Lecture 04: Advanced Scheduling

CS343 – Operating Systems
Branden Ghena – Fall 2022

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

- Scheduling Overview
- Scheduler Metrics

- Batch Systems
    1. First In First Out scheduling
    2. Shortest Job First scheduling
    3. Shortest Remaining Processing Time scheduling

- **Interactive Systems**
    1. Round Robin scheduling
    2. **Multi-Level Feedback Queue scheduling**

Jobs can be I/O-bound or CPU-bound

- CPU-bound process
  - Lots of computation between each I/O request
  - Actually needs to do computation on a processor
  - Example: doing matrix math

- I/O-bound process
  - Very little computation between each I/O request
  - Just needs a processor to figure out its next I/O request
  - Example: searching a file system for a file name

Scheduling goal: I/O-bound before CPU-bound

- First maximize I/O
  - Run the I/O-bound jobs as quickly as possible,
  - So they can send next I/O request,
  - And our disks, network cards, etc. are maximally used
  - And our processor becomes free again quickly (faster than a timeslice)

- Then fill up the processor(s)
  - Lots of room for multiprogramming between the I/O requests
  - Blocked jobs are still “progressing” as their I/O is fetched

Scheduling goal: I/O-bound before CPU-bound

- First maximize I/O
  - Run the I/O-bound jobs as quickly as possible,
  - So they can send next I/O request,
  - And our disks, network cards, etc. are maximally used

- Then fill up the processor(s)
  - Lots of room for multiprogramming between the I/O requests
  - Blocked jobs are still “progressing” as their I/O is fetched

- But how do you know when a job is going to use I/O?
  - Can’t know the future
  - Can track past behavior of the job

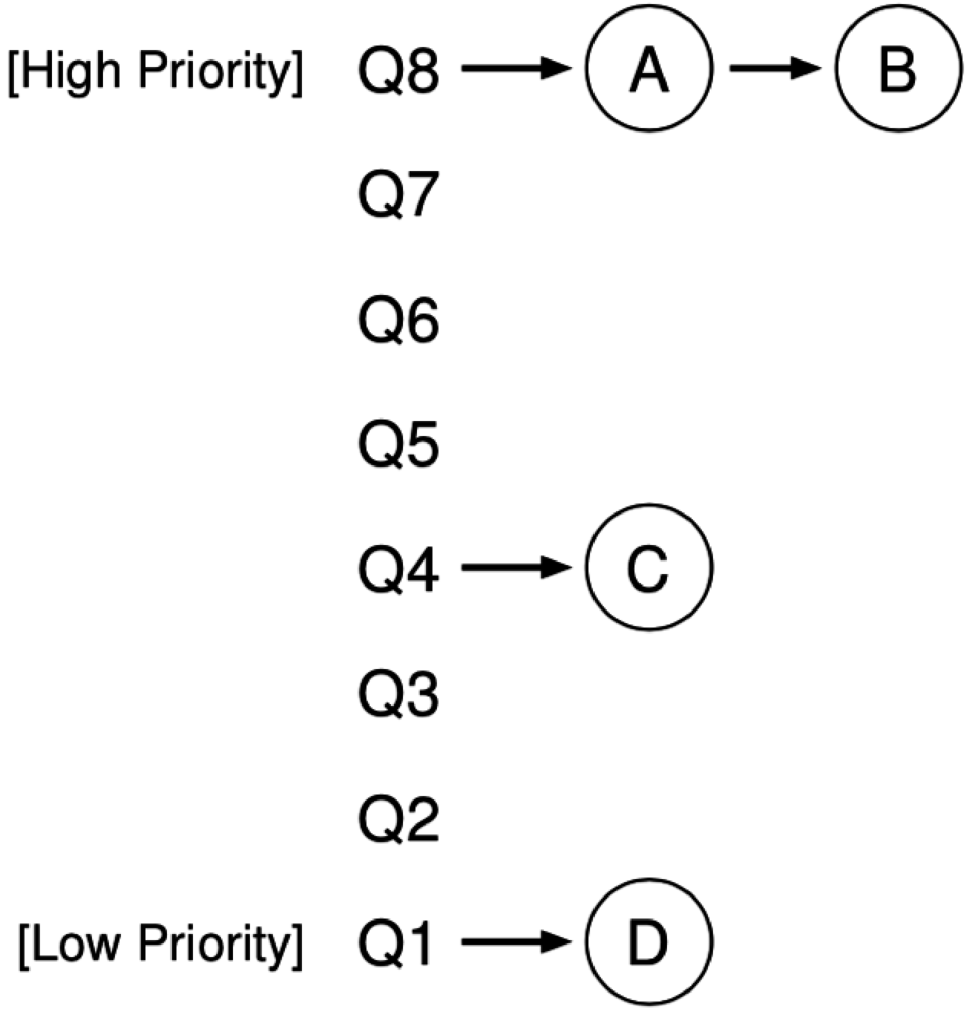
2. Multi-Level Feedback Queue (MLFQ)

- General purpose scheduler to support multiple goals
  - Good response time for interactive jobs
  - Good turnaround time for batch jobs
  - Achieves this by prioritizing I/O bound jobs over CPU bound jobs

- Policy
  - Automatically attach priority to jobs:
    - Interactive, I/O bound jobs should be highest priority
    - CPU bound, batch jobs should be lowest priority
    - Apply different round robin timeslices to each priority level

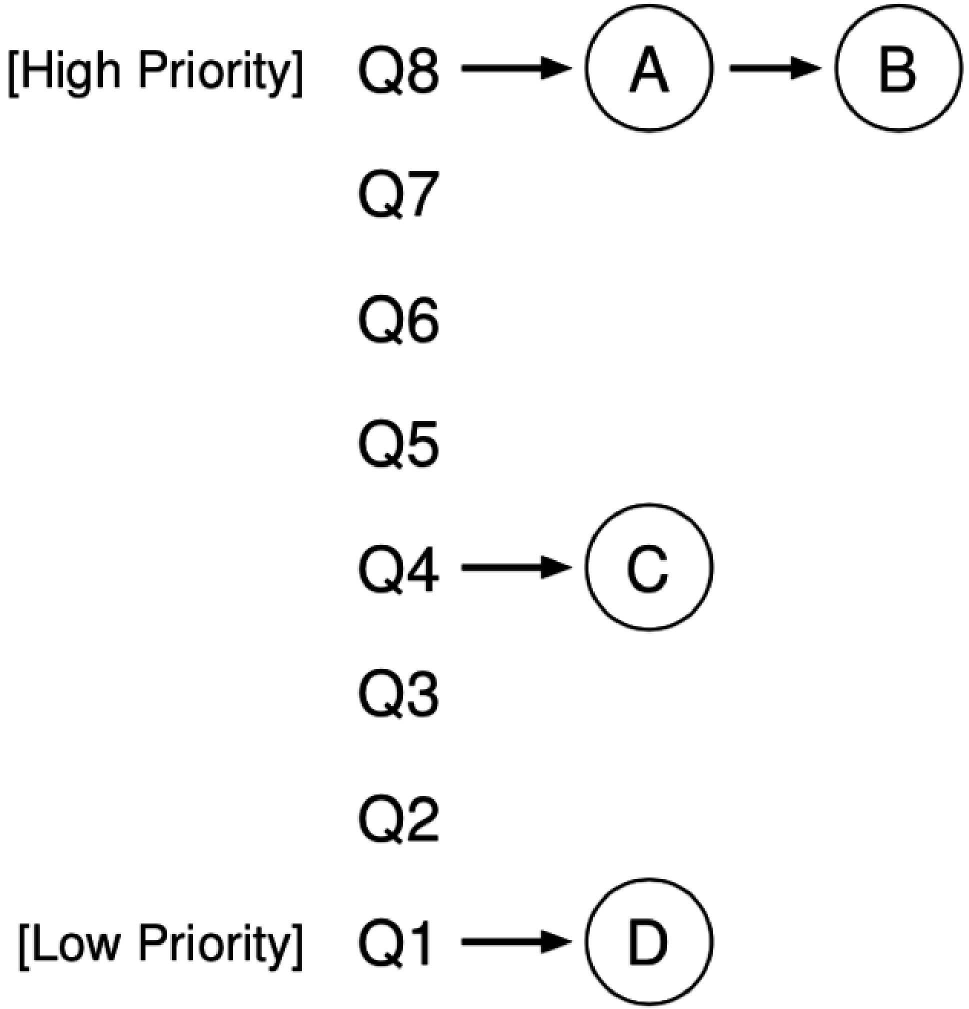
Multi-Level Feedback Queue Details

- Run highest priority level available
  - Round robin among jobs there
- When all jobs at a level are blocked on I/O
  - Move down to next lower level
- Long running jobs lose priority
  - Processor usage quota at a given level
  - When used up, demote job one level

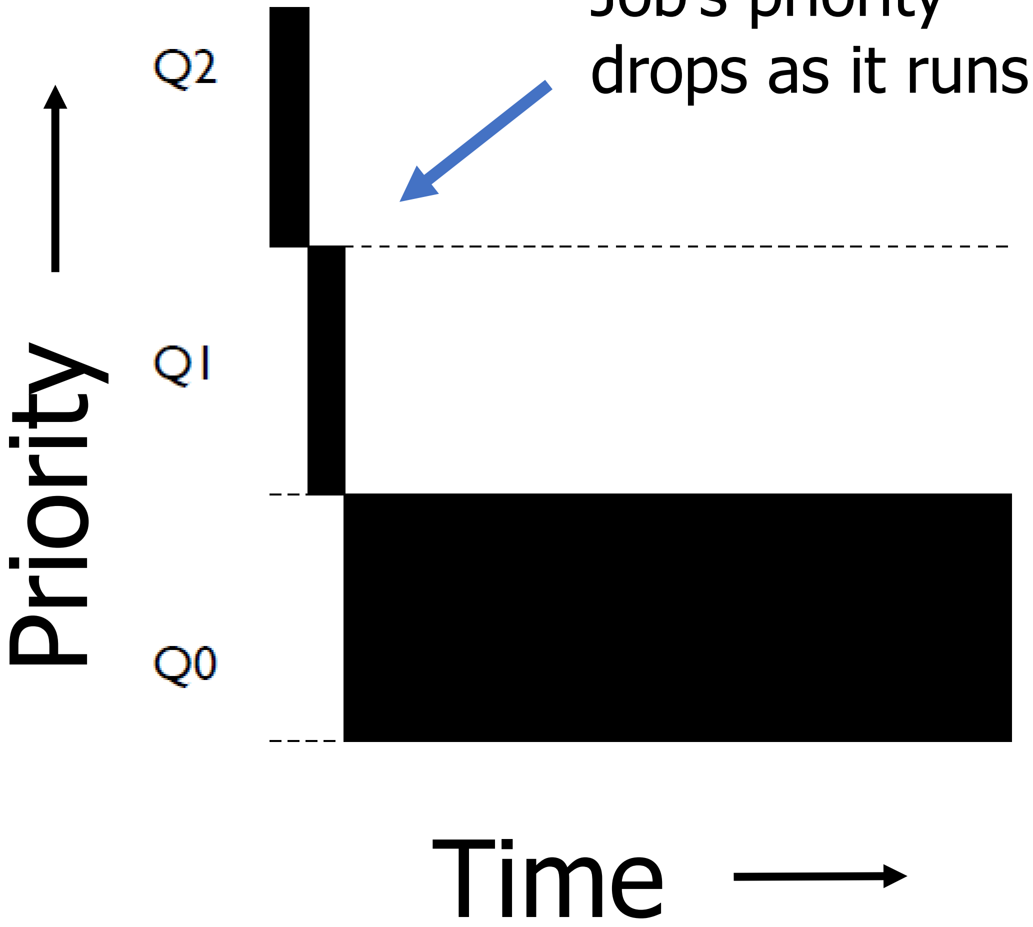


MLFQ Rules

1. If Priority($\mathbf{J_1}$) > Priority($\mathbf{J_2}$), $\mathbf{J_1}$ runs
2. If Priority($\mathbf{J_1}$) = Priority($\mathbf{J_2}$), $\mathbf{J_1}$ and $\mathbf{J_2}$ run in Round Robin
3. Jobs start at top priority
4. When a job uses its time quota for a level, demote it one level
5. Every **S** seconds, reset priority of all jobs to top

