Lecture 08: Advanced Scheduling

CS343 – Operating Systems

Branden Ghena – Fall 2020

Some slides borrowed from:
Wang Yi (Uppsala), and UC Berkeley CS149 and CS162

Today's Goals

- Describe real-time systems
- Understand scheduling policies based on deadlines
- Explore modern operating system schedulers

Outline

- **Real Time Operating Systems**
  - Earliest Deadline First scheduling
  - Rate Monotonic scheduling
- Modern Operating Systems
  - Linux O(1) scheduler
  - Lottery and Stride scheduling
  - Linux Completely Fair Scheduler

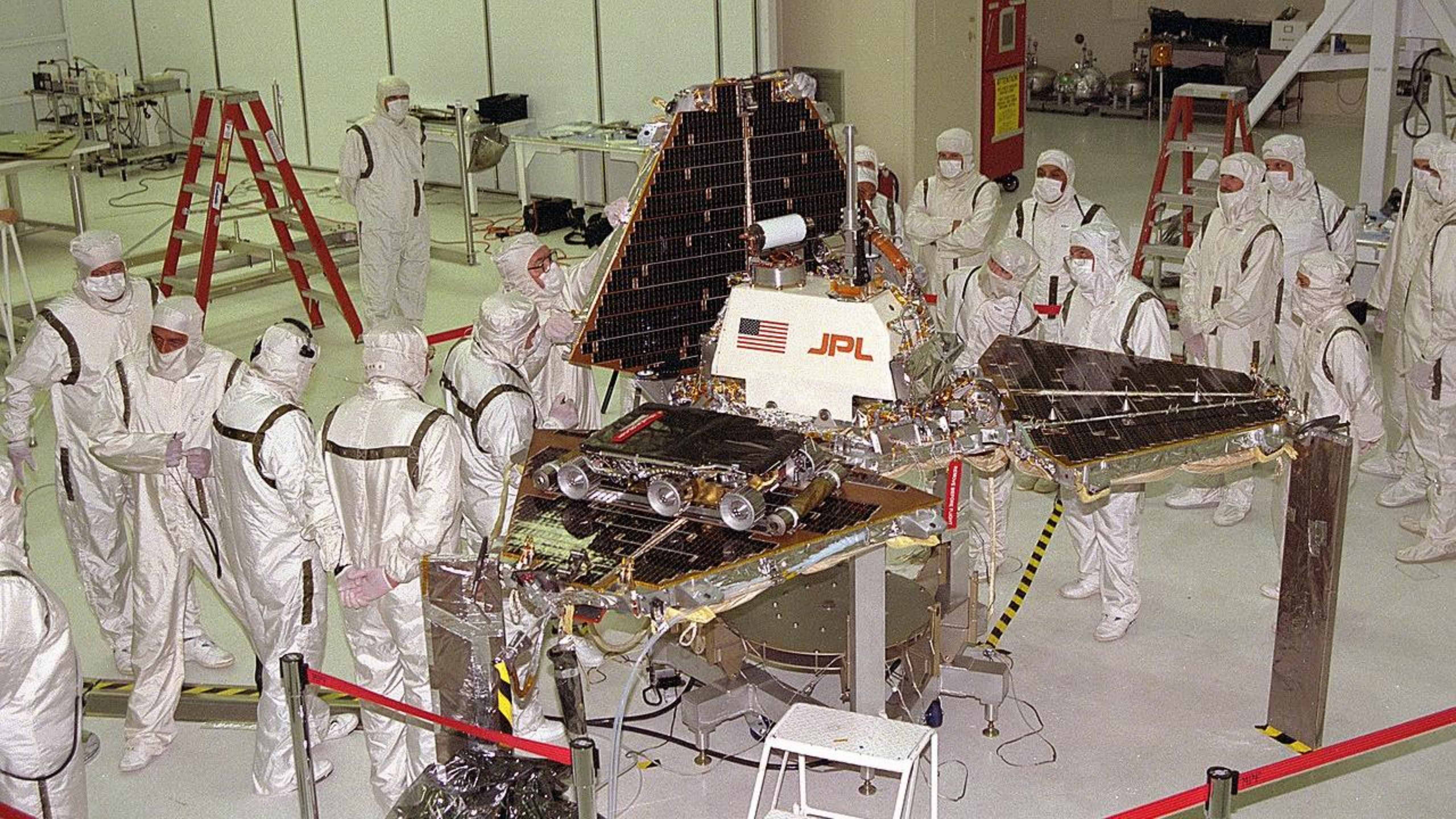
Normal OSes don’t cut it for all use cases

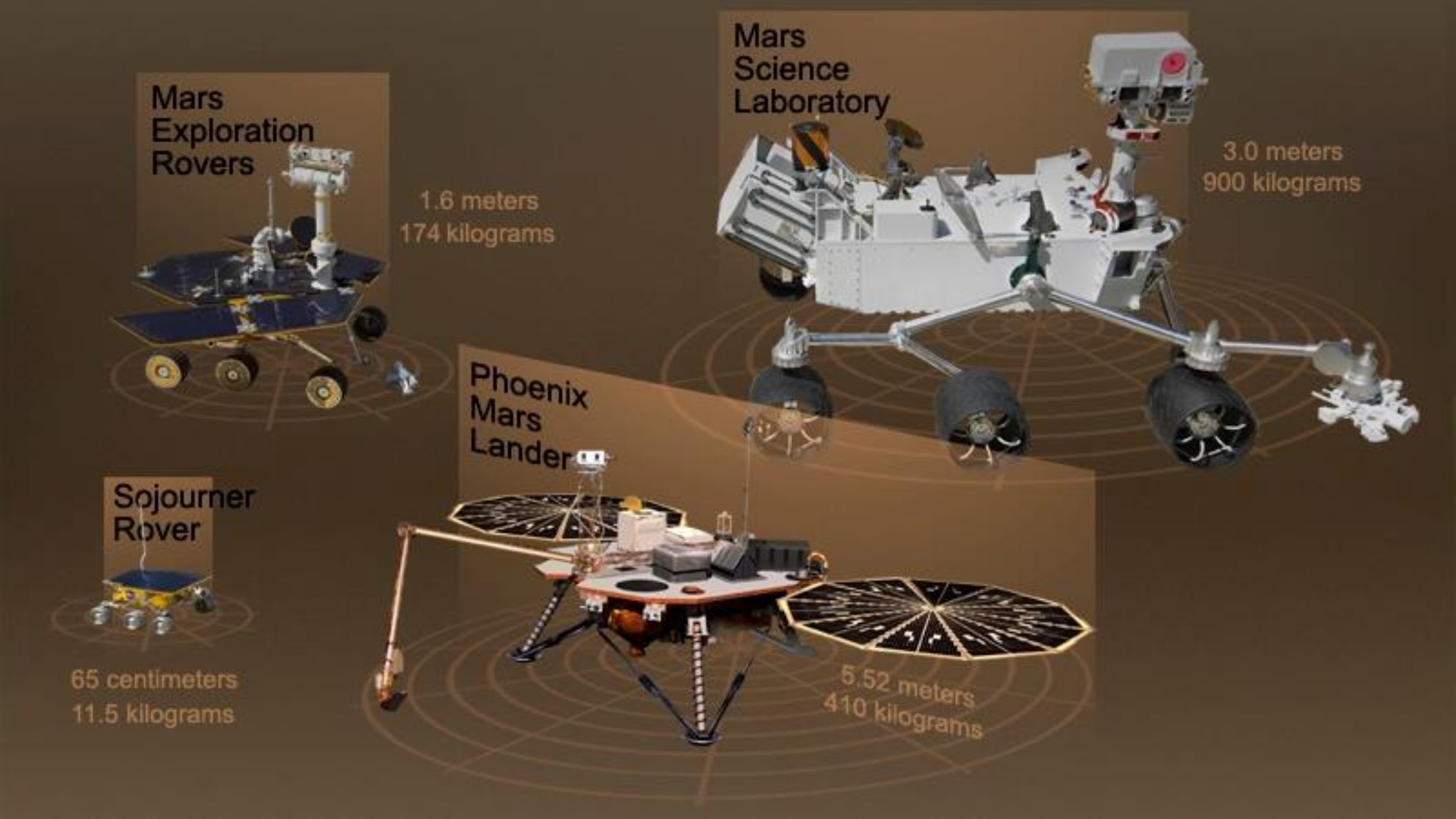
- Some environments need very specialized systems
  - Flight controls
  - Autonomous vehicles
  - Space exploration

- In each of these scenarios
  - Computer failures are unacceptable
  - Humans can’t intervene to resolve issues
  - We’re going to need a computer system with performance *guarantees*

Example: Pathfinder

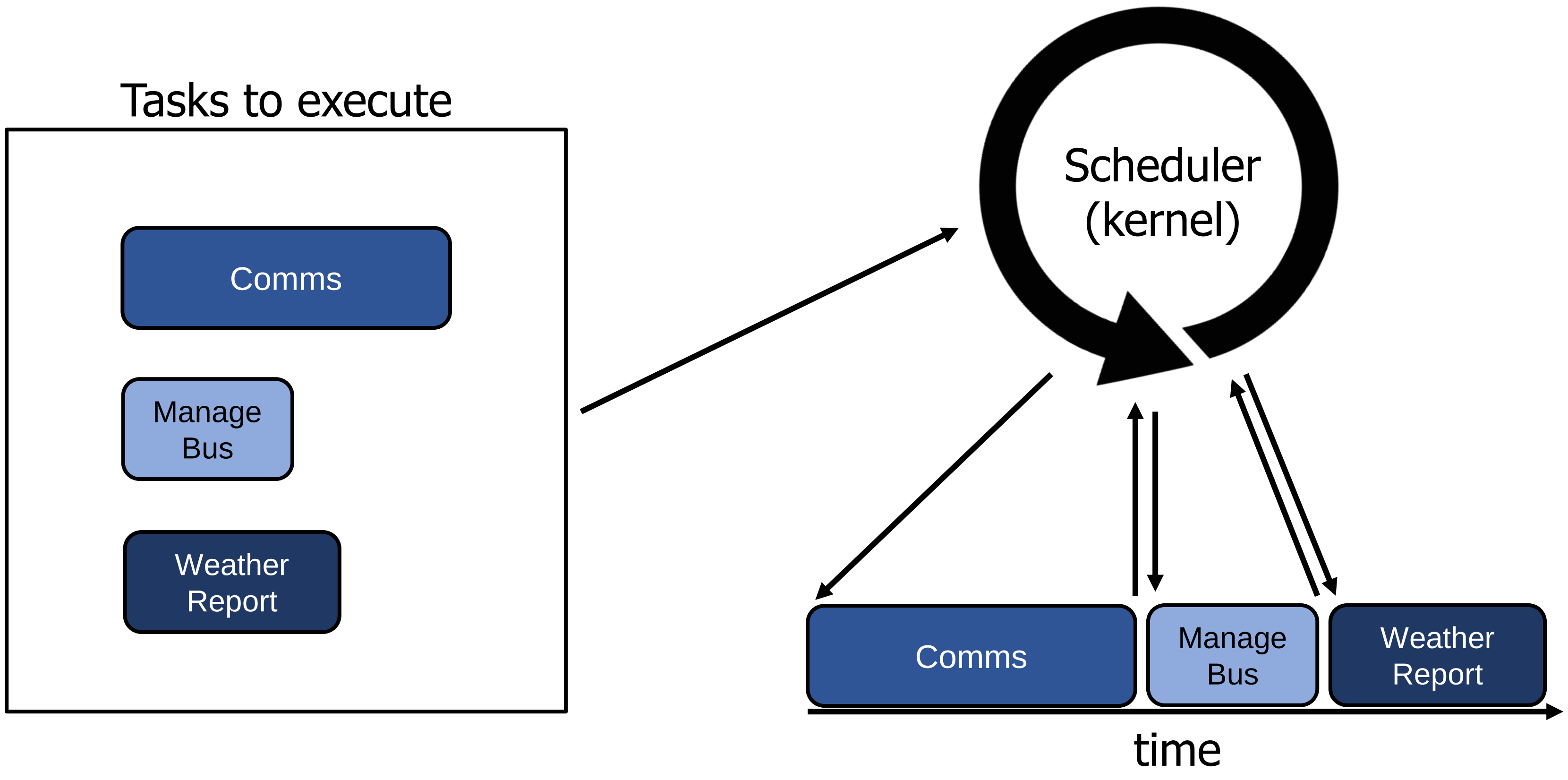
Radiation-hardened IBM CPU

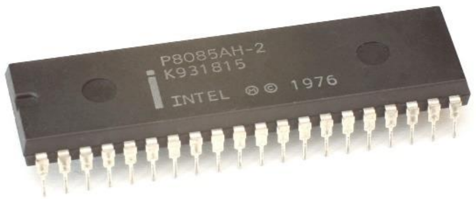

JPL



Mars Exploration Rovers
1.6 meters
174 kilograms
Mars Science Laboratory
3.0 meters
900 kilograms
Phoenix Mars Lander
5.52 meters
410 kilograms
Sojourner Rover
65 centimeters
11.5 kilograms

