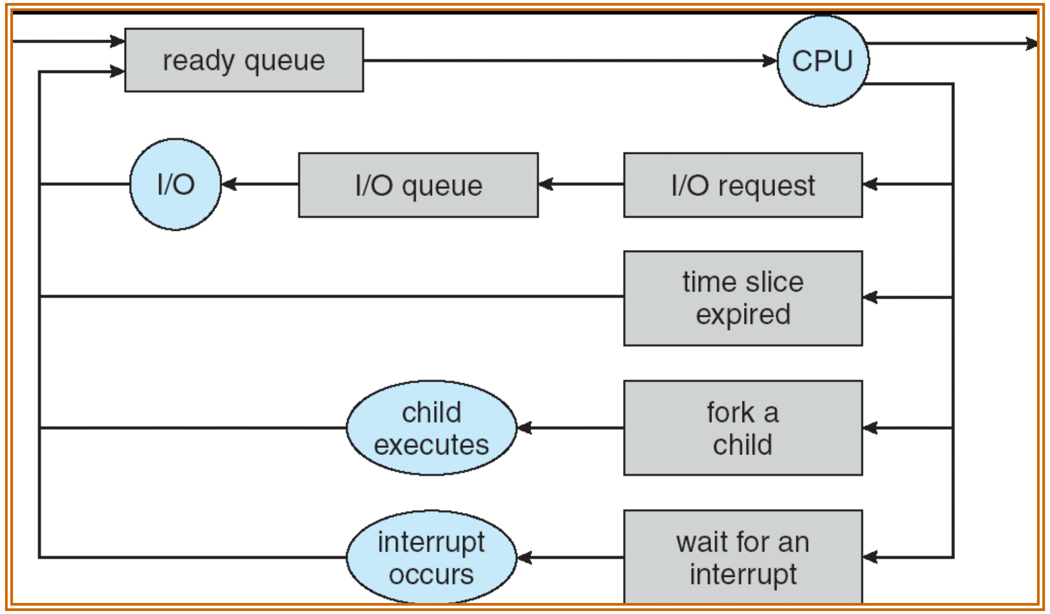
CS162
Operating Systems and
Systems Programming
Lecture 11

Scheduling 2:
Case Studies, Real Time, and Forward Progress

Administrivia

- Happy hour: by Friday 26/2, send us a picture with a beverage of your choice of you meeting in a group!
- Submit midsemester survey. Extra credit point in the game if 80% of class does it!
- Academic misconduct. More details on Piazza. Please come forward using the form
- Project 1: Project code due tomorrow (26/2). Final report due Sunday (28/2)
- Don’t forget to turn on camera for discussion sections!

Recall: Scheduling



- Question: How is the OS to decide which of several tasks to take off a queue?
- Scheduling: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

Recall: Scheduling Policy Goals/Criteria

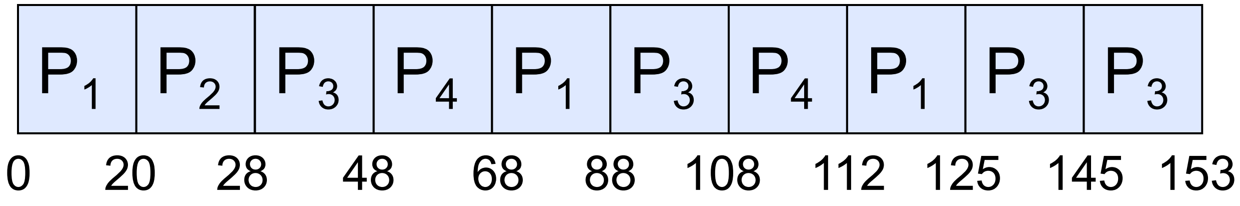
- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

Recall: Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart is:



  - Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½

Recall: What if we Knew the Future?

- Shortest Job First (SJF):
  - Run whatever job has least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)

- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)

- These can be applied to whole program or current CPU burst
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

History of Schedulers in Linux

- O(n) scheduler
  – Linux 2.4 to Linux 2.6

- O(1) scheduler
  – Linux 2.6 to 2.6.22

- CFS scheduler
  – Linux 2.6.23 onwards

Case Study: Linux O(n) Scheduler

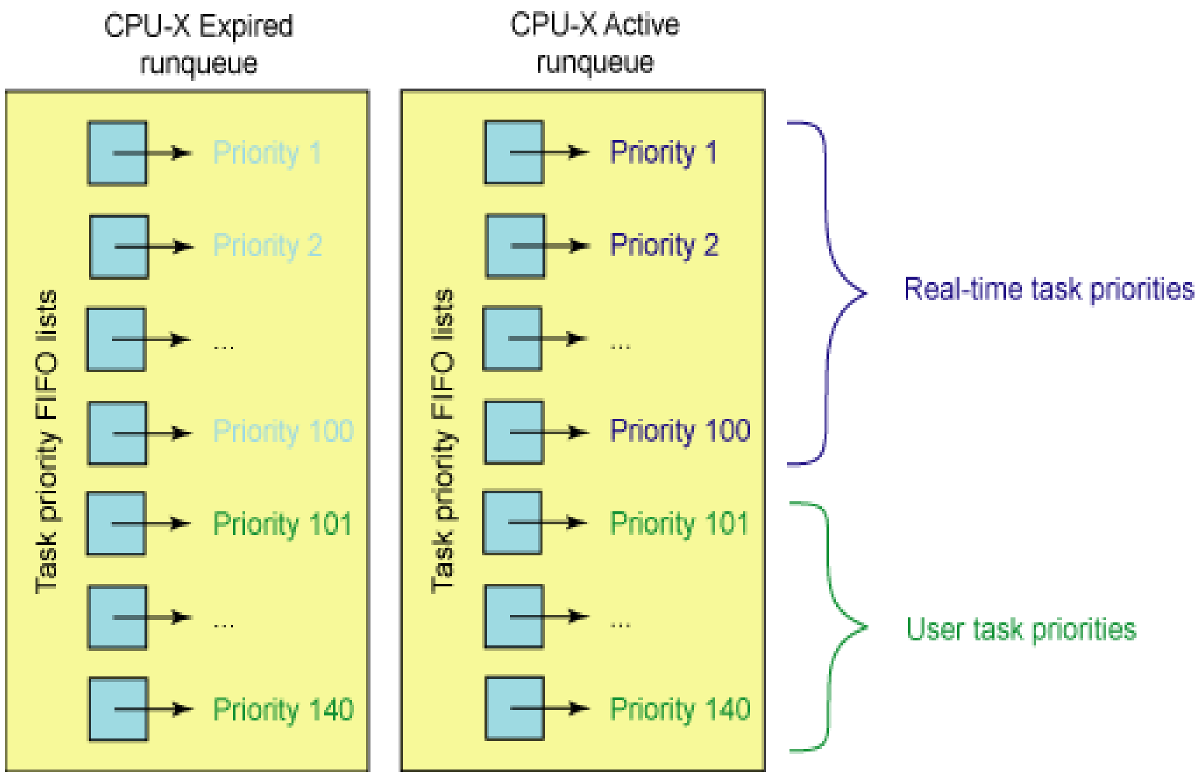
- At every context switch:
  - Scan full list of processes in the ready queue
  - Compute relevant priorities
  - Select the best process to run
- Scalability issues:
  - Context switch cost increases as number of processes increase
  - Single queue even in multicore systems

Case Study: Linux O(1) Scheduler

- Next process to run is chosen in **constant time**

- Priority-based scheduler with 140 different priorities
  - Real-time/kernel tasks assigned priorities 0 to 99 (0 is highest priority)
  - User tasks (interactive/batch) assigned priorities 100 to 139 (100 is highest priority)
    - » Can be set using the nice system call.