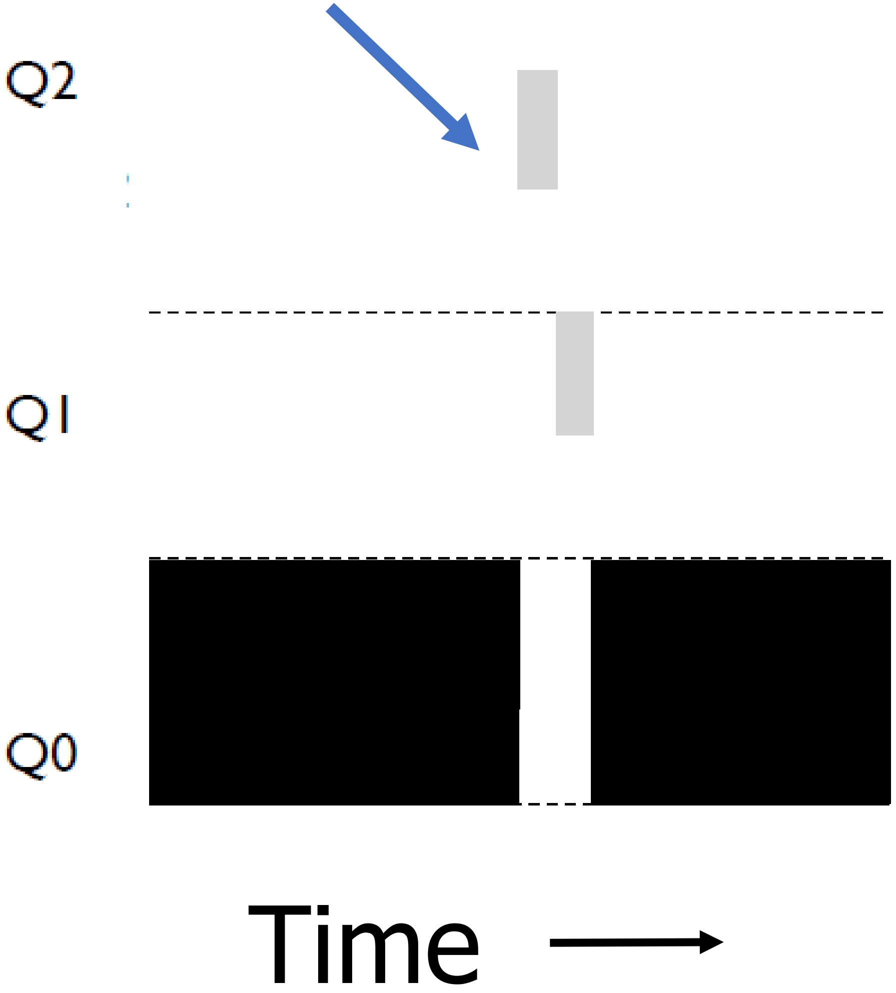


MLFQ Example



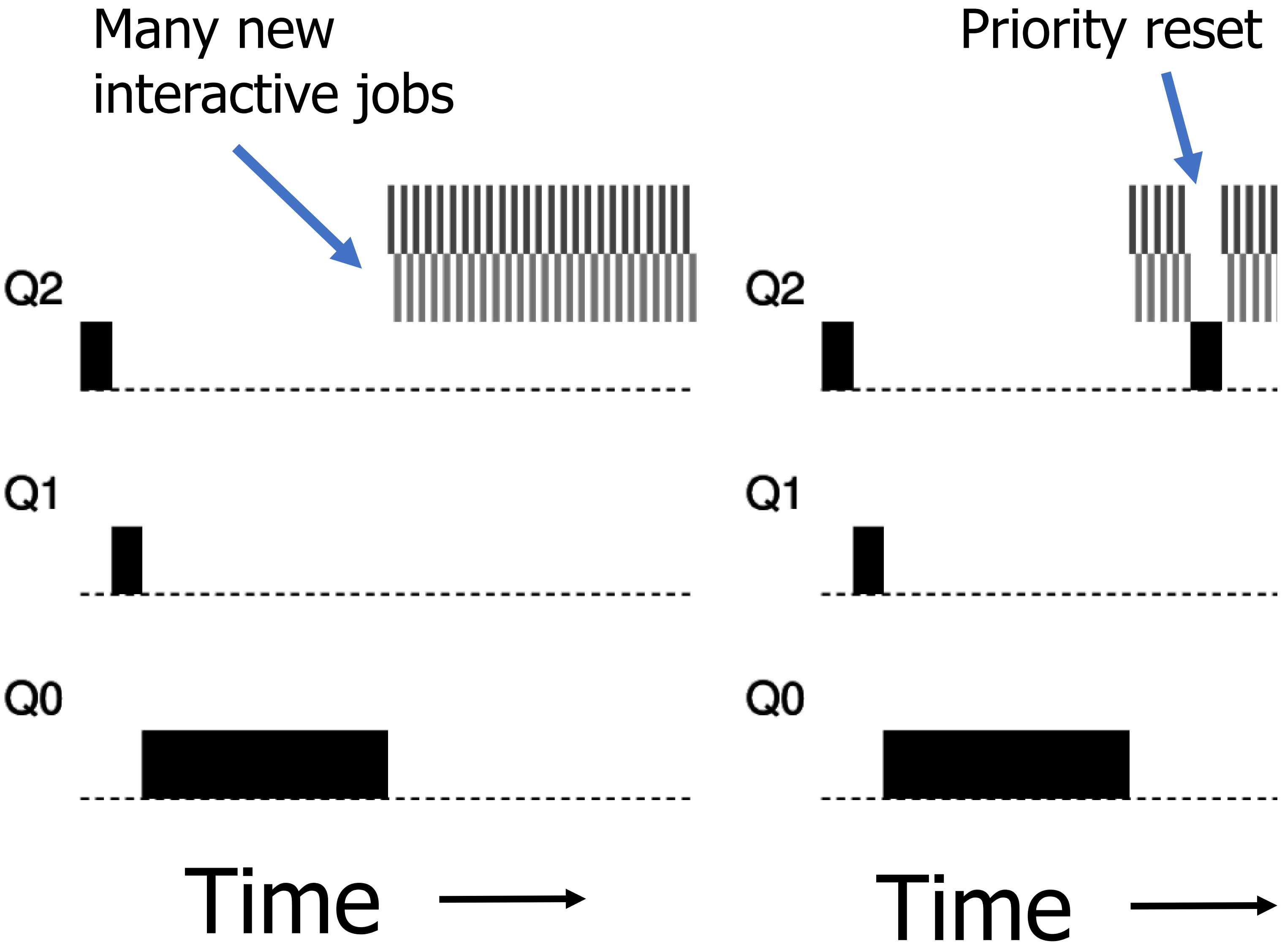
Job's priority drops as it runs

Higher priority jobs run first



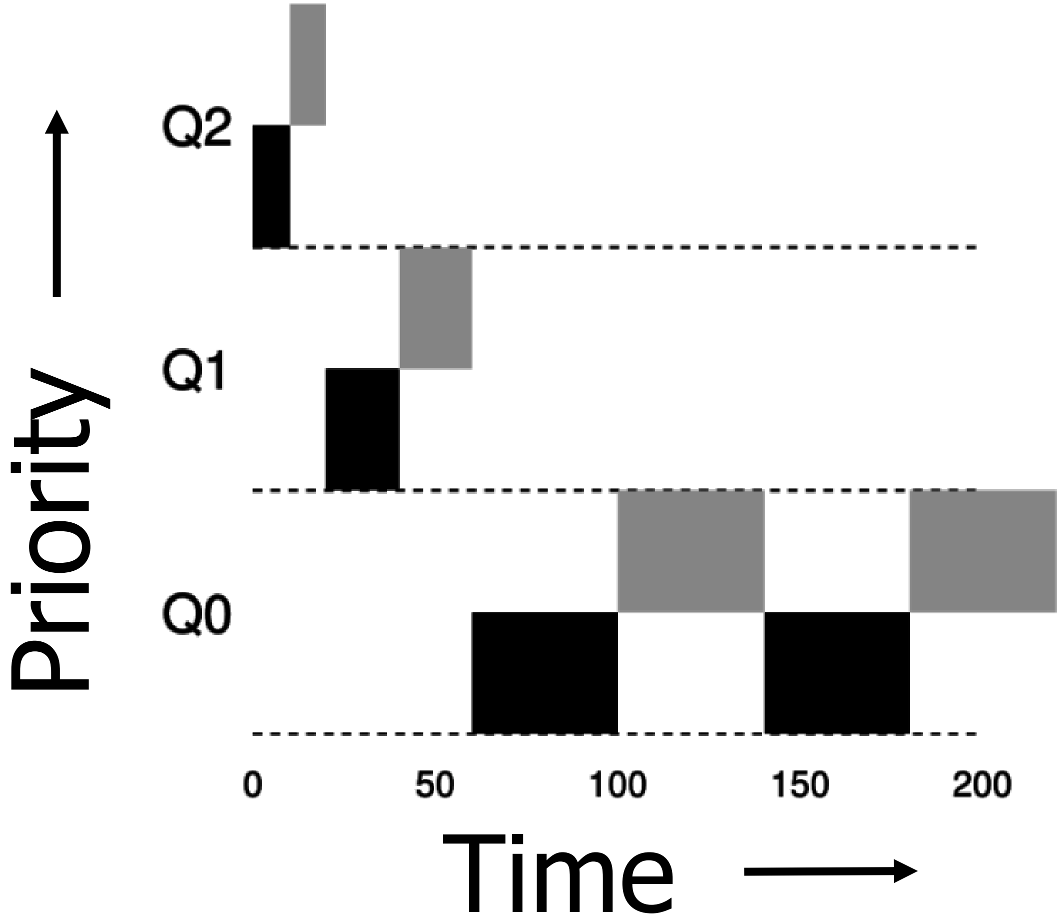
MLFQ avoids starvation with periodic priority reset

- Low priority jobs could starve if there are enough interactive jobs
- MLFQ avoids starvation by periodically resetting priorities



Change timeslices to optimize response and turnaround

- Lower priority jobs are CPU bound, not interactive
  - So we can use longer timeslices to minimize context switches



MLFQ parameters

- Every MLFQ implementation needs to choose a bunch of parameters
  - How many queues/priority levels?
  - When does a job get demoted in priority?
  - How often to reset priority for everything?
  - How large is the timeslice at each priority level?

MLFQ in the wild

- The embedded OS I work on has an MLFQ scheduler!
  - https://github.com/tock/tock/blob/master/kernel/src/scheduler/mlfq.rs

- How many queues/priority levels?
  - Three

- When does a job get demoted in priority?
  - If it ever uses its whole timeslice without blocking

- How often to reset priority for everything?
  - Every five seconds

- How large is the timeslice at each priority level?
  - 10 ms, 20 ms, 50 ms

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

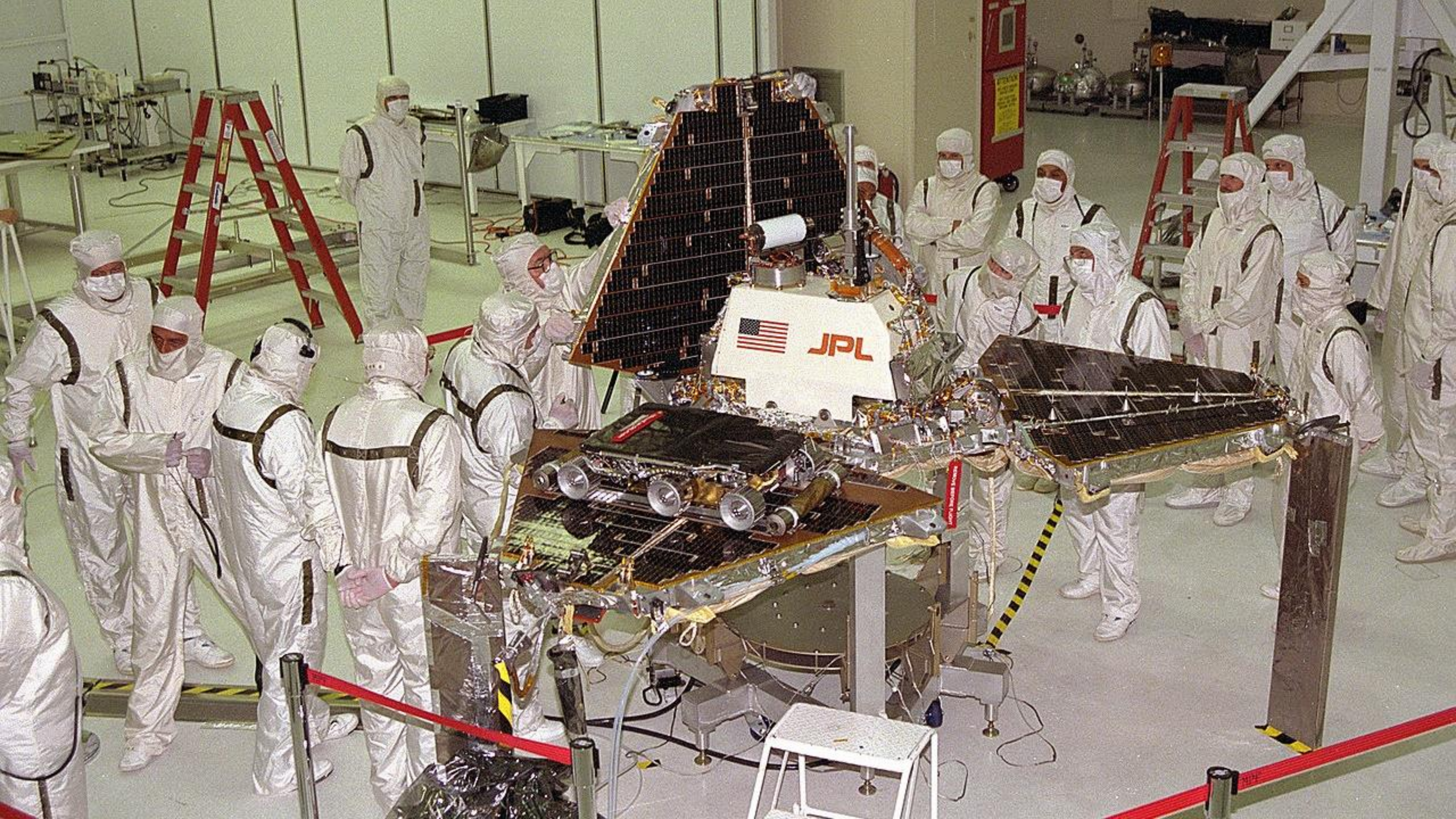
Normal OSes don’t cut it for all use cases

