

# **ECE 350**

## **Real-time**

## **Operating**

## **Systems**



# Lecture 4: Uniprocessor Scheduling

---

Prof. Seyed Majid Zahedi

<https://ece.uwaterloo.ca/~smzahedi>

# Outline

---

- History
- Definitions
  - Response time, throughput, scheduling policy, ...
- Uniprocessor scheduling policies
  - FCFS, SJF/SRTF, RR, ...

# A Bit of History on Scheduling

---

- By year 2000, scheduling was considered a solved problem

*“And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”*

Linus Torvalds, 2001<sup>[1]</sup>

- End to Dennard scaling in 2004, led to multiprocessor era
  - Designing new (multiprocessor) schedulers gained traction
  - Energy efficiency became top concern
- In 2016, it was shown that bugs in Linux kernel scheduler could cause up to **138x slowdown** in some workloads with proportional energy waist<sup>[2]</sup>

[1] L. Torvalds. The Linux Kernel Mailing List. <http://tech-insider.org/linux/research/2001/12/5.html>, Feb. 2001.

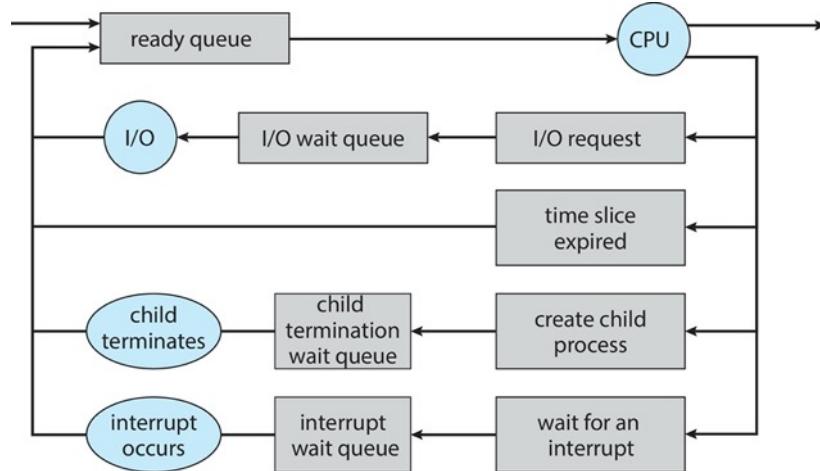
[2] Lozi, Jean-Pierre, et al. "The Linux scheduler: a decade of wasted cores." Proceedings of the Eleventh European Conference on Computer Systems. 2016.

# Definitions

---

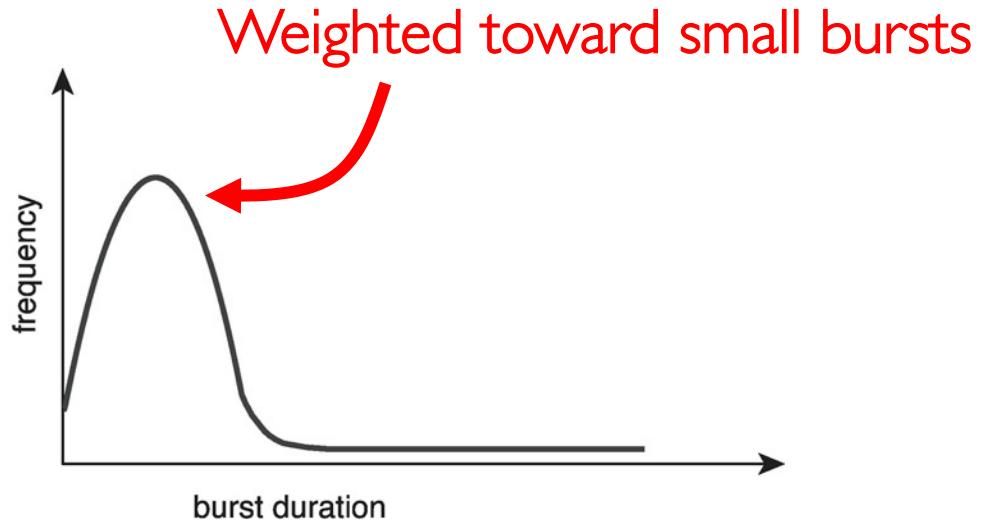
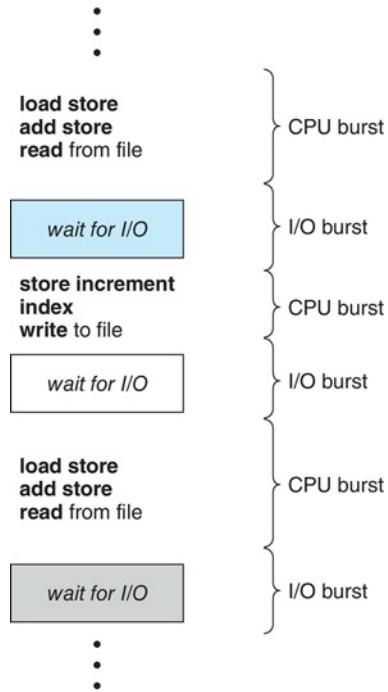
- Task, thread, process, job: unit of work
  - E.g., mouse click, web request, shell command, etc.)
- Workload: set of tasks
- Scheduling algorithm: takes workload as input, decides which tasks to do first
- Overhead: amount of extra work that is done by scheduler
- Preemptive scheduler: CPU can be taken away from a running task
- Work-conserving scheduler: CPUs won't be left idle if there are ready tasks to run
  - For non-preemptive schedulers, work-conserving is not always better (why?)
- Only preemptive, work-conserving schedulers to be considered in this lecture!

