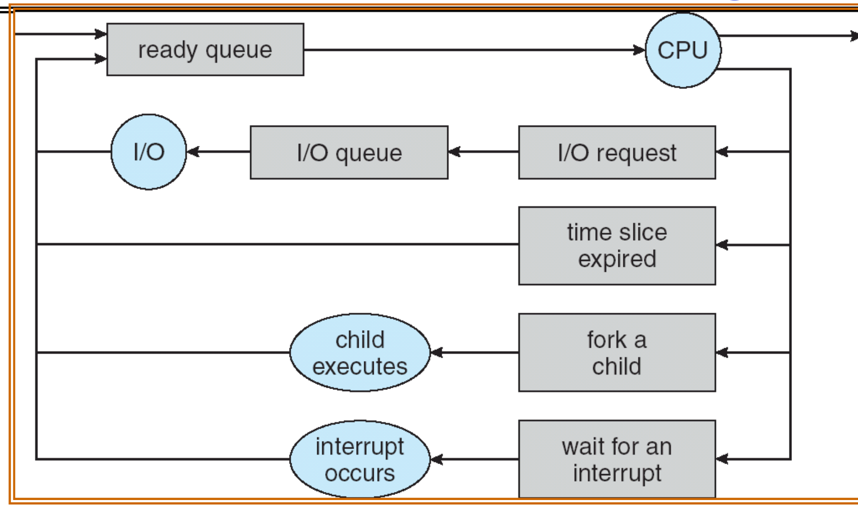
# CS162
# Operating Systems and Systems Programming
# Lecture 11

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

February 24, 2022

Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu

# 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:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- 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:

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|---|

0 20 28 48 68 88 108 112 125 145 153

  – 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

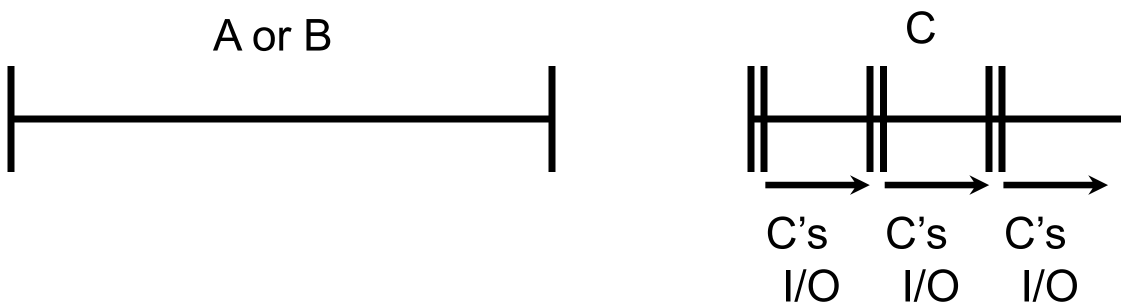
  – Average completion time = (125+28+153+112)/4 = 104½

- Thus, Round-Robin Pros and Cons:
  – Better for short jobs, Fair (+)
  – Context-switching time adds up for long jobs (-)

# Recall: What if we Knew the Future?

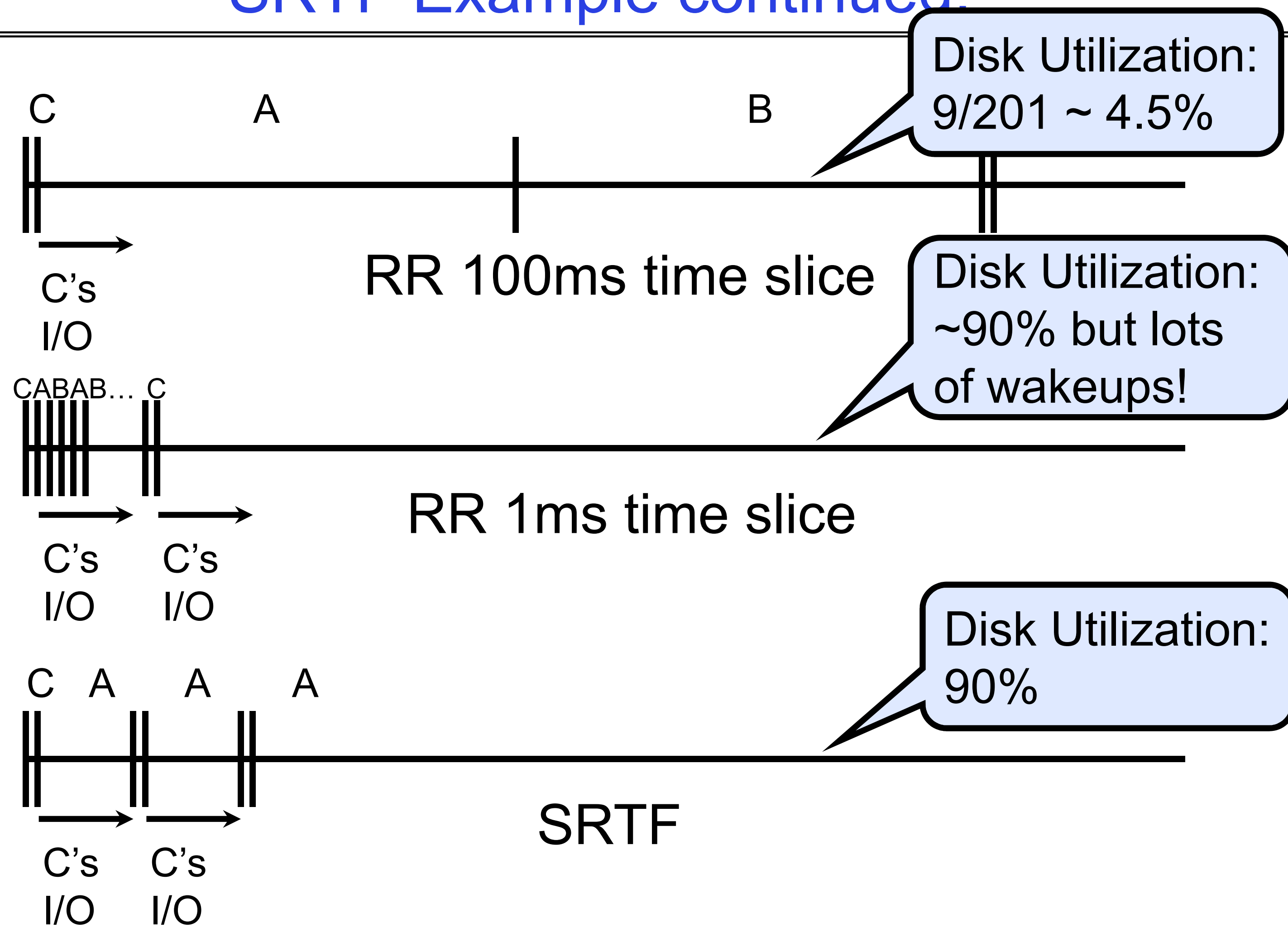
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  – Run whatever job has least amount of computation to do
  – Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
  – Preemptive version of SJF: if job arrives and has a shorter time to completion 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 response time

# Example to illustrate benefits of SRTF




- Three jobs:
  - A, B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FCFS:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