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

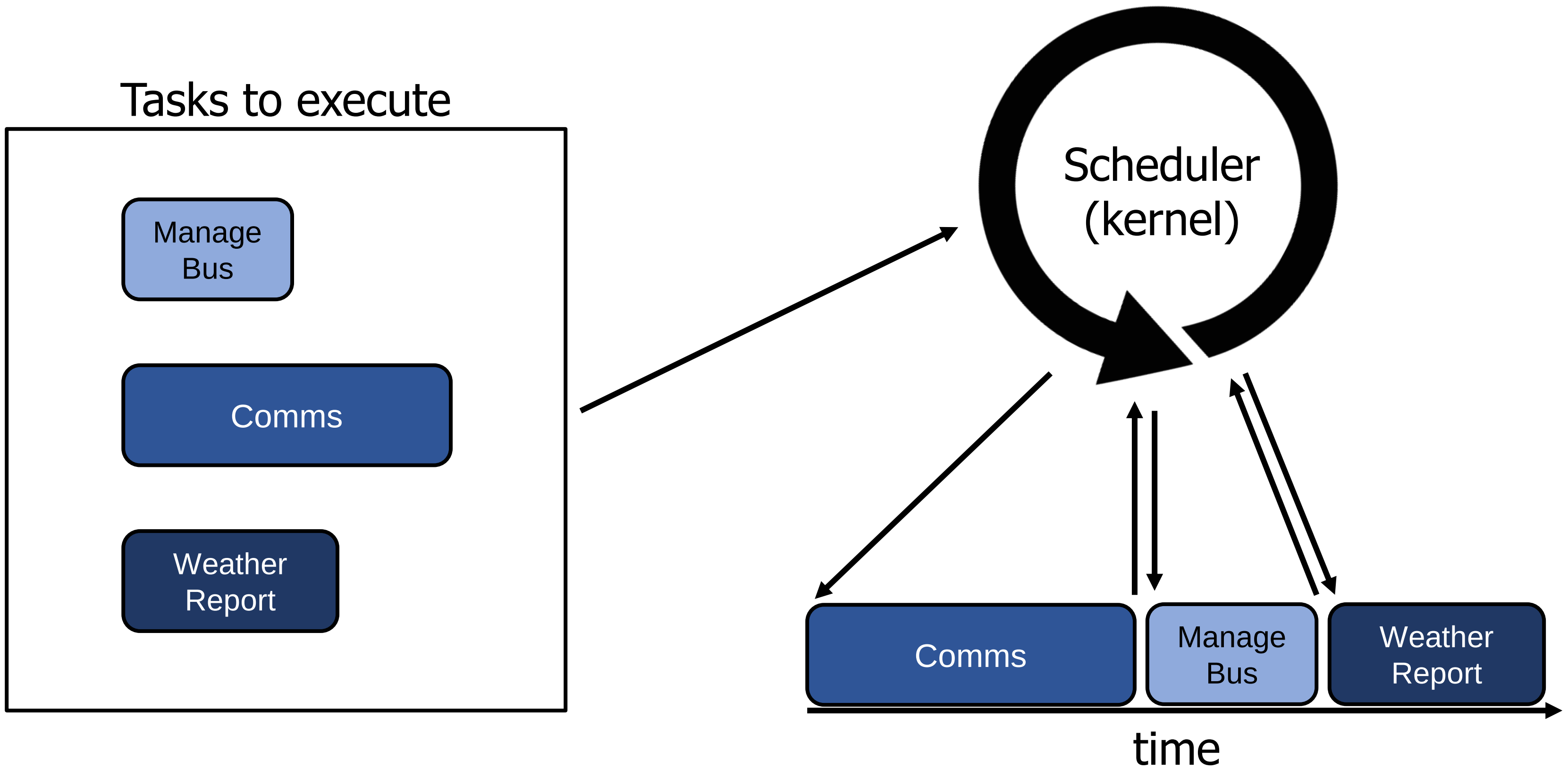
- In each of these scenarios
  - Computer failures are unacceptable
  - Humans can’t intervene to resolve issues
  - We’re going to need a computer system with performance *guarantees*

Example: Pathfinder

Radiation-hardened IBM CPU


JPL

Pathfinder had periodic tasks that must be executed



Tasks to execute

Manage Bus

Comms

Weather Report

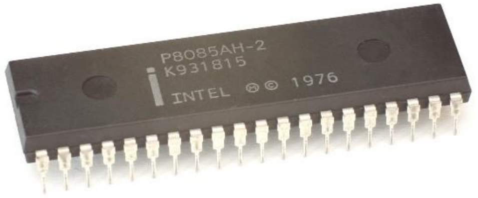
Scheduler (kernel)

Comms

Manage Bus

Weather Report

time



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

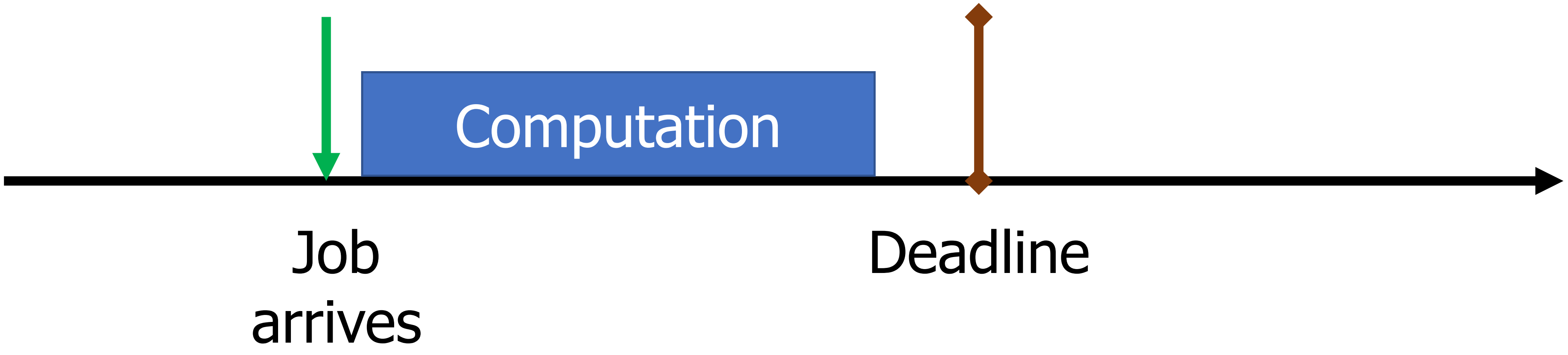
- Hard real-time:
  - Meet **all deadlines**
    - Otherwise decline to accept the job
  - Ideally: determine in advance if deadlines will be met

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

Real-time jobs

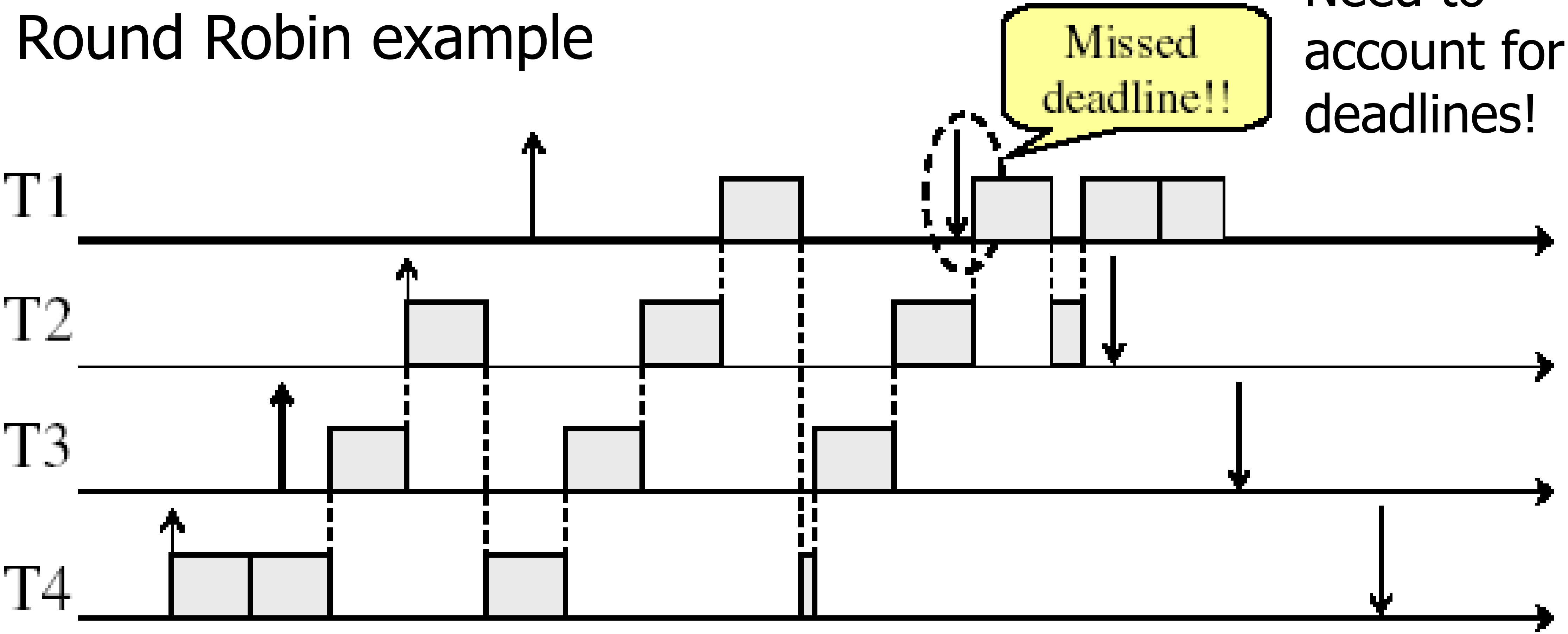
- Preemptable jobs with known deadlines (D) and computation (C)
  - Computation duration here are the worst-case execution times

  - Computation MUST complete before deadline and start after arrival
    - Can happen anywhere between those boundaries though



Prior scheduling policies don’t apply here

Round Robin example



Need to account for deadlines!

Types of real-time jobs

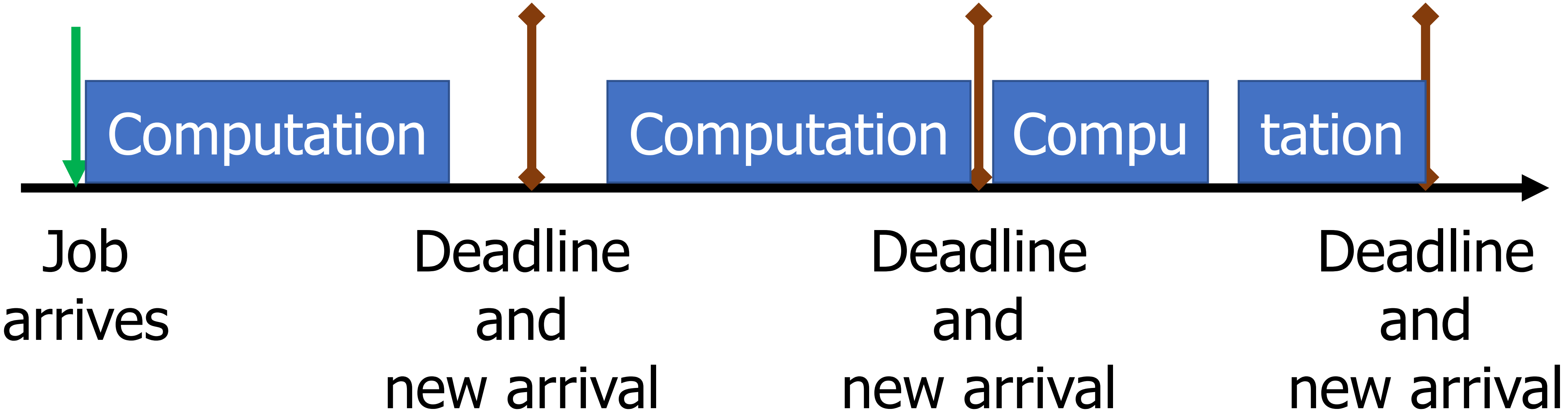
- Aperiodic
  - Jobs we are already accustomed to
  - Unpredictable start times, no deadlines (not real-time)

- Sporadic
  - Unpredictable start time, has a deadline
  - Must decide feasibility at runtime and either accept or reject job

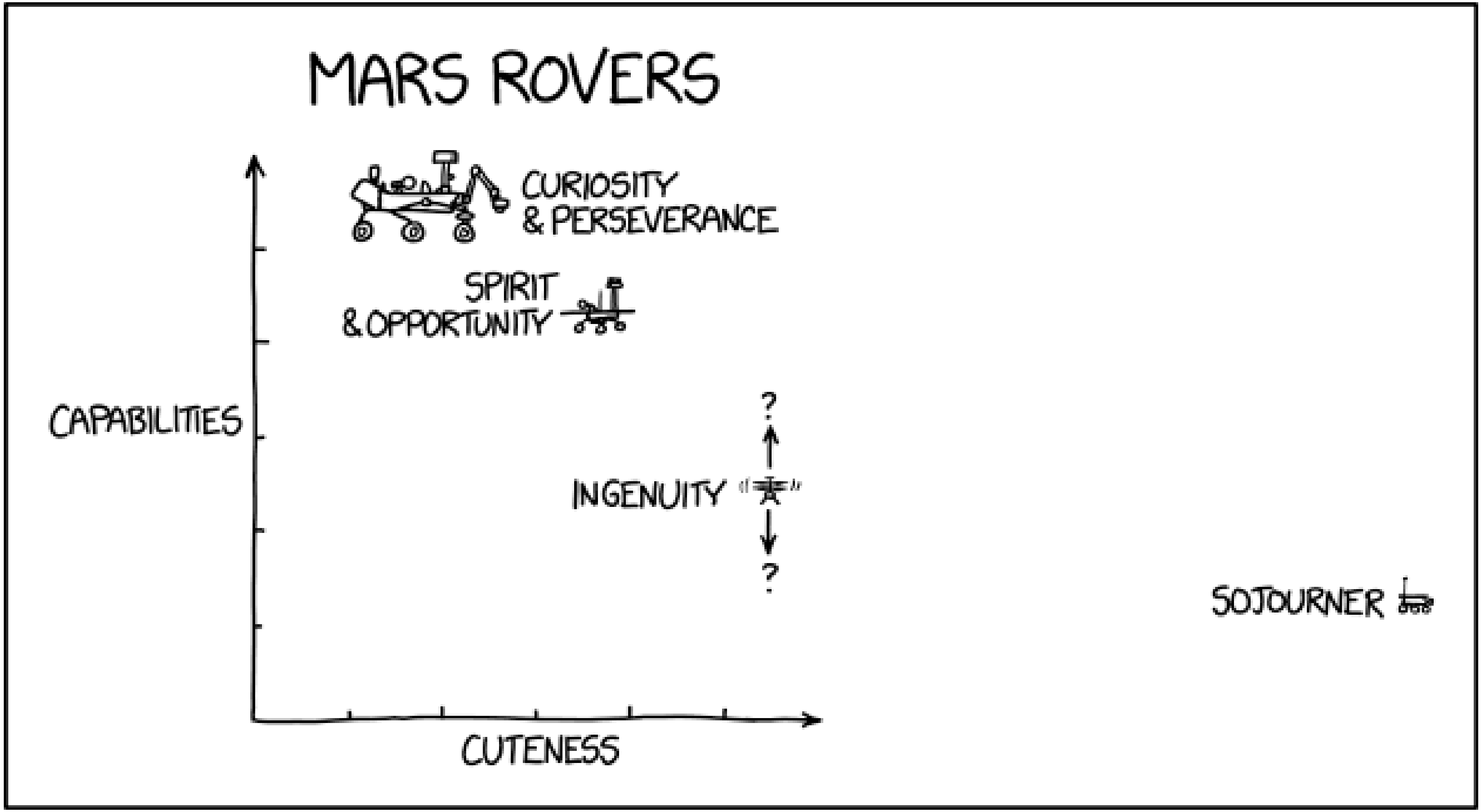
- Periodic (we'll focus on these)
  - Recurs at a certain time interval
  - Deadline for completion is before the start of the next time interval
    - i.e. deadline equals the period
  - Can decide *feasibility* of schedule at compile-time