Case Study: O(1) Scheduler – User tasks



- Per priority-level, each CPU has **two ready queues**
  - An active queue, for processes which have not used up their time quanta
  - An expired queue, for processes who have
- Timeslices/priorities/interactivity credits all computed when jobs finishes timeslice
- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into "Timeslice Granularity" chunks – round robin through priority

O(1) Scheduler – User tasks – Priority Adjustment

- User-task priority adjusted ±5 based on heuristics
  - » p->sleep_avg = sleep_time – run_time
  - » Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
- Interactive Credit
  - » Earned when a task sleeps for a "long" time
  - » Spend when a task runs for a "long" time
  - » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
- However, "interactive tasks" get special dispensation
  - » To try to maintain interactivity
  - » Placed back into active queue, unless some other task has been starved for too long…

Case Study: O(1) Scheduler – Real tasks

- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    - » SCHED_FIFO: preempts other tasks, no timeslice limit
    - » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

Real-Time Scheduling

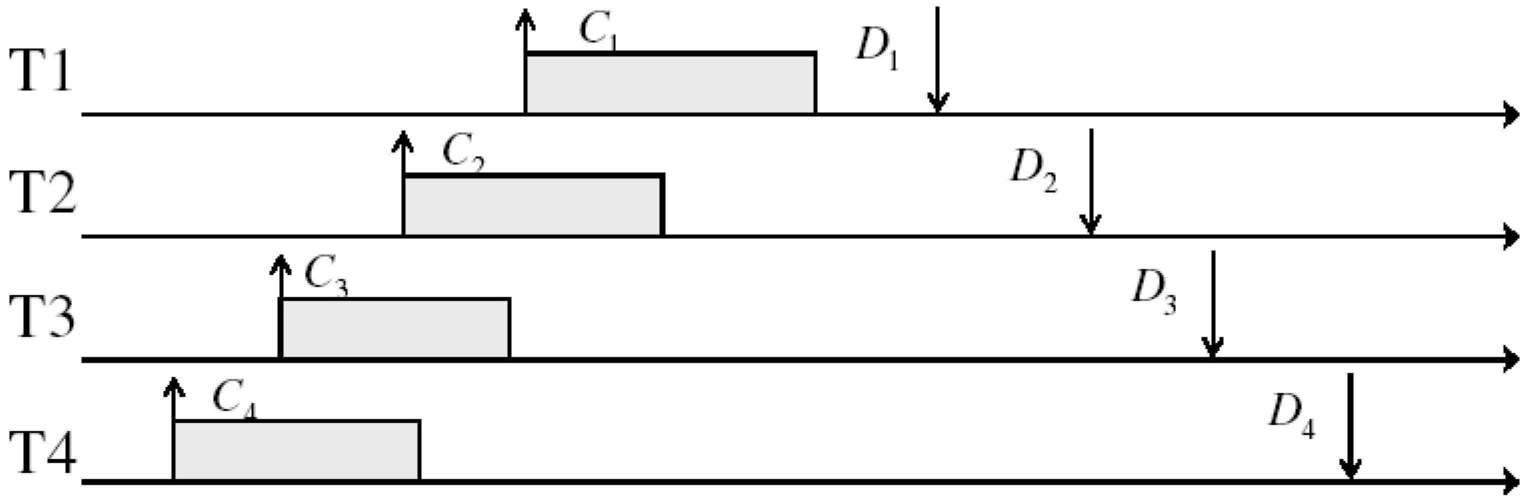
- Goal: Predictability of Performance!
  - We need to predict with confidence worst case response times for systems!
  - In RTS, performance guarantees are:
    - » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - » System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!

Real-Time Scheduling

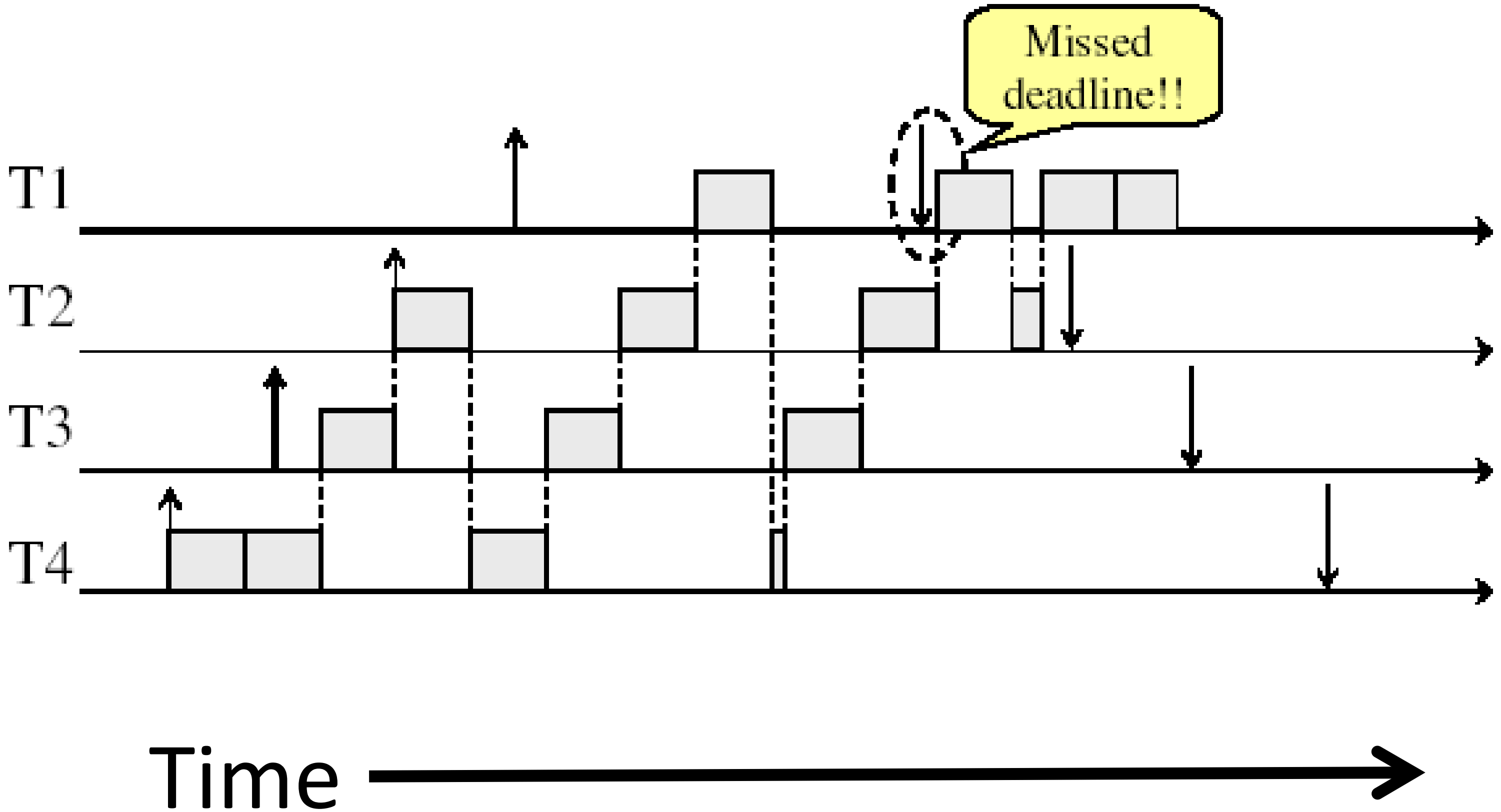
- Hard real-time: for time-critical safety-oriented systems
  - Meet all deadlines (if at all possible)
  - Ideally: determine in advance if this is possible
  - Earliest Deadline First (EDF), Least Laxity First (LLF), Rate-Monitonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)
- Soft real-time: for multimedia
  - Attempt to meet deadlines with high probability
  - Constant Bandwidth Server (CBS)

Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:

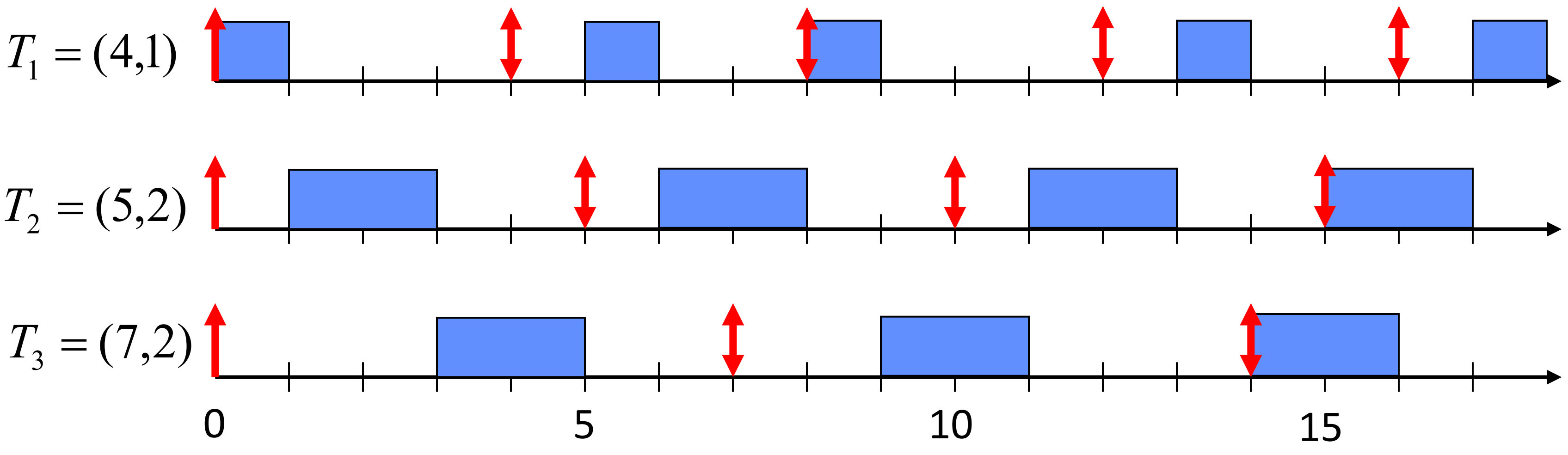


Example: Round-Robin Scheduling Doesn't Work



Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P_i, C_i) for each task i
- Preemptive priority-based dynamic scheduling:
  - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. $D_i^{t+1} = D_i^t + P_i$ for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline



EDF Feasibility Testing

- For n tasks with computation time C and deadline D, a feasible schedule exists if:

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

Case 1:

T1: (2,1) T2: (2,1)

½ + ½ = 1

Case 1:

T1: (2,2) T2: (2,1)

1 + ½ = 1.5

Ensuring Progress

- Starvation: thread fails to make progress for an indefinite period of time

- Starvation (this lecture) ≠ Deadlock (next lecture) because starvation *could* resolve under right circumstances
  – Deadlocks are unresolvable, cyclic requests for resources

- Causes of starvation:
  – Scheduling policy never runs a particular thread on the CPU
  – Threads wait for each other or are spinning in a way that will never be resolved

