

CS5460: Operating Systems

Lecture 6:

Advanced Scheduling (Ch 8 and 9)

Slide Credit: Andrea Arpaci-Dusseau

Assignment 2

- Due Tue Feb 16

Scheduling Basics

Workloads:

arrival_time

run_time

Schedulers:

FIFO

SJF

STCF

RR

Metrics:

turnaround_time

response_time

Scheduling Policy Review

Workload

Job	arrival	run
A	0	40
B	0	20
C	5	10

Timelines

Schedulers:

FIFO

SJF

STCF

RR

0 20 40 60 80 0 20 40 60 80

RR

SJF

0 20 40 60 80 0 20 40 60 80

STCF

FIFO

Scheduling Policy Review

Workload

Job	arrival	run
A	0	40
B	0	20
C	5	10

Timelines

ABCABCABABAAAA



0 20 40 60 80

RR

B C A



0 20 40 60 80

SJF

Schedulers:

FIFO

SJF

STCF

RR

B C B A



0 20 40 60 80

STCF

A B C



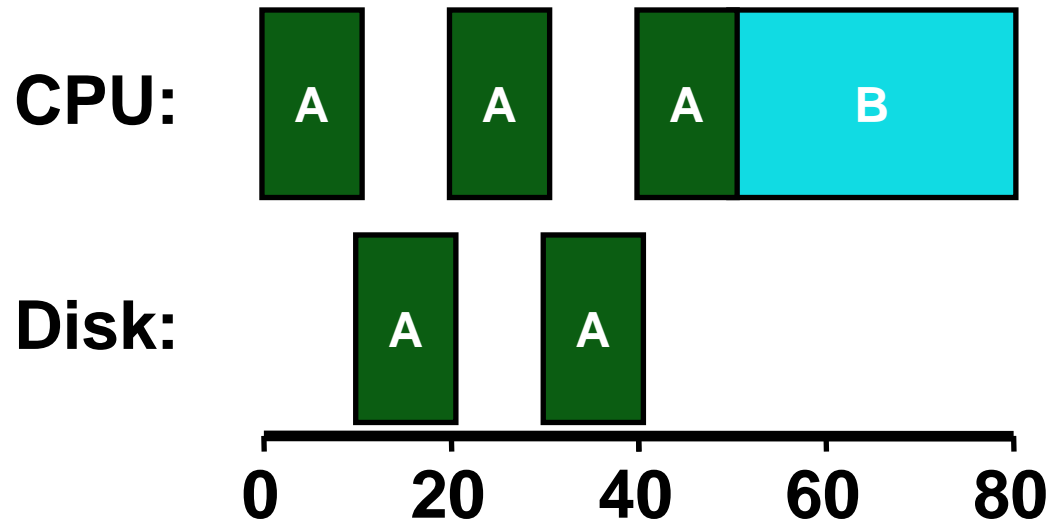
0 20 40 60 80

FIFO

Workload Assumptions

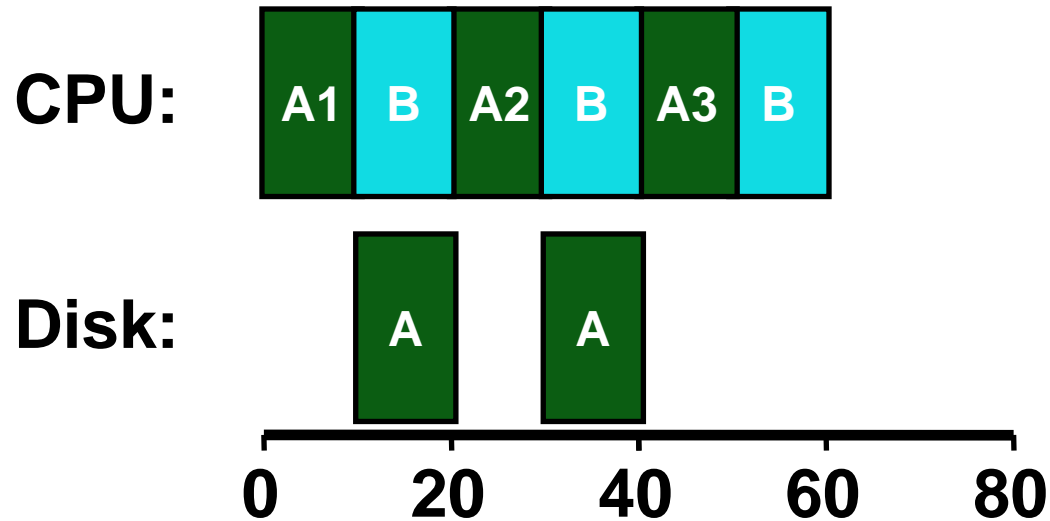
- ~~1. Each job runs for the same amount of time~~
- ~~2. All jobs arrive at the same time~~
- ~~3. All jobs only use the CPU (no I/O)~~
4. The run-time of each job is known

