Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2024

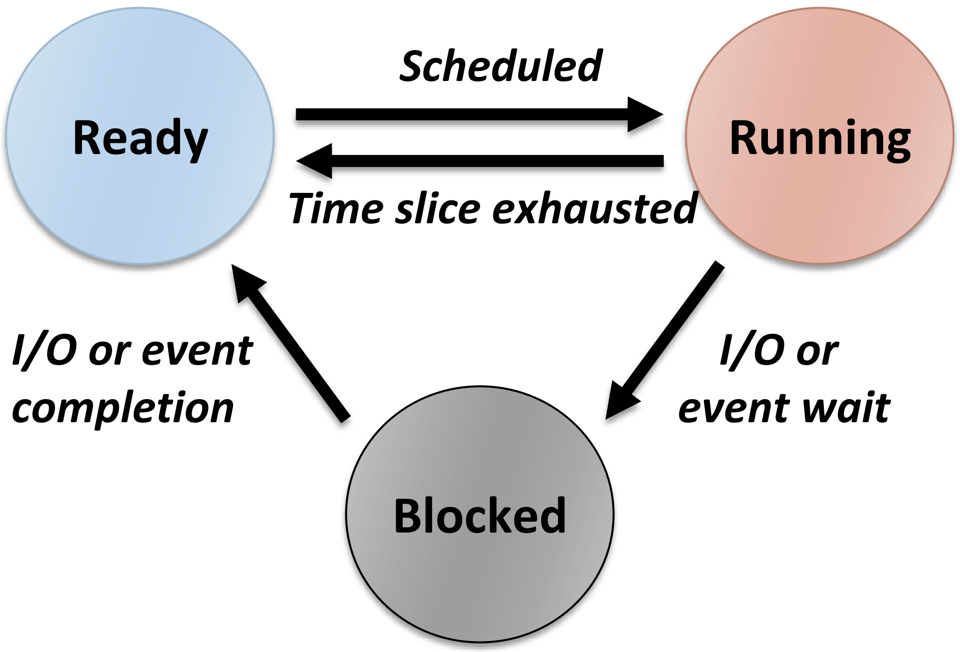
CPU Scheduling

CPU Scheduling

- A policy deciding which process to run next, given a set of runnable processes
  - Happens frequently, hence should be fast

- Policy
  - Who's next?
  - How long?

- Mechanism
  - How to transition?



Basic Approaches

- ______________ scheduling
  - The scheduler waits for the running process to voluntarily yield the CPU
  - Processes should be cooperative

- Preemptive scheduling
  - The scheduler can interrupt a process and force a context switch
  - What happens
    - If a process is preempted in the midst of updating the shared data?
    - If a process in a system call is preempted?

Terminologies

- Workload
  - A set of job descriptions
  - e.g., arrival time, run time, etc.
- Scheduler
  - A logic that decides when jobs run
- Metric
  - Measurement of scheduling quality
  - e.g., turnaround time, response time, fairness, etc.

Workload Assumptions

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

- Metric: Turnaround time

$$T_{turnaround} = T_{completion} - T_{arrival}$$

FIFO

- First-Come, First-Served
  - Jobs are scheduled in order that they arrive
  - "Real-world" scheduling of people in lines
    - e.g., supermarket, bank tellers, McDonalds, etc.
  - Non-preemptive
  - Jobs are treated equally: no starvation

- Problems
  - __________ effect:
    Average turnaround time can be large if small jobs wait behind long ones

SJF

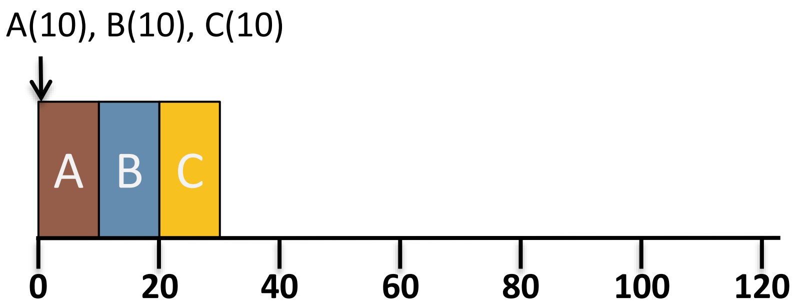
1. ~~*Each job runs for the same amount of time*~~
2. *All jobs arrive at the same time*
3. *Once started, each job runs to completion*
4. *All jobs only use the CPU (no I/O)*
5. *The run time of each job is known*

- Shortest Job First
  - Each job has a variable run time (Assumption 1 relaxed)
  - Choose the job with the smallest run time
  - Can prove that SJF shows the optimal turnaround time under our assumptions
  - Non-preemptive

