

CS162
Operating Systems and
Systems Programming
Lecture 12

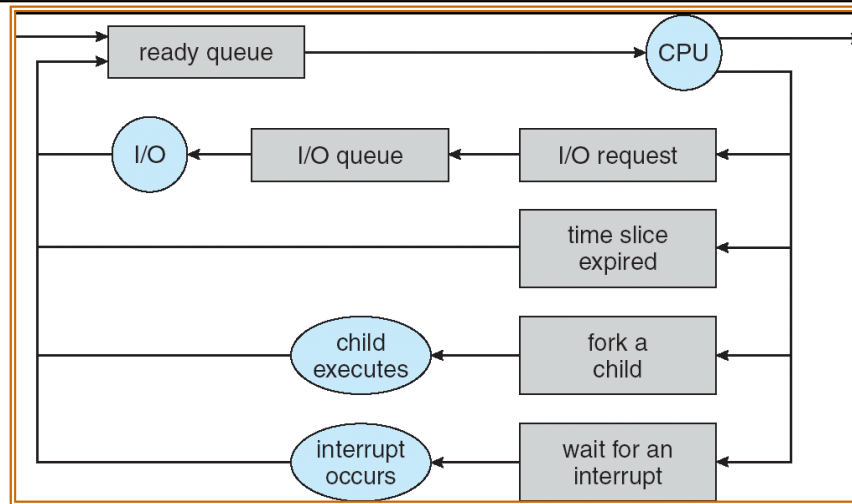
Scheduling 2:
Classic Policies (Con't), Case Studies,
Starvation, Priority Inversion

March 3rd, 2026

Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

Recall: Scheduling

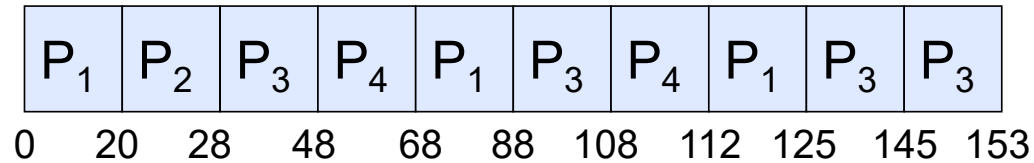


- Question: How is the OS to decide which of several tasks to take off a queue?
- **Scheduling**: deciding which threads are given access to resources from moment to moment
 - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

Recall: Example of RR with Time Quantum = 20

Example:	<u>Process</u>	<u>Burst Time</u>
	P_1	53
	P_2	8
	P_3	68
	P_4	24

– The Gantt chart is:



– Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$

$$P_2 = (20 - 0) = 20$$

$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P_4 = (48 - 0) + (108 - 68) = 88$$

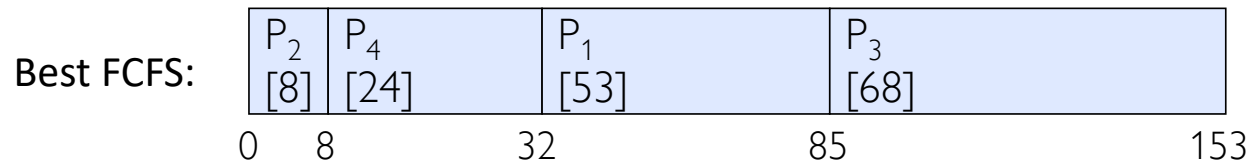
– Average waiting time = $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$

– Average completion time = $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

• Thus, Round-Robin Pros and Cons:

- Better for short jobs, Fair (+)
- Context-switching time adds up for long jobs (-)

Recall: Scheduling Example with Different Time Quantum



	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

What if we Knew the Future?

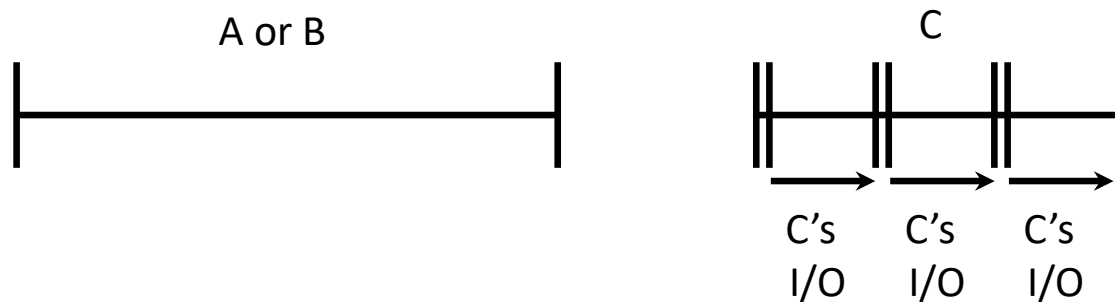
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
 - Run whatever job has least amount of computation to do
 - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
 - Preemptive version of SJF: if job arrives and has a shorter time to completion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



