Duke Systems

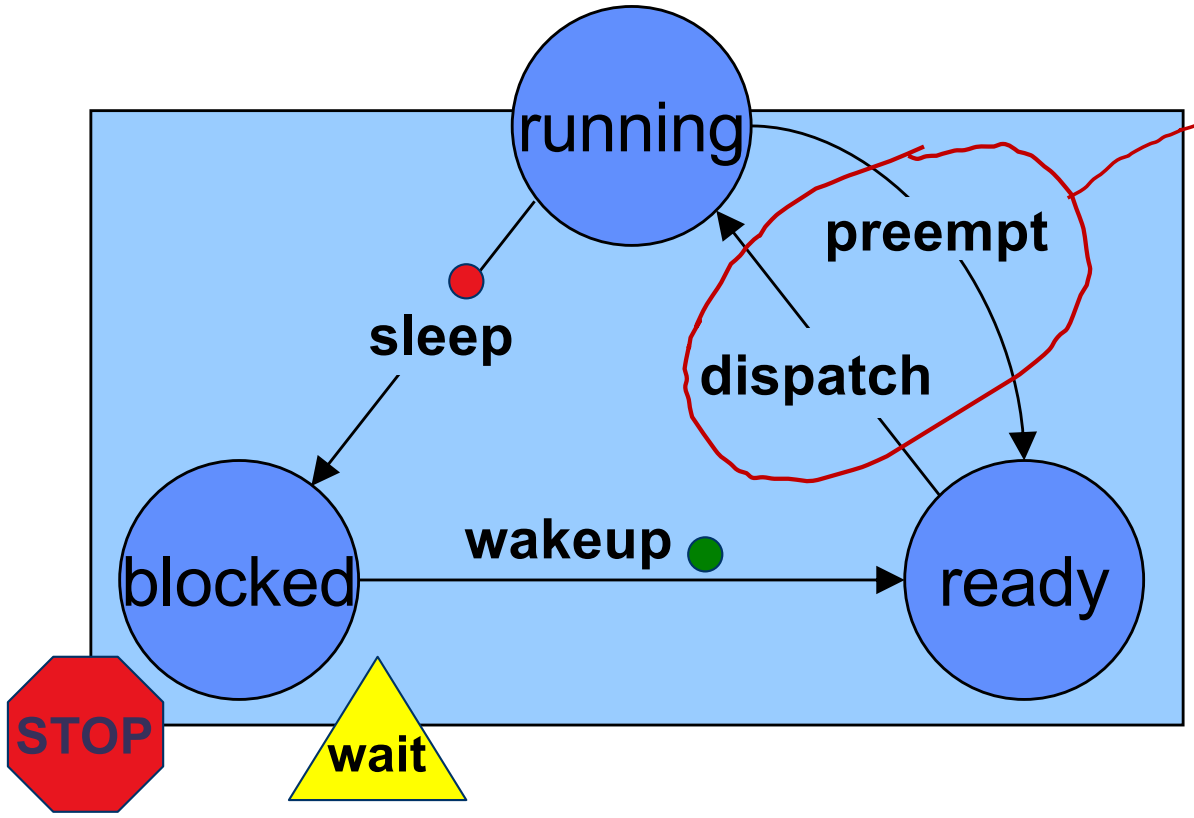
CPU Scheduling

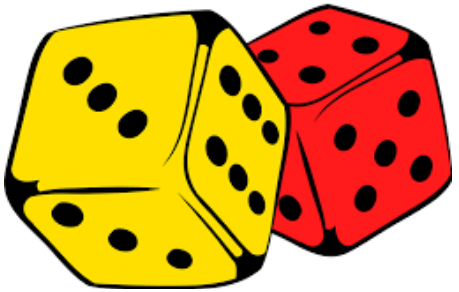
Jeff Chase

Duke University

CPU scheduling: the role of policy

- Thread in **ready** state may **dispatch** “at any time”.
- Thread in **running** state may **preempt** “at any time”.
- CPU scheduling **policy** governs these transitions.





“nondeterministic”

The Linux Scheduler: a Decade of Wasted Cores

Jean-Pierre Lozi
Université Nice Sophia-Antipolis
jplozi@unice.fr

Baptiste Lepers
EPFL
baptiste.lepers@epfl.ch

Justin Funston
University of British Columbia
jfunston@ece.ubc.ca

Fabien Gaud
Coho Data
me@fabiengaud.net

Vivien Quéma
Grenoble INP / ENSIMAG
vivien.quema@imag.fr

Alexandra Fedorova
University of British Columbia
sasha@ece.ubc.ca

Abstract

As a central part of resource management, the OS thread scheduler must maintain the following, simple, invariant: make sure that ready threads are scheduled on available cores. As simple as it may seem, we found that this invariant is often broken in Linux. Cores may stay idle for seconds while ready threads are waiting in runqueues. In our experiments, these performance bugs caused many-fold performance degradation for synchronization-heavy scientific

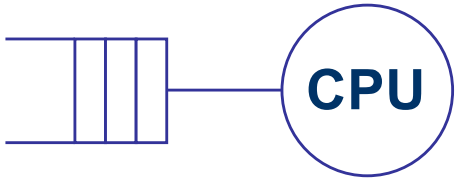
Classical scheduling problems revolve around setting the length of the scheduling quantum to provide interactive responsiveness while minimizing the context switch overhead, simultaneously catering to batch and interactive workloads in a single system, and efficiently managing scheduler run queues. By and large, by the year 2000, operating systems designers considered scheduling to be a solved problem; the Linus Torvalds quote is an accurate reflection of the general opinion at that time.

EuroSys 2016

Workload: tasks and jobs

- We often use the words **task** or **job** with scheduling.
- The concepts are more general than thread or process.
- Let's suppose for simplicity that a task t_i is a thread with a CPU **service demand** of D_i time units.
  - Arrive at random.
  - Queue on runqueue.
  - Dispatch when selected.
  - Exit (or sleep) after time D_i.