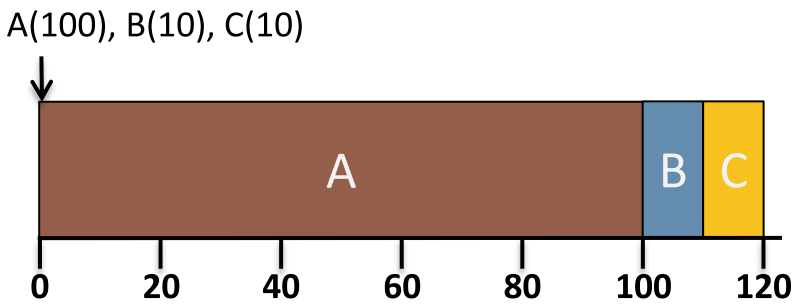
- Problems
  - Not optimal when jobs arrive at any time
  - Can potentially starve

FIFO vs. SJF

- FIFO

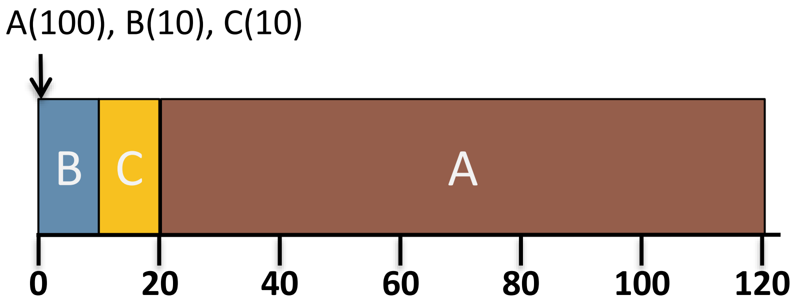


$$T_{turnaround} = (10 + 20 + 30)/3 = 20$$

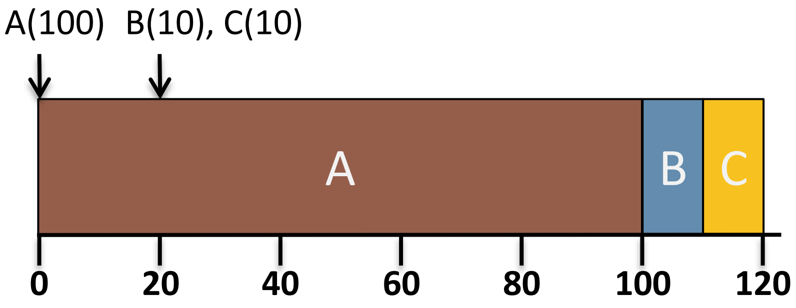


$$T_{turnaround} = (100 + 110 + 120)/3 = 110$$

- SJF



$$T_{turnaround} = (10 + 20 + 120)/3 = 50$$

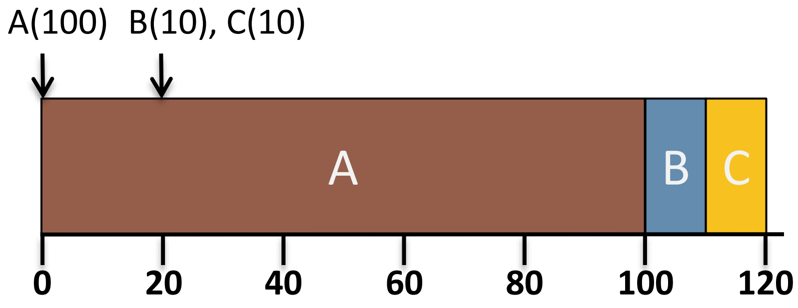


$$T_{turnaround} = (100 + 90 + 100)/3 = 96.7$$

STCF

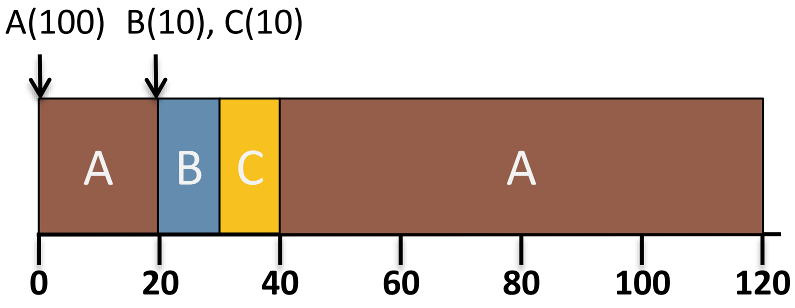
1. ~~*Each job runs for the same amount of time*~~
2. ~~*All jobs arrive at the same time*~~
3. ~~*Once started, each job runs to completion*~~
4. *All jobs only use the CPU (no I/O)*
5. *The run time of each job is known*

