

Scheduling 2: Starvation

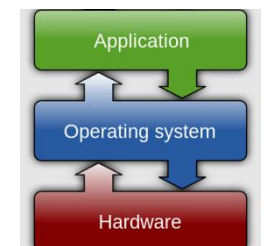
Lecture 10

Hartmut Kaiser

<https://teaching.hkaiser.org/spring2025/csc4103/>

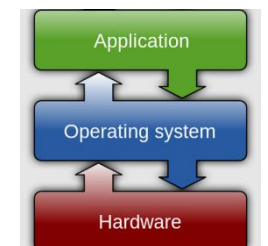
Evaluating Schedulers

- **Response Time** (ideally low)
 - What user sees: from keypress to character on screen
 - Or completion time for non-interactive
- **Throughput** (ideally high)
 - Total operations (jobs) per second
 - Overhead (e.g. context switching), artificial blockers
- **Fairness**
 - Fraction of resources provided to each
 - May conflict with best avg. throughput, resp. time



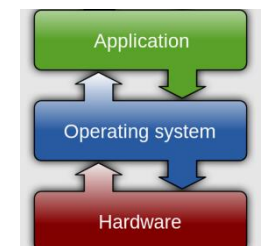
Recall: Classic Scheduling Policies

- **First-Come First-Served**: Simple, vulnerable to convoy effect
- **Round-Robin**: Fixed CPU time quantum, cycle between ready threads
- **Priority**: Respect differences in importance
- **Shortest Job/Remaining Time First**: Optimal for average response time, but unrealistic
- **Multi-Level Feedback Queue**: Use past behavior to approximate SRTF and mitigate overhead

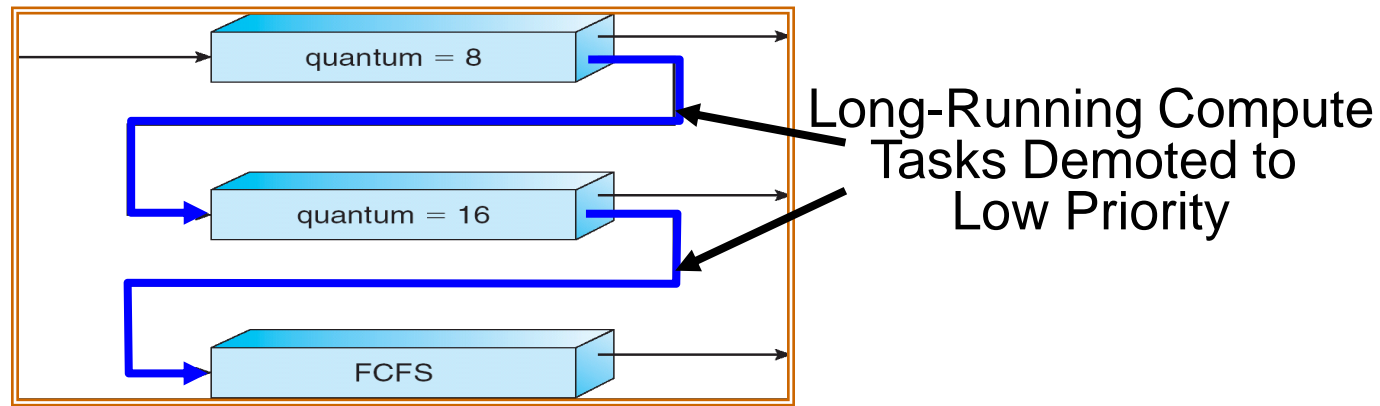


Adaptive Scheduling

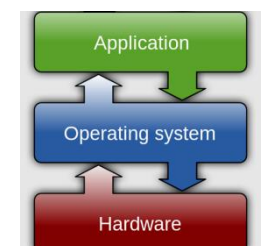
- How can we adapt the scheduling algorithm based on threads' past behavior?
- Two steps:
 - Based on past observations, predict what threads will do in the future.
 - Make scheduling decisions based on those predictions.



Multi-Level Feedback Queue (MLFQ)

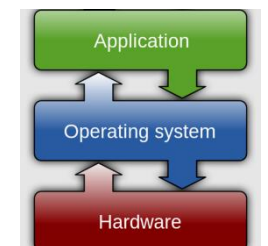


- Intuition: approximate SRTF by setting priority level proportional to burst length
- Job Exceeds Quantum: Drop to lower queue
- Job Doesn't Exceed Quantum: Raise to higher queue

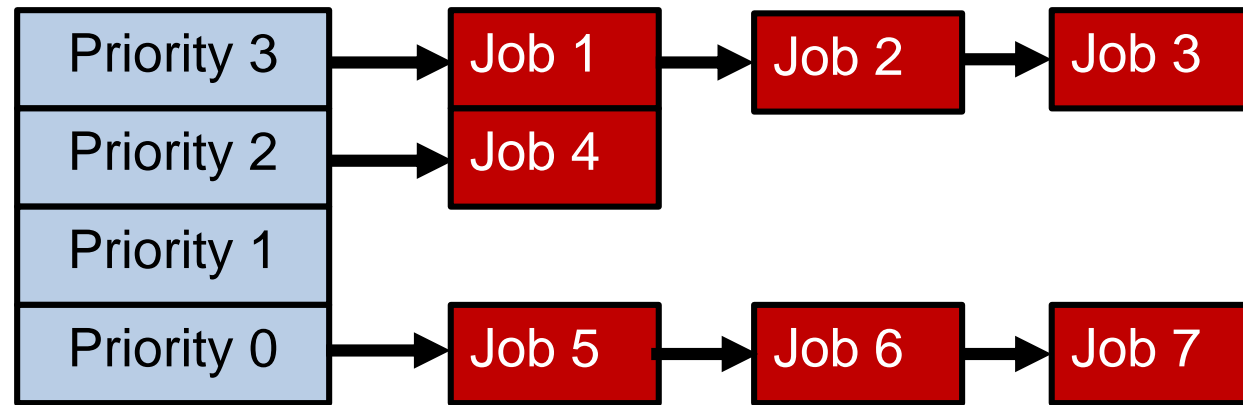


How to Implement MLFQ in the Kernel?

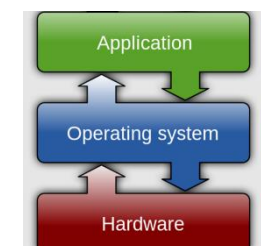
- We could explicitly build the queue data structures
- Or, we can leverage priority-based scheduling!



Recall: Policy Based on Priority Scheduling



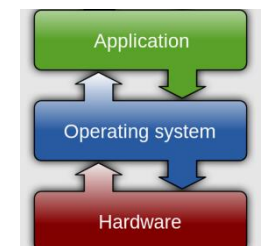
- Systems may try to set priorities according to some policy goal
- Example: Give interactive higher priority than long calculation
 - Prefer jobs waiting on I/O to those consuming lots of CPU
- Try to achieve fairness: elevate priority of threads that don't get CPU time (ad-hoc, bad if system overload)



