CS3281 / CS5281

Real-Time Scheduling

Will Hedgecock
Sandeep Neema
Bryan Ward

**Some lecture slides borrowed and adapted from the UNC Real-Time Systems Group*

Motivation: Cyber-Physical Systems

Surgical Robotics

Industrial Internet of Things (IIoT)

Power and Utilities

Satellites

Autonomous Vehicles

Drones & DoD Systems

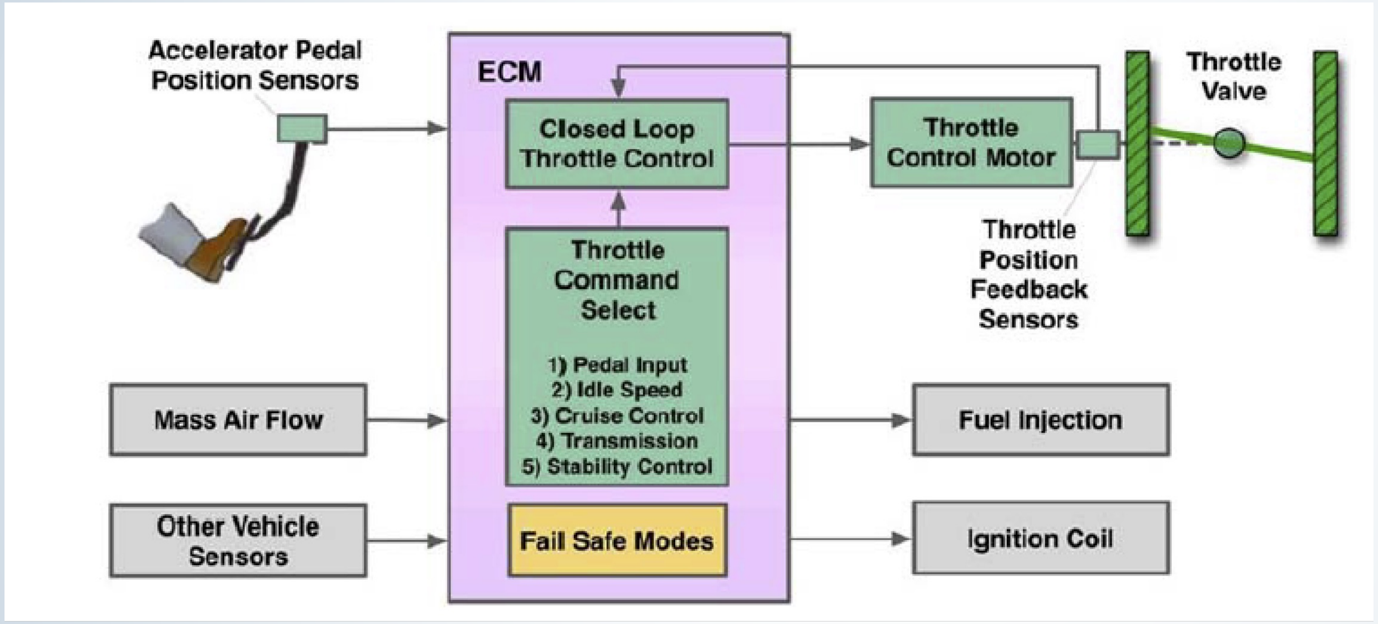
Cyber Physical Systems - key characteristics

- Computing is deeply embedded and integral to safe operation of the physical system

- A large class of CPS are *safety-critical systems*
  - def: failure or malfunction may result in injury, death or severe damage to equipment/environment

- In CPS, *software-managed interactions* need to conform to laws of physics
  - that includes timing

CPS software – an example



https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

- Electronic Throttle Control System
- Controls air + fuel + spark --> engine power
- At the heart of *Toyota's Unintended Acceleration* disaster – 89 deaths & 57 injuries

Real-Time Systems

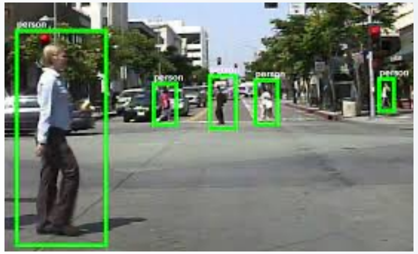
Enterprise Systems

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

Degree of Timing Requirements

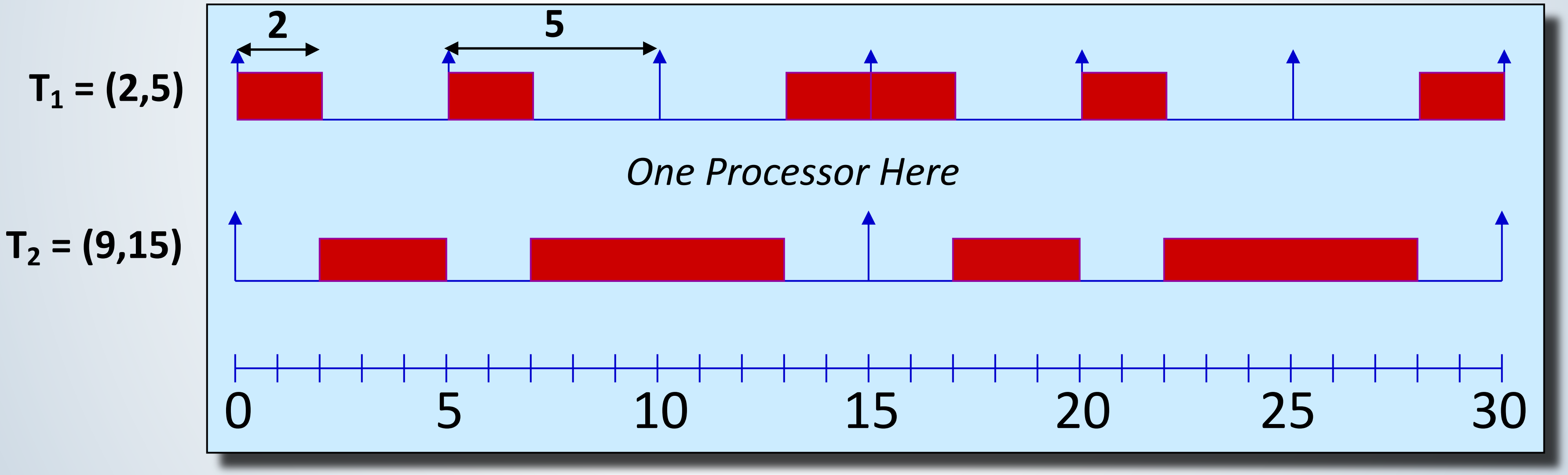
Interaction with the physical world requires keeping time with the real world.
Many CPS, especially safety- and mission-critical systems have strict timing requirements.