# SRTF Example continued:

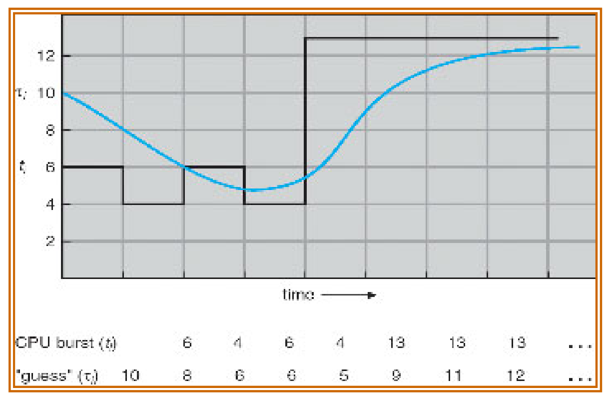


# SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: hard to predict job's runtime even for non-malicious users
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

# Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  – CPU scheduling, in virtual memory, in file systems, etc
  – Works because programs have predictable behavior
    » If program was I/O bound in past, likely in future
    » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  – Use an estimator function on previous bursts:
  Let tn-1, tn-2, tn-3, etc. be previous CPU burst lengths.
  Estimate next burst $\tau$n = f(tn-1, tn-2, tn-3, …)
  – Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  – For instance, exponential averaging
  $\tau$n = $\alpha$tn-1+(1-$\alpha$)$\tau$n-1
  with (0<$\alpha$≤1)



# 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

# Lottery Scheduling Example (Cont.)

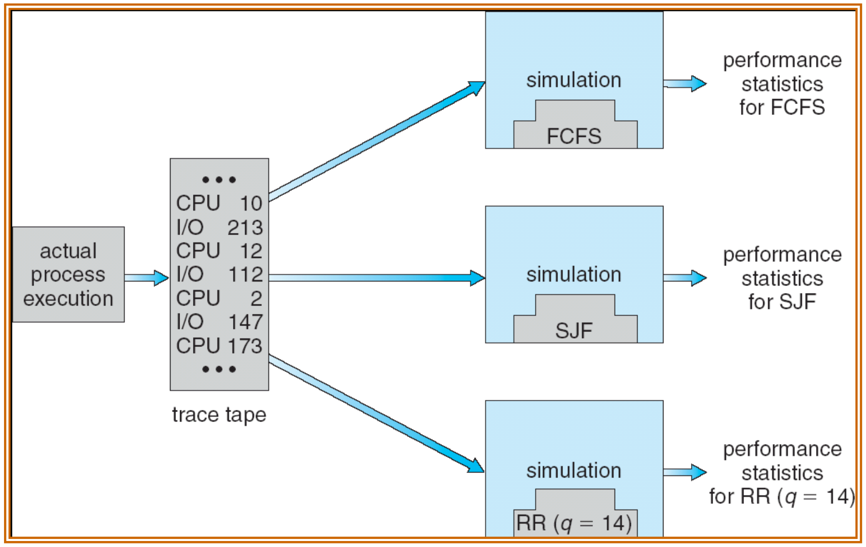
- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

| # short jobs/ # long jobs | % of CPU each short jobs gets | % of CPU each long jobs gets |
|---|---|---|
| 1/1 | 91% | 9% |
| 0/2 | N/A | 50% |
| 2/0 | 50% | N/A |
| 10/1 | 9.9% | 0.99% |
| 1/10 | 50% | 5% |

  - What if too many short jobs to give reasonable response time?
    - » If load average is 100, hard to make progress
    - » One approach: log some user out

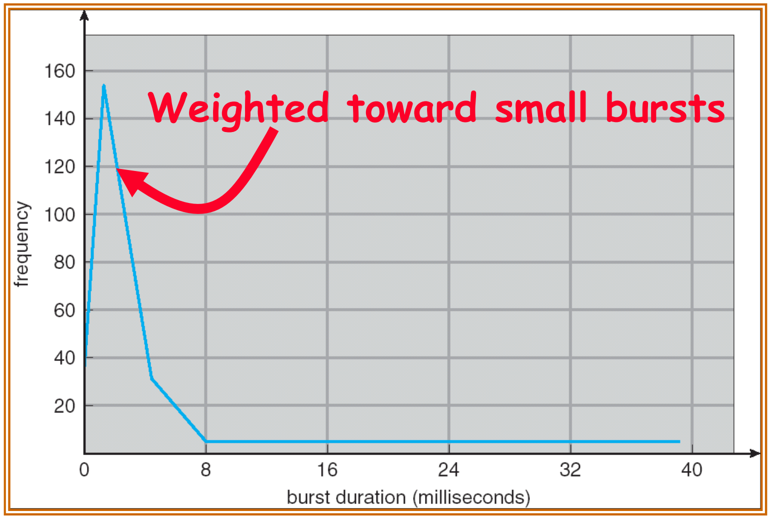
# How to Evaluate a Scheduling algorithm?

- 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



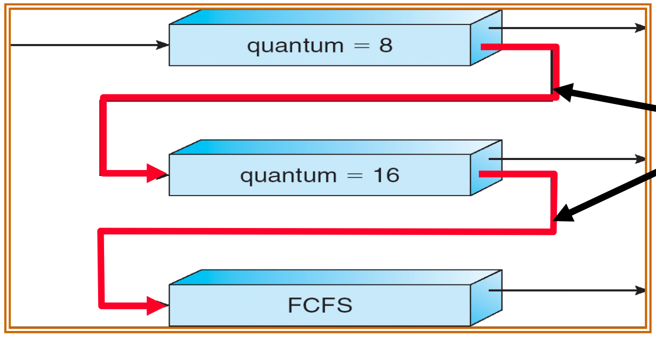
# How to Handle Simultaneous Mix of Diff Types of Apps?

- Consider mix of interactive and high throughput apps:
  - How to best schedule them?
  - How to recognize one from the other?
    - » Do you trust app to say that it is “interactive”?
  - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst Time (observed) useful to decide which application gets CPU time?
  - Short Bursts ⇒ Interactivity ⇒ High Priority?




- Assumptions encoded into many schedulers:
  - Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  - Apps that compute a lot should get low(er?) priority, since they won’t notice intermittent bursts from interactive apps
- Hard to characterize apps:
  - What about apps that sleep for a long time, but then compute for a long time?
  - Or, what about apps that must run under all circumstances (say periodically)