Discussion

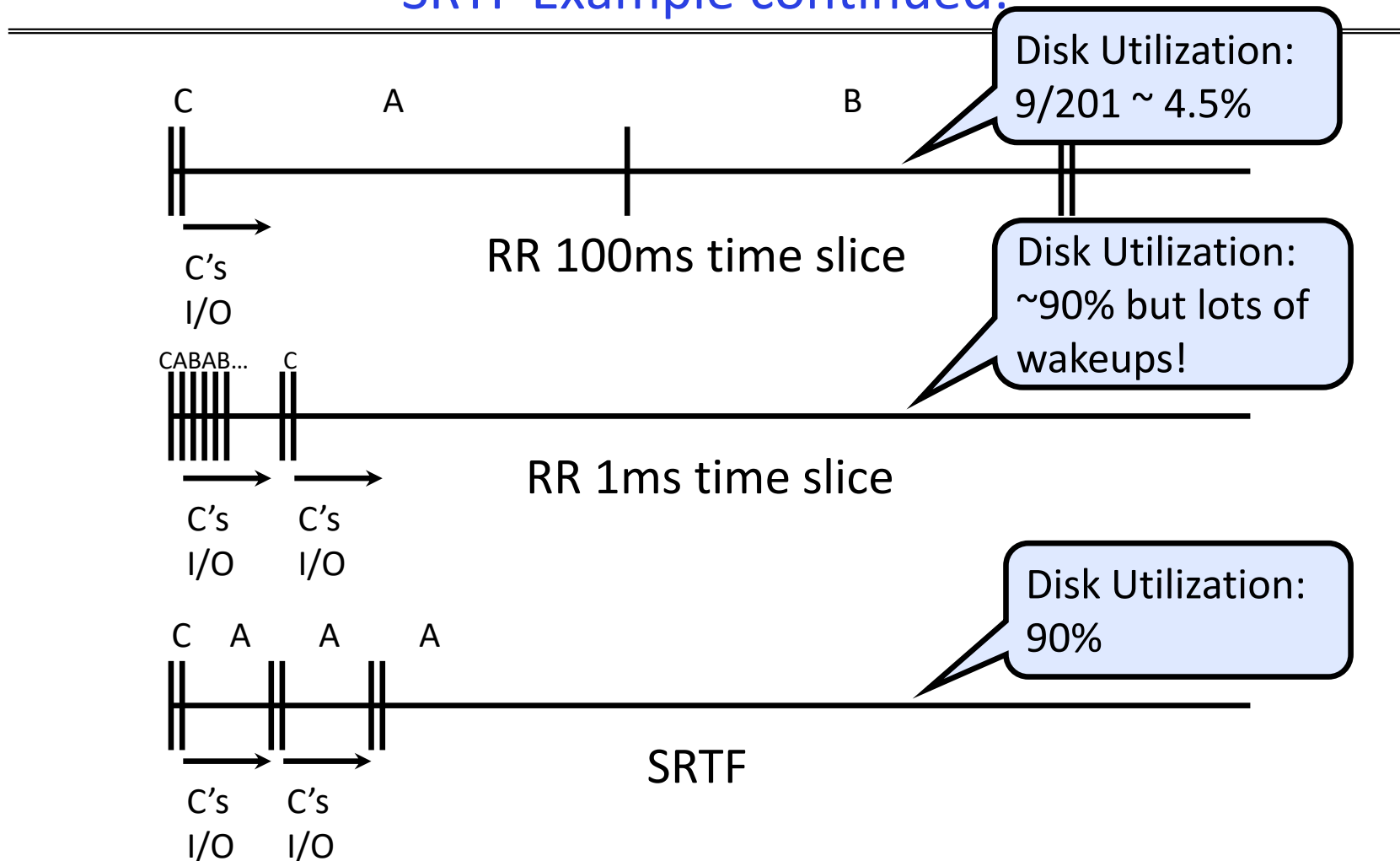
- SJF/SRTF are the best you can do at minimizing average response time
 - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS
 - What if all jobs the same length?
 - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
 - What if jobs have varying length?
 - » SRTF: short jobs not stuck behind long ones

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

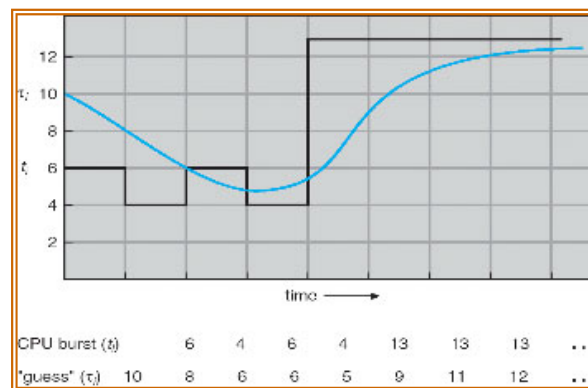
- Starvation
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run
- Somehow need to predict future
 - How can we do this?
 - Some systems ask the user
 - » When you submit a job, have to say how long it will take
 - » To stop cheating, system kills job if takes too long
 - 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