- Let's explore what sorts of problems we might encounter and how to avoid them…

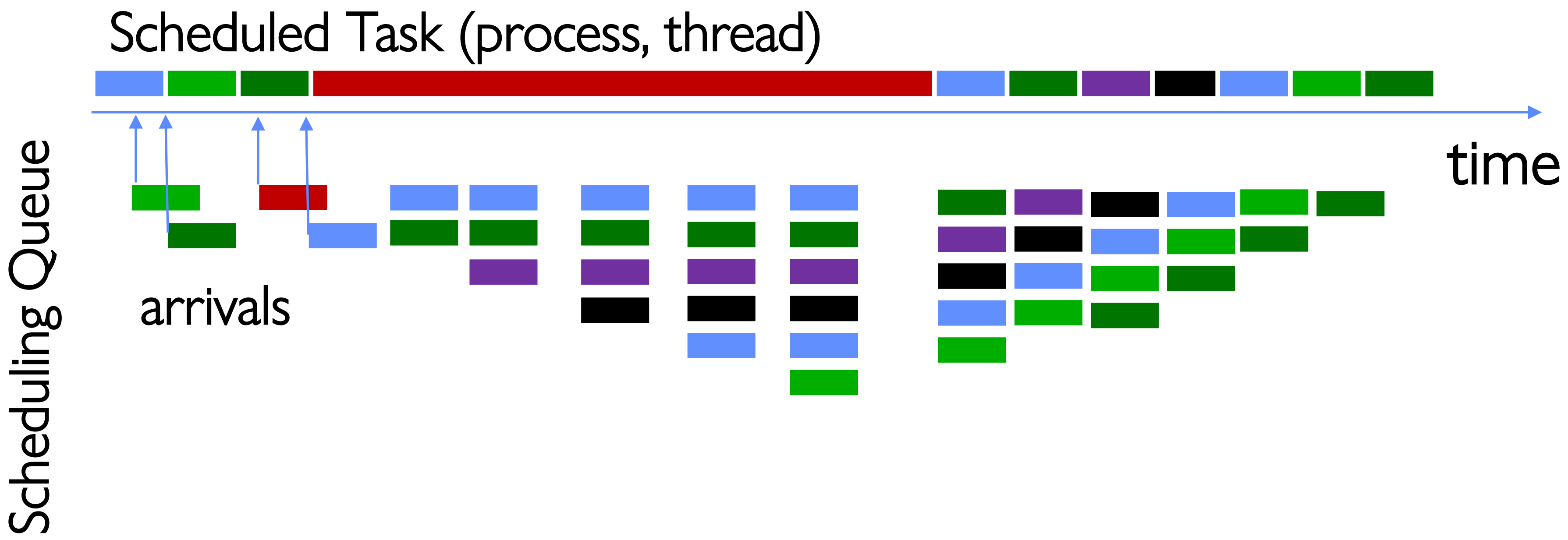
Strawman: Non-Work-Conserving Scheduler

- A *work-conserving* scheduler is one that does not leave the CPU idle when there is work to do
- A non-work-conserving scheduler could trivially lead to starvation
- In this class, we'll assume that the scheduler is work-conserving (unless stated otherwise)

Strawman: Last-Come, First-Served (LCFS)

- Stack (LIFO) as a scheduling data structure
  - Late arrivals get fast service
  - Early ones wait – extremely unfair
  - In the worst case – *starvation*
- When would this occur?
  - When arrival rate (offered load) exceeds service rate (delivered load)
  - Queue builds up faster than it drains
- Queue can build in FIFO too, but "serviced in the order received"…

Is FCFS Prone to Starvation?



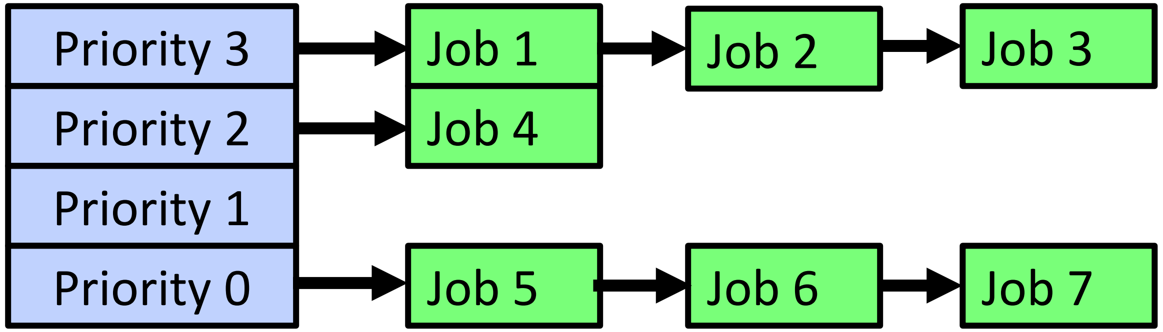
- If a task never yields (e.g., goes into an infinite loop), then other tasks don't get to run
- Problem with all non-preemptive schedulers…
  - And early personal OSes such as original MacOS, Windows 3.1, etc

Is Round Robin (RR) Prone to Starvation?

- Each of *N* processes gets ~1/*N* of CPU (in window)
  - With quantum length *Q* ms, process waits at most *(N-1)*Q* ms to run again
  - So a process can't be kept waiting indefinitely
- So RR is fair in terms of *waiting time*
  - Not necessarily in terms of throughput…

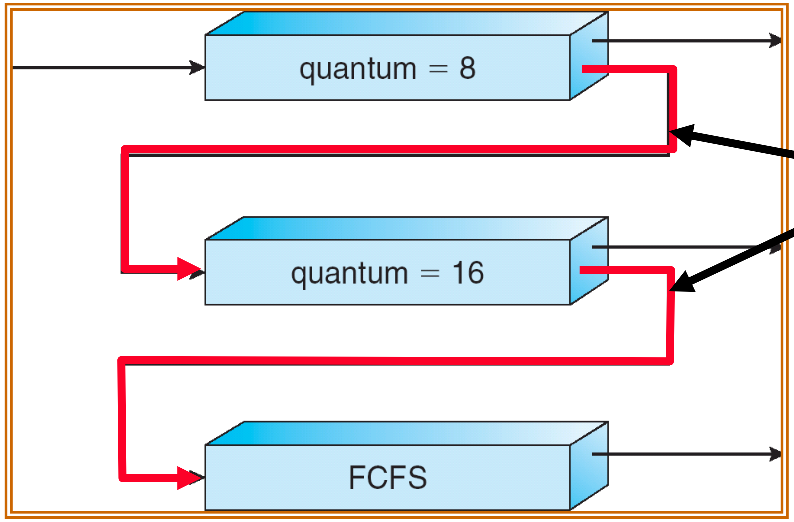
Is Priority Scheduling Prone to Starvation?

- Recall: Priority Scheduler always runs the thread with highest priority
  - Low priority thread might never run!
  - Starvation…



- But there are more serious problems as well…
  - Priority inversion: even high priority threads might become starved

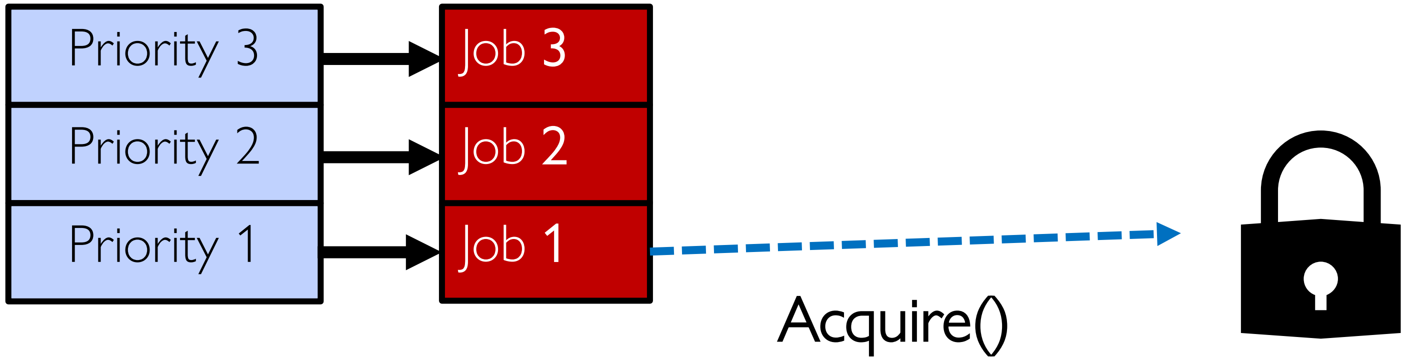
Are SRTF and MLFQ Prone to Starvation?



