# Operating Systems
# (Honor Track)

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

Xin Jin

Spring 2023

Acknowledgments: Ion Stoica, Berkeley CS 162

# Recap: First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also “First In, First Out” (FIFO) or “Run until done”
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks

- Example:

| Process | Burst Time |
| --- | --- |
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

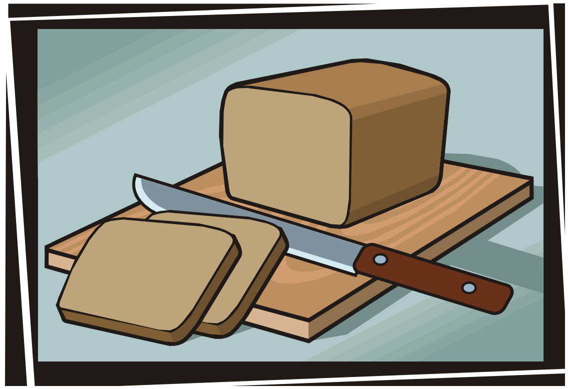
  - Suppose processes arrive in the order: $P_1$ , $P_2$ , $P_3$ The Gantt Chart for the schedule is:


  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process stuck behind long process

# Recap: Round Robin (RR) Scheduling

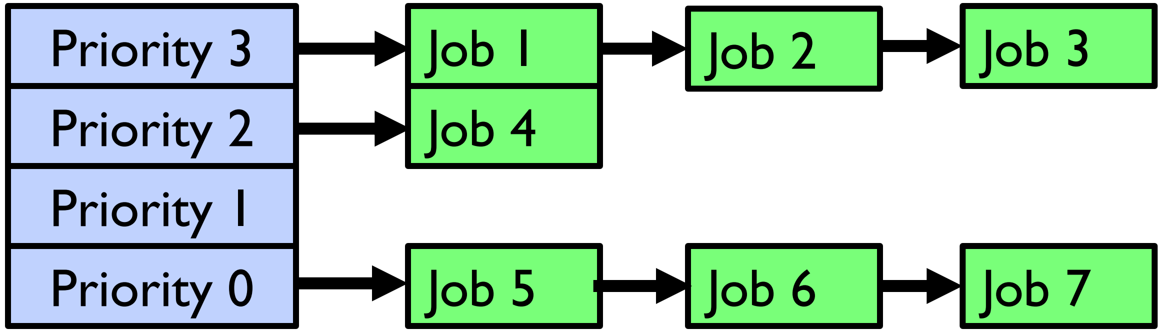
- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin Scheme: Preemption!
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - $n$ processes in ready queue and time quantum is $q \Rightarrow$
    - » Each process gets $1/n$ of the CPU time
    - » In chunks of at most $q$ time units
    - » No process waits more than $(n-1)q$ time units

# Recap: Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - » Waiting time suffers
  - What if infinite ($\infty$)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!
- Actual choices of time slice:
  - Initially, UNIX time slice one second:
    - » Worked ok when UNIX was used by one or two people.
    - » What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput
    - » Typical time slice today is between 10ms – 100ms

# Recap: Handling Differences in Importance: Strict Priority Scheduling




- Execution Plan
  - Always execute highest-priority runable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation:
    - » Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - » Happens when low priority task has lock needed by high-priority task
    - » Usually involves third, intermediate priority task preventing high-priority task from running
- How to fix problems?
  - Dynamic priorities: adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

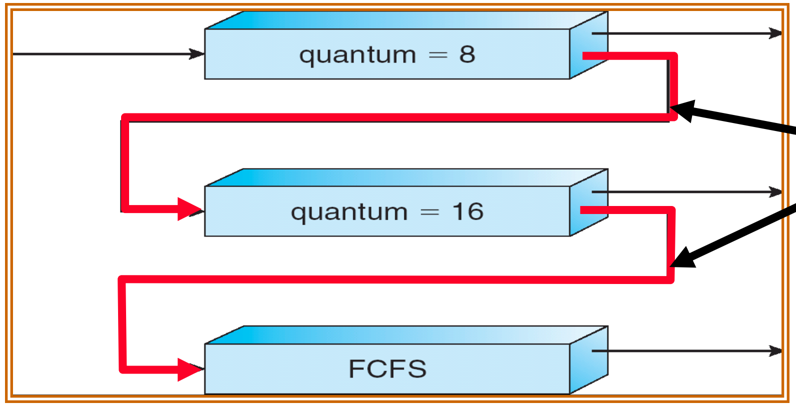
# Recap: What if we Knew the Future?

- Could we always mirror best FCFS?
- 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 completion time

# Recap: Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

# Recap: Multi-Level Feedback Scheduling




Long-Running Compute Tasks Demoted to Low Priority

- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next: 2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

# Multi-Core Scheduling

- Algorithmically, not a huge difference from single-core scheduling

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

- *Affinity scheduling*: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  - Cache reuse

# Spinlocks for multiprocessing

- Spinlock implementation:

```
int value = 0; // Free
Acquire() {
   while (test&set(&value)) {}; // spin while busy
}
Release() {
   value = 0;                   // atomic store
}
```

- Spinlock doesn't put the calling thread to sleep—it just busy waits
  - When might this be preferable?
    - » Waiting for limited number of threads at a barrier in a multiprocessing (multicore) program
    - » Wait time at barrier would be greatly increased if threads must be woken inside kernel
- Every `test&set()` is a write, which makes value ping-pong around between core-local caches
  - So – really want to use `test&test&set()` !
- The extra read eliminates the ping-ponging issues:

```
// Implementation of test&test&set():
Acquire() {
   do {
      while(value);            // wait until might be free
   } while (test&set(&value)); // exit if acquire lock
}
```

# Gang Scheduling and Parallel Applications

- When multiple threads work together on a multi-core system, try to schedule them together
  - Makes spin-waiting more efficient (inefficient to spin-wait for a thread that's suspended)

- Alternative: OS informs a parallel program how many processors its threads are scheduled on (*Scheduler Activations*)
  - Application adapts to number of cores that it has scheduled
  - "Space sharing" with other parallel programs can be more efficient, because parallel speedup is often sublinear with the number of cores

# So, Does the OS Schedule Processes or Threads?

- Many textbooks use the “old model”—one thread per process
- Usually it's really: **threads** (e.g., in Linux)

