Lecture 08: Advanced Scheduling

CS343 – Operating Systems

Branden Ghena – Fall 2024

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

Administrivia

- PC Lab due today
  - See Piazza post with reminder of the submission steps

- Midterm Exam next week Tuesday
  - Be sure to get here early. We'll start at 12:30 sharp

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

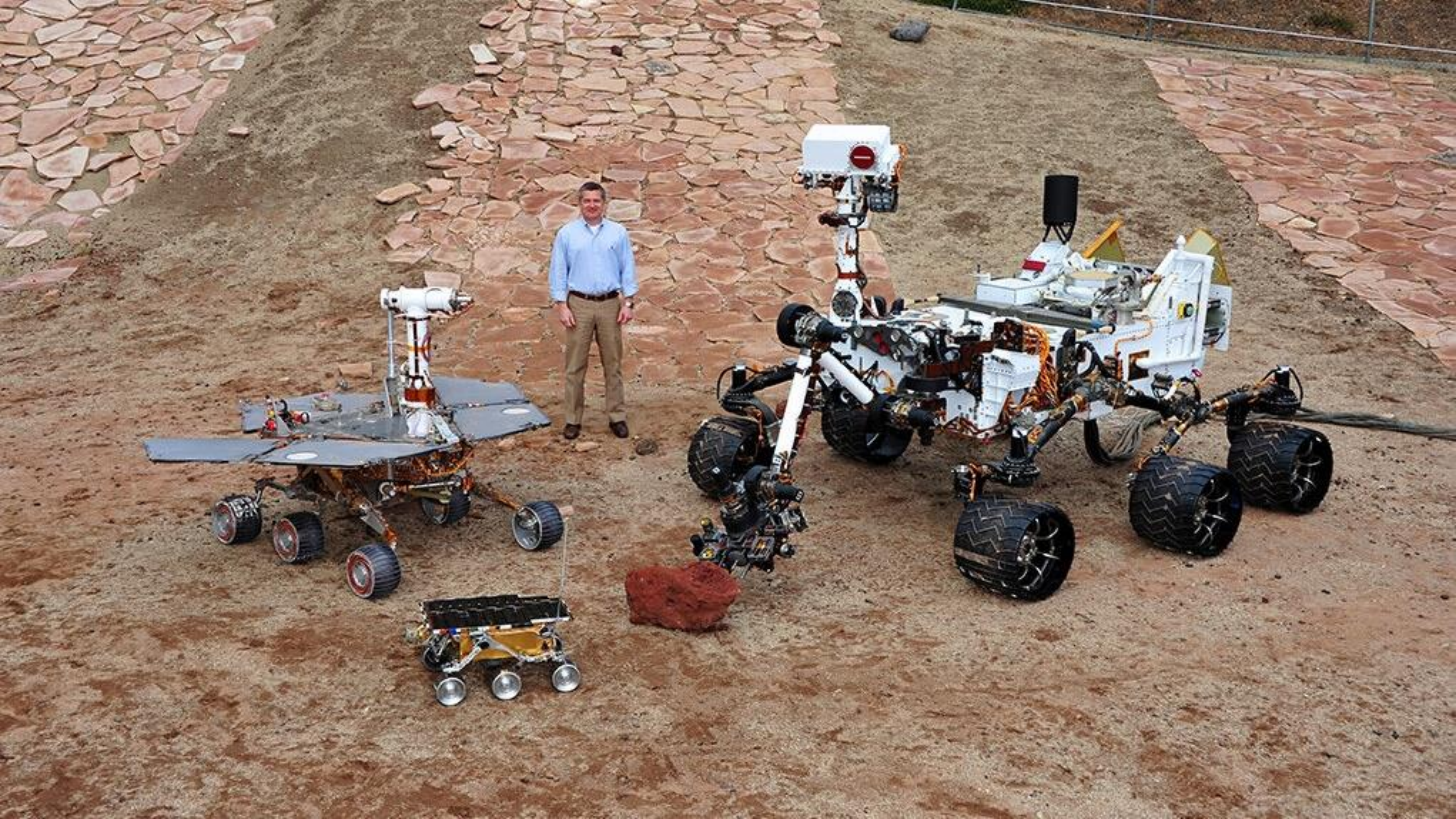
Normal OSes don’t cut it for all use cases

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

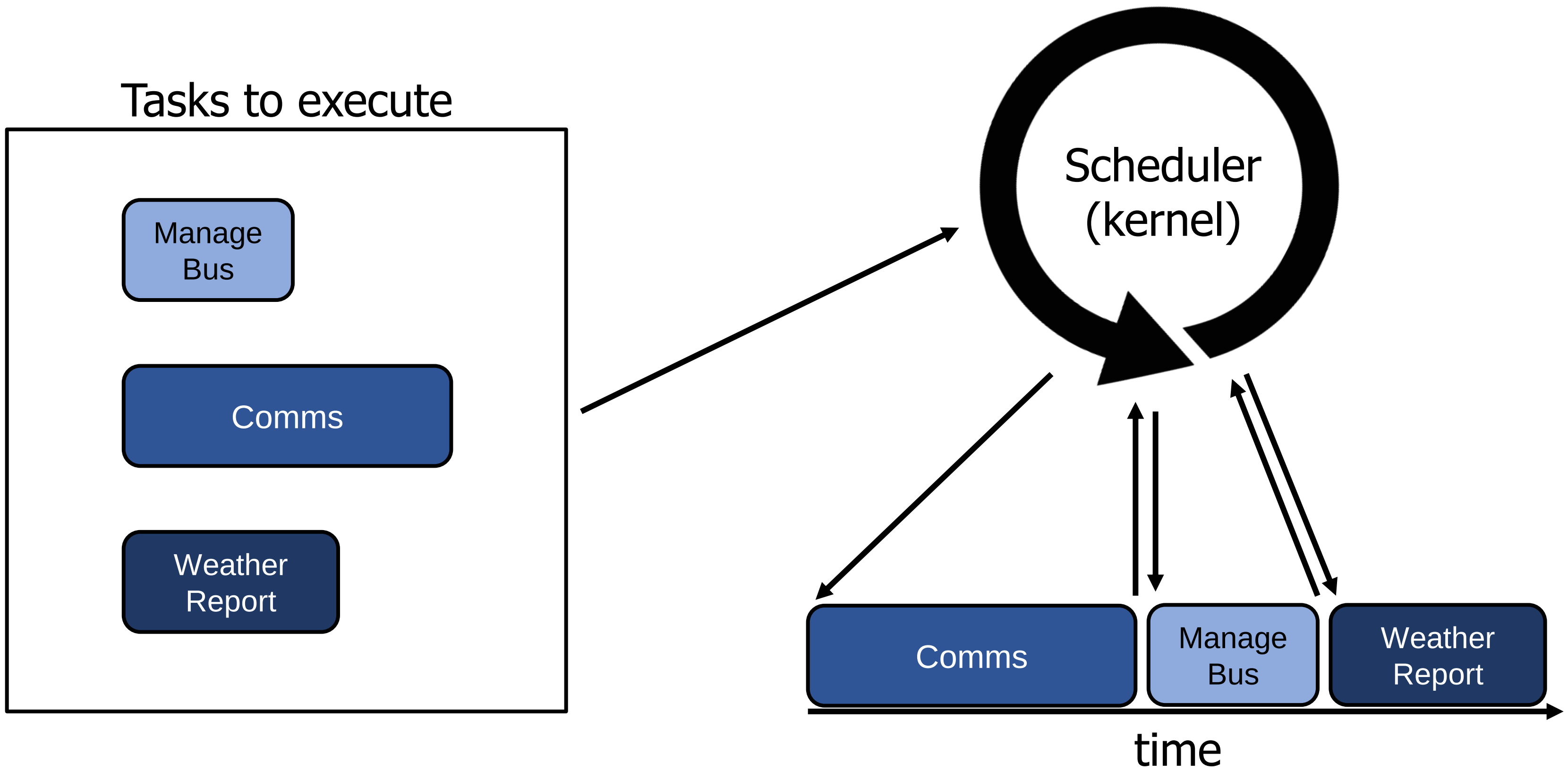
- In each of these scenarios
  - Computer failures are unacceptable
  - Humans can’t intervene to resolve issues
  - We’re going to need a computer system with performance *guarantees*

Example: Pathfinder

Radiation-hardened
IBM CPU



Pathfinder had periodic tasks that must be executed



Real-Time Operating Systems

- Goal: guaranteed performance
  - Meet *deadlines* even if it means being 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

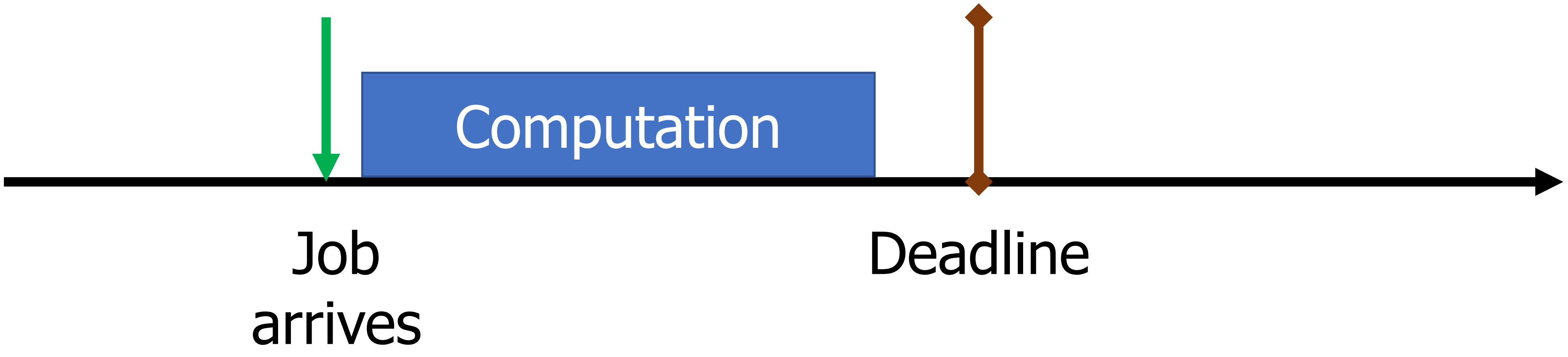
Types of real-time schedulers

- 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 as they don't account for deadlines