# Recall: CPU Scheduling



- Earlier, we talked about life-cycle of threads
  - Threads work their way from ready to running to various waiting queues
- **Question:** how does OS decide which thread to dequeue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- **Scheduling:** deciding which thread gets resource from moment to moment

# Execution Model



- Programs alternate between bursts of CPU and I/O
  - Use CPU for some period, then do I/O, then use CPU again, etc.
- CPU scheduling is about choosing thread which gets CPU for its next CPU burst
- With preemption, thread may be forced to give up CPU before finishing its burst

# CPU Scheduling Assumptions

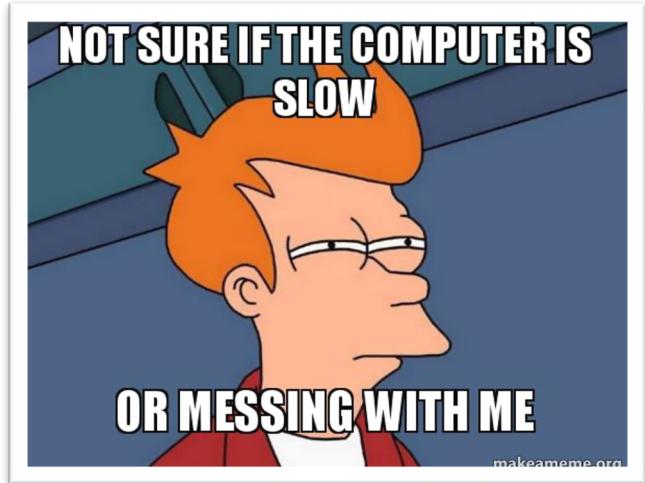
---

- There are many implicit assumptions for CPU scheduling
  - One program per user
  - One thread per program
  - Programs are independent
- These may not hold in all systems, but they simplify the problem
- High-level goal is to divide CPU time to optimize some desired properties

# Performance

---

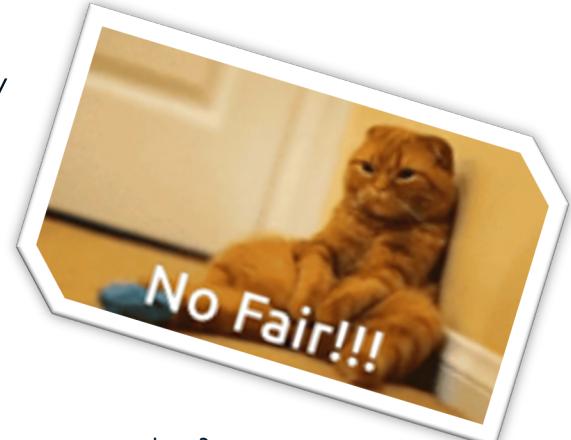
- Minimize **average response time**
  - Minimize avg. elapsed time to finish tasks
  - Response time is what users see
    - E.g., time to echo a keystroke in editor
    - E.g., time to compile a program
  - **Real-time tasks** must meet **deadlines**  
(more on this later)
- Maximize **throughput**
  - Maximize tasks per time unit (e.g., tasks per second)
  - Related to response time, but not identical
    - Minimizing response time could lead to more context switching which will than hurt throughput (more on this later)
  - Two parts to maximizing throughput
    - Minimize overhead (e.g., context-switching)
    - Efficient use of resources (e.g., CPU, disk, memory, etc.)



# Fairness

---

- Share CPU time among *users* in some *equitable* way
- What does equitable mean?
  - Equal share of CPU time?
    - What if some tasks don't need their full share?
  - Minimize variance in worst case performance?
    - What if some tasks were running when no one else was running?
- Who are users? Actual users or programs?
  - If A runs one thread and B runs five, B could get five times as much CPU time on many OS's
- Fairness is not minimizing average response time
  - Improving average response time could make system less fair  
(more on this later)



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

- First-come, first-served (FCFS)
  - Also “first-in, first-out” (FIFO)
  - In early systems, FCFS meant one program scheduled until done (including its I/O activities)
  - Now, it means that program keeps CPU until the end of its CPU burst

- Example

| <u>Thread</u> | <u>CPU Burst Time</u> |
|---------------|-----------------------|
| $T_1$         | 24                    |
| $T_2$         | 3                     |
| $T_3$         | 3                     |

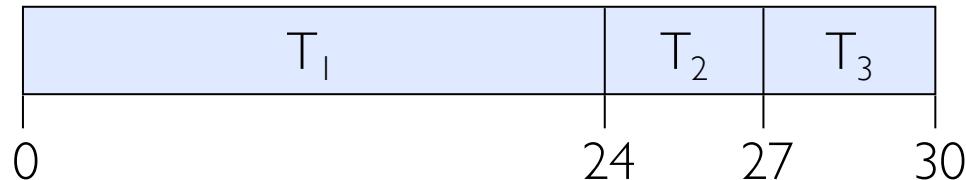
- Suppose threads arrive in order: $T_1, T_2, T_3$



# FCFS Scheduling (cont.)

---

- Example continued



- Wait time for T<sub>1</sub> is 0, for T<sub>2</sub> is 24, and for T<sub>3</sub> is 27
- Average wait time is  $(0 + 24 + 27)/3 = 17$
- Average response time is  $(24 + 27 + 30)/3 = 27$