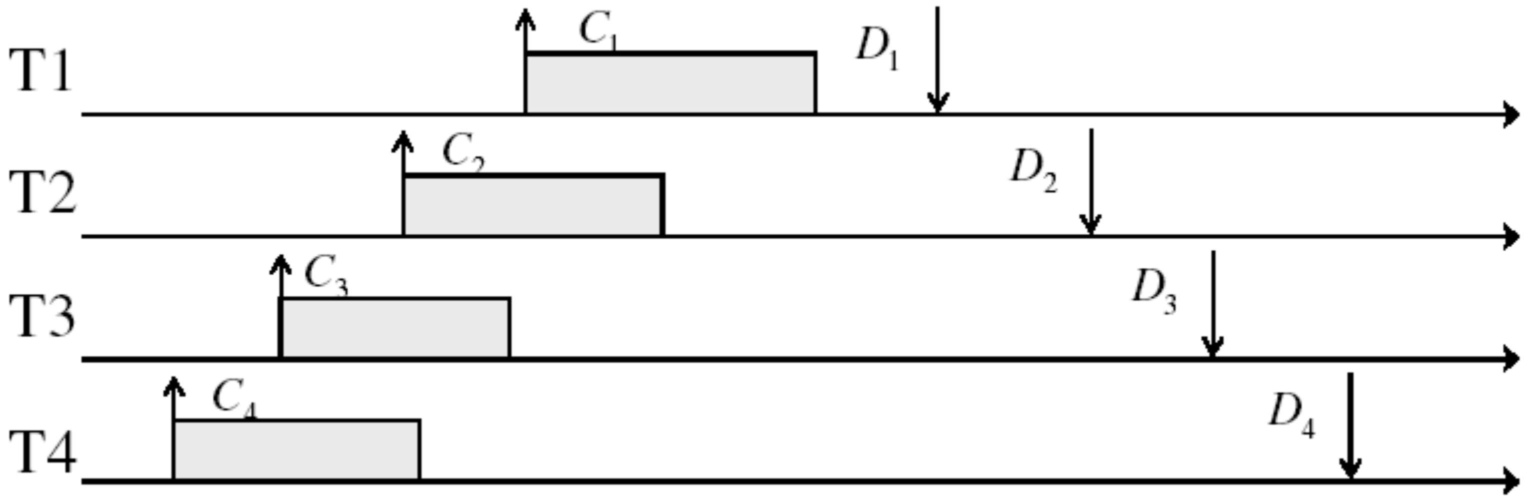
- One point to notice: switching threads vs. switching processes incurs different costs:
  - Switch threads: Save/restore registers
  - Switch processes: Change active address space too!
    - Expensive
    - Disrupts caching

# Real-Time Scheduling

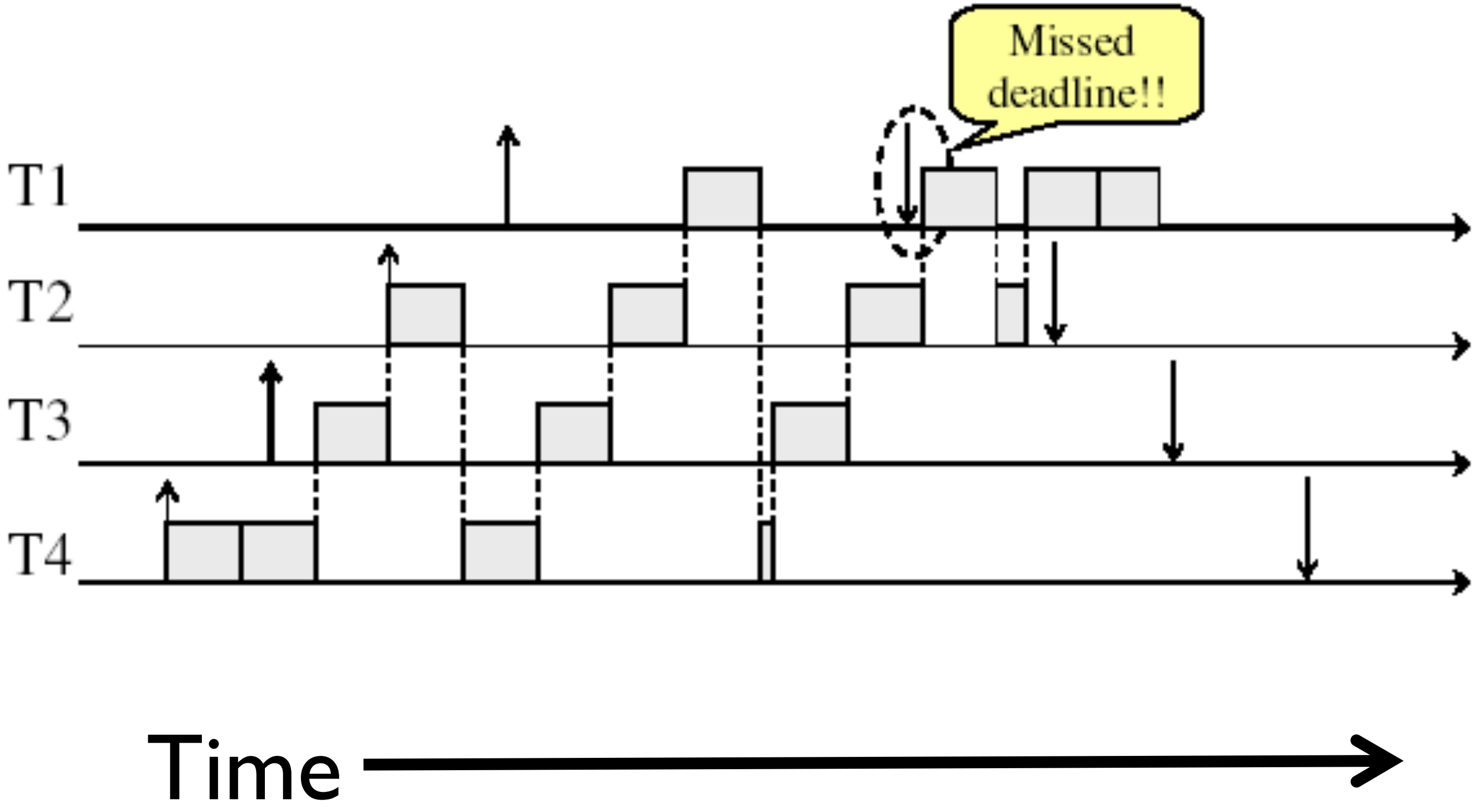
- Goal: Predictability of Performance!
  - We need to predict with confidence worst case response times for systems!
  - In RTS, performance guarantees are:
    - » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - » System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing *predictability*; does not equal fast computing!!!
- Hard real-time: for time-critical safety-oriented systems
  - Meet all deadlines (if at all possible)
  - Ideally: determine in advance if this is possible (admission control)
  - Earliest Deadline First (EDF)
    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:

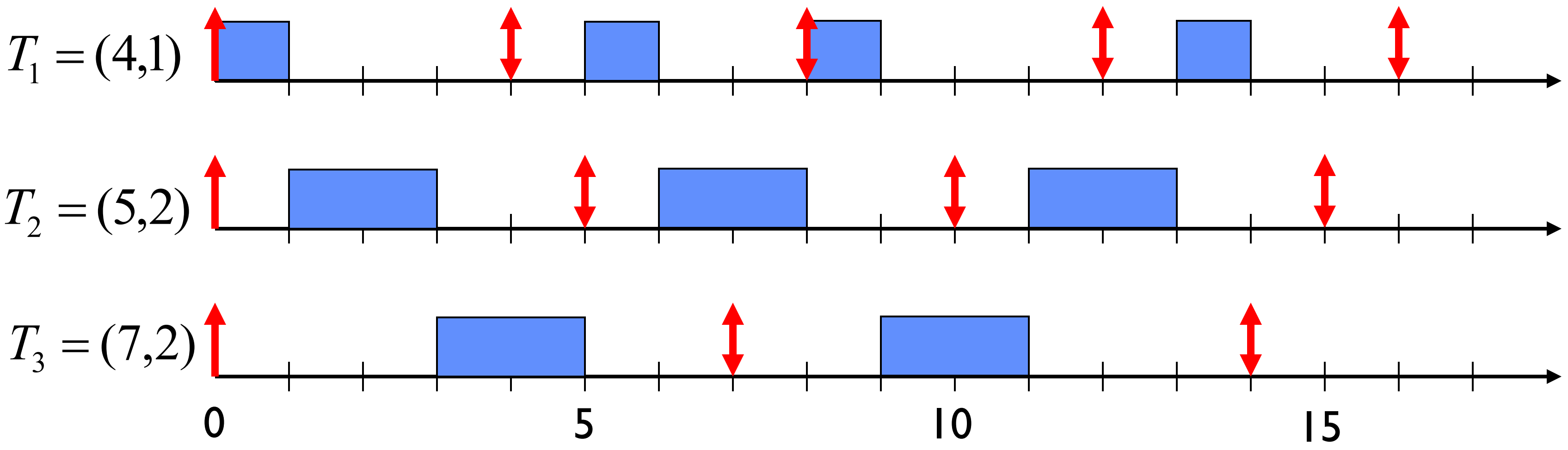


# Example: Round-Robin Scheduling Doesn't Work



# Earliest Deadline First (EDF)

- Tasks $i$ is periodic with period $P_i$ and computation $C_i$ 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

- Even EDF won't work if you have too many tasks
- For $n$ tasks with computation time $C_i$ and deadline $D_i$, a feasible schedule exists if:

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

$$\frac{1}{4} + \frac{2}{5} + \frac{2}{7} = 0.936 \leq 1$$

# Ensuring Progress

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

- Starvation ≠ Deadlock
  - Deadlock: cyclic requests for resources

- 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

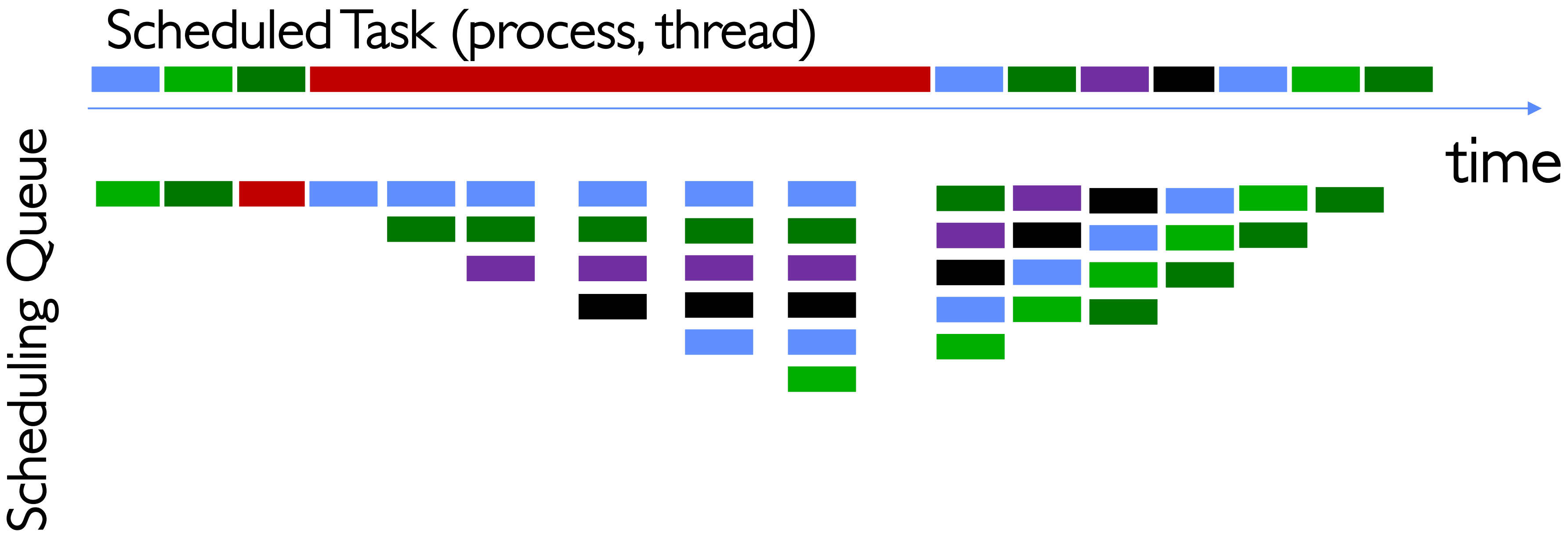
- A non-work-conserving scheduler could trivially lead to starvation

- 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.

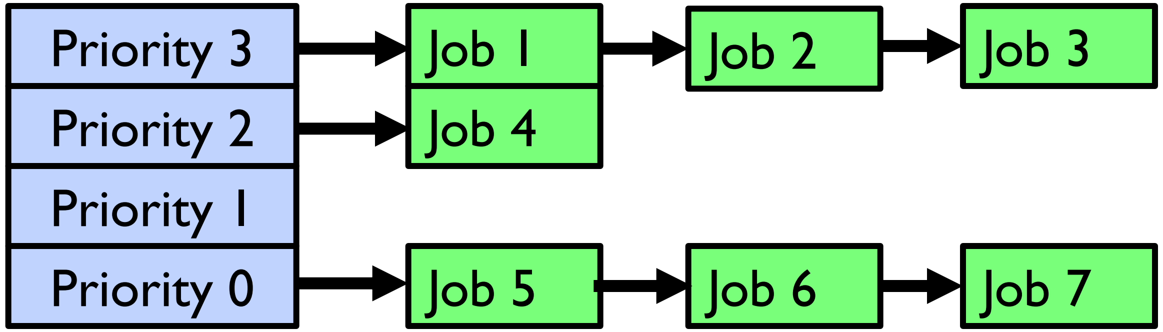
# Is Round Robin (RR) Prone to Starvation?