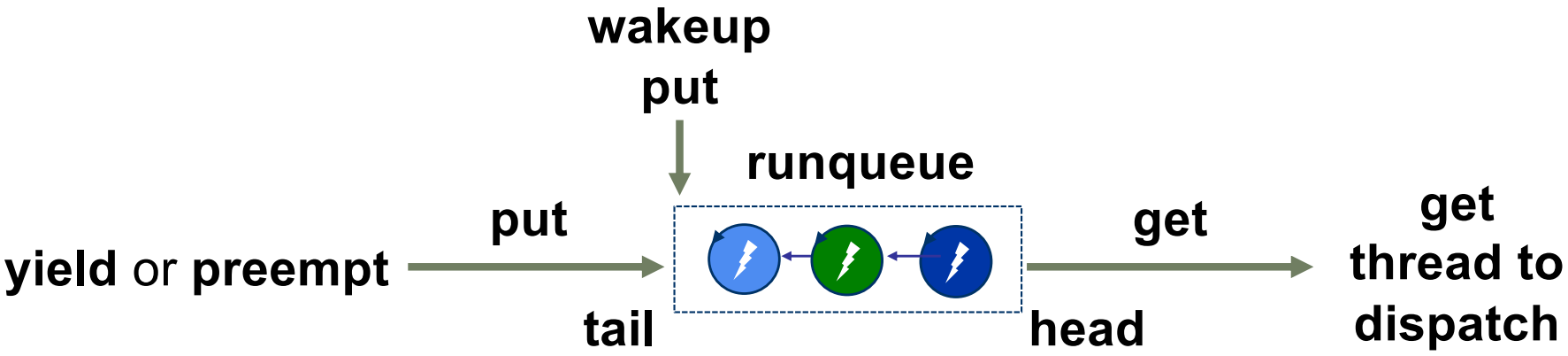
ready queue (runqueue)



A simple policy: FCFS/FIFO

The most basic scheduling policy is **first-come-first-served (FCFS)**, also called **first-in-first-out (FIFO)**.

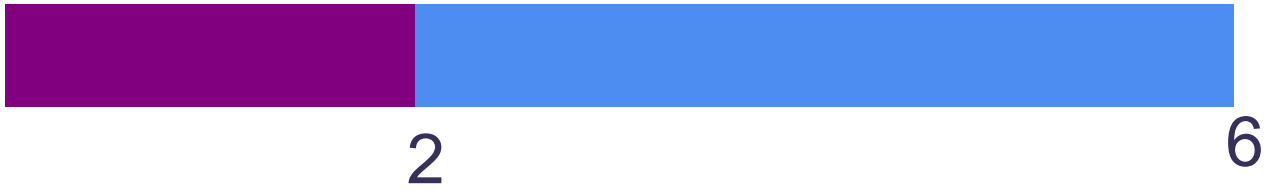
- FCFS is just like the checkout line at the QuickiMart.
- Maintain a queue ordered by time of arrival.
- **GetNextToRun** selects from the front (head) of the queue.



Metric: response time

- **How long to get the job done?**
- Suppose two jobs arrive “together”.
- **Schedule 1**: blue (D=4) goes first.

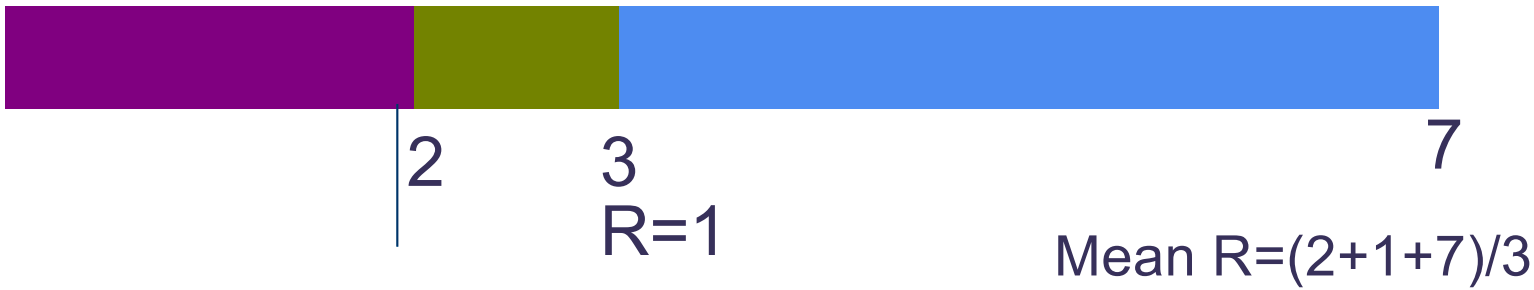
- **Schedule 2**: purple (D=2) goes first.



- S1 slows down purple by 3x. Mean = 5.
- S2 slows down blue by only 50%. Mean = 4. **Better**.

Is it fair?

- **Run the small job first!**
- But what if another small one arrives?
- What if a D=1 arrives at time t=2?
- Run it? Defer blue again?
- Slows blue down by only 15%!
- Otherwise slow new arrival by 5x!
- Then what if **another** short task arrives…



Metric: fairness

- **Scheduler allocates a resource: CPU. Is it fair?**
- What does **fairness** mean? What makes it "fair"?
  - Early bird gets the worm; everyone is fed eventually?
  - Low variance? (E.g., equal slowdown, Jain fairness index)
  - "Divide the pie" evenly? (Or according to weights?)
  - Freedom from starvation? (E.g., upper bound on wait time)
  - Serve the clients who pay the most? (Market-based)
  - Serve the clients who benefit most? (Maximize global welfare)
  - Freedom from envy?
- This is a deep topic. But we gloss over it.

Minimizing Response Time: SJF (STCF)

Shortest Job First (SJF) is provably optimal if the goal is to minimize average-case ***R***.

