Scheduling: MFQ

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Scheduling: Goals

1. Generate illusion of concurrency

2. Maximize resource utilization (e.g., mix CPU and I/O bound processes appropriately)

3. Meet needs of both I/O-bound and CPU-bound processes
   - Give I/O-bound processes better interactive response
   - Do not starve CPU-bound processes

4. Support Real-Time (RT) applications

Algorithm: Multi-level Feedback Queue (MFQ)

★ **Algorithm:** Given a small, initial amount of CPU time to every task as soon as it needs the CPU ("P1 queue").

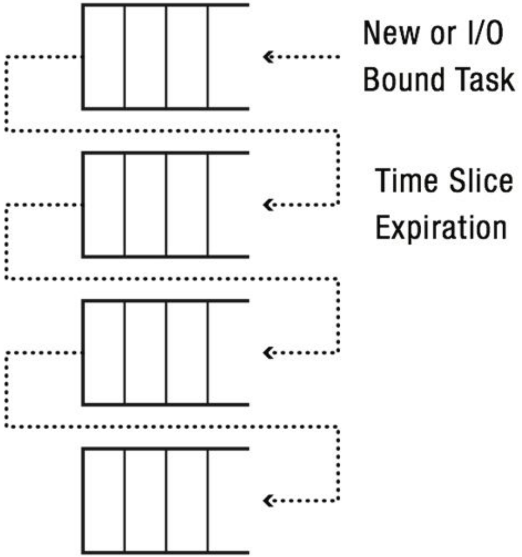
★ If the task still needs additional CPU time, move the job to a lower priority queue (ex: "P2 queue").

★ All jobs in the highest priority queue will run first, but CPU time allocated increases in the lower-priority queues.

Algorithm: Multi-level Feedback Queue (MFQ)