Not I/O Aware



Don't let Job A hold on to CPU while blocked waiting for disk

I/O Aware (Overlap)



Treat Job A as 3 separate **CPU bursts** (sub-jobs)
When Job A completes I/O, another Job A is ready

Each CPU burst is shorter than Job B, so with SCTF,
Job A preempts Job B

Workload Assumptions

- ~~1. Each job runs for the same amount of time~~
- ~~2. All jobs arrive at the same time~~
- ~~3. All jobs only use the CPU (no I/O)~~
- ~~4. The run-time of each job is known
(need smarter, fancier scheduler)~~

MLFQ

(Multi-Level Feedback Queue)

Goal: general-purpose scheduling

Must support two job types with distinct goals


- **interactive** programs care about response time
- **batch** programs care about turnaround time

Approach: multiple levels of round-robin;
each level has higher priority than lower levels and
preempts them


Priorities

Rule 1: If $\text{Priority}(A) > \text{Priority}(B)$, A runs

Rule 2: If $\text{Priority}(A) == \text{Priority}(B)$, A & B run in RR

Q3 → 

“Multi-level”

Q2 → 

How do we set priorities?

Q1

Q0 →  → 

Approach 1: nice

Approach 2: history, “feedback”

History

- Use past behavior of process to predict future behavior
 - Common technique in systems
- Processes alternate between I/O and CPU work
- Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process

More MLFQ Rules

Rule 1: If $\text{priority}(A) > \text{Priority}(B)$, A runs

Rule 2: If $\text{priority}(A) == \text{Priority}(B)$, A & B run in RR

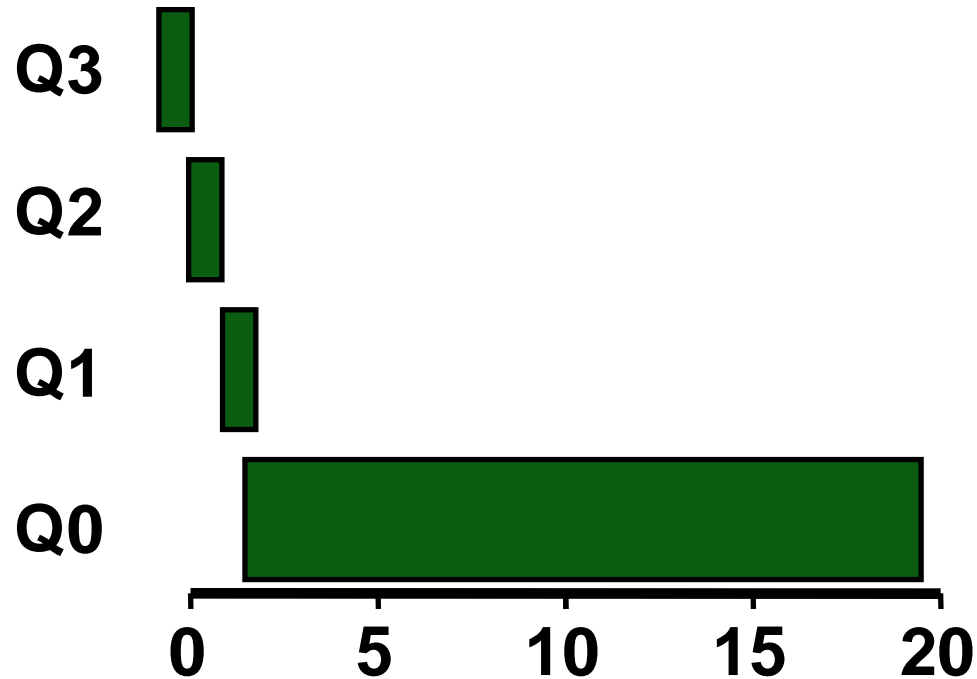
More rules:

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process,
else leave at same level

(longer time slices at lower priorities)

One Long Job (Example)

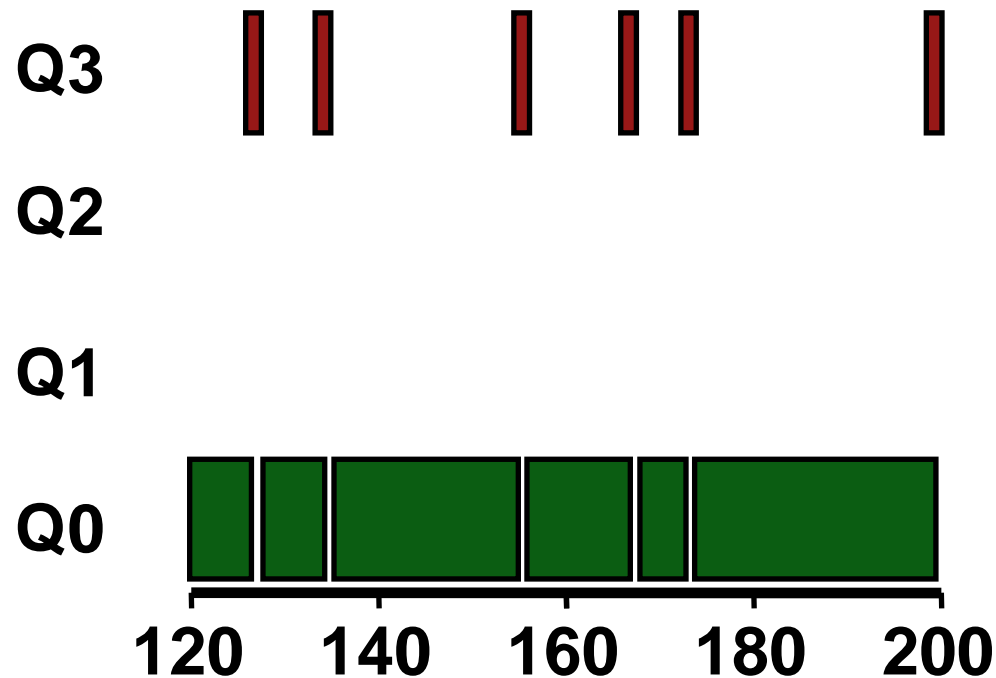


An Interactive Process Joins



Interactive process never uses entire time slice, so never demoted

