

Lecture 13:

Real-Time Scheduling

CS 3281

Motivation – Cyber-Physical Systems



Surgical Robotics



Industrial Internet of Things (IIoT)



Power and Utilities



Satellites



Autonomous Vehicles



Drones & DoD Systems

Real-Time Systems

Enterprise Systems



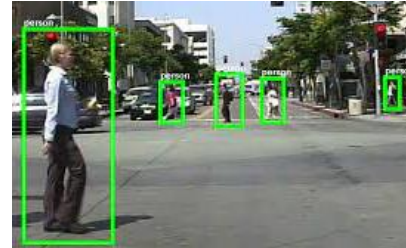
Servers, desktops, web browsing, emails, etc.

"Real Fast" Systems



Interactive processing, i.e., video games

Soft Real-Time System



Pedestrian Detection

Hard Real-Time System



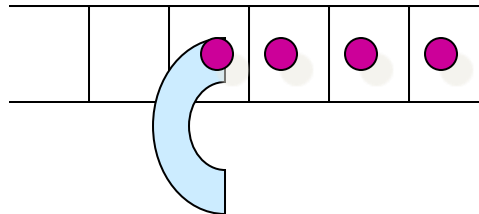
Arc-Flash Relays: ~2ms to break circuit



Interaction with the physical world requires keeping time with the real world.
Many CPS, especially safety- and mission-critical systems have strict timing requirements.

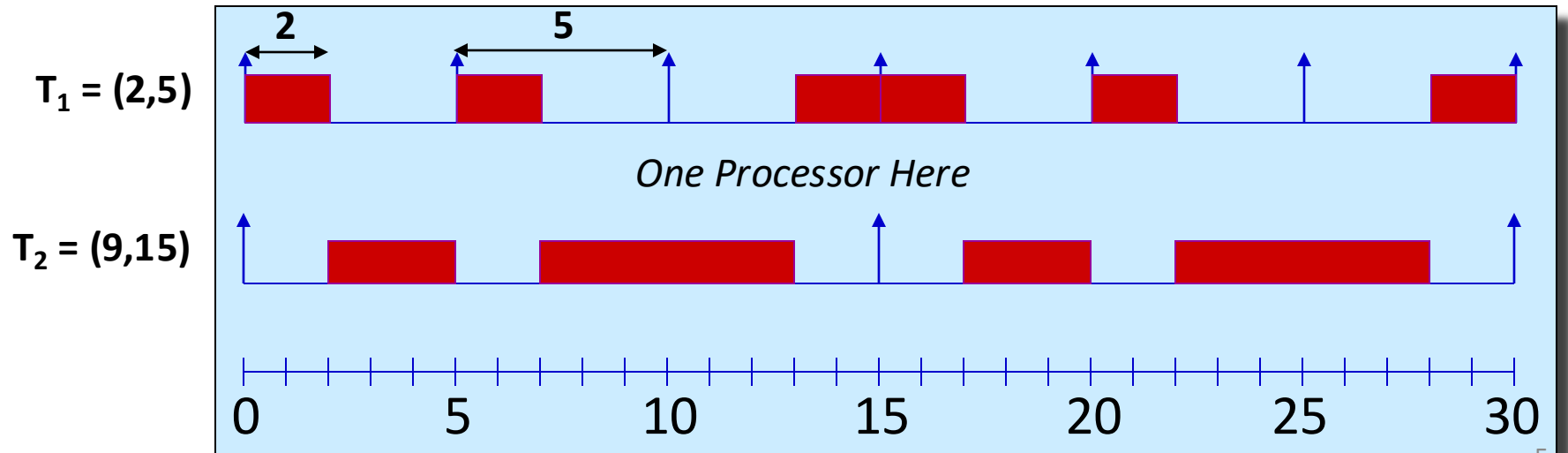
What is a Real-Time System?

- A system with a dual notion of correctness:
 - *Logical correctness* (“it does the right thing”);
 - *Temporal correctness* (“it does it on time”).
- A system wherein *predictability* is as important as *performance*.
- Real-time systems are designed based on worst case, rather than average case
- **A simple example:** A robot arm picking up objects from a conveyor belt.