Also called **Shortest Time to Completion First (STCF)** or **Shortest Remaining Processing Time (SRPT)**.

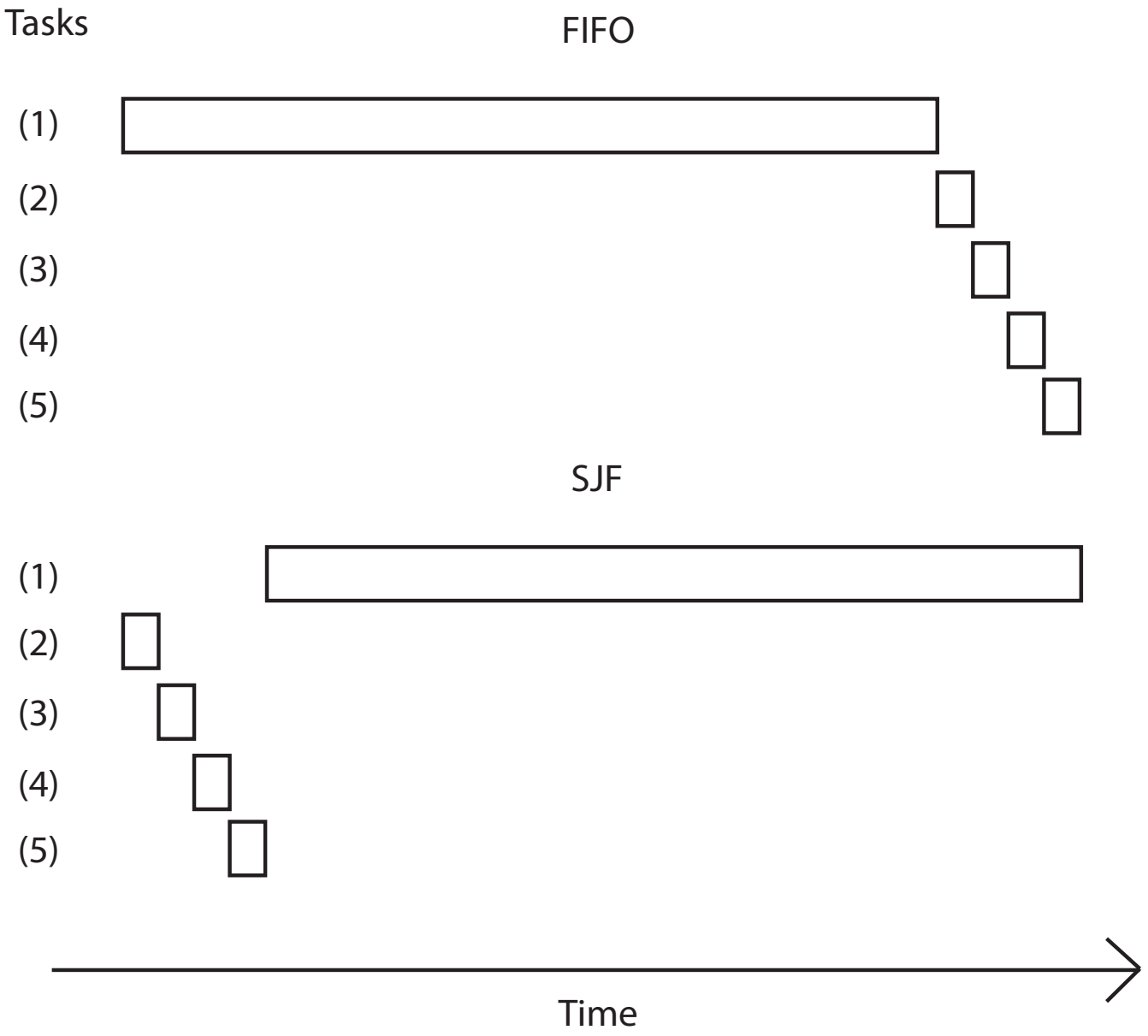
Example: express lanes at the MegaMart

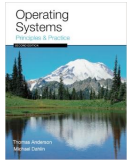
Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.

Intuition: longest jobs do the least possible damage to the wait times of their competitors.