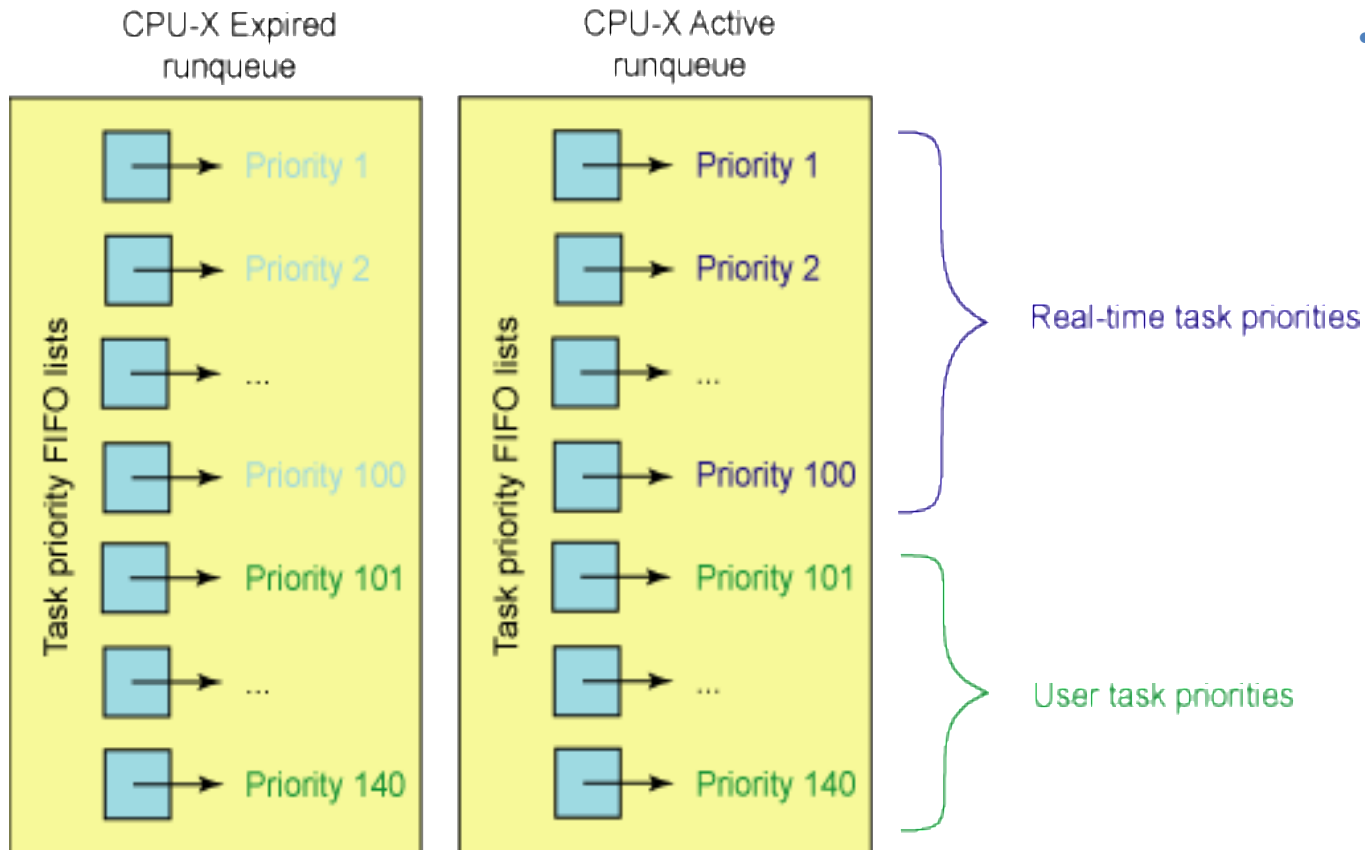
Linux O(1) Scheduler



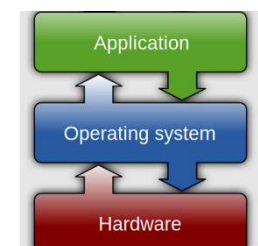
- MLFQ-Like Scheduler with 140 Priority Levels
 - 40 for user tasks, 100 “realtime” tasks
 - All algorithms $O(1)$ complexity – low overhead
 - Timeslices/priorities/interactivity credits all computed when job finishes time slice
- Active and expired queues at each priority
 - Once active is empty, swap them (pointers)
 - Round Robin within each queue (varying quanta)
- Timeslice depends on priority – linearly mapped onto timeslice range



Linux O(1) Scheduler

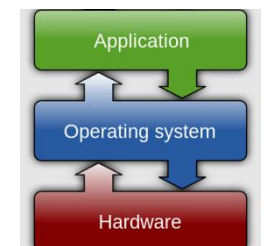


- Lots of ad-hoc heuristics
 - Try to boost priority of I/O-bound tasks
 - Try to boost priority of starved tasks



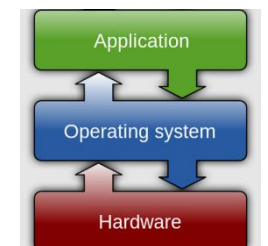
So, Does the OS Schedule Processes or Threads?

- 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



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

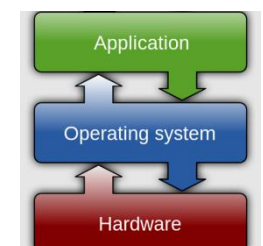


Recall: Spinlock

- Spinlock implementation:

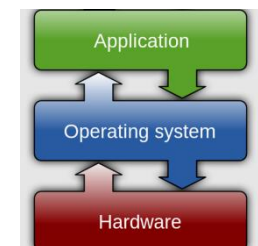
```
int value = 0; // Free
Acquire() {
    while (test&set(value)) {}; // spin while busy
}
Release() {
    value = 0;                // atomic store
}
```

- Spinlock doesn't put the calling thread to sleep—it just busy waits
 - When might this be preferable?
- For multiprocessor cache coherence: every test&set() is a write, which makes value ping-pong around in cache (using lots of memory BW)



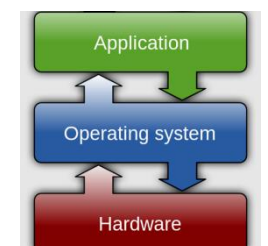
Gang Scheduling and Parallel Applications

- 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



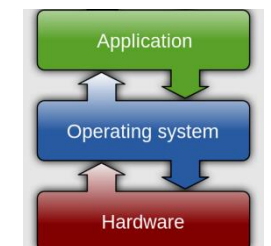
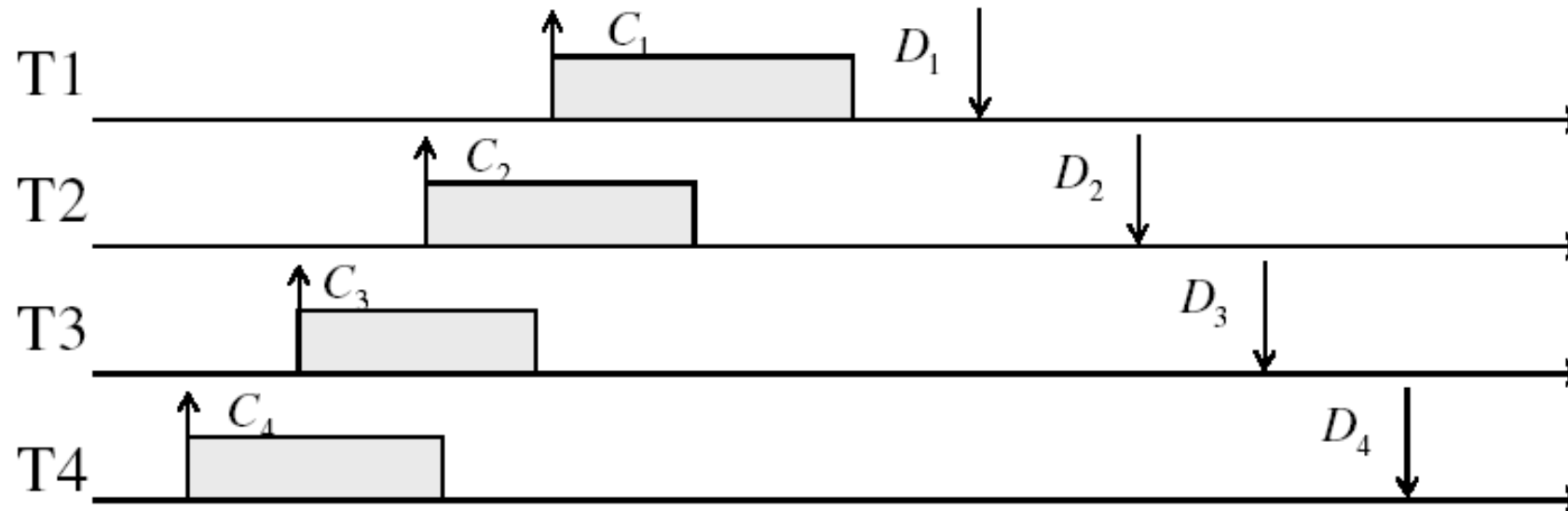
Real-Time Scheduling

- Goal: Guaranteed Performance
 - Meet deadlines even if it means being unfair or slow
 - Limit how bad the worst case is
- Hard real-time
 - Meet all deadlines (if possible)
 - Ideally: determine in advance if this is possible
 - Earliest Deadline First (EDF), Least Laxity First (LLF)
- Soft real-time
 - Attempt to meet deadlines with high probability
 - Constant Bandwidth Server (CBS)