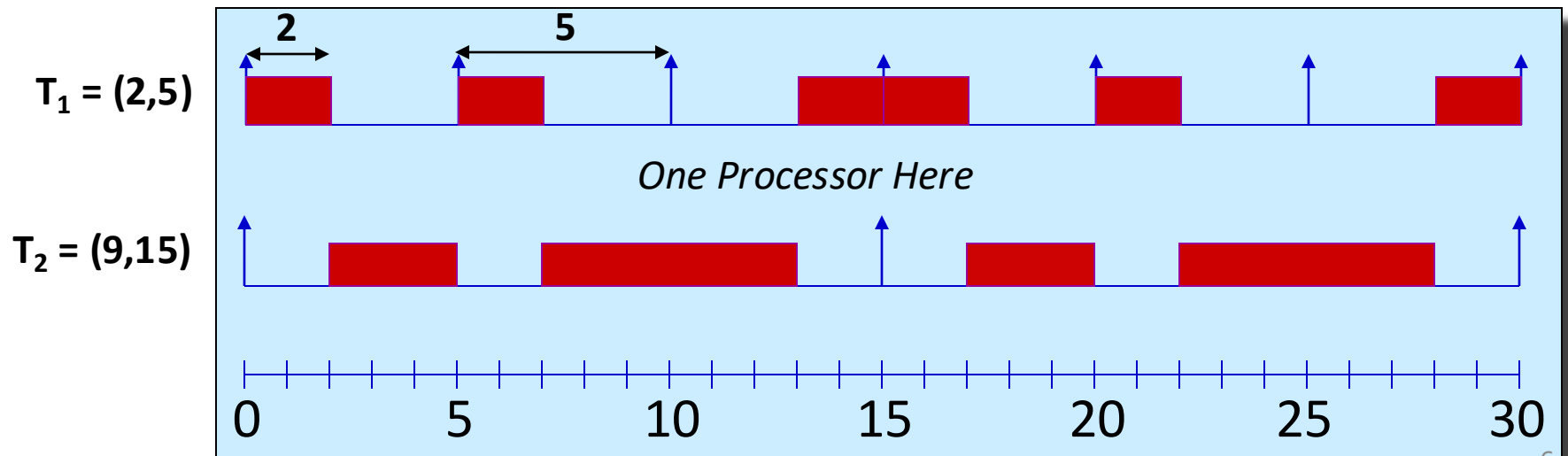
Periodic Task Systems

- Set τ of periodic tasks scheduled on M cores:



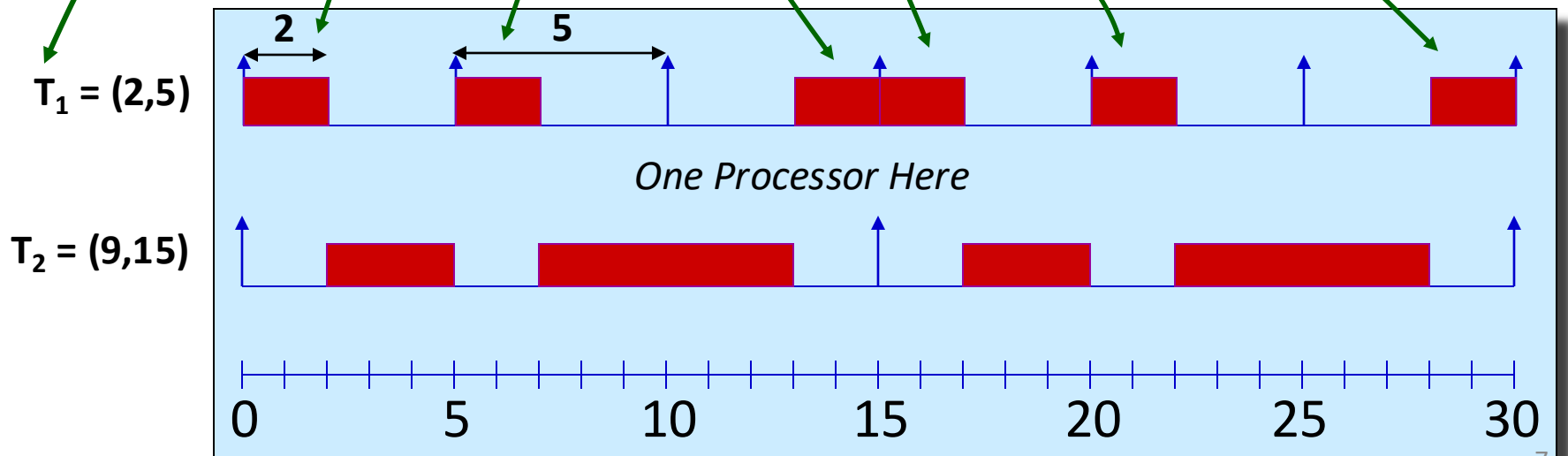
Periodic Task Systems

- Set τ of **periodic tasks** scheduled on **M cores**:
 - Task $T_i = (e_i, p_i)$ releases a **job** with exec. cost e_i every p_i time units. p_i is known as *period*. It is a time interval for processing an input
 - T_i 's **utilization** (or **weight**) is $u_i = e_i/p_i$.
 - Total utilization** is $U(\tau) = \sum_{T_i} e_i/p_i$.