$$R = (1 + 3 + 6)/3 = 3.33$$

FIFO vs. SJF





Preemptive FIFO: Round Robin

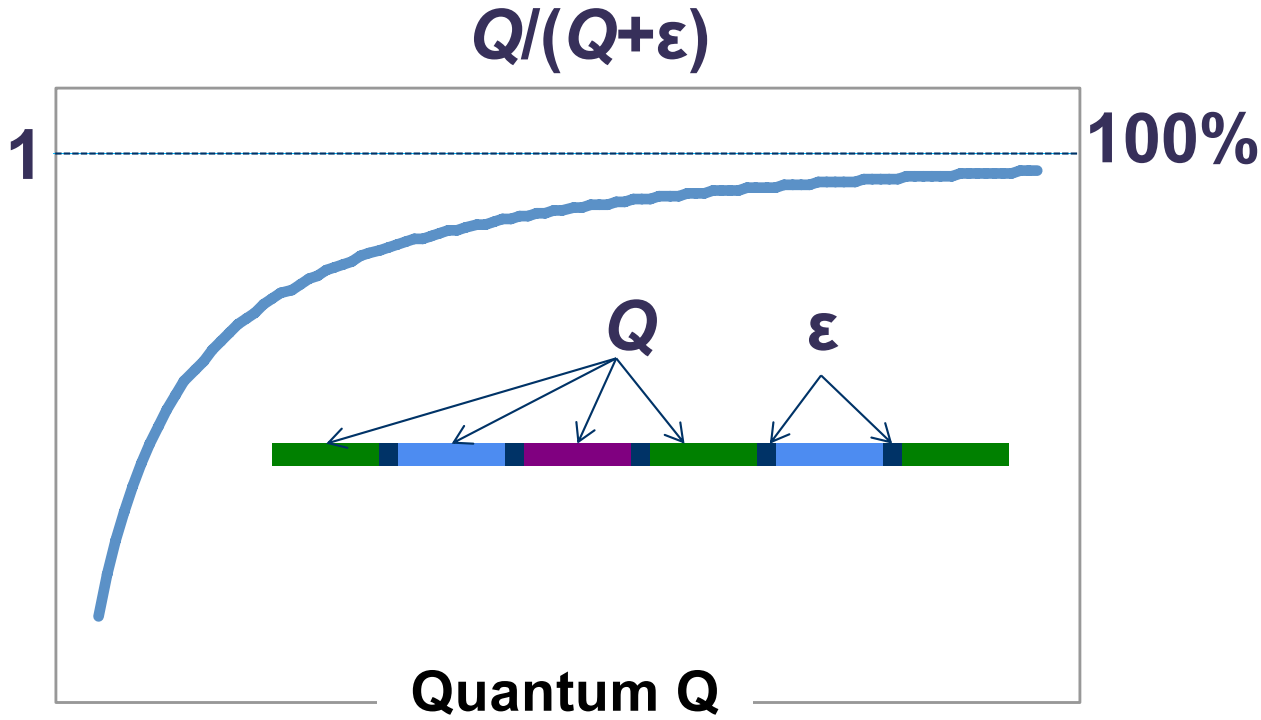
- **FIFO with preemption each quantum Q of time.**
- **Timeslicing** shares evenly among contending jobs.
- Minimize variance→reduce impact of large jobs.
- Delay is proportional to one's demand.
- Overhead to context switch every Q.
- Q=1 yields mean R = 4.

Overhead and goodput

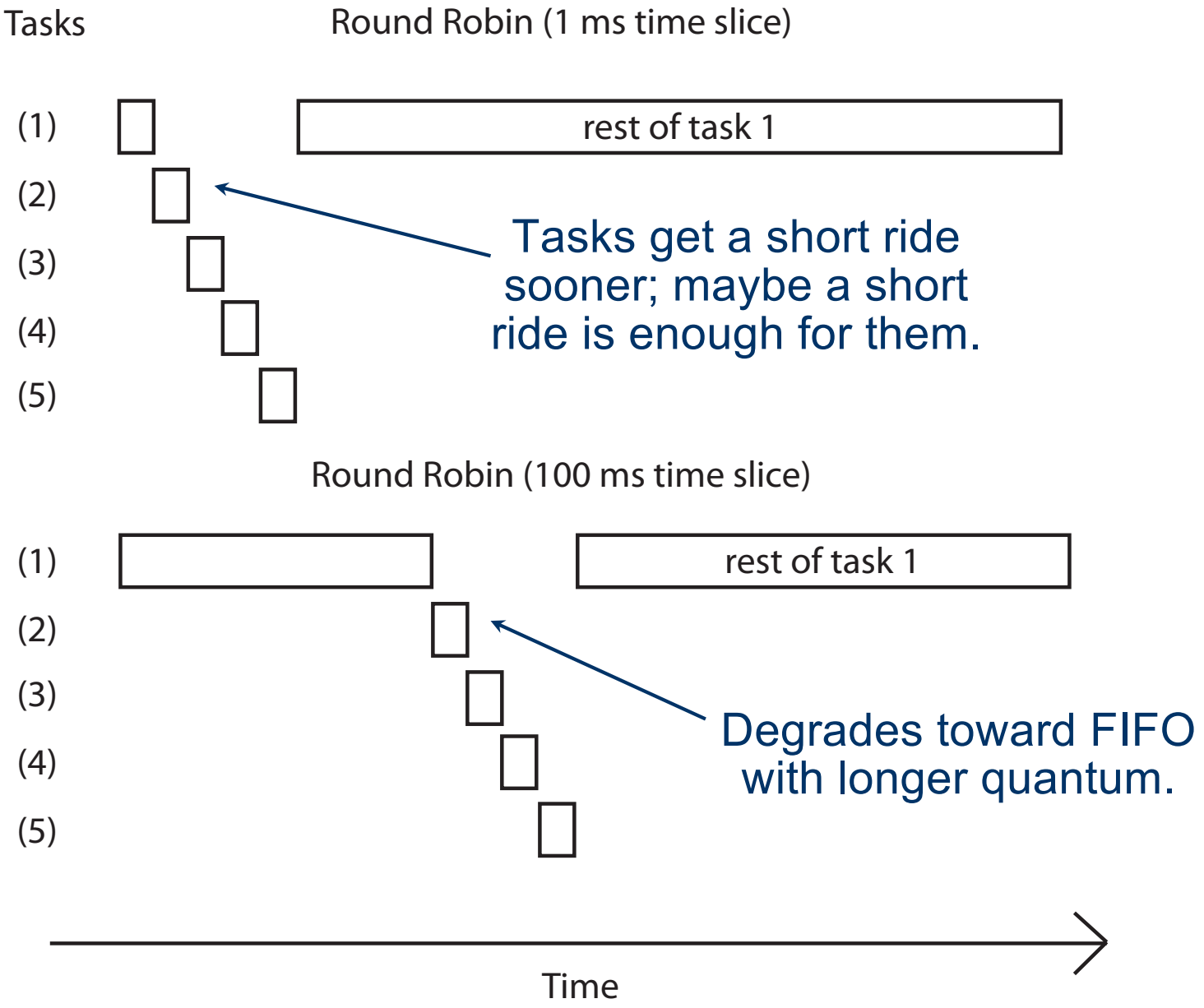
Context switching is overhead: "wasted effort". It is a cost that the system imposes in order to get the work done. It is not actually doing the work.

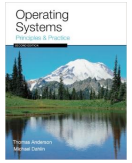
This graph is obvious. It applies to so many things in computer systems and in life.

Efficiency
or goodput
What percentage of the time is the busy resource doing useful work?