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

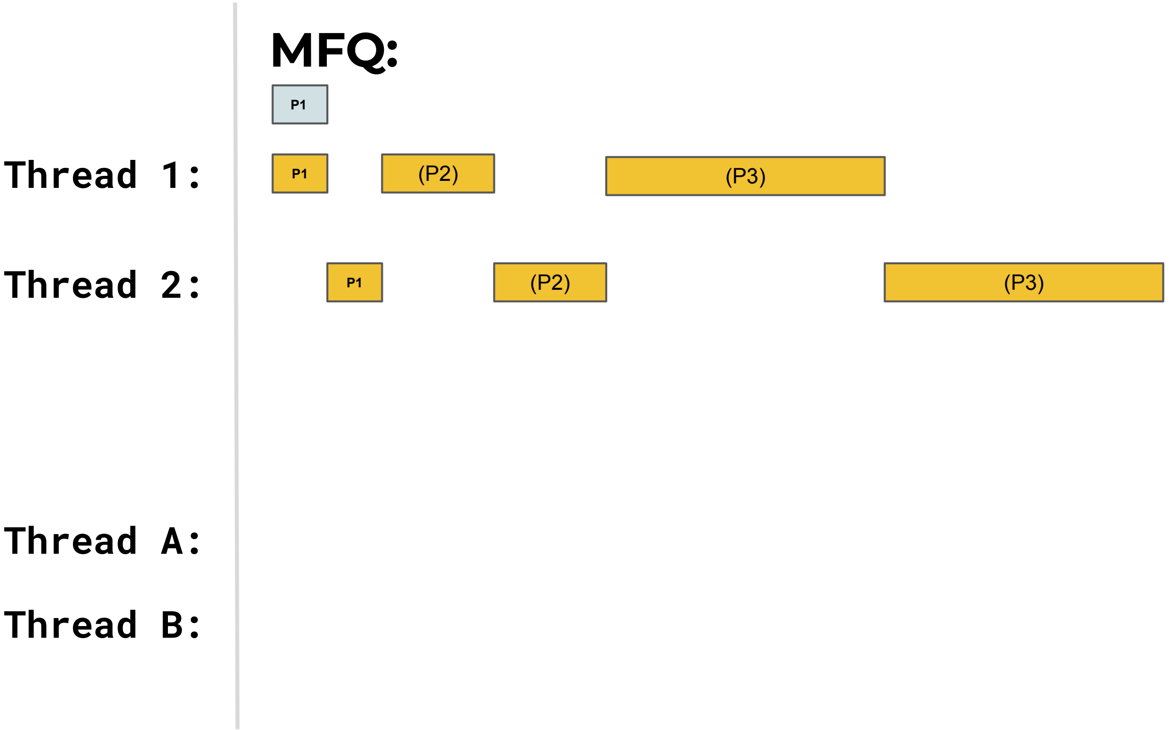
Round Robin Queues



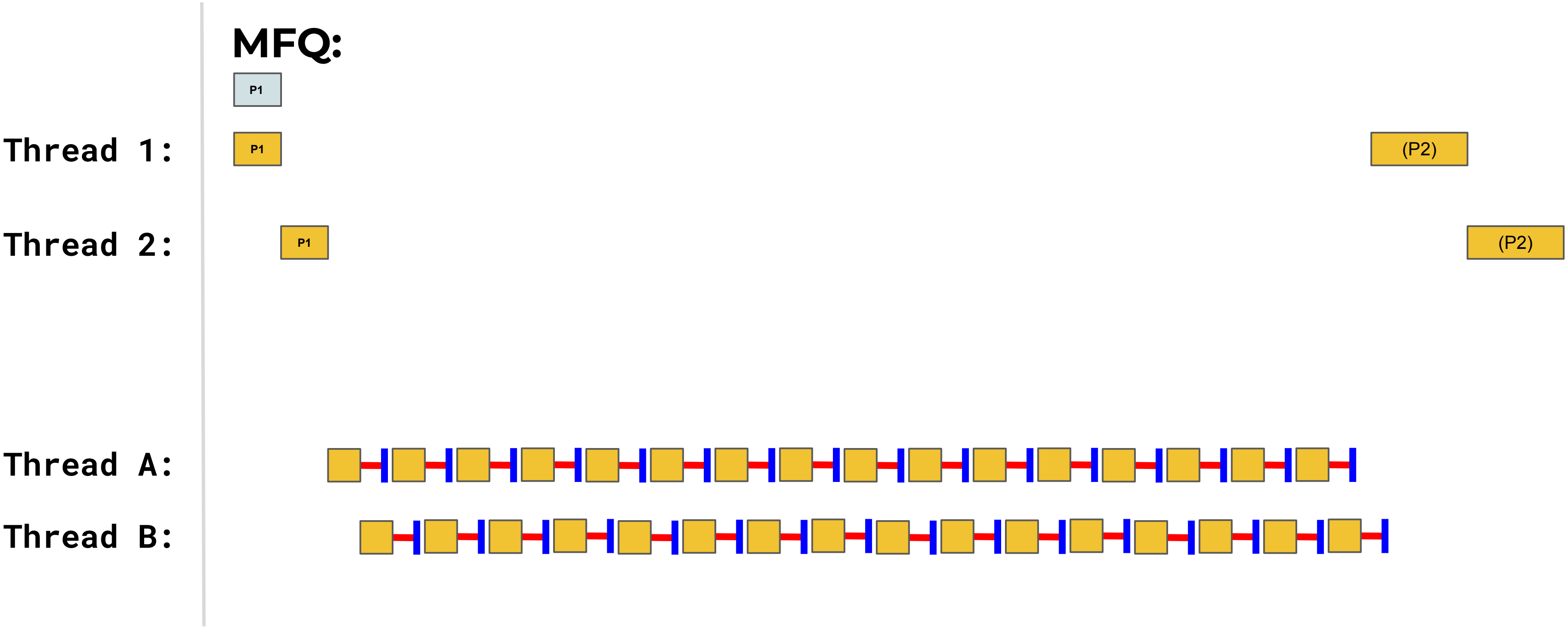
Why is MFQ a good design?

- ★ How to design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?
  - SJF assumes to know the future (how short is the job?)