Types of real-time jobs

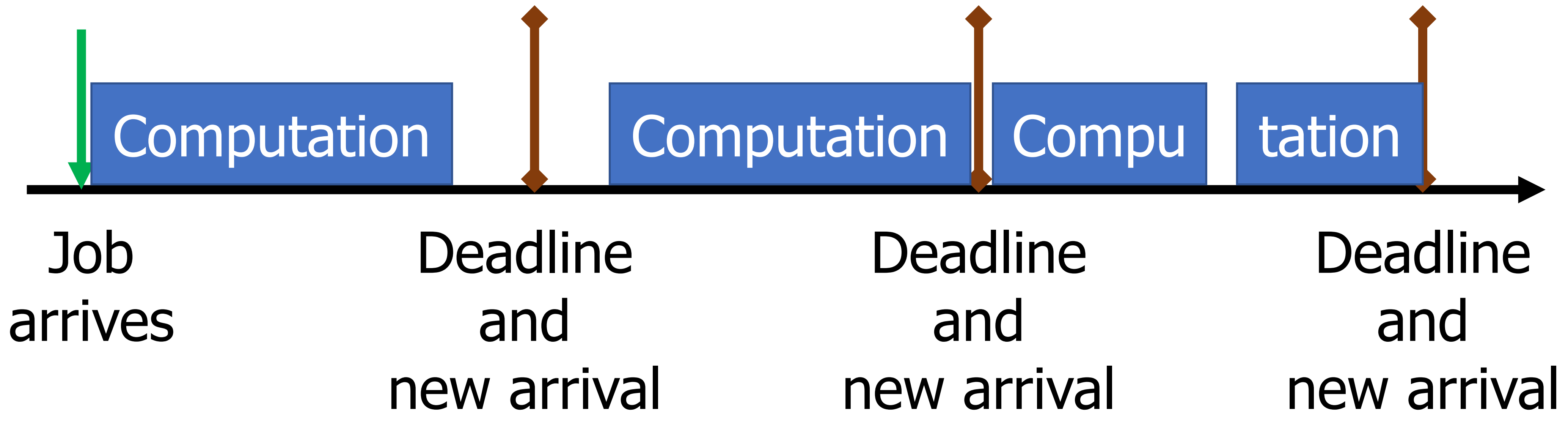
- Aperiodic
  - Jobs we are already accustomed to
  - Unpredictable start times, no deadlines (really, not real-time at all)

- 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

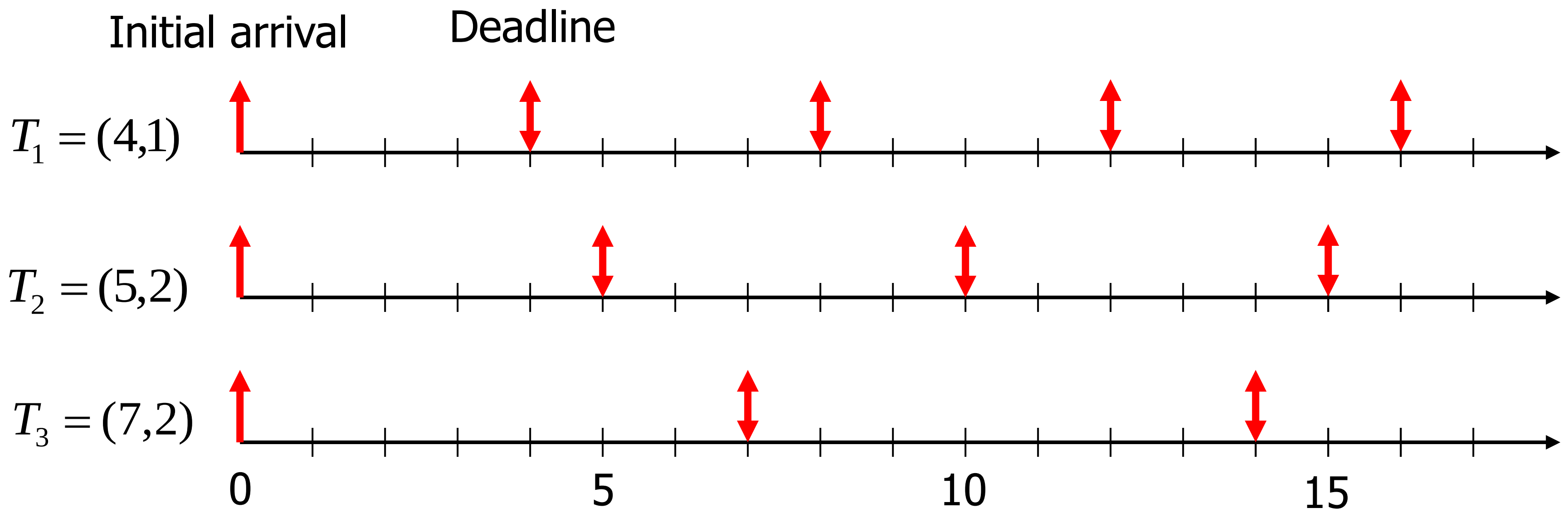


Outline

- **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

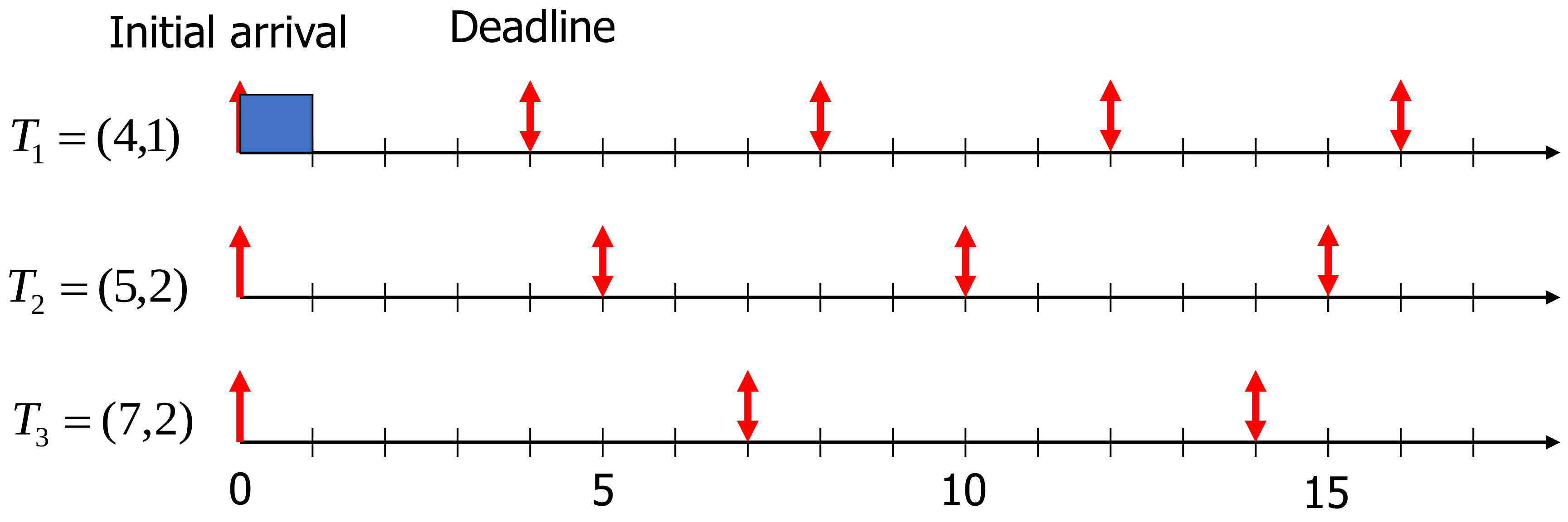
Earliest Deadline First (EDF) Scheduling

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



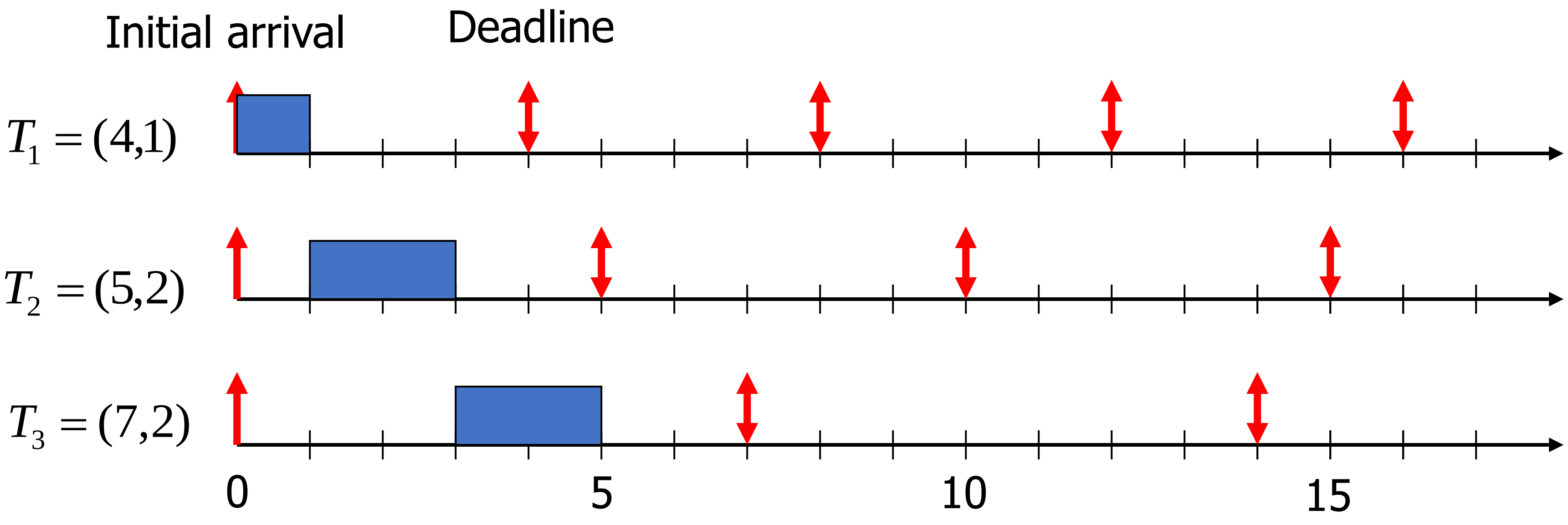
Earliest Deadline First (EDF) Scheduling