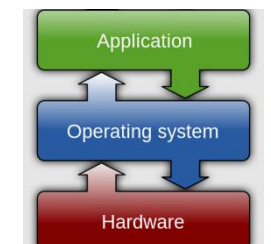
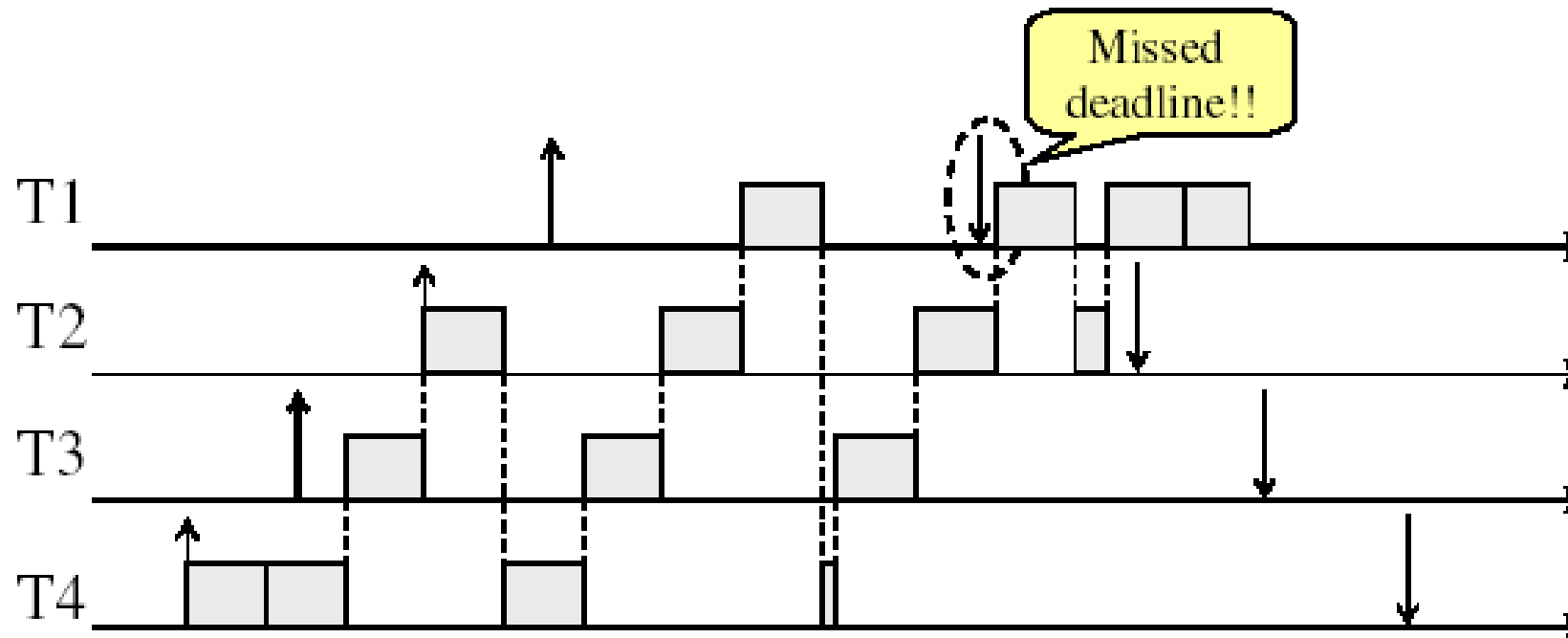


Real-Time Example

- Preemptible tasks with known deadlines (D) and known burst times (C)

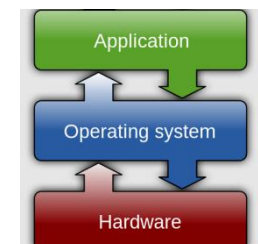
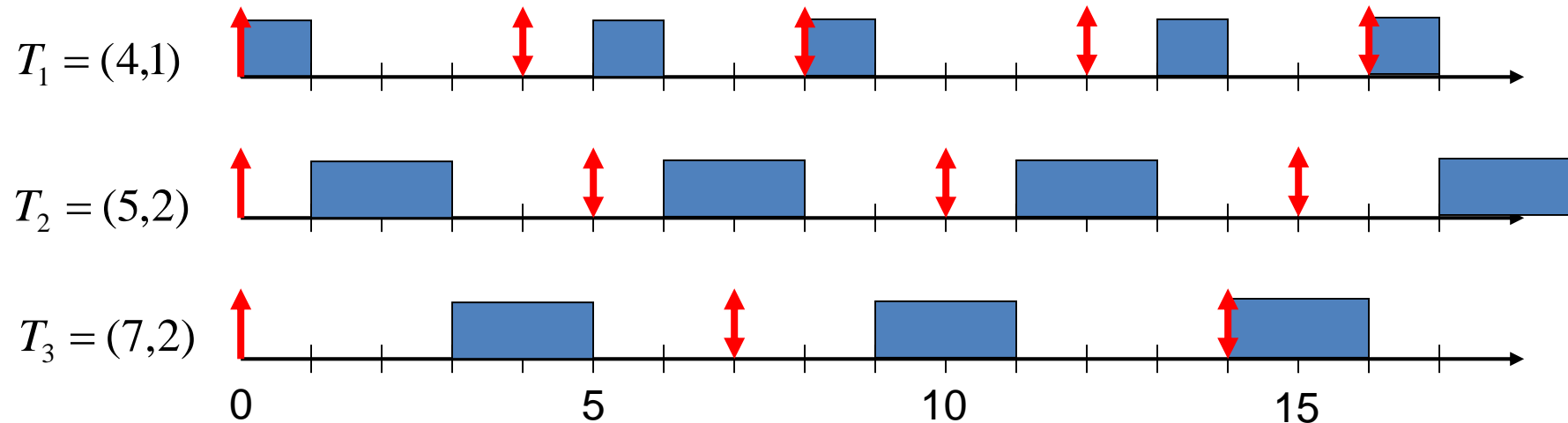


What if we try Round-Robin?



Earliest Deadline First (EDF)

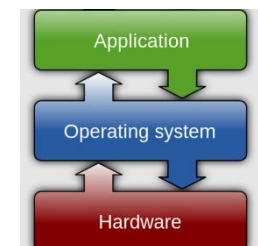
- Priority scheduling with preemption
- Prefer task with earliest deadline
 - Priority (inverse) proportional to time until deadline
- Example with periodic tasks:



EDF Feasibility Testing

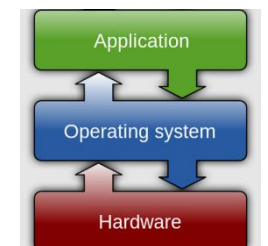
- Even EDF won't work if you have too many tasks
- For n tasks with computation time C_i and deadline D_i , a feasible schedule exists if:

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



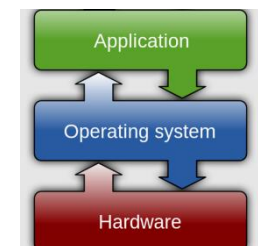
Ensuring Progress

- Starvation: thread fails to make progress for an indefinite period of time
- 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 fall into and how to avoid them...



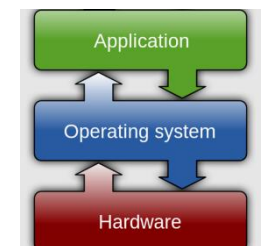
Today: Schedulers Prone to Starvation

- What kinds of schedulers are prone to starvation?
- Of the scheduling policies we've studied, which are prone to starvation? And can we fix them?
- How might we design scheduling policies that avoid starvation entirely?
 - Arguably more relevant now than when CPU scheduling was first developed...



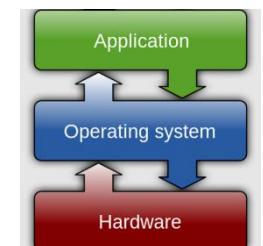
Non-Work-Conserving Scheduler

- 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)



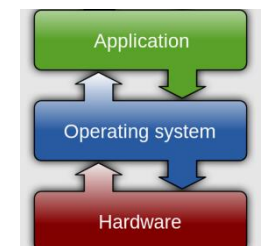
Last-Come, First-Served (LCFS)

- Stack (LIFO) as a scheduling data structure
- Late arrivals get fastest 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”...