MFQ Runtime



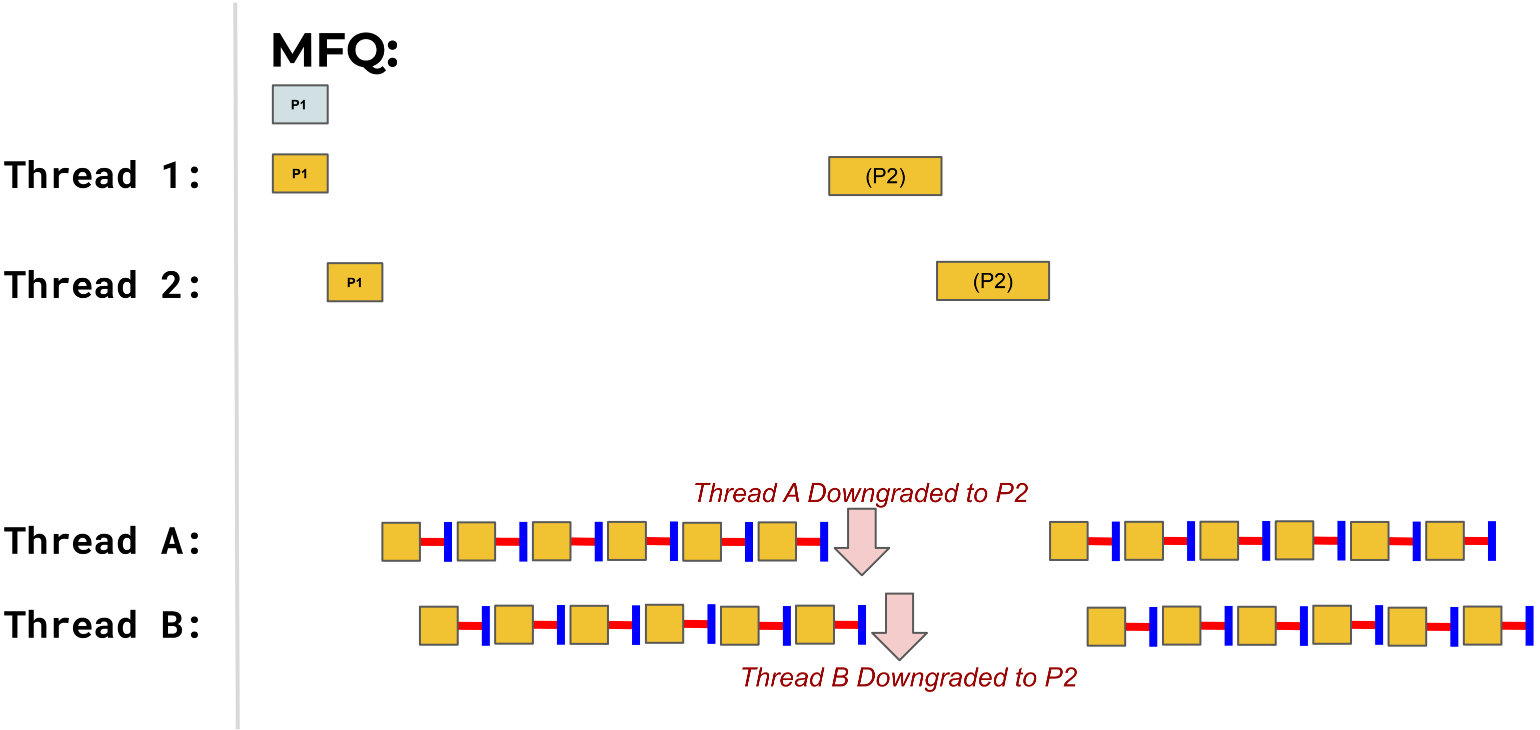
MFQ Runtime



MFQ Accounting

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 (i.e., it moves down one queue).

MFQ Runtime


MFQ:
P1
Thread 1:
P1
(P2)
Thread 2:
P1
(P2)
Thread A Downgraded to P2
Thread A:
Thread B:
Thread B Downgraded to P2

MFQ Design Questions?

★ How many queues should there be?

★ How big should the time slice be per queue?

★ How often should priority be boosted in order to avoid starvation and account for changes in behavior?