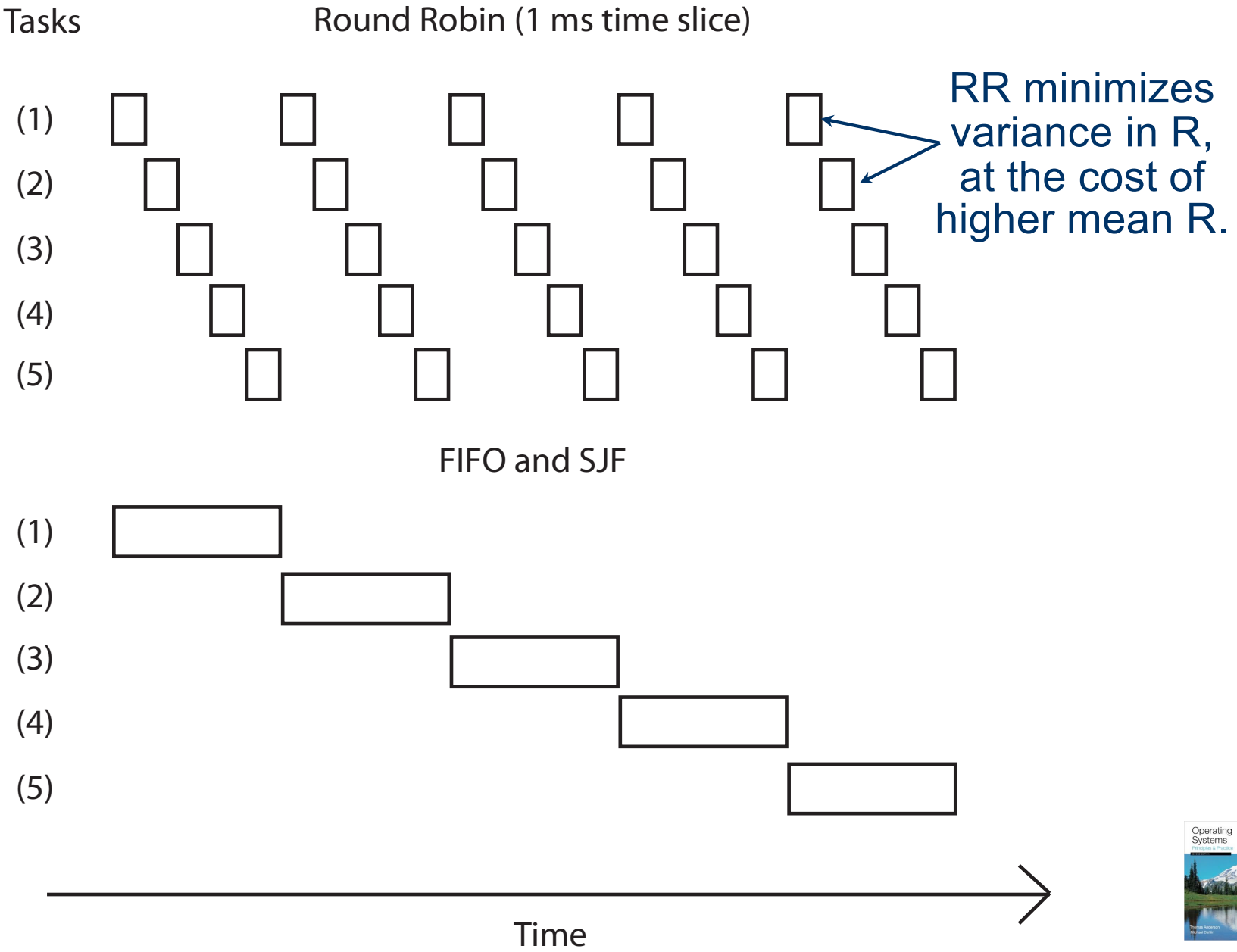


Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
(2)
(3)
(4)
(5)
rest of task 1
Tasks get a short ride sooner; maybe a short ride is enough for them.
Round Robin (100 ms time slice)
(1)
(2)
(3)
(4)
(5)
rest of task 1
Degrades toward FIFO with longer quantum.
Time

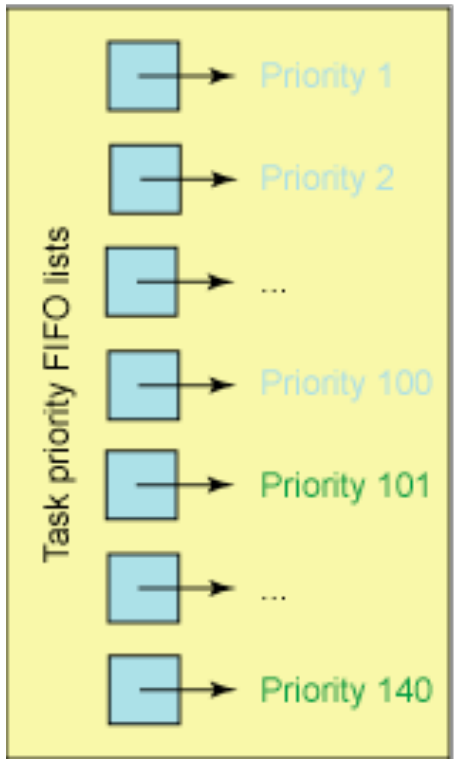

Operating Systems

Round Robin vs. FIFO



Task priority

In a typical OS, each task has a **priority** value (e.g., an integer) which may change over time.



- Pick the task with the highest priority.
- Or round-robin if there's a tie.
- Preempt if a higher-priority task appears.
- Threads inherit a base priority.
- User-settable relative priority.
- E.g., Unix **nice** command.
- Internal priority adjustments in scheduler.