- Shortest Time-to-Completion First
  - Jobs are not available simultaneously (Assumption 2 relaxed)
  - Preemptive version of SJF (Assumption 3 relaxed)
  - If a new job arrives with the run time less than the remaining time of the current job, preempt it



$$T_{turnaround} = (100 + 90 + 100)/3 = 96.7$$

SJF



$$T_{turnaround} = (120 + 10 + 20)/3 = 50$$

STCF

RR

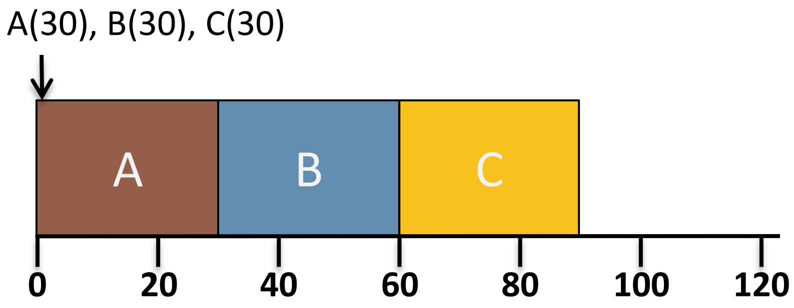
- Round Robin
  - Run queue is treated as a circular FIFO queue
  - Each job is given a time slice (or scheduling quantum)
    - Multiple of the timer-interrupt period or the timer _______
    - Too short → higher context switch overhead
    - Too long → less responsive
    - Usually, 10 ~ 100ms
  - Runs a job for a time slice and then switches to the next job in the run queue
  - Preemptive
  - No starvation
  - Improved response time: great for time-sharing

SJF vs. RR

- RR focuses on a new metric: "response time"

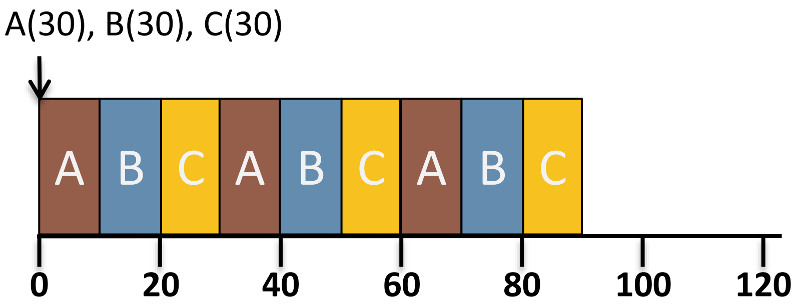
$$T_{response} = T_{firstrun} - T_{arrival}$$

  - Typically, RR has higher turnaround time than SJF, but better response time



$$T_{turnaround} = (30 + 60 + 90)/3 = 60$$
$$T_{response} = (0 + 30 + 60)/3 = 30$$

SJF



$$T_{turnaround} = (70 + 80 + 90)/3 = 80$$
$$T_{response} = (0 + 10 + 20)/3 = 10$$

RR

(Static) Priority Scheduling

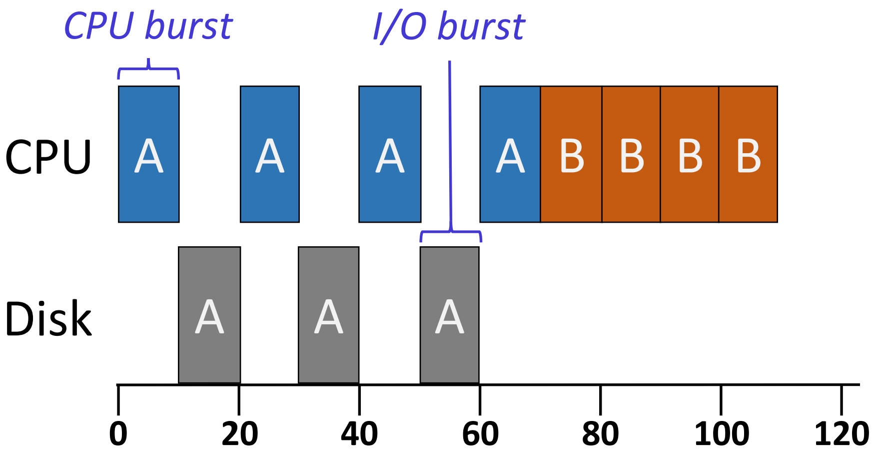
- Each job has a (static) priority
  - cf.) `nice()`, `renice()`, `setpriority()`, `getpriority()`
- Choose the job with the highest priority to run next
- Round-robin or FIFO within the same priority
- Can be either preemptive or non-preemptive

- Starvation problem
  - If there is an endless supply of high priority jobs, no low priority job will ever run

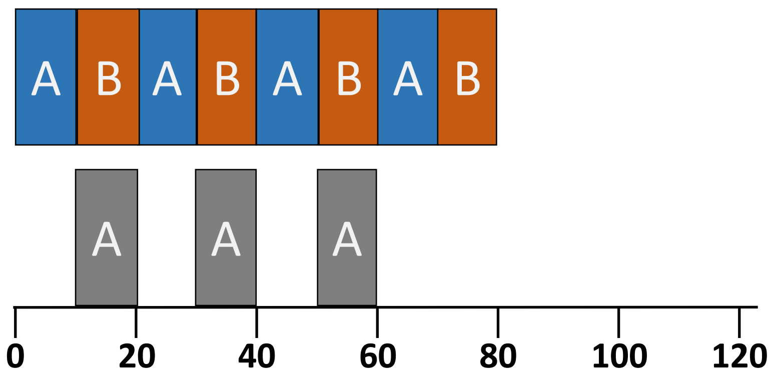
Incorporating I/O

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

- I/O-aware scheduling
  - Assumption 4 relaxed
  - Overlap computation with I/O
  - Treat each CPU burst as an independent job
- Example: A (interactive) + B (CPU-intensive)