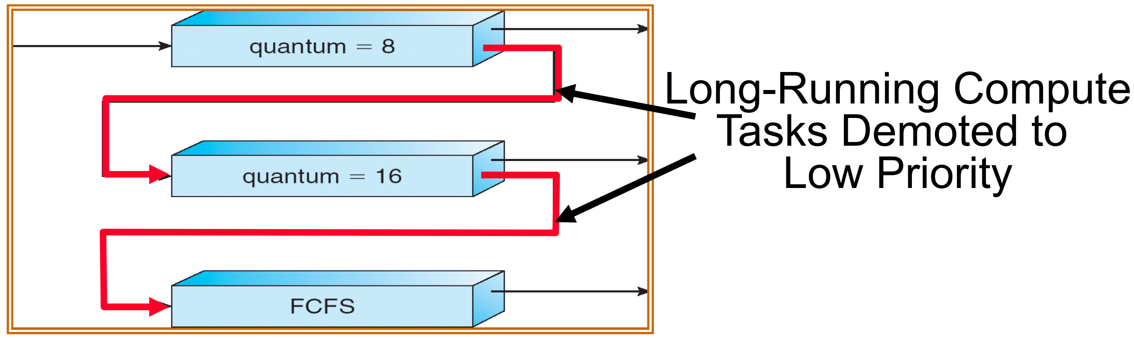
# Multi-Level Feedback Scheduling




Long-Running Compute Tasks Demoted to Low Priority

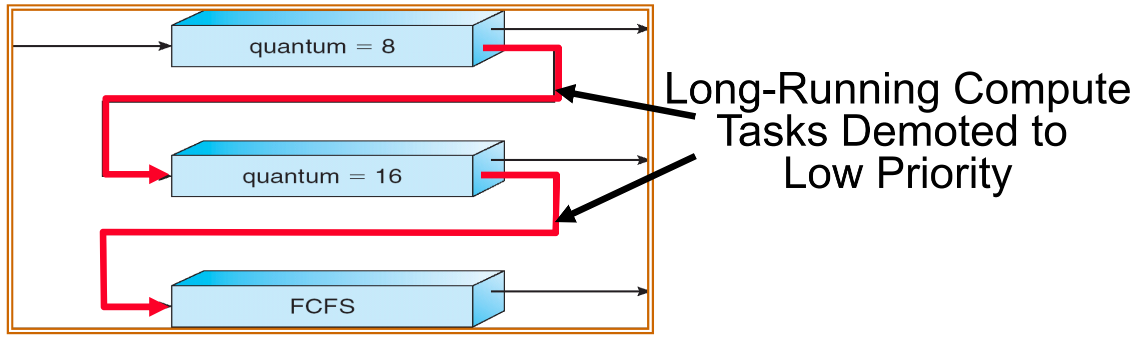
- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple 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)

# Scheduling Details




- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - » serve all from highest priority, then next priority, etc.
  - Time slice:
    - » each queue gets a certain amount of CPU time
    - » e.g., 70% to highest, 20% next, 10% lowest

# Scheduling Details

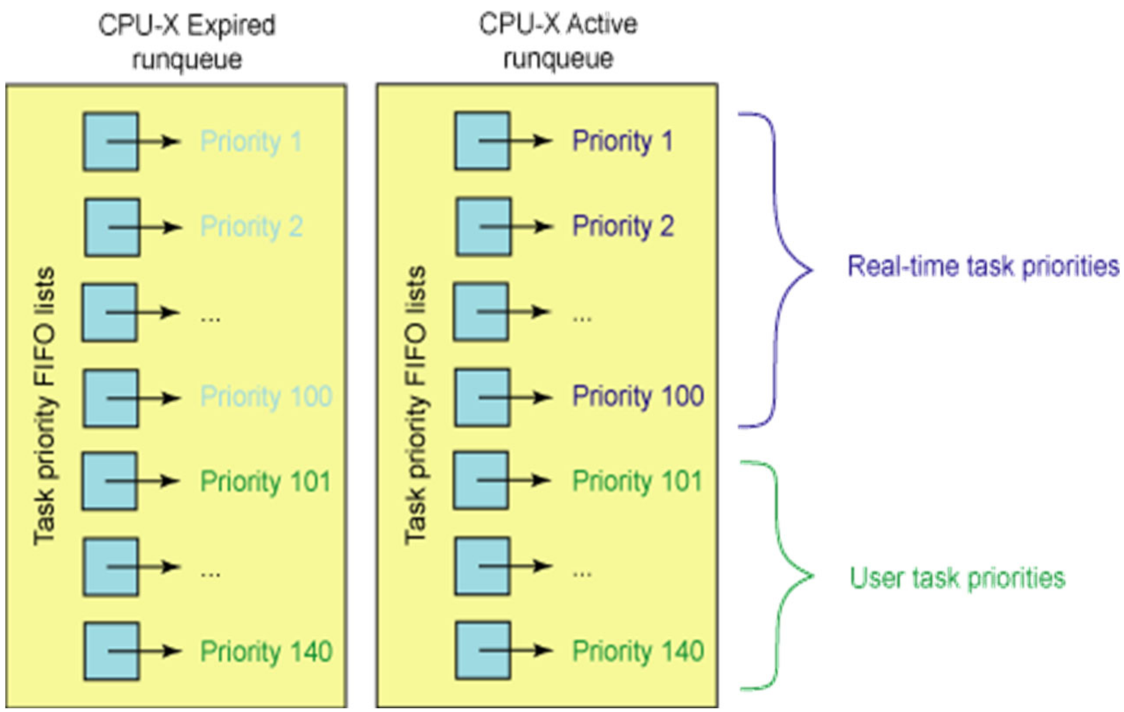



- Countermeasure: user action that can foil intent of the OS designers
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    - » Put in printf's, ran much faster!

# Case Study: Linux O(1) Scheduler


- Priority-based scheduler: 140 priorities
  - 40 for "user tasks" (set by "nice"), 100 for "Realtime/Kernel"
  - Lower priority value ⇒ higher priority (for realtime values)
  - Highest priority value ⇒ Lower priority (for nice values)
  - All algorithms O(1)
    - » Timeslices/priorities/interactivity credits all computed 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

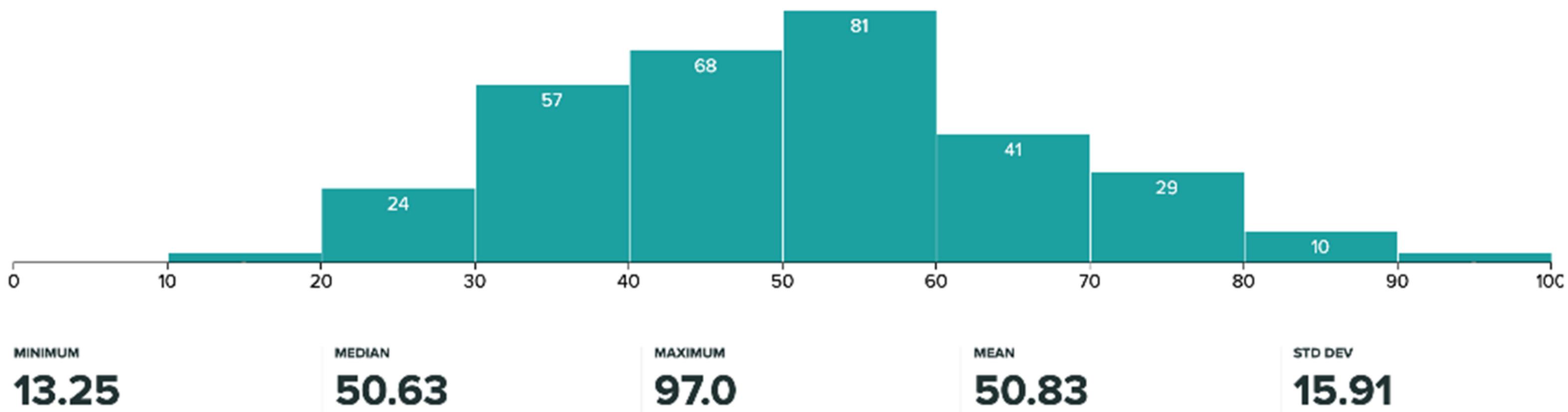



- Lots of ad-hoc heuristics
  - –Try to boost priority of I/O-bound tasks
  - –Try to boost priority of starved tasks

# O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    - » p->sleep_avg = sleep_time – run_time
    - » 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