# FCFS Scheduling (cont.)

---

- If threads arrive in order:  $T_2, T_3, T_1$ , then we have

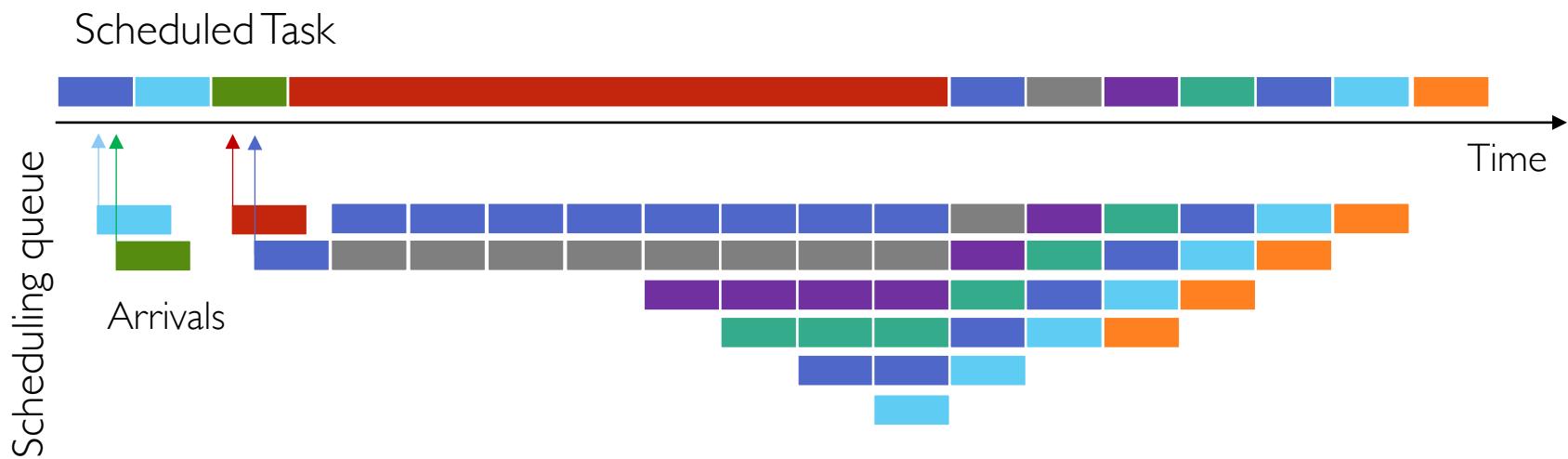


- Wait time for  $T_1$  is 6, for  $T_2$  is 0, and for  $T_3$  is 3
- Average wait time is  $(6 + 0 + 3)/3 = 3$
- Average response time is  $(3 + 6 + 30)/3 = 13$
- Average wait time is much better (before it was 17)
- Average response time is better (before it was 27)
- Pros and cons of FCFS
  - + Simple
  - - Short tasks get stuck behind long ones

# Convoy Effect

---

- With FCFS, convoys of small tasks tend to build up when a large one is running



# Round Robin (RR) Scheduling

---

- FCFS is potentially bad for short tasks!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand...
- Round robin
  - Each thread gets small unit of CPU time, called *time quantum* (usually 10-100 milliseconds)
  - Once quantum expires, thread is preempted and added to end of ready queue
  - $N$  threads in ready queue and time quantum is  $q \Rightarrow$ 
    - Each thread gets  $1/N$  of CPU time in chunks of **at most**  $q$  time units
    - **No thread waits more than  $(N-1)q$  time units**

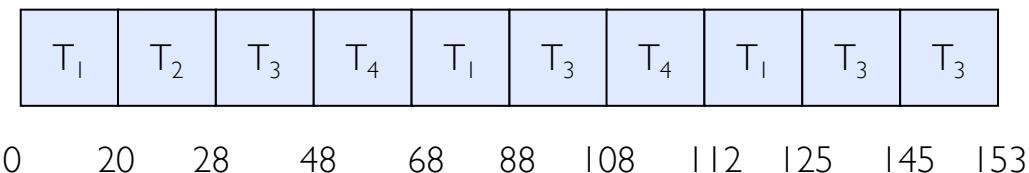


# Example: RR with Time Quantum of 20

---

- Example

| <u>Thread</u>  | <u>Burst Time</u> |
|----------------|-------------------|
| T <sub>1</sub> | 53                |
| T <sub>2</sub> | 8                 |
| T <sub>3</sub> | 68                |
| T <sub>4</sub> | 24                |

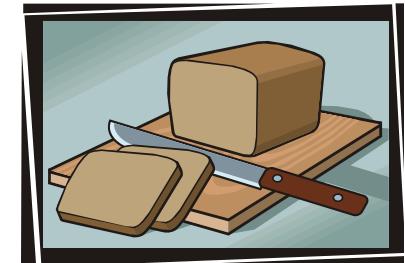


- Wait time for T<sub>1</sub> = (68 - 20) + (112 - 88) = 72  
$$T_2 = (20 - 0) = 20$$
$$T_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$T_4 = (48 - 0) + (108 - 68) = 88$$
- Average wait time is  $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$
- Average response time is  $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

# Round Robin Discussion

---

- Pros and cons of RR
  - + Better for short tasks, fair
  - - Context-switching time adds up for long tasks
- How does performance change with time quantum?
  - What if it's too long?
    - Response time suffers!
  - What if it's too short?
    - Throughput suffers!
  - What if it's infinite ( $\infty$ )?
    - RR  $\Rightarrow$  FCFS
  - Time quantum must be long compared to context switching time, otherwise overhead will be too high



# Round Robin Time Quantum

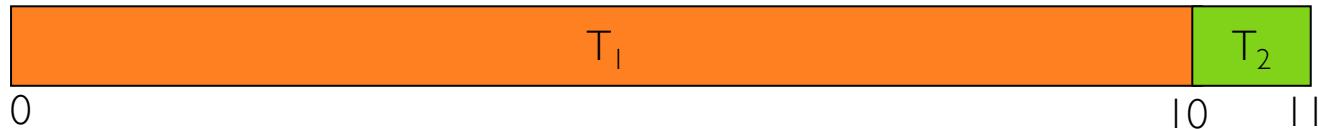
---

- Assume there is no context switching overhead
- What happens when we decrease  $Q$ ?
  1. Avg. response time always decreases or stays the same
  2. Avg. response time always increases or stays the same
  3. Avg. response time can increase, decrease, or stays the same