How many priority levels?
32 (Windows) to 128 (MacOS)

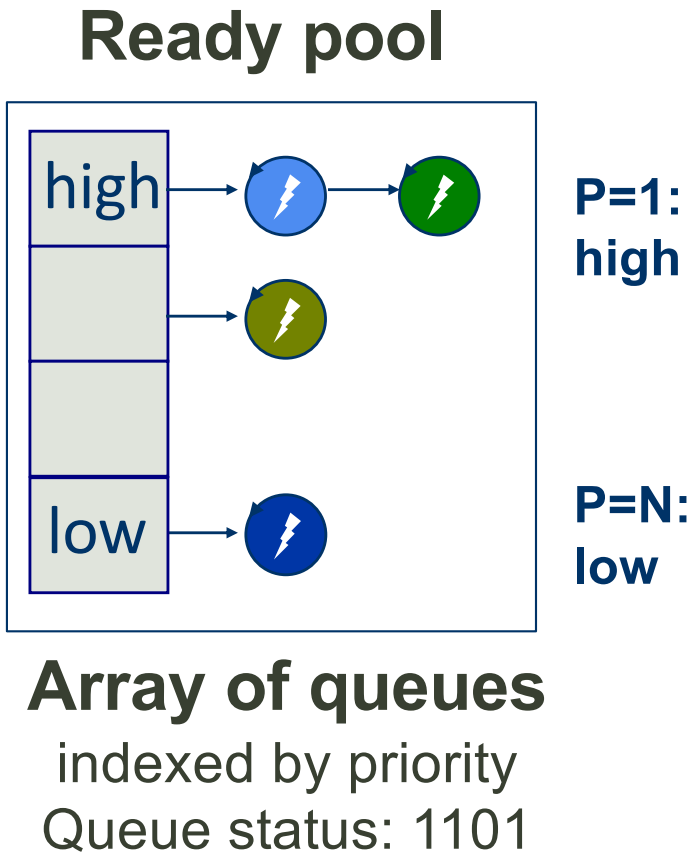
Multi-level priority queue

Multi-level priority queue is a common structure.

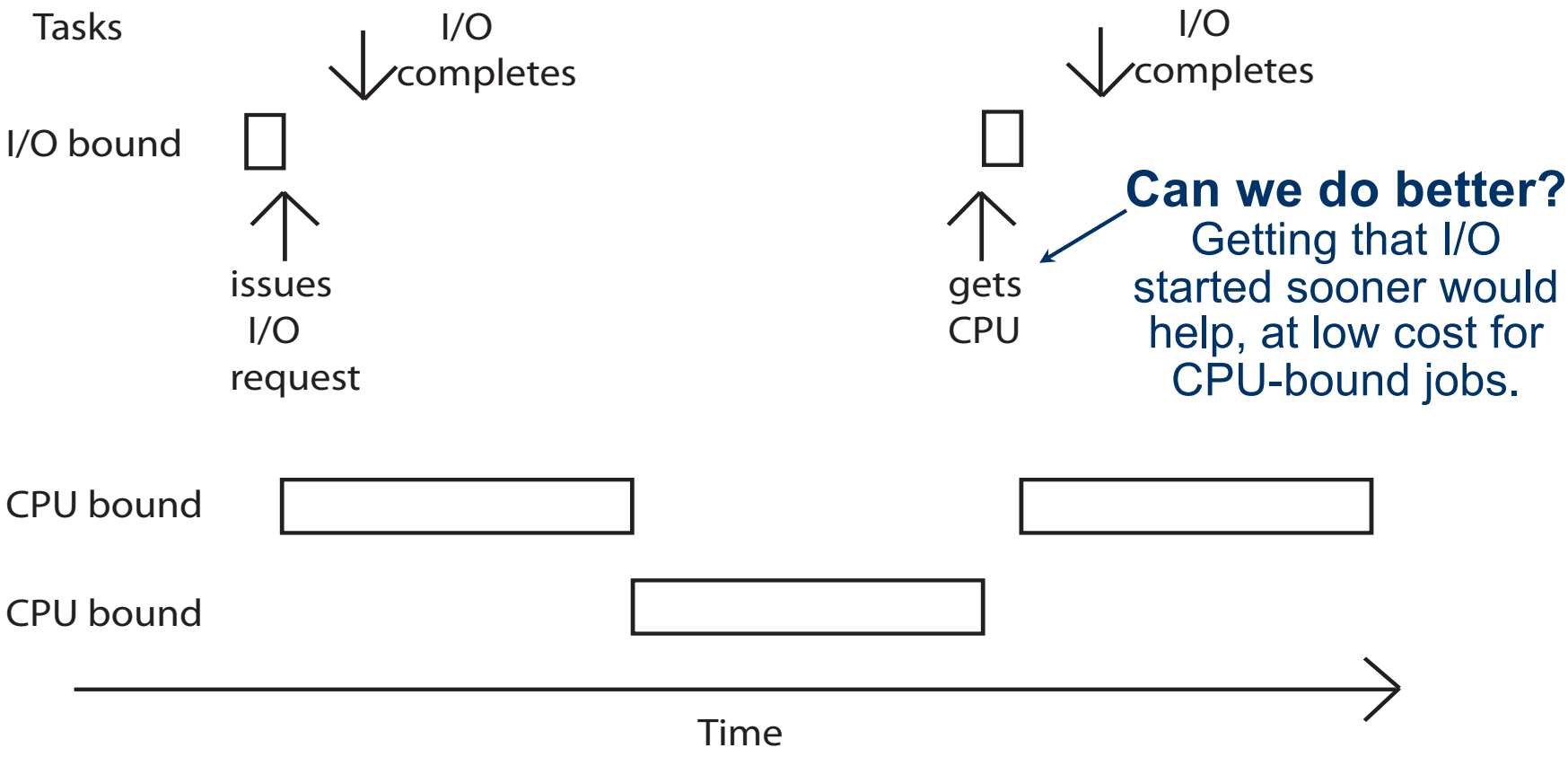
1. Bit vector of non-empty queues.
2. Find highest non-empty queue.
3. Take task from its head.
4. If now empty, clear its bit.
5. Dispatch task.