Periodic real-time jobs

- Repeat at their deadline
  - New work cannot be started until the deadline
  - Work can take place anytime between deadlines
    - But MUST finish before the deadline hits



Break + xkcd



https://xkcd.com/2433/

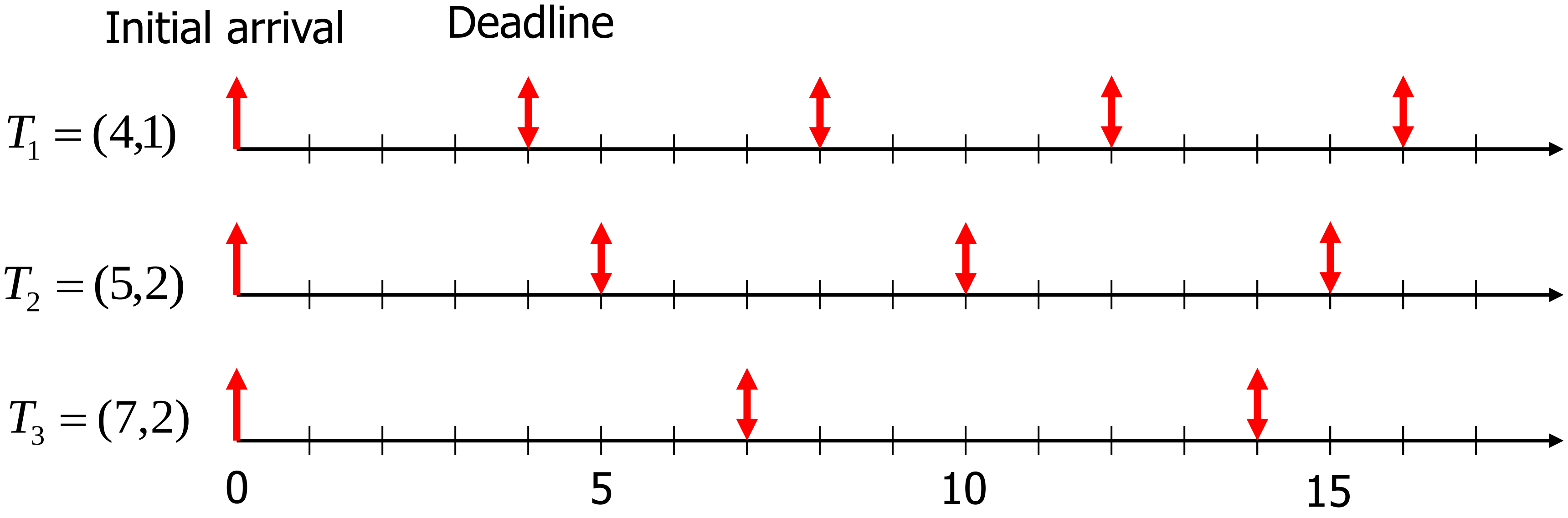
Outline

- **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

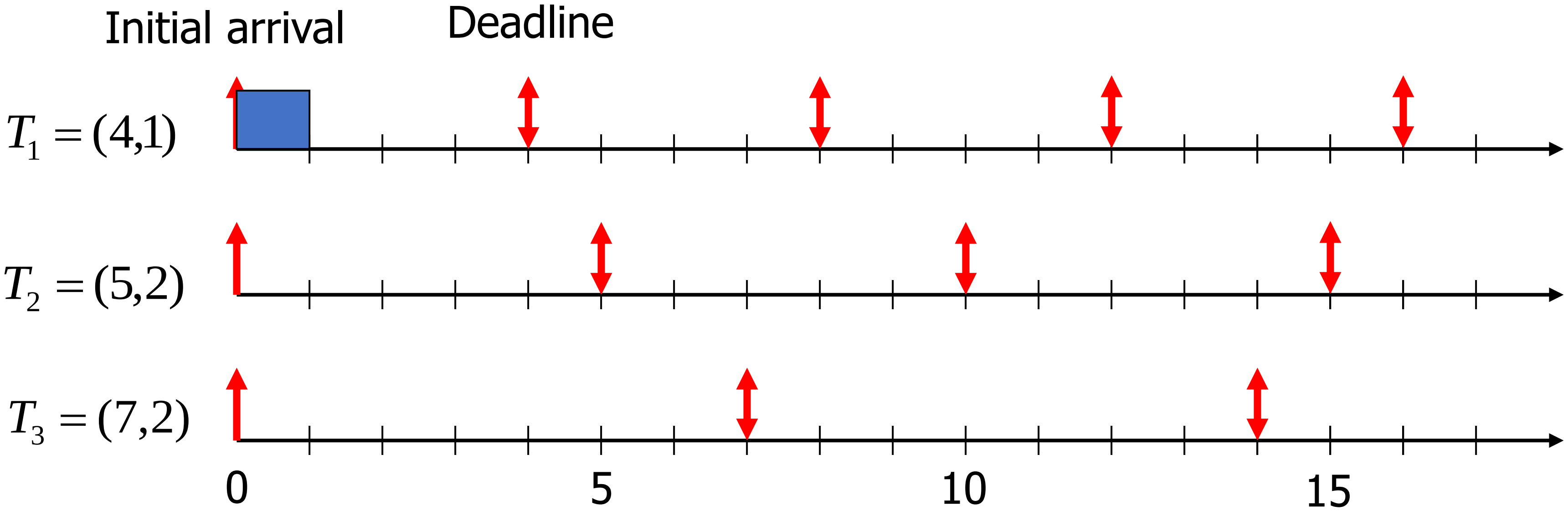
Earliest Deadline First (EDF) Scheduling

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



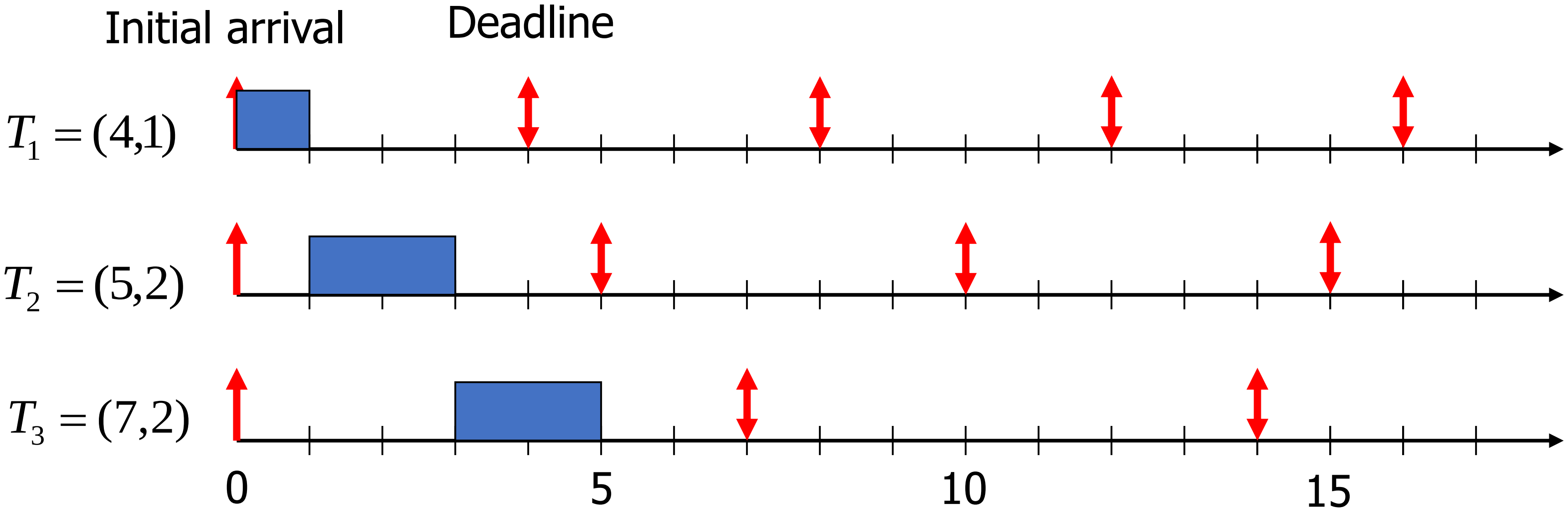
Earliest Deadline First (EDF) Scheduling

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



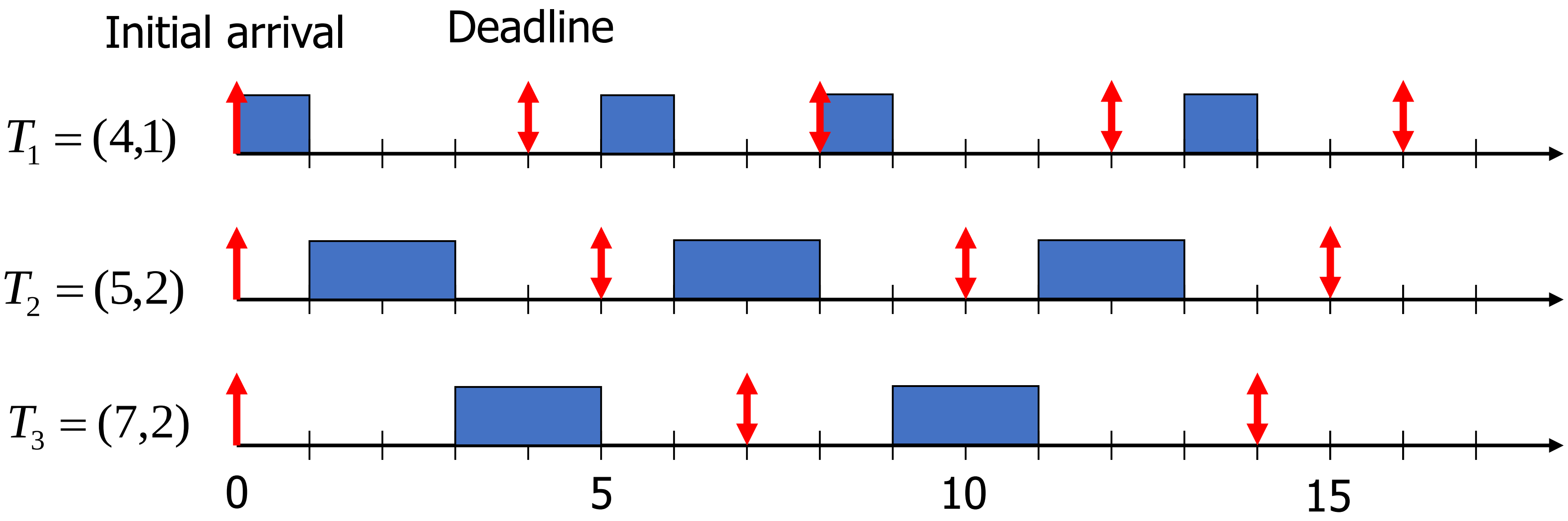
Earliest Deadline First (EDF) Scheduling

- 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%
  - All deadlines **will** be met

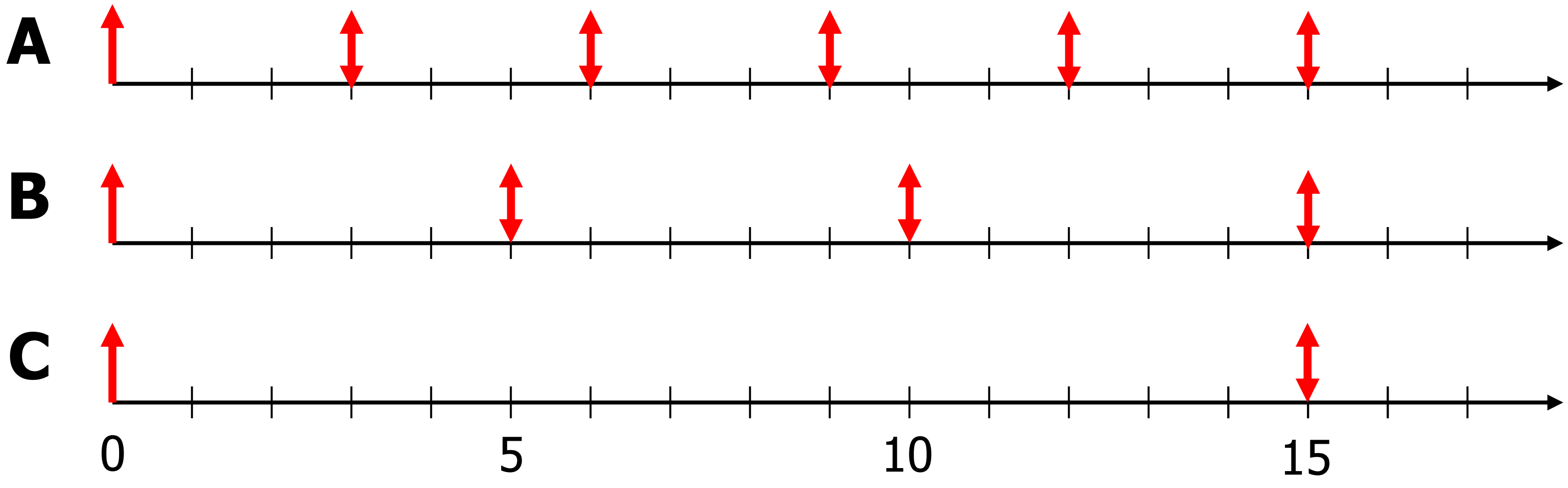
- For *n* tasks with computation time *C* and deadline (period) *D*
  - A feasible schedule exists if **utilization** is less than or equal to one:

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

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

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

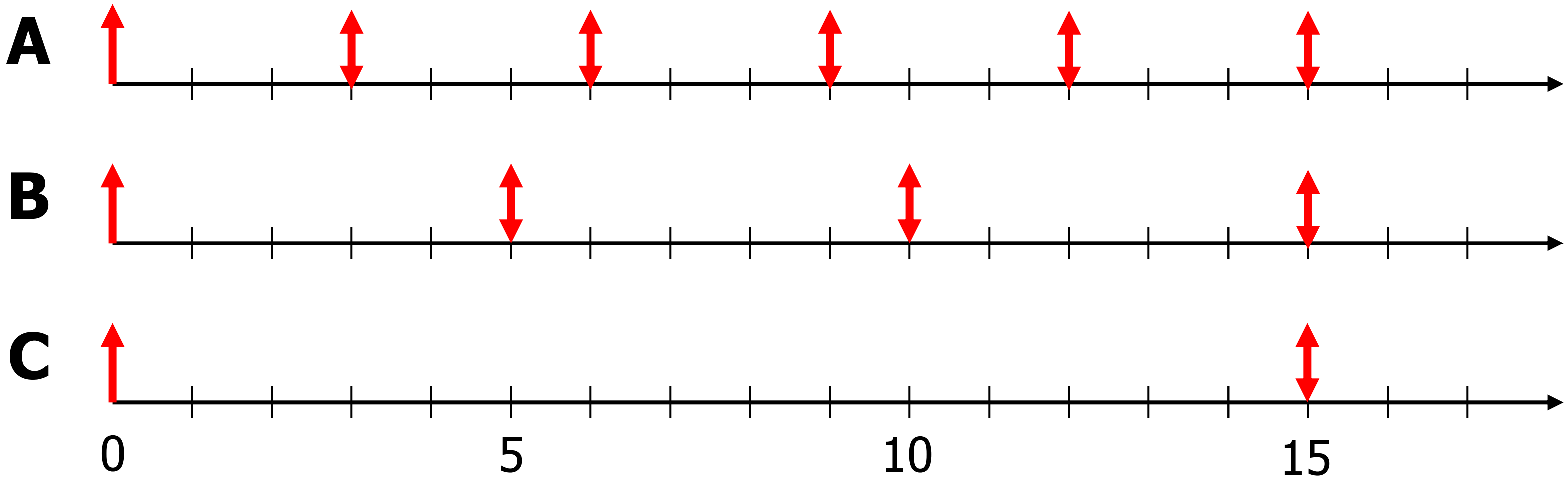


Check your understanding

- 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

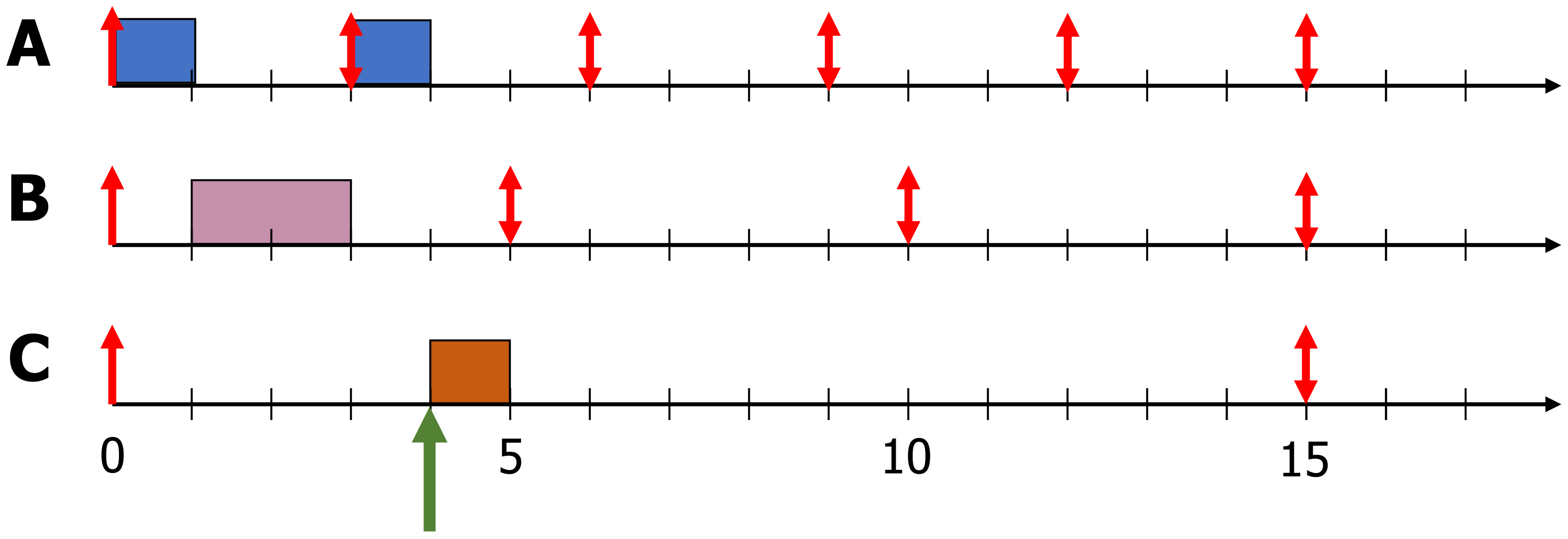


Check your understanding

- 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



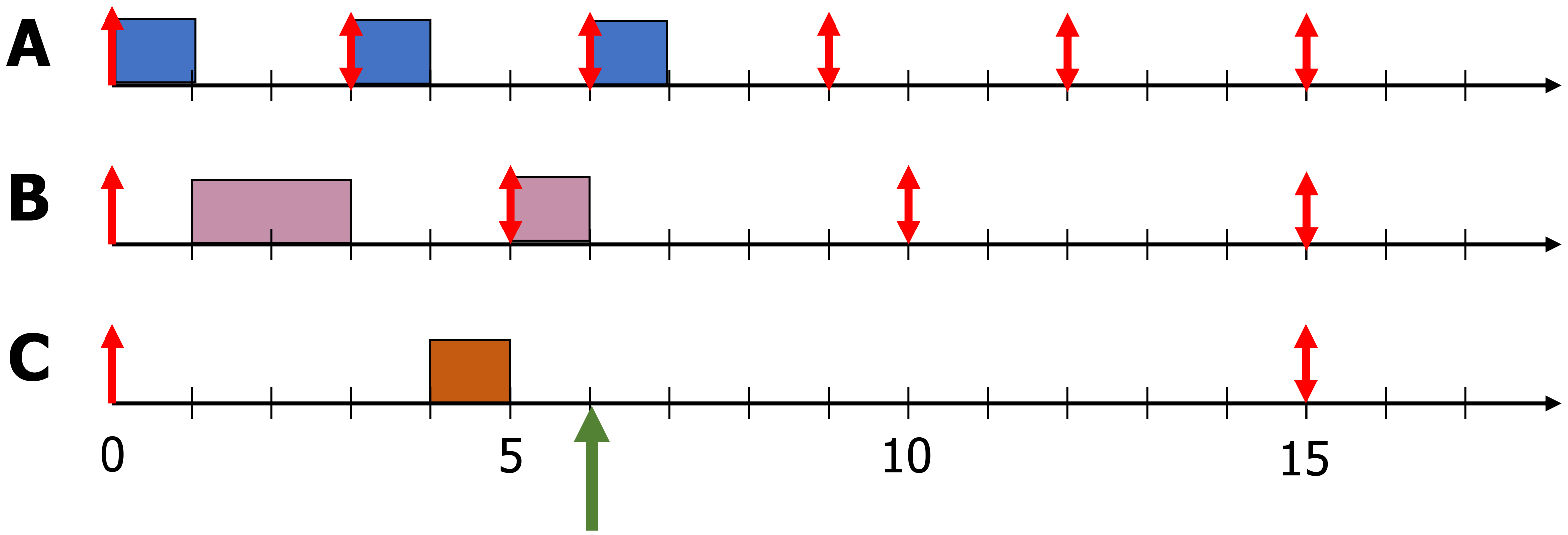
Can’t start a job *before* its period

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



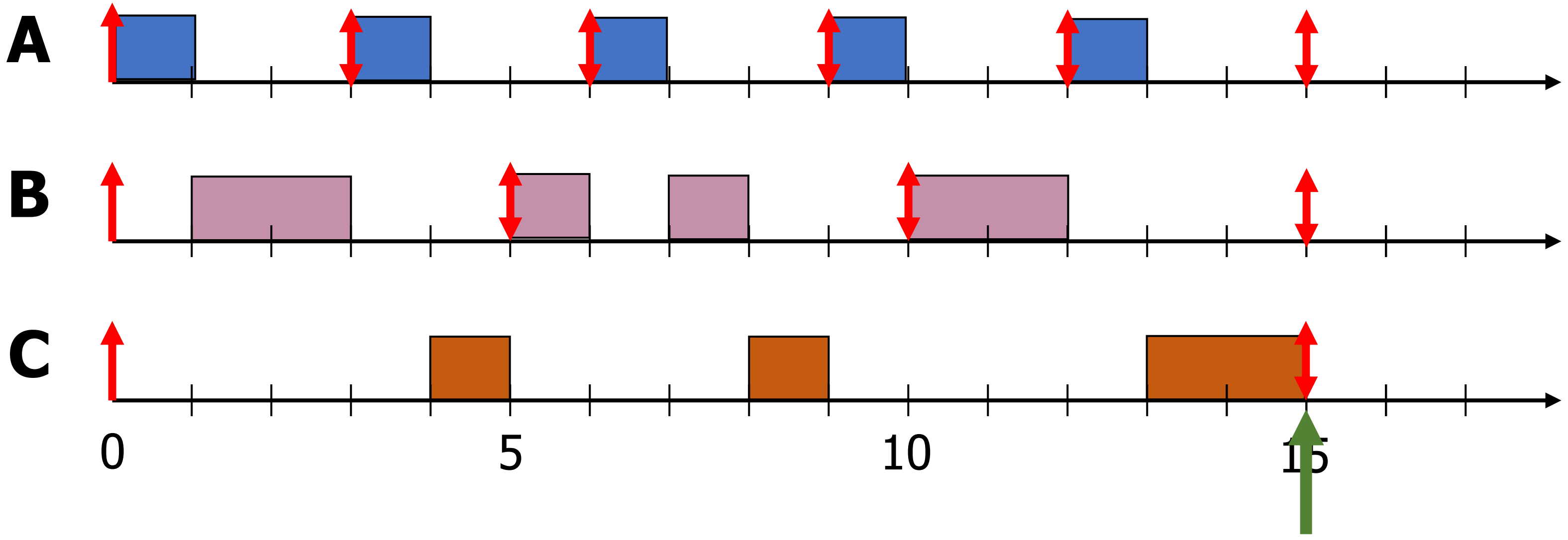
Earliest deadline changes, preempting Job B