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



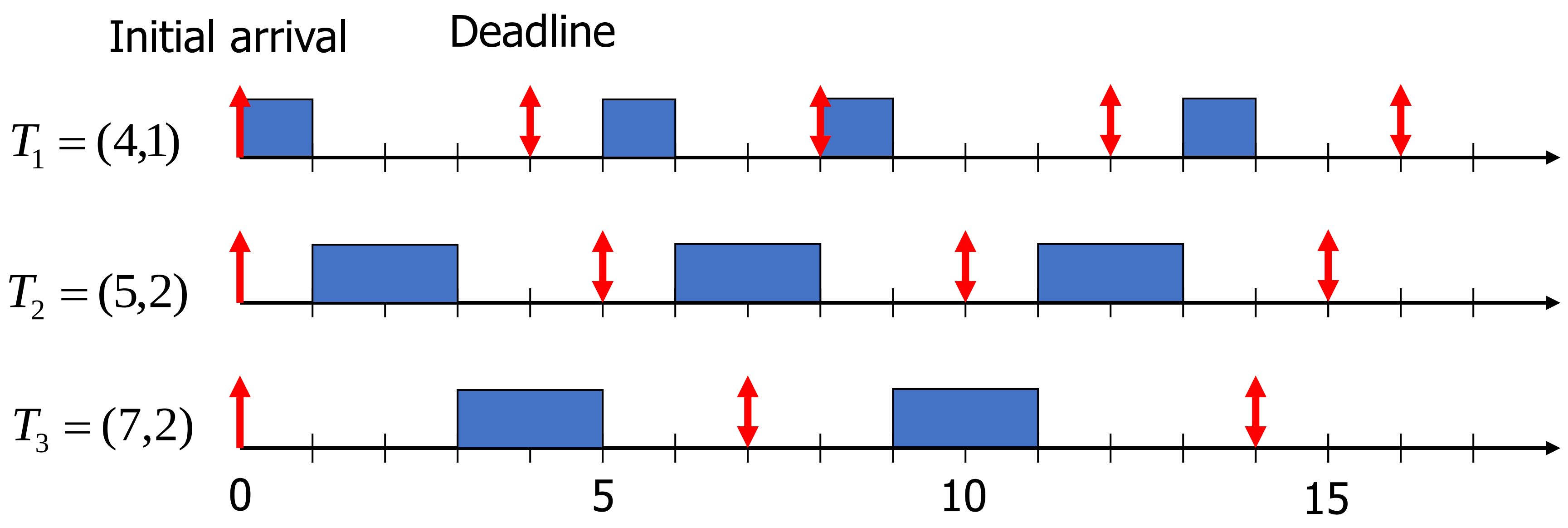
Earliest Deadline First (EDF) Scheduling

- 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

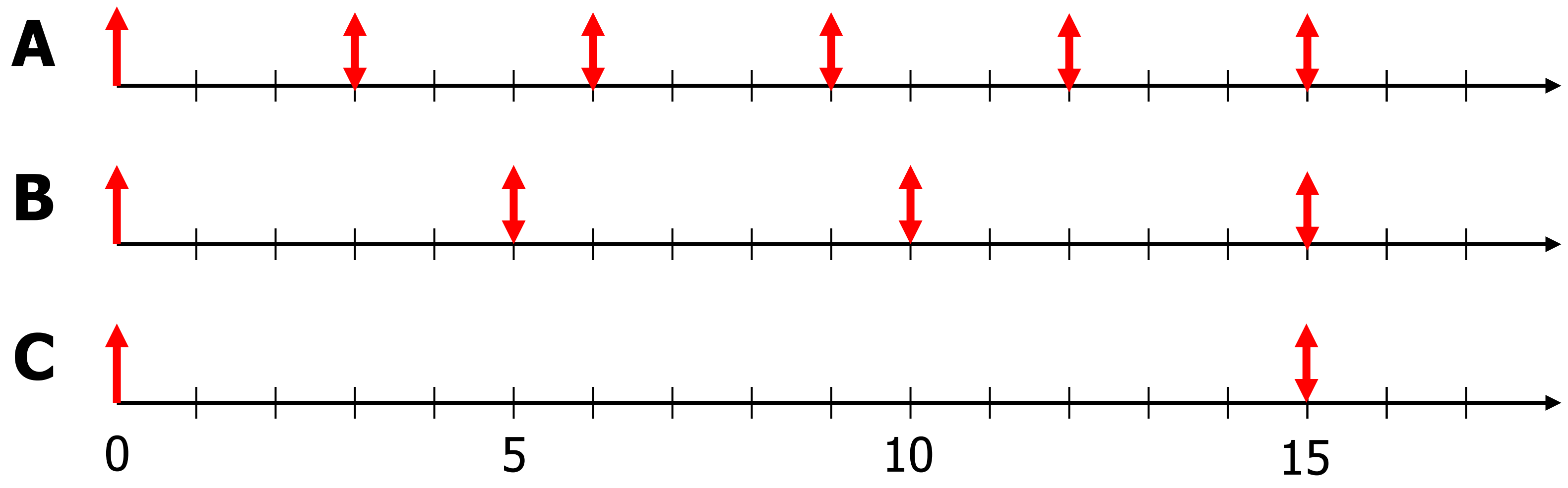
- For *n* tasks with computation time *C* and deadline (period) *D*
  - A feasible schedule exists if **utilization** is less than or equal to one:

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

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

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

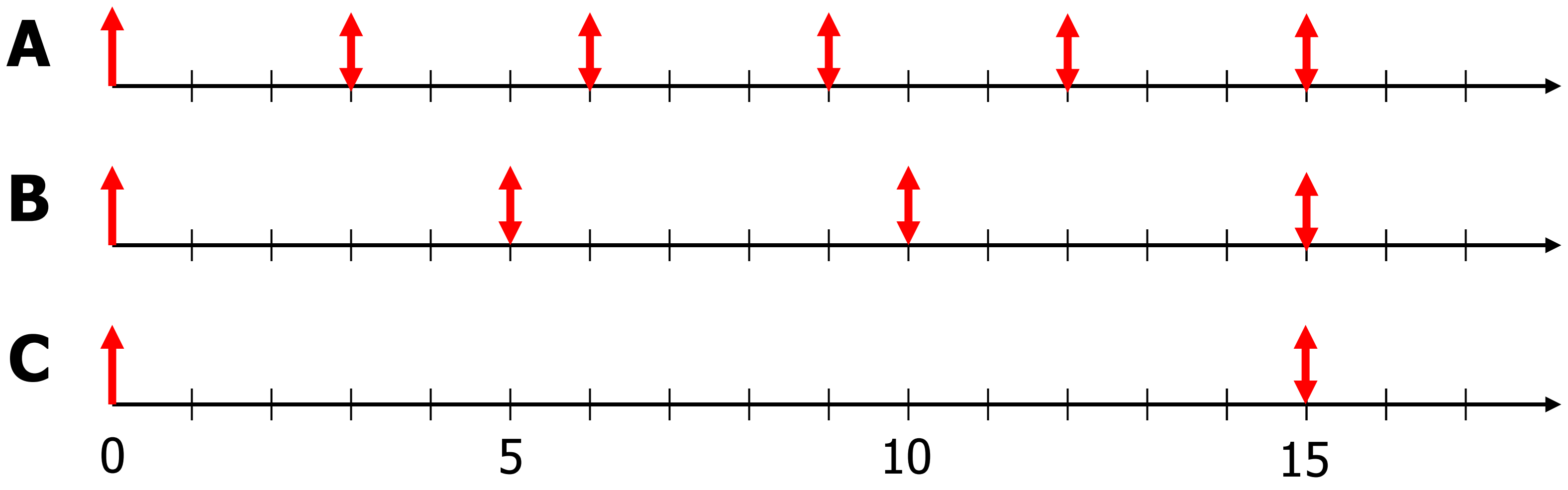


Check your understanding

- 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

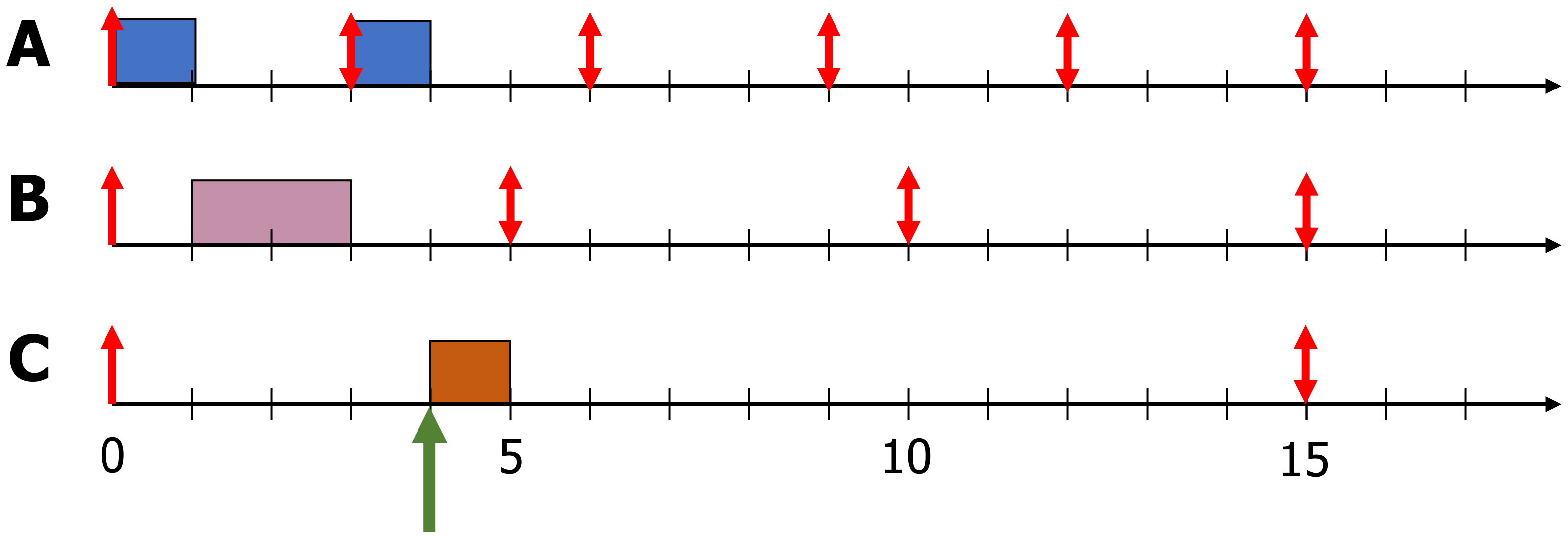


Check your understanding

- 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



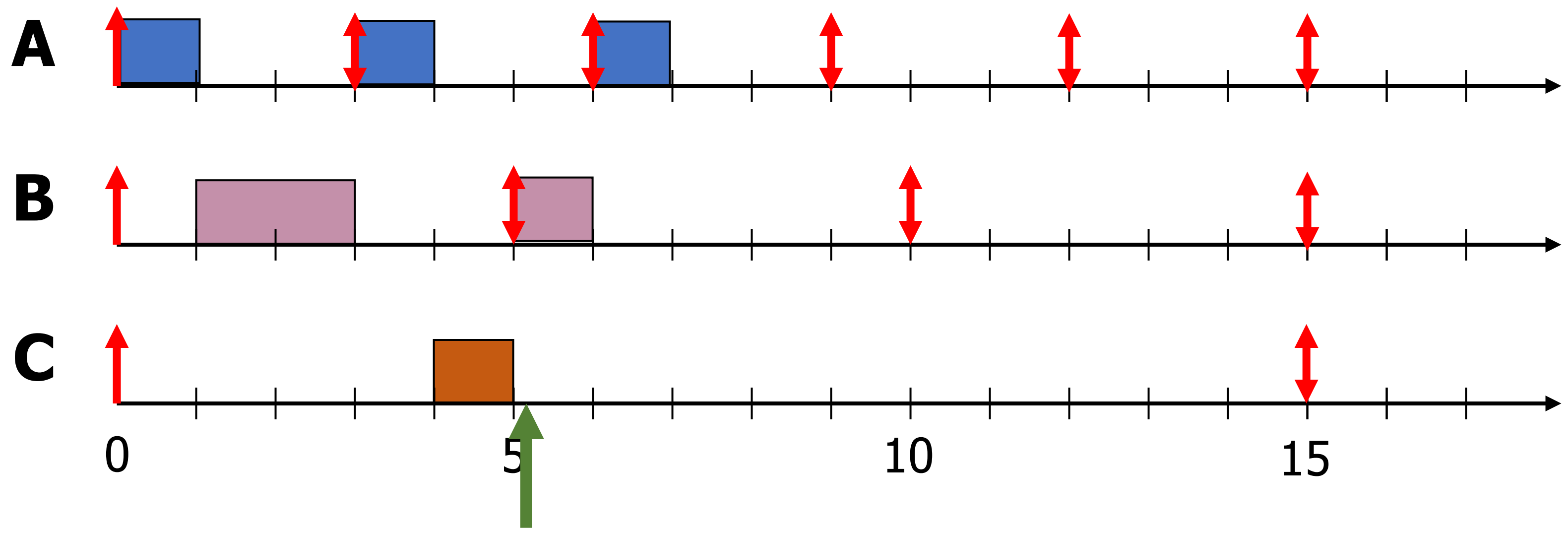
Can’t start a job *before* its period

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



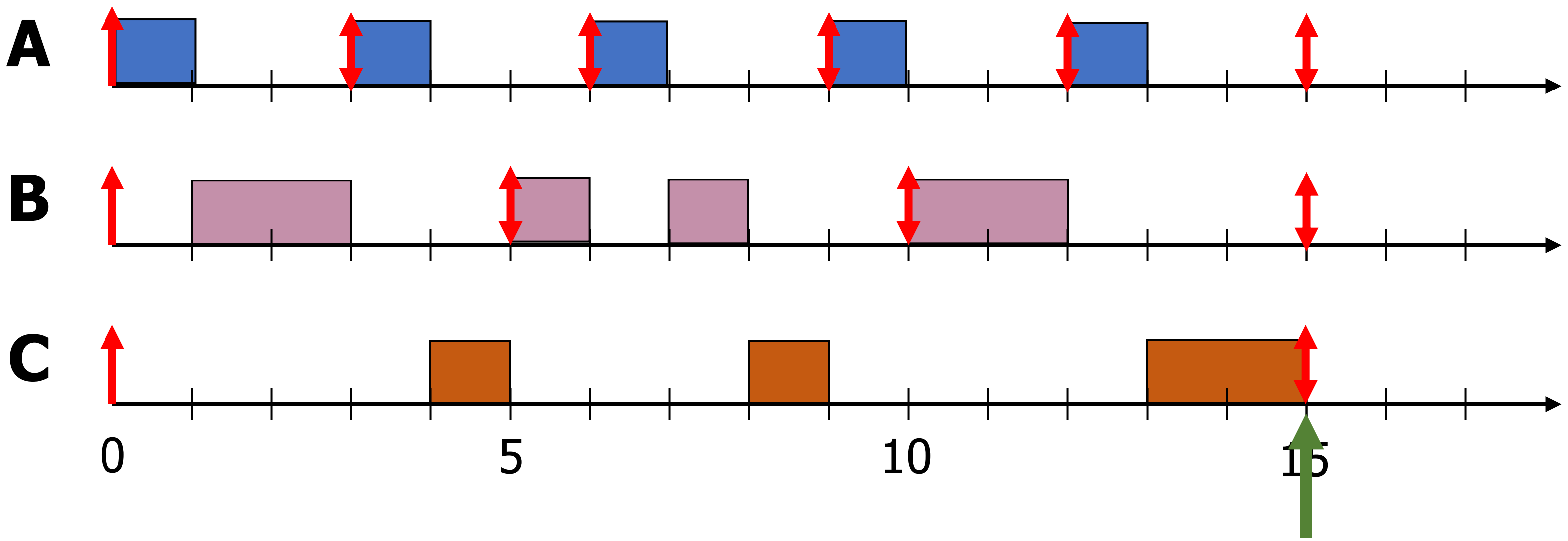
Earliest deadline changes, preempting Job C