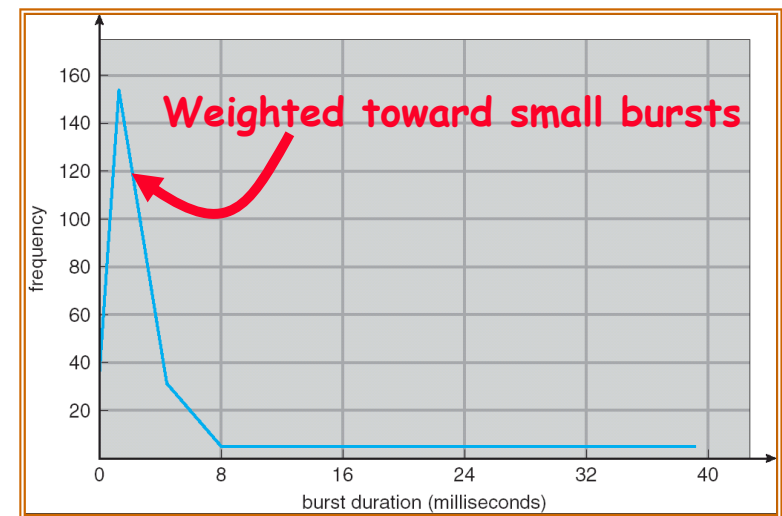
- **Adaptive:** Changing policy based on past behavior
 - CPU scheduling, in virtual memory, in file systems, etc
 - Works because programs have predictable behavior
 - » If program was I/O bound in past, likely in future
 - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated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

exponential averaging
 $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
with $(0 < \alpha \leq 1)$

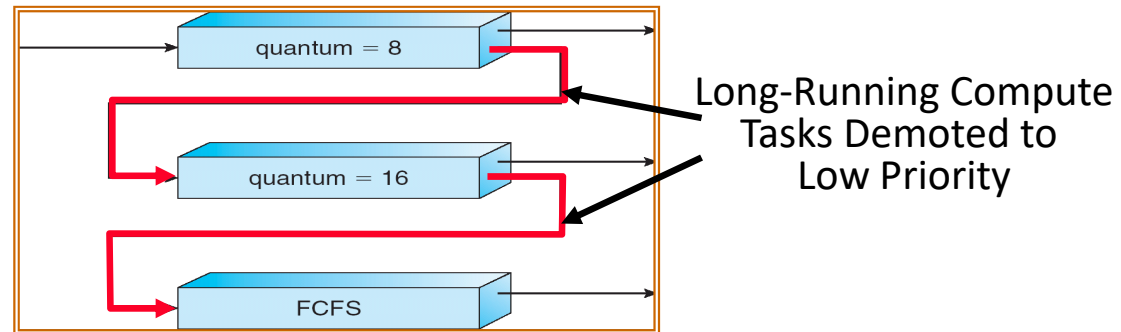


How to Handle Simultaneous Mix of Diff Types of Apps?

- 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 \Rightarrow Interactivity \Rightarrow 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)