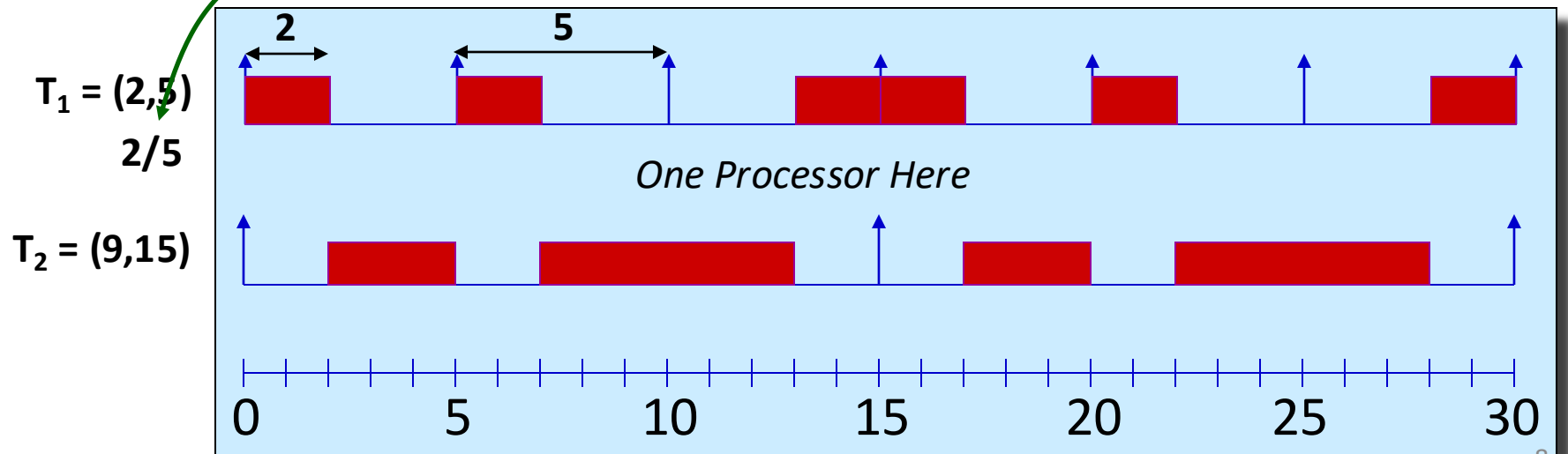
Periodic Task Systems

- Set τ of **periodic tasks** scheduled on **M cores**:
 - Task $T_i = (e_i, p_i)$ releases a **job** with exec. cost e_i every p_i time units. p_i is known as *period*. It is a time interval for processing an input
 - T_i 's **utilization** (or **weight**) is $u_i = e_i/p_i$.
 - Total utilization** is $U(\tau) = \sum_{T_i} e_i/p_i$.