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

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

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

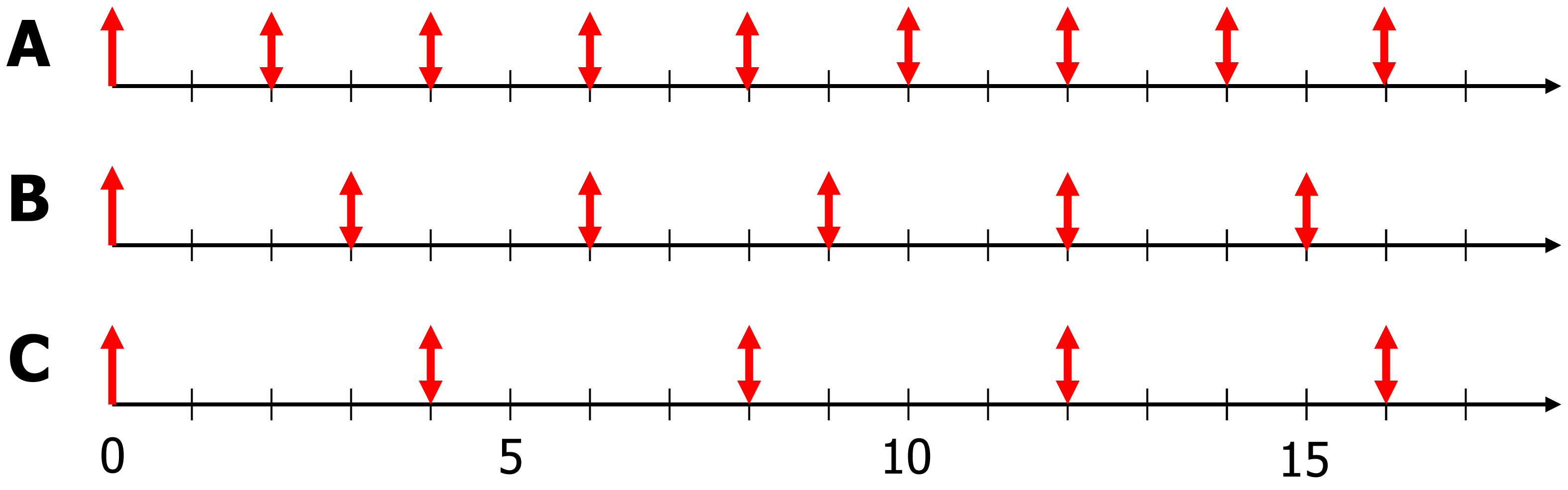


Schedule repeats at least common multiple

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

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

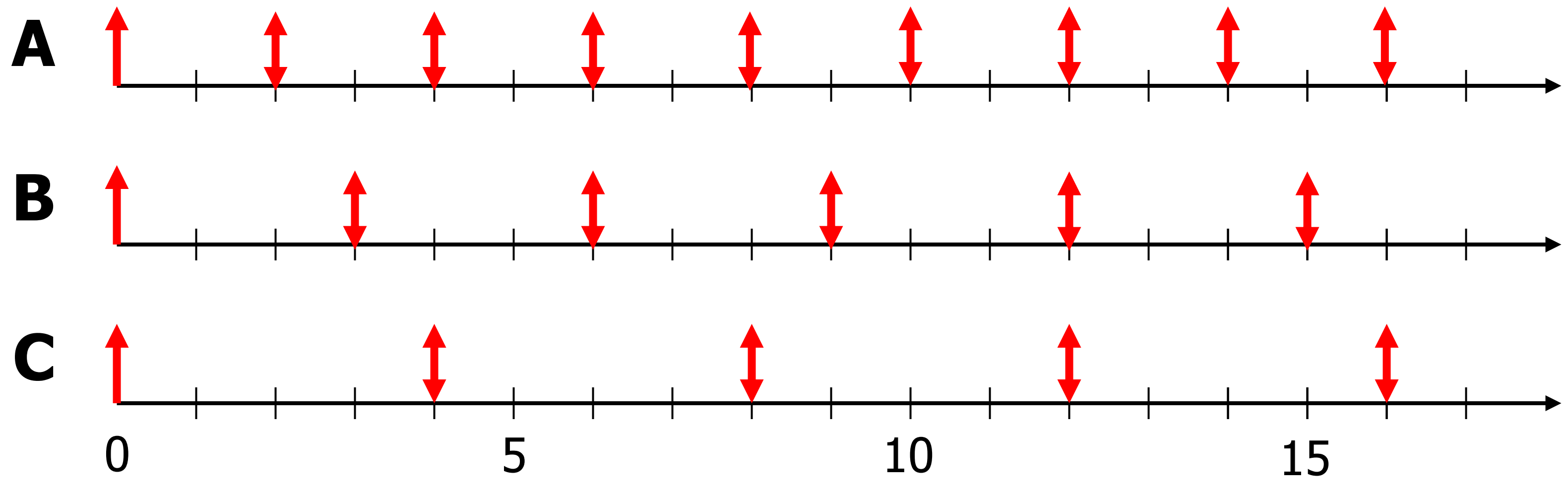


Check your understanding

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

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

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

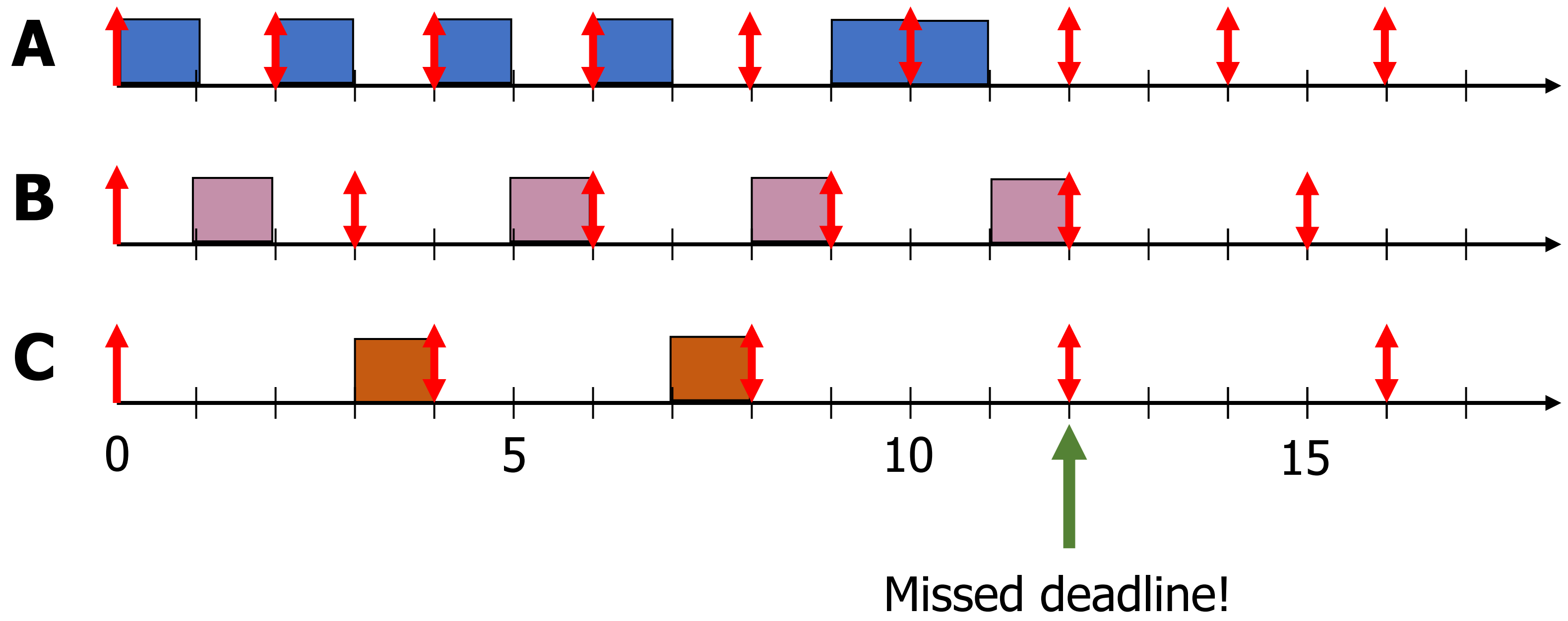


Check your understanding

- 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



Break + Thinking

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

Break + Thinking

- Where do 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 (can use 100% of the CPU)

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

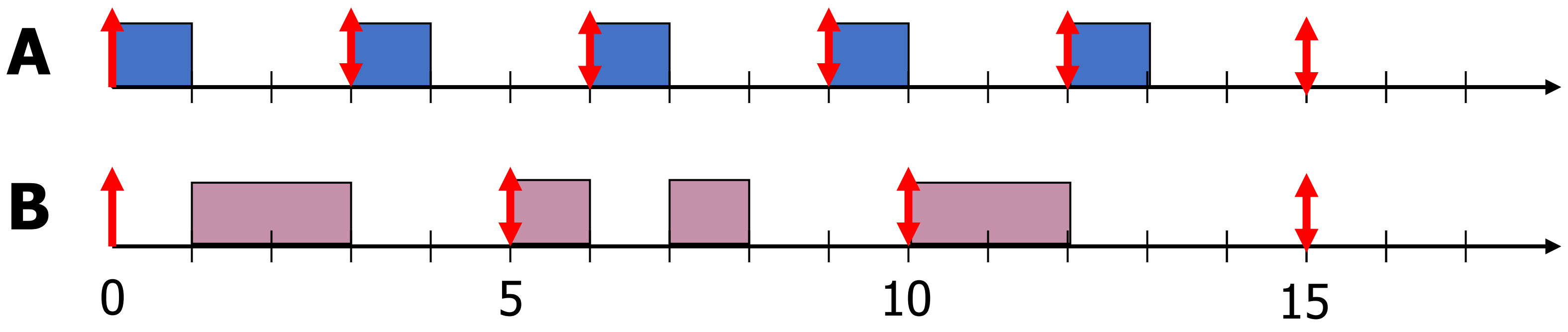
Rate Monotonic Scheduling (RMS)

- Priority scheduling
- Assign fixed priority of 1/Period for each job
  - Makes the scheduling algorithm simple and stable
  - Deterministic failures: 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