Pathfinder has periodic tasks that must be executed





Real-Time Operating Systems

- Goal: guaranteed performance
  - Meet *deadlines* even if it means being unfair or slow
  - Limit how bad the *worst case* is
    - Usually mathematically

- It's not about speed, it's about guaranteed performance
  - Good turnaround and response time are nice, but insufficient
  - Predictability is key to providing a guarantee

- RTOS is actually a whole other class worth of material
  - Last taught by Peter Dinda in 2005…

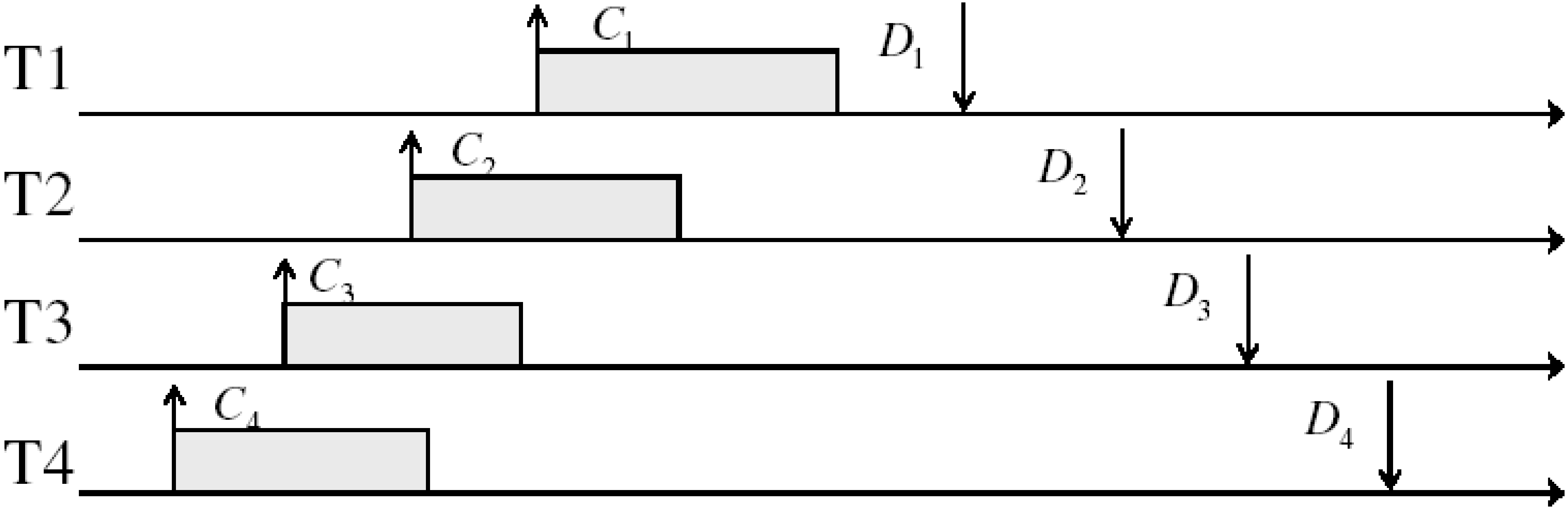
Types of real-time schedulers

- Hard real-time:
  - Meet **all deadlines**
    - Otherwise decline to accept the job
  - Ideally: determine in advance if this is possible

- Soft real-time
  - Attempt to meet deadlines with high probability
  - Often good enough for many non-safety-critical applications
    - Quadcopter software

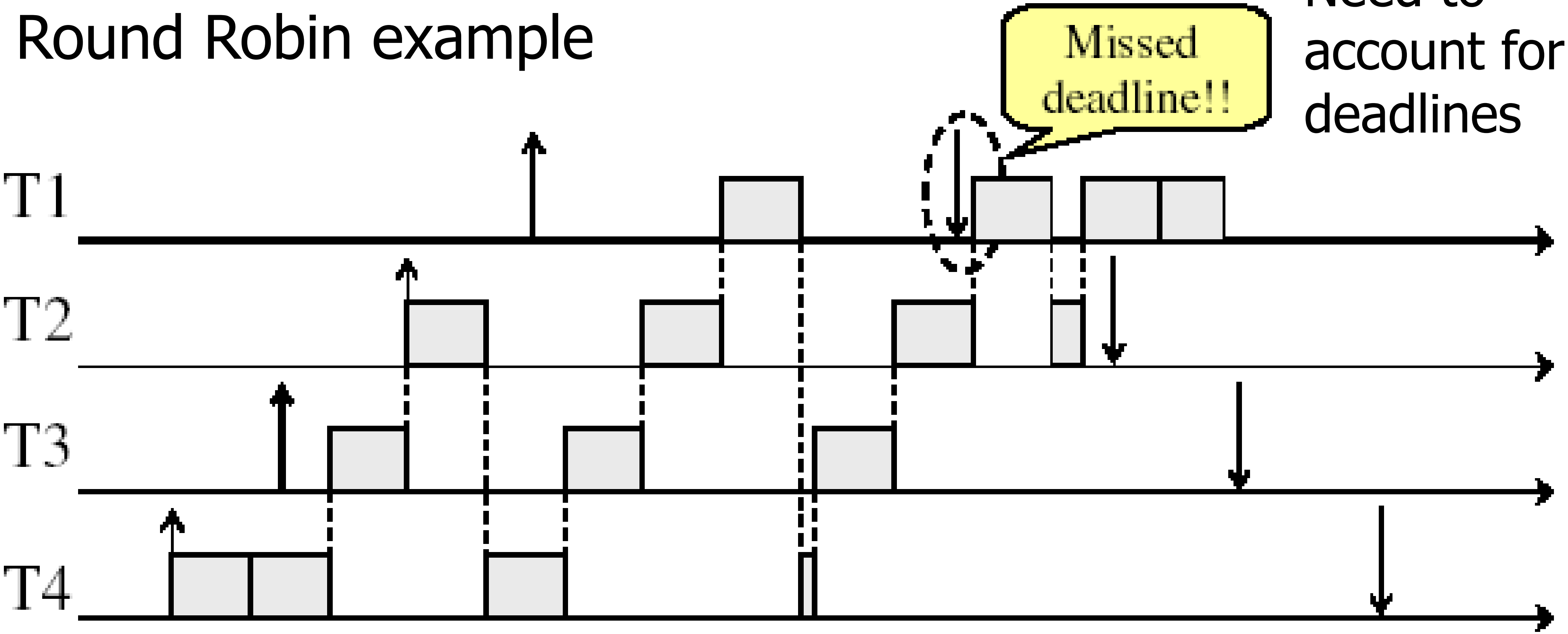
Real-time example

- Preemptable jobs with known deadlines (D) and computation (C)
  - Durations here are worst-case execution times



Prior scheduling policies don't apply here

Round Robin example



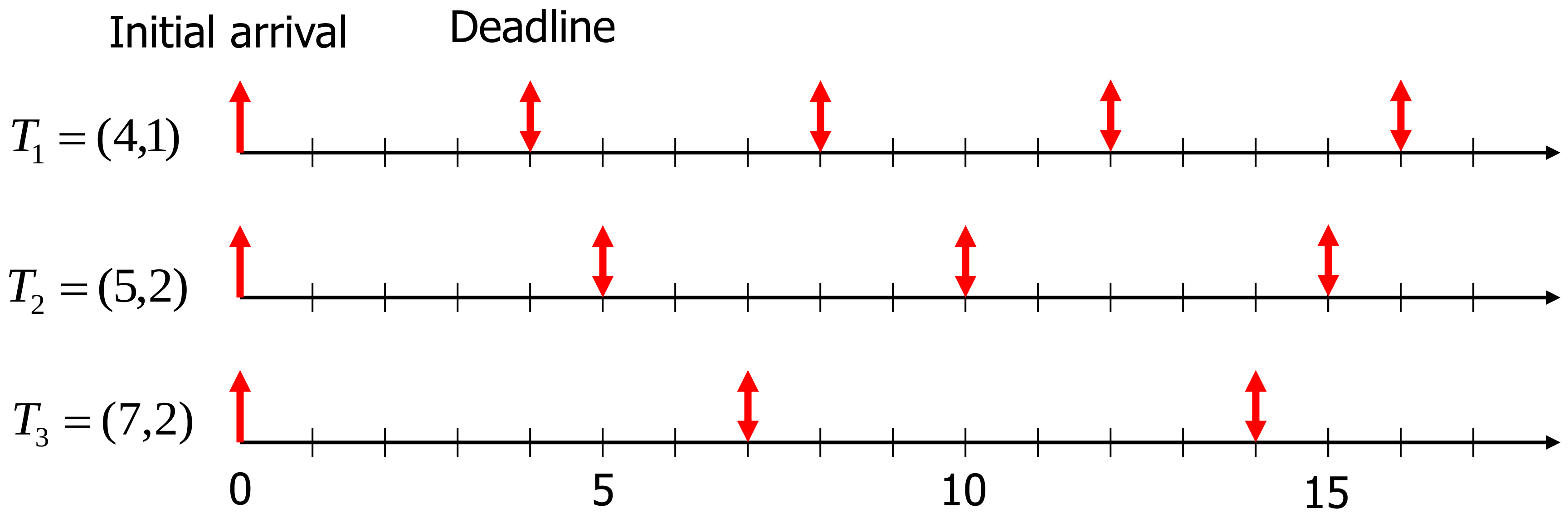
Need to account for deadlines

Types of real-time jobs

- Aperiodic
  - Jobs we are used to
  - Unpredictable start times, no deadlines
- Periodic (we'll focus on these)
  - Recurs at a certain time interval
  - Deadline for completion is before the start of the next time interval
    - i.e. deadline equals the period
  - Can decide *feasibility* of schedule at compile-time
- Sporadic
  - Unpredictable start time, has a deadline
  - Must decide feasibility at runtime and either accept or reject job