# Administrivia




- Midterm I graded:
  - Mean 47.8, Std Dev: 12.8, Low: 17.5, High: 83.0
  - Regrade requests before Sunday 2/27@midnight
    - » We will take reasonable arguments for regrades..!
- Solutions are posted

# Administrivia (Con't)

- Project 1 final report is due Tuesday March 1st
- Also due Tuesday March 1st: Peer evaluations
  - These are a required mechanism for evaluating group dynamics
  - Project scores are a zero-sum game
    - » In the normal/best case, all partners get the same grade
    - » In groups with issues, we may take points from non-participating group members and give them to participating group members!
- How does this work?
  - You get 20 points/partner to distribute as you want: Example—4 person group, you get 3 x 20 = 60 points
    - » If all your partners contributed equally, give the 20 points each
    - » Or, you could do something like:
      - 22 points partner 1
      - 22 points partner 2
      - 16 points partner 3
  - DO NOT GIVE YOURSELF POINTS!
    - » You are NOT an unbiased evaluator of your group behavior

# 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

- Recall, However: Simultaneous Multithreading (or “Hyperthreading”)
  – Different threads interleaved on a cycle-by-cycle basis and can be in different processes (have different address spaces)

# 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

# Recall: *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 (using lots of memory!)
  - So – really want to use `test&test&set()` !
- As we discussed in Lecture 8, 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

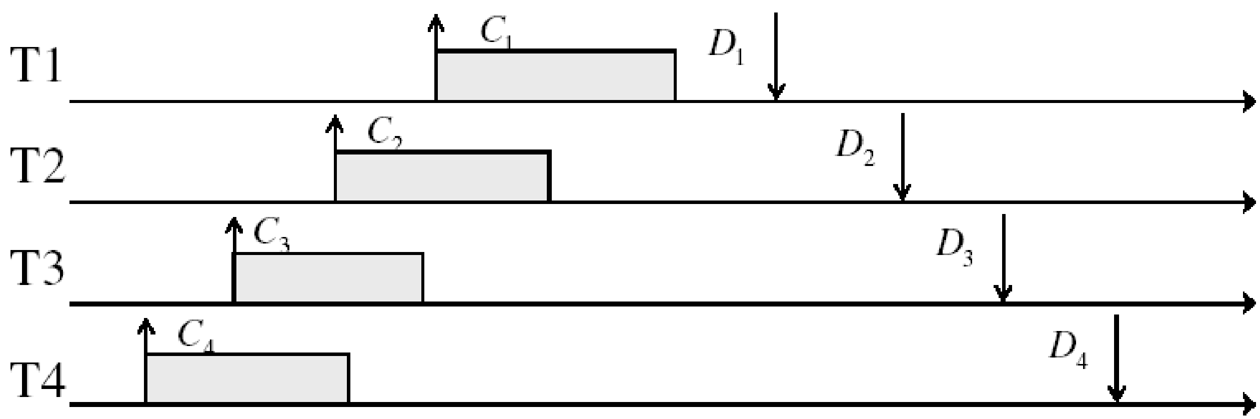
- When multiple threads work together on a multi-core system, try to schedule them together
  - Makes spin-waiting more efficient (inefficient to spin-wait for a thread that's suspended)
- Alternative: OS informs a parallel program how many processors its threads are scheduled on (*Scheduler Activations*)
  - 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

# Real-Time Scheduling

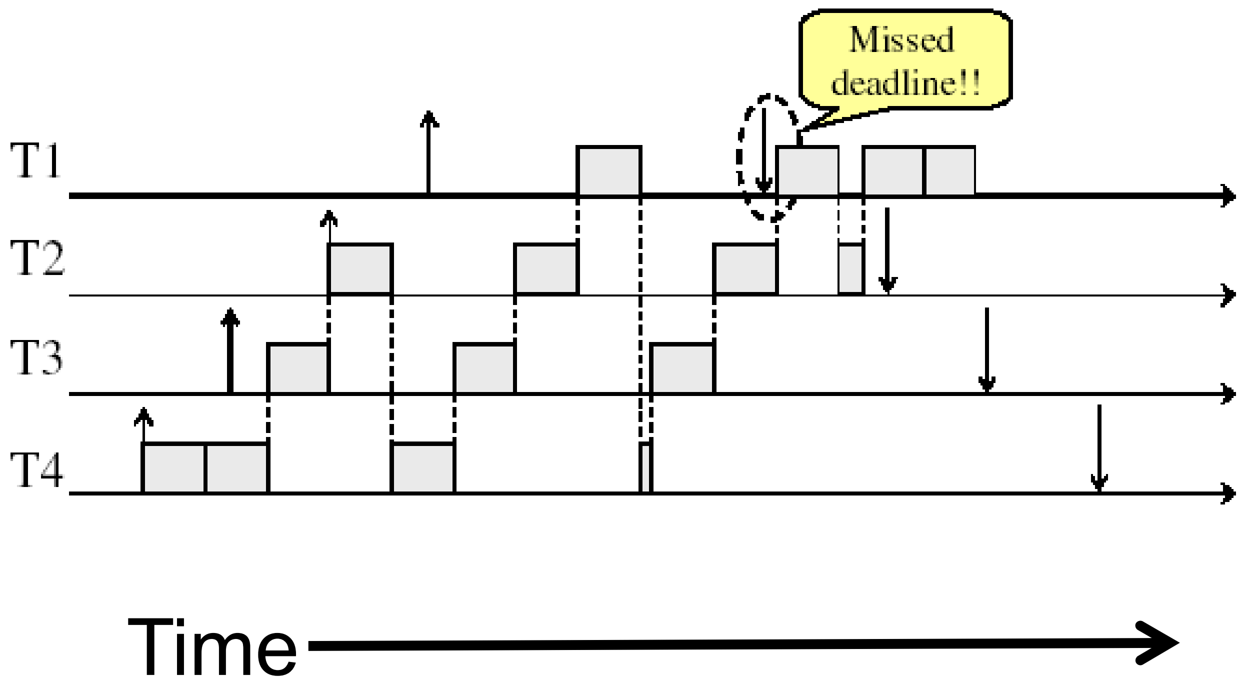
- Goal: Predictability of Performance!
  - We need to predict with confidence worst case response times for systems!
  - In RTS, performance guarantees are:
    - » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - » System/throughput oriented with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