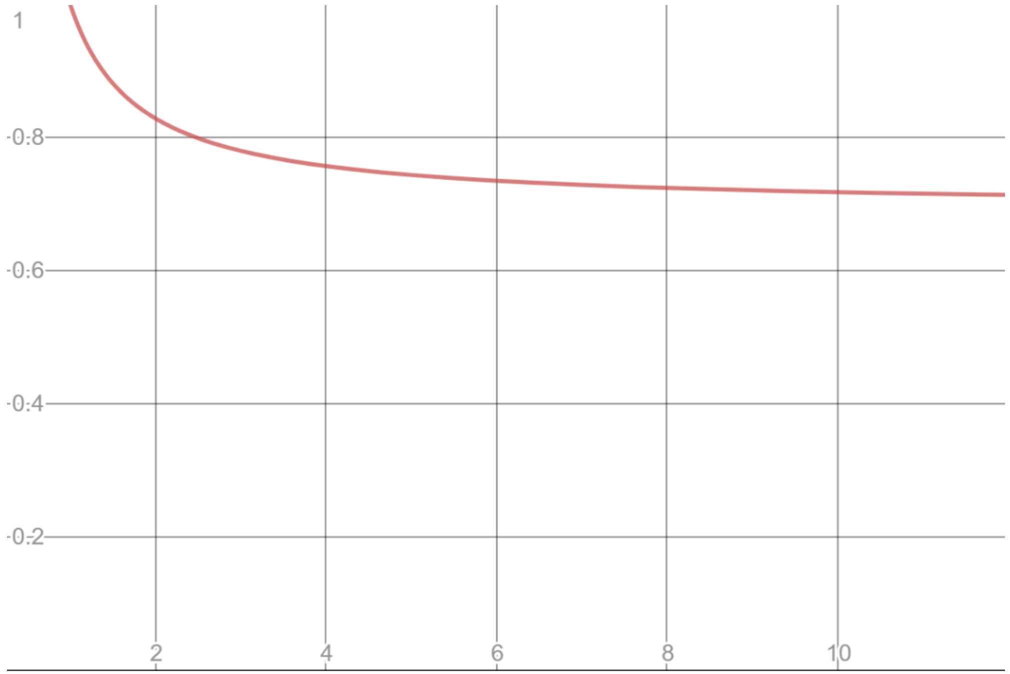


Schedulability test for RMS

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

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

Lower Bound on schedulability



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

  …
- U(∞) = 0.693

RMS schedulability test is conservative

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

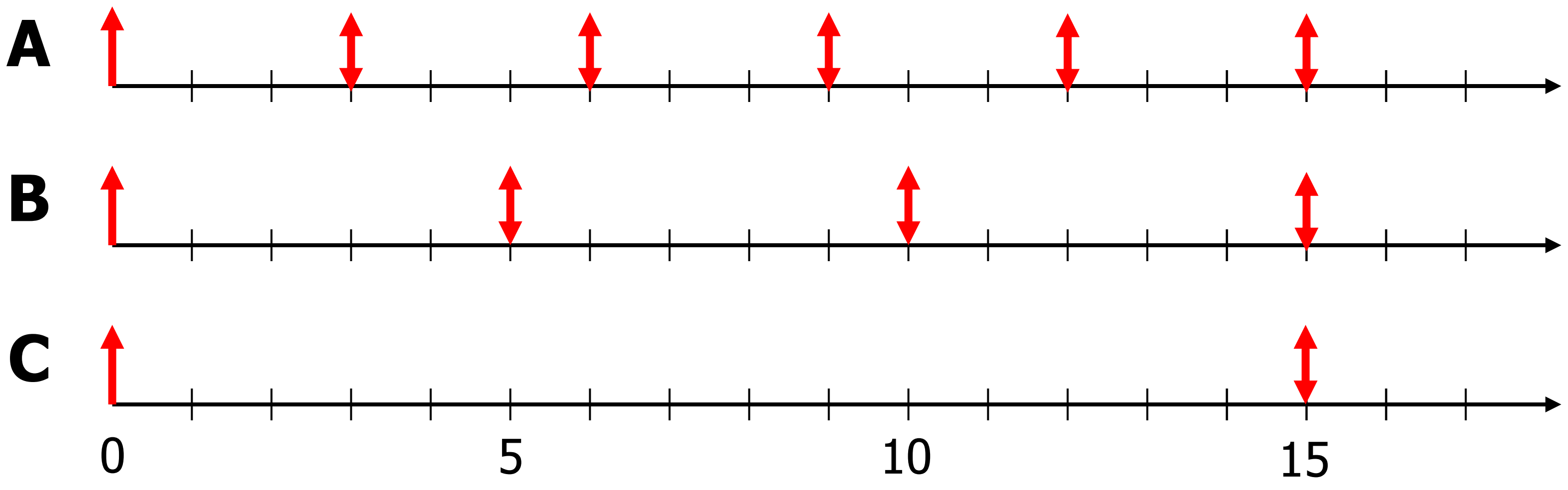
- $0 \leq U \leq n * (2^{\frac{1}{n}} - 1)$
  - Schedulable! (so less than 69% is always schedulable)

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

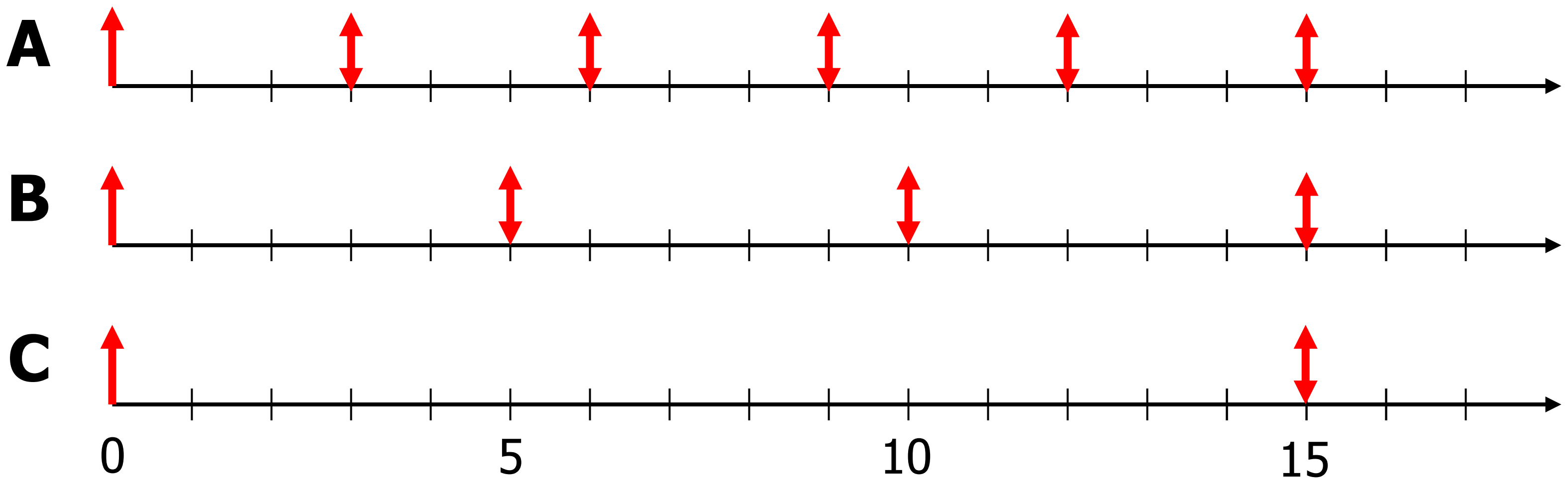


Check your understanding

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

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

U = 1
Maybe schedulable!

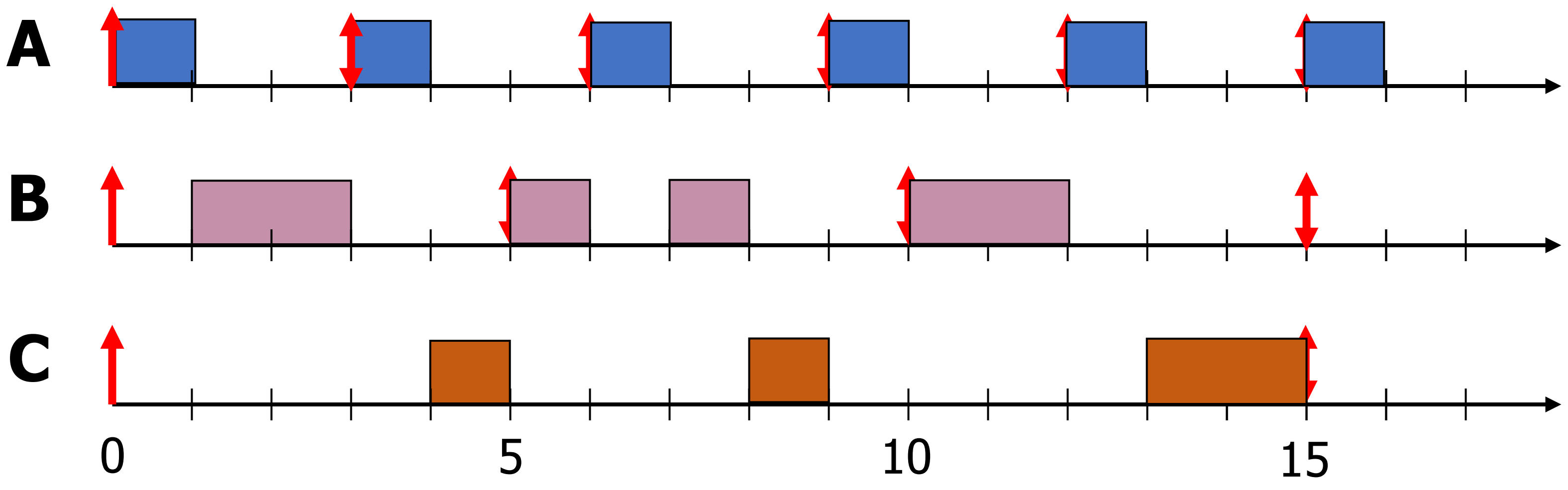


Check your understanding

- 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)

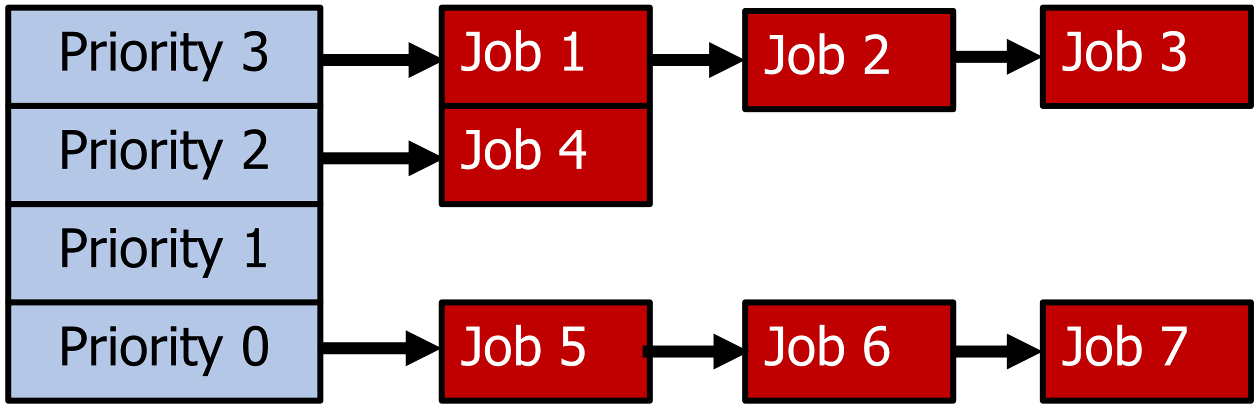
Outline

- 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)