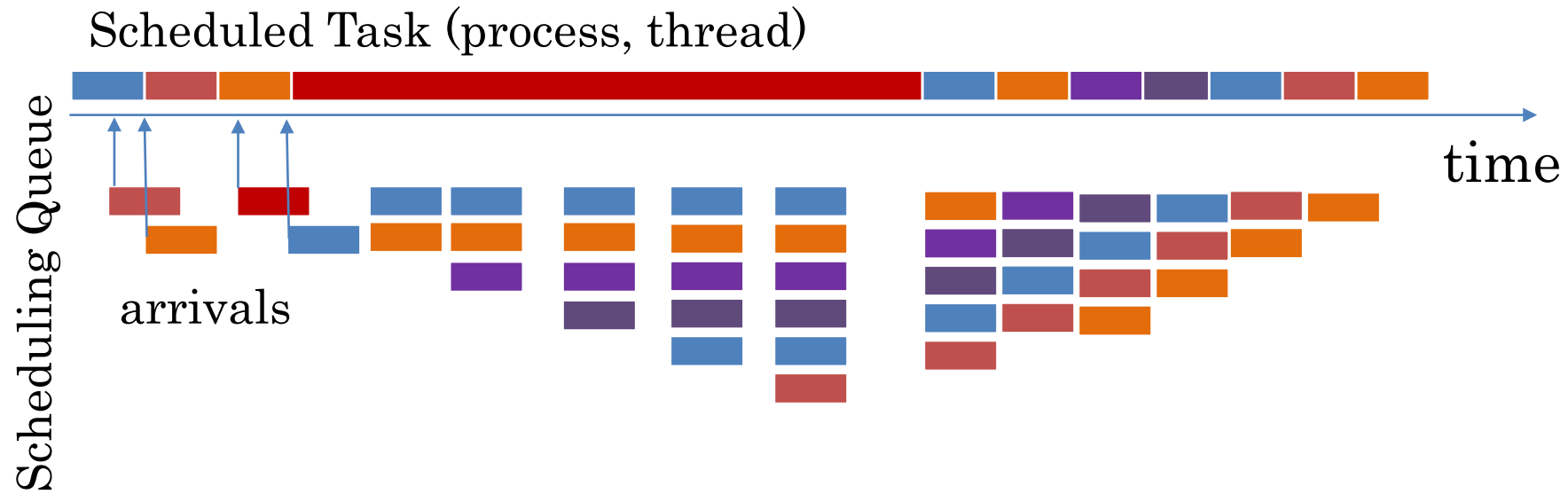


Today: Schedulers Prone to Starvation

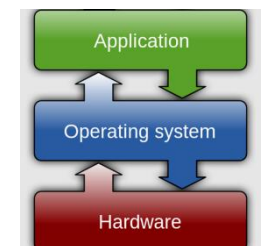
- What kinds of schedulers are prone to starvation?
- Of the scheduling policies we've studied, which are prone to starvation? And can we fix them?
- How might we design scheduling policies that avoid starvation entirely?
 - Arguably more relevant now than when CPU scheduling was first developed...



Is FCFS Prone to Starvation?

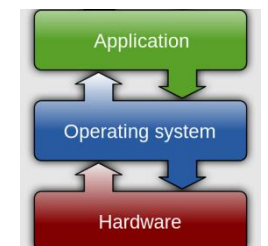


- If a task never yields (e.g., goes into an infinite loop), then other tasks don't get to run
- Problem with all non-preemptive schedulers...



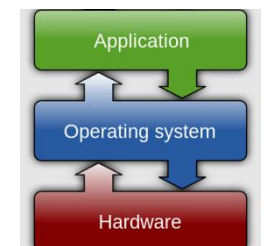
Is Round Robin (RR) Prone to Starvation?

- Each of N processes gets $\sim 1/N$ 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...

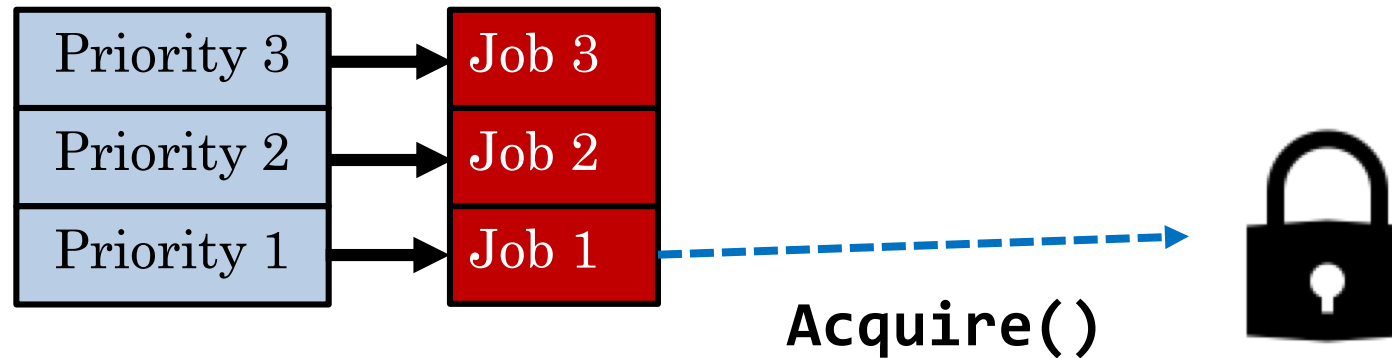


Is Priority Scheduling Prone to Starvation?

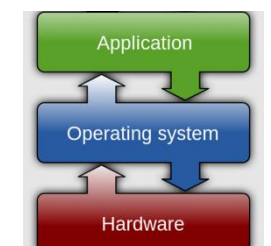
- Always run the ready thread with highest priority
 - Low priority thread might never run!
 - Starvation
- But there are more serious problems as well...
 - Priority inversion: even high priority threads might become starved



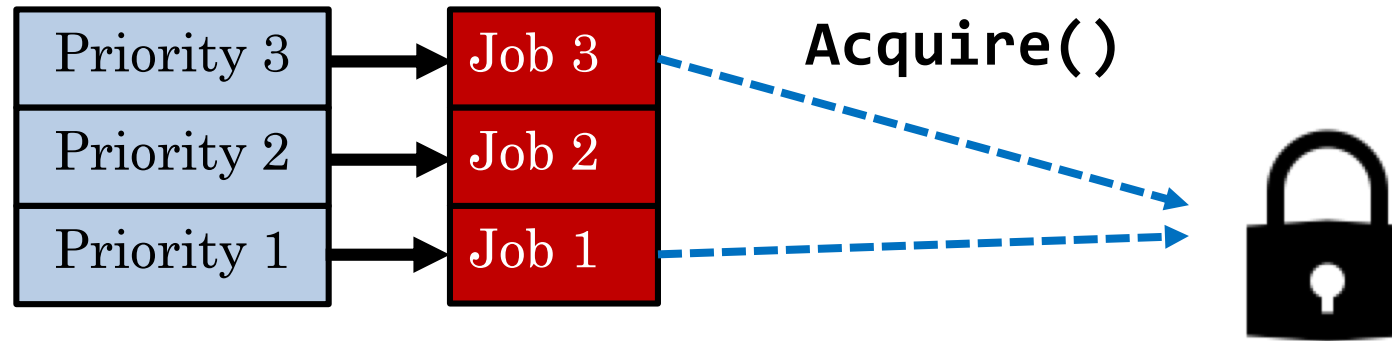
Priority Inversion



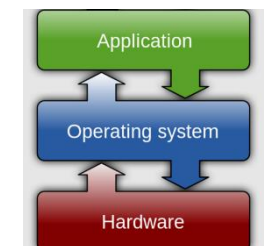
- Job 1 holds lock and suspends
 - At this point, which job does the scheduler choose?
- Job 3 (Highest priority)