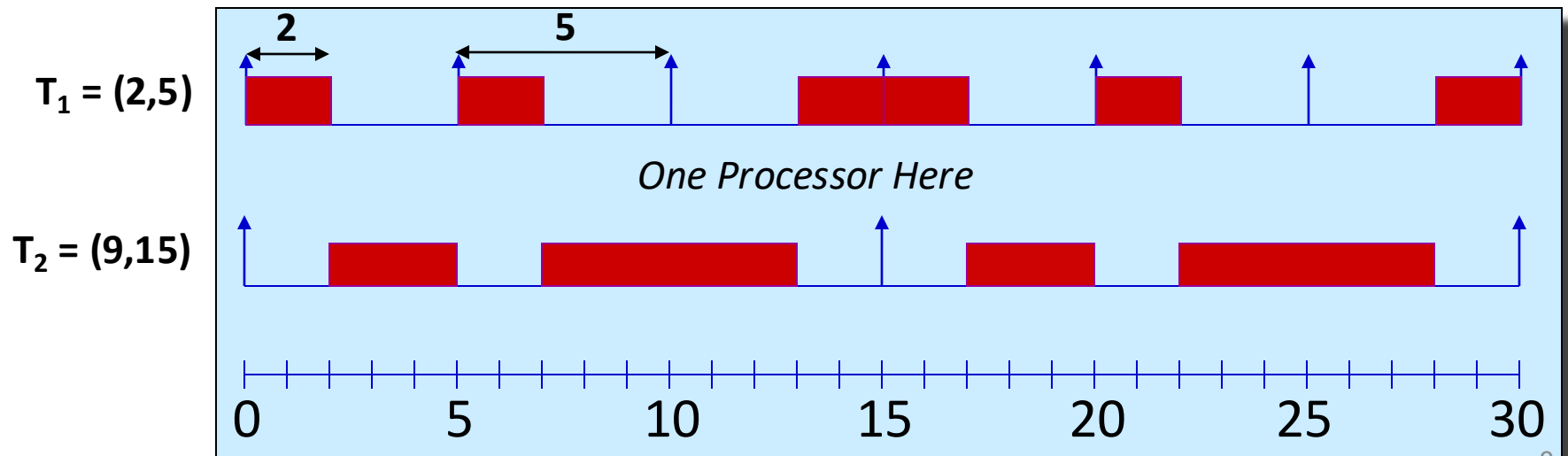
Periodic Task Systems

- Set τ of **periodic tasks** scheduled on **M cores**:
 - Task $T_i = (e_i, p_i)$ releases a **job** with exec. cost e_i every p_i time units. p_i is known as *period*. It is a time interval for processing an input
 - T_i 's **utilization** (or **weight**) is $u_i = e_i/p_i$.
 - Total utilization** is $U(\tau) = \sum_{T_i} e_i/p_i$.



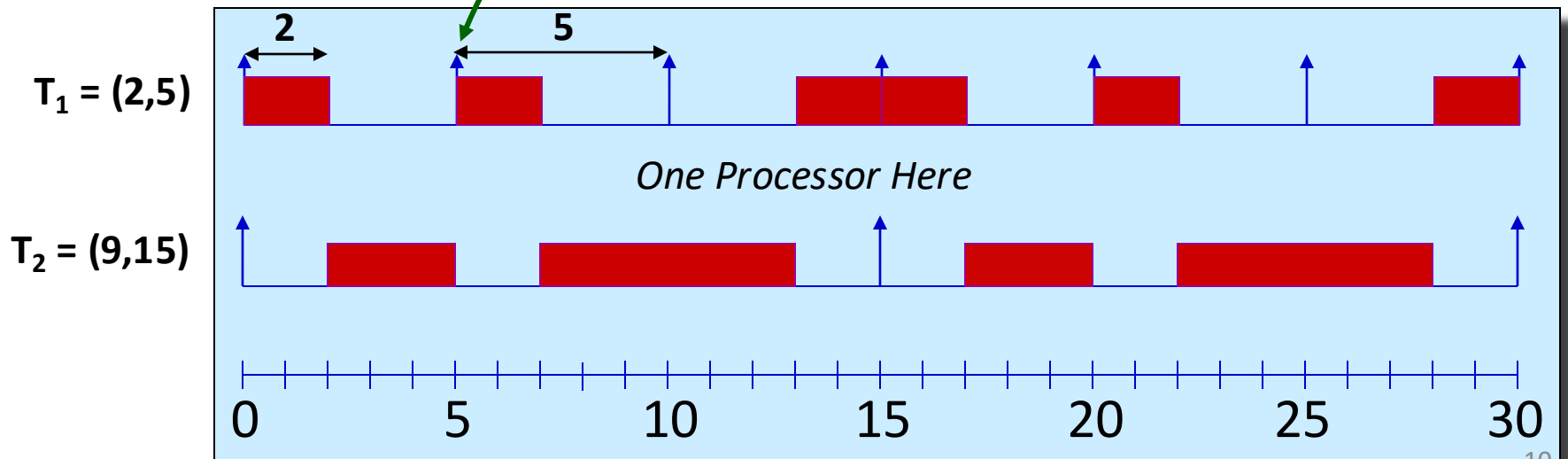
Periodic Task Systems

- Set τ of **periodic tasks** scheduled on **M cores**:
 - Task $T_i = (e_i, p_i)$ releases a **job** with exec. cost e_i every p_i time units. p_i is known as *period*. It is a time interval for processing an input
 - T_i 's **utilization** (or **weight**) is $u_i = e_i/p_i$.
 - Total utilization** is $U(\tau) = \sum_{T_i} e_i/p_i$.
 - Each job of T_i has a **deadline** at the next job release of T_i .