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

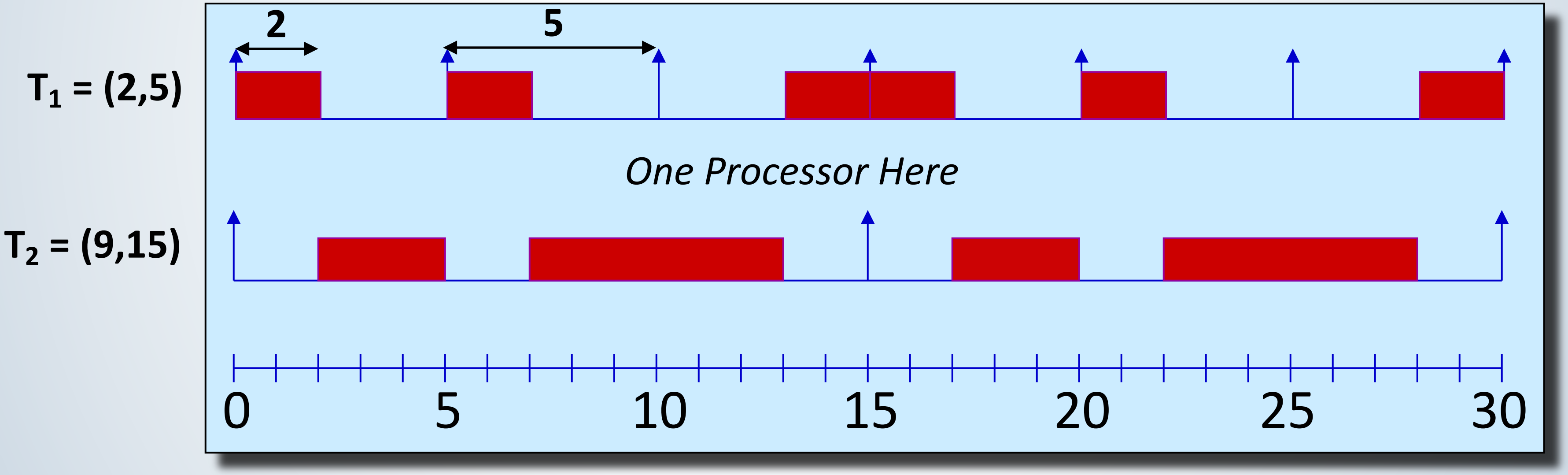
Periodic Task Systems

- Set τ of periodic tasks scheduled on M cores:



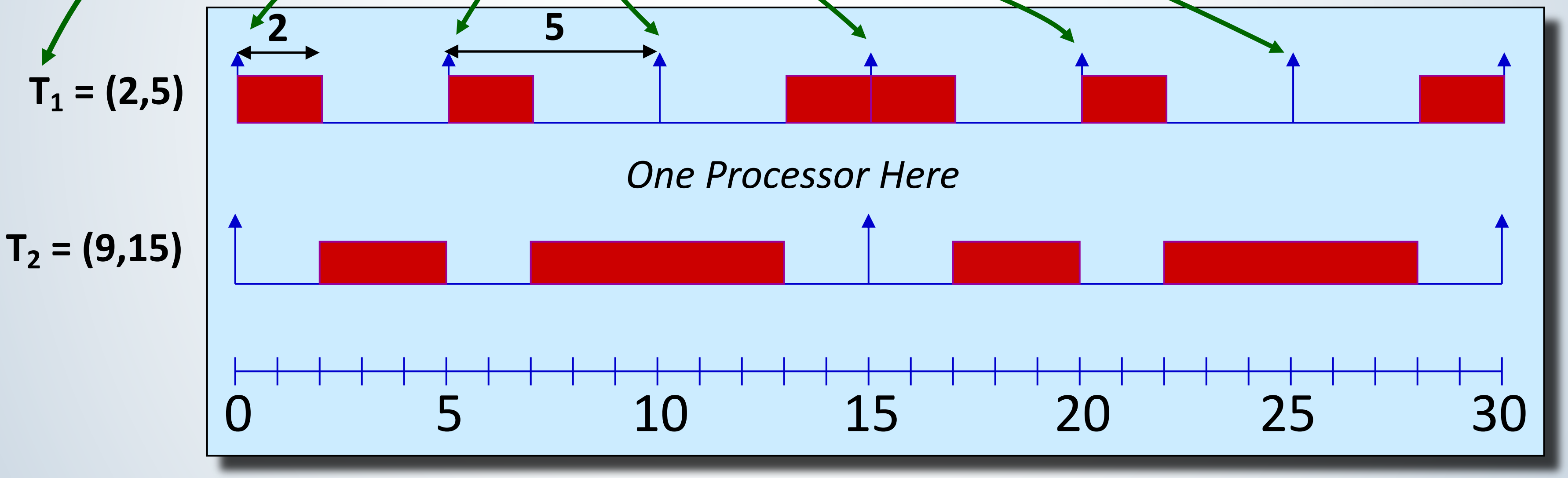
Periodic Task Systems

- 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.
    - T_i's *utilization* (or *weight*) is $u_i = e_i/p_i$.
    - *Total utilization* is $U(\tau) = \sum_{T_i} e_i/p_i$.