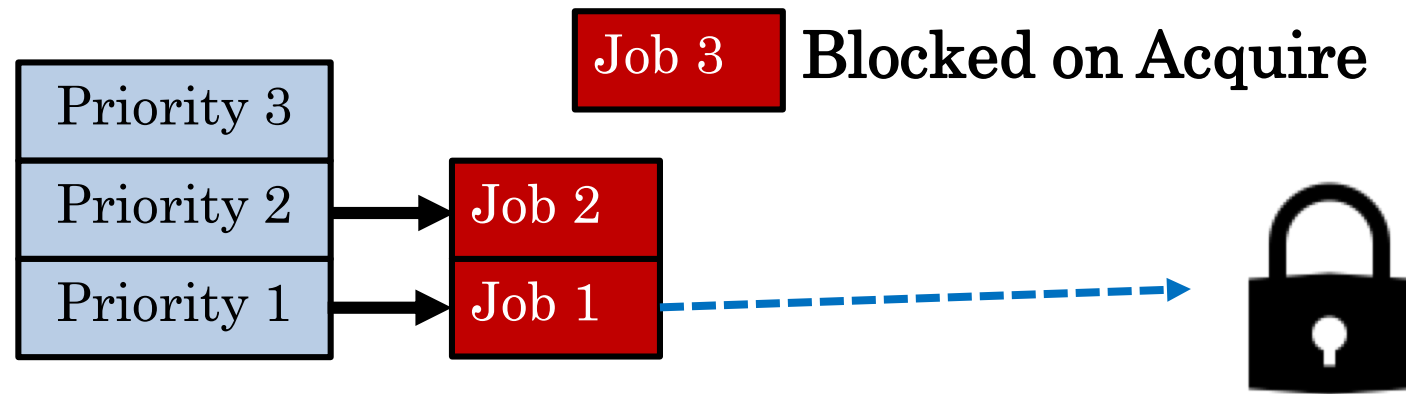
Priority Inversion



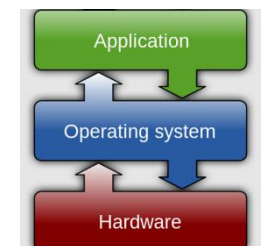
- Job 3 attempts to acquire lock held by Job 1



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”?

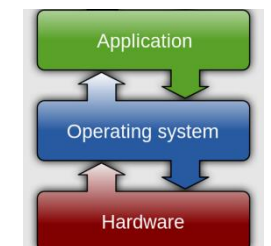

High Priority



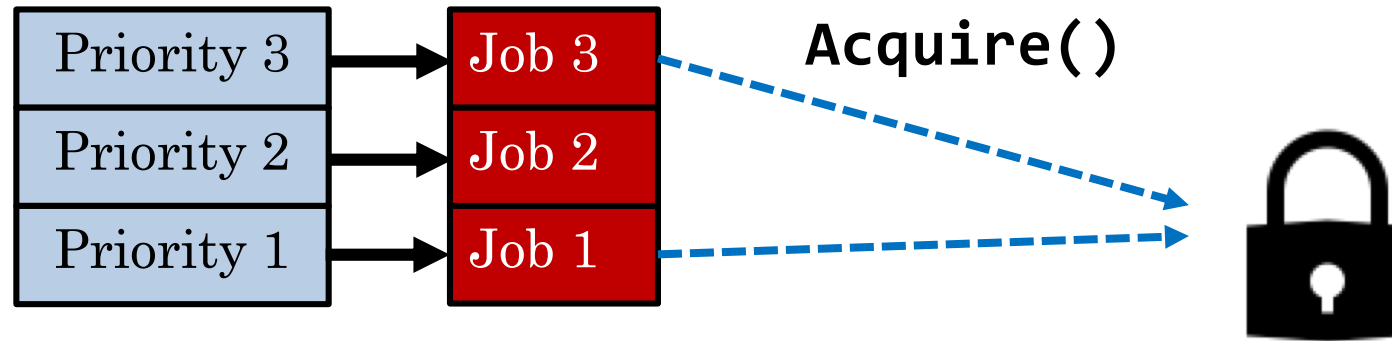
```
while (try_lock) {  
...  
}
```

Low Priority

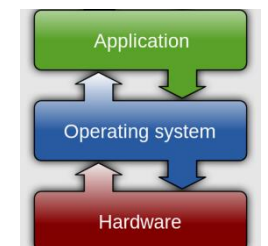
```
lock.acquire(...)  
...  
lock.release(...)
```



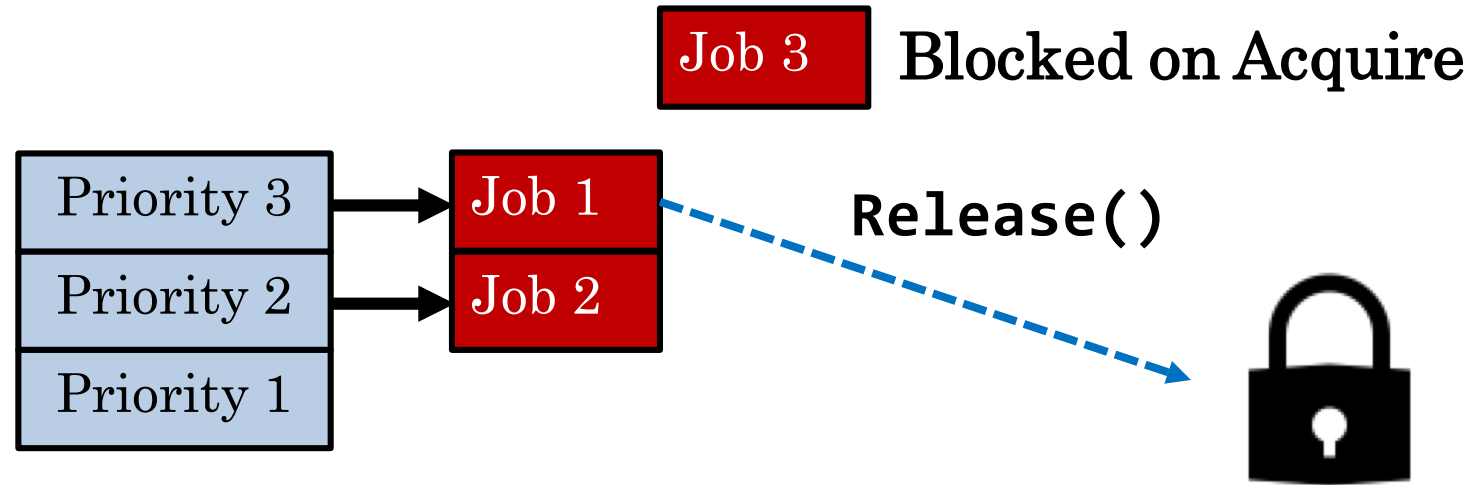
One Solution: Priority Donation



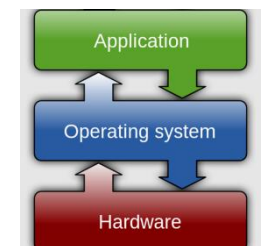
- Job 3 temporarily grants Job 1 its “high priority” to run on its behalf