STCF



I/O-aware STCF

Towards a General CPU Scheduler

- Goals
  - Optimize turnaround time
  - Minimize response time for interactive jobs

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

- Challenge: No *a priori* knowledge on the workloads
  - ~~The run time of each job is known (Assumption 5)~~

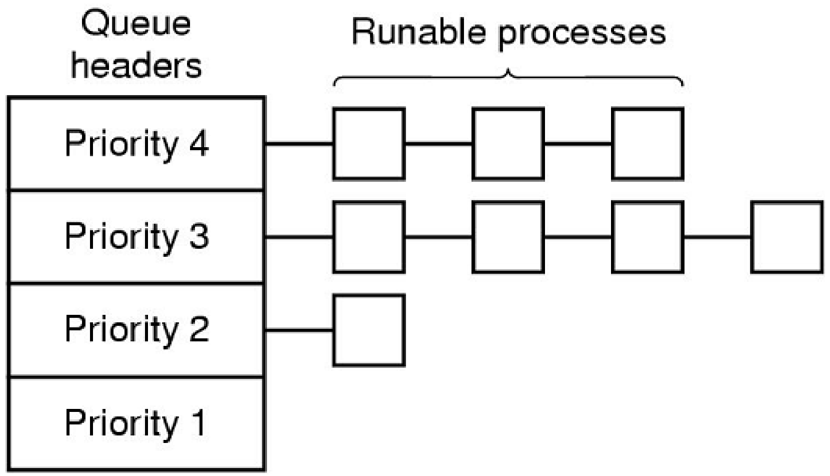
- How can the scheduler learn the characteristics of the jobs and make better decisions?
  - Learn from the past to predict the future
    (as in branch predictors or cache algorithms)

MLFQ

- Multi-Level Feedback Queue
  - A number of distinct queues for each priority level
  - Priority scheduling between queues, round-robin in the same queue

> **Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).**
>
> **Rule 2: If Priority(A) = Priority(B), A & B run in RR.**

  - Priority is varied based on its observed behavior



Changing Priority

- Typical workload: a mix of
  - Interactive jobs: short-running, require fast response time
  - CPU-intensive jobs: need a lot of CPU time, don't care about response time
- Attempt #1: Dynamic Priority Change

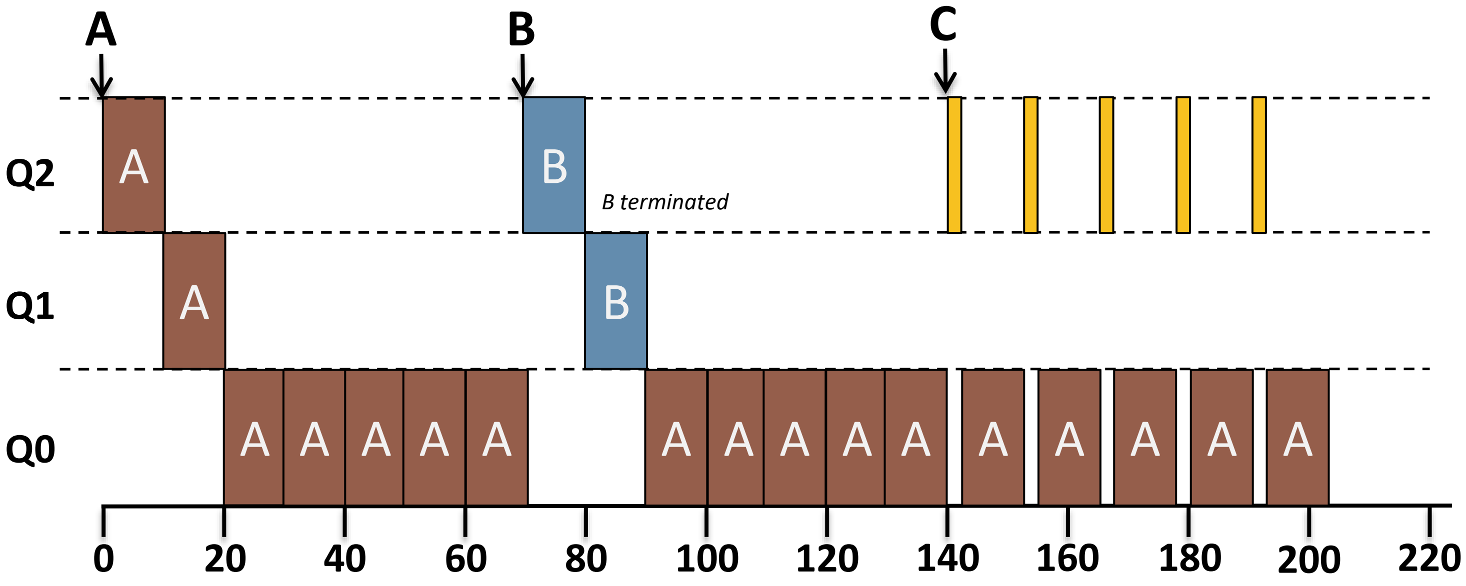
Rule 3: **When a job enters the system, it is placed at the highest priority (the topmost queue).**