Constant time, no sorting.

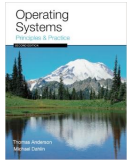
CPUs have an instruction to find the highest (or lowest) bit set in a word: find-first-set (**ffs**), count-leading-zero (**clz**). Gives index of highest-priority non-empty queue.



Challenge: Mixed workload



The issue also applies to threads that interact with a user (UI threads).
Goal: improve responsiveness to user.

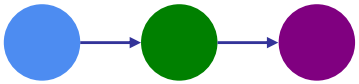


Challenge: CPU scheduling and I/O

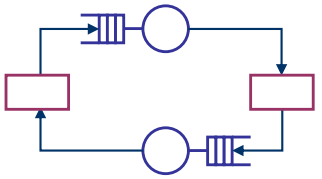
- Suppose thread T does a lot of I/O.
- Suppose T blocks while its I/O is in progress.
- When each I/O completes, T gets back on the readyQ.
- Where T waits for threads that use a lot of CPU time.
  - While the disk or other I/O device sits idle!
- T needs “just a smidgen” on CPU to start its next I/O.
- Maybe let T jump the queue, and get it started? So that both the disk and CPU can work in parallel?
- “Like SJF”

Two Schedules for CPU/Disk

1. Naive **Round Robin**



CPU busy 25/37: U = 67%
Disk busy 15/37: U = 40%



2. Add internal priority boost for I/O completion

CPU busy 25/25: U = 100%
Disk busy 15/25: U = 60%

33% improvement in utilization

Keep all units busy doing useful work when possible—improves all metrics.

Estimating time-on-core

- How to predict which tasks have short CPU demand for their next ride on the core?
  - Like guessing the weather: predict from recent past.
  - Judge jobs by their behavior.
  - **Heuristic**: could be wrong! But mostly works over time.
- Example heuristic**: adaptive internal priority**
  - Quantum expires? → task is **CPU-bound**.
  - Drop its priority **down** one level.
  - Task blocks? → task is **I/O bound** (short rides)
  - So bump its priority **up** one level.
  - CPU-bound tasks sink to the bottom, I/O bound rise up.

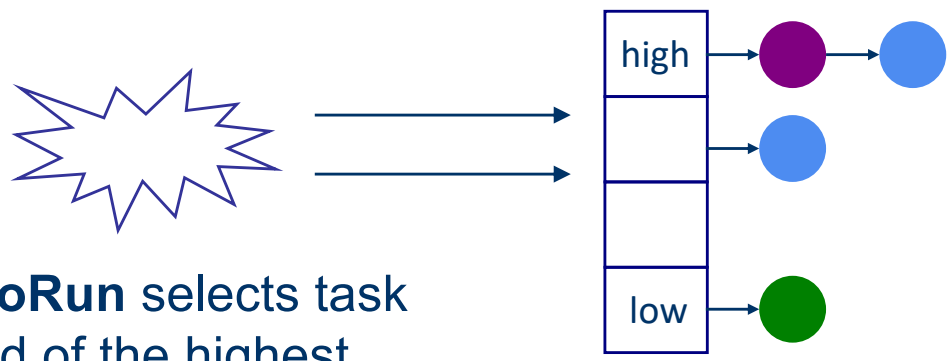
Multilevel Feedback Queue

Many systems (e.g., Unix variants) implement internal priority using a **multilevel feedback queue**.

- **Multilevel priority queue**. A queue for each of *N* priority levels.

  Use RR on each queue; look at queue *i+1* only if queue *i* is empty.

- **Feedback**. Adaptive internal priority.



I/O bound jobs

jobs holding resources
jobs with high external priority

CPU-bound jobs

GetNextToRun selects task at the head of the highest non-empty queue: constant time, no sorting.

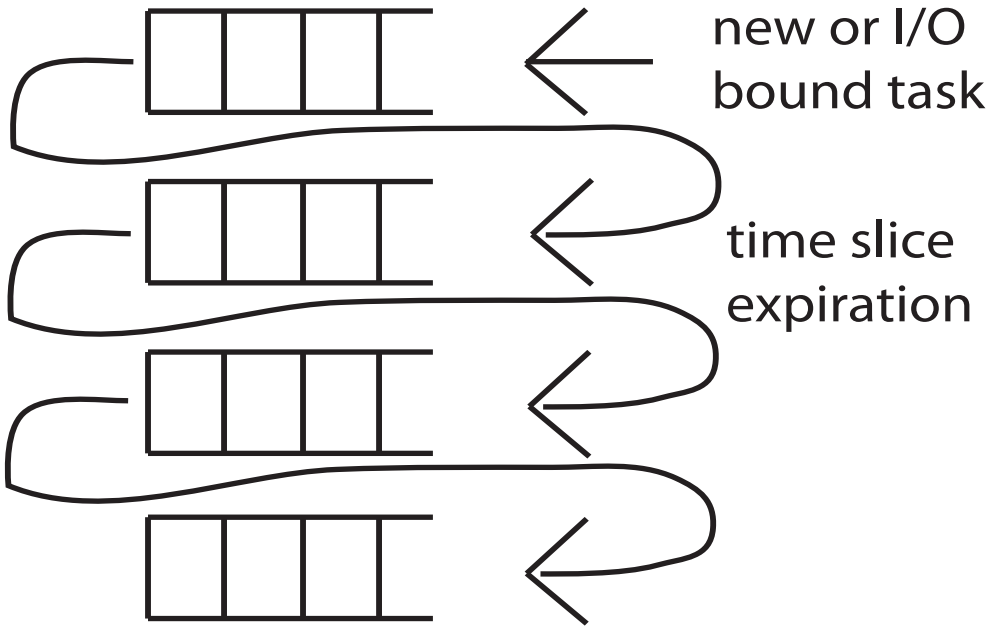
ready queues
indexed by priority

Priority of CPU-bound jobs decays with system load and service received.