- 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…

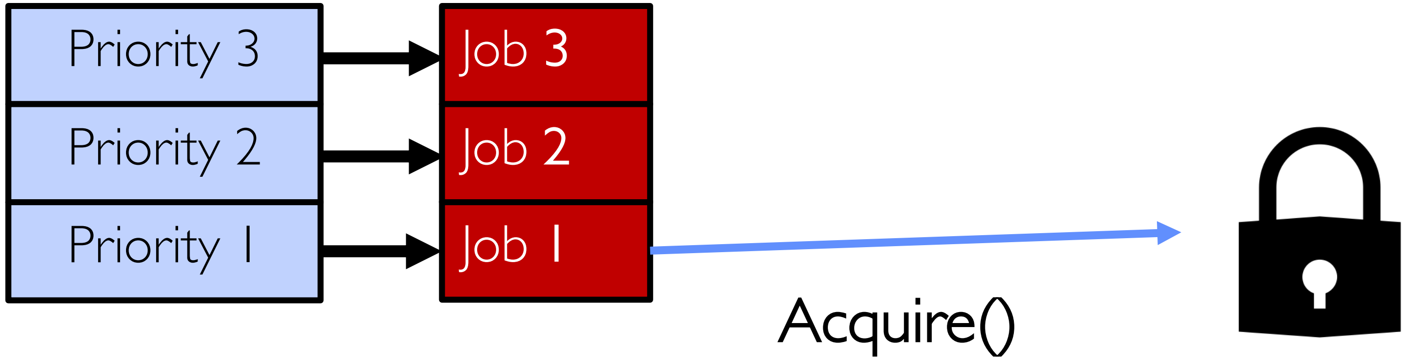
# Is Priority Scheduling Prone to Starvation?

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



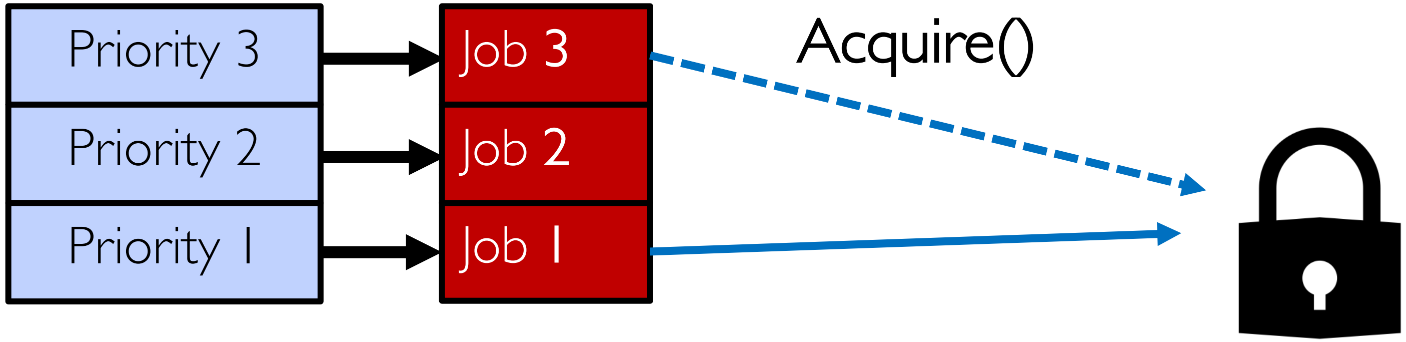

- But there are more serious problems as well…
  - Priority inversion: even high priority threads might become starved

# Priority Inversion



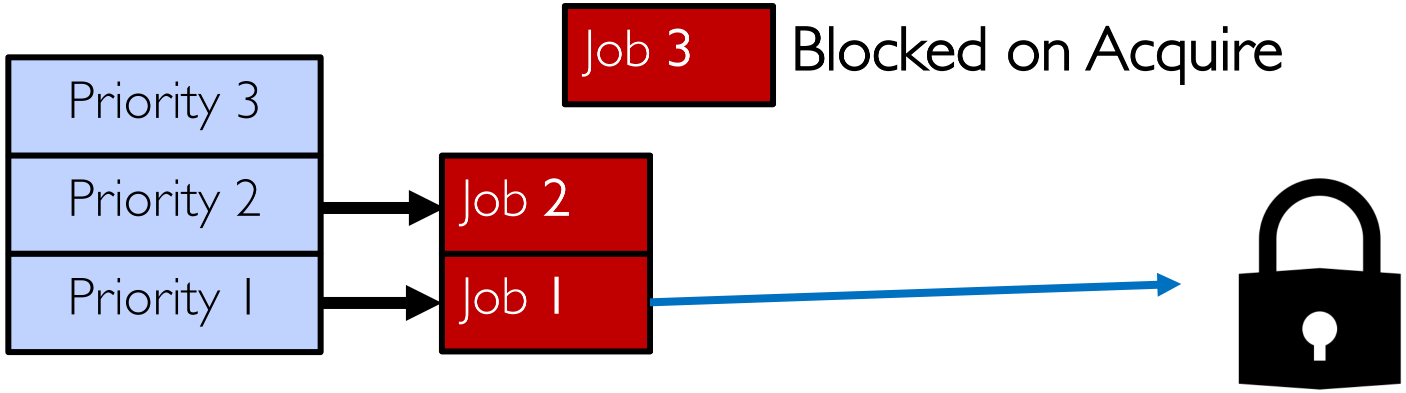

- At this point, which job does the scheduler choose?
- Job 3 (Highest priority)

# Priority Inversion




- Job 3 attempts to acquire lock held by Job 1

# Priority Inversion



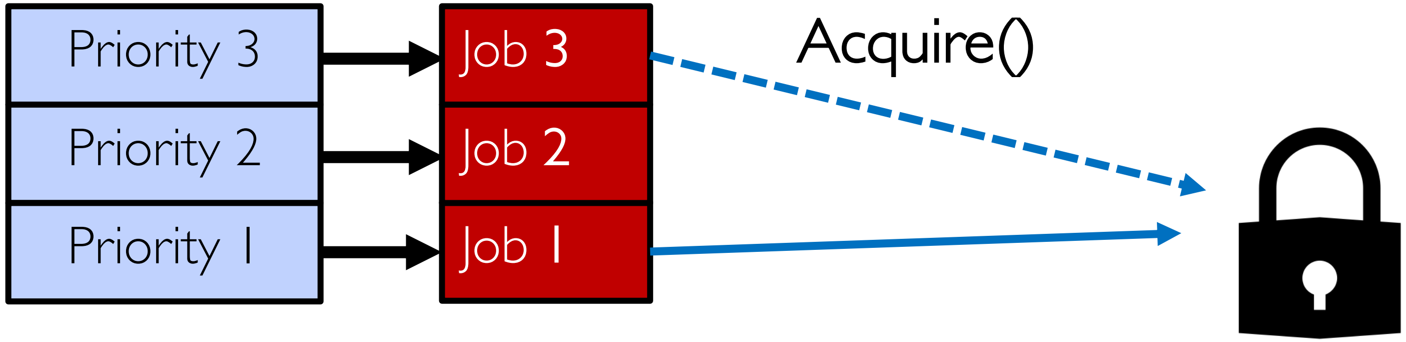

- At this point, which job does the scheduler choose?
- Job 2 (Medium Priority)
- Priority Inversion