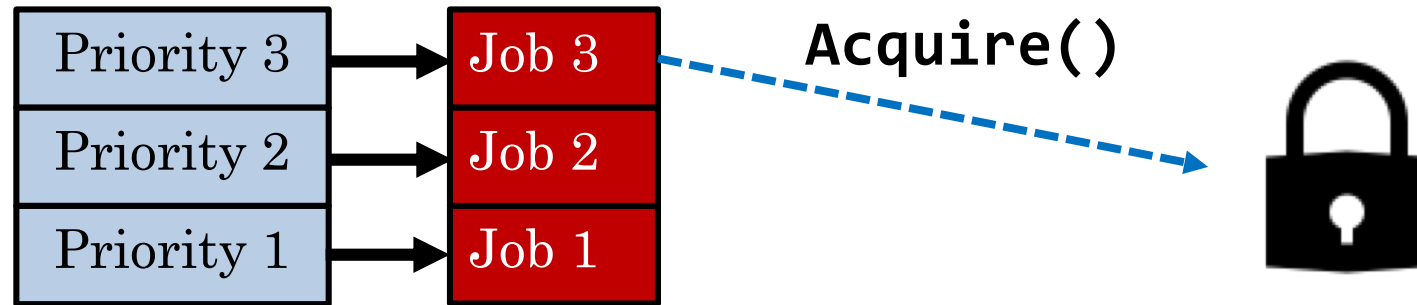
One Solution: Priority Donation



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



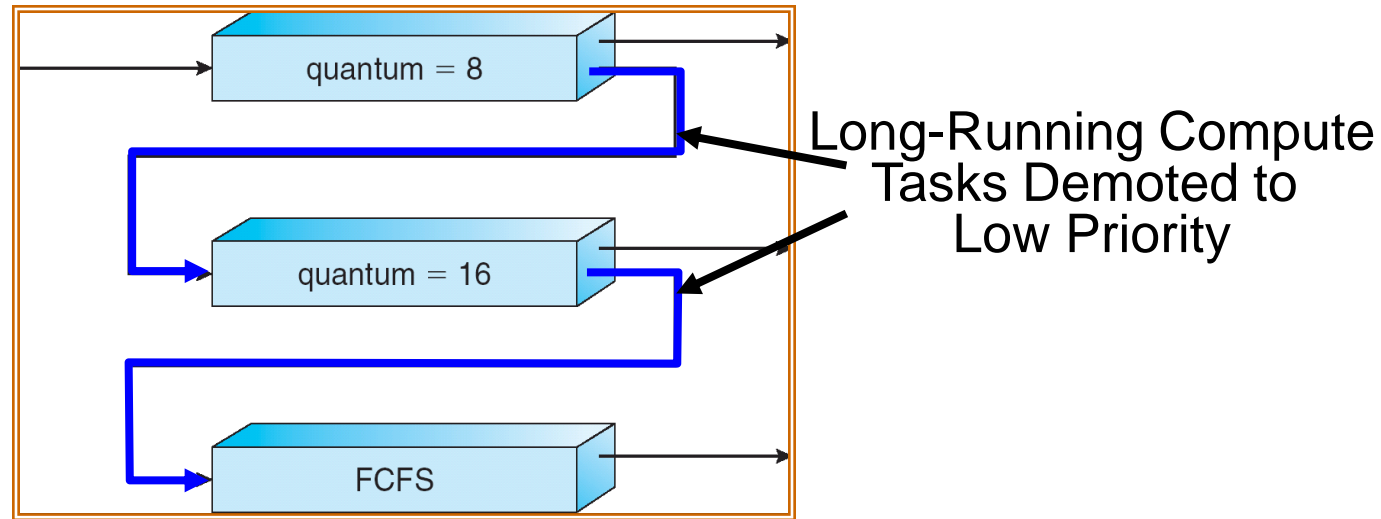
One Solution: Priority Donation



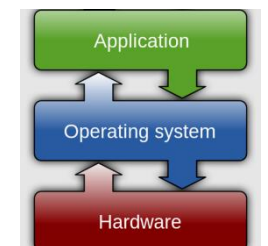
- Job 1 completes critical section and releases lock
- Job 3 acquires lock, runs again
- How does the scheduler know?



Are SRTF and MLFQ Prone to Starvation?

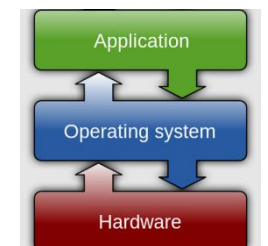


- In SRTF, long jobs are starved in favor of short ones
 - Same fundamental problem as priority scheduling
- MLFQ is an approximation of SRTF, so it suffers from the same problem



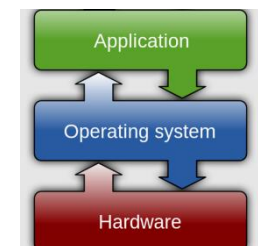
Announcements

- Project 1: deadline extended to Monday, March 31
- Assignment 2: due Monday, April 7



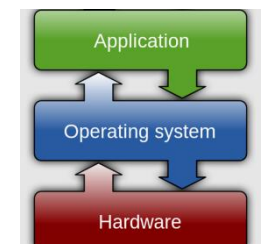
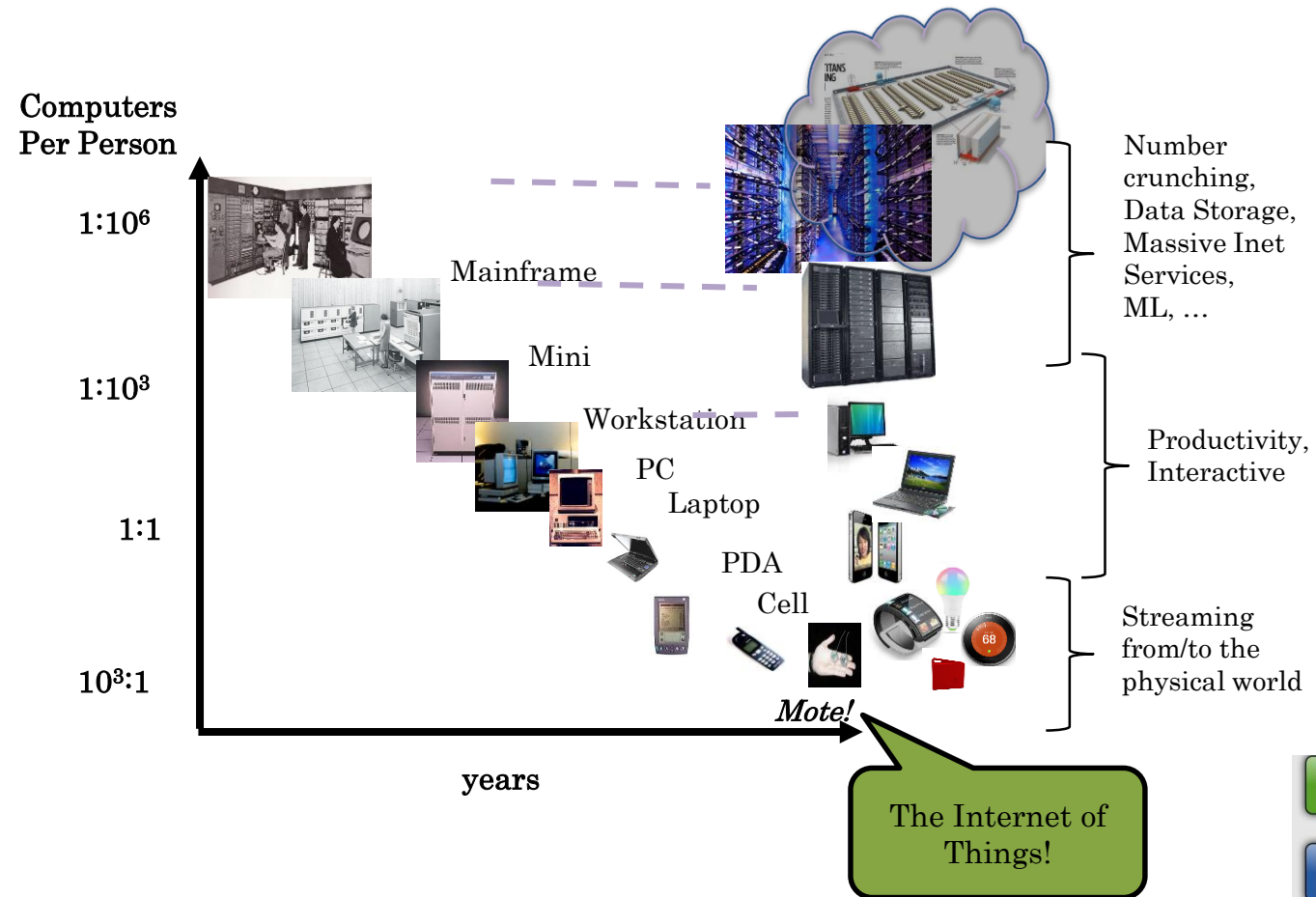
Cause for Starvation: Priorities?

- 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



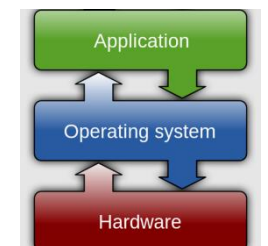
Recall: Changing Landscape...

Bell's Law: New computer class every 10 years

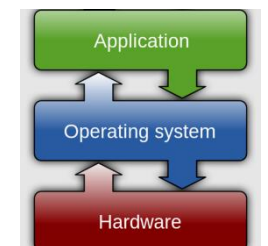


Changing Landscape of Scheduling

- 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, 95th percentile performance guarantees

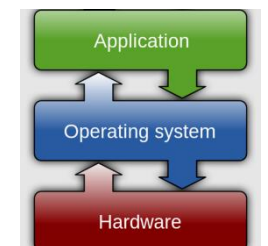


Does prioritizing some jobs necessarily starve those that aren't prioritized?