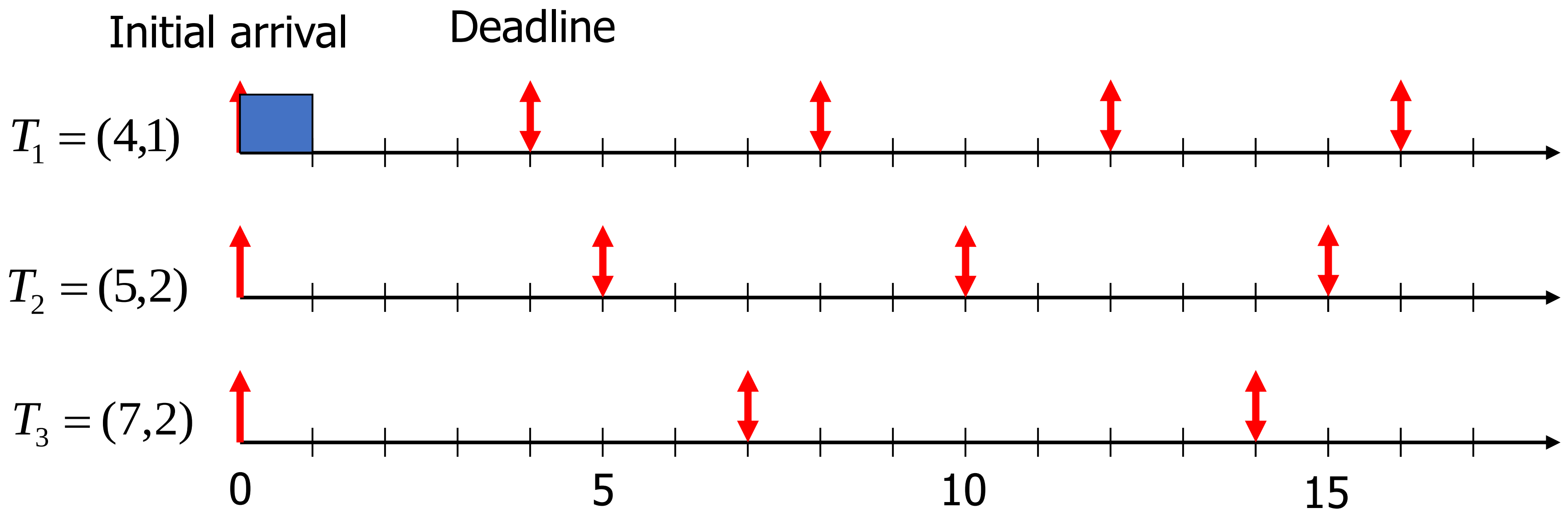
Earliest Deadline First (EDF) Scheduling

- Priority scheduling with pre-emption
- Highest priority given to task with soonest deadline
  - Task = (Period, Duration)



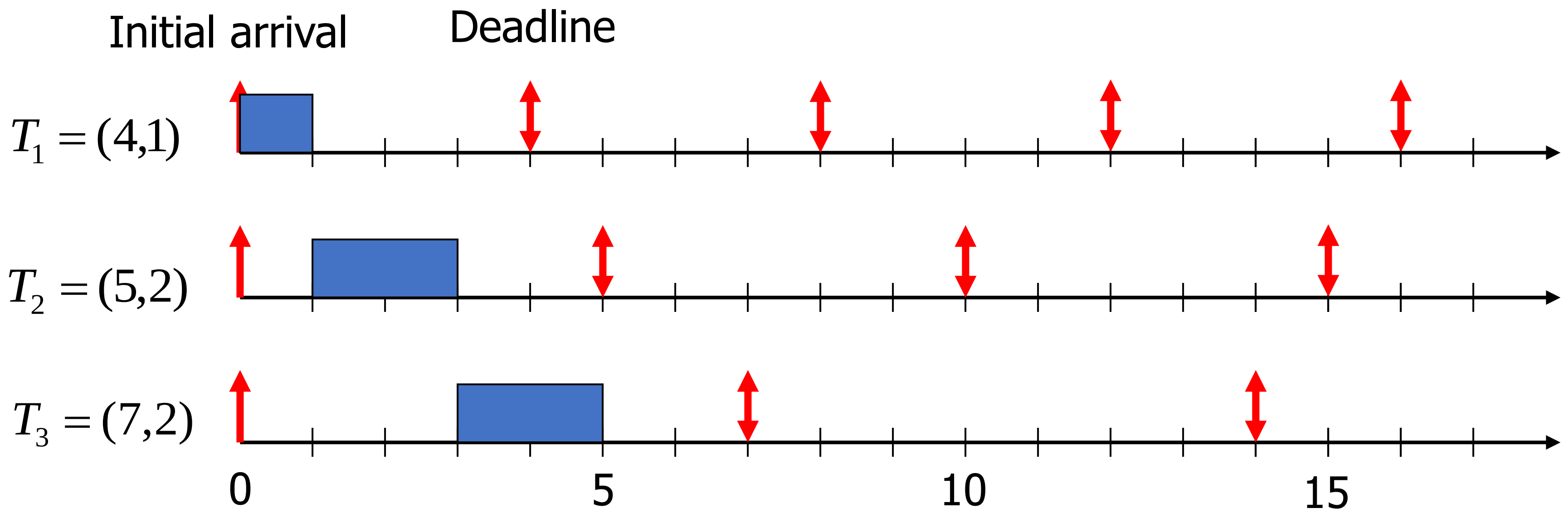
Earliest Deadline First (EDF) Scheduling

- Priority scheduling with pre-emption
- Highest priority given to task with soonest deadline
  - Task = (Period, Duration)



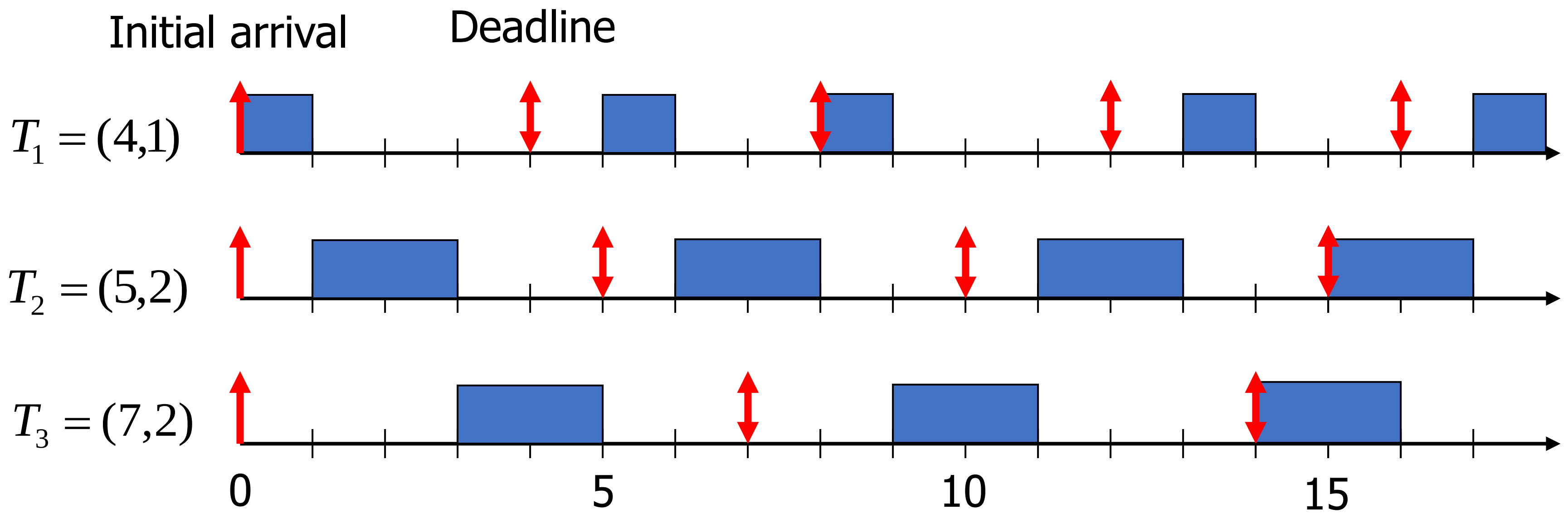
Earliest Deadline First (EDF) Scheduling

- Priority scheduling with pre-emption
- Highest priority given to task with soonest deadline
  - Task = (Period, Duration)



Earliest Deadline First (EDF) Scheduling

- Priority scheduling with pre-emption
- Highest priority given to task with soonest deadline
  - Task = (Period, Duration)



Schedulability test for EDF

- Guarantees schedule feasibility if total load is not more than 100%

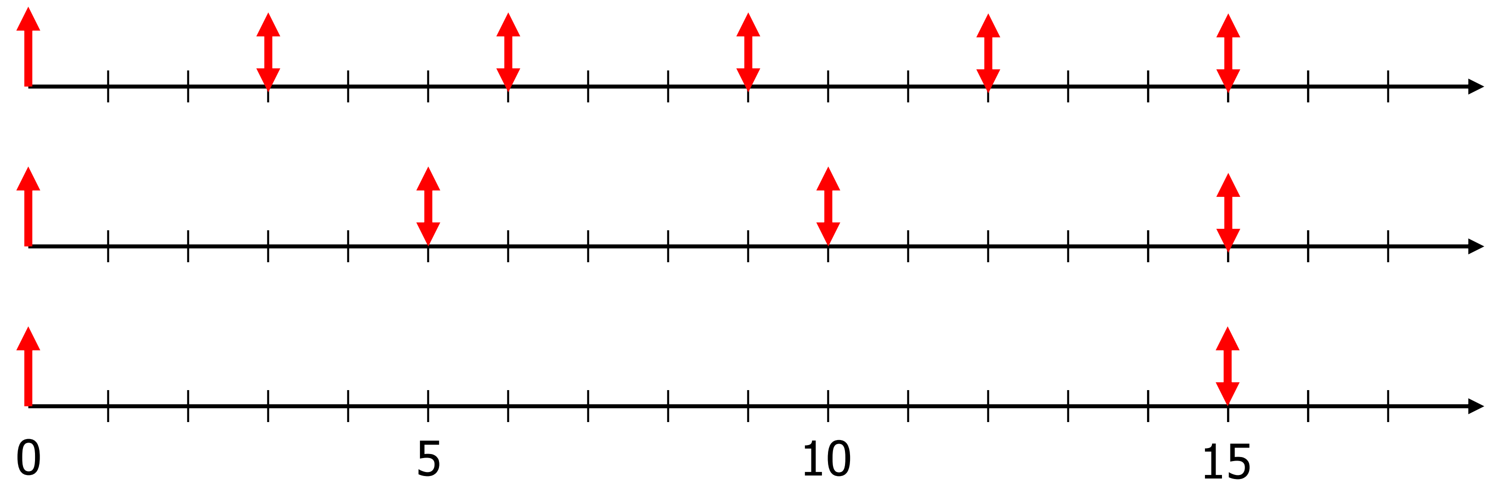
- For *n* tasks with computation time *C* and deadline (period) *D*
  - A feasible schedule exists if utilization is less than or equal to one:

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i} \right) \leq 1$$

Check your understanding

- Can we schedule the following workload?
  - Job A: period 3, computation 1
  - Job B: period 5, computation 2
  - Job C: period 15, computation 4