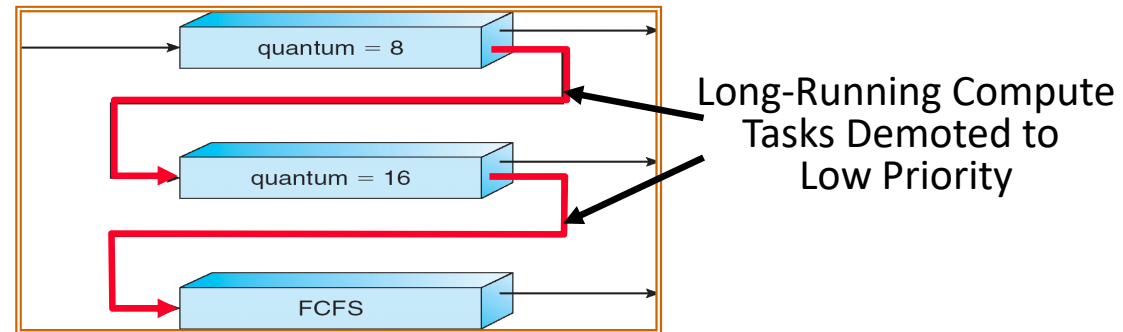


Multi-Level Feedback Scheduling



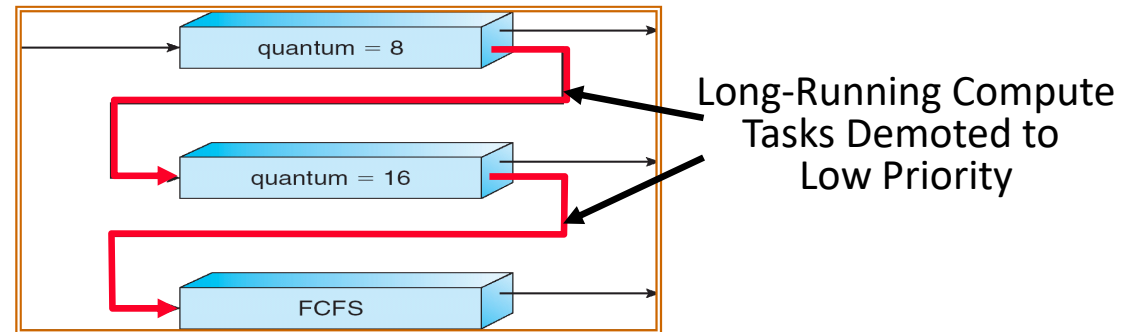
- Another method for exploiting past behavior (first use in CTSS)
 - Multiple queues, each with different priority
 - » Higher priority queues often considered “foreground” tasks
 - Each queue has its own scheduling algorithm
 - » e.g. foreground – RR, background – FCFS
 - » Sometimes multiple RR priorities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

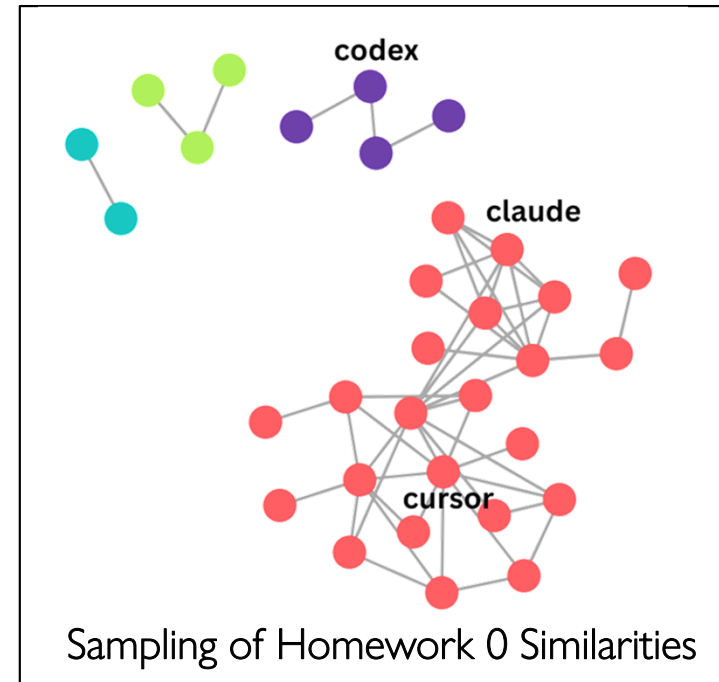
Scheduling Details



- **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!

Administrivia

- Midterm I:
 - Grading still in process, should be done in next day or so!
 - Solutions are up off the resources page.
- Exam was long? Probably
 - Let's get you to where interpreting code is much easier \Rightarrow practice
 - Please take time to understand Problems 3 and 5
 - Get informed on GDB. We had many people who were unable to answer GDB question on Midterm, suggesting they haven't used it.
- Reminder: Use of LLMs are strictly forbidden
 - We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders
 - We also compare project submissions against AI-generated solutions.
 - Do not do it! This is strictly against policy.
 - We have begun to submit Student Conduct cases for some blatant examples of violating policy.



CS 162 Collaboration Policy



- Explaining a concept to someone in another group
 - Discussing algorithms/testing strategies with other groups
 - Discussing debugging approaches with other groups
 - Searching online for generic algorithms (e.g., hash table)
-



- Sharing code or test cases with another group
- Copying OR reading another group's code or test cases
- Copying OR reading online code or test cases from prior years
- Helping someone in another group to debug their code

USE OF LLMs IS STRICTLY FORBIDDEN: Like having a friend do your work!

- » Any time you see misconduct with a person, above, you have same issue using AI
- » Any adaptation/copy/paste of code from a google search (which uses AI) is also violation of policy
- » Use of auto-suggestions from your IDE is suspect. **TURN THESE OFF, LEARN C ON YOUR OWN**

- Go to office hours for help. Do not risk a student-conduct filing,
- If a friend helps you, **BOTH OF YOU ARE VIOLATING POLICY**

Scheduling Fairness

- What about fairness?
 - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
 - » long running jobs may never get CPU
 - » Urban legend: In Multics, shut down machine, found 10-year-old job ⇒
Ok, probably not...
 - Must give long-running jobs a fraction of CPU even when there are shorter jobs to run
 - Tradeoff: fairness gained by hurting avg response time!