# 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

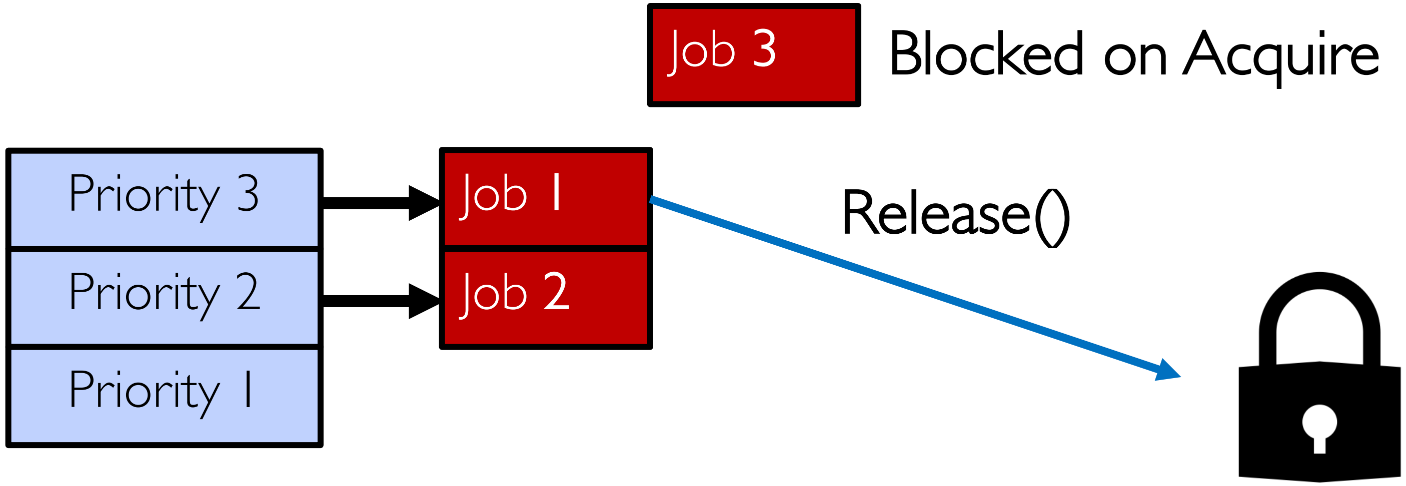
- When else might priority lead to starvation or "live lock"?

# One Solution: Priority Donation/Inheritance



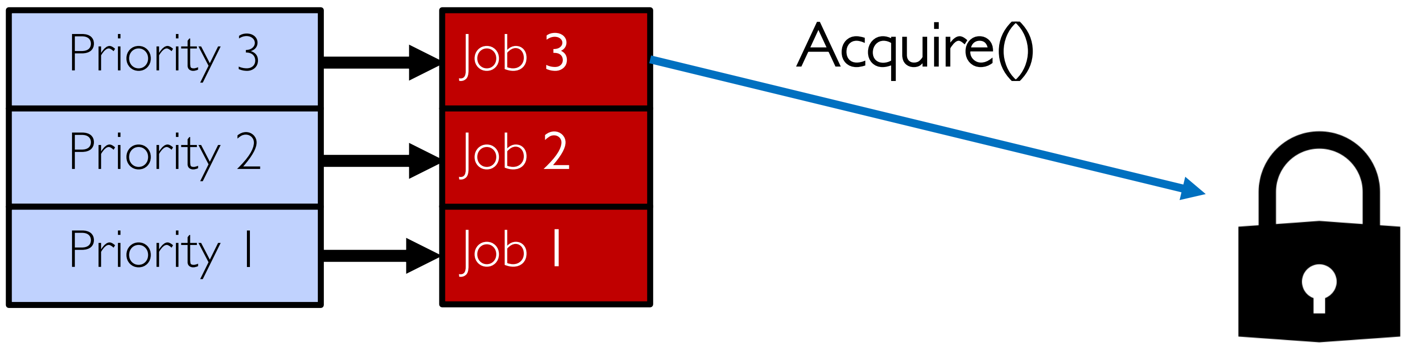

- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

# One Solution: Priority Donation/Inheritance




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

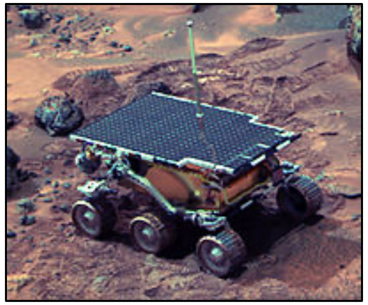
# One Solution: Priority Donation/Inheritance



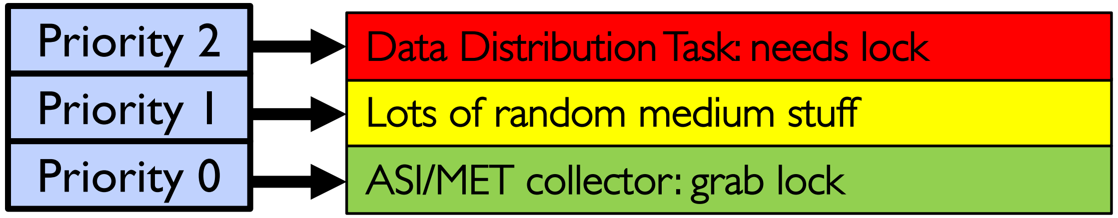

- Job 1 completes critical section and releases lock
- Job 3 acquires lock, runs again

# Case Study: Martian Pathfinder Rover

- July 4, 1997 – Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover
  - Novel delivery mechanism: inside air-filled balloons bounced to stop on the surface from orbit!