Periodic Task Systems

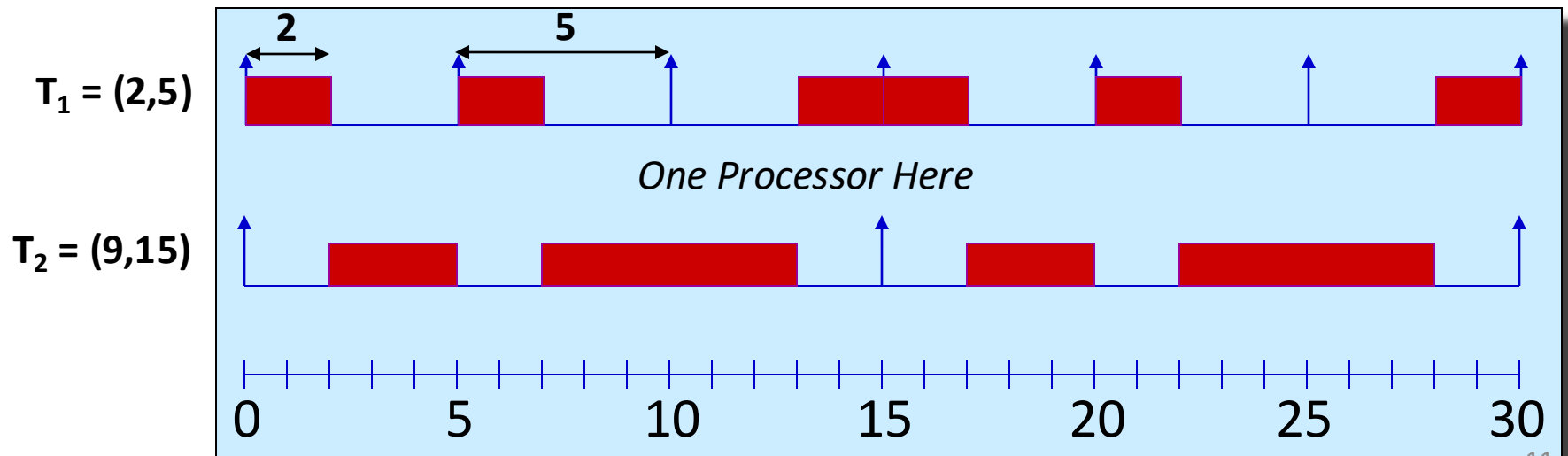
- Set τ of **periodic tasks** scheduled on **M cores**:
 - Task $T_i = (e_i, p_i)$ releases a **job** with exec. cost e_i every p_i time units. p_i is known as *period*. It is a time interval for processing an input
 - T_i 's **utilization** (or **weight**) is $u_i = e_i/p_i$.
 - Total utilization** is $U(\tau) = \sum_{T_i} e_i/p_i$.
 - Each job of T_i has a **deadline** at the next job release of T_i . The deadline is usually the period



Periodic Task Systems

- Set τ of **periodic tasks** scheduled on **M cores**:
 - Task $T_i = (e_i, p_i)$ releases a **job** with exec. cost e_i every p_i time units. p_i is known as *period*. It is a time interval for processing an input
 - T_i 's **utilization** (or **weight**) is $u_i = e_i/p_i$.
 - Total utilization** is $U(\tau) = \sum_{T_i} e_i/p_i$.

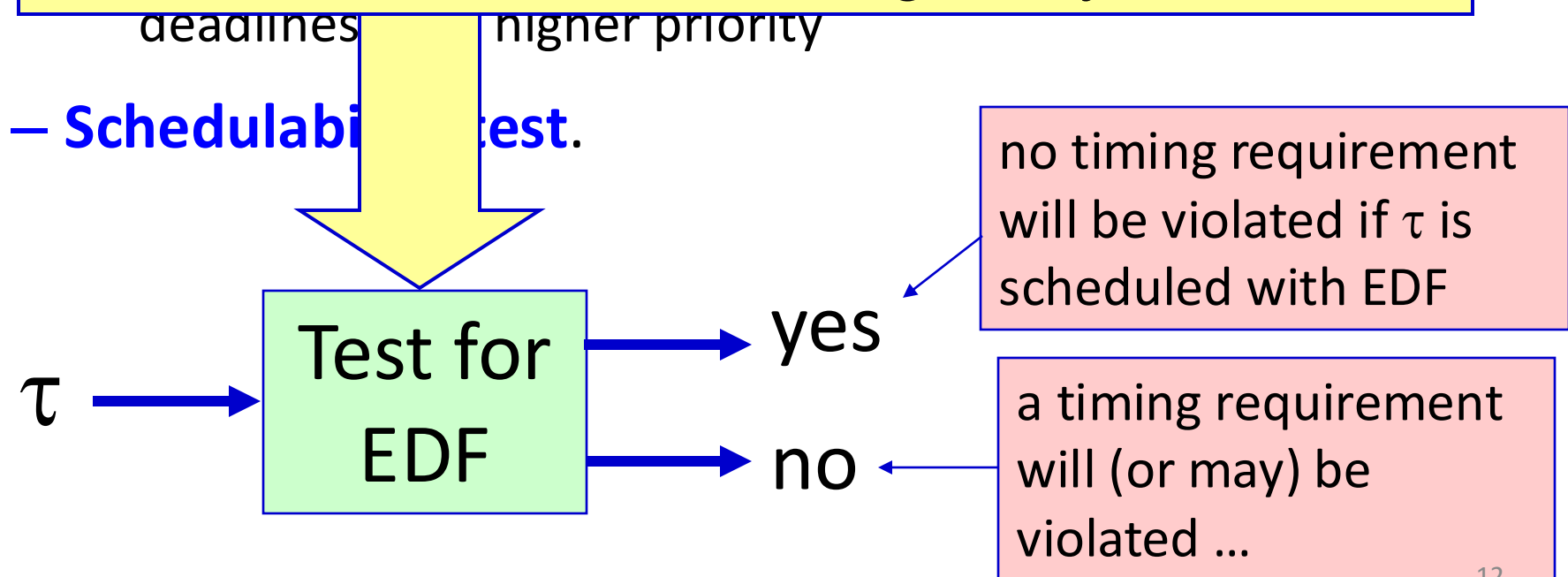
This is an example of an **earliest-deadline-first (EDF)** schedule.