Rule 4a: **If a job uses up an entire time slice while running, its priority is reduced (i.e., moves down one queue).**

Rule 4b: **If a job gives up the CPU before the time slice is up, it stays at the same priority level.**

Scheduling Under Rules 1-4

- Workload
  - A: long-running job, B: short-running job, C: interactive job



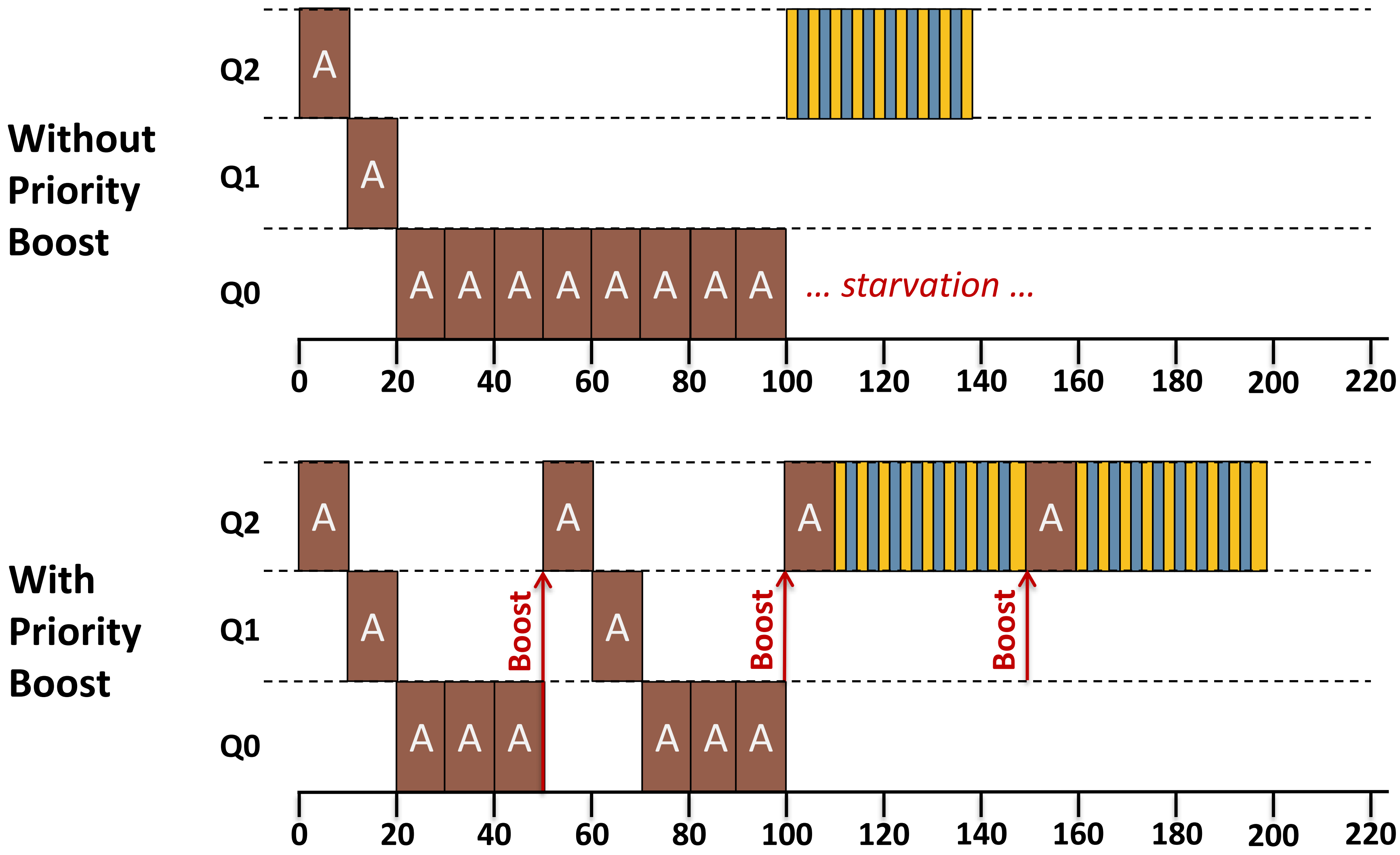
Priority Boost

- Problems in Attempt #1
  - Long-running jobs can starve due to too many interactive jobs
  - A malicious user can game the scheduler by relinquishing the CPU just before the time slice is expired
  - A program may change its behavior over time

- Attempt #2: Priority Boost

> **Rule 5: After some time period *S*, move all the jobs in the system to the topmost queue.**