Long-Running Compute Tasks Demoted to Low Priority

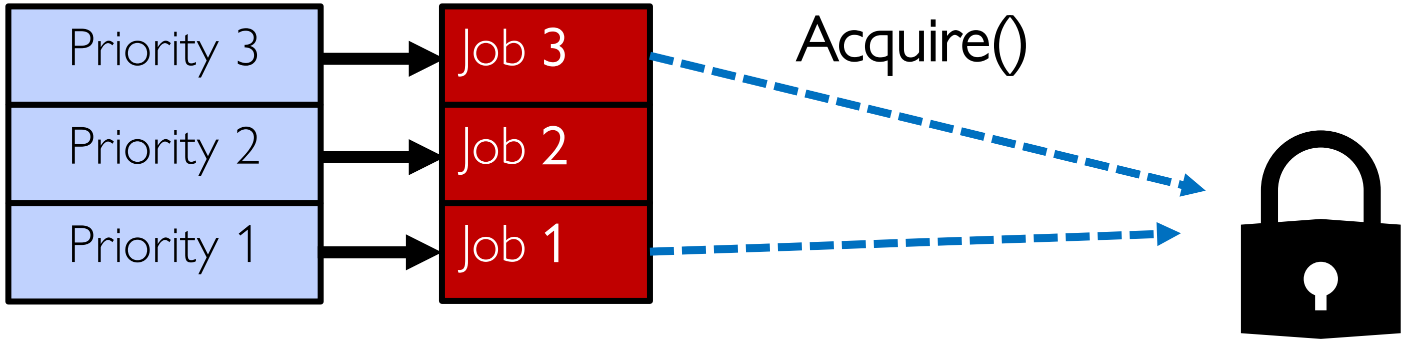
- In SRTF, long jobs are starved in favor of short ones
  – Same fundamental problem as priority scheduling
- MLFQ is an approximation of SRTF, so it suffers from the same problem

Priority Inversion



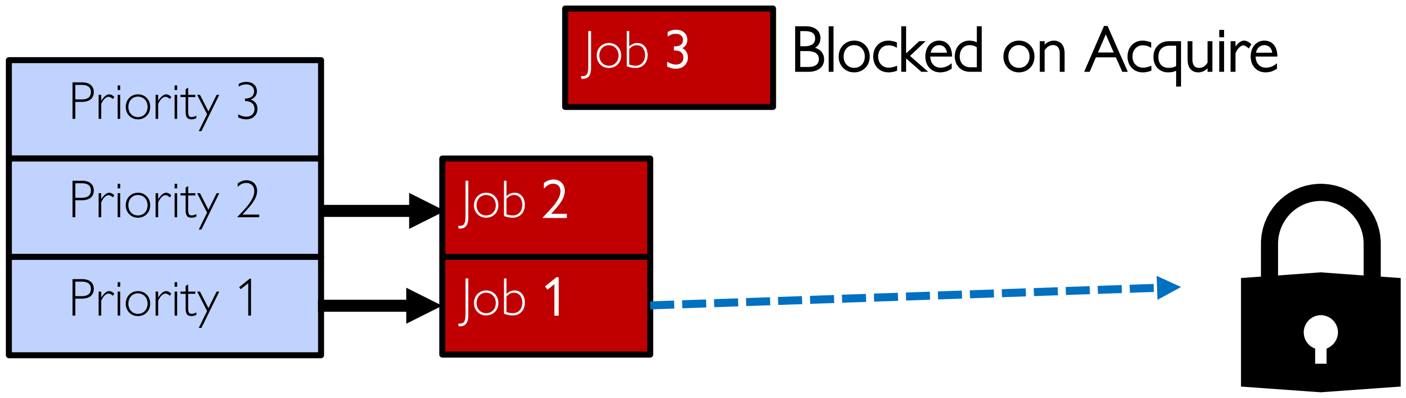
- At this point, which job does the scheduler choose?
- Job 3 (Highest priority)

Priority Inversion



- Job 3 attempts to acquire lock held by Job 1

Priority Inversion



- At this point, which job does the scheduler choose?
- Job 2 (Medium Priority)
- Priority Inversion

Priority Inversion

- Where high priority task is blocked waiting on low priority task
- Low priority one *must* run for high priority to make progress
- Medium priority task can starve a high priority one

- When else might priority lead to starvation or “live lock”?

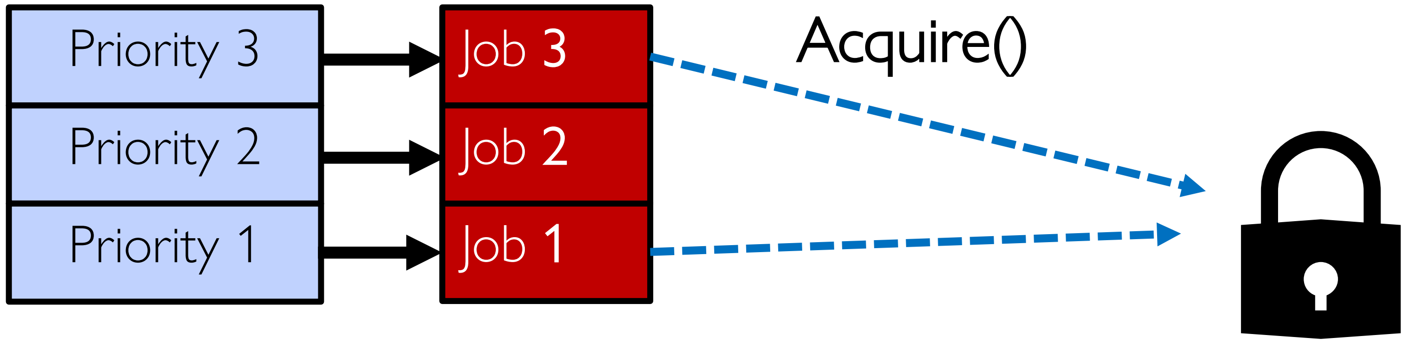
High Priority

```
while (try_lock) {
…
}
```