Scheduling vs. Schedulability

- W.r.t. scheduling, we actually care about two kinds of algorithms:

Utilization loss occurs when a test requires utilizations to be restricted to get a “yes” answer.



Optimality and Feasibility

- A schedule is **feasible** if all timing constraints are met
 - For a feasible schedule, $U(\tau) \leq 1$. $U(\tau) = 1$ means CPU is utilized 100%
- A task set τ is **schedulable** using scheduling algorithm A if A produces a feasible schedule for τ
- A scheduling algorithm is **optimal** if it provides a feasible schedule for a schedulable task set
 - EDF is an optimal algorithm

Static-Priority Scheduling

- Under **fixed-priority** scheduling, different jobs of a task are assigned the same priority.
- We will assume that tasks are indexed in decreasing priority order, i.e., **T_i has higher priority than T_k if $i < k$.**
- The ready task with the highest priority is always scheduled.

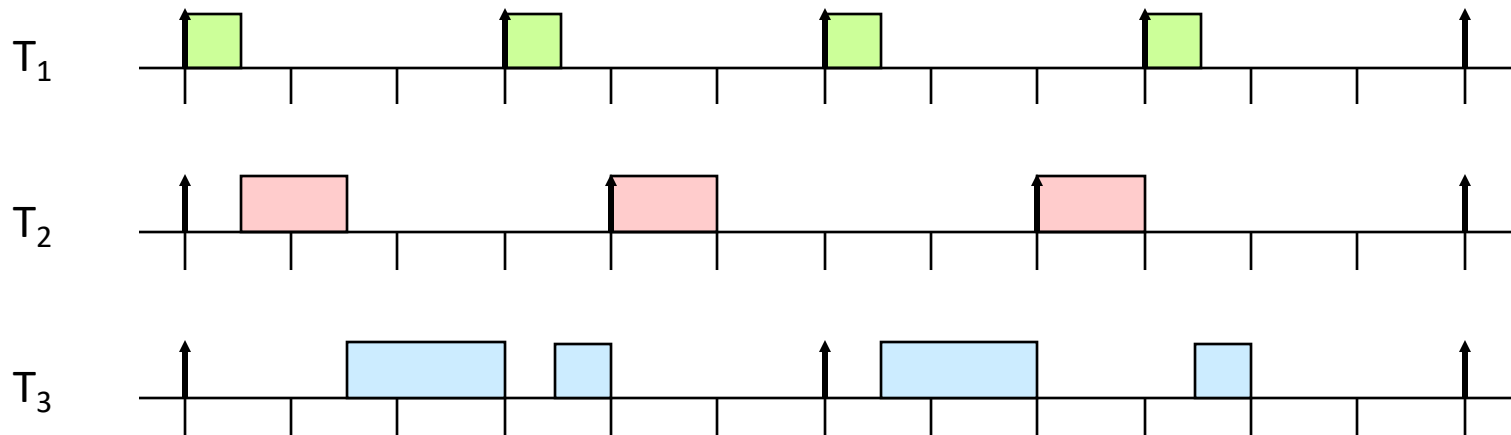
Rate-Monotonic Scheduling

(Liu and Layland)

Priority Definition: Tasks with smaller periods have higher priority.

Example Schedule: Three tasks, $T_1 = (0.5, 3)$, $T_2 = (1, 4)$, $T_3 = (2, 6)$.

Rate-monotonic scheduling finds a feasible schedule when $U(\tau) \leq 0.7$



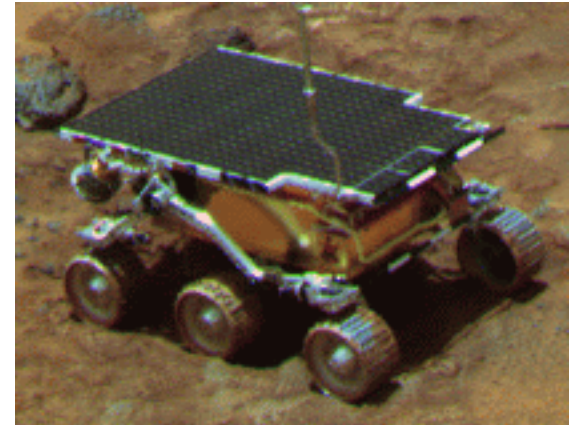
RT Synchronization 101

Priority Inversions

So far we've assumed all jobs are independent.