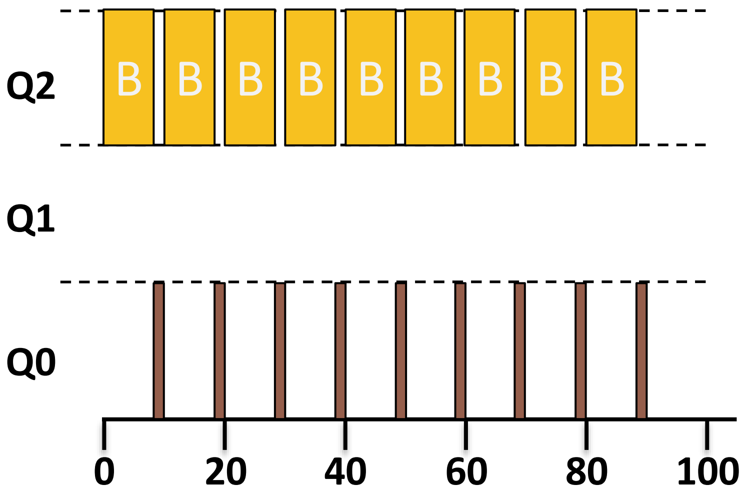
Scheduling Under Rules 1-5



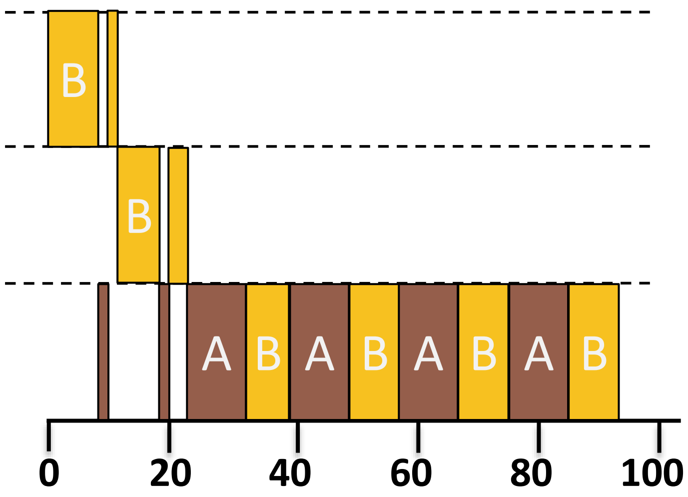
Better Accounting

- Attempt #3: Revise Rule 4a/4b for better accounting

> **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.



Without precise accounting



With precise accounting

Summary: Unix Scheduler

- MLFQ
  - Preemptive priority scheduling
  - Time-shared based on time slice
  - Processes dynamically change priority
  - 3~4 classes spanning ~170 priority levels (Solaris 2)
- Favor interactive processes over CPU-bound processes
- Use _______: no starvation
  - Increase priority as a function of wait time
  - Decrease priority as a function of CPU time
- Many ugly heuristics for *voo-doo* constants

Linux CFS
(Completely Fair Scheduler)

Linux Scheduler Evolution

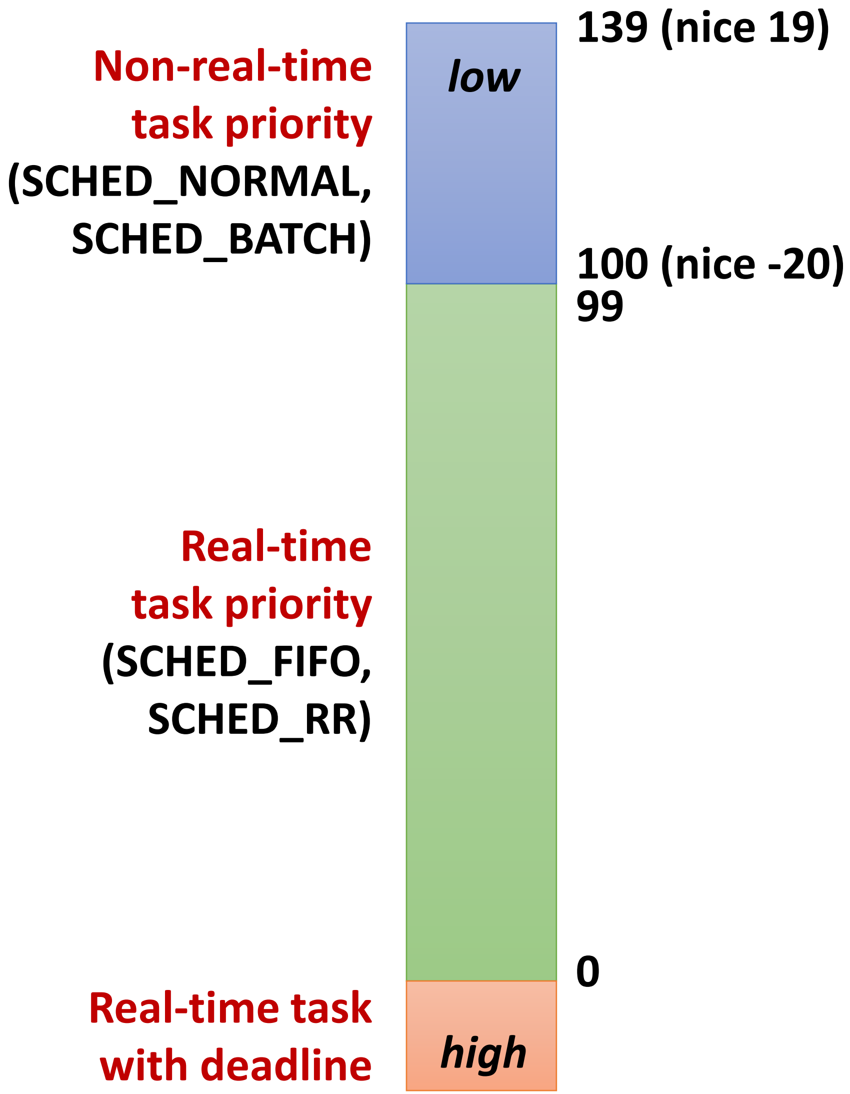
Kernel version	CPU Scheduler
Linux 2.4	• Epoch-based priority scheduling • O(*n*) scheduler
Linux 2.6 ~ 2.6.22	• Active / expired arrays with bitmaps • Per-core run queue • O(1) scheduler
Linux 2.6.23 ~	• CFS (Completely Fair Scheduler) by Ingo Molnar
Linux 3.14 ~	• Sporadic task model deadline scheduling (SCHED_DEADLINE)