Low Priority

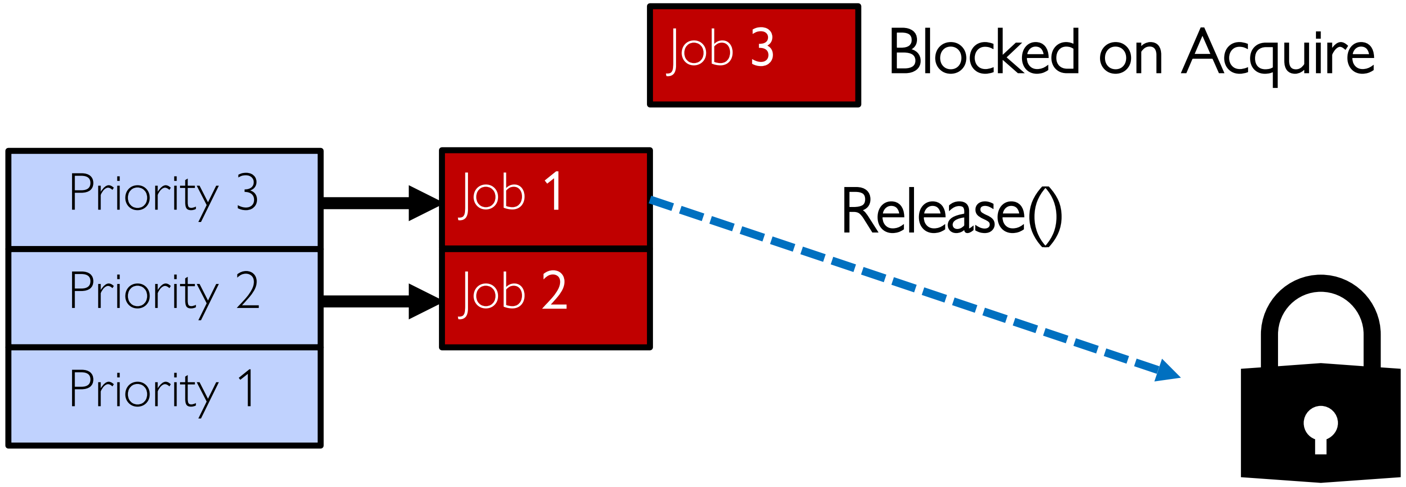
```
lock.acquire(…)
…
lock.release(…)
```

One Solution: Priority Donation/Inheritance



- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

One Solution: Priority Donation/Inheritance



- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

Case Study: Martian Pathfinder Rover

- July 4, 1997 – Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover

- And then…a few days into mission…:
  - Multiple system resets occur to realtime OS (VxWorks)
  - System would reboot randomly, losing valuable time and progress
- Problem? Priority Inversion!
  - Low priority task grabs mutex trying to communicate with high priority task:

Priority 2	Data Distribution Task: needs lock
Priority 1	Lots of random medium stuff
Priority 0	ASI/MET collector: grab lock

  - Realtime watchdog detected lack of forward progress and invoked reset to safe state
    - » High-priority data distribution task was supposed to complete with regular deadline
- Original developers turned off priority donation !

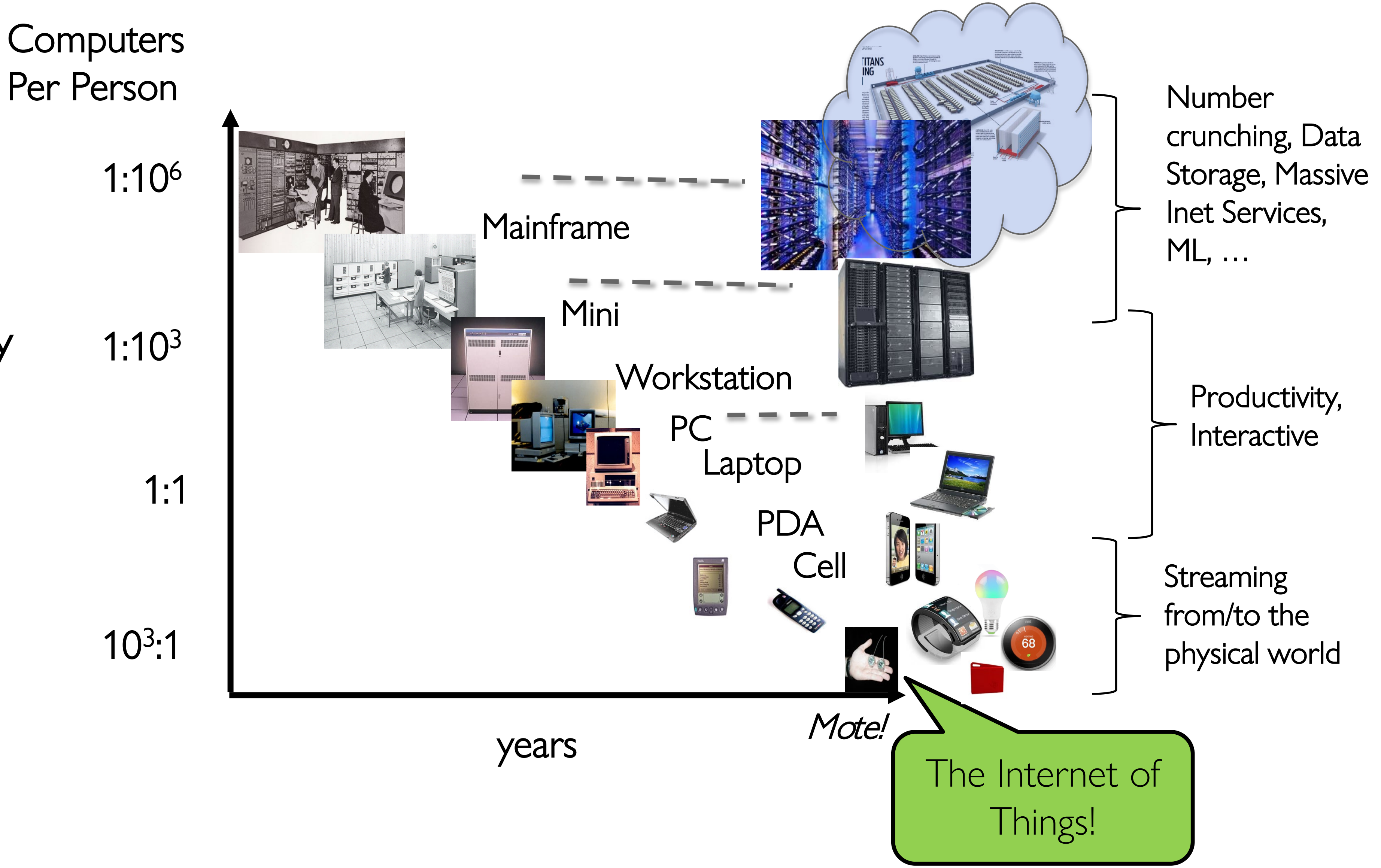
Cause for Starvation: Priorities?