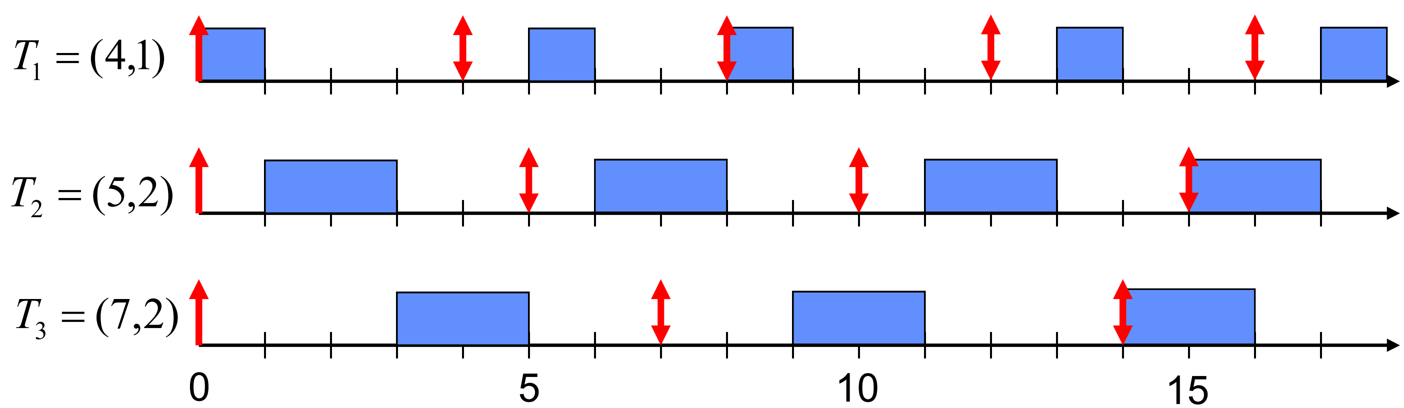


# Example: Round-Robin Scheduling Doesn't Work



# 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

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

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

# 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

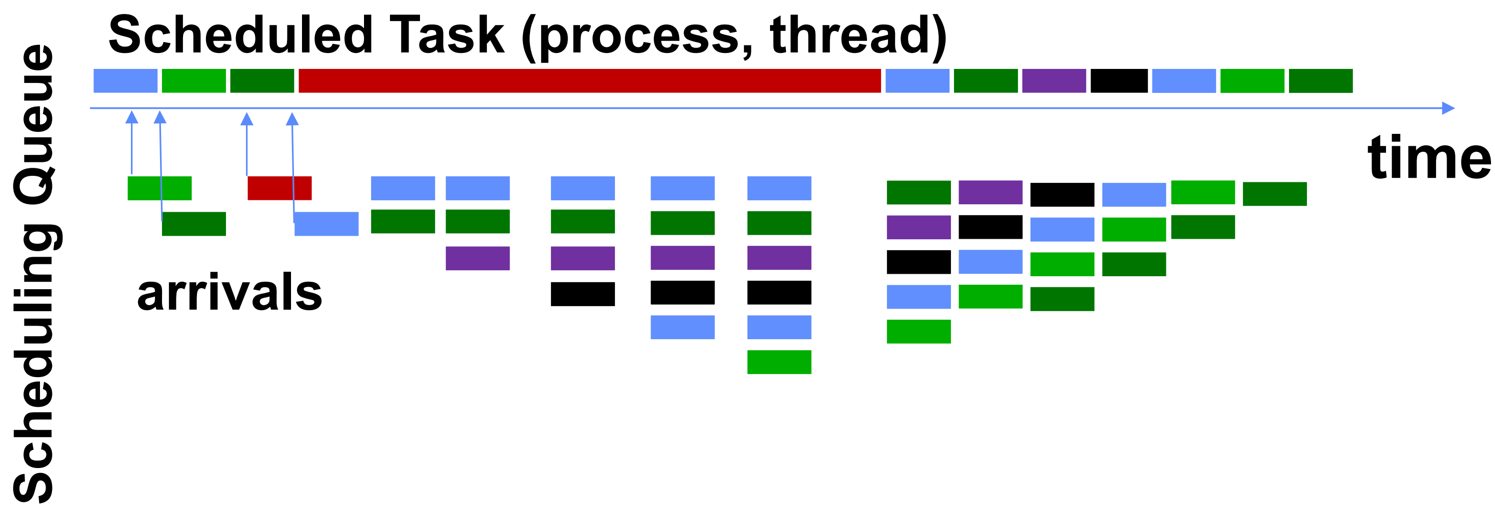
- A non-work-conserving scheduler could trivially lead to starvation

- 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

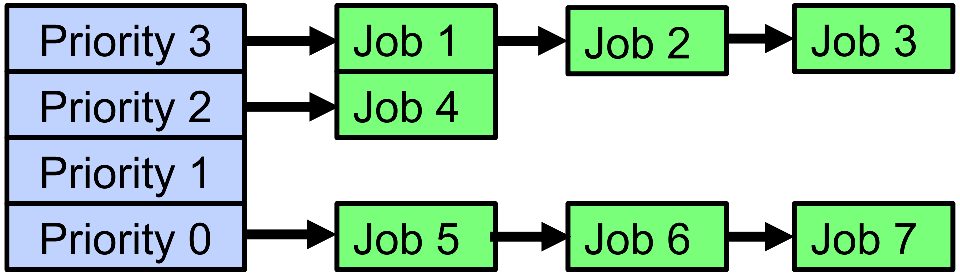
# Is Round Robin (RR) Prone to Starvation?

- 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… (if you give up your time slot early, you don't get the time back!)

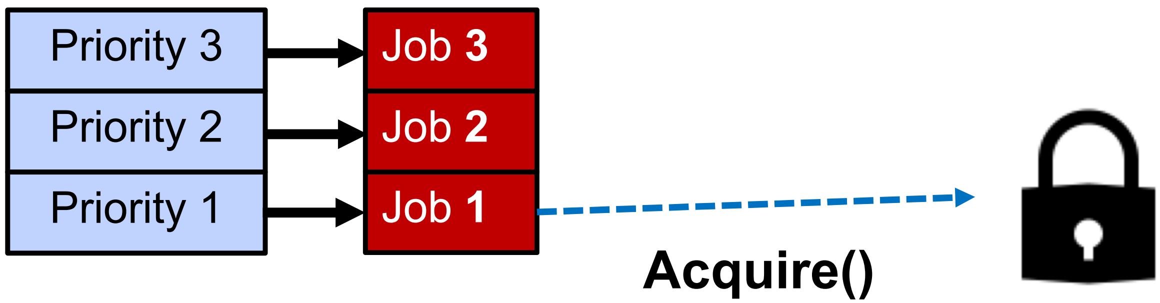
# Is Priority Scheduling Prone to Starvation?

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



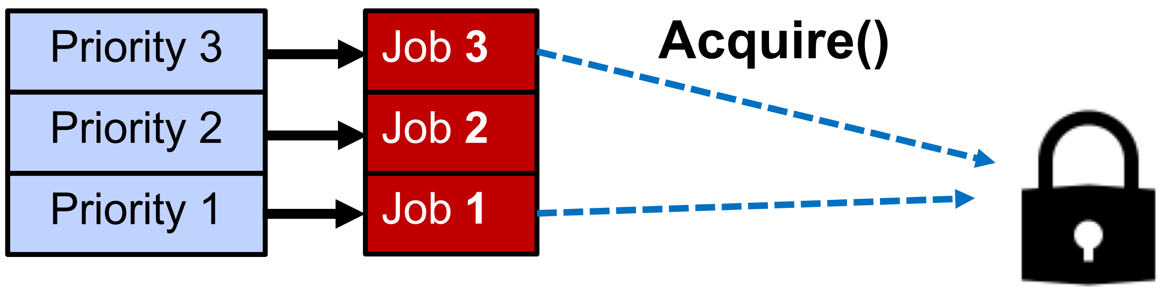

- But there are more serious problems as well…
  - Priority inversion: even high priority threads might become starved