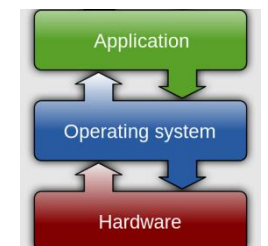
Key Idea: Proportional-Share Scheduling

- 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)

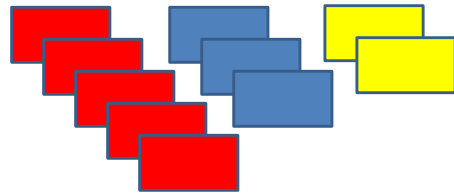


Today: Schedulers Prone to Starvation

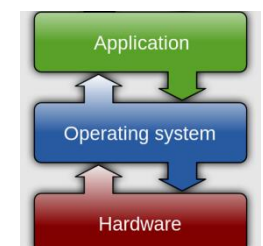
- What kinds of schedulers are prone to starvation?
- Of the scheduling policies we've studied, which are prone to starvation? And can we fix them?
- How might we design scheduling policies that avoid starvation entirely?
 - Arguably more relevant now than when CPU scheduling was first developed...



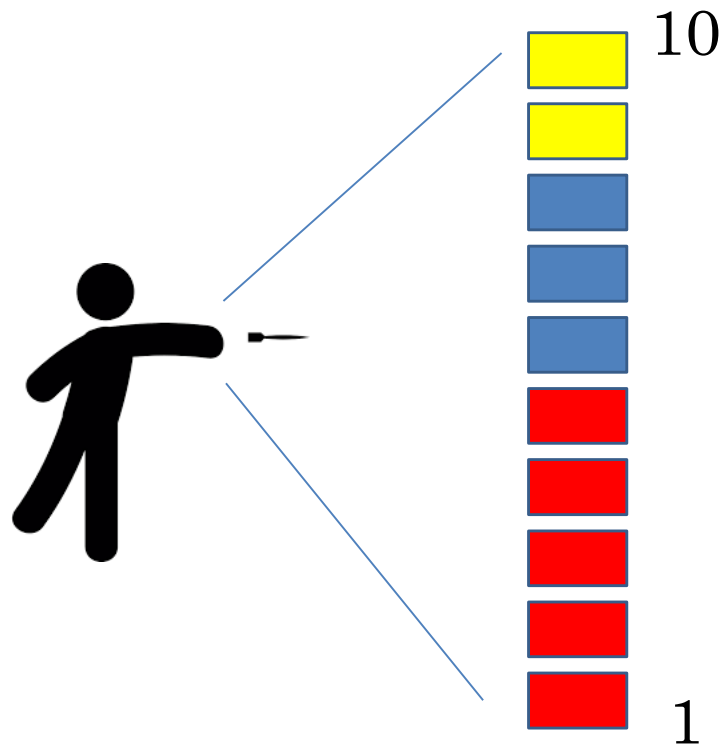
Lottery Scheduling



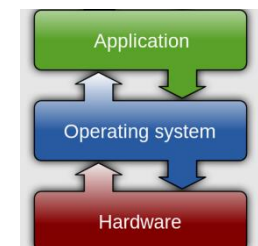
- Given a set of jobs (the mix), provide each with a share of a resource
 - e.g., 50% of the CPU for **Job A**, 30% for **Job B**, and 20% for **Job C**
- Idea: Give out tickets according to the proportion each should receive
- Every quantum (tick): draw one at random, schedule that job (thread) to run



Lottery Scheduling: Simple Mechanism



- $N_{ticket} = \sum N_i$
- Pick a number d in $1 \dots N_{ticket}$ 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 .



Unfairness

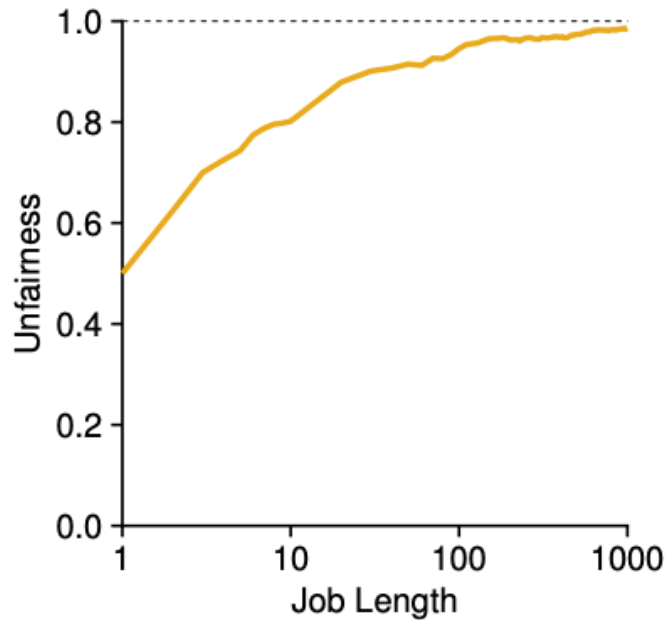
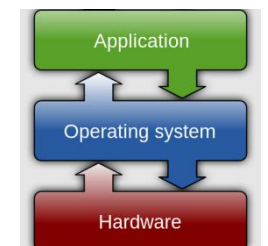


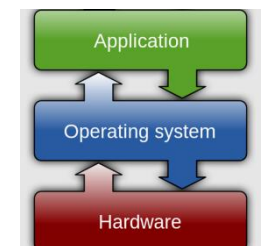
Figure 9.2: Lottery Fairness Study

- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%
- $U = \text{finish time of first} / \text{finish time of last}$
- As a function of run time



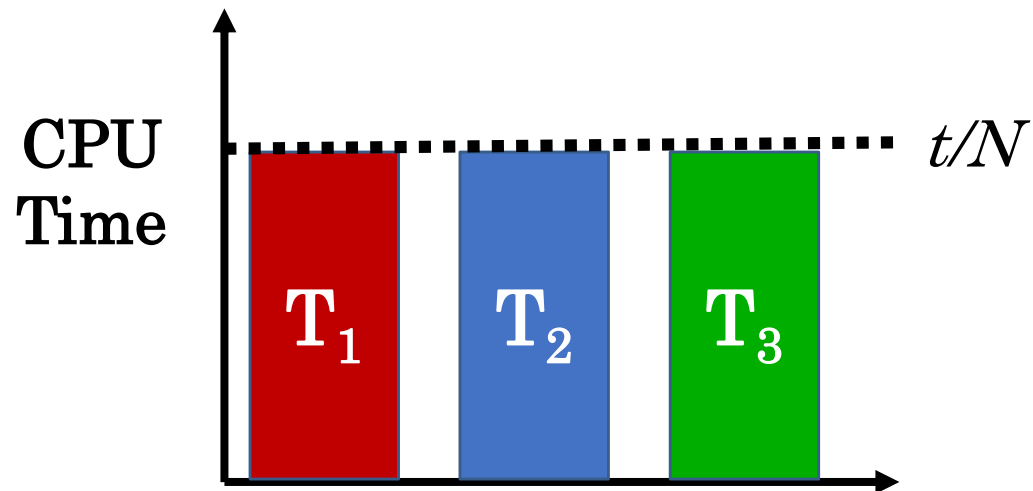
Stride Scheduling

- 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 has 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, ...