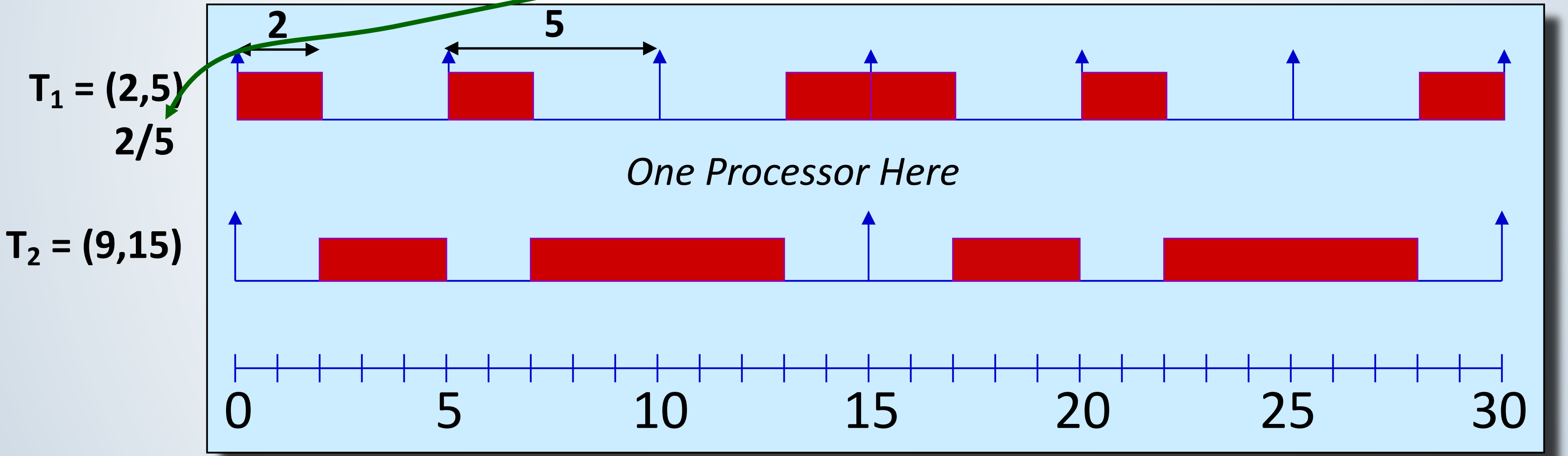
Periodic Task Systems

- 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.
    - 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.
    - T_i's *utilization* (or *weight*) is $u_i = e_i/p_i$.
    - *Total utilization* is $U(\tau) = \sum_{T_i} e_i/p_i$.

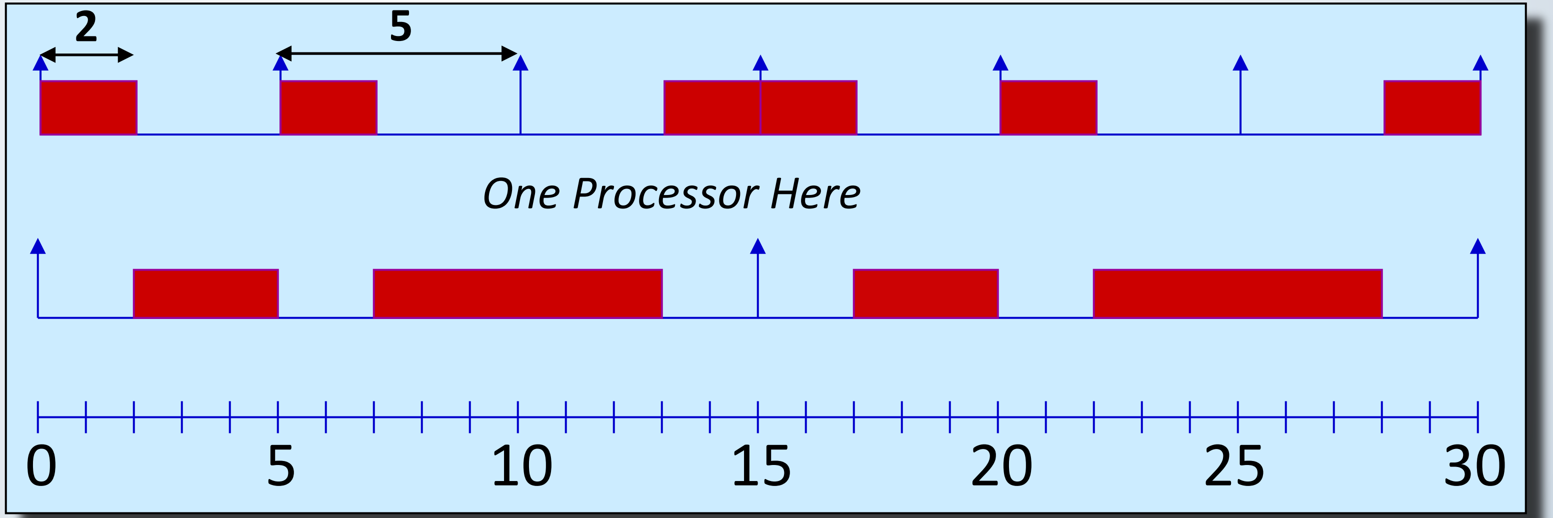


Periodic Task Systems

- 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.
    - 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.