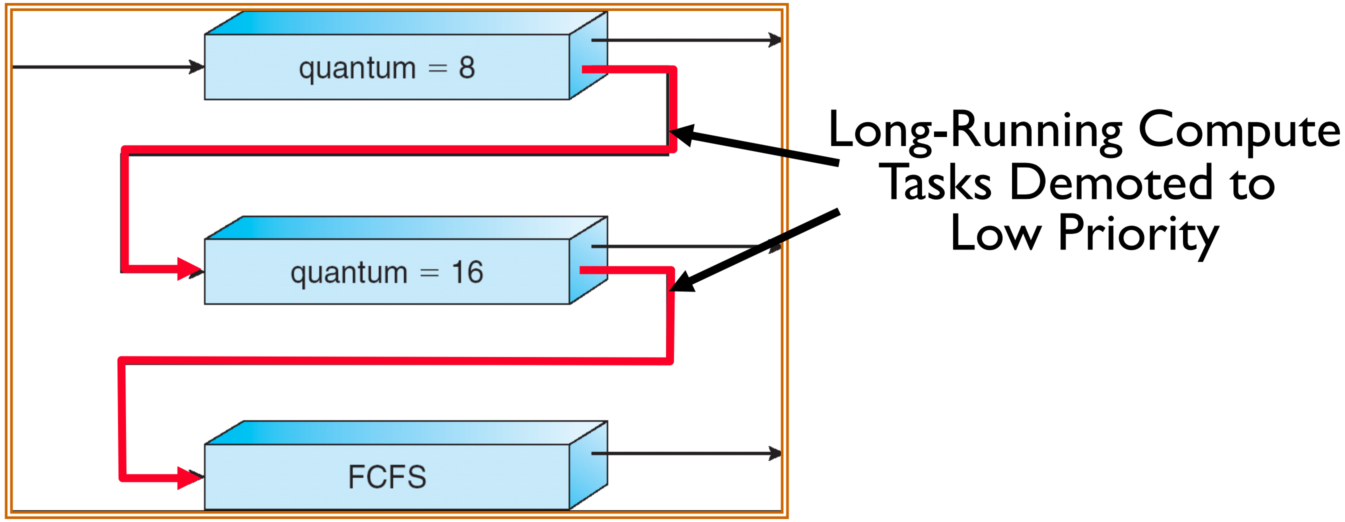


- And then…a few days into mission…:
  - Multiple system resets occur to realtime OS (VxWorks)
  - System would reboot randomly, losing valuable time and progress
- Problem? Priority Inversion!
  - Low priority task grabs mutex trying to communicate with high priority task




  - 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
- Solution: Turn priority donation back on and upload fixes!
- Original developers turned off priority donation (also called priority inheritance)
  - Worried about performance costs of donating priority!

# Are SRTF and MLFQ Prone to Starvation?




- 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

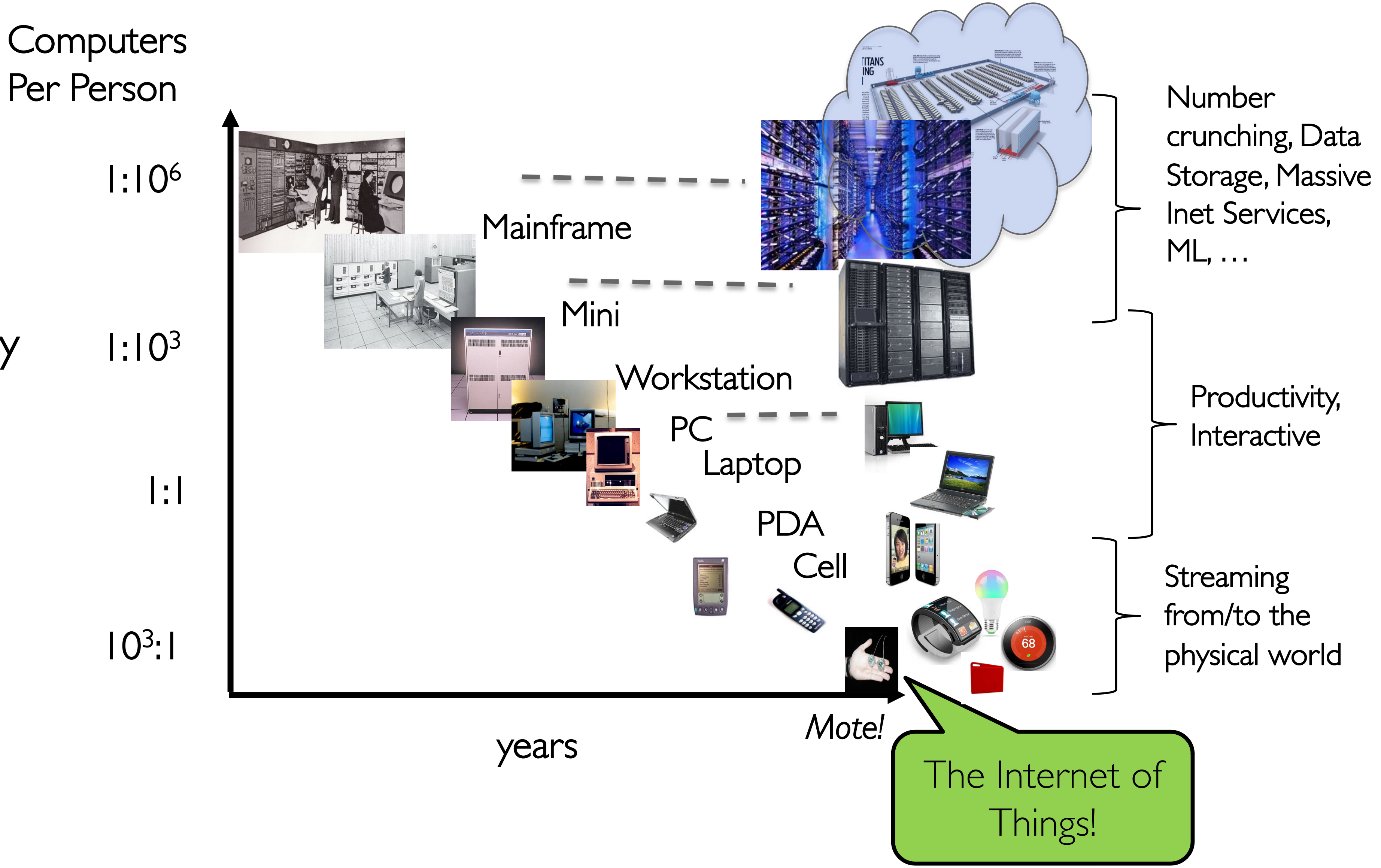
# Cause for Starvation: Priorities?