- 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

Recall: Changing Landscape…

Bell's Law: New computer class every 10 years



Changing Landscape of Scheduling

- Priority-based scheduling rooted in "time-sharing"
  - Allocating precious, limited resources across a diverse workload
    - » CPU bound, vs interactive, vs I/O bound
- 80's brought about personal computers, workstations, and servers on networks
  - Different machines of different types for different purposes
  - Shift to fairness and avoiding extremes (starvation)
- 90's emergence of the web, rise of internet-based services, the data-center-is-the-computer
  - Server consolidation, massive clustered services, huge flashcrowds
  - It's about predictability, 95^{th} percentile performance guarantees

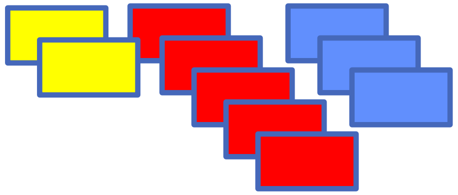
DOES PRIORITIZING SOME JOBS *NECESSARILY* STARVE THOSE THAT AREN'T PRIORITIZED?

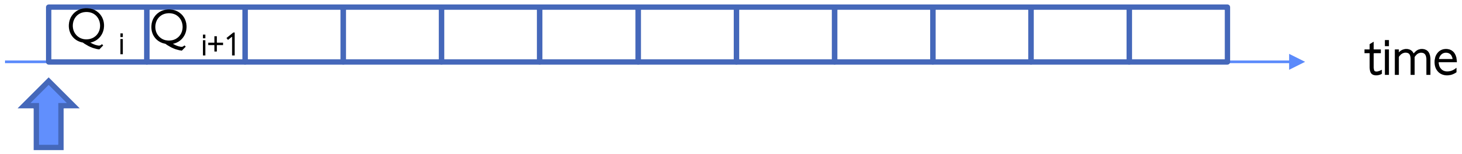
Key Idea: Proportional-Share Scheduling

- 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

- 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)

Recall: Lottery Scheduling





- Given a set of jobs (the mix), provide each with a share of a resource
  - e.g., 50% of the CPU for Job A, 30% for Job B, and 20% for Job C
- Idea: Give out tickets according to the proportion each should receive,
- Every quantum (tick): draw one at random, schedule that job (thread) to run

Unfairness

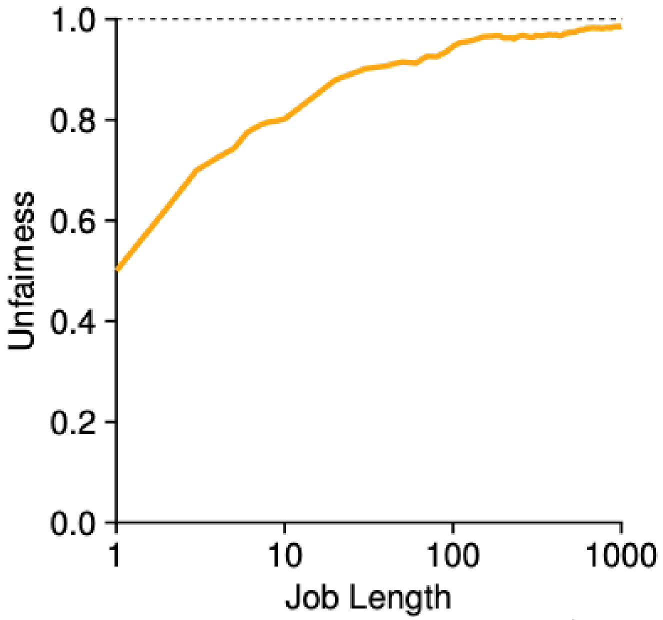


Figure 9.2: **Lottery Fairness Study**

- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%,

  U = finish time of first / finish time of last

- As a function of run time

Stride Scheduling

- Deterministic proportional fair sharing
- "Stride" of each job is $\frac{big\#W}{N_i}$
  - The larger your share of tickets, the smaller your stride
  - Ex: W = 10,000, A=100 tickets, B=50, C=250
  - A stride: 100, B: 200, C: 40
- Each job as a "pass" counter . Scheduler: pick job with lowest *pass*, runs it, add its *stride* to its *pass*
- Low-stride jobs (lots of tickets) run more often
  - Job with twice the tickets gets to run twice as often

Stride Scheduling

W = 10,000, A=200 tickets, B=100 tickets, C=50 tickets

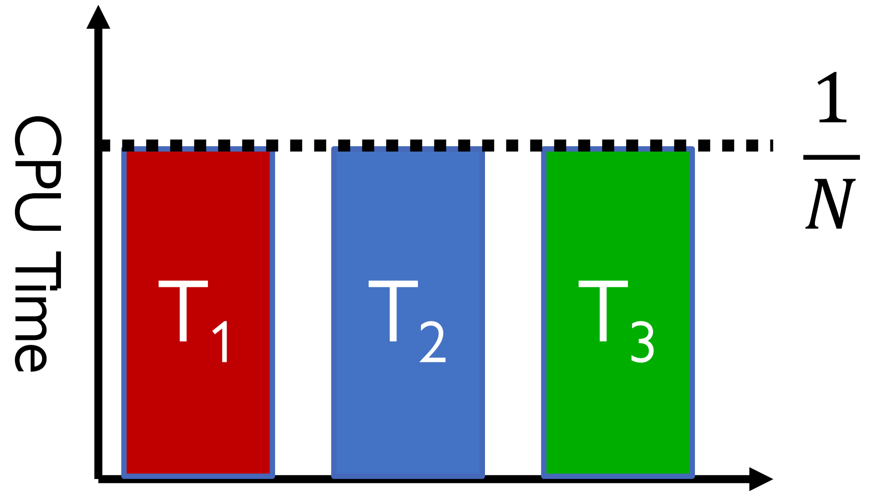
Strides: 50 100 200

Schedule	50	100	100	150	200	200	200
Ready Queue	100	100	150	200	200	200	250
	200	200	200	200	200	250	300

Linux Completely Fair Scheduler (CFS)

- Goal: Each process gets an equal share of CPU
  - *N* threads "simultaneously" execute on $\frac{1}{N}$ of CPU
  - The *model* is somewhat like simultaneous multithreading – each thread gets $\frac{1}{N}$ of the cycles
- In general, can't do this with real hardware
  - OS needs to give out full CPU in time slices
  - Thus, we must use something to keep the threads roughly in sync with one another