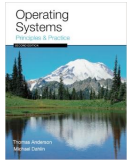
MFQ

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



Note: OSTEP has a thorough treatment and calls it MLFQ.



Some deep lessons

- Resource allocation! It's everywhere.
- "Separate policy and mechanism."
- Policy matters. We can measure it!
- But there are multiple competing metrics.
- The results depend on workload.
- Use heuristics: "simple" (and maybe wrong).
- Interplay of heuristics and workload: "it's complicated".

More slides on the same stuff

EXTRA

Example: a recent Linux rev

"Tasks are determined to be I/O-bound or CPU-bound based on an interactivity heuristic. A task's **interactiveness metric** is calculated based on how much time the task executes compared to how much time it sleeps. Note that because I/O tasks schedule I/O and then wait, an I/O-bound task spends more time sleeping and waiting for I/O completion. This increases its interactive metric."

Key point: interactive tasks get higher priority for the CPU, when they want the CPU (which is not much).

Evaluating Round Robin

$R = (5+6)/2 = 5.5$

$R = (2+6+\varepsilon)/2 = 4 + \varepsilon$

- **Response time**. RR reduces it for short jobs.
  - For a given load, wait time is proportional to the job's total service demand *D*.
- **Fairness**. RR reduces variance in wait times.
  - *But*: RR forces jobs to wait for other jobs that arrived later.
- **Throughput**. RR imposes extra context switch **overhead**.
  - Degrades to FCFS-RTC with large *Q*.

Evaluating FCFS

How well does FCFS achieve the goals?

- **Throughput**. FCFS is as good as any non-preemptive policy.

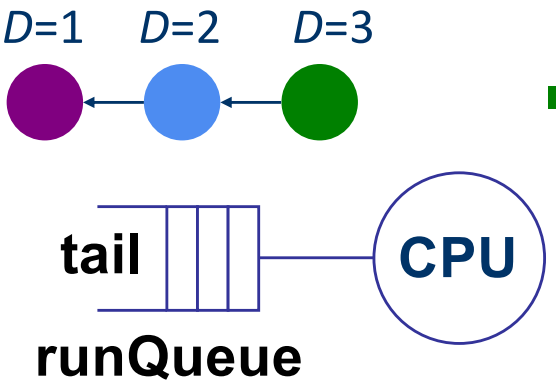
  ….if the CPU is the only schedulable resource in the system.

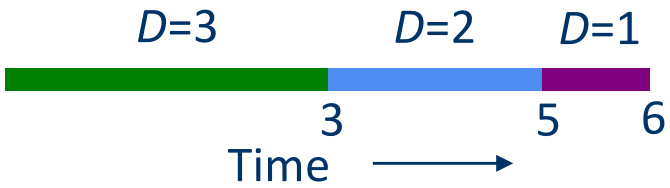
- **Fairness**. FCFS is intuitively fair…sort of.

  "The early bird gets the worm"…and everyone is fed…eventually.

- **Response time**. **Long jobs keep everyone else waiting.**

  Consider **service demand** (D) for a task (i.e., its runtime).





$R = (3 + 5 + 6)/3 = 4.67$

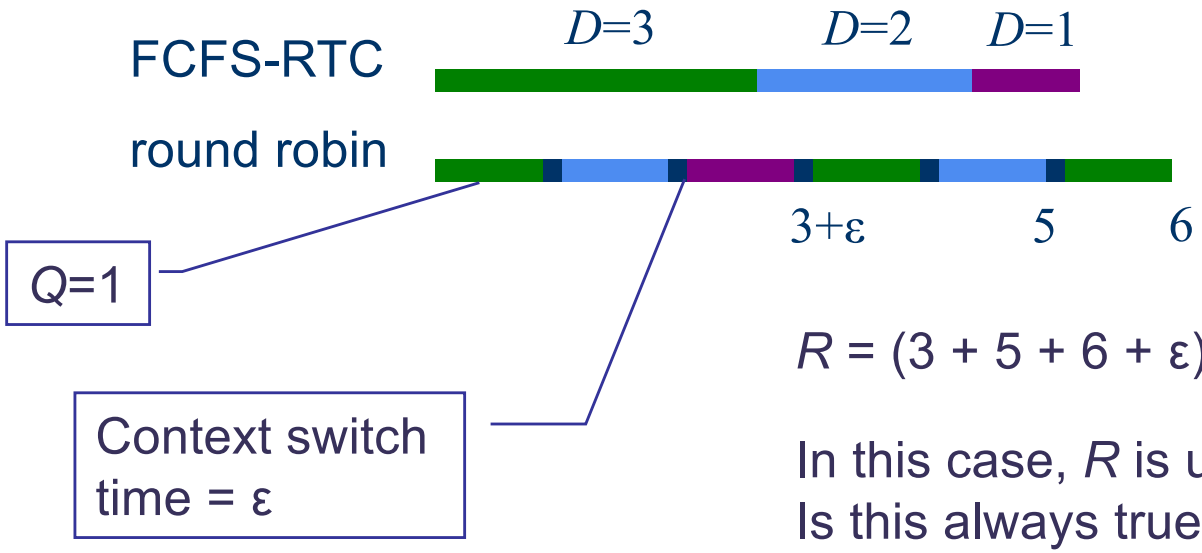
Preemptive FCFS: Round Robin

Preemptive timeslicing is one way to improve fairness of FCFS.

If job does not block or exit, force an involuntary context switch after each **quantum** *Q* of CPU time.

FCFS without preemptive timeslicing is “run to completion” (RTC).

FCFS with preemptive timeslicing is called **round robin**.



$R = (3 + 5 + 6 + \varepsilon)/3 = 4.67 + \varepsilon$

In this case, *R* is unchanged by timeslicing.
Is this always true?