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

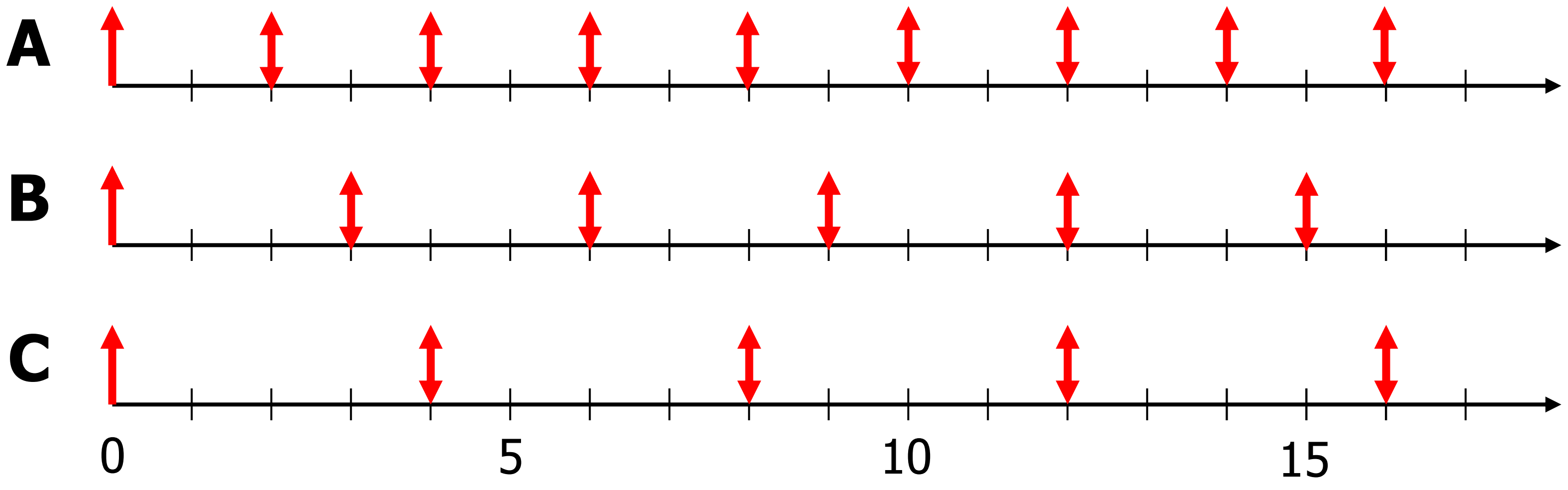


Schedule repeats at least common multiple

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

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

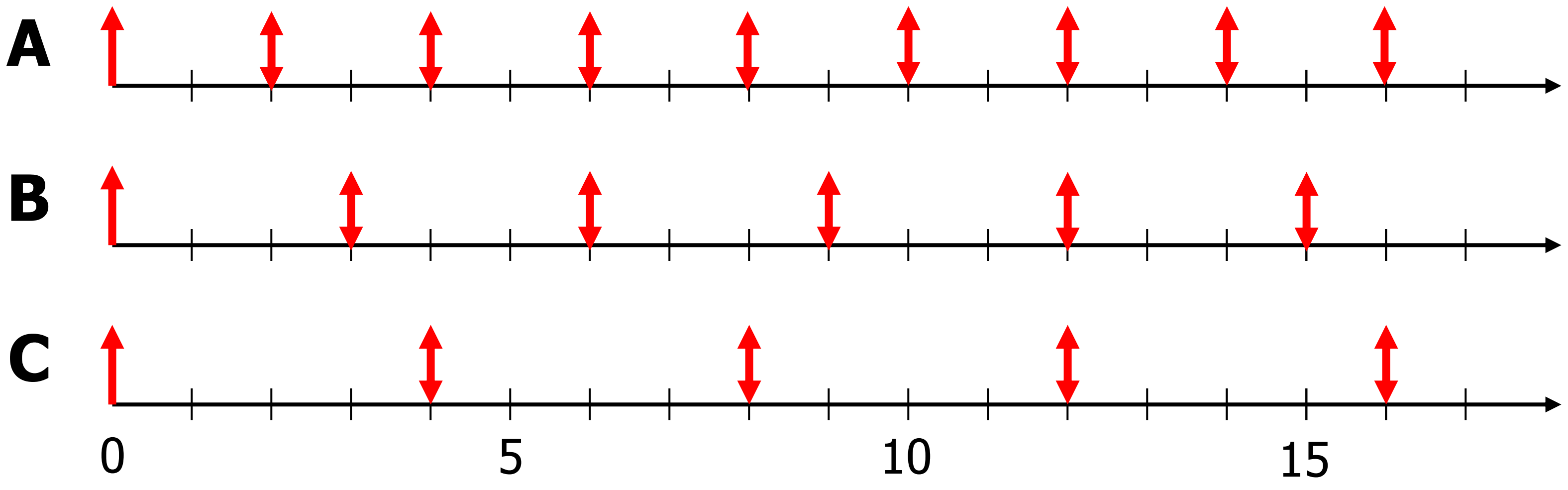


Check your understanding

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

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

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

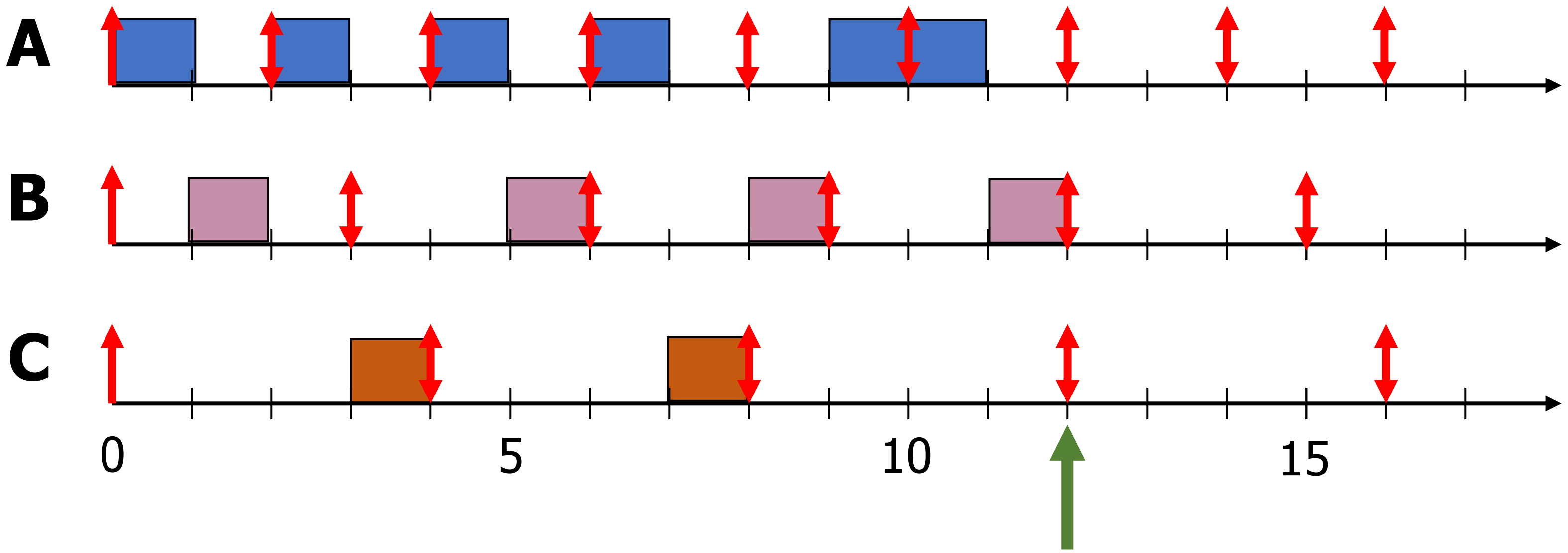


Check your understanding

- 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



Missed deadline!

Break + Thinking

- Where do the job deadlines come from? Provide an example.

Break + Thinking

- Where do the job deadlines come from? Provide an example.
  - Real-world constraints!
  - Autonomous vehicle:
    - “If I don’t finish the detection algorithm by time N,
      then I will no longer be able to stop in time to avoid what it detects.”
    - In this example, deadline might vary with velocity,
      or maybe we just choose a deadline based on fastest velocity.

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

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

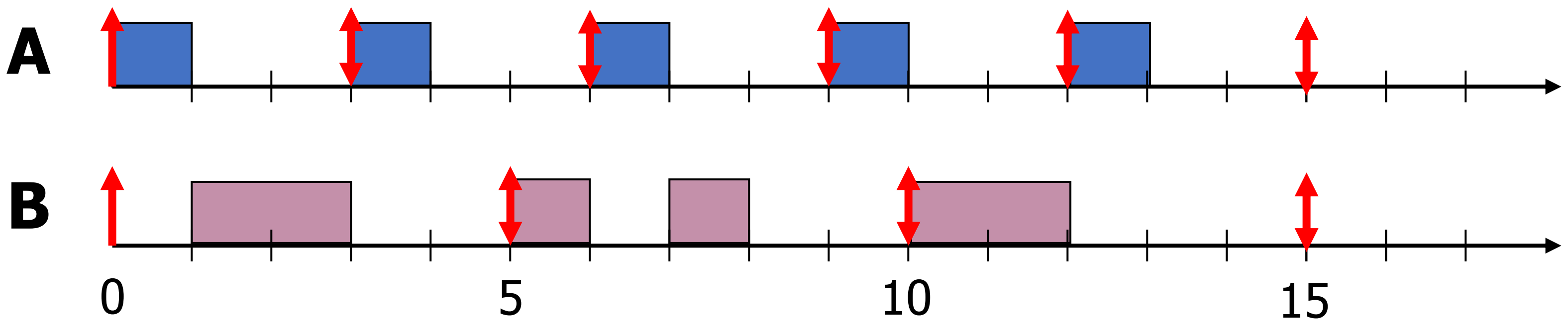
Rate Monotonic Scheduling (RMS)

- 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
  - There could be dynamic-priority systems that beat it
  - But they would be more complicated and take more cycles to run

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

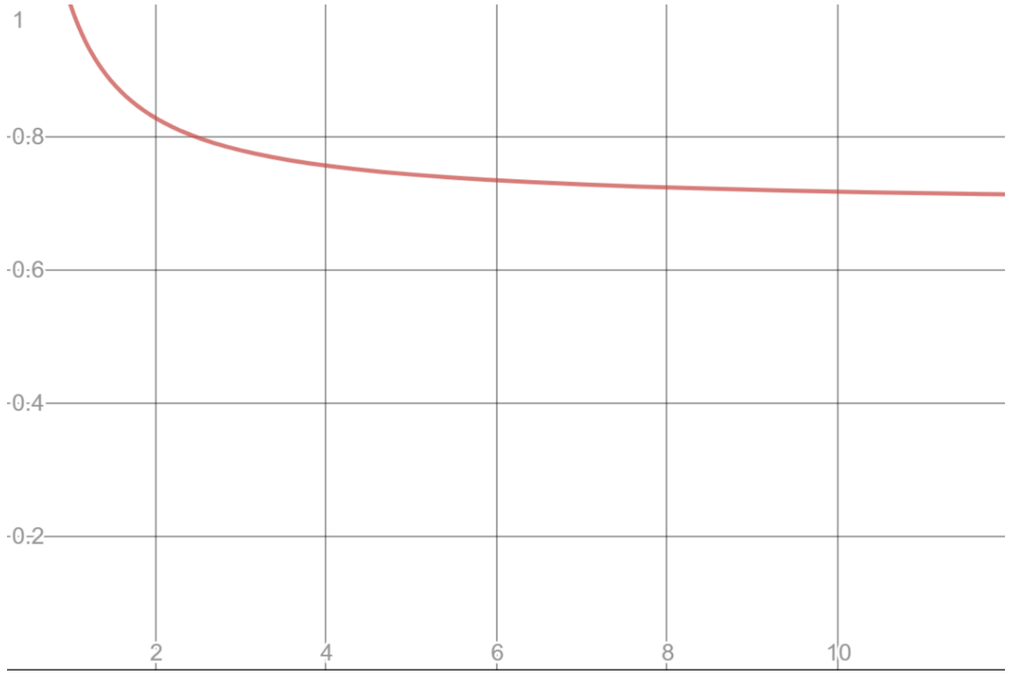
- Schedulability is more complicated for RMS unfortunately
  - For a workload of ***n*** jobs with computation time ***C*** and period ***D***

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

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

  …
- U(∞) = 0.693

Lower Bound on schedulability



RMS schedulability test is conservative

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

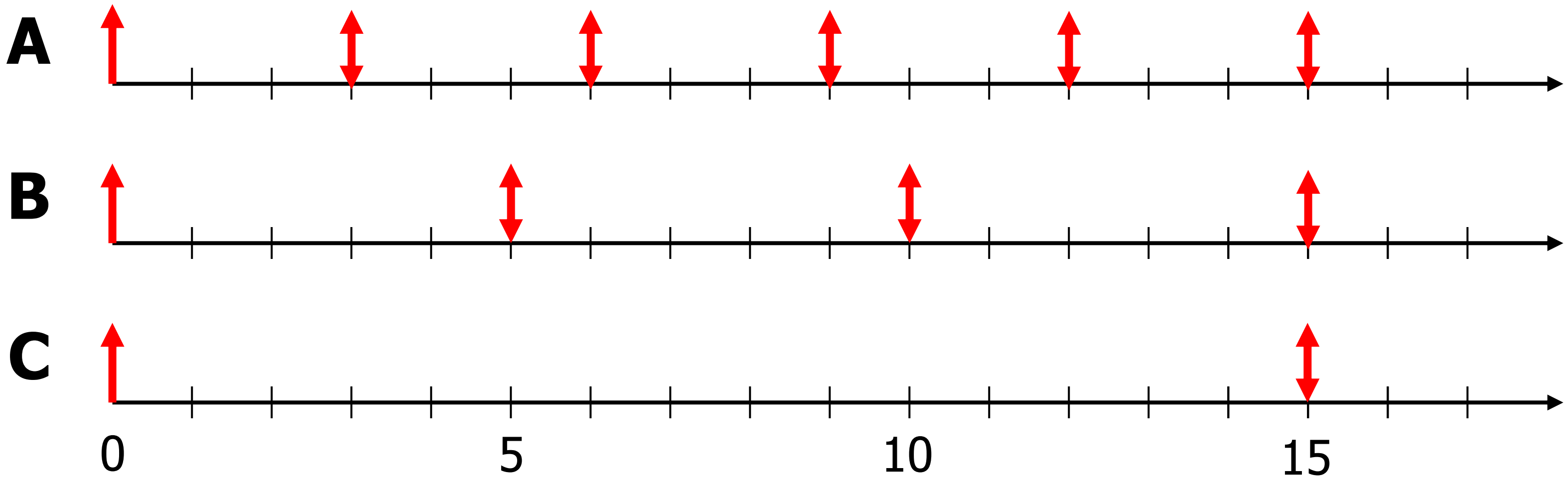
- $0 \leq U \leq n * (2^{\frac{1}{n}} - 1)$
  - Schedulable! (so less than 69% is always schedulable)

- $n * (2^{\frac{1}{n}} - 1) < U \leq 1$
  - 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

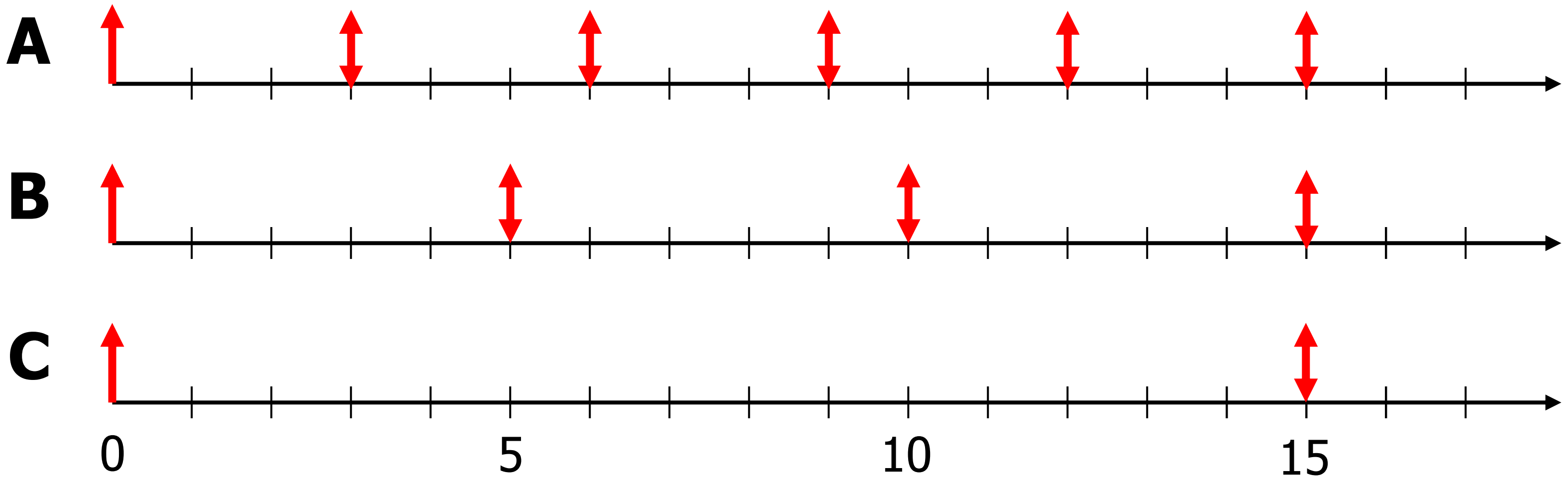


Check your understanding

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

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

U = 1
Maybe schedulable!