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

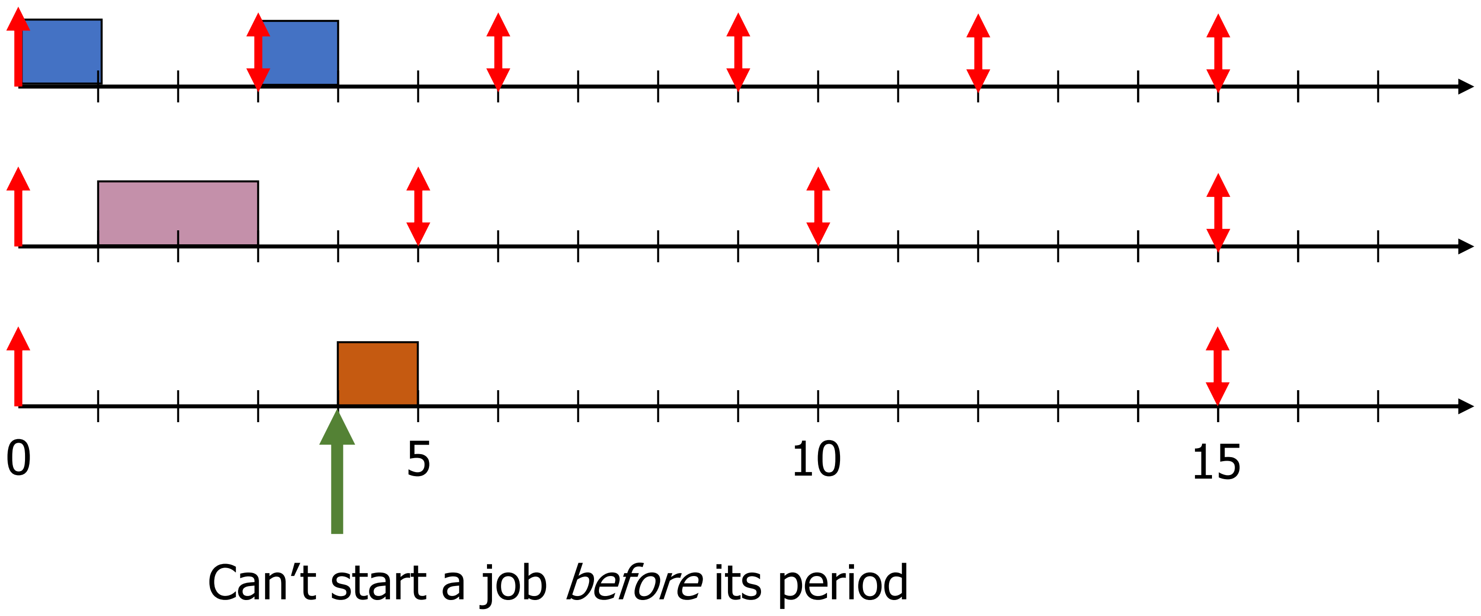


Check your understanding

- Can we schedule the following workload?
  - Job A: period 3, computation 1
  - Job B: period 5, computation 2
  - Job C: period 15, computation 4

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i} \right) \leq 1$$

1/3 + 2/5 + 4/15 = 1

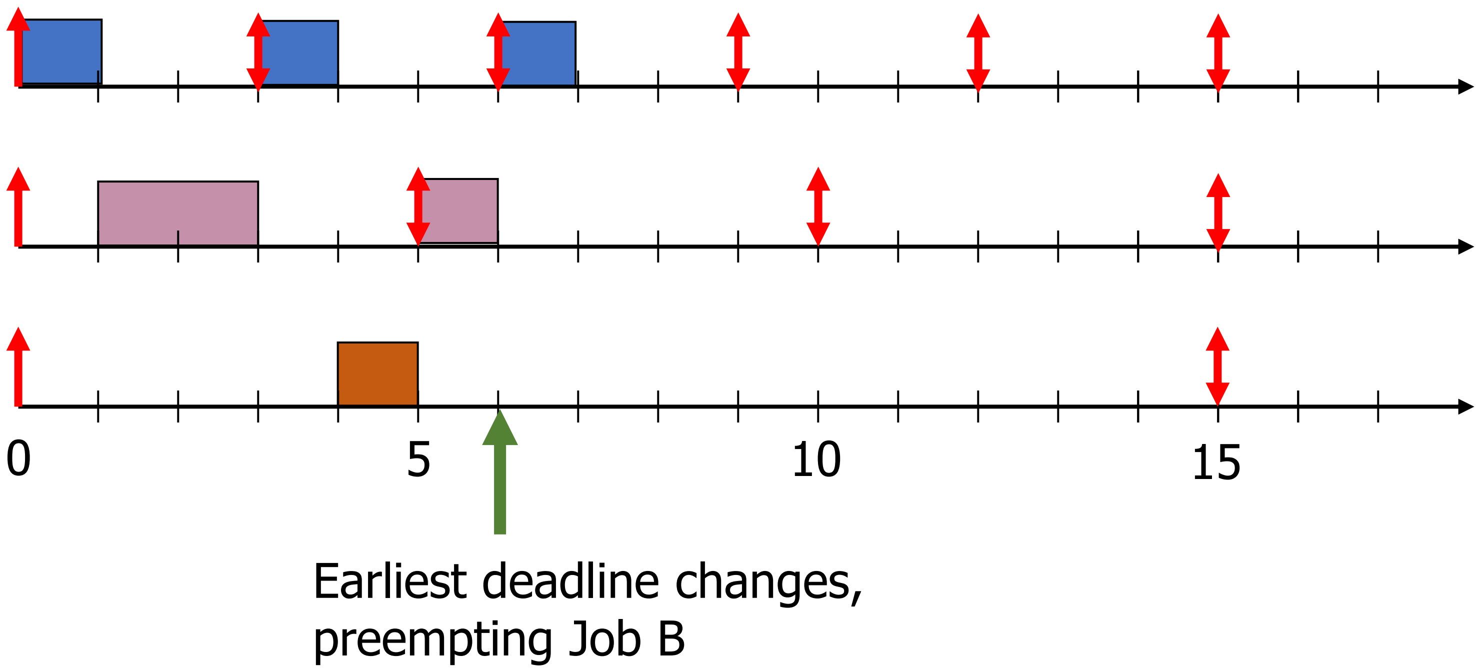


Check your understanding

- Can we schedule the following workload?
  - Job A: period 3, computation 1
  - Job B: period 5, computation 2
  - Job C: period 15, computation 4

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

1/3 + 2/5 + 4/15 = 1

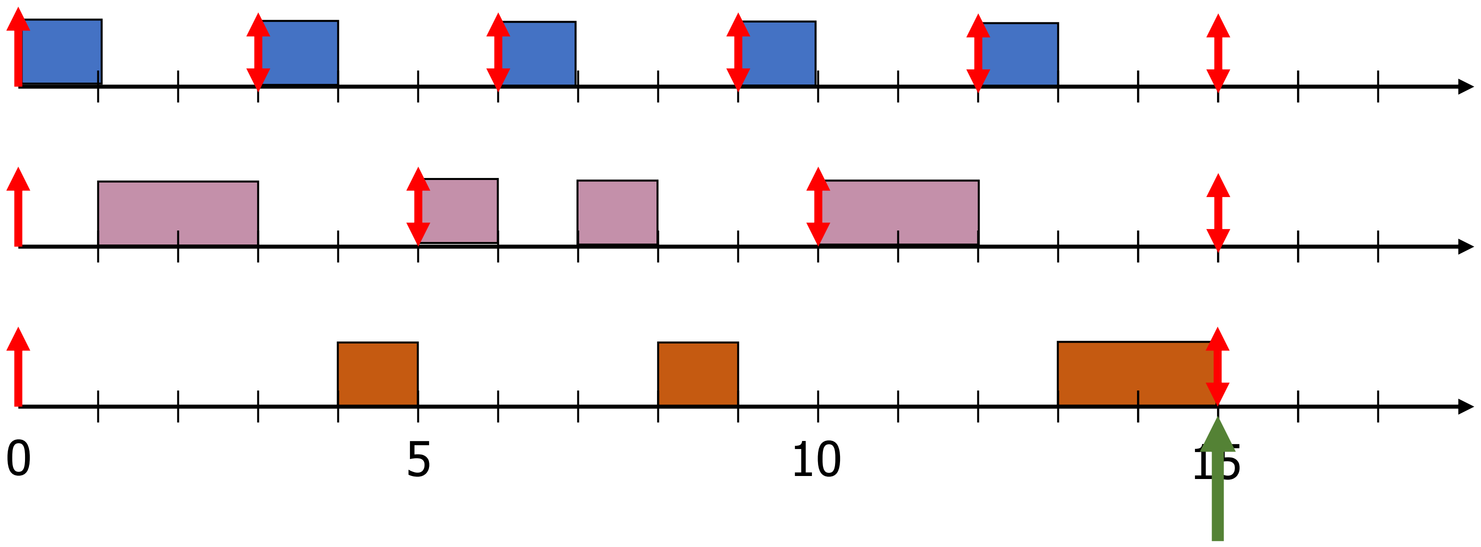


Check your understanding

- Can we schedule the following workload?
  - Job A: period 3, computation 1
  - Job B: period 5, computation 2
  - Job C: period 15, computation 4

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

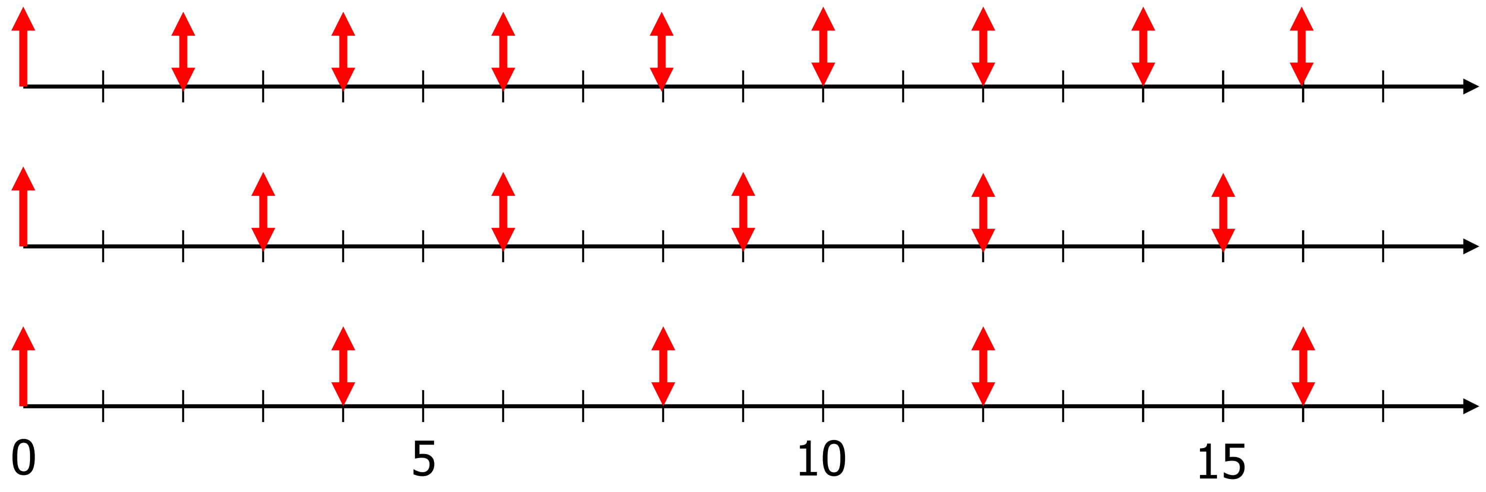
1/3 + 2/5 + 4/15 = 1



Check your understanding

- Can we schedule the following workload?
  - Job A: period 2, computation 1
  - Job B: period 3, computation 1
  - Job C: period 4, computation 1