Scheduling Fairness

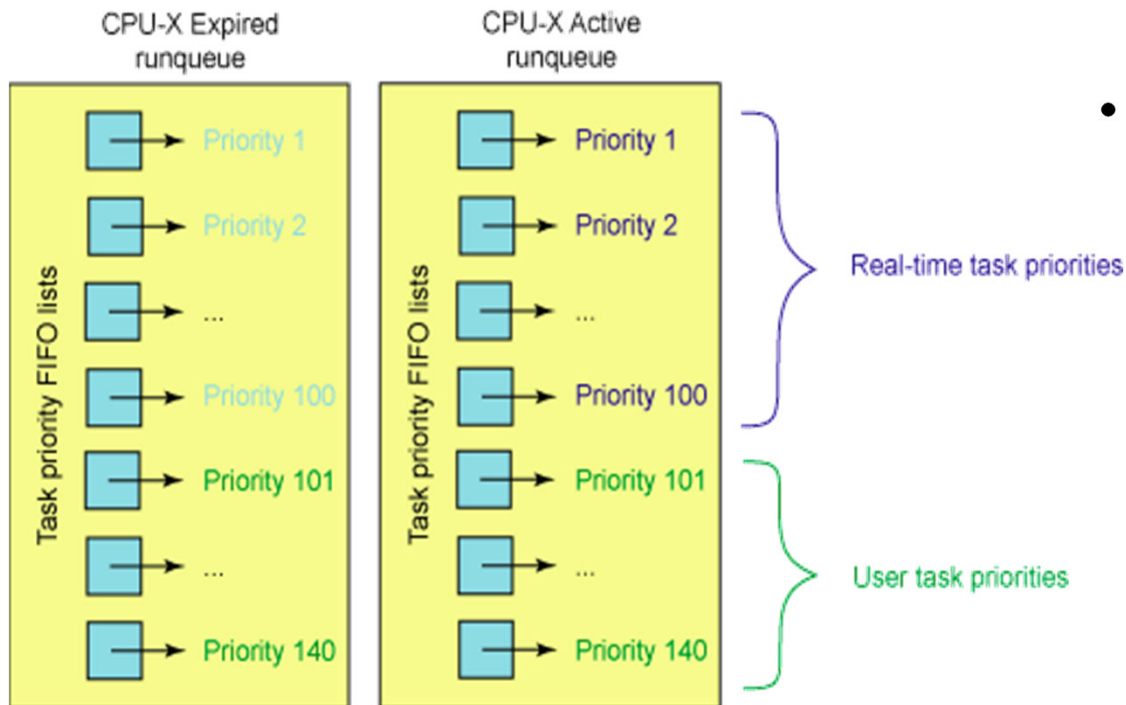
- How to implement fairness?
 - Could give each queue some fraction of the CPU
 - » What if one long-running job and 100 short-running ones?
 - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
 - Could increase priority of jobs that don't get service
 - » What is done in some variants of UNIX
 - » This is ad hoc—what rate should you increase priorities?
 - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

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 \Rightarrow higher priority (for realtime values)
 - Highest priority value \Rightarrow 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
 - » $p \rightarrow \text{sleep_avg} = \text{sleep_time} - \text{run_time}$
 - » Higher `sleep_avg` \Rightarrow 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

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) but can be **task groups** (also Linux)
- Note: 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)
 - Hardware supported interleaving – parallel within single pipeline

Multi-Core Scheduling

- Algorithmically, not a huge difference from single-core scheduling
- Implementation-wise, helpful to have *per-core* scheduling data structures
 - Cache coherence
 - Scheduling of per-core queue can be synchronized with interrupt enable/disable
- *Affinity scheduling*: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
 - Cache reuse, branch prediction
 - Example for $O(1)$ scheduler: 1 set of queues/core with background rebalancing

Recall: *Spinlocks for multiprocessing*

- Spinlock implementation:

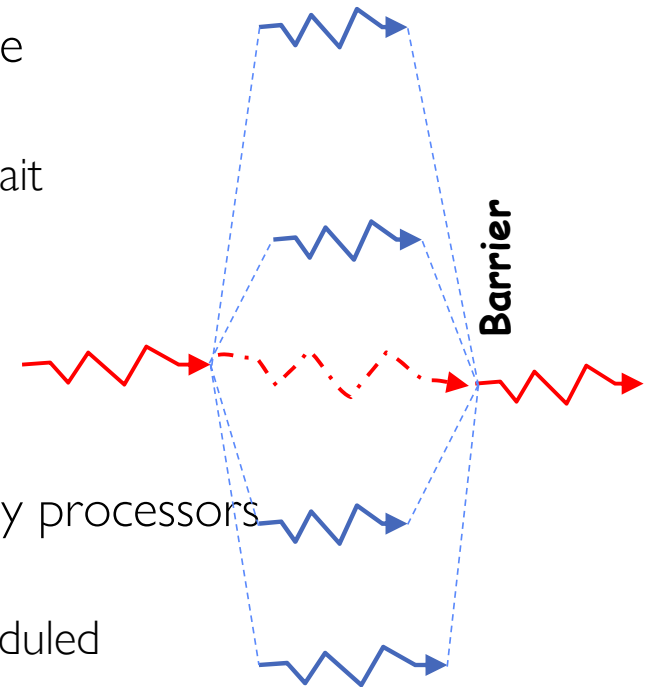
```
int value = 0; // Free
Acquire() {
    while (test&set(&value)) {}; // spin while busy
}
Release() {
    value = 0; // atomic store
}
```

- 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)
 - Multiple phases of parallel and serial execution
- Additionally: 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



Ensuring Progress

- Starvation: thread fails to make progress for an indefinite period of time
- Starvation \neq Deadlock 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...