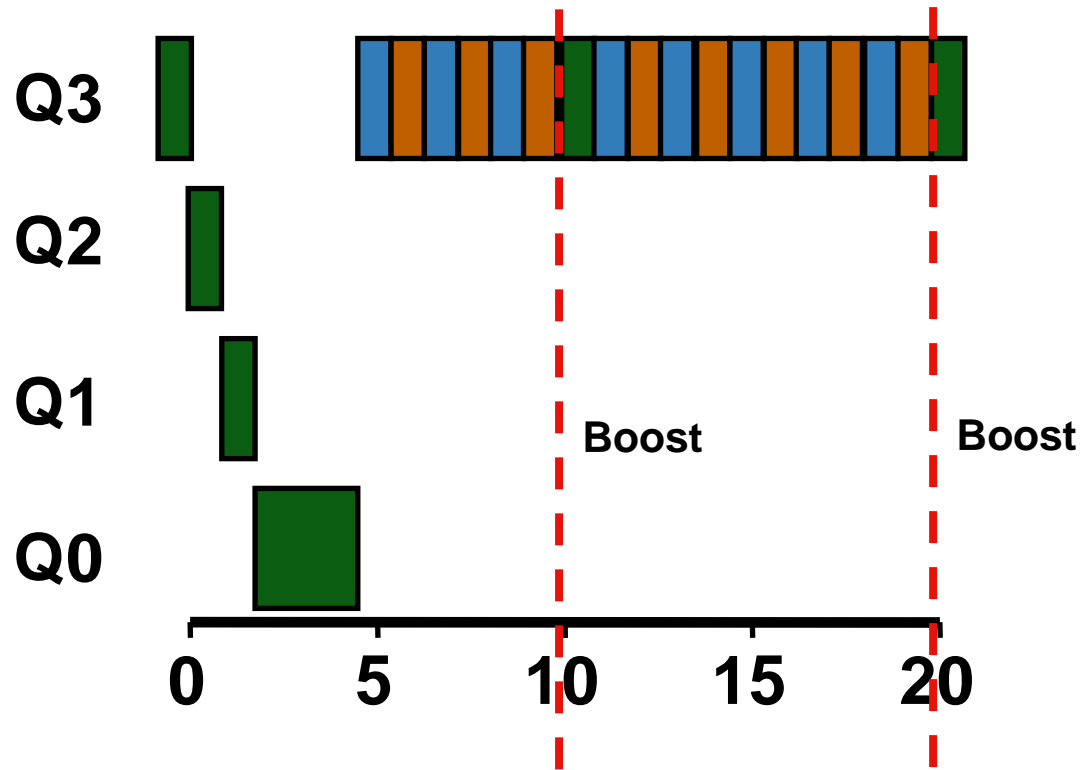
Problems with MLFQ?



Problems

- Unforgiving + **starvation**
- Gaming the system

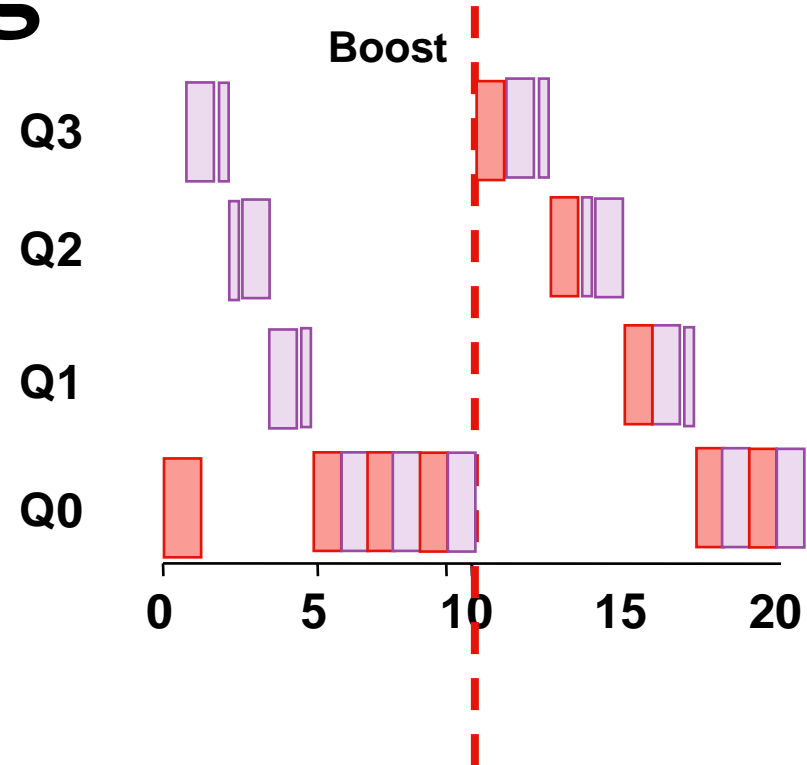
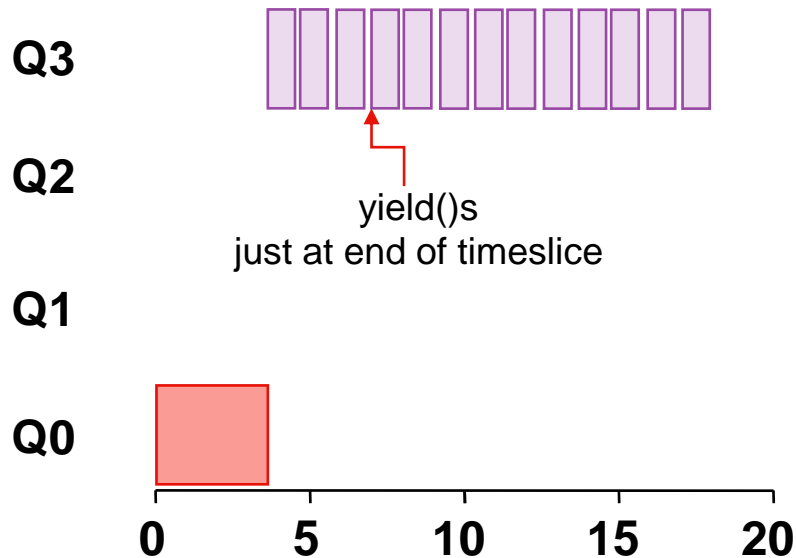
Prevent Starvation