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i} \right) \leq 1$$

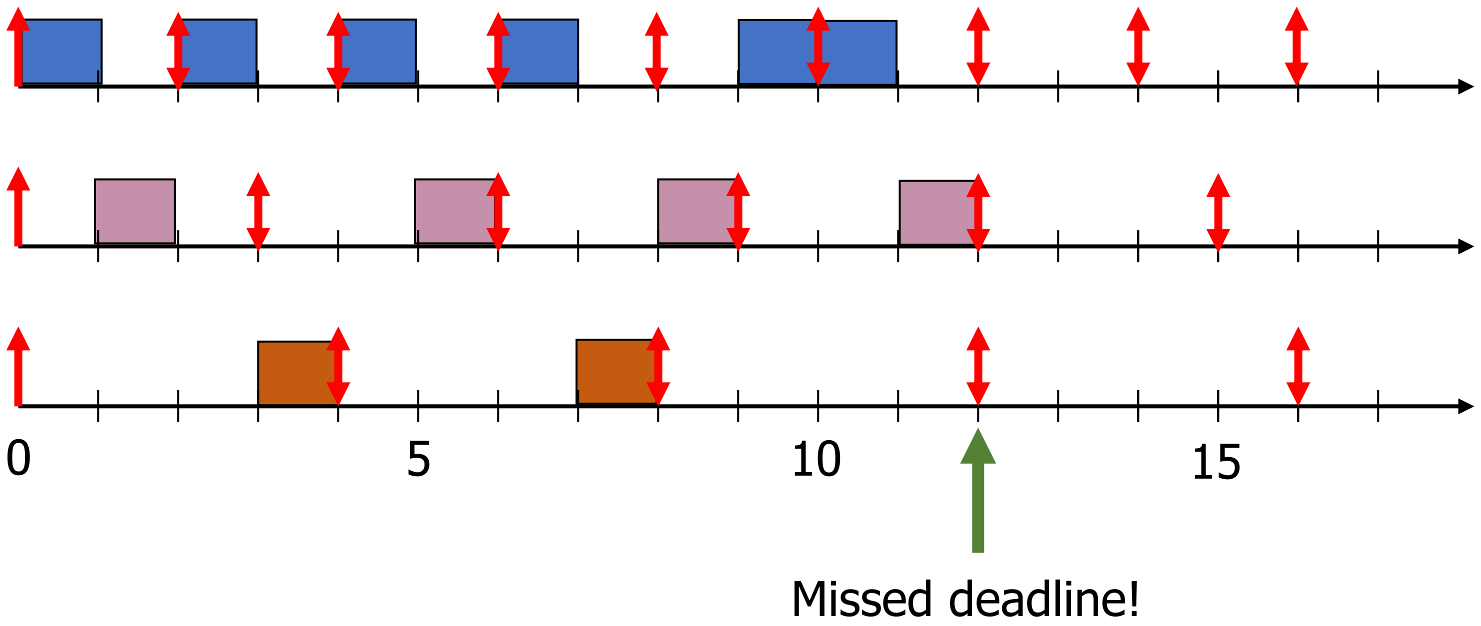


Check your understanding

- Can we schedule the following workload?
  - Job A: period 2, computation 1
  - Job B: period 3, computation 1
  - Job C: period 4, computation 1

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

1/2 + 1/3 + 1/4 = 1.08



Earliest Deadline First tradeoffs

Good qualities

- Simple concept and simple schedulability test
- Excellent CPU utilization

Bad qualities

- Hard to implement in practice
  - Need to constantly recalculate task priorities
  - CPU time spent in scheduler needs to be counted against load
- Unstable: Hard to predict which job will miss deadline
  - Utilization was greater than 1, so we knew there was a problem
  - But we had to work out the whole schedule to see Job C missed

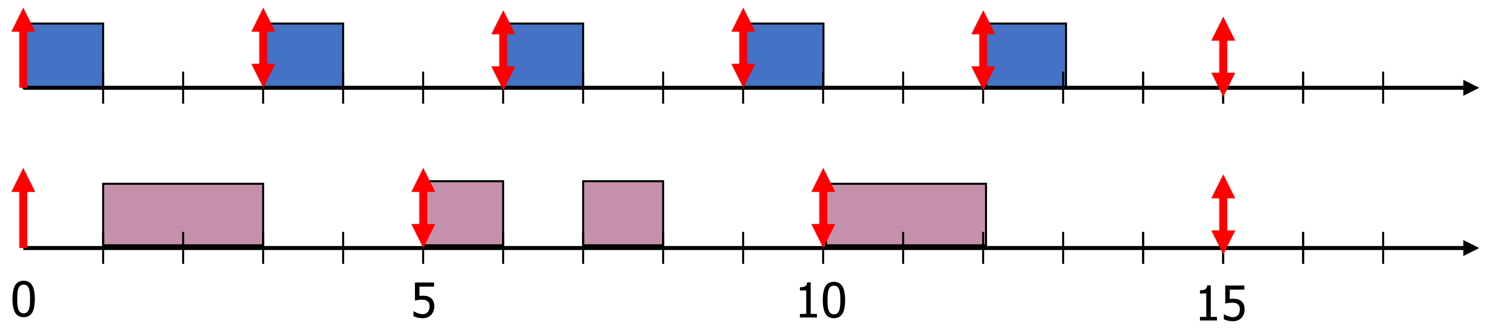
Rate Monotonic Scheduling (RMS)

- Priority scheduling
- Assign fixed priority of 1/Period for each job
  - Makes the scheduling algorithm simple and stable
  - Only lowest priority jobs might miss deadlines

- If any fixed priority scheduling algorithm can schedule a workload, So can Rate Monotonic Scheduling

Rate Monotonic Scheduling example

- Schedule the following workload with RMS
  - Job A: period 3, computation 1 -> Priority 1/3
  - Job B: period 5, computation 2 -> Priority 1/5

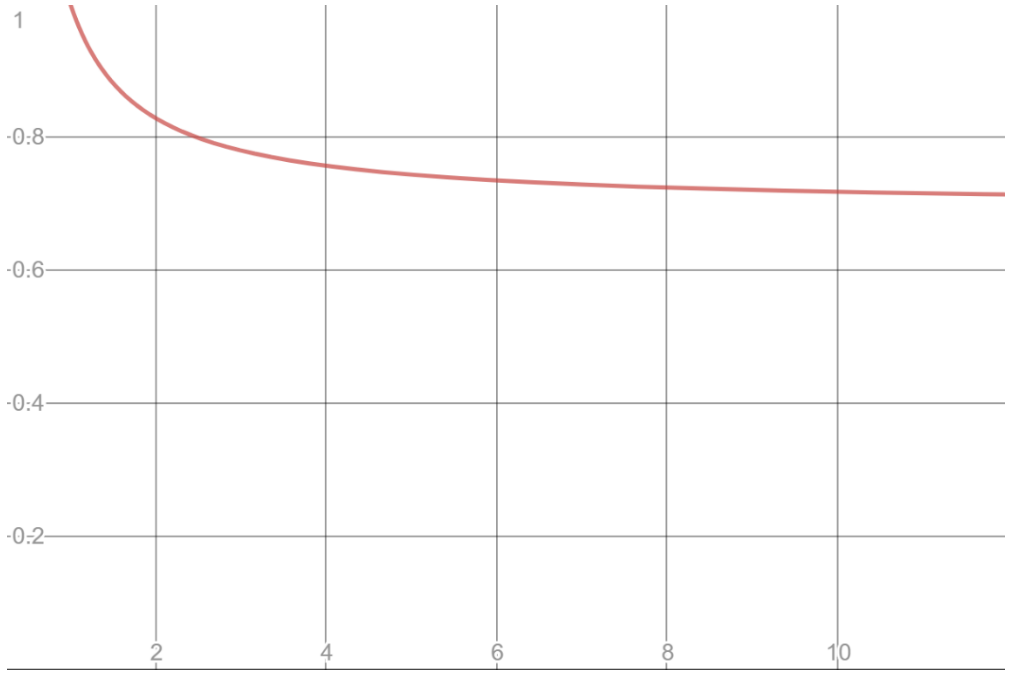


Schedulability test for RMS

- Schedulability is more complicated for RMS unfortunately
  - For a workload of *n* jobs with computation time *C* and period *D*

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq n * (2^{\frac{1}{n}} - 1)$$

Lower Bound on schedulability



- U(1) = 1.0
- U(2) = 0.828
- U(3) = 0.779

  …
- U(∞) = 0.693

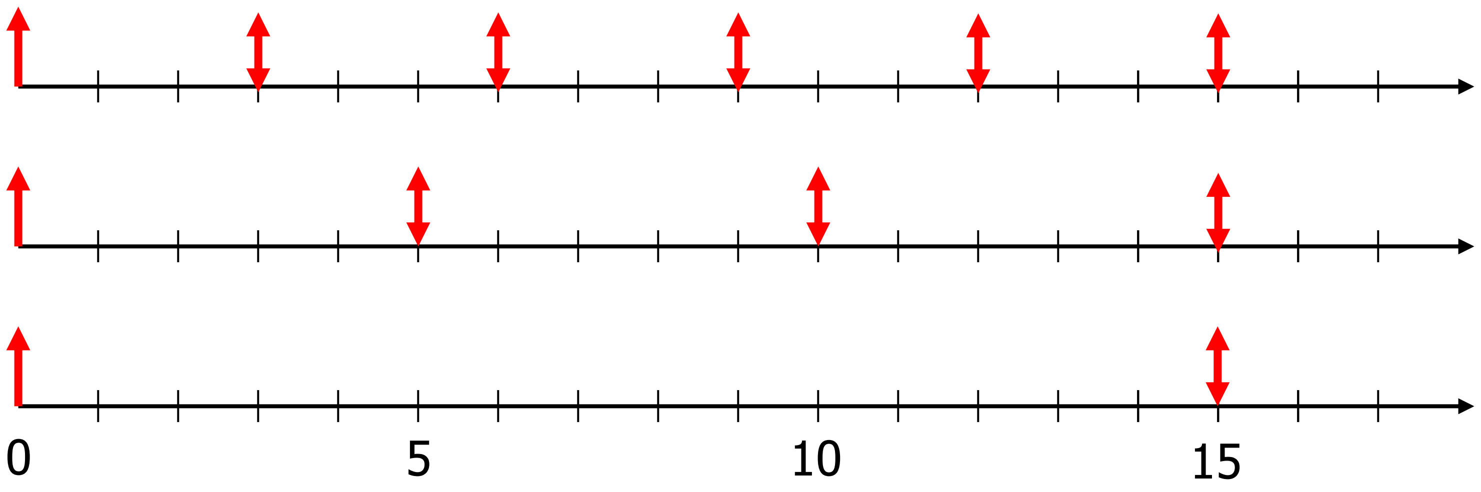
RMS schedulability test is conservative

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq n * (2^{\frac{1}{n}} - 1)$$

- $0 \leq U \leq n * (2^{\frac{1}{n}} - 1)$
  - Schedulable! (so less than 69% is always schedulable)
- $n * (2^{\frac{1}{n}} - 1) < U \leq 1$
  - Maybe schedulable
- $1 < U$
  - Not schedulable

