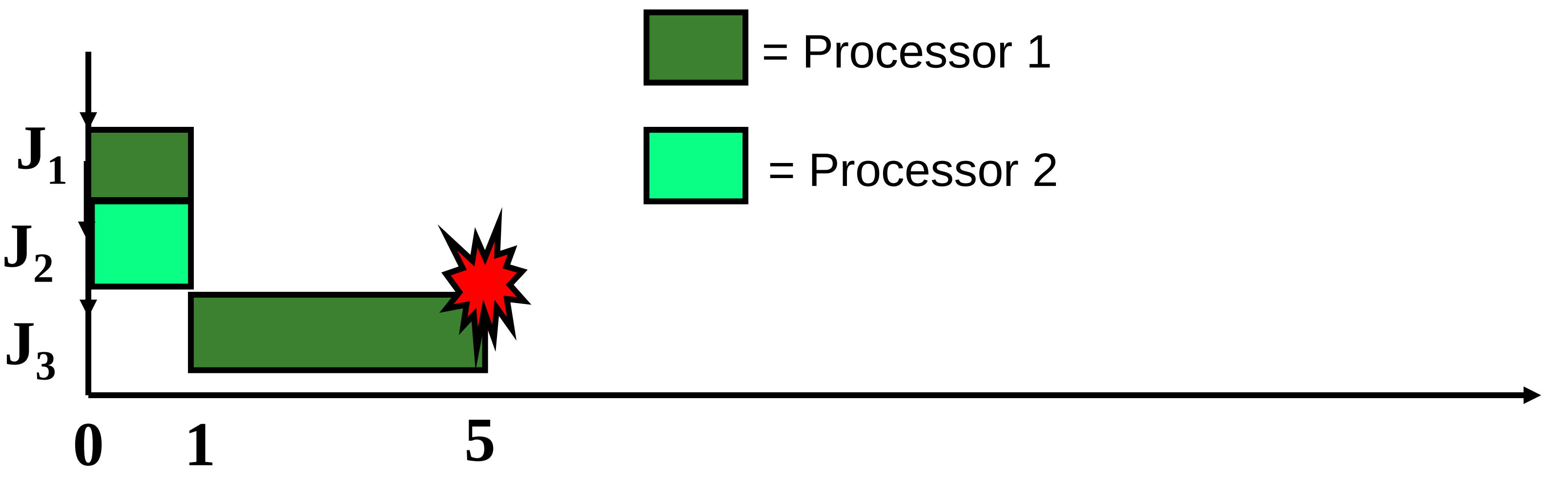
$$
\begin{array}{lcccccc}
 & & & r_i & d_i & e_i & \\
J_1 & = & ( & 0, & 10, & 3 & ) \\
J_2 & = & ( & 2, & 14, & 6 & ) \\
J_3 & = & ( & 4, & 12, & 4 & )
\end{array}
$$

- When more than one processor is used:

$$
\begin{array}{lcccccc}
 & & & r_i & d_i & e_i & \\
J_1 & = & ( & 0, & 4, & 1 & ) \\
J_2 & = & ( & 0, & 4, & 1 & ) \\
J_3 & = & ( & 0, & 5, & 5 & )
\end{array}
$$

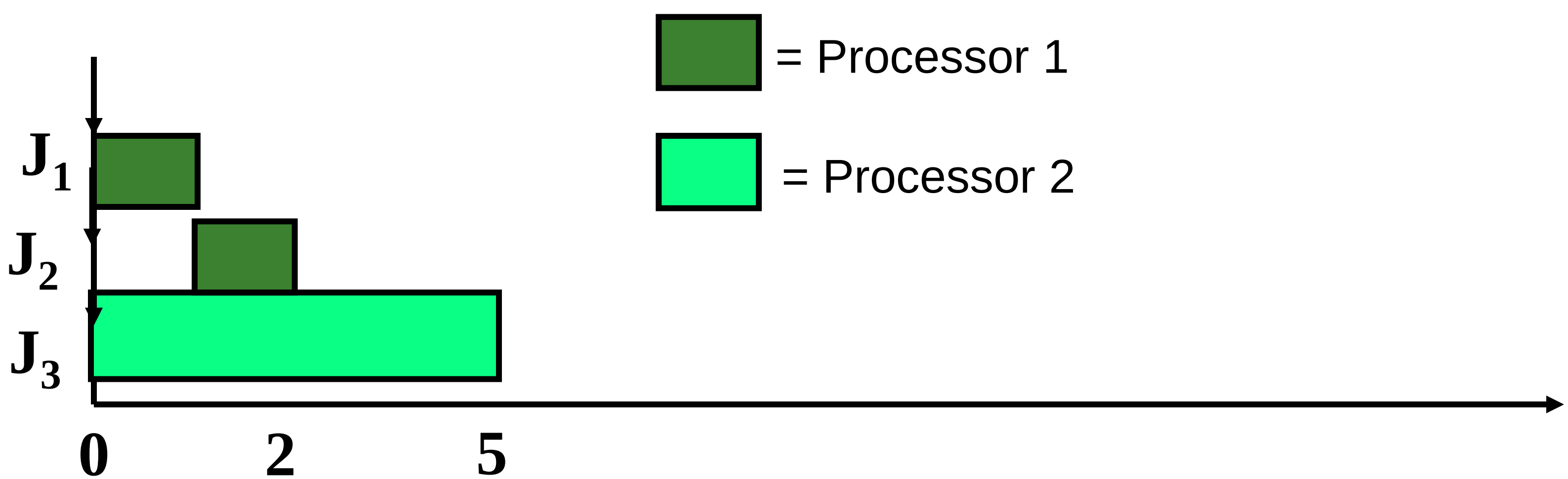
# EDF + More than one processor

$$\begin{array}{llllll} & & & r_i & d_i & e_i \\ J_1 & = & ( & 0, & 4, & 1 & ) \\ J_2 & = & ( & 0, & 4, & 1 & ) \\ J_3 & = & ( & 0, & 5, & 5 & ) \end{array}$$



# Optimal + More than one processor

$$\begin{array}{lcccccc} & & & r_i & d_i & e_i & \\ J_1 & = & ( & 0, & 4, & 1 & ) \\ J_2 & = & ( & 0, & 4, & 1 & ) \\ J_3 & = & ( & 0, & 5, & 5 & ) \end{array}$$



# For Next Time

- Read Ch. 1-3, 5.
- Static Cyclic Scheduling (Ch. 5)
- After that, Real-Time Scheduling – Commonly Used Approaches (Ch. 4)