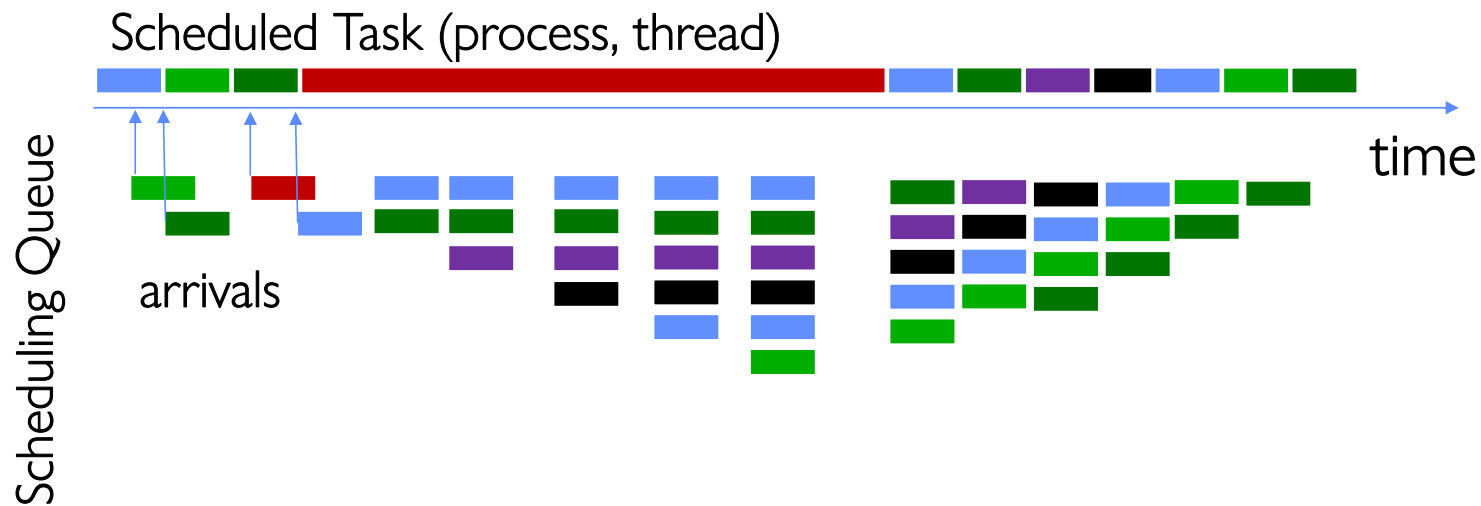
Strawman: Non-Work-Conserving Scheduler

- 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?

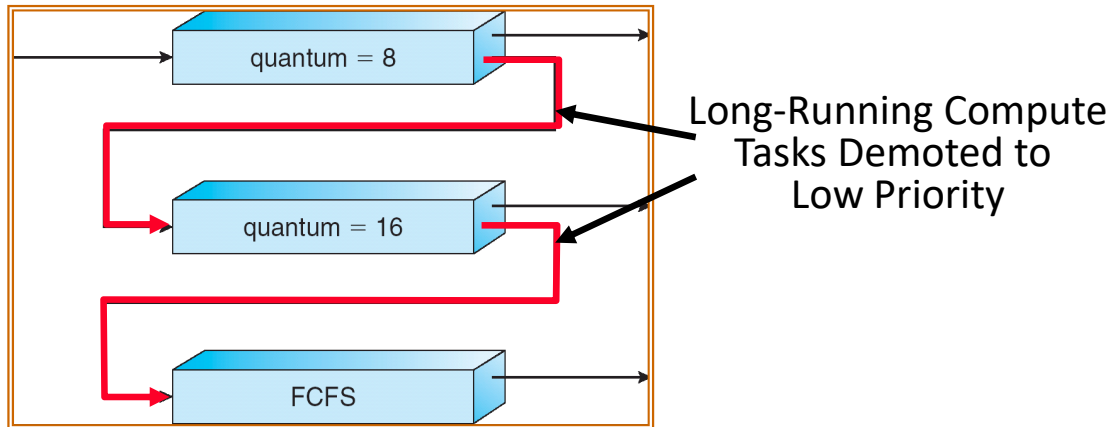


- If a task never yields (e.g., goes into an infinite loop), then other tasks don't get to run
- Problem with all non-preemptive schedulers...
 - And early personal OSes such as original MacOS, Windows 3.1, etc

Is Round Robin (RR) Prone to Starvation?

- Each of N processes gets $\sim 1/N$ 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... (if you give up your time slot early, you don't get the time back!)