Check your understanding

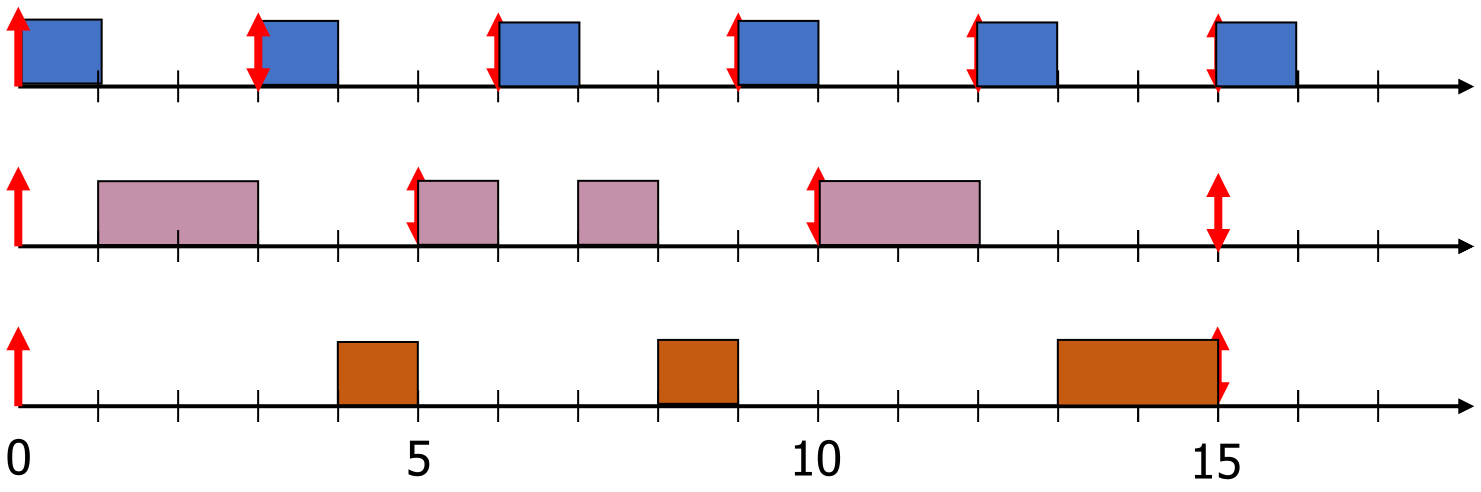
- Can we schedule the following workload with RMS?
  - Job A: period 3, computation 1
  - Job B: period 5, computation 2
  - Job C: period 15, computation 4



Check your understanding

- Can we schedule the following workload with RMS?
  - Job A: period 3, computation 1 -> Highest priority
  - Job B: period 5, computation 2 -> Middle priority
  - Job C: period 15, computation 4 -> Lowest priority

U = 1
Maybe schedulable!



Rate Monotonic Scheduling tradeoffs

Upsides

- Still conceptually simple
- Easy to implement
- Stable (lower priority jobs will fail to meet deadlines in overload)

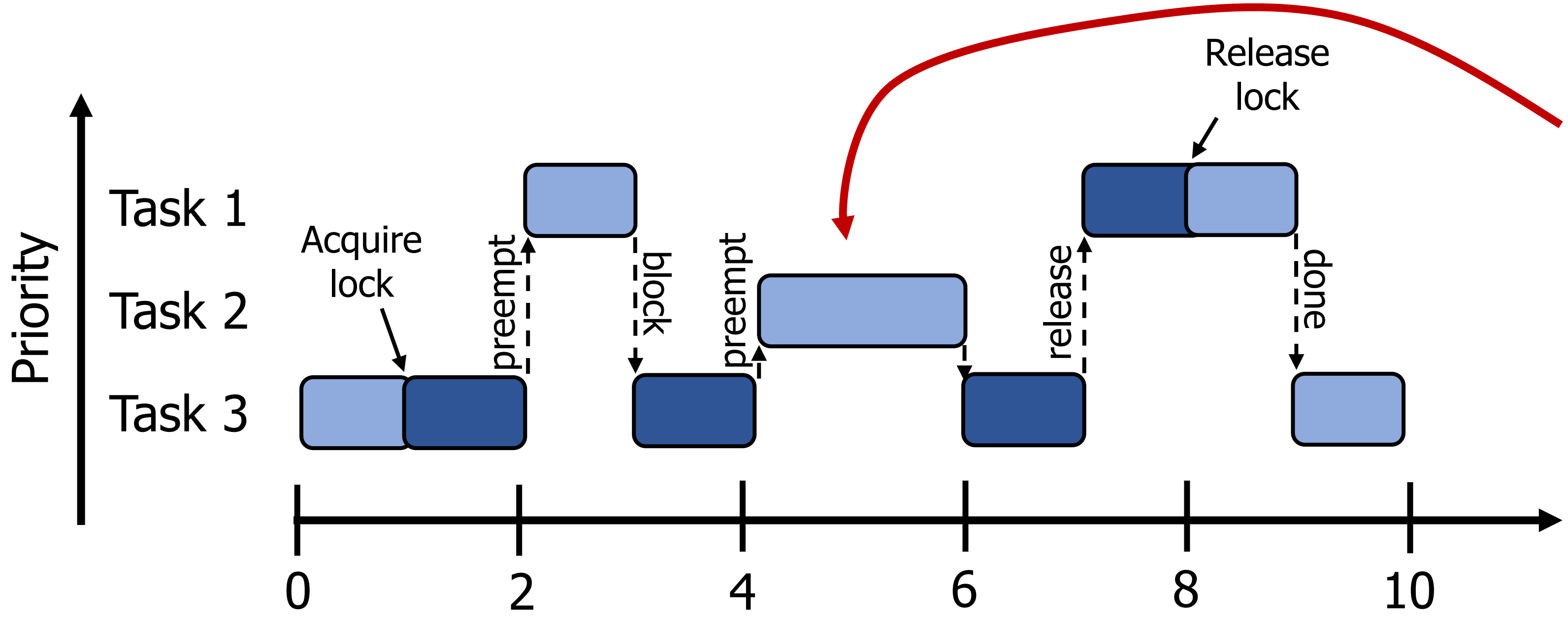
Downsides

- Lower CPU utilization
    - Might not be able to utilize more than 70% of the processor
- Non-precise schedulability analysis

A problem with priority schedulers: priority inversion

- Other concepts from OS still apply when we're scheduling
  - Particularly locks and synchronization

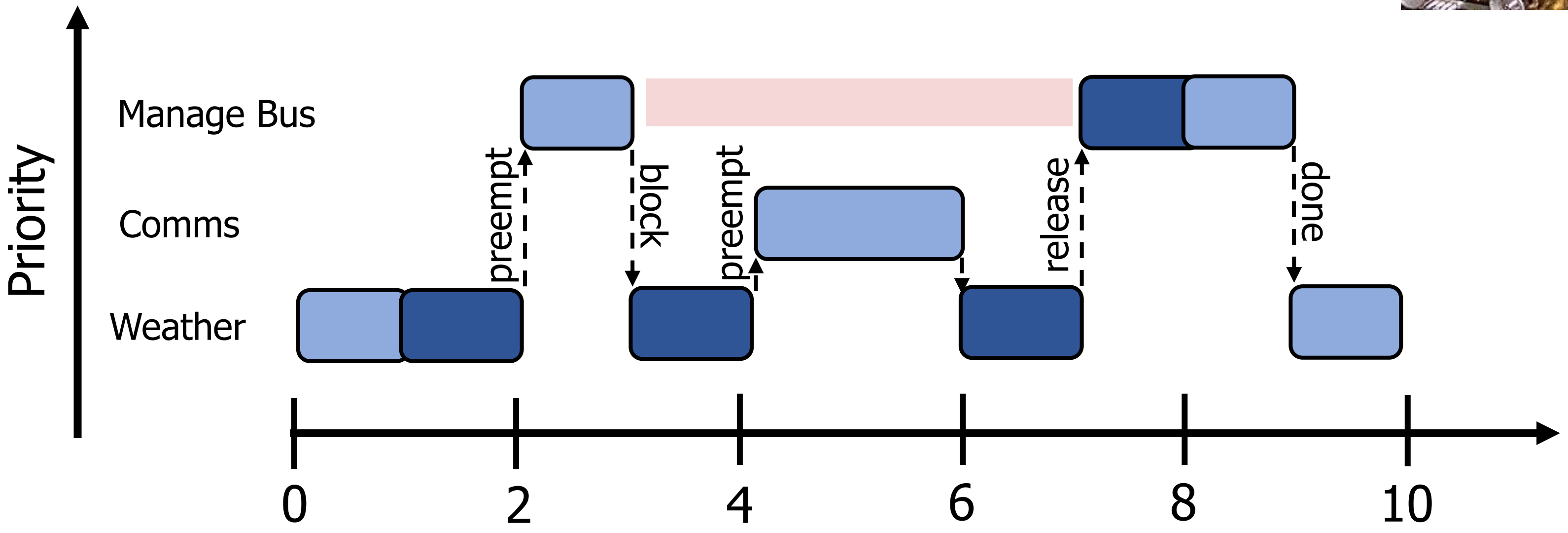
- Imagine Task 1 and Task 3 both need to share a lock



Task 1 is waiting on Task 2!!

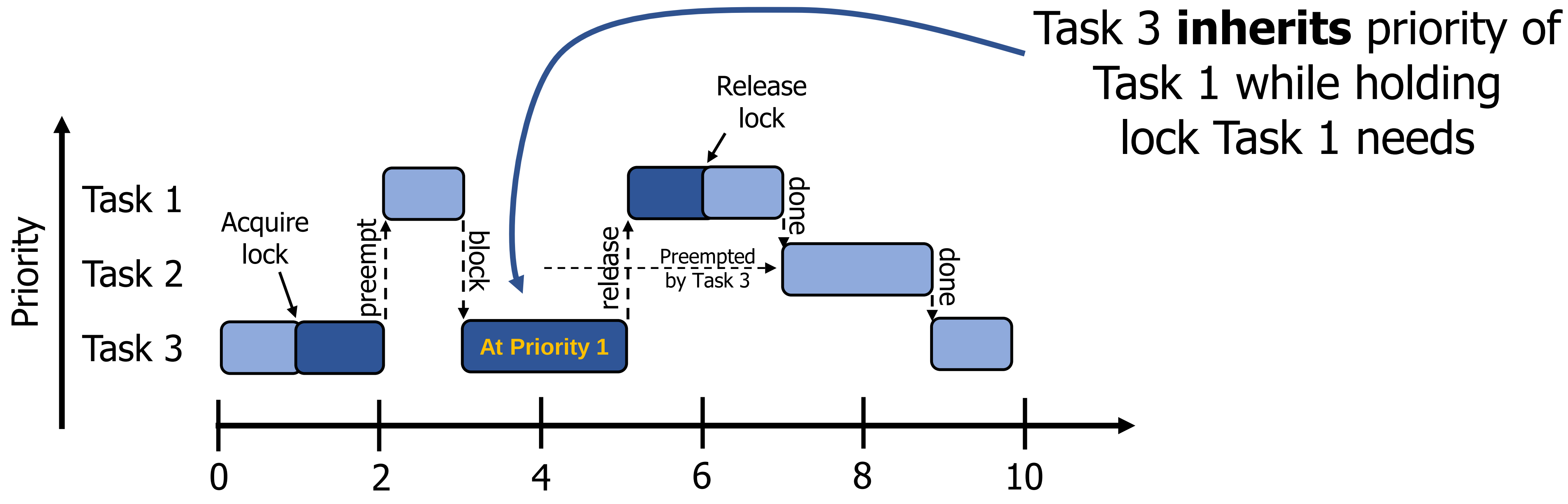
Priority inversion occurred on Pathfinder!