# Priority Inversion



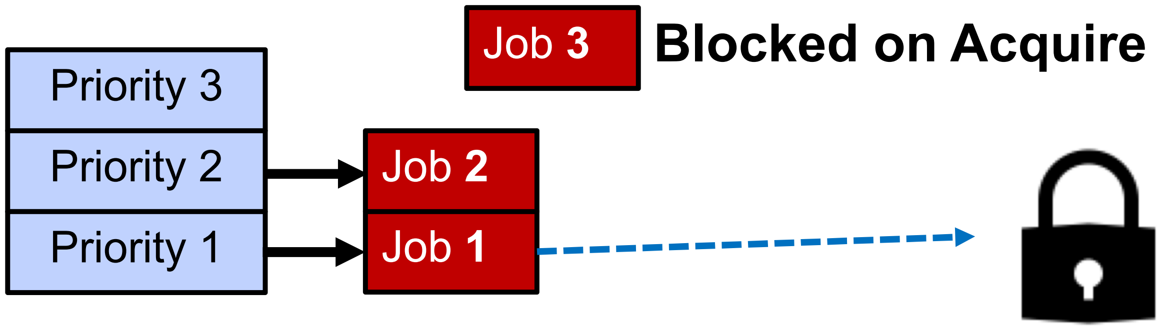

- **At this point, which job does the scheduler choose?**
- Job 3 (Highest priority)

# 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”?

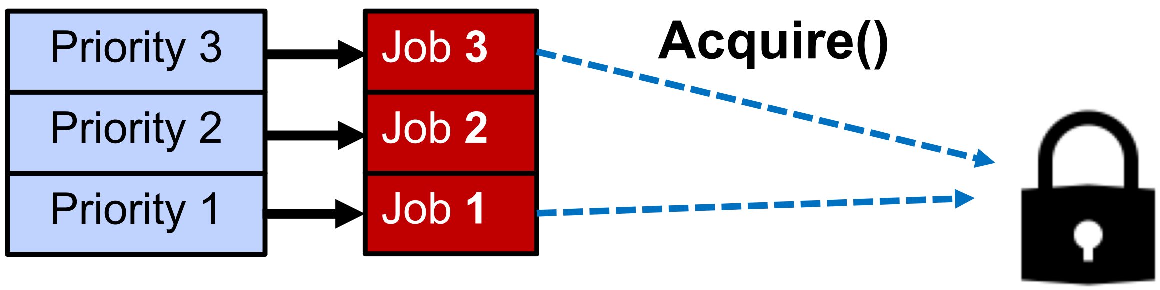
**High Priority**

```
while (try_lock) {
…
}
```

**Low Priority**

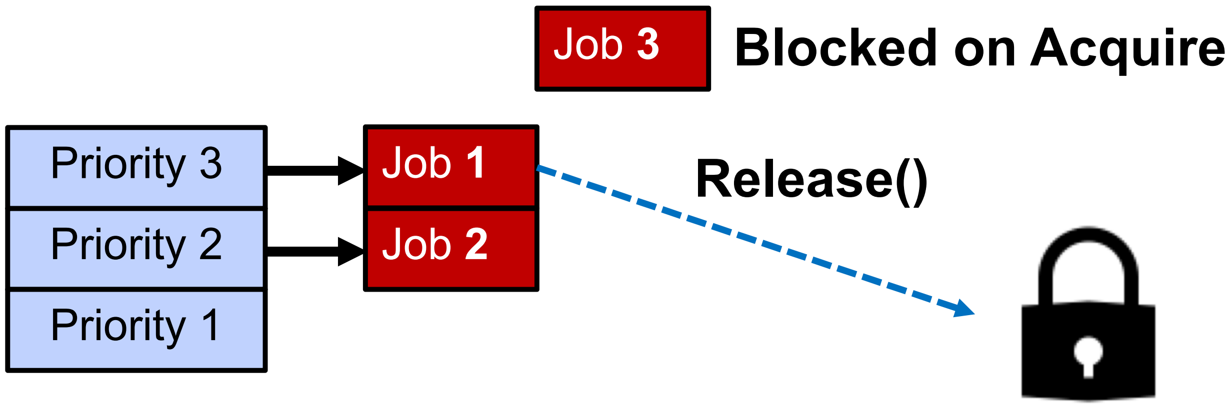
```
lock.acquire(…)
…
lock.release(…)
```

# One Solution: Priority Donation/Inheritance



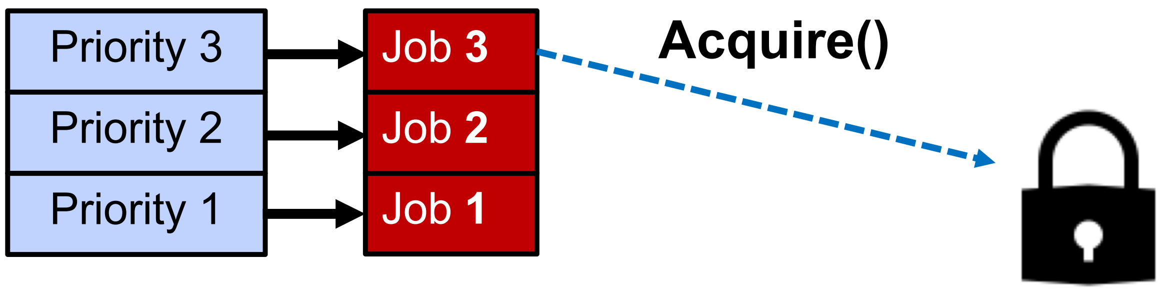

- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

# One Solution: Priority Donation/Inheritance




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

# One Solution: Priority Donation/Inheritance




- Job 1 completes critical section and releases lock
- Job 3 acquires lock, runs again
- How does the scheduler know?

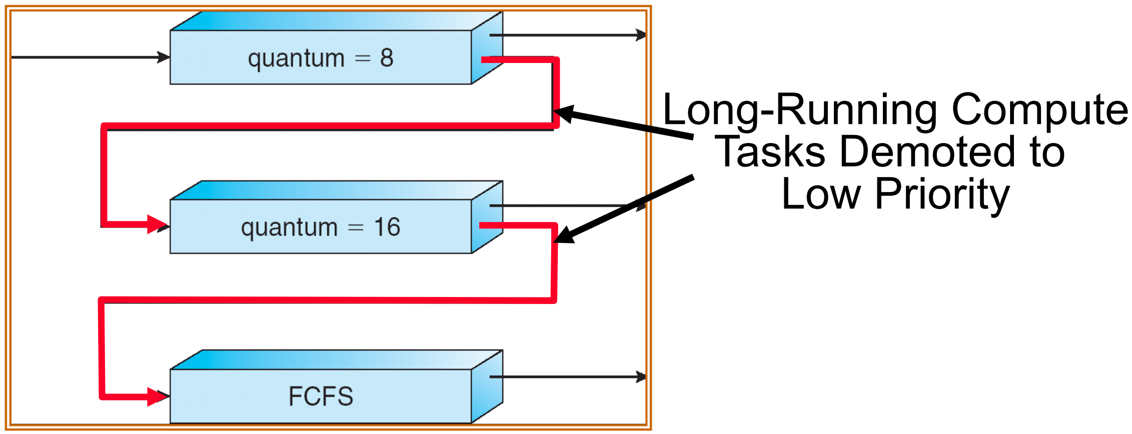
# Case Study: Martian Pathfinder Rover

- 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:

| | |
|---|---|
| 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
- 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