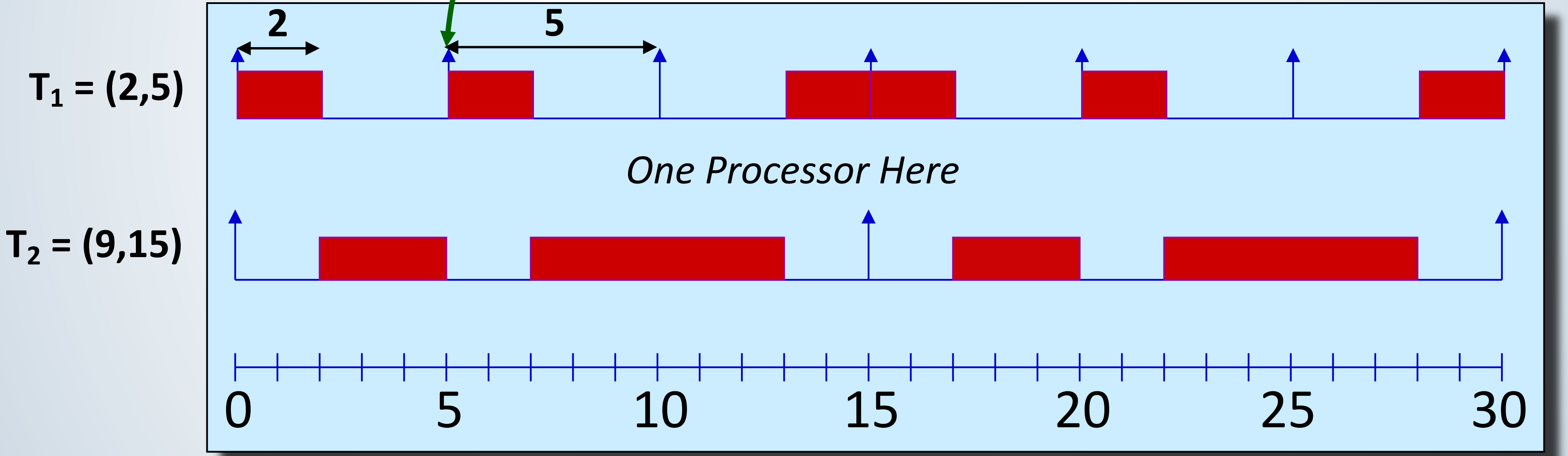
$T_1 = (2,5)$

$T_2 = (9,15)$



Periodic Task Systems

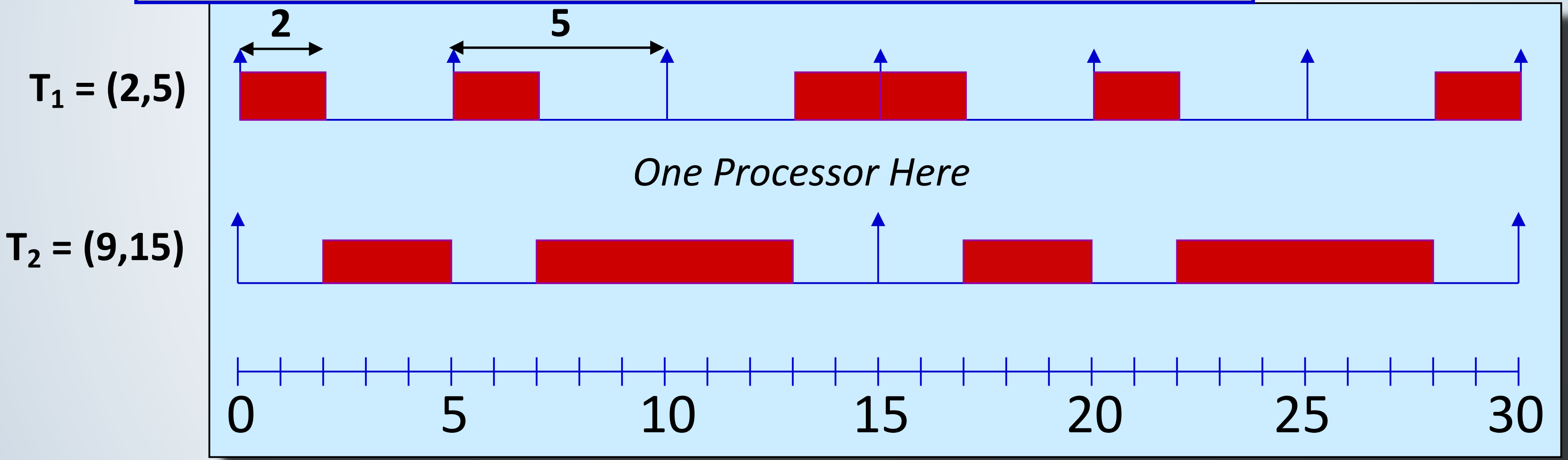
- Set τ of periodic tasks scheduled on M cores:
  - Task $T_i = (e_i, p_i)$ releases a *job* with exec. cost e_i every p_i time units.
    - T_i's *utilization* (or *weight*) is $u_i = e_i/p_i$.
    - *Total utilization* is $U(\tau) = \sum_{T_i} e_i/p_i$.
  - Each job of T_i has a *deadline* at the next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