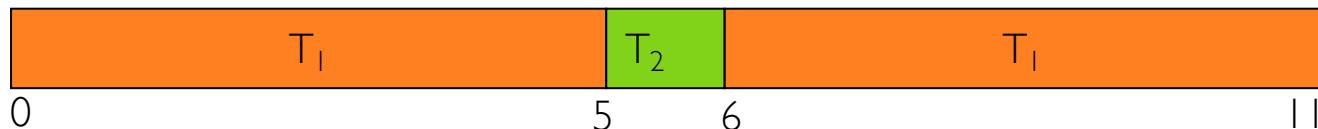
# Decrease Response Time

---

- $T_1$ : burst time 10
- $T_2$ : burst time 1
- $Q = 10$



- Average response time =  $(10 + 11)/2 = 10.5$
- $Q = 5$

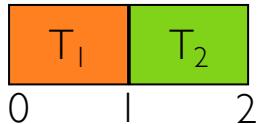


- Average response time =  $(6 + 11)/2 = 8.5$

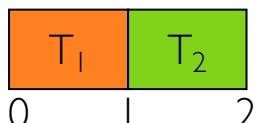
# Same Response Time

---

- $T_1$ : burst time 1
- $T_2$ : burst time 1
- $Q = 10$



- Average response time =  $(1 + 2)/2 = 1.5$
- $Q = 1$

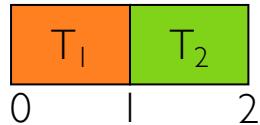


- Average response time =  $(1 + 2)/2 = 1.5$

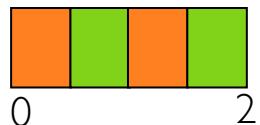
# Increase Response Time

---

- $T_1$ : burst time |
- $T_2$ : burst time |
- $Q = |$



- Average response time =  $(1 + 2)/2 = 1.5$
- $Q = 0.5$



- Average response time =  $(1.5 + 2)/2 = 1.75$

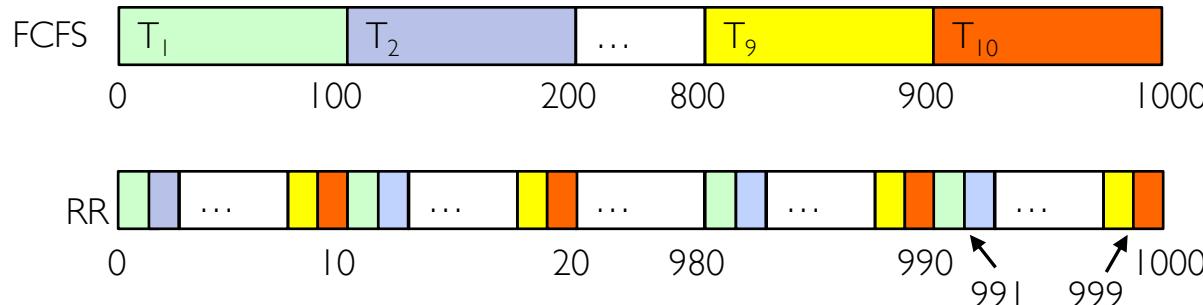
# Round Robin Discussion (cont.)

---

- Actual choices of time quantum
  - Initially, UNIX time quantum was one second
    - Worked ok when UNIX was used by one or two users
    - What if you use text editor while there are three compilations going on?
      - It takes 3 seconds to echo each keystroke!
  - Need to balance short-task performance and long-task throughput
    - Typical time quantum today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

# FCFS vs. RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Suppose there are 10 tasks, each take 100s of CPU time, RR quantum is 1s



- Completion times

| Task # | FCFS | RR   |
|--------|------|------|
| 1      | 100  | 991  |
| 2      | 200  | 992  |
| ...    | ...  | ...  |
| 9      | 900  | 999  |
| 10     | 1000 | 1000 |

# FCFS vs. RR (cont.)

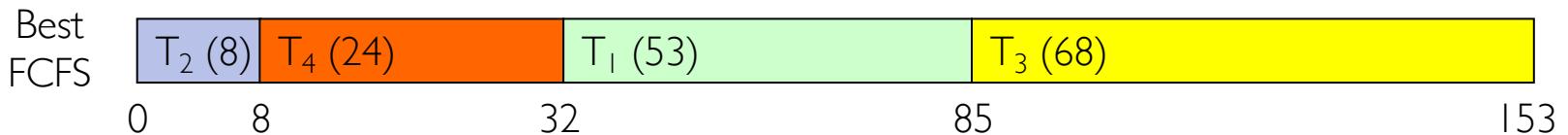
---

- Completion times

| Task # | FCFS | RR   |
|--------|------|------|
| 1      | 100  | 991  |
| 2      | 200  | 992  |
| ...    | ...  | ...  |
| 9      | 900  | 999  |
| 10     | 1000 | 1000 |

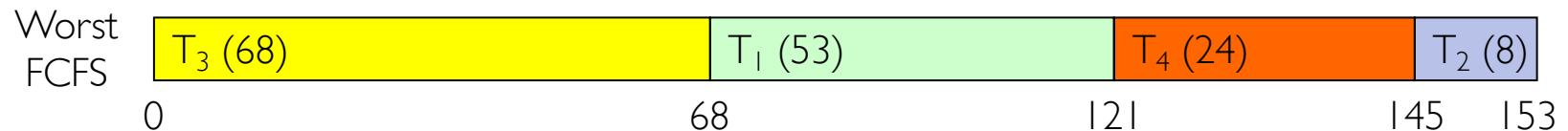
- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  - Bad when all threads have the same length
- Also, cache must be shared between all tasks with RR but can be devoted to each task with FIFO
  - Total time for RR is longer even for zero-cost context switching!

# Earlier Example: RR vs. FCFS, Effect of Different Time Quanta



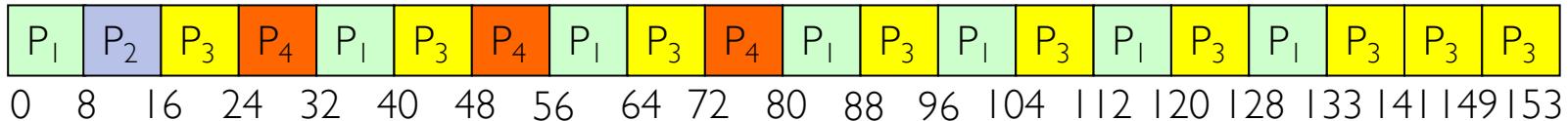
|               | Quantum    | T1 | T2 | T3  | T4 | Average |
|---------------|------------|----|----|-----|----|---------|
| Wait Time     | Best FCFS  | 32 | 0  | 85  | 8  | 31½     |
|               | 1          |    |    |     |    |         |
|               | 5          |    |    |     |    |         |
|               | 8          |    |    |     |    |         |
|               | 10         |    |    |     |    |         |
|               | 20         |    |    |     |    |         |
|               | Worst FCFS |    |    |     |    |         |
| Response Time | Best FCFS  | 85 | 8  | 153 | 32 | 69½     |
|               | 1          |    |    |     |    |         |
|               | 5          |    |    |     |    |         |
|               | 8          |    |    |     |    |         |
|               | 10         |    |    |     |    |         |
|               | 20         |    |    |     |    |         |
|               | Worst FCFS |    |    |     |    |         |

# Earlier Example: RR vs. FCFS, Effect of Different Time Quanta (cont.)



|               | Quantum    | T1  | T2  | T3  | T5  | Average |
|---------------|------------|-----|-----|-----|-----|---------|
| Wait Time     | Best FCFS  | 32  | 0   | 85  | 8   | 31¼     |
|               | 1          |     |     |     |     |         |
|               | 5          |     |     |     |     |         |
|               | 8          |     |     |     |     |         |
|               | 10         |     |     |     |     |         |
|               | 20         |     |     |     |     |         |
|               | Worst FCFS | 68  | 145 | 0   | 121 | 83½     |
| Response Time | Best FCFS  | 85  | 8   | 153 | 32  | 69½     |
|               | 1          |     |     |     |     |         |
|               | 5          |     |     |     |     |         |
|               | 8          |     |     |     |     |         |
|               | 10         |     |     |     |     |         |
|               | 20         |     |     |     |     |         |
|               | Worst FCFS | 121 | 153 | 68  | 145 | 121¾    |