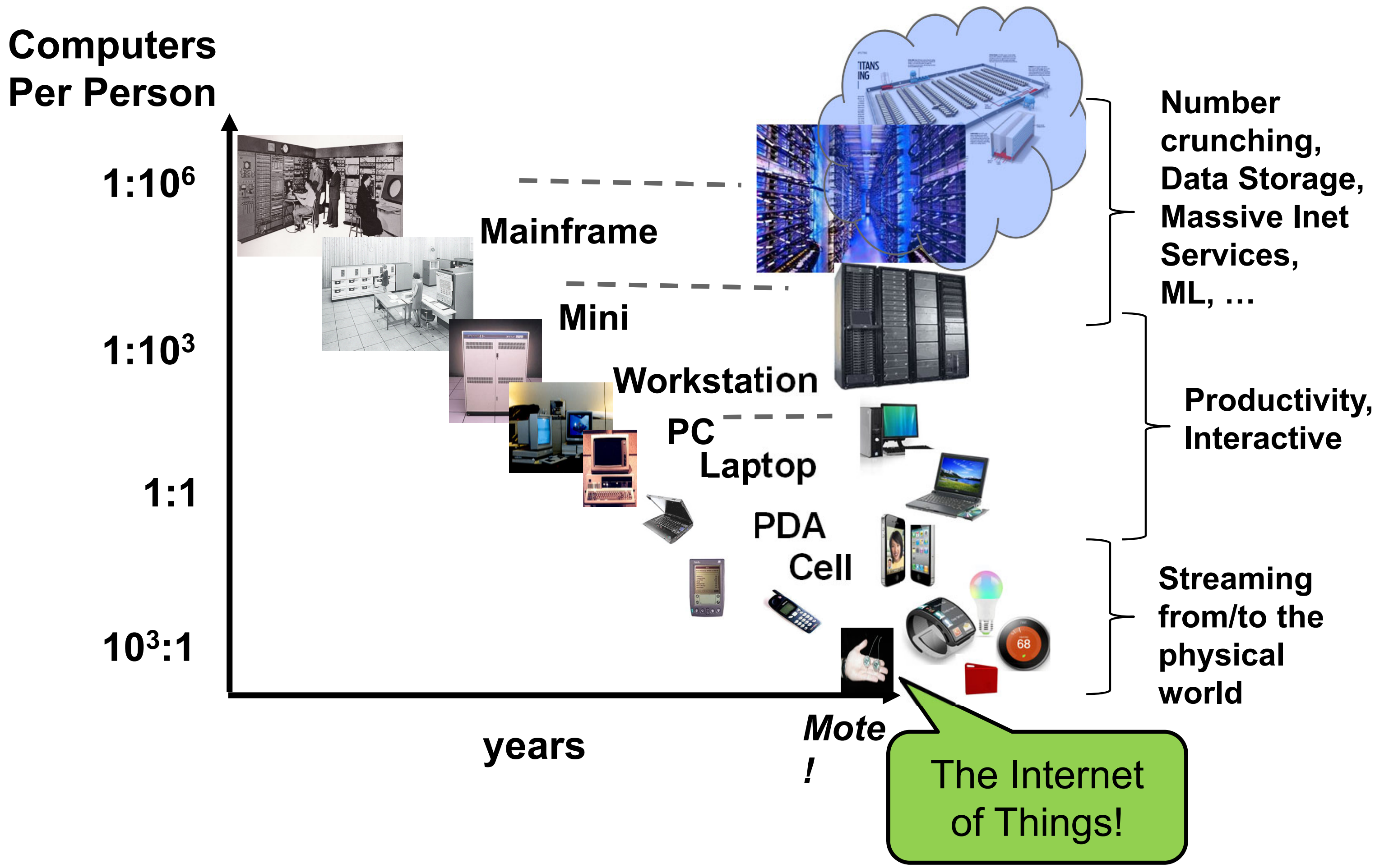
# Cause for Starvation: Priorities?

- 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

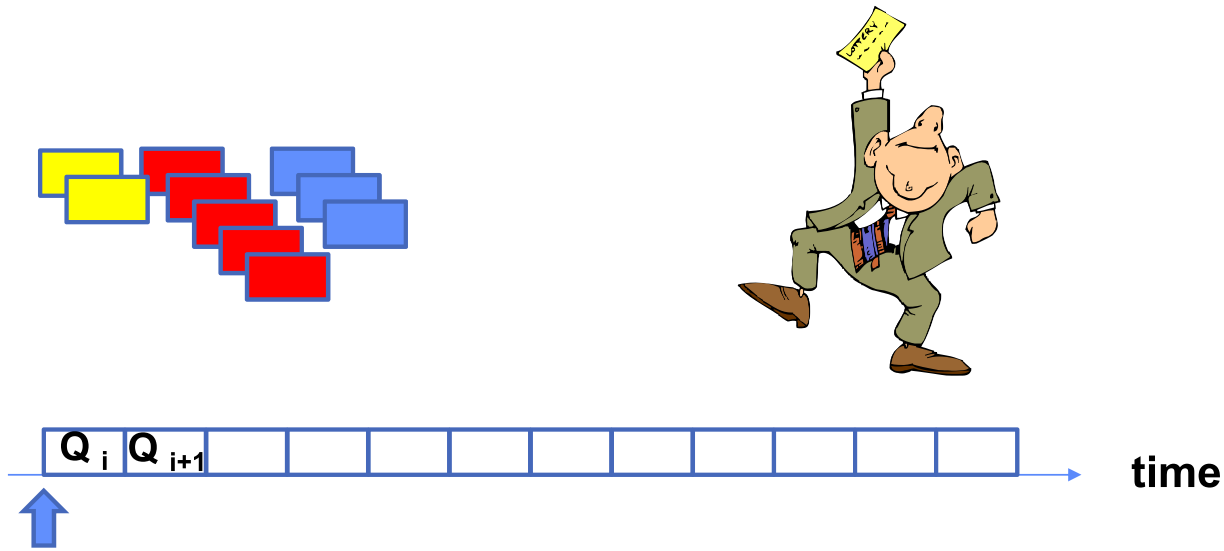
# DOES PRIORITIZING SOME JOBS *NECESSARILY* STARVE THOSE THAT AREN'T PRIORITIZED?

# 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

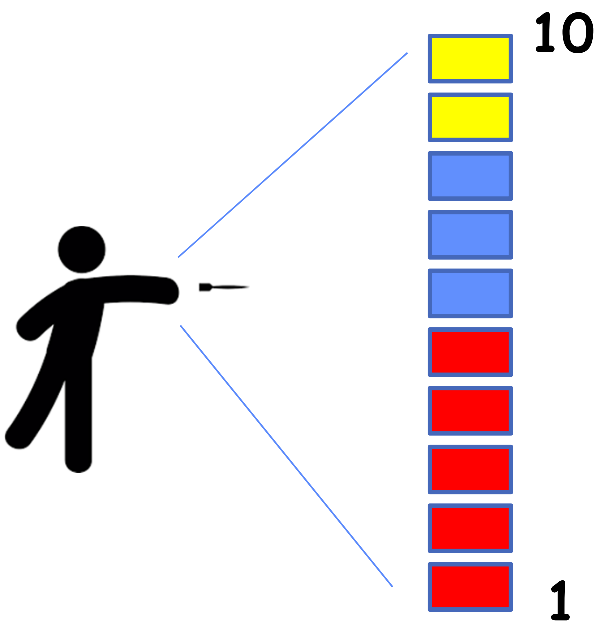
- Instead, we can share the CPU *proportionally*
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)