Linux Scheduling Classes

Class	Description	Policy
DL	• For real-time tasks with deadline • Highest priority	SCHED_DEADLINE
RT	• For real-time tasks	SCHED_FIFO SCHED_RR
Fair	• For time-sharing tasks	SCHED_NORMAL SCHED_BATCH
Idle	• For per-CPU idle tasks	SCHED_IDLE

Linux Task Priority

- Total 140 levels (0 ~ 139)
  - A smaller value means higher priority
- Setting priority for non-real-time tasks
  - `nice()`, `setpriority()`
  - -20 ≤ nice value ≤ 19
  - Default nice value = 0 (priority value 120)
- Setting priority for real-time tasks
  - `sched_setattr()`
  - Static priority for SCHED_FIFO & SCHED_RR
  - Runtime, deadline, period for SCHED_DEADLINE



Proportional Share Scheduling

- Basic concept
  - A weight value is associated with each task
  - The CPU is allocated to task in proportion to its weight

Task A (weight 2)

Task B (weight 1)

Task C (weight 4)

Task D (weight 1)

Task A's share = $\frac{weight_A}{\sum weight_i} = \frac{2}{8} = 25.0\%$

Nice to Weight

- How to map nice values to weights?
  - Wants a task to get ~10% less CPU time when it goes from nice i to nice i+1
  - This will make another task remained on nice i have ~10% more CPU time
  - weight(i)/weight(i+1) = 0.55/0.45 = 1.22 (or ≃ 25% increase)
- Examples
  - T_1 (nice 0), T_2 (nice 1)
    - T_1: 1024/(1024+820) = 55.5%
    - T_2: 820/(1024+820) = 44.5%
  - + T_3 (nice 1)
    - T_1: 1024/(1024+820*2) = 38.4%
    - T_2: 820/(1024+820*2) = 30.8%
    - T_3: 820/(1024+820*2) = 30.8%