# Earlier Example: RR vs. FCFS, Effect of Different Time Quanta (cont.)



|               | Quantum    | T1  | T2  | T3  | T5  | Average |
|---------------|------------|-----|-----|-----|-----|---------|
| Wait Time     | Best FCFS  | 32  | 0   | 85  | 8   | 31¼     |
|               | 1          | 84  | 22  | 85  | 57  | 62      |
|               | 5          | 82  | 20  | 85  | 58  | 61¼     |
|               | 8          | 80  | 8   | 85  | 56  | 57¼     |
|               | 10         | 82  | 10  | 85  | 68  | 61¼     |
|               | 20         | 72  | 20  | 85  | 88  | 66¼     |
|               | Worst FCFS | 68  | 145 | 0   | 121 | 83½     |
| Response Time | Best FCFS  | 85  | 8   | 153 | 32  | 69½     |
|               | 1          | 137 | 30  | 153 | 81  | 100½    |
|               | 5          | 135 | 28  | 153 | 82  | 99½     |
|               | 8          | 133 | 16  | 153 | 80  | 95½     |
|               | 10         | 135 | 18  | 153 | 92  | 99½     |
|               | 20         | 125 | 28  | 153 | 112 | 104½    |
|               | Worst FCFS | 121 | 153 | 68  | 145 | 121¾    |

# Shortest Task First (SJF) Scheduling

---

- Could we always mirror best FCFS?
- Shortest task first (SJF)
  - Run task that 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 task arrives and has shorter time to completion than remaining time on current task, immediately preempt current task
  - Sometimes called “shortest remaining time to completion first” (SRTCF)
- These can be applied to whole program or current CPU burst
  - Key idea: get short tasks out of system
  - Big effect on short tasks, only small effect on long ones
  - Better average response time

# SJF/SRTF Optimality

---

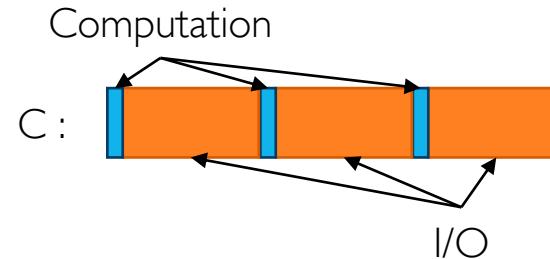
- SJF/SRTF minimize average response time! Why?
  - Consider alternative policy P (not SJF/SRTF) that is optimal
  - At some point, P chooses to run task that is not the shortest
  - Keep order of tasks the same, but run the shorter task first
  - This reduces average response time  $\Rightarrow$  contradiction!

# SJF/SRTF Discussion

---

- SJF/SRTF are **best you can** do to minimize average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, we can just focus on SRTF
- Comparison of SRTF with FCFS
  - What if all tasks are the same length?
    - SRTF  $\Rightarrow$  FCFS (*i.e., FCFS is best we can do if all tasks have the same length*)
  - What if tasks have varying length?
    - Unlike FCFS, with SRTF, short tasks do not get stuck behind long ones

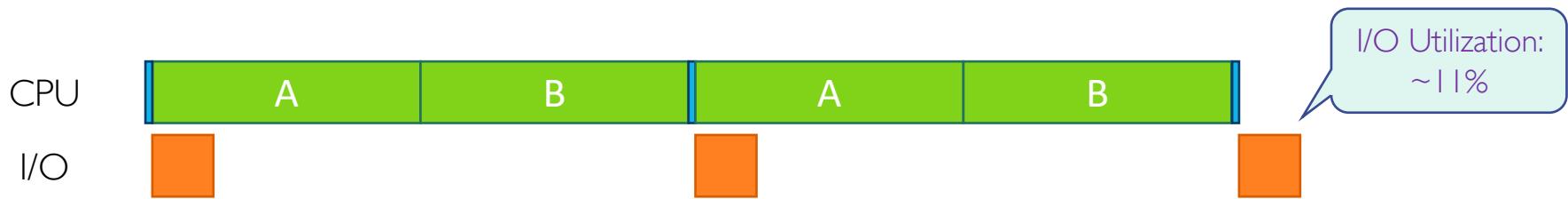
# Mix of CPU and I/O Bound Tasks: FCFS vs. RR vs. SRTF



- Example: suppose there are three tasks
  - A and B are both CPU bound with CPU bursts that last for a week
  - C is I/O bound with iterations of 1ms CPU burst followed by 9ms I/O burst
  - If A or B run by themselves, CPU utilization is 100% and I/O utilization is 0%
  - If C runs by itself, CPU utilization is 10% and I/O utilization is 90%
- What happens under FCFS scheduling policy?
  - Once A or B get in, keep CPU for two weeks  $\Rightarrow$  poor avg. response time
- What about RR or SRTF?
  - Easier to see with a timeline

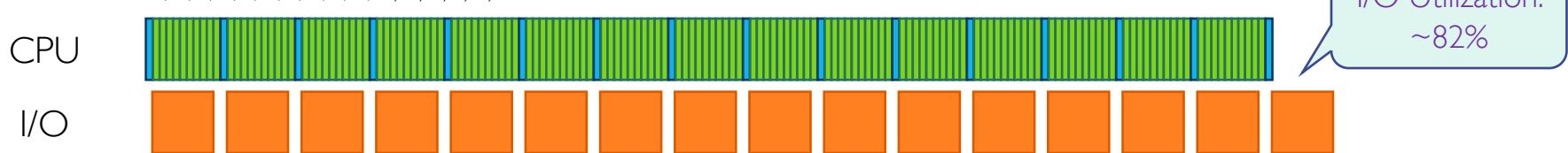
# Mix of CPU and I/O Bound Tasks: FCFS vs. RR vs. SRTF (cont.)

## RR with 40ms time quantum



## RR with 1ms time quantum

C,A,B,A,B,A,B,A,B,C,A,B,A,...