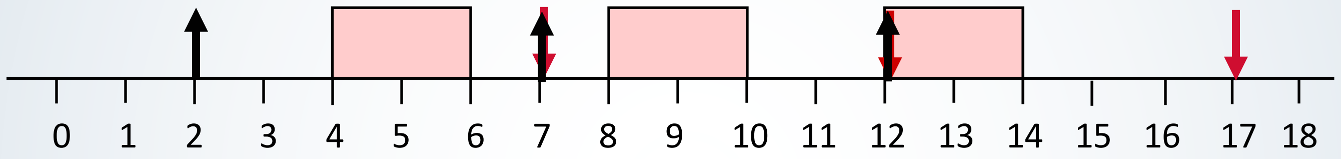


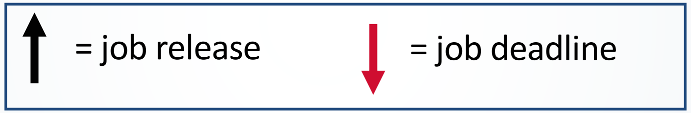
Other Kinds of Tasks

- **Sporadic:** p_i is a minimum separation between job releases of T_i.
- For periodic or sporadic, relative deadlines d_i can be
  - **implicit:** $d_i = p_i$ (assumed unless stated otherwise);
  - **constrained:** $d_i \leq p_i$;
  - **arbitrary**.
- Also can have **aperiodic** (one-shot) jobs.
  - **hard aperiodic**: job has a deadline

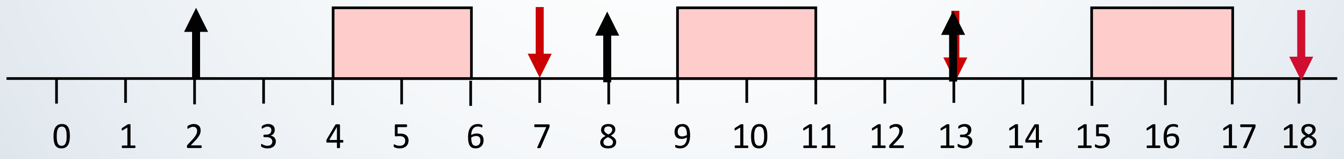
Periodic vs. Sporadic

An implicit-deadline **periodic** task T_i with $p_i = 5$ and $e_i = 2$ could execute like this:





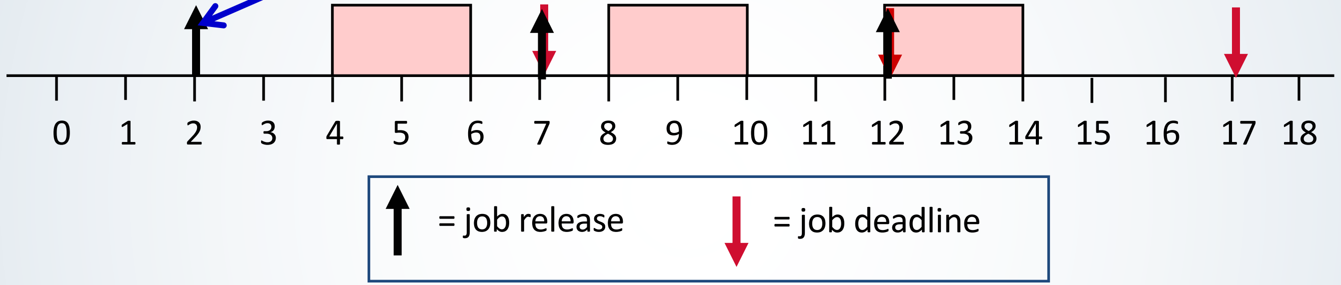
If **sporadic**, could execute like this:



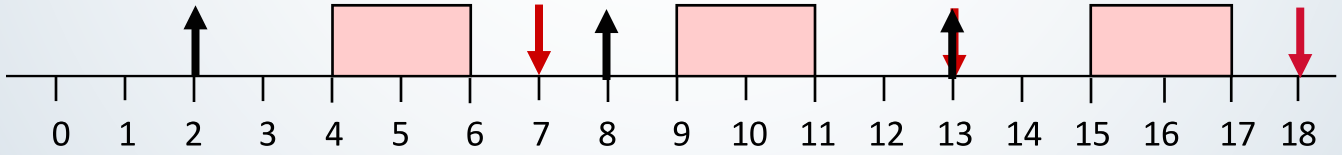
Periodic vs. Sporadic

An implicit-deadline could execute like this:

A task system is called synchronous if all tasks start at time 0, and asynchronous otherwise.

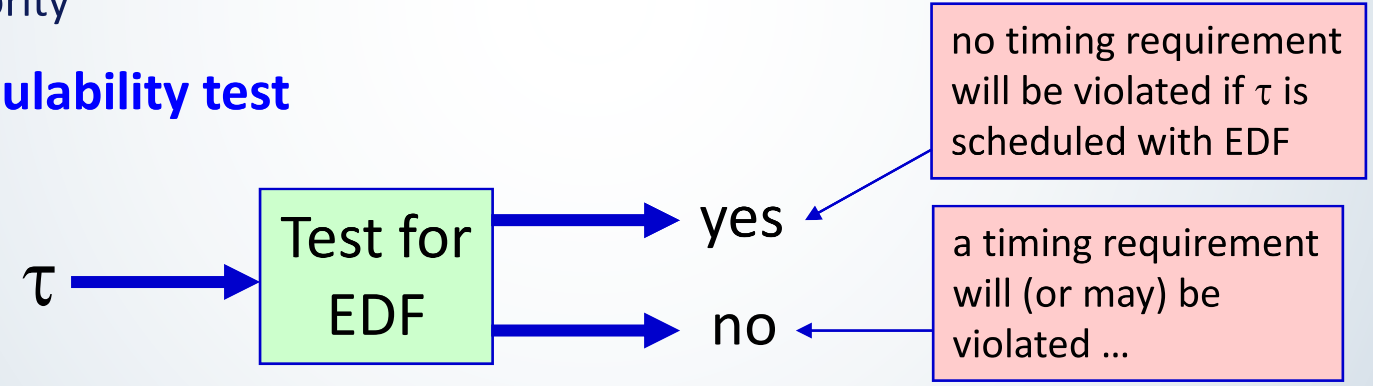


If **sporadic**, could execute like this:



Scheduling vs. Schedulability

- W.r.t. scheduling, we actually care about *two* kinds of algorithms:
  - **Scheduling algorithm** (of course)
    - **Example:** Earliest-deadline-first (EDF): Jobs with earlier deadlines have higher priority
  - **Schedulability test**



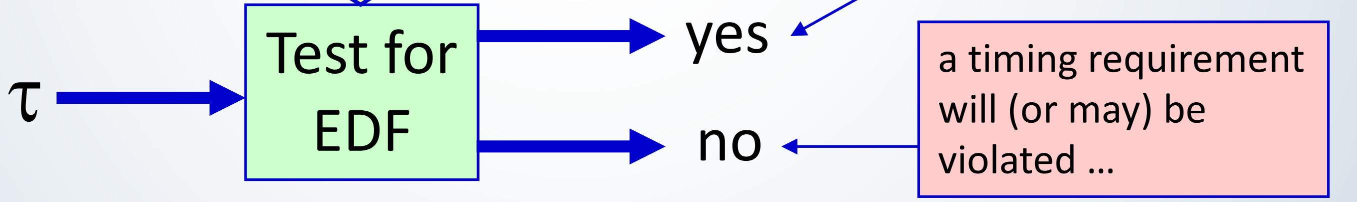
τ
Test for EDF
yes
no
no timing requirement will be violated if τ is scheduled with EDF
a timing requirement will (or may) be violated …

Scheduling vs. Schedulability

- W.r.t
  algo

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

  – **Scheduling algorith** course).
    - **Example:** Earliest-d -first (EDF): Jobs with earlier deadlines have higher priority
  – **Schedulability test**



no timing requirement will be violated if τ is scheduled with EDF

a timing requirement will (or may) be violated …

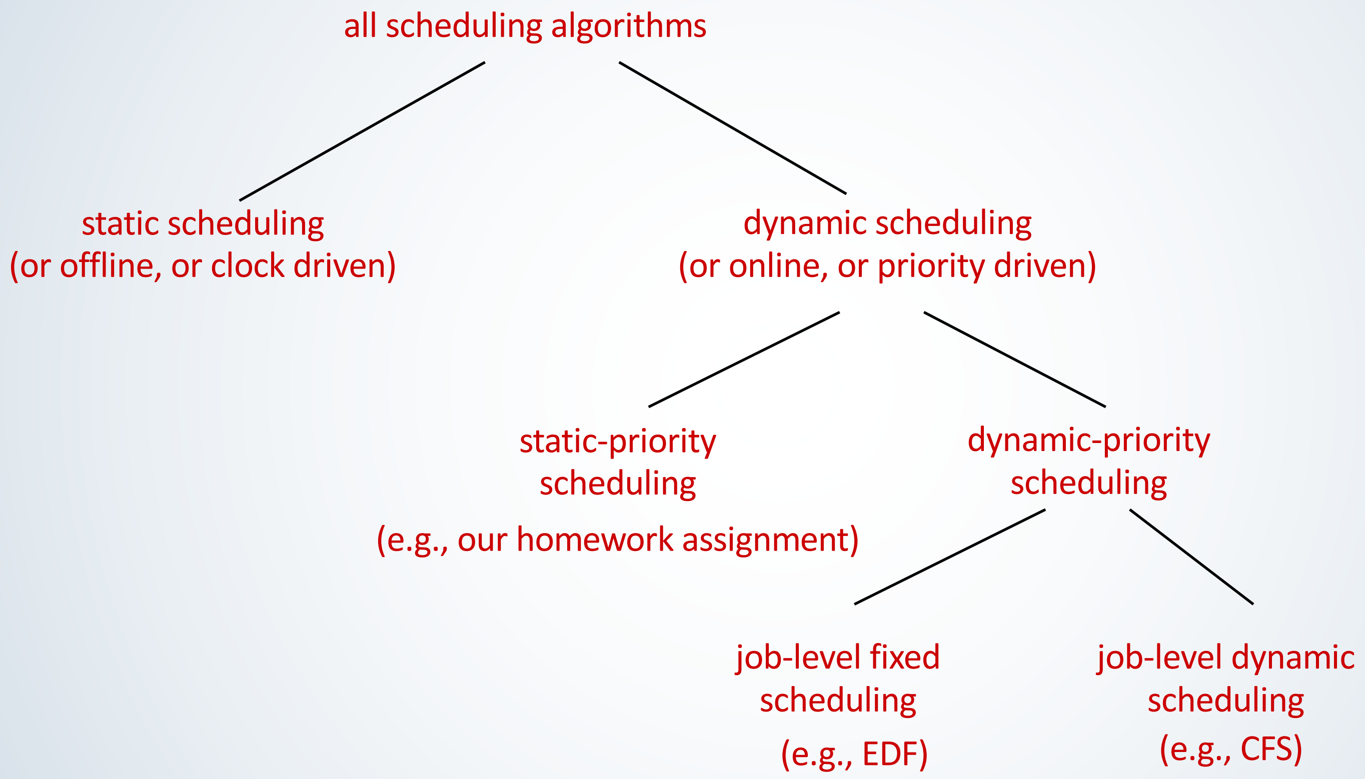
Optimality and Feasibility

- A schedule is **feasible** if all timing constraints are met
- A task set T is **schedulable** using scheduling algorithm A if A always produces a feasible schedule for T
- A scheduling algorithm is **optimal** if it always produces a feasible schedule when one exists (under any scheduling algorithm)