# 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

# Lottery Scheduling: Simple Mechanism




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

# Unfairness

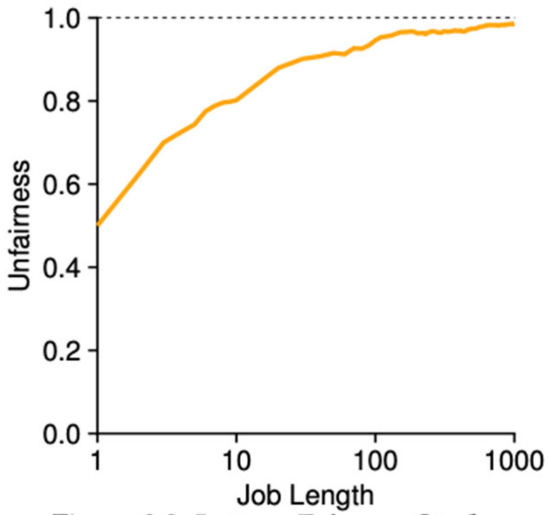


Figure 9.2: **Lottery Fairness Study**

- 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

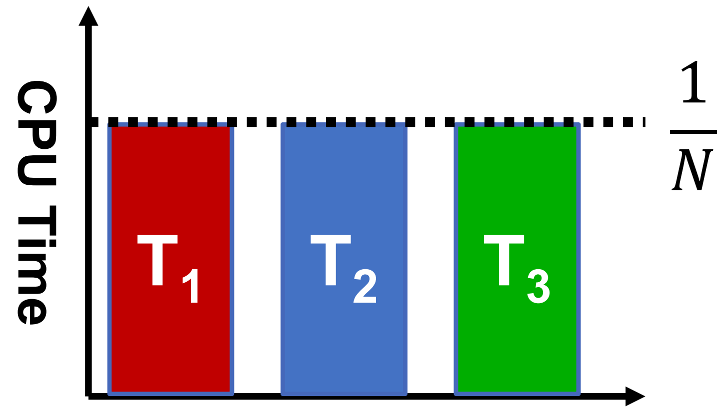
# Stride Scheduling

- Achieve proportional share scheduling without resorting to randomness, and overcome the “law of small numbers” problem.
- “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
- Some messiness of counter wrap-around, new jobs, …

# 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

Model: "Perfectly" subdivided CPU:

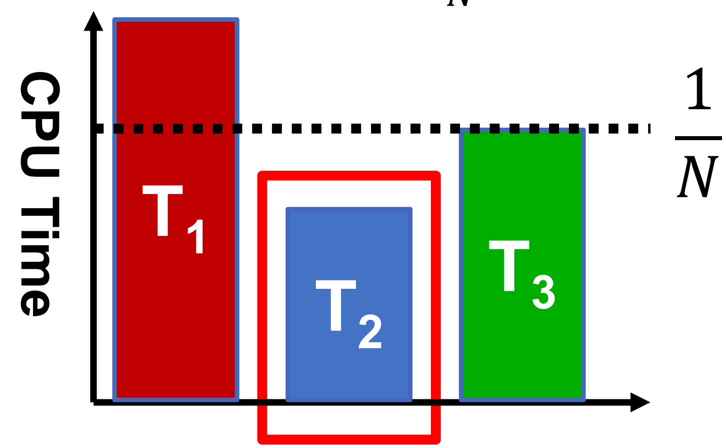



- In general, can't do this with real hardware
  - OS needs to give out full CPU in time slices
  - Thus, we must use something to keep the threads roughly in sync with one another

# 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 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

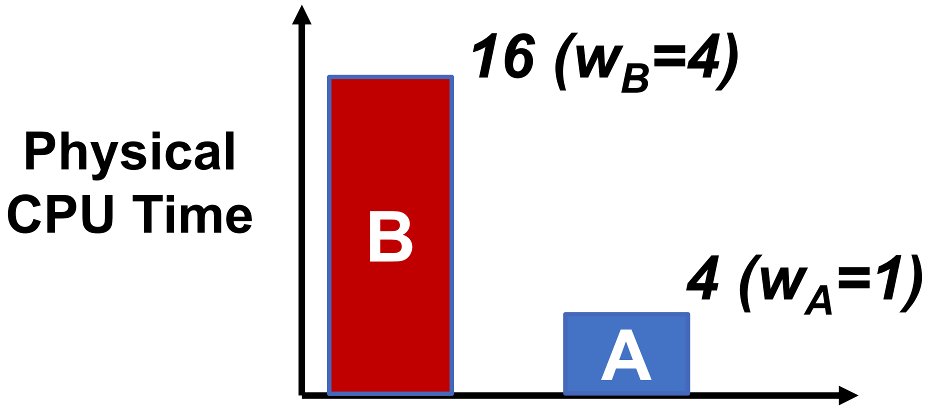
# Linux CFS: Proportional Shares

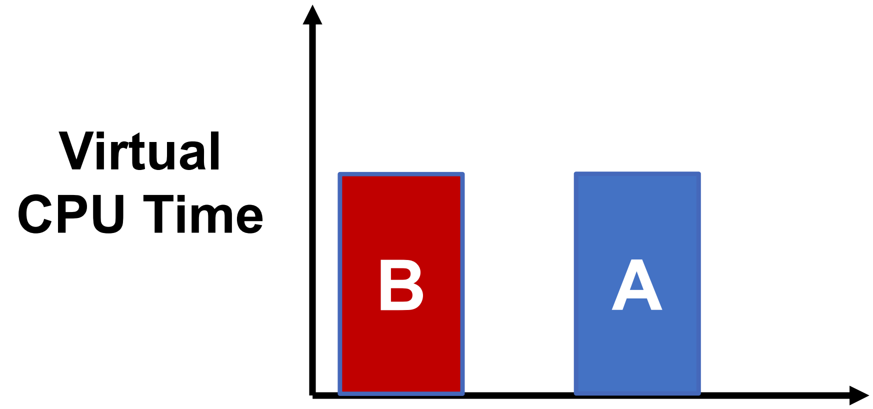
- What if we want to give more CPU to some and less to others in CFS (proportional share) ?
  - Allow different threads to have different *rates* of execution (cycles/time)
- Use weights! Key Idea: Assign a weight $w_i$ to each process *I* to compute the switching quanta $Q_i$
  - Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$
  - 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 ⇒ 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 ⇒<br>Task with lower nice value has 3 times the weight, since $(1.25)^5 \approx 3$
- 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



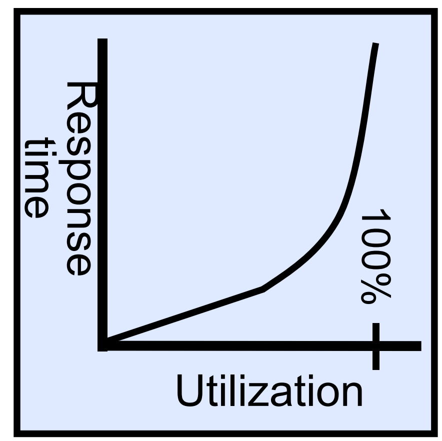

- Track a thread's *virtual* runtime rather than its true physical runtime
  - Higher weight: Virtual runtime increases more slowly
  - Lower weight: Virtual runtime increases more quickly
- Scheduler’s Decisions are based on Virtual CPU Time
- Use of Red-Black tree to hold all runnable processes as 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 |

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