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     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,
};
```

Virtual Runtime

- Approximate the "ideal multitasking" that CFS is modeling
- Normalize the actual runtime to the case with nice value 0

$$VR(T) = \frac{Weight_0}{Weight(T)} \times PR(T) = \left(Weight_0 \times \boxed{\frac{2^{32}}{Weight(T)}} \times PR(T) \right) \gg 32$$

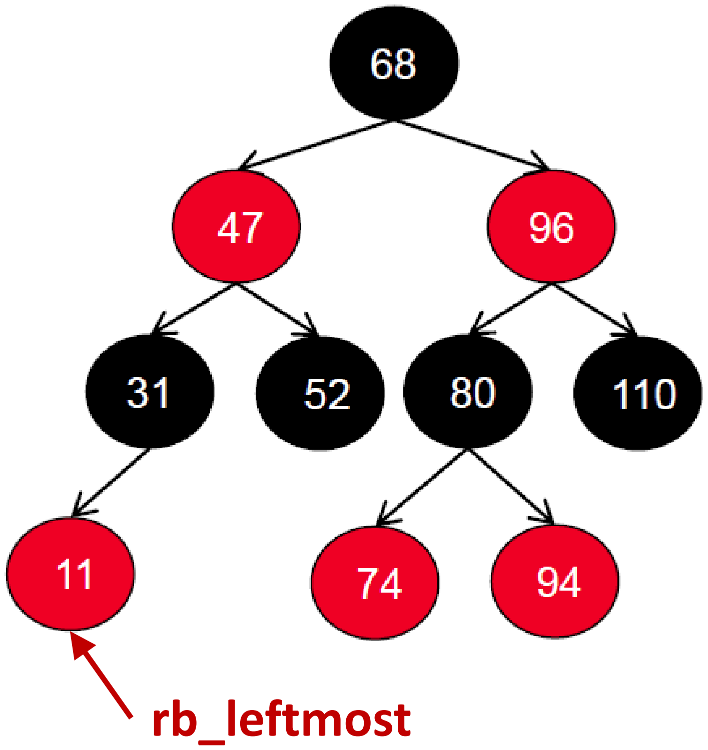
precomputed:
sched_prio_to_wmult[]

  - *Weight₀*: the weight of nice value 0
  - *Weight(T)*: the weight of the task T
  - *PR(T)*: the actual runtime of the task T
  - *VR(T)*: the virtual runtime (*vruntime*) of the task T

- For a high-priority task, its *vruntime* increases slowly

Runqueue

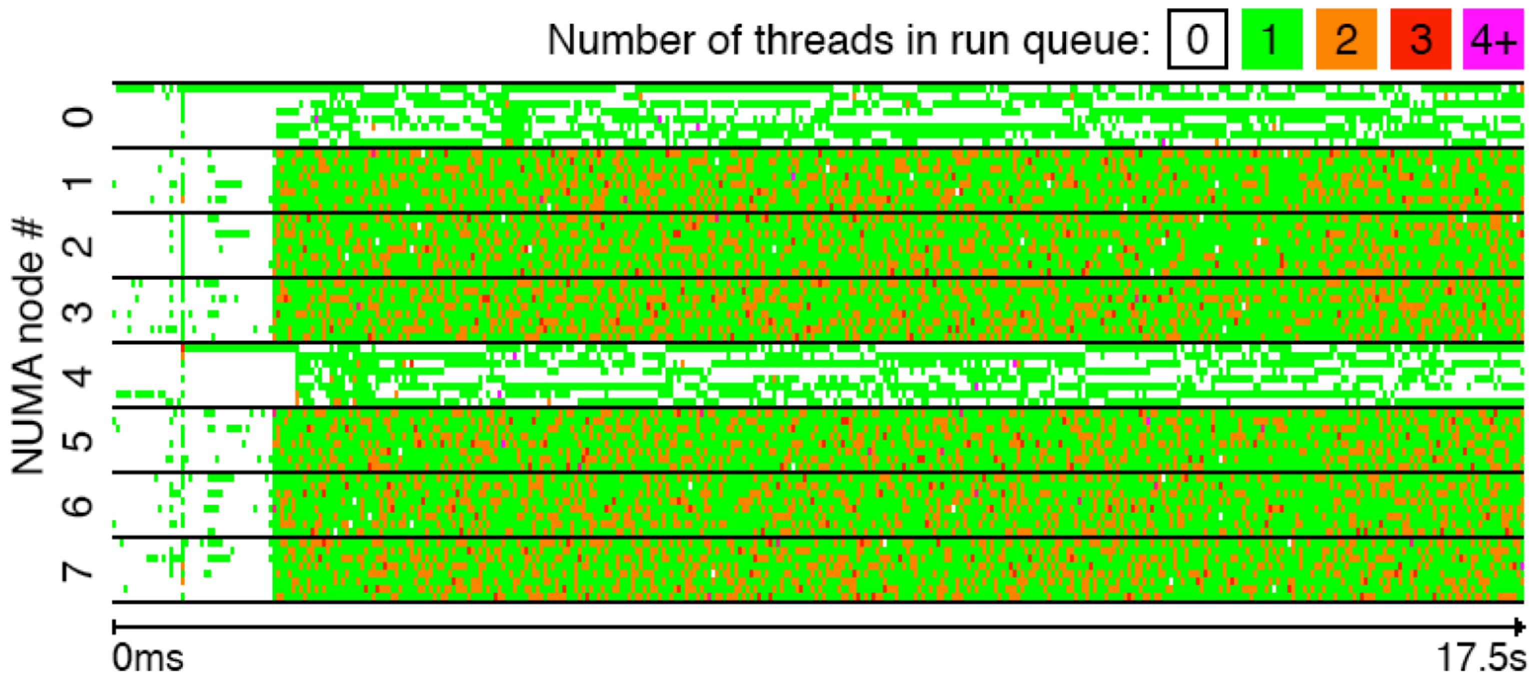
- CFS maintains a red-black tree where all runnable tasks are sorted by *vruntime*
  - Self-balancing binary search tree
  - The path from the root to the farthest leaf is no more than twice as long as the path to the nearest leaf
  - Tree operations in O(log N) time
  - The leftmost node indicates the smallest vruntime



- Choose the task with the smallest virtual runtime (*vruntime*)
  - Small virtual runtime means that the task has received less CPU time than what it should have received

Challenges

- Fairness between groups of threads
  - Session groups, cgroups
- Load balancing among CPU cores



Source: J.-P. Lozi et al., The Linux Scheduler: a Decade of Wasted Cores, EuroSys, 2016.