Problem: Low priority job may never get scheduled
and processes may switch between CPU and I/O phases

Periodically boost priority of all jobs
(or all jobs that haven't been scheduled; **aging**)

Prevent Gaming



- Job can yield just before time slice ends to retain CPU
- Fix: Account for job's total run time at priority level (instead of just this time slice); downgrade when threshold exceeded

Lottery Scheduling

Goal: proportional (fair) share, but allow for weights

Approach:

- Give processes lottery tickets
- Whoever wins runs
- More tickets → higher priority/share

Amazingly simple to implement

Lottery example

```
int counter = 0;  
int winner = getRandom(0, totaltickets);
```

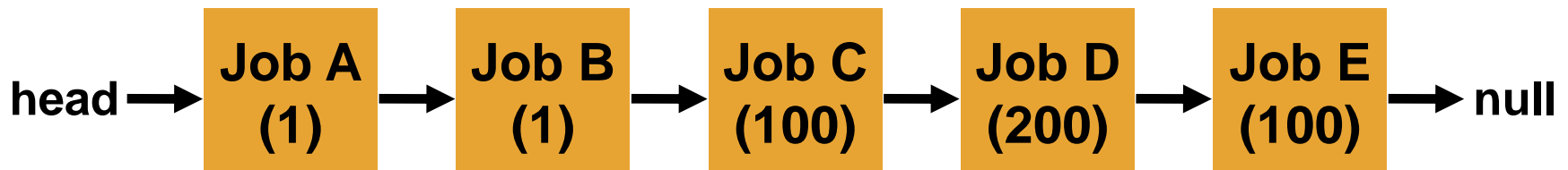
Sum of tickets of all
ready processes

```
node_t *current = head;  
while (current) {  
    counter += current->tickets;  
    if (counter > winner) break;  
    current = current->next;  
}
```

```
// current gets to run
```

Who runs if **winner** is:

50
350
0



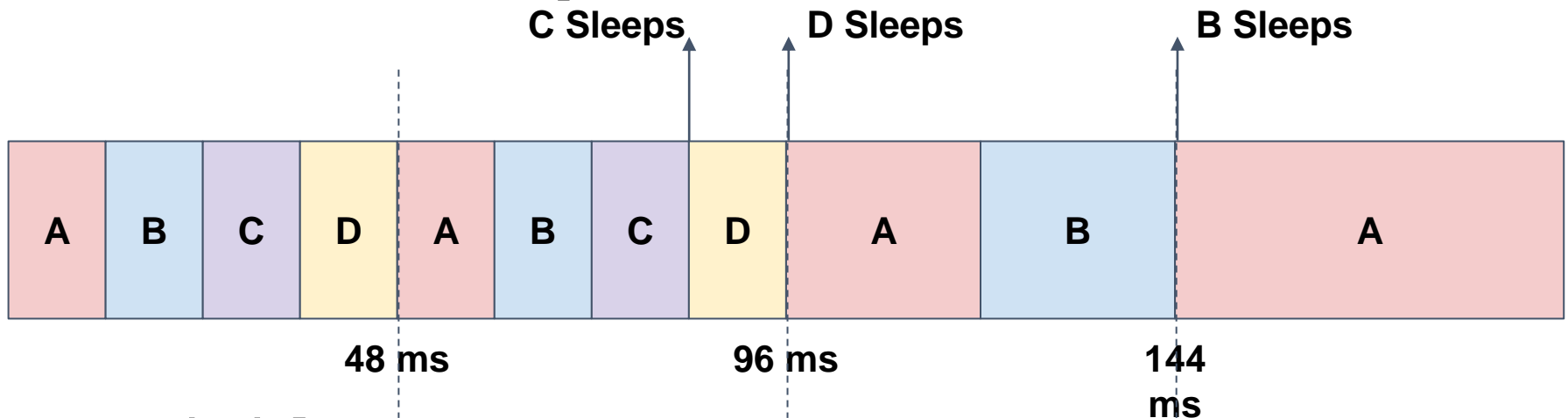
Past Linux Schedulers

- Early Linux: round-robin
- Linux 2.4: tweaks to improve interactive tasks
 - give back half of yielded time in future timeslices
 - iterated over ready queue to score tasks - didn't scale
- Linux 2.6 O(1) scheduler
 - keep per-priority ready queue; pop off head of queue
 - scalable, but lots of heuristics to compensate for fairness, starvation, etc.

Completely Fair Scheduler (CFS)

- In Linux $\geq 2.6.23$
- Proportional share/weighted fair queueing
 - Weight each process according to its priority
 - Divide CPU time evenly among processes
- Challenges
 - **Fairness**: proportional share favors fairness
 - **Interactivity**: maintained with varying timeslice, ensure each process gets some CPU within a fixed period of time by varying per-process timeslice to hit goal
 - **Scaling**: need to sort tasks based on how much CPU share they have consumed; incorporate a red-black tree to maintain order

Basic CFS Operation



Each **sched_latency** interval (default 48 ms), run all runnable tasks once

$$\text{Timeslice} = 48 \text{ ms} / (\text{nr_runnable})$$

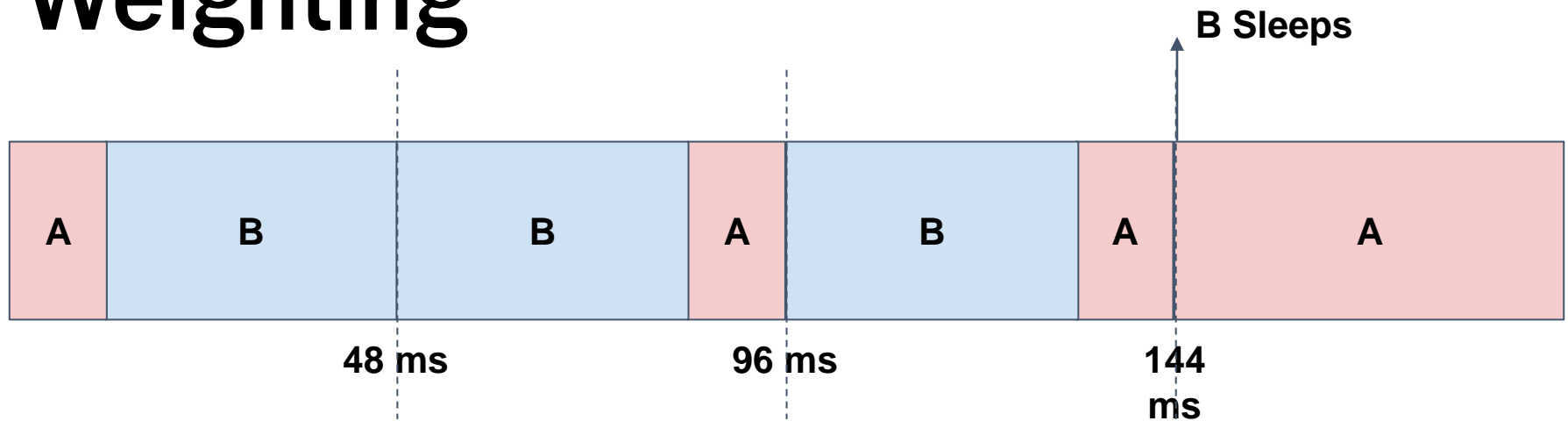
Maintain **vruntime** per task to track CPU consumed