- Most of 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

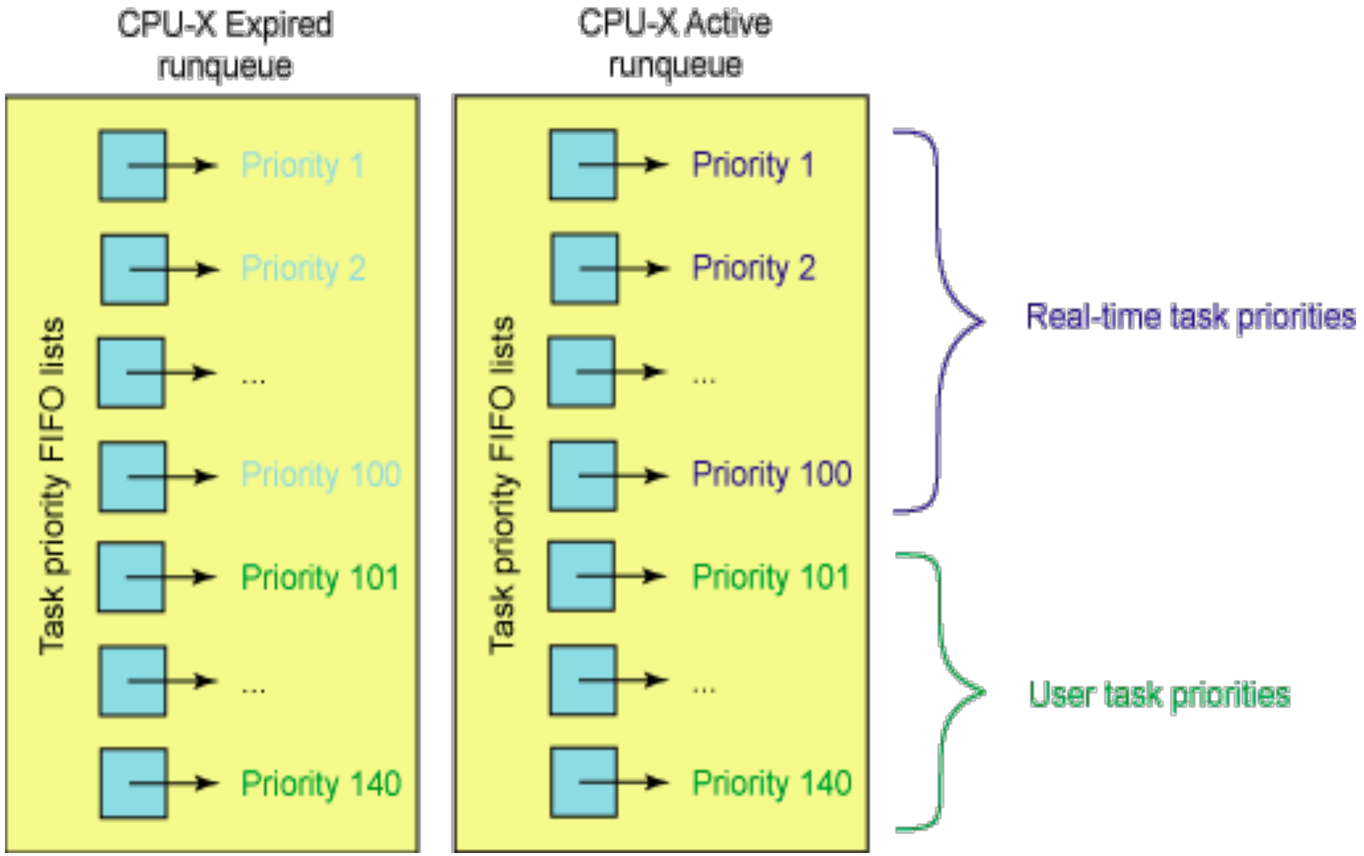
# Priority in Unix – Being Nice

- The industrial operating systems of the 60s and 70s 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)

# Case Study: Linux O(1) Scheduler


- Priority-based scheduler: 140 priorities
  - 40 for "user tasks" (set by "nice"), 100 for "Realtime/Kernel"
  - Lower nice value $\Rightarrow$ higher priority
  - Higher nice value $\Rightarrow$ lower priority
  - All algorithms O(1)
    - » Timeslices/priorities/interactivity credits all compute when job finishes time slice
    - » 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues: "active" and "expired"
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
- 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

# Linux O(1) Scheduler




- Lots of ad-hoc heuristics
  - Try to boost priority of I/O-bound tasks
  - Try to boost priority of starved tasks

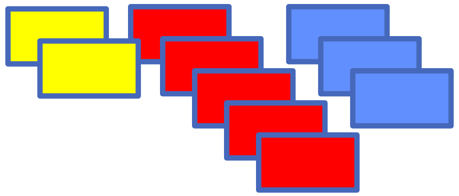
# O(1) Scheduler Continued

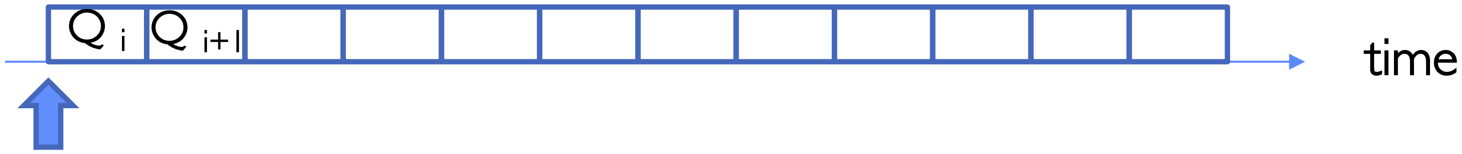
- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    - » P→sleep_avg = (sleep_time – run_time) x coefficient
    - » Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - » Earned when a task sleeps for a “long” time
    - » Spend when 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…
- 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

# Proportional-Share Scheduling

- Instead using priorities, 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)