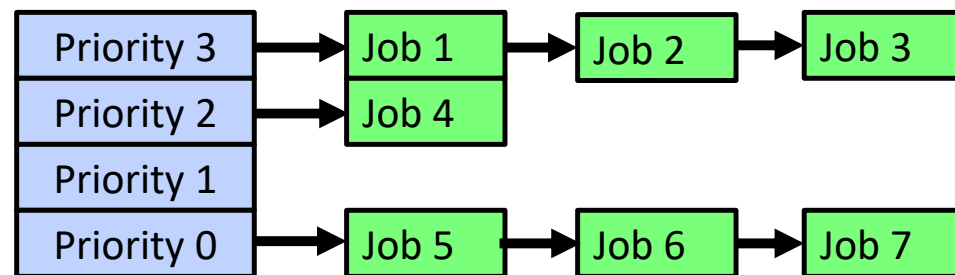
Are SRTF and MLFQ Prone to Starvation?



- 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

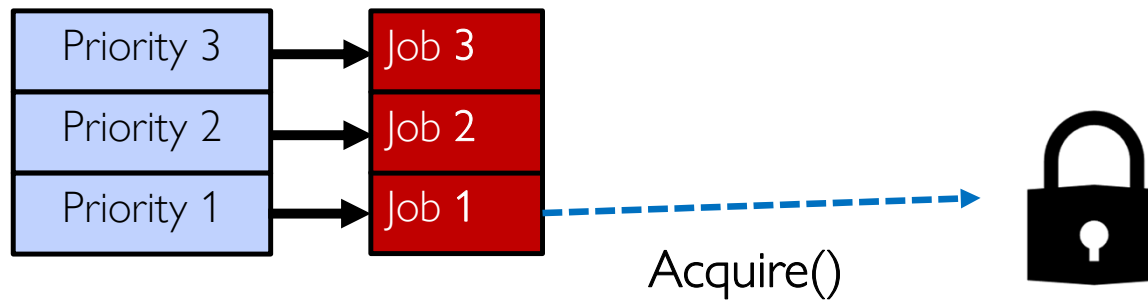
Is Priority Scheduling Prone to Starvation?

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



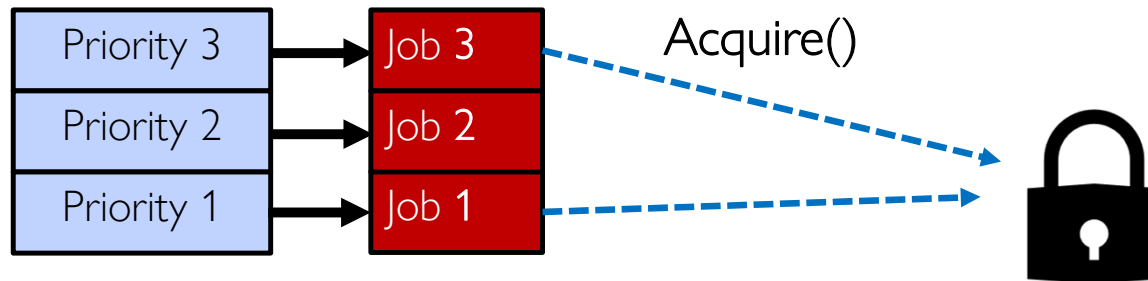
- But there are more serious problems as well...
 - Priority inversion: even high priority threads might become starved

Priority Inversion



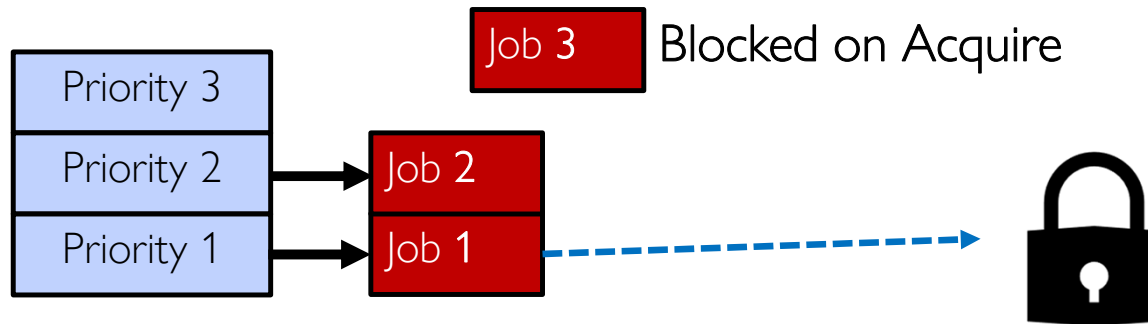
- At this point, which job does the scheduler choose?
- Job 3 (Highest priority)