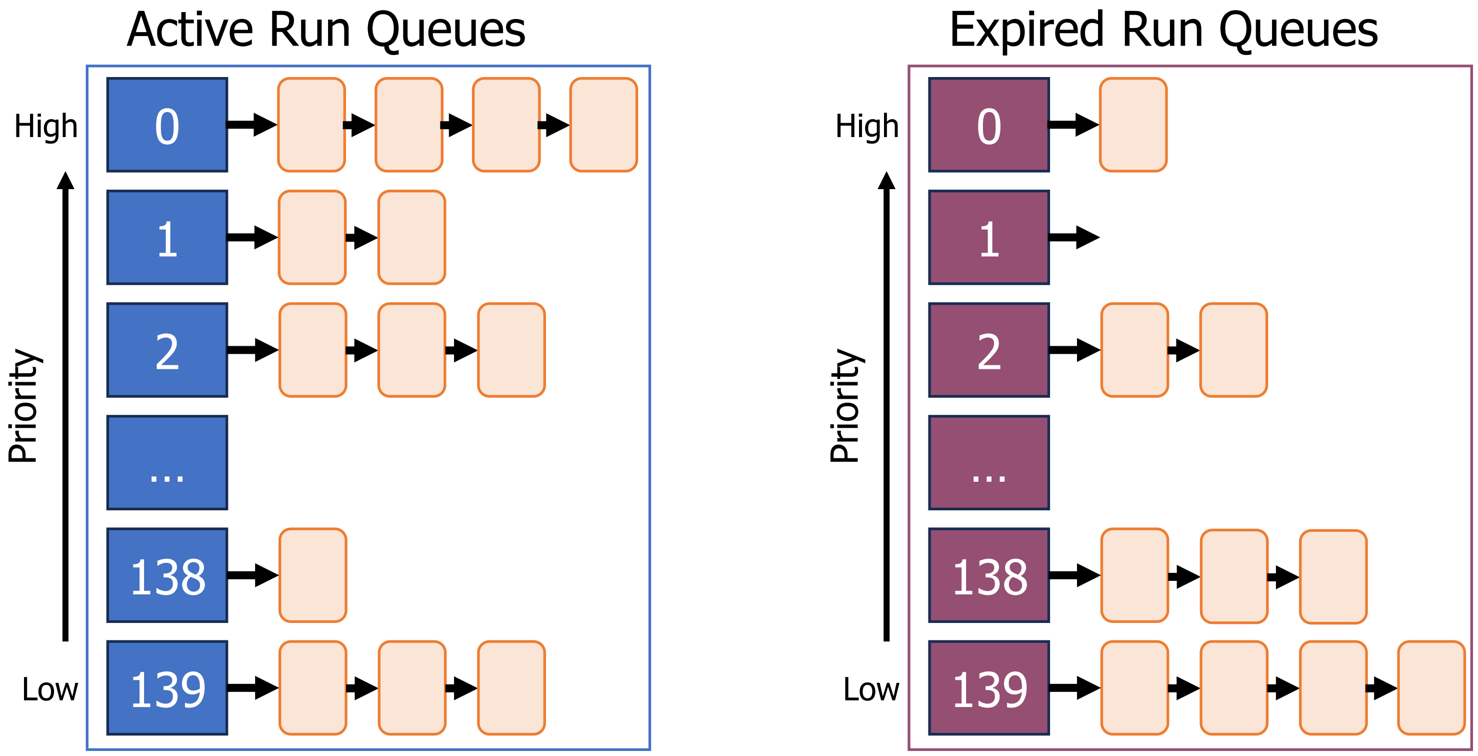
Linux O(1) scheduler (Linux 2.6, 2003-2007)

- 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…

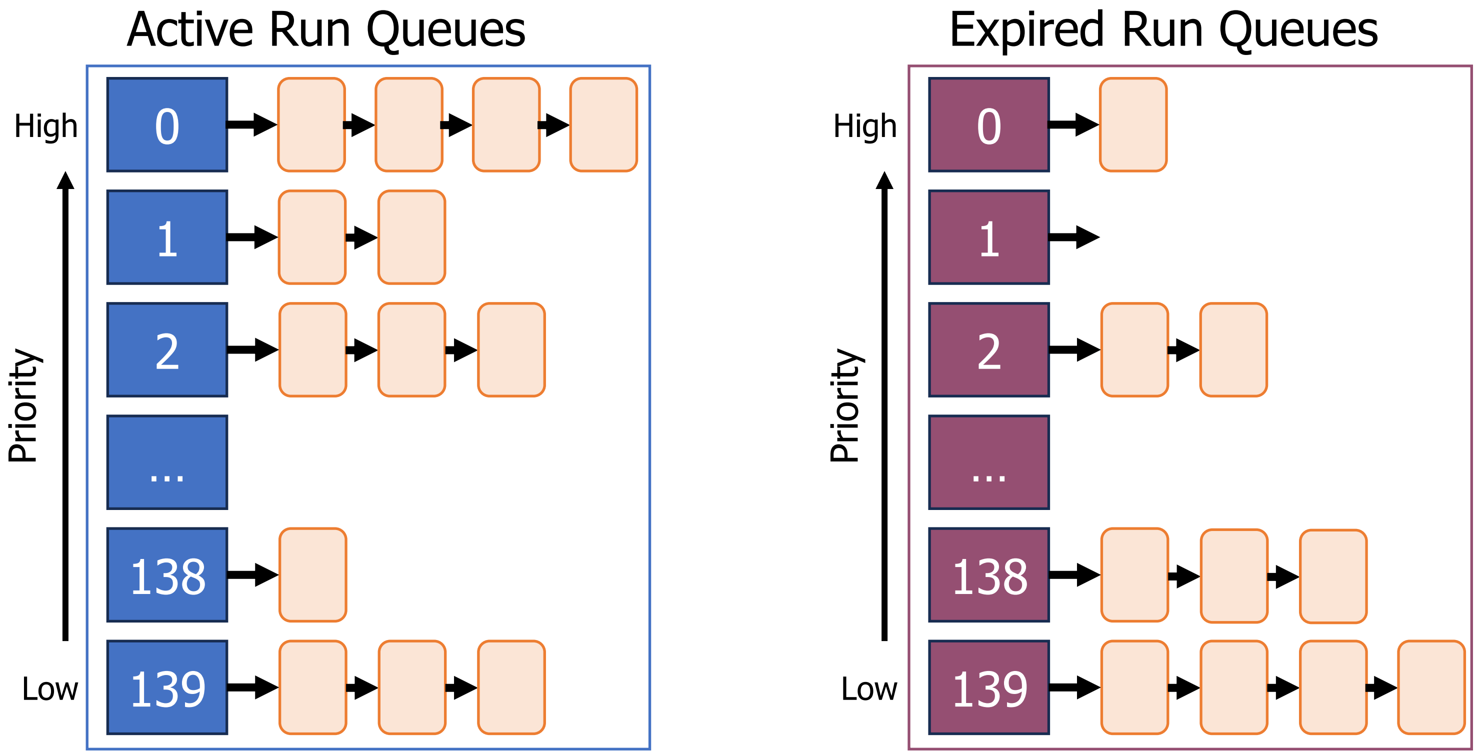
O(1) scheduler: scheduling algorithm

- Find the highest priority run queue that's not empty
- Remove and run the first job from it



O(1) scheduler: when swapping out a job

- Always recalculate job priority
  - Heuristics: interactivity guess, process "niceness", possibly other measurements
- If job has not expired its quota, place at end of correct active queue (round-robin at a priority level)
- If job has expired its quota, place at end of correct expired queue
  - When all jobs are gone from the active queue, swap which queue is "active" and which is "expired"



- Issue with O(1) scheduler:
  - Determining priority is challenging
  - "Complex heuristics" make decisions hard to understand at runtime

Priorities can lead to starvation

- In priority-based schedulers we've seen so far:
  - **Always prefer to give the CPU to a prioritized job**
  - Non-prioritized jobs may never get to run
    - So they need some special mechanism to *occasionally* run them
    - "Time quota" at a priority level, or periodic "resets"

- 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

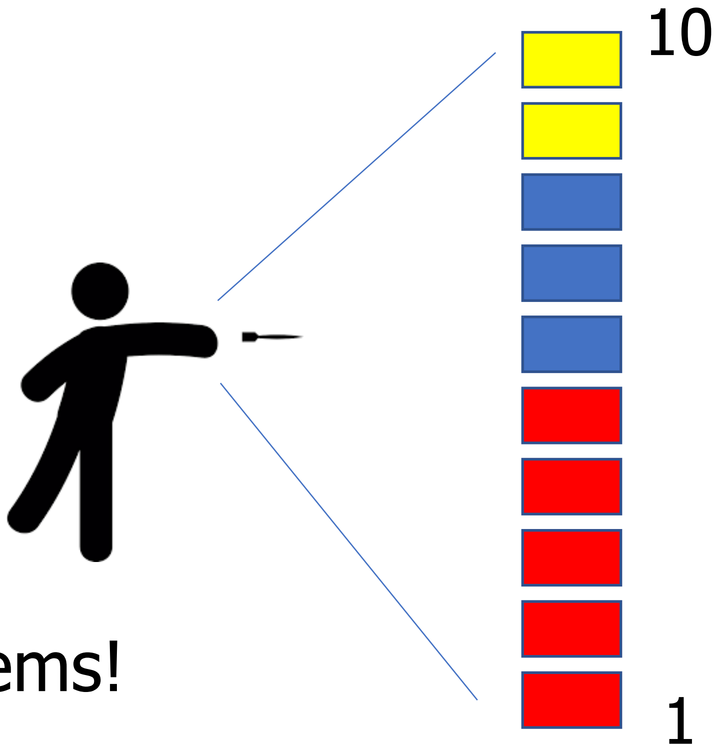
- Instead, we can share the CPU proportionally
  - Give each job a portion of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)