- Bus management missed deadlines while waiting on meteorology because medium-priority tasks were taking too long
  - System rebooted when deadline was missed



Priority inheritance solution to priority inversion

- A solution is to temporarily increase priority for tasks holding resources that high priority tasks need

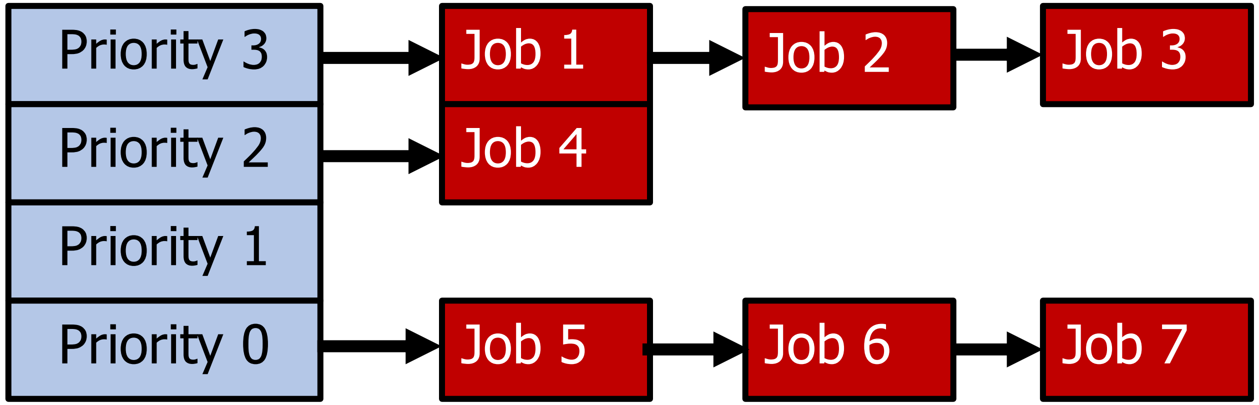


Outline

- Real Time Operating Systems
  - Earliest Deadline First scheduling
  - Rate Monotonic scheduling
- **Modern Operating Systems**
  - Linux O(1) scheduler
  - Lottery and Stride scheduling
  - Linux Completely Fair Scheduler

Priority scheduling policies

- Systems may try to set priorities according to some **policy goal**
- MLFQ Example:
  - Give interactive higher priority than long calculation
  - Prefer jobs waiting on I/O to those consuming lots of CPU
- Try to achieve fairness:
  - elevate priority of threads that don’t get CPU time (ad-hoc, bad if system overloaded)



Linux O(1) scheduler (Linux 2.6)

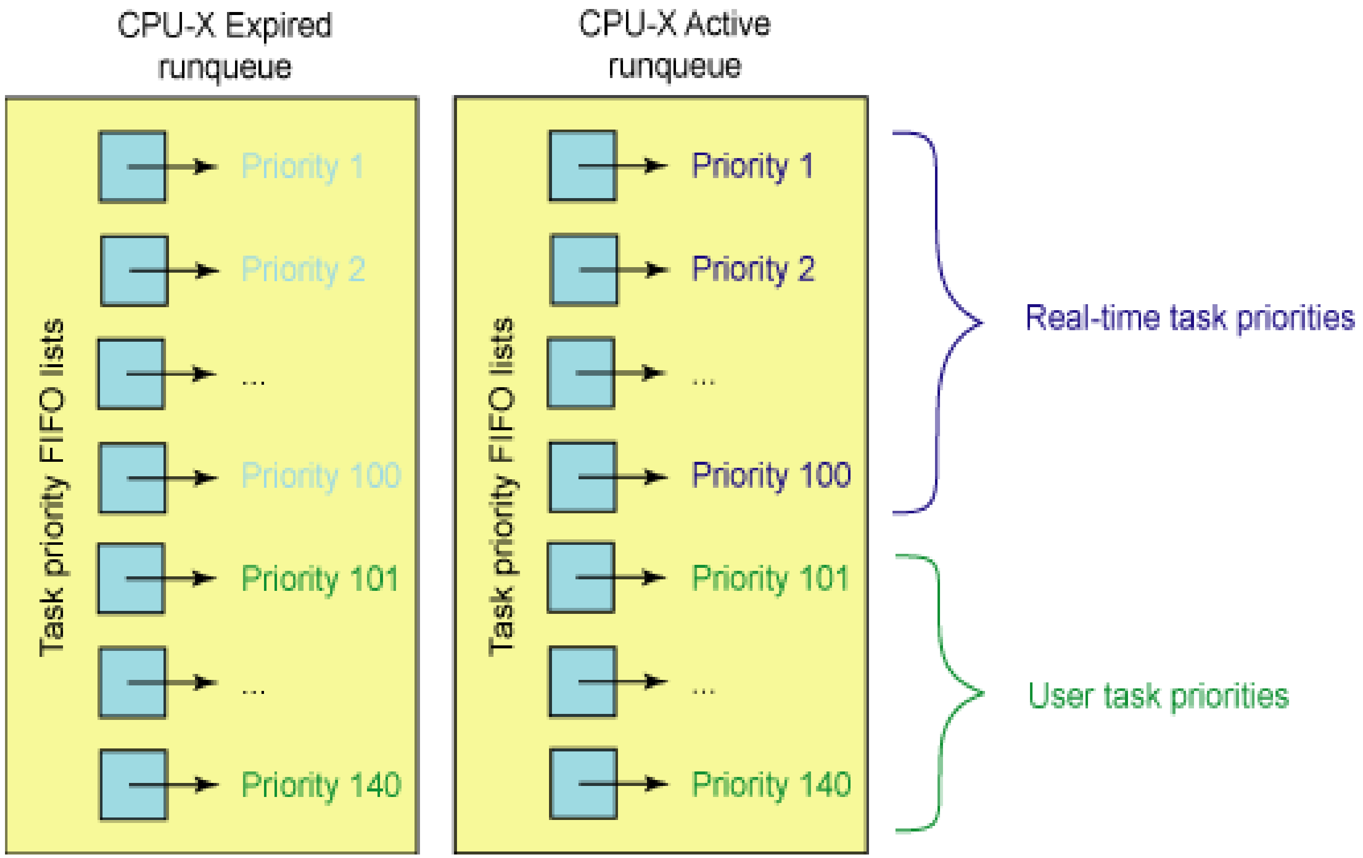
- Goals
  - Keep the runtime of the scheduler itself short
    - Avoid O(n) algorithms
    - Instead only make adjustments to a single job when it is swapped
  - Predictable algorithm
  - Identify interactive versus noninteractive processes with heuristics
    - Processes with long average sleep time get a priority boost

- Note my machines right now:
  - Ubuntu VM: 332 processes (867 threads)
  - Windows: 224 processes (2591 threads)
  - MacOS: 430 processes (2249 threads)

Priority in Linux O(1) scheduler

- MLFQ-Like Scheduler with 140 Priority Levels
  - 40 for user tasks, 100 soft “realtime” tasks
- Timeslice depends on priority – linearly mapped onto timeslice range

Workings of the O(1) scheduler



- Round robin at priority levels like MLFQ
- Each priority level gets a run quota
- On expiration of quota
  - Recalculate priority
  - Insert in expired queue
- When all jobs are gone from active queue
  - Swap expired and active queue pointers

Priorities can lead to starvation

- The policies we've studied so far:
  - **Always prefer to give the CPU to a prioritized job**
  - Non-prioritized jobs may never get to run

- But priorities were a means, not an end
- Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  - Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  - Give the interactive ones enough CPU to respond to an input and wait (on those slow humans)
  - Let the CPU bound ones grind away without too much disturbance

Idea: proportional-share scheduling

- Many of the policies we've studied always prefer to give CPU to a prioritized job
  - Non-prioritized jobs may never get to run

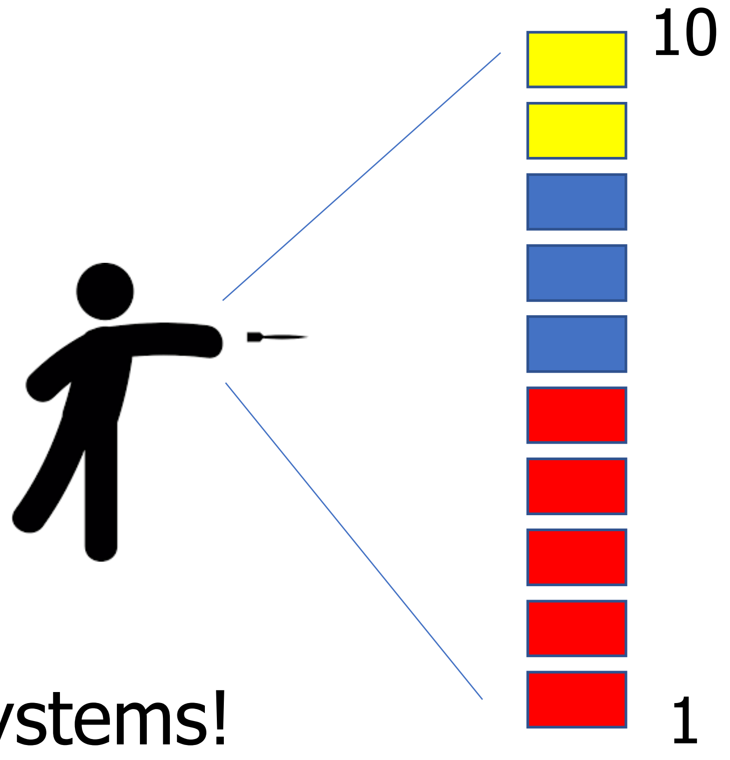
- Instead, we can share the CPU proportionally
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)

Lottery Scheduling

- Give out "tickets" according to proportion each job should receive
- Every quantum:
  - Draw one ticket at random
  - Schedule that job to run
- If there are N jobs, probability of pick a job is:

$$\frac{priority(job_i)}{\sum_{j=0}^{n-1} priority(job_j)}$$

- Definitely not suitable for real-time systems!



Stride Scheduling

- Same idea, but lets remove the random element
- Give each job a stride number inversely proportional to tickets
  - A=100 tickets, B=50 tickets, C=250 tickets
  - A=100 stride, B=200 stride, C= 40 stride
- Scheduler
  - Pick job with lowest cumulative strides and run it
  - Increment its cumulative strides by its stride number
- Essentially: low-stride (high-ticket) jobs get run more often

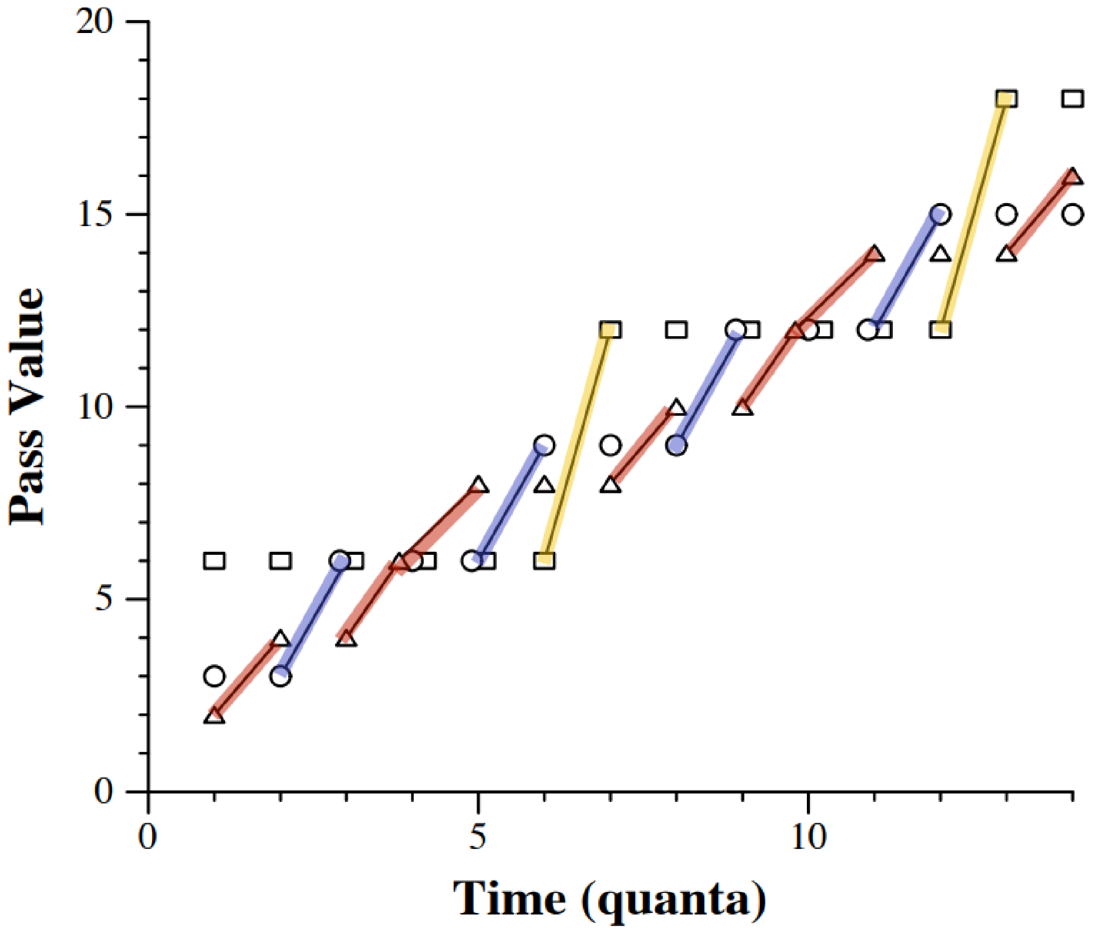
Stride scheduling in practice

Triangle is high priority (low stride)