After 48 ms, $\text{vruntime}_A = 12 \text{ ms}$

Context switch costs rise with processes; **min_granularity** (default 6 ms)

Fudge at > 8 runnable threads; fix timeslice, vary response time

Weighting



Tasks are assigned a weighted share of CPU

$$\text{timeslice}_A = (\text{weight}_A / \text{sum}(\text{weight}_X \text{ for runnable } X)) * \text{sched_latency}$$

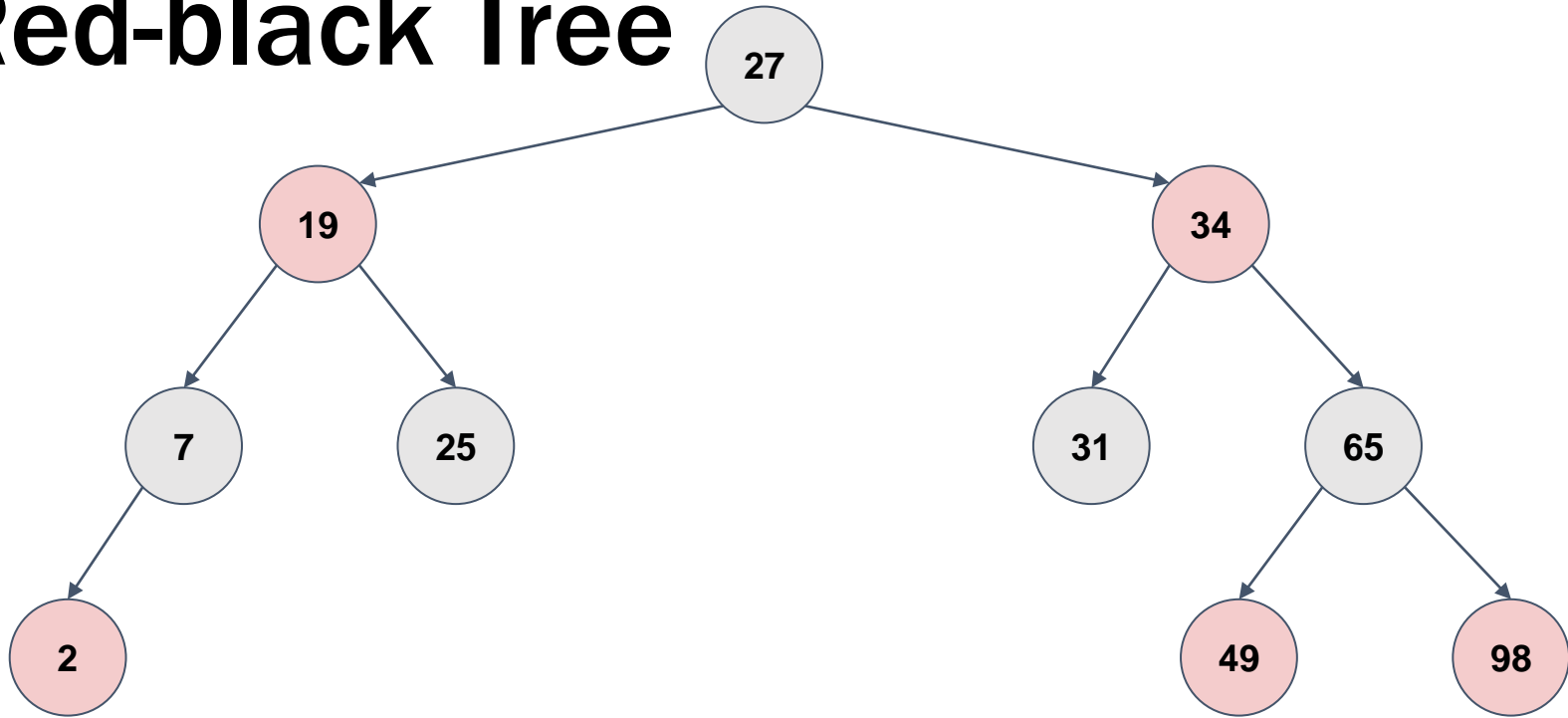
vruntime advances differently per task to enforce share