Feasibility vs. Schedulability

- To most people in the real-time systems community:
  - a **feasibility test** indicates whether some algorithm from a class of algorithms can correctly schedule a given task set;
  - a **schedulability test** indicates whether a specific algorithm (e.g., EDF) can correctly schedule a given task set.
- Such tests can either be **exact** or only **sufficient**

Classification of Scheduling Algorithms


all scheduling algorithms
static scheduling
(or offline, or clock driven)
dynamic scheduling
(or online, or priority driven)
static-priority scheduling
(e.g., our homework assignment)
dynamic-priority scheduling
job-level fixed scheduling
(e.g., EDF)
job-level dynamic scheduling
(e.g., CFS)

Static-Priority Scheduling

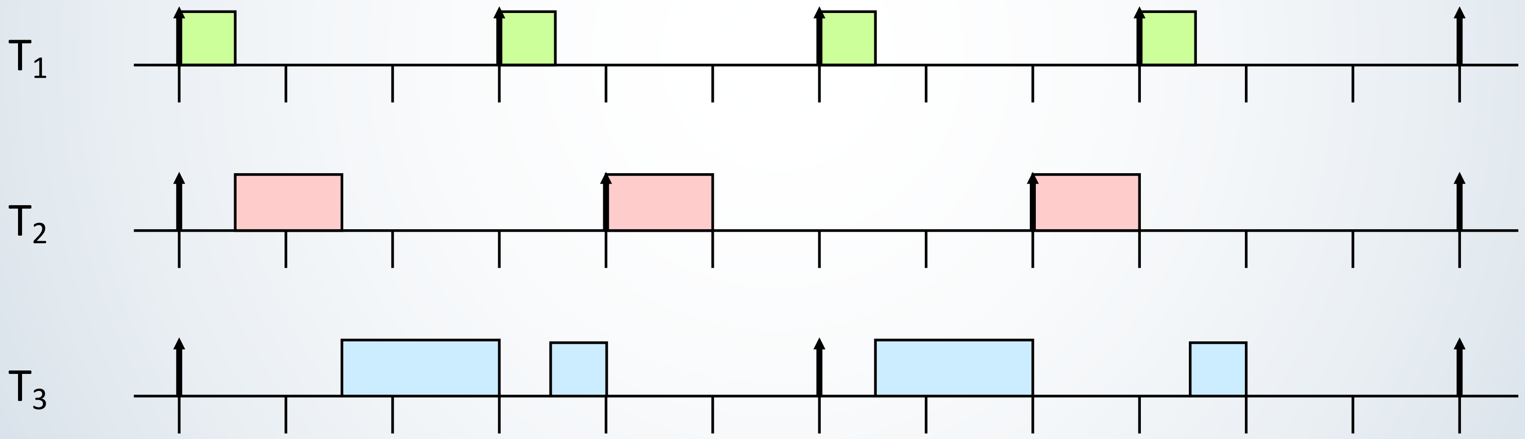
- Under fixed-priority scheduling, different jobs of a task are assigned the same priority.
- We will assume that tasks are indexed in decreasing priority order, i.e., T_i has higher priority than T_k if $i < k$.
- The ready task with the highest priority is always scheduled.

In-class examples

- Consider the following example implicit-deadline task system
  - $T_1 = (3,10)$, $T_2 = (1,3)$, $T_3 = (1,5)$
- Assume the task system is synchronous (i.e., all start at t=0)
- At what time does the first job of T_2 complete under static-priority scheduling where T_1 is the highest priority and T_3 is the lowest?
- At what times does T_1 complete under EDF?
  - Draw out the schedule to see

Rate-Monotonic Scheduling

- **Priority Definition:** Tasks with smaller periods have higher priority.
- **Example Schedule:** Three tasks, $T_1 = (0.5, 3)$, $T_2 = (1, 4)$, $T_3 = (2, 6)$.



Optimality (or not) of RM

Theorem: RM is not optimal.

Exception: RM is optimal if task set is harmonic (“simply periodic”).

Proof:

Consider $T_1 = (1, 2)$ and $T_2 = (2.5, 5)$.

Total utilization is one, so the system is feasible.

However, under RM, a deadline will be missed, regardless of how we choose to (statically) prioritize T_1 and T_2.

You can work through both cases to convince yourself of this.

This proof actually shows that *no* static-priority algorithm is optimal.

Utilization-based RM Schedulability Test

Theorem: [Liu and Layland] A system of n independent, preemptable sporadic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the RM algorithm if its total utilization U is at most

$$U_{RM}(n) = n(2^{1/n} - 1)$$

Note that this is only a **sufficient** schedulability test.

Note: Utilization Loss = $1 - U_{RM}(n)$

Static-Priority Schedulability Test

(Uniprocessor, Static-Priority)

Definition: The time-demand function of the task T_i, denoted $w_i(t)$, is defined as follows.

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \cdot e_k \qquad \text{for } 0 < t \leq p_i$$

Note: We assume tasks are indexed by priority.