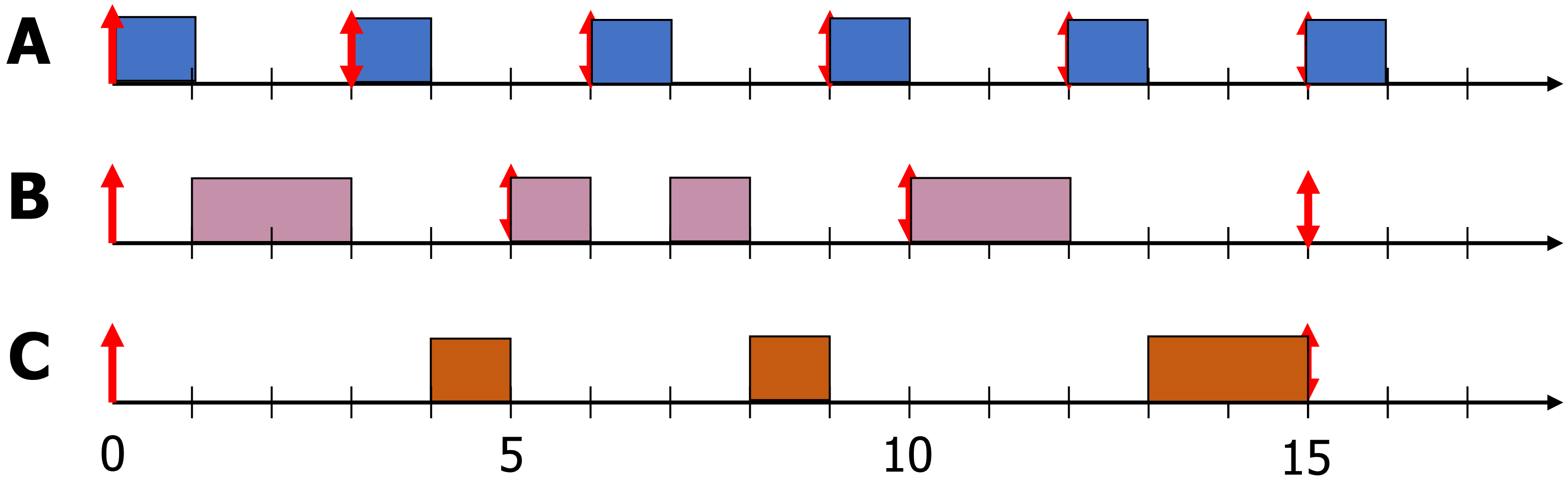


Check your understanding

- 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

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

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

Break + Open Question

- How would you handle sporadic jobs in these systems?
    - Unpredictable start time, has a deadline, not repeated

Break + Open Question

- How would you handle sporadic jobs in these systems?
  - Unpredictable start time, has a deadline, not repeated

- Must decide feasibility at runtime and either accept or reject job
  - Calculate new Utilization accounting for the additional job
  - Determine whether the schedule will definitely (or maybe) work
  - Schedule or reject the job

  - If scheduled, works just like any other job
    - Either EDF based on deadline of the job
    - Or given an RMS priority, based on period (duration)

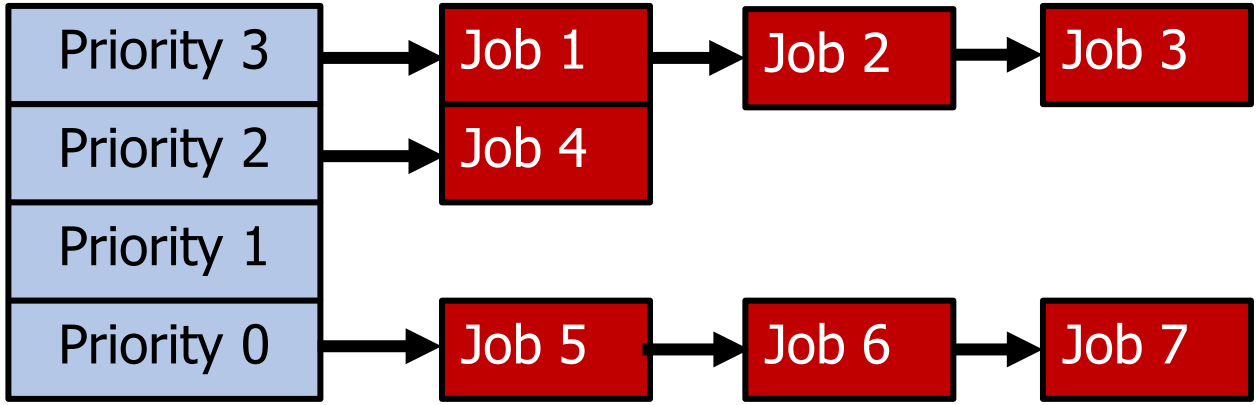
Outline

- 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 jobs higher priority than long calculations
  - 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)

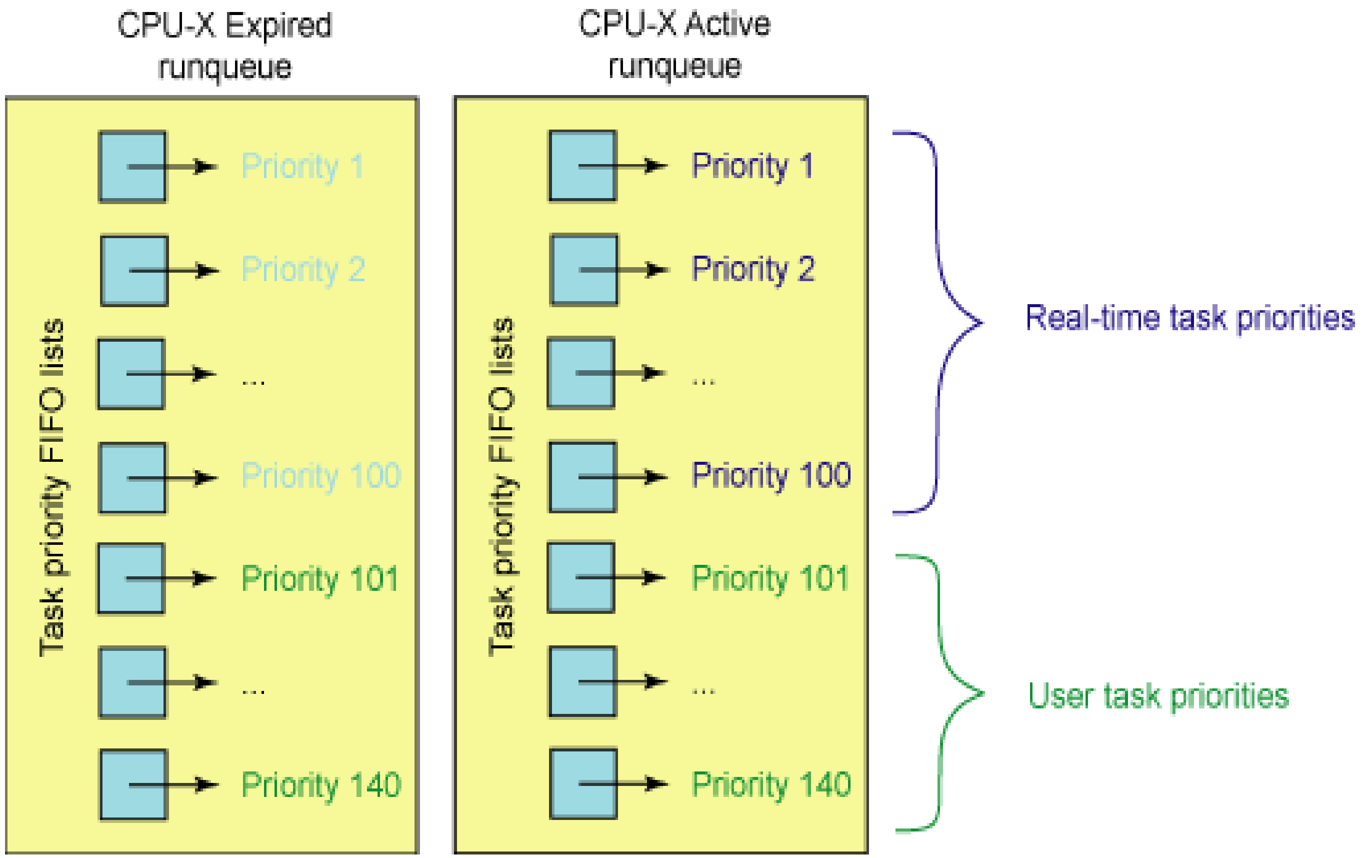
- Goals
  - Keep the runtime of the scheduler itself short
    - Avoid O(n) algorithms
    - Instead, only adjust a single job when it is swapped
  - Predictable algorithm
  - Identify interactive versus noninteractive processes with heuristics
    - Processes with long average sleep time get a priority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

- Note my machines right now:
  - Ubuntu VM: 332 processes (867 threads)
  - Windows: 224 processes (2591 threads)
  - MacOS: 430 processes (2249 threads)
  - **Major concern:** many processes mean O(n) could be long…

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
- The **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

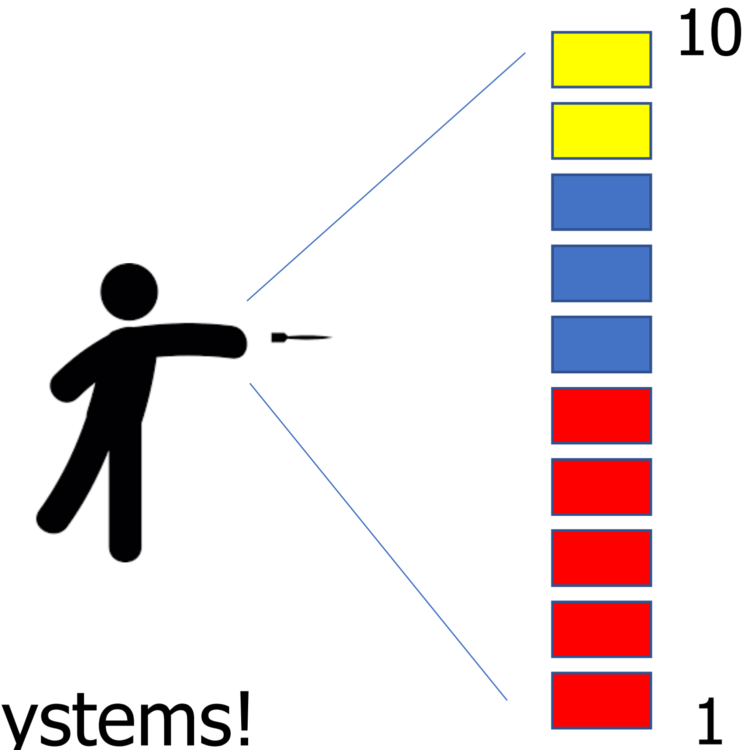
- Instead, we can share the CPU proportionally
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)

First attempt: 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:

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

- Definitely not suitable for real-time systems!



Better idea: stride scheduling

- Same idea, but remove the random element
- Give each job a stride number inversely proportional to priority
  - Priority: A=100, B=50, C=10
  - Stride: A=1, B=2, C=10

$$stride = \frac{N}{priority}$$

Where *N* is some arbitrary large number
This example: 100

- 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
  - But starvation is no longer possible

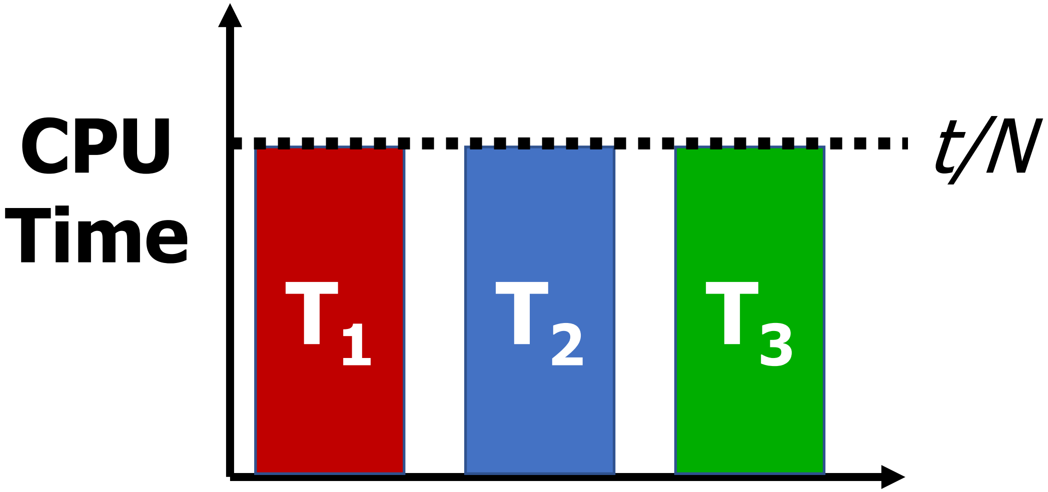
Stride scheduling example

- Workload
  - Priority: A=100, B=50, C=10
  - Stride: A=1, B=2, C=10

Step	Dynamic Priority (a.k.a. Pass)			Result
	A	**B**	**C**	
1	0	0	0	A
2	1	0	0	B
3	1	2	0	C
4	1	2	10	A
5	2	2	10	A
6	3	2	10	B
7	3	4	10	A

Proportional-share scheduling is impossible instantaneously

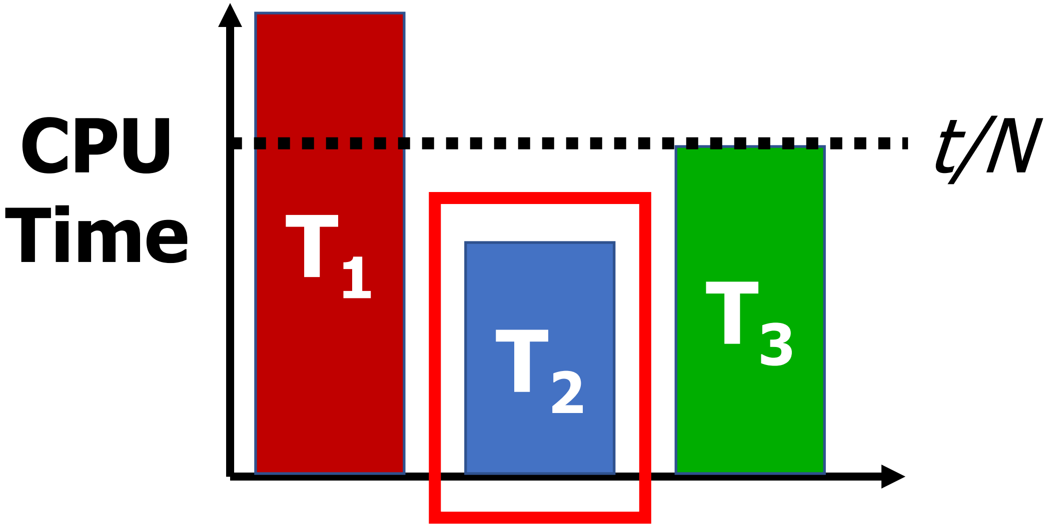
At *any* time *t*
we want to observe:



- 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

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 = max(Target_latency/N, minimum_length)

CFS priorities are applied as “virtual runtime”

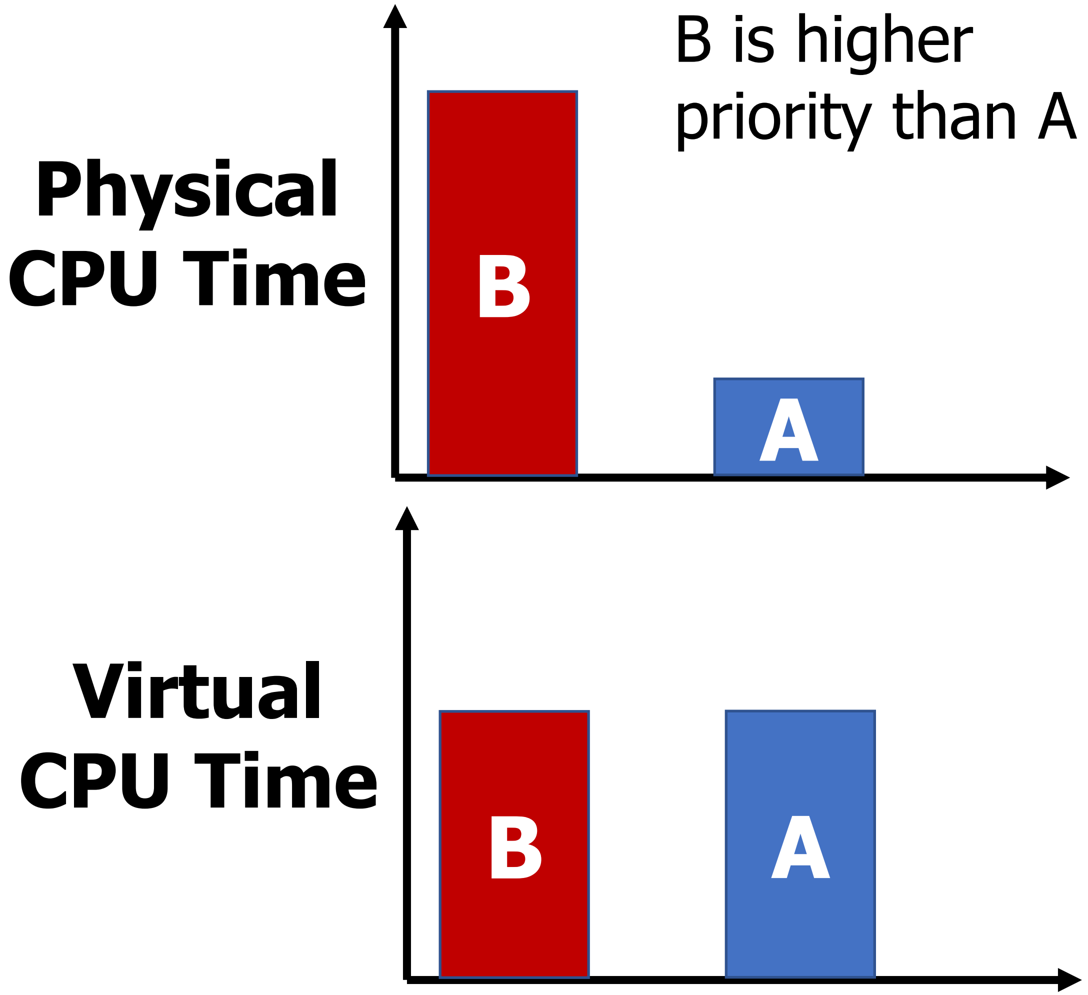
- Virtual runtime doesn’t have to match wall time
- Create a conversion from actual runtime to virtual runtime
  - **High priority jobs:**

    1 second realtime
    -> 0.5 seconds virtual time

  - **Low priority jobs:**

    1 second realtime
    -> 2 seconds virtual time

- Scheduler makes decisions solely based on equal virtual runtime



Multicore scheduling

- *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
  - Each core can make its own scheduling decisions
  - Can steal work from other cores, if nothing to do

Active work in scheduling

- Getting scheduling right on multicore can be difficult
    - No way to know whether a process will be more I/O or CPU bound in the future
    - Want to keep threads on the same core, but also not waste cores

- In 2016, researchers found issues in Linux scheduler implementation that lead to 13%+ slowdown in jobs
    - https://blog.acolyer.org/2016/04/26/the-linux-scheduler-a-decade-of-wasted-cores/

- Another metric: energy use

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
Fair CPU Time Usage	Linux CFS
Meeting Deadlines	EDF or RMS

Outline

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

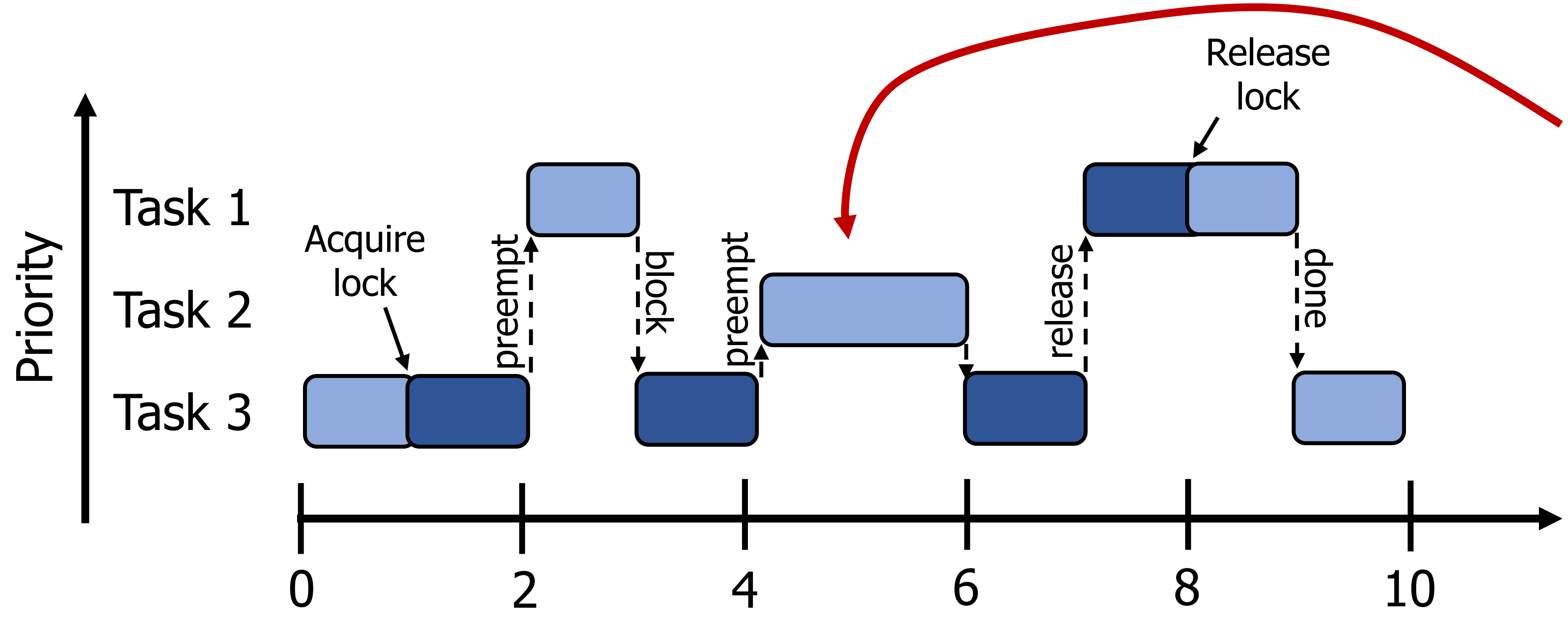
- Modern Operating Systems
  - Linux O(1) scheduler
  - Lottery and Stride scheduling
  - Linux Completely Fair Scheduler

- Bonus

A problem with priority schedulers: priority inversion

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

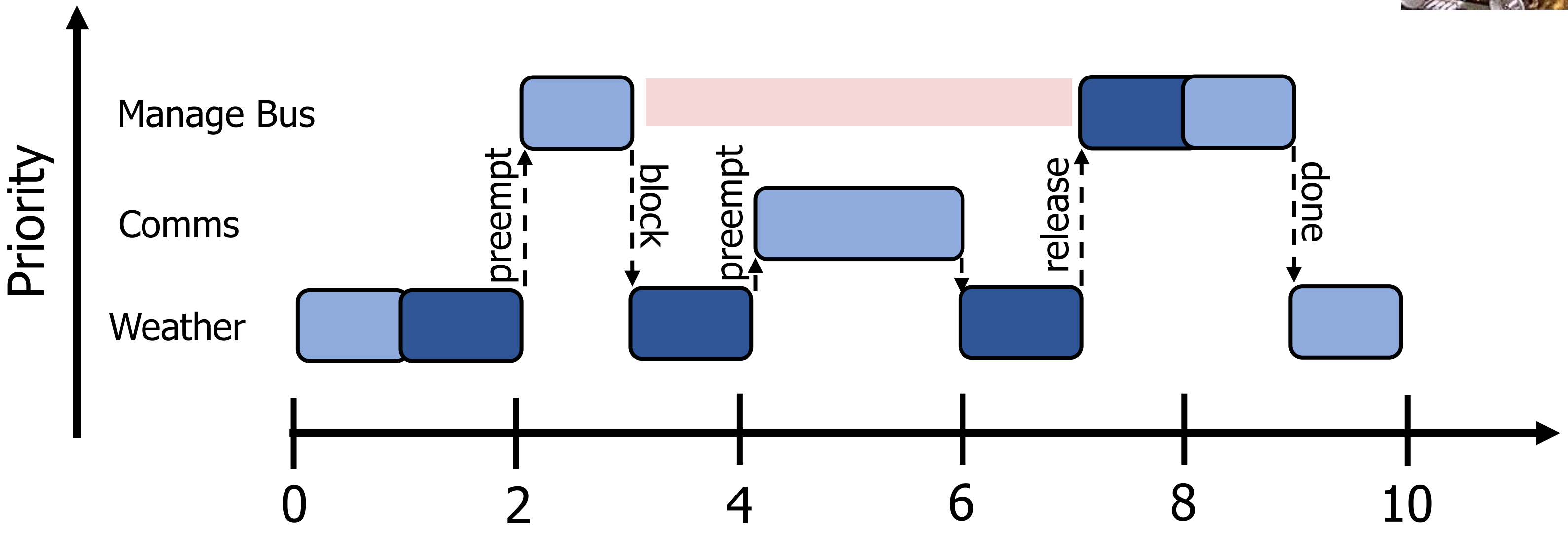
- Imagine Task 1 and Task 3 both need to share a lock



Task 1 is waiting on Task 2!!

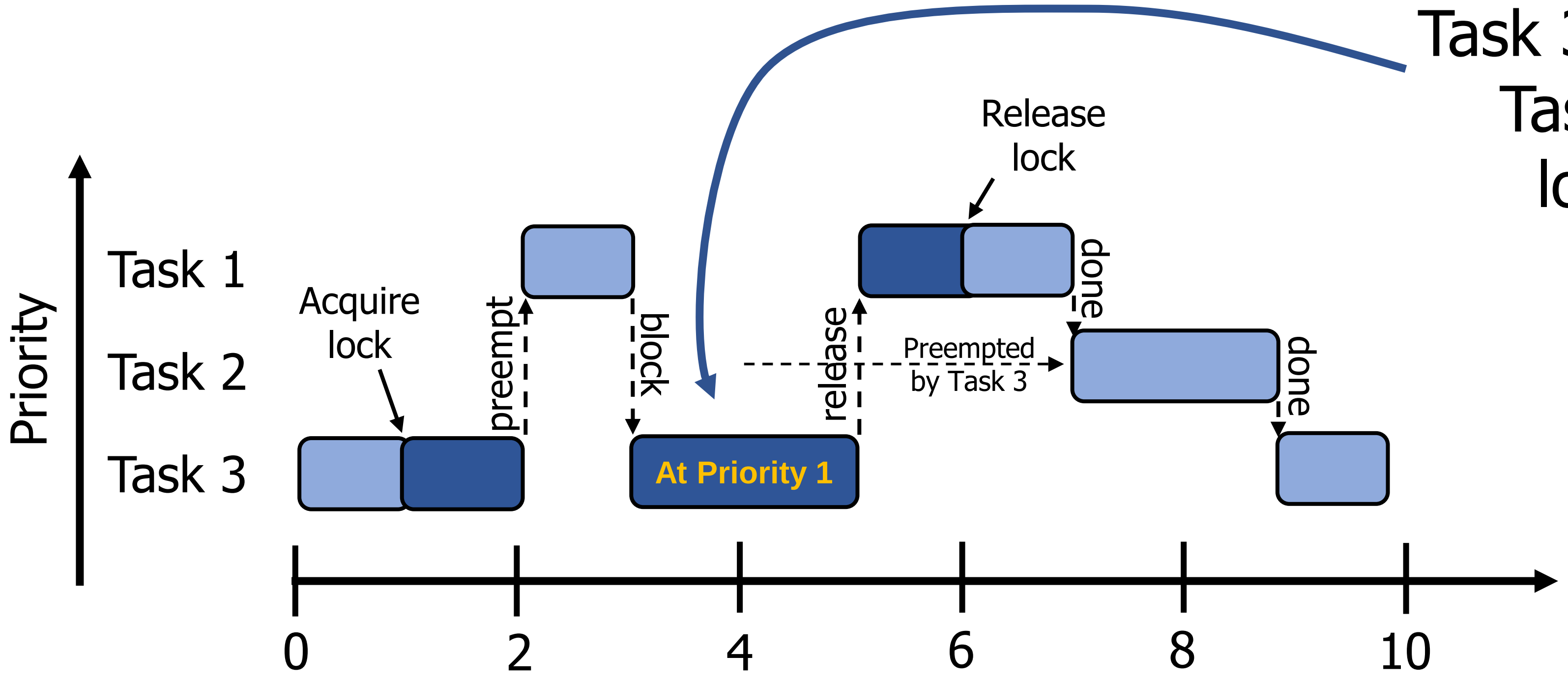
Priority inversion occurred on Pathfinder!

- 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



Task 3 **inherits** priority of Task 1 while holding lock Task 1 needs