SRTF



# Problems with SRTF

---

- **Starvation**: large tasks may never run if short ones keep coming
- **Overhead**: short tasks preempt long ones  $\Rightarrow$  too many context switches
- **Unfair**: large tasks are penalized, there is high variance in response time

- **Impractical**: we need to somehow predict future

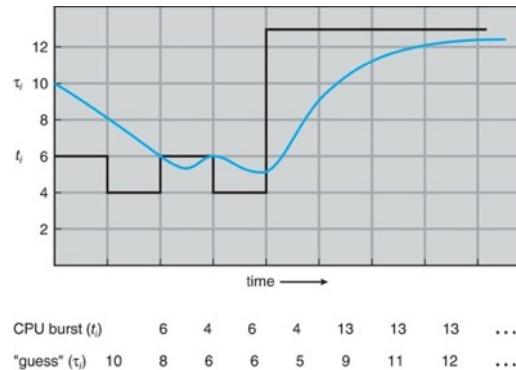
- Some systems ask users
  - When you submit your task, you have to say how long it will take
  - Users could maliciously misreport length of their task
  - E.g., would it work if a supermarket uses SJF?
    - Customers could game the system: come with one item at a time
    - To prevent cheating, systems may kill tasks if they take too long
- It's hard to predict task's runtime even for non-malicious users



# Predicting Length of Next CPU Burst

---

- **Adaptive**: dynamically make predictions based on past behavior
  - Works because programs have predictable behavior
    - If program was I/O bound in past, it'll likely be I/O bound in future
    - If behavior were random, this approach wouldn't help



- Example: use estimator function on previous bursts
  - Let  $t_{n-1}, t_{n-2}, t_{n-3}, \dots, t_1$  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 any time series estimator (e.g., Kalman filters, etc.)
  - For instance, exponential averaging  $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$  with  $(0 < \alpha \leq 1)$

# Aside: Application Types

---

- Can we use past burst times to identify application types?
- Consider mix of *interactive* and *high-throughput* programs
  - How to best schedule them?
  - How to recognize one from the other?
    - Do you trust applications to say that they are “interactive”?
  - Should you schedule the set of applications identically on servers, workstations, pads, and cellphones?

# Aside: Application Types (cont.)

---

- Assumptions encoded into many schedulers (e.g., *O(1) Scheduler*)
  - Applications that sleep a lot and have short bursts must be interactive
    - Give them high priority
  - Applications that compute a lot must be high-throughput apps
    - Give them lower priority, since they won't notice intermittent bursts from interactive apps
- In general, it is hard to characterize applications
  - What about apps that sleep for a long time, and compute for a long time?
  - What about applications that must run under all circumstances

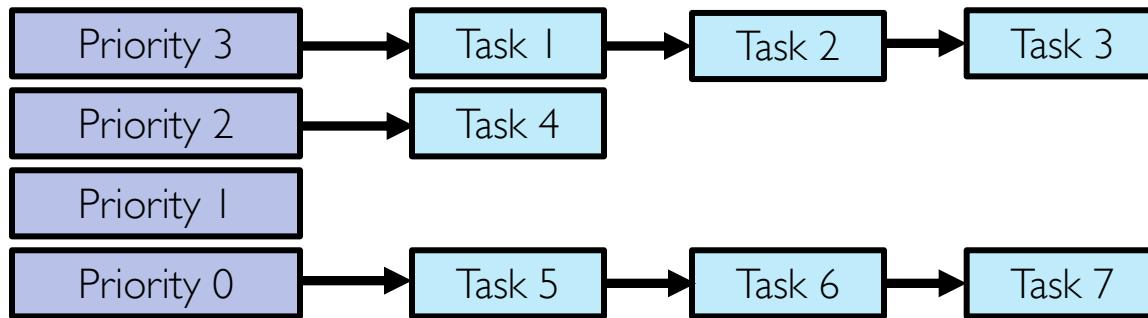
# SRTF Final Notes

---

- Bottom line, we can't really know how long tasks will take
  - However, we can use SRTF as yardstick for measuring other policies
  - Optimal, so we can't do any better
- Pros & cons of SRTF
  - + Optimal (average response time)
  - – Hard to predict future
  - – Too many context switches
  - – Unfair

# Strict Priority Scheduling

---



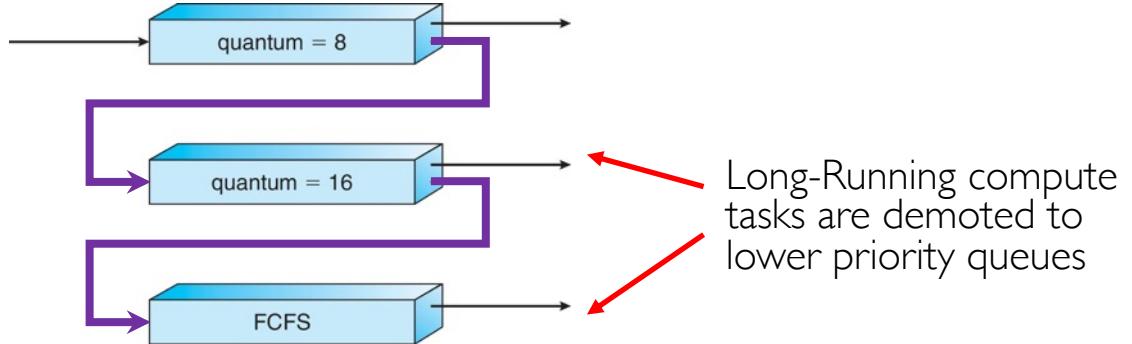
- Execution plan
  - Always execute highest-priority runnable tasks to completion
  - Each queue can be threaded in RR with some time-quantum
- Notice any problems?
  - **Starvation**
    - Lower-priority tasks may never run because of higher-priority tasks
  - **Priority inversion**
    - Low-priority task delays high-priority task by holding resources needed by high-priority task (more on this later)

# Fairness

---

- Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc.)
  - long running tasks may never get any CPU time
  - In Multics, shut down machine, found 10-year-old task
- One approach: give each queue some fraction of CPU
  - What if there are 100 short tasks and only one long task?
    - Like express lanes in a supermarket, sometimes express lanes get so long, get better service by going into one of other lines
- Another approach: increase priority of tasks 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 task gets CPU time, so everyone increases in priority  $\Rightarrow$  Interactive tasks suffer
- Tradeoff: fairness is usually gained by hurting average response time!