$$\text{vruntime}_A += \text{timeslice}_A * (\text{weight}_0 / \text{weight}_A)$$

e.g. $\text{weight}_A = 1$, $\text{weight}_B = 3$

Need efficient tracking of processes on sort of **vruntime**

Red-black Tree



Maintain runnable processes by vruntime in a per-core red-black tree

Pop from leftmost internal node, push on wake/spawn into correct location

$O(\lg n)$ in runnable processes both to wake/spawn and to schedule

Scales to thousands of runnable processes

Priorities

Nice values range:

-20 to 19

(high prio to lowest)

Map to weights

Smart, geometric mapping

Ensures equal distances in
niceness result in same
shares

nice(A) = 10, nice(B) = 11

gives same CPU shares as

nice(A) = 0, nice(B) = 1

```
static const int prio to weight[40] = {  
    /* -20 */ 88761, 71755, 56483, 46273, 36291,  
    /* -15 */ 29154, 3254, 18705, 14949, 11916,  
    /* -10 */ 9548, 7620, 6100, 4904, 3906,  
    /* -5 */ 3121, 2501, 1991, 1586, 1277,  
    /* 0 */ 1024, 820, 655, 526, 423,  
    /* 5 */ 335, 272, 215, 172, 137,  
    /* 10 */ 110, 87, 70, 56, 45,  
    /* 15 */ 36, 29, 23, 18, 15,  
};
```

$$110 / (110 + 87) = 0.558$$

$$1024 / (1024 + 820) = 0.555$$

Dealing with I/O and Sleep

- Sleeping processes don't gain vruntime
 - Could allow short-term starvation after a long sleep
- On thread wake, set vruntime of waker to min of runnable processes
 - Waking task executes almost immediately
 - Prefers to give CPU to interactive tasks quickly
- On thread create, set vruntime of new thread to max of runnable
 - New task runs last in the next pass over the processes
- Maintains invariant: no task has $>$ one timeslice of vruntime difference
 - Avoids starvation, but slightly biases to scheduling wakers quickly

Group Scheduling

- Spawning lots of threads or processes gives an advantage
 - Before 2.6.38 each thread's allotment was independent of others
- Break into cgroups based on controlling (p)tty
 - (and now sessions/process groups)
- cgroup vruntime is the sum of all its threads vruntime
 - Within a cgroup choose the min vruntime
 - Works hierarchically with cgroups of cgroups
- systemd policy
 - Each user in a cgroup (fair over users)
 - Then, weighted fair sharing among applications of the user

Nesting Schedulers

- CFS can't meet all needs
- e.g. hard(er) latency bounds
- scheduler classes
 - DEADLINE
 - RT: FIFO, RR (Priority 1-99)
 - OTHER: CFS, or BATCH (Priority 0)
 - IDLE
- FIFO preempts all CFS-tasks and runs to completion
 - Preemption due to higher priority RT tasks will restore task to head of its priority queue
 - yield or I/O puts the task at the end of its priority queue
- RR re-enqueues at the tail of its priority queue at the end of its timeslice
- Newer DEADLINE scheduler
 - Specify **period**, **runtime**, and **deadline**
 - Kernel schedules **runtime** of compute in advance of **deadline** per **period**
 - Under-estimated **runtime** still results in preemption until next period

Summary

Understand goals (metrics) and workload, then design scheduler around that

General purpose schedulers need to support processes with different goals

Past behavior is good predictor of future behavior

Random algorithms (lottery scheduling) can be simple to implement, and avoid corner cases.