Circle is medium priority

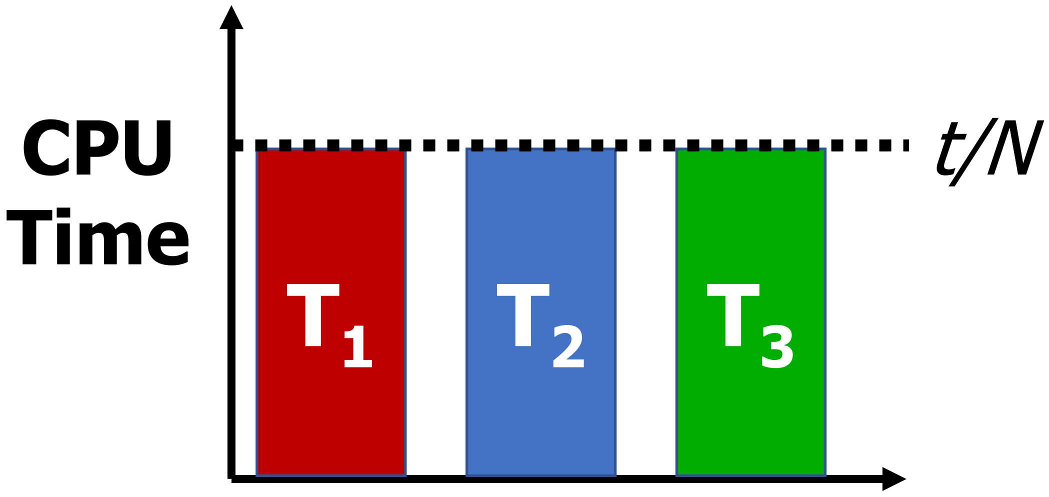
Square is low priority (high stride)

- "Pass value" is the cumulative stride count
- Each colored line is an instance where a job runs
  - And stride count is increased afterwards



Proportional-share scheduling is impossible instantaneously

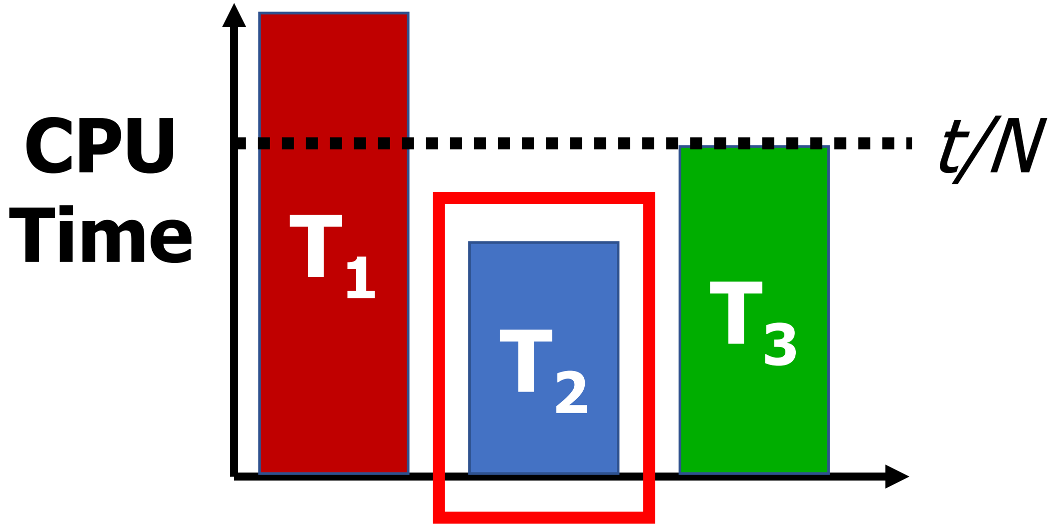
At *any* time *t*
we want to observe:



- Goal: each process gets an equal share of processor
- N threads "simultaneously" execute on $1/N^{th}$ of processor
- Doesn't work in the real world
  - Jobs block on I/O
  - OS needs to give out timeslices

Linux Completely Fair Scheduler (CFS)

What if we make shares proportional over a longer period?



- Track processor time given to job so far
- Scheduling decision
  - Choose thread with minimum processor time to schedule
  - "Repairs" illusion of fairness
- Update processor time when the job finishes
  - Timeslice expiration is a big update
  - Blocking I/O results in maintaining small processor time

Linux CFS: responsiveness and throughput

- Constraint 1: target latency
  - Want a maximum duration before a job gets some service
  - Dynamically set timeslice based on number of jobs
  - Quanta = Target_latency / N
  - 20 ms max latency => 5 ms timeslice for 4 jobs, or 0.1 ms for 200 jobs

- Check your understanding. What’s the problem here?

Linux CFS: responsiveness and throughput

- Constraint 1: target latency
  - Want a maximum duration before a job gets some service
  - Dynamically set timeslice based on number of jobs
  - Quanta = Target_latency / N
  - 20 ms max latency => 5 ms timeslice for 4 jobs, or 0.1 ms for 200 jobs

- Check your understanding. What’s the problem here?
  - Timeslice needs to stay much greater than context switch time

Linux CFS: responsiveness and throughput

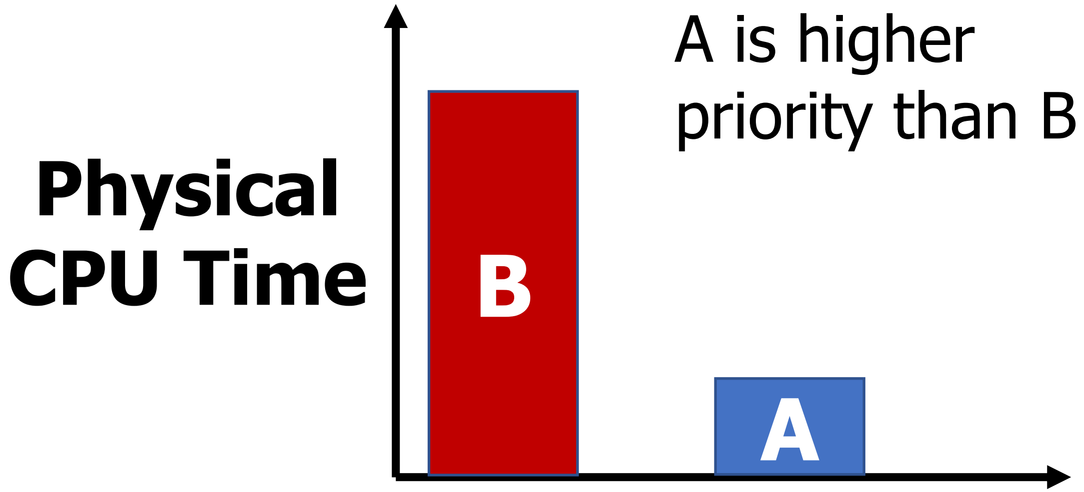
- Constraint 1: target latency
  - Want a maximum duration before a job gets some service
  - Dynamically set timeslice based on number of jobs
  - Quanta = Target_latency / N
  - 20 ms max latency => 5 ms timeslice for 4 jobs, or 0.1 ms for 200 jobs

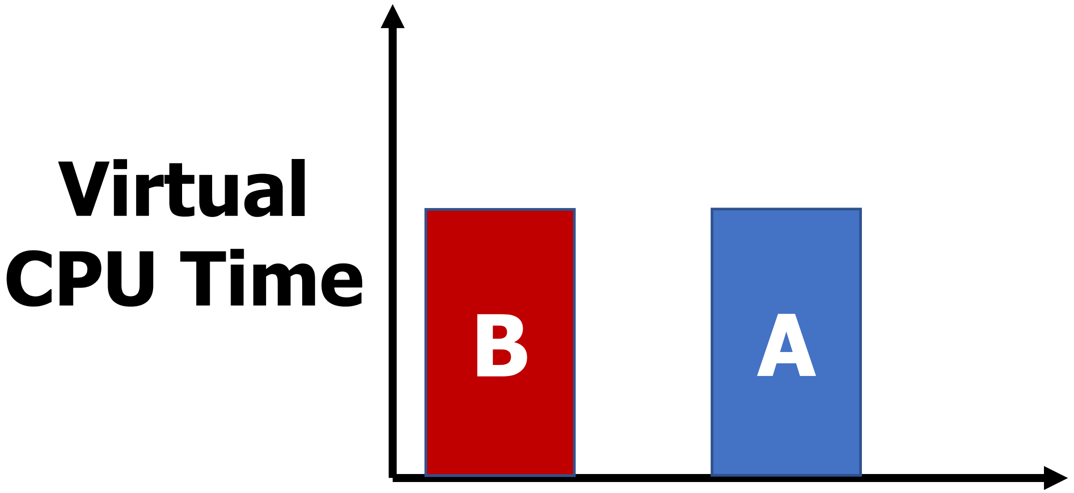
- Constraint 2: avoid excessive overhead
  - Don’t want to spend all our time context switching if there are many jobs
  - Set a minimum length for timeslices
  - Quanta = min(Target_latency/N, 1)

CFS priorities are applied as “virtual runtime”

- Track a job’s virtual runtime
  - Higher priority jobs have slower virtual runtime
  - Lower priority have faster virtual runtime

- Scheduler’s decisions on made to evenly proportion virtual runtime





Multicore scheduling

- Algorithmically, not a huge difference from single-core scheduling
  - Difference is the need to distribute work among cores

- *Affinity scheduling*: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  - Cache reuse
  - Grouping threads could help or hurt…

- Implementation-wise, helpful to have *per-core* scheduling data structures

Summary on schedulers

If You care About:	Then Choose:
CPU Throughput	First-In-First-Out
Average Turnaround Time	Shortest Remaining Processing Time
Average Response Time	Round Robin
Favoring Important Tasks	Priority
Fairness (CPU Time)	Linux CFS
Meeting Deadlines	EDF or RMS

Outline

- Real Time Operating Systems
  - Earliest Deadline First scheduling
  - Rate Monotonic scheduling

- Modern Operating Systems
  - Linux O(1) scheduler
  - Lottery and Stride scheduling
  - Linux Completely Fair Scheduler