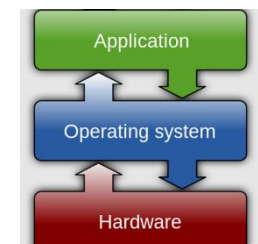


Linux Completely Fair Scheduler (CFS)

At *any* time t we would observe:

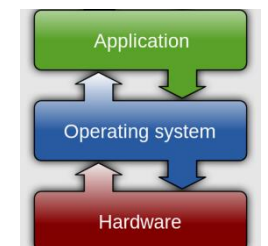
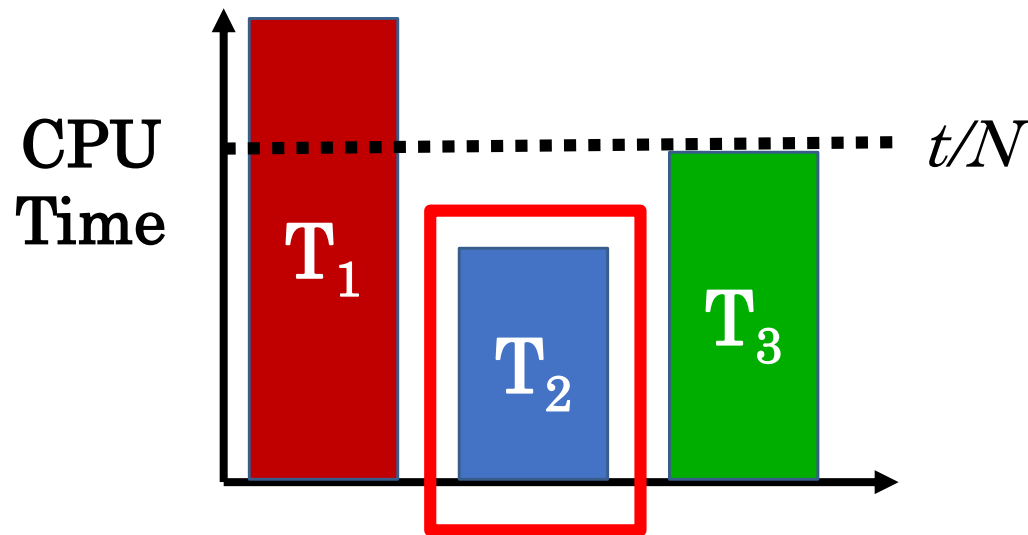


- Goal: Each process gets an equal share of CPU
- N threads “simultaneously” execute on $1/N$ th of CPU
- Can't do this with real hardware
 - OS needs to give out full CPU in time slices



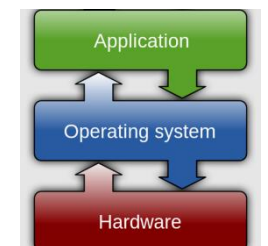
Linux Completely Fair Scheduler (CFS)

- Instead: track CPU time given to a thread so far
- Scheduling Decision:
 - “Repair” illusion of complete fairness
 - Choose thread with minimum CPU time
- Reset CPU time if thread goes to sleep and wakes back up



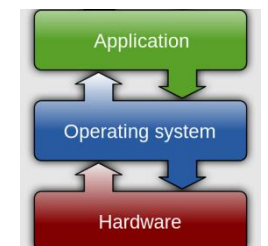
Linux CFS: Responsiveness

- In addition to fairness, we want low response time
- 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



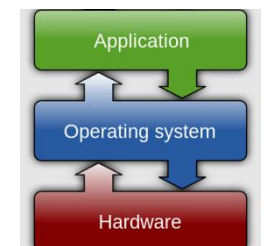
Linux CFS: Throughput

- 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

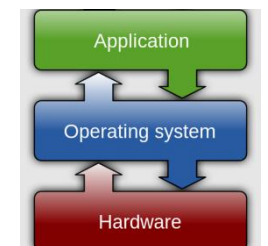
- The industrial operating systems of the 60s and 70's provided priority to enforced desired usage policies.
- When it was being developed at Berkeley, instead it provided ways to “be nice”.
- *nice* values range from -20 to 19
 - Negative values are “not nice”
 - If you wanted to let your friends get more time, you would nice up your job
- Schedule puts higher nice (lower priority) to sleep more ...



Linux CFS: Proportional Shares

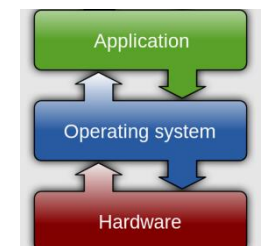
- What if we want to give more CPU to some and less to others (proportional share) ?
- Reuse *nice* value to reflect share, rather than priority
- Key Idea: Assign a weight w_i to each process i
- Basic equal share: $Q = \text{Target Latency} \cdot \frac{1}{N}$
- Weighted Share:

$$Q_i = \text{Target Latency} \cdot \left(w_i / \sum_p w_p \right)$$

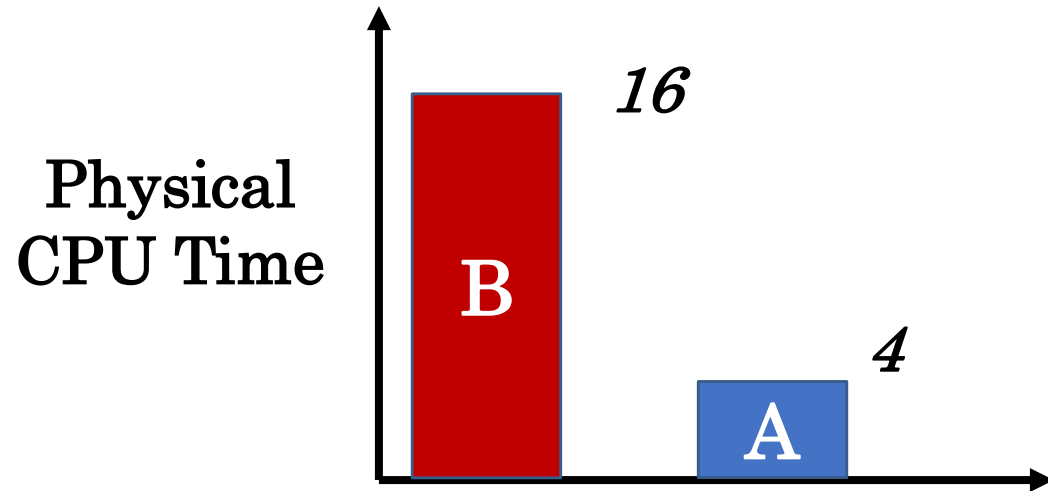


Linux CFS: Proportional Shares

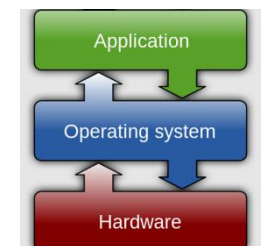
- Target Latency = 20ms
- Minimum Granularity = 1ms
- Two CPU-Bound Threads
 - Thread A has weight 1
 - Thread B has weight 4
- Time slice for A? 4 ms
- Time slice for B? 16 ms



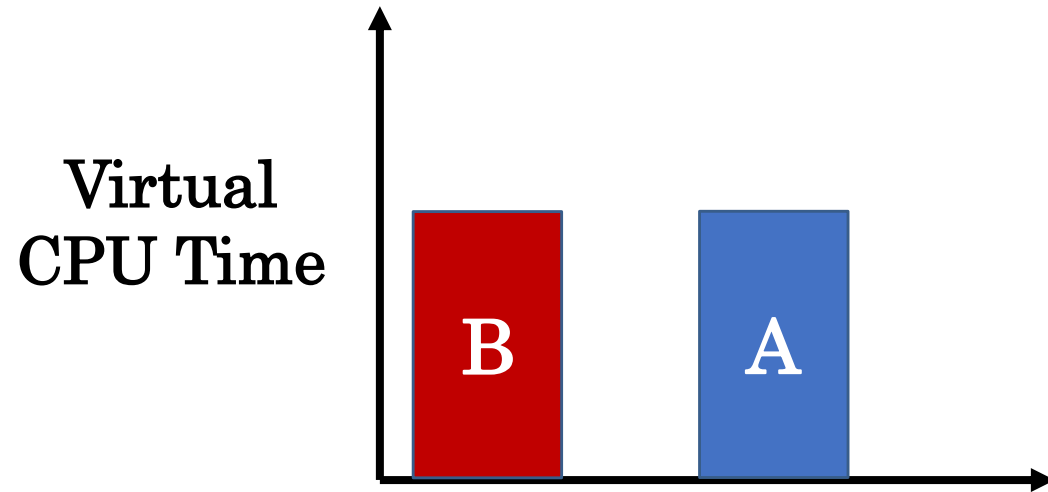
Linux CFS: Proportional Shares



- 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

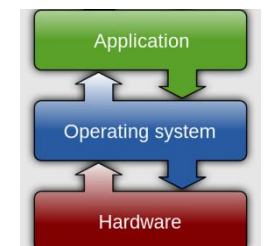


Linux CFS: Proportional Shares



Scheduler's Decisions are
based on Virtual CPU Time

- 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



Summary: Choosing the Right Scheduler

If You Care About:	Then Choose:
CPU Throughput	FCFS
Average 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