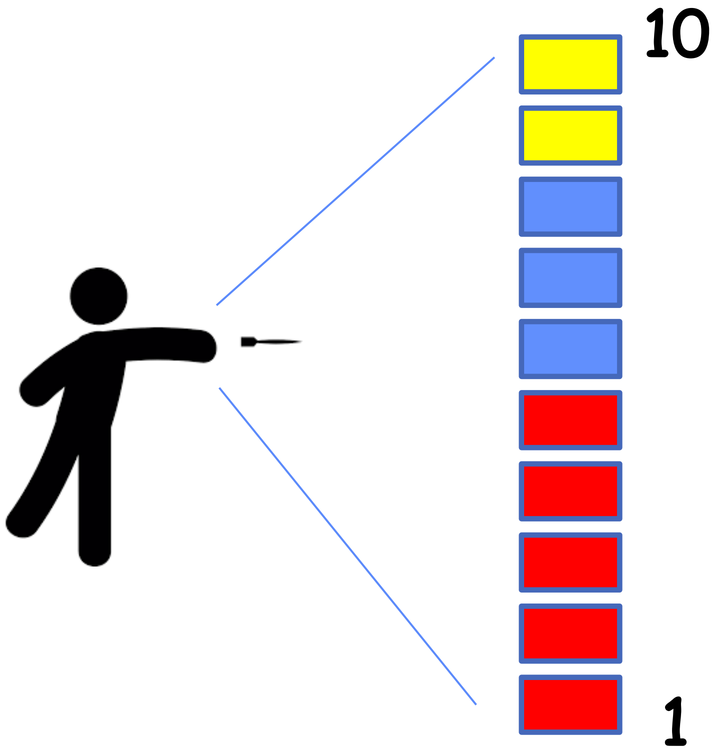
# Recall: Lottery Scheduling






- 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

# Lottery Scheduling: Simple Mechanism

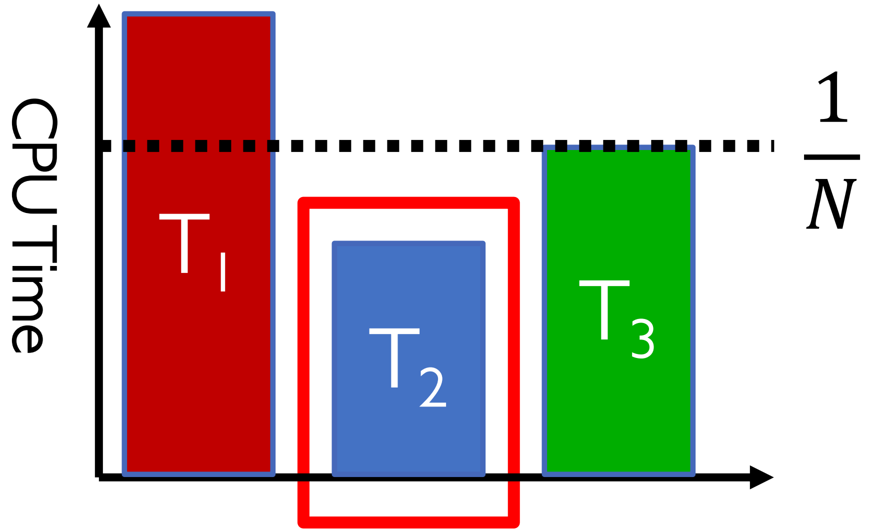



- $N_{ticket} = \sum N_i$
- Pick a number $d$ in $1 \; .. \; N_{ticket}$ as the random "dart"
- Jobs record their $N_i$ of allocated tickets
- Order them by $N_i$
- Select the first j such that $\sum N_i$ up to j exceeds *d*.

# Linux Completely Fair Scheduler (CFS)

- Basic Idea: track CPU time per thread and schedule threads to match up average rate of execution
- Scheduling Decision:
  - "Repair" illusion of complete fairness
  - Choose thread with minimum CPU time
  - Closely related to Fair Queueing
- Use a heap-like scheduling queue 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 waiting 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  (n: number of processes)
- 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, 100 processes
  - Each process gets 1 ms time slice

# Linux CFS: Proportional Shares

- What if we want to give more CPU to some and less to others in CFS (proportional share) ?
  - Allow different threads to have different *rates* of execution (cycles/time)
- Use weights: assign a weight $w_i$ to each process $i$ to compute the switching quanta $Q_i$
  - Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$
  - Weighted Share: $Q_i = \left(w_i / \sum_p w_p\right) \cdot \text{Target Latency}$
- Reuse `nice` value to reflect share, rather than priority,
  - Remember that lower nice value $\Rightarrow$ higher priority
  - CFS uses nice values to scale weights exponentially: Weight=1024/(1.25)$^{\text{nice}}$
    - » Two CPU tasks separated by nice value of 5 $\Rightarrow$
      Task with lower nice value has 3 times the weight, since $(1.25)^5 \approx 3$

# Choosing the Right Scheduler

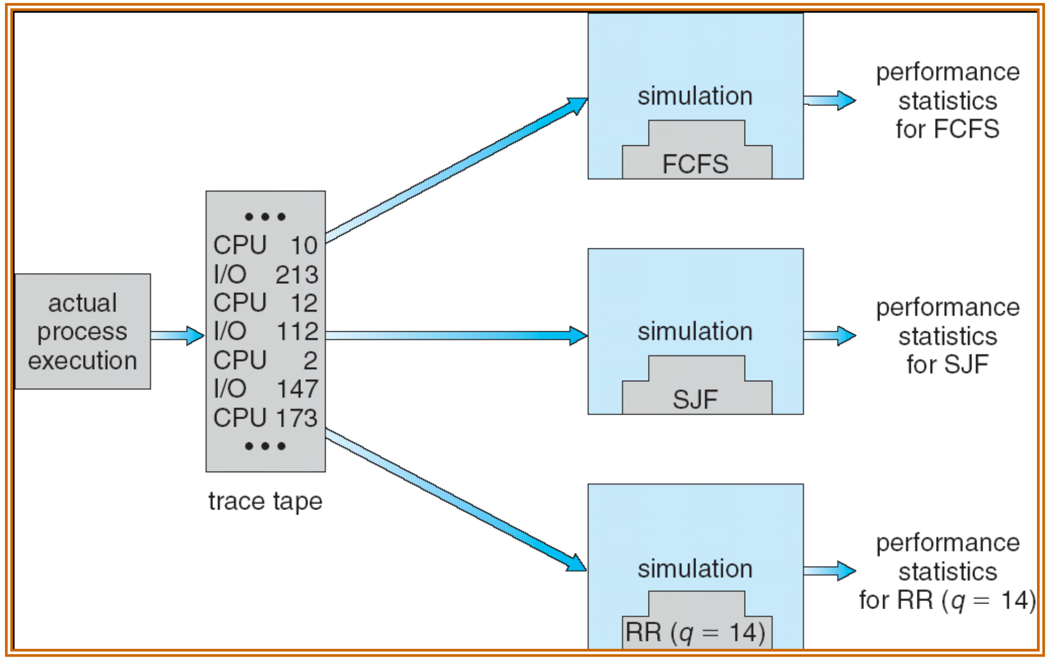
| I Care About: | Then Choose: |
|---|---|
| CPU Throughput | |
| Avg. Completion Time | |
| I/O Throughput | |
| Fairness (CPU Time) | |
| Fairness<br>(Wait Time to Get CPU) | |
| Meeting Deadlines | |
| Favoring Important Tasks | |

# Choosing the Right Scheduler

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

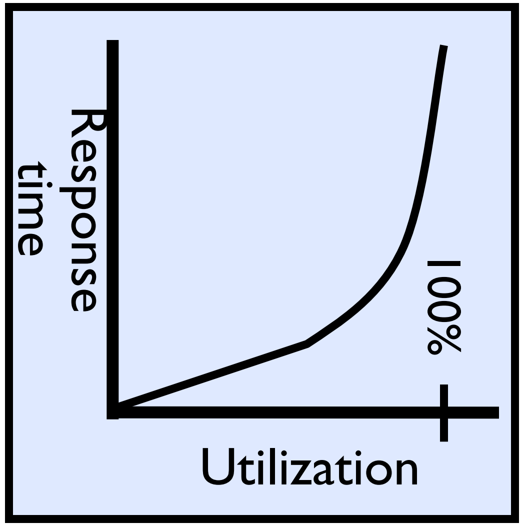
# How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data
  - Most flexible/general



# A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Perhaps you're paying for worse response time in reduced productivity, customer angst, etc…
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%
- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve



# Summary (1 of 2)

- Scheduling Goals:
  - Minimize Completion 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$\Rightarrow$more tokens)
- Linux CFS Scheduler: Fair fraction of CPU
  - Approximates an "ideal" multitasking processor
  - Practical example of "Fair Queueing"