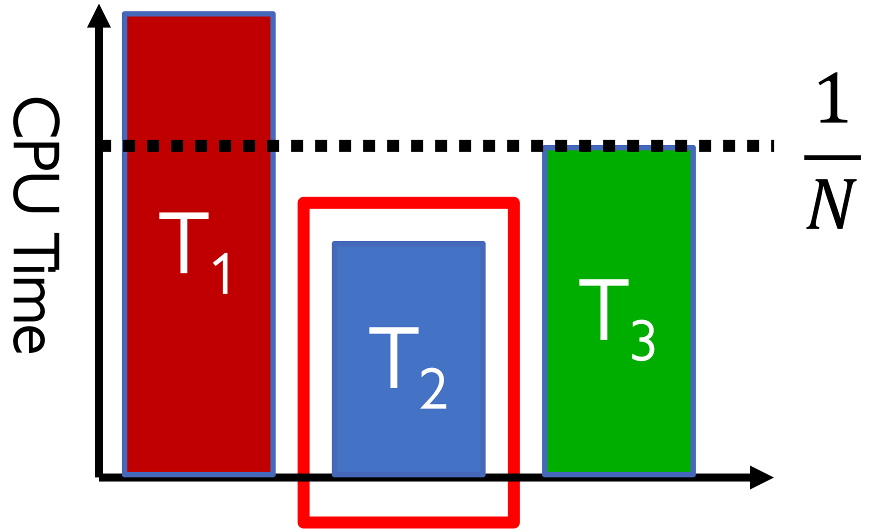
Model: "Perfectly" subdivided CPU:



Linux Completely Fair Scheduler (CFS)

- Basic Idea: track CPU time per thread
- Scheduling Decision:
  - "Repair" illusion of complete fairness
  - Choose thread with minimum CPU time
  - Closely related to Fair Queueing
- Use red-black tree for this…
  - O(log N) to add/remove threads, where N is number of threads
- Sleeping threads don't advance their CPU time, so they get a boost when they wake up again…
  - Get interactivity automatically!

CFS: Average rate of execution = $\frac{1}{N}$:



Linux CFS: Responsiveness/Starvation Freedom

- In addition to fairness, we want **low response time** and starvation freedom
  - Make sure that everyone gets to run at least a bit!

- Constraint 1: *Target Latency*
  - Period of time over which every process gets service
  - Quanta = Target_Latency / n

- Target Latency: 20 ms, 4 Processes
  - Each process gets 5ms time slice

- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice (!!!)
  - Recall Round-Robin: large context switching overhead if slice gets to small

Linux CFS: Throughput

- Goal: Throughput
  - Avoid excessive overhead
- Constraint 2: Minimum Granularity
  - Minimum length of any time slice

- Target Latency 20 ms, Minimum Granularity 1 ms, 200 processes
  - Each process gets 1 ms time slice

Aside: Priority in Unix – Being Nice

- The industrial operating systems of the 60s and 70's provided priority to enforced desired usage policies.
  - When it was being developed at Berkeley, instead it provided ways to "be nice".
- `nice` values range from -20 to 19
  - Negative values are "not nice"
  - If you wanted to let your friends get more time, you would nice up your job
- Scheduler puts higher nice-value tasks (lower priority) to sleep more …
  - In O(1) scheduler, this translated fairly directly to priority (and time slice)
- How does this idea translate to CFS?
  - Change the rate of CPU cycles given to threads to change relative priority

Linux CFS: Proportional Shares

- How to we achieve proportional fair sharing?
  - Allow different threads to have different *rates* of execution (cycles/time)

- Use weights! Key Idea: Assign a weight w_i to each process *I* to compute the switching quanta Q_i
  - Basic equal share: $Q_i = \textbf{Target Latency} \cdot \frac{1}{N}$
  - Weighted Share: $Q_i = \left(w_i / \sum_p w_p \right) \cdot \textbf{Target Latency}$

- Reuse **`nice`** value to reflect share, rather than priority,
  - Remember that lower nice value ⇒ higher priority
  - CFS uses nice values to scale weights exponentially: Weight=1024/(1.25)^{nice}

- So, we use "Virtual Runtime" instead of CPU time

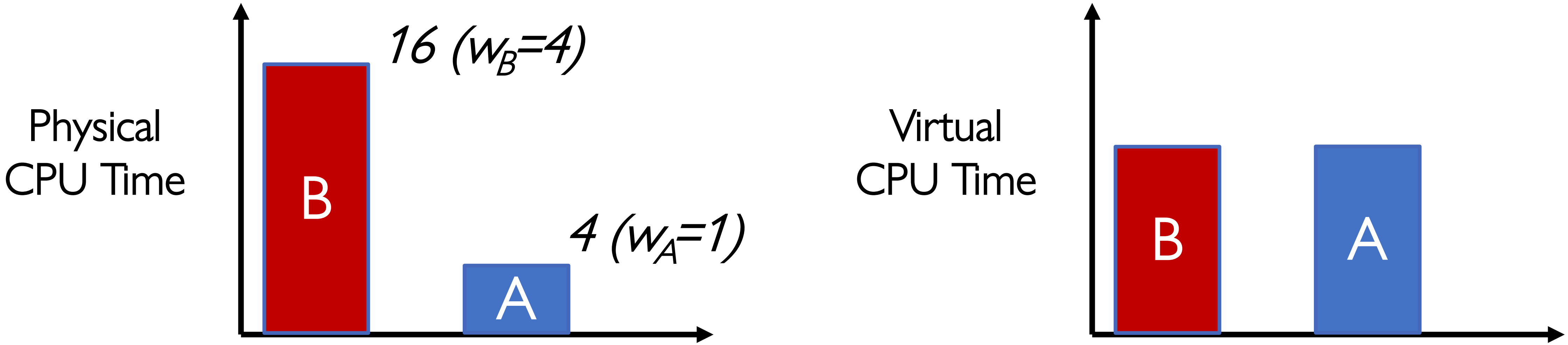
Example: Linux CFS: Proportional Shares

- Target Latency = 20ms
- Minimum Granularity = 1ms

- Example: Two CPU-Bound Threads
  - Thread A has weight 1
  - Thread B has weight 4

- Time slice for A? 4 ms
- Time slice for B? 16 ms

Linux CFS: Proportional Shares



- Track a thread's *virtual* runtime rather than its true physical runtime
  - Higher weight: Virtual runtime increases more slowly
  - Lower weight: Virtual runtime increases more quickly

Linux CFS: Proportional Shares

- Scheduler's Decisions are based on Virtual CPU Time
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - O(1) time to find next thread to run (top of heap!)
  - O(log N) time to perform insertions/deletions
    - » Cash the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).

Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

Summary (1 of 2)

- Scheduling Goals:
  - Minimize Response Time (e.g. for human interaction)
  - Maximize Throughput (e.g. for large computations)
  - Fairness (e.g. Proper Sharing of Resources)
  - Predictability (e.g. Hard/Soft Realtime)
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Summary (2 of 2)

- Realtime Schedulers such as EDF
  - Guaranteed behavior by meeting deadlines
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)
- Stride Scheduling
  - Always fair, unlike lotter scheduling:
- Linux O(1) scheduler
  - Scales as number of processes grows
  - Became overly complex because of heuristics
- Linux CFS Scheduler: Fair fraction of CPU
  - Approximates an "ideal" multitasking processor
  - Practical example of "Fair Queueing"