Priority Inversion



- Job 3 attempts to acquire lock held by Job 1

Priority Inversion



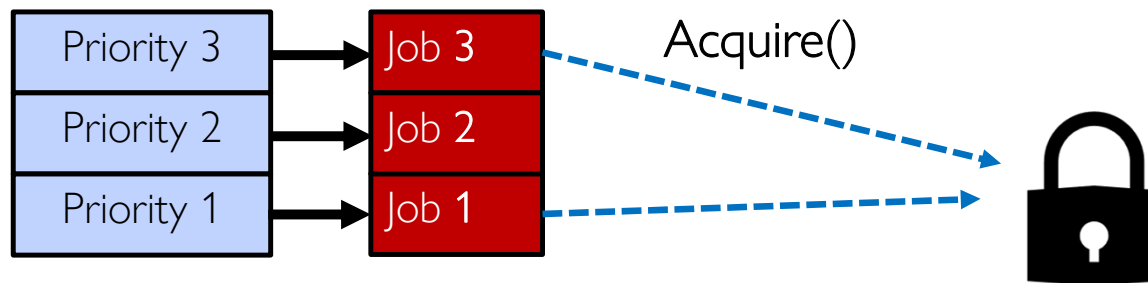
- At this point, which job does the scheduler choose?
- Job 2 (Medium Priority)
- Priority Inversion

Priority Inversion

- Clear Example (textbook?) of priority inversion:
 - 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”?

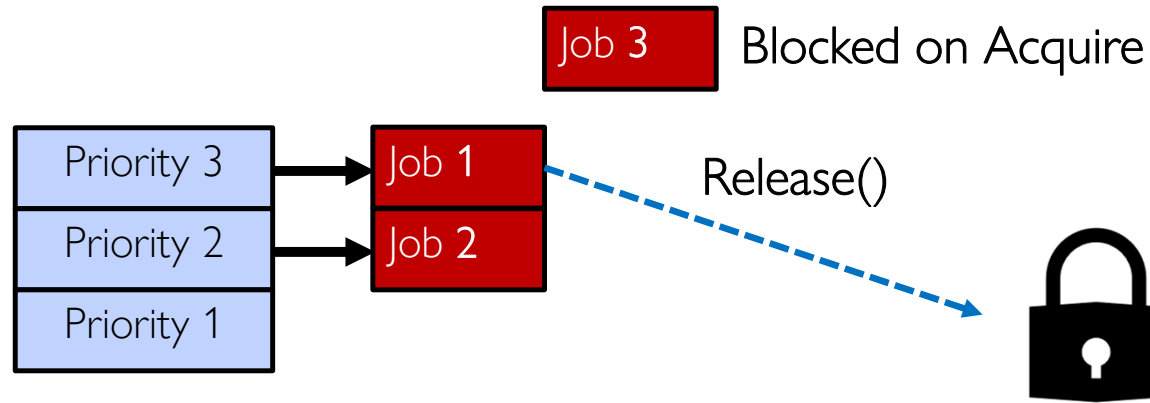


One Solution: Priority Donation/Inheritance



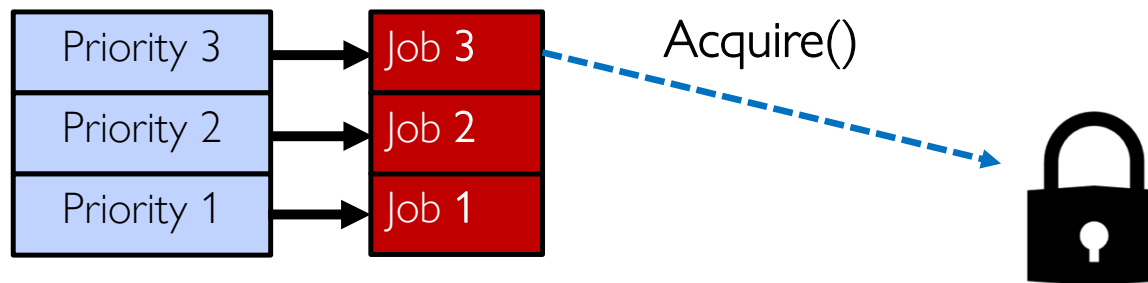
- Job 3 temporarily grants Job 1 its “high priority” to run on its behalf

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
- How does the scheduler know?



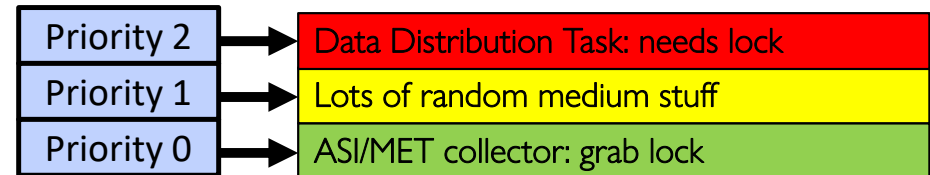
Case Study: Martian Pathfinder Rover

- 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:



- 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!

Conclusion

- **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
- **Priority Inversion**
 - A higher-priority task is prevented from running by a lower-priority task
 - Often caused by locks and through the intervention of a middle-priority task
- **Next Time: Proportional Share Scheduling**
 - 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)