For any fixed-priority **algorithm A** with $d_i \leq p_i$ for all i …

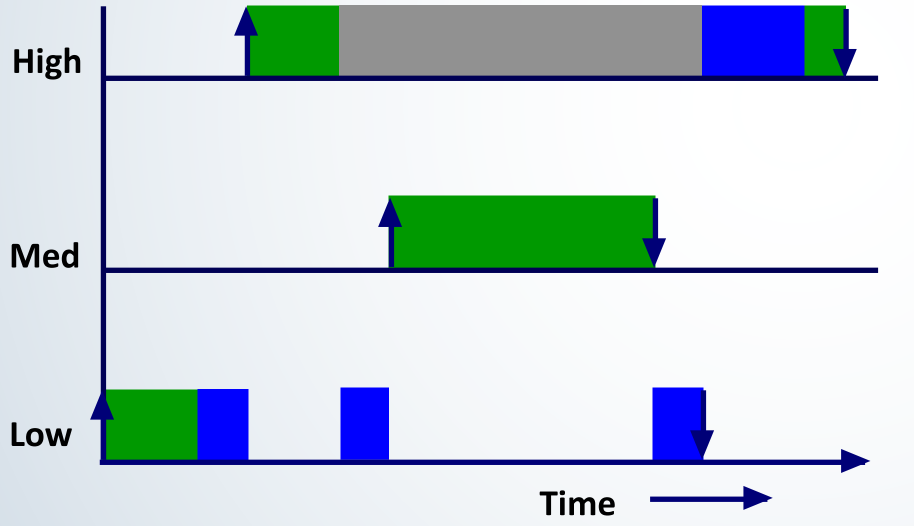
Theorem: A system τ of sporadic, independent, preemptable tasks is schedulable on one processor by algorithm A if

$$(\forall i :: (\exists t : 0 < t \leq d_i :: w_i(t) \leq t))$$

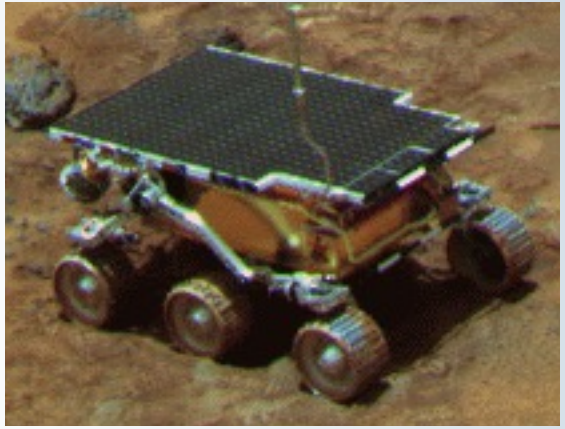
holds.

Real-Time Synchronization: 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



🟦 Critical Section ▨ Priority Inversion 🟩 Computation Outside of CS's



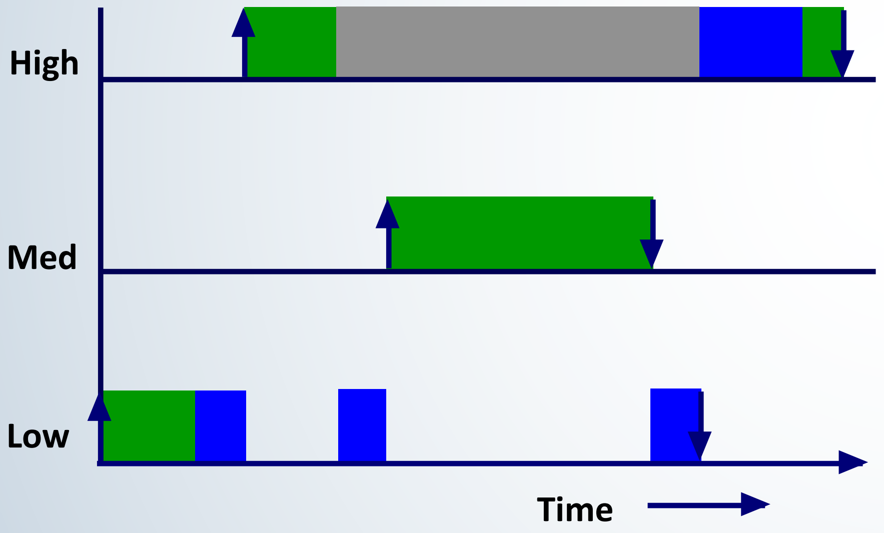
Mars Pathfinder infamously had a priority inversion when deployed and it almost caused a mission failure. A patch was sent remotely to fix it. It changed one bit.

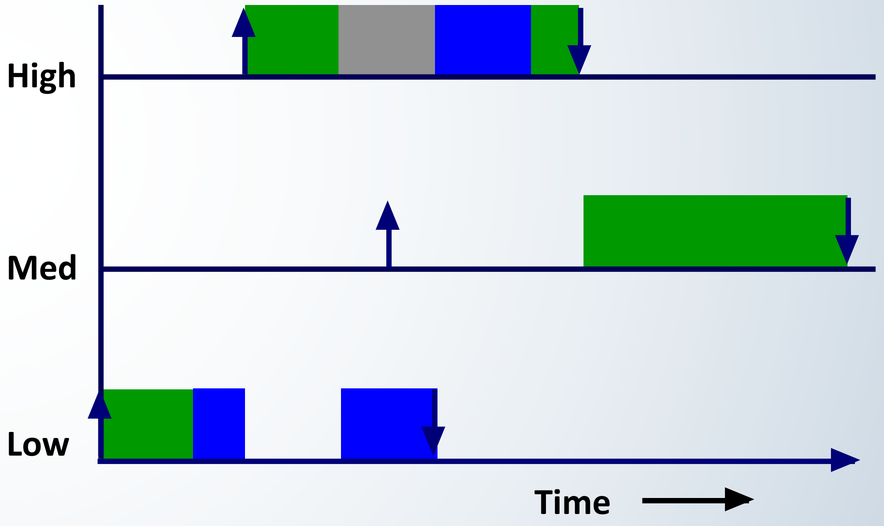
https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft

Real-Time Synchronization: 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

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 exist
- Analysis complements the scheduling implementation to prove temporal correctness