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!
  - Probabilistic in nature

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

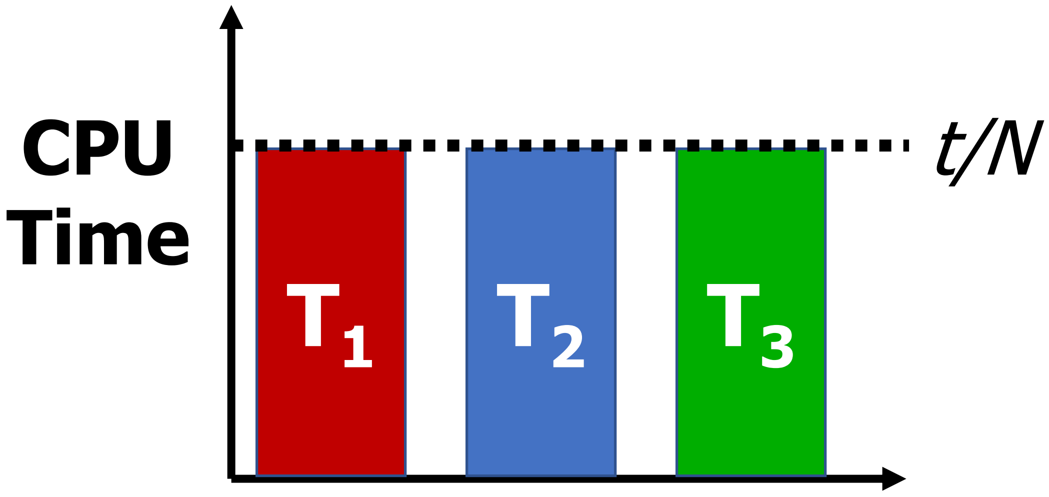
Stride scheduling example

- 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

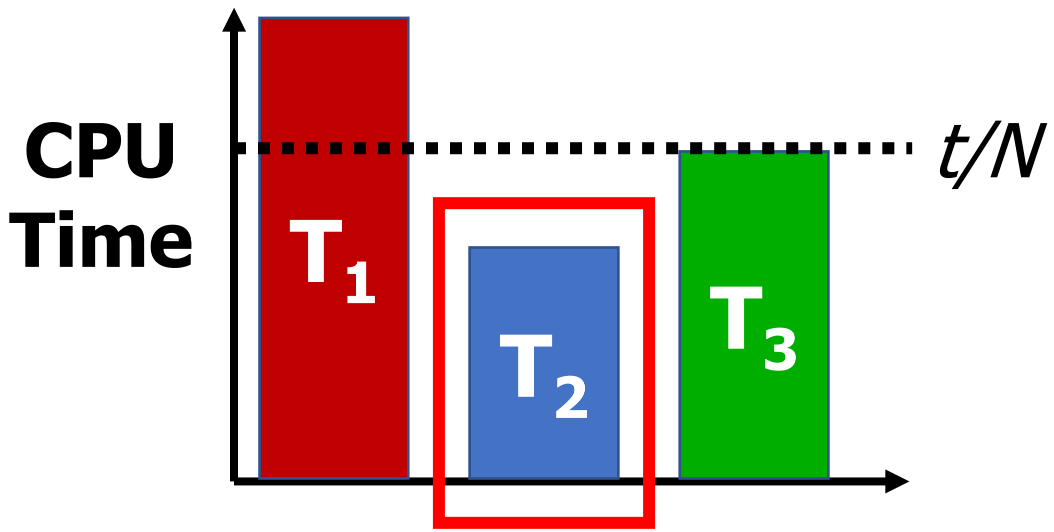
At *any* time *t*
we want to observe:



- 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) (2007-2023)

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 scheduling occurs again
  - 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”

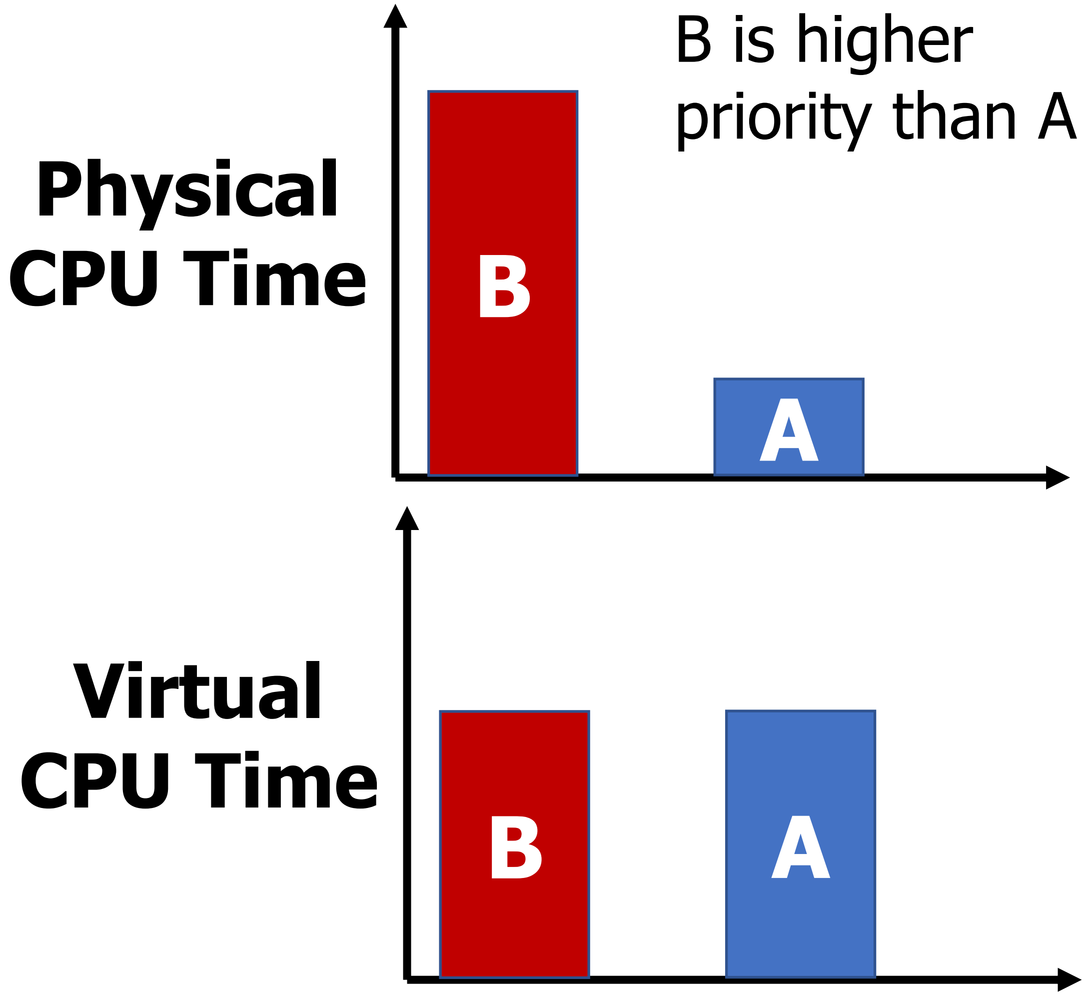
- Virtual runtime doesn’t have to match wall time
- Create a conversion from actual runtime to virtual runtime
  - **High priority jobs:**

    1 second real-time
    -> 0.5 seconds virtual-time

  - **Low priority jobs:**

    1 second real-time
    -> 2 seconds virtual-time

- Scheduler makes decisions solely based on equalizing *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

CFS updates over time

- 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 led to 13%+ slowdown in jobs
  - https://blog.acolyer.org/2016/04/26/the-linux-scheduler-a-decade-of-wasted-cores/

Modern scheduling challenges

- Fair sharing of CPU time is insufficient
    - Maximize cache usage
    - Maximize processor affinity
    - Reduce energy consumption
    - Hybrid systems with heterogeneous processing capabilities

- Particular focus: latency requirements
    - Some processes need to respond quickly to new data
    - They don’t need more processing *time*. They need the time more quickly
    - Heuristic shortcuts were added to CFS to allow some jobs to jump the queue

Earliest Eligible Virtual Deadline First (EEVDF) (2023-Present)

- Algorithm first described in a 1995 research paper
  - Run job with earliest "virtual deadline"
  - TLDR: share processor time proportionally, but schedule within that based on latency

- Still divides processor time equally between jobs, like CFS
  - Biased by priority of the job. Higher priority means larger share

- Calculate "lag" for each job
  - Measurement of how far it's behind a fair share of processor time
  - Negative lag means a job has run more than its fair share already
    - Job won't be eligible to run until lag >= 0
    - Lag increases automatically as other jobs run. So time until lag >= 0 can be calculated

- Virtual deadline for job: time until lag >= 0, plus duration it should run for
  - Now + timeslice for any jobs below fair share of processor time
  - Future + timeslice for any jobs above fair share of processor time
  - Where timeslices vary by priority of the job

https://lwn.net/Articles/925371/

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 or EEVDF
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