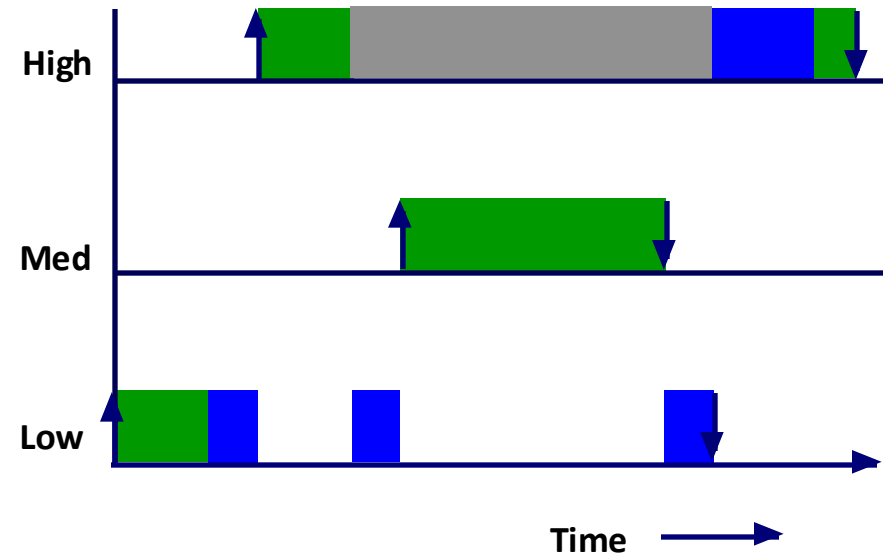
A *priority inversion* occurs when a high-priority job is blocked by a low-priority one.

This is bad because HP jobs usually have more stringent timing constraints.



Mars Pathfinder famously had a priority inversion when deployed and it almost caused a mission failure. A patch was sent remotely to fix.

<https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft>



Critical Section



Priority Inversion



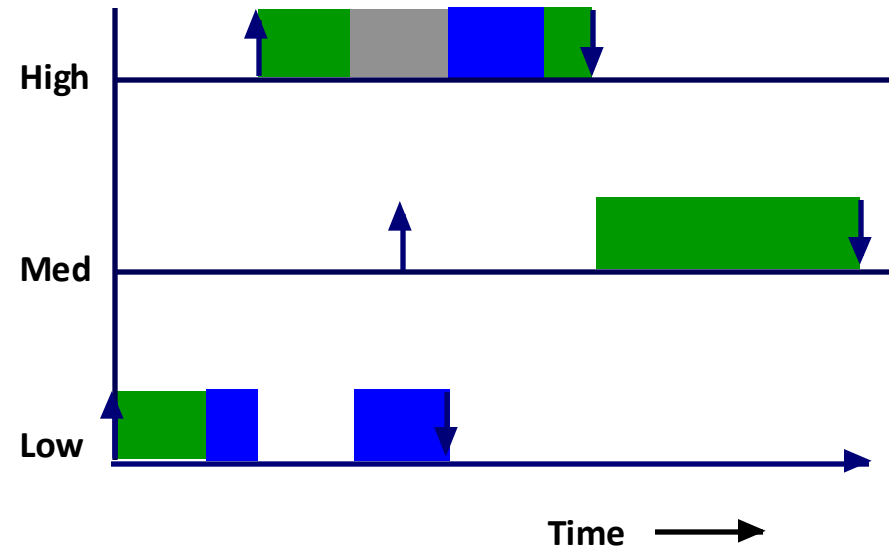
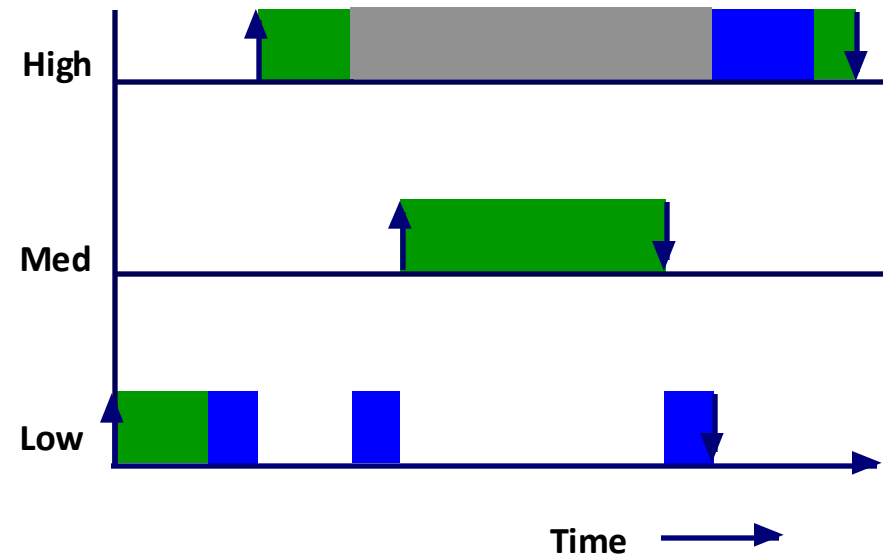
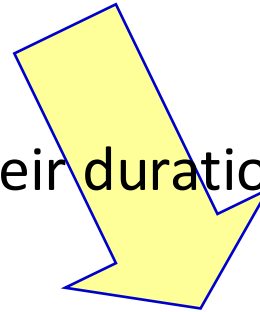
Computation Outside of CS's