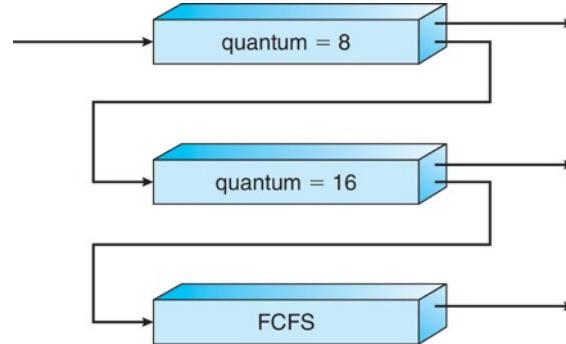
# Multi-level Feedback Queue



- 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 task’s priority as follows (details vary)
  - Task starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)

# Multi-level Feedback Queue (cont.)

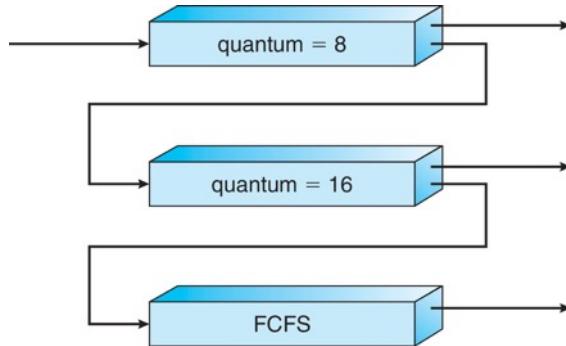
---



- Result approximates SRTF
  - CPU bound tasks drop like a rock
  - Short-running I/O bound tasks stay near top
- Scheduling must be done between queues
  - **Fixed priority scheduling**
    - Serve all from highest priority, then next priority, etc.
  - **Time slicing**
    - Each queue gets fraction of CPU time
    - E.g., 70% to highest, 20% next, 10% lowest

# Multi-level Feedback Queue (cont.)

---



- **Countermeasure:** user action that foil intent of OS designers
  - For multilevel feedback, put simple I/O's to keep task's priority high
  - Example of MIT Othello Contest
    - Cheater put `printf`'s, ran much faster than competitors!
    - Of course, if everyone did this, wouldn't work!

# Lottery Scheduling

---

- Give each task  $i$  some number of lottery tickets  $N_i$
- On each time slice, randomly pick a winning ticket
- Lottery scheduling achieves **proportional-share allocations**
  - On average, CPU time is proportional to # of tickets given to task
- How to assign tickets?
  - Give tasks tickets proportional to their priorities
  - To approximate SRTF, give short tasks more and long tasks fewer
  - To avoid starvation, give every task at least one ticket (everyone makes progress)
- Compared to strict priority, lottery scheduling behaves gracefully as load changes
  - Adding or deleting one task affects all tasks proportionally, independent of how many tickets each task possesses



# Lottery Scheduling Example

---

- Assume short tasks get 10 tickets, long tasks get 1 ticket

| # short tasks/<br># long tasks | % of CPU each<br>short tasks gets | % of CPU each<br>long tasks 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 tasks to give reasonable response time?
  - If load average is 100, hard to make progress
  - One approach is to log some users out

# Unfairness of Lottery Scheduling

---

- Define unfairness for two tasks with the same burst time as
  - Unfairness = finish time of first one / finish time of last one
- As function of burst time

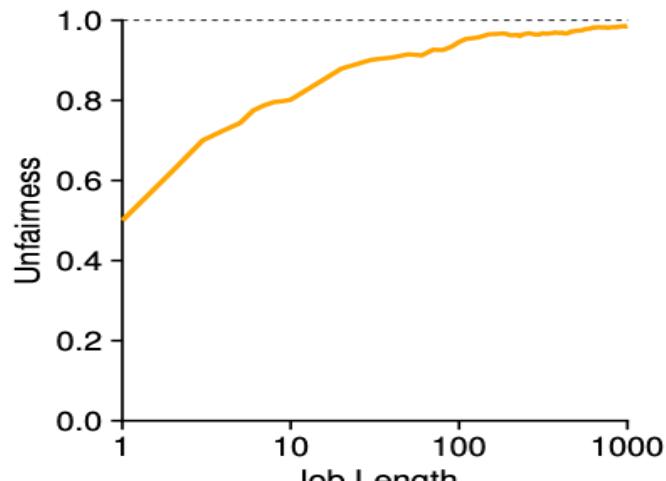


Figure 9.2: Lottery Fairness Study

# Stride Scheduling

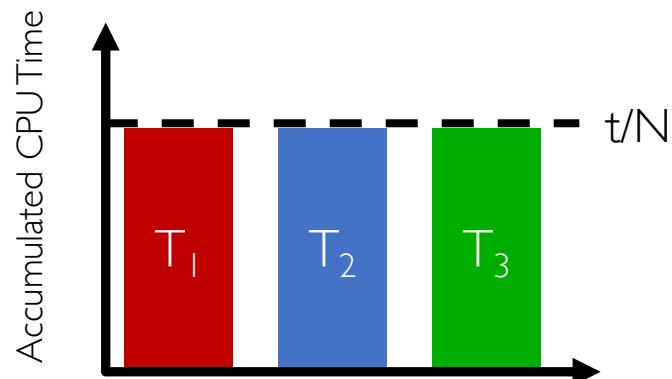
---

- Achieves proportional-share scheduling without resorting to randomness
- Defines *stride* for each thread to be  $\text{Big\#}W/N_i$ 
  - The larger your share of tickets, the smaller your stride
  - E.g., with  $W = 10,000$ , and A, B, and C each having 100, 50, and 250 tickets, strides for A, B, and C are 100, 200, and 40, respectively
- Maintains *pass* counter for each thread
- Runs thread with lowest pass and adds its *stride* to its *pass*
  - Low-stride threads (lots of tickets) run more often
  - Thread with twice the tickets gets to run twice as often
  - Some messiness of counter wrap-around, new threads, ...