RT Synchronization 101

Priority Inheritance

A Common Solution: Use *priority inheritance* (blocking job executes at blocked job's priority).

Doesn't prevent inversions but limits their duration.



Critical Section



Priority Inversion



Computation Outside of CS's

Scheduler Classes

- Linux has different algorithms for scheduling different types of processes
 - Called scheduler classes
- Each class implements a different but “pluggable” algorithm for scheduling
 - Within a class, you can set the policy
- RT class: `SCHED_FIFO`, `SCHED_RR`
- Non-RT class: `SCHED_OTHER` (CFS)

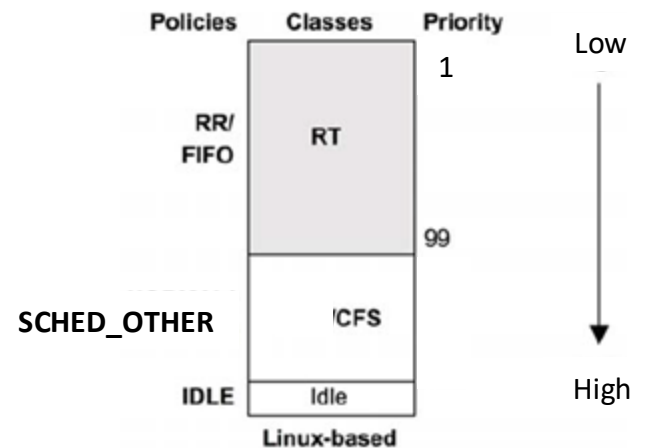


Figure from, “*Systems Performance: Enterprise and the Cloud*” by Brendan Gregg

Exercise

Consider two processes, T1 and T2, where $p_1 = 50$, $e_1 = 25$, $p_2 = 75$, $e_2 = 30$.

Illustrate the scheduling of these two processes using

- earliest-deadline-first (EDF) scheduling
- rate-monotonic scheduling

Exercise

EDF

0	25	55	80	110	135	150
T1		T1		T1		
	T2		T2			

Rate monotonic
T1 priority > T2 priority
T2 misses deadline

0	25	50	75
T1		T1	
	T2		

Exercise

Let A, B, and C be three tasks. A has the highest priority and C has the lowest priority. A and C use a shared resource protected by a mutex.

Suppose a scheduler makes decisions about scheduling tasks based on their priorities. That is, the scheduler runs, among tasks, the one with the highest priority. Such a task runs to completion or blocks for any reason.

- What are the completion times of A, B, C before applying the priority inheritance?
- What are the completion times of A, B, C after applying the priority inheritance?

Task	Start time	Before lock	Critical section	After lock
A	2	1	2	1
B	3.8	4.2		
C	0	1	3	1

Exercise

Priority inversion

	0	1	2	3	3.8	8	9.2	11.2	12.2	13.2
A			BL				L	AL		
B					BL					
C	BL	L		L		L			AL	

Priority inheritance

	0	1	2	3	5	7	8	12.2	13.2
A			BL		L	AL			
B							BL		
C	BL	L		L				AL	

BL: Before Lock

L: Lock (Critical section)

AL: After Lock

Summary

- Real-time systems differ from general-purpose ones in that there exist timing requirements
- Common in cyber-physical and safety-critical systems, such as avionics, automotive, and other embedded devices.
- Timing requirements inform how scheduling should be handled
- Many classes of real-time scheduling algorithms
- Analysis complements the scheduling implementation to prove temporal correctness