# Max-min Fair (MMF) Scheduling

---

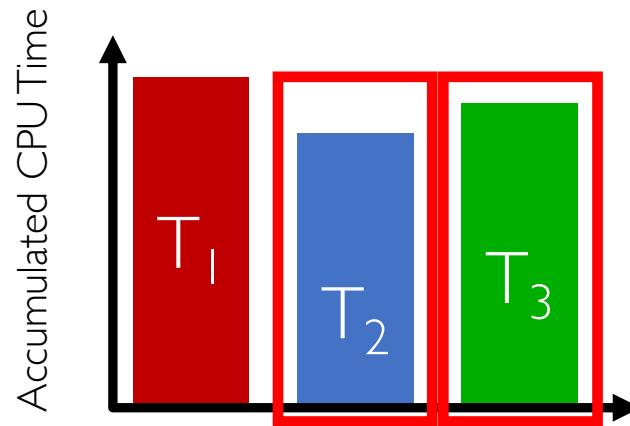
- Always choose thread with lowest accumulated CPU time
  - If chosen thread doesn't have CPU burst, schedule second lowest ...
  - Break ties randomly if multiple threads equally have lowest CPU time
- Goal is to give each thread equal share of CPU time
  - With  $N$  runnable threads, each thread should get  $1/N^{\text{th}}$  of CPU time
- At any time  $t$  we want to have



# MMF Scheduling (cont.)

---

- Strict MMF causes too many context switches
  - It effectively turns to running one instruction of each thread
- Relaxed MMF runs thread with lowest accumulated CPU time for fixed time quantum before choosing next thread



- Notice any problem?
  - Fixed quantum leads to poor response time as # of threads increases

# MMF Scheduling (cont.)

---

- Solution: dynamically change time quantum
- **Target latency**: interval during which all threads should run at least once
- Time quantum = Target latency / N
  - E.g., with 20ms target latency and 4 threads, time quantum is 5ms
- Notice any problem?
  - With 20ms target latency and 200 threads, time quantum becomes 0.1ms
  - Recall RR: large context switching overhead if time quantum gets too small
- **Minimum granularity**: minimum length of any time quantum
  - E.g., with target latency 20ms, 1ms minimum granularity, and 200 processes, time quantum is 1ms

# Weighted Max-min Fair Scheduling

---

- What if we want to give more to some and less to others (proportional share)?
- **Key Idea:** assign weight  $w_i$  to each thread  $i$
- MMF uses single time quantum for all threads

$$Q = \frac{\text{Target latency}}{N}$$

- Weighted MMF uses different time quanta for different threads

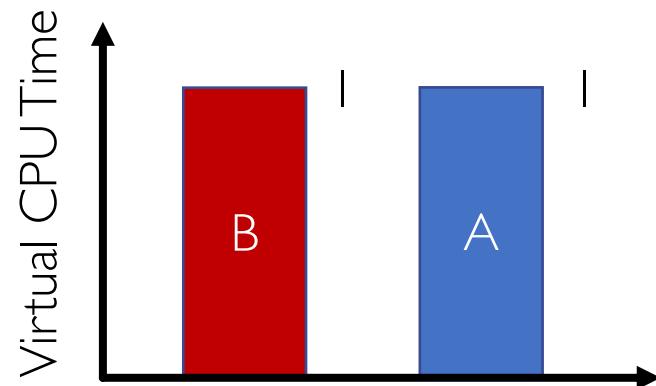
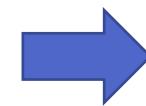
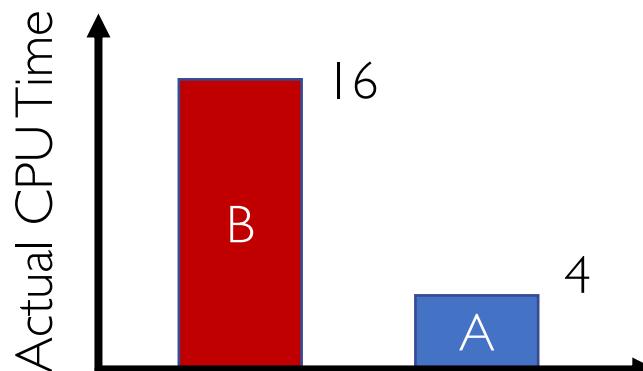
$$Q_i = \frac{w_i \times \text{Target latency}}{\sum_{j=1}^N w_j}$$

- E.g., with 20ms target latency, 1ms minimum granularity, and 2 threads: A with weight 1 and B with weight 4
  - Time quantum for A is 4 ms
  - Time quantum for B is 16 ms

# Weighted MMF Scheduling (cont.)

---

- Also track threads' *virtual runtime* rather than their true wall-clock runtime
- Higher weight: virtual runtime increases more slowly
- Lower weight: virtual runtime increases more quickly
- Linux *completely fair scheduler* deploys very similar ideas
  - Ready queue is implemented as red-black tree, in which threads are sorted in increasing order of their virtual runtime



# How to Evaluate Scheduling Algorithms?

---

- Deterministic modeling
  - Pick workload and compute performance of each algorithm
- Queueing models
  - Mathematical approach for handling stochastic workloads  
(more on this later)
- Simulations/Implementations
  - Build system which allows actual algorithms to be run against actual data – most flexible/general

# Starvation and Sample Bias

---

- Suppose you want to compare scheduling policies
  - Create some infinite sequence of arriving tasks
  - Start measuring
  - Stop at some point
  - Compute ART for finished tasks between start and stop
- Is this valid or invalid?
  - SJF and FCFS would complete different sets of tasks
    - Their ARTs are not directly comparable
    - E.g., suppose you stopped at any point in FCFS vs. SJF slide

# Solutions for Sample Bias

---

- For both systems, measure for long enough that  
 $\# \text{ of completed tasks} \gg \# \text{ of uncompleted tasks}$
- Start and stop system in idle periods
  - Idle period: no work to do
  - If algorithms are work-conserving, both will complete the same set of tasks

# Choosing Right Scheduling Algorithm

---

| If you 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 | RR                            |
| Favoring important threads      | Priority                      |
| Proportional sharing            | Lottery and stride scheduling |

# Summary

---

- First-come, first-served (FCFS)
  - Threads are served in the order of their arrival
- Round robin (RR)
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
- Shortest task first (SJF) / shortest remaining time first (SRTF)
  - Run whatever thread that has the least amount of computation to do/least remaining amount of computation to do
- Multi-level feedback queue (MFQ)
  - Multiple queues of different priorities and scheduling algorithms
- Lottery and stride scheduling
  - Give each thread a priority-dependent number of tickets
- Max-min fair (MMF)
  - Give each thread equal share of CPU time

# Questions?

---



# Acknowledgment

---

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, Canny