Scheduling in Linux

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Early Linux Schedulers

★ **Linux 1.2 (1995)**: circular queue w/ round-robin policy.
  ○ Simple and minimal.
  ○ Did not meet many of the scheduling goals we discussed

★ **Linux 2.2 (2000)**: introduced scheduling classes:
  ○ real-time
  ○ non-real-time

```
/* Scheduling Policies
*/
#define SCHED_OTHER  0 // Normal user tasks (default)
#define SCHED_FIFO   1 // RT: Will almost never be preempted
#define SCHED_RR     2 // RT: Prioritized RR queues
```

Early Linux Schedulers

```
/* Scheduling Policies
*/
#define SCHED_OTHER  0 // Normal user tasks (default)
#define SCHED_FIFO   1 // RT: Will almost never be preempted
#define SCHED_RR     2 // RT: Prioritized RR queues
```

- **SCHED_FIFO**
  - Used for real-time processes
  - Conventional preemptive fixed-priority scheduling
    - Current process continues to run until it ends or a higher-priority real-time process becomes runnable
  - Same-priority processes are scheduled FIFO

```
/* Scheduling Policies
*/
#define SCHED_OTHER  0 // Normal user tasks (default)
#define SCHED_FIFO   1 // RT: Will almost never be preempted
#define SCHED_RR     2 // RT: Prioritized RR queues
```

- **SCHED_RR**
  - Used for real-time processes
  - CPU "partitioning" among same priority processes
    - Current process continues to run until it ends or its time quantum expires
    - Quantum size determines the CPU share
  - Processes of a lower priority run when no processes of a higher priority are present

Early Linux Schedulers

★ **Linux 2.4 (Jan. 2001)**: introduced time slicing:

- Epochs → slices: when blocked before the slice ends, half of the remaining slice is added in the next epoch.

- Simple.
- Lacked scalability.
- Weak for real-time systems.

Modern Linux Scheduling

- **Linux 2.6.23 (Oct. 2007)**: Completely Fair Scheduler (CFS)
  - O(1) scheduler
  - Tasks are indexed according to their priority:
    - Real-time tasks ⇒ [0, 99]
    - Non-real-time tasks ⇒ [100, 139]

SCHED_NORMAL

- ★ Used for non real-time processes with a complex heuristic to balance the needs of I/O and CPU centric applications.

- ★ **Static Priority:**
  - Processes start at 120 by default
    - Augmented by a “nice” value: 19 to -20.
    - Inherited from the parent process
    - Altered by user (negative values require special permission)
- ★ **Dynamic Priority:**
  - Based on static priority and applications characteristics (interactive or CPU-bound)
  - Favor interactive applications over CPU-bound ones
- ★ Timeslice is mapped from priority.

Static Priority CPU Translation

Description	Static priority	Nice value	Base time quantum
Highest static priority	100	-20	800 ms
High static priority	110	-10	600 ms
Default static priority	120	0	100 ms
Low static priority	130	+10	50 ms
Lowest static priority	139	+19	5 ms

top

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1559	apache	20	0	316540	9132	5016	S	8.0	0.5	0:00.48	httpd
1542	apache	20	0	320796	14888	10660	R	6.0	0.7	0:00.43	httpd
1318	apache	20	0	316576	10840	6604	S	0.3	0.5	0:00.54	httpd
1487	apache	20	0	316120	8940	5296	S	0.3	0.4	0:00.02	httpd
1552	apache	20	0	316292	8900	4912	S	0.3	0.4	0:00.25	httpd
1	root	20	0	125660	4080	2476	S	0.0	0.2	10:19.63	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:03.76	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
7	root	20	0	0	0	0	S	0.0	0.0	2:00.42	ksoftirqd/0
8	root	20	0	0	0	0	I	0.0	0.0	30:04.95	rcu_sched
9	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_bh
10	root	rt	0	0	0	0	S	0.0	0.0	0:01.49	migration/0
11	root	rt	0	0	0	0	S	0.0	0.0	0:30.39	watchdog/0
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0
13	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/1
14	root	rt	0	0	0	0	S	0.0	0.0	0:30.86	watchdog/1
15	root	rt	0	0	0	0	S	0.0	0.0	0:01.35	migration/1
16	root	20	0	0	0	0	S	0.0	0.0	1:47.62	ksoftirqd/1
18	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/1:0H
20	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kdevtmpfs
21	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	netns

PR: System Priority:

- rt == real-time
- [0, 39] == non-real-time

NI: “Nice” Value

- [-20, 19] == niceness

top

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
19314	yikait2	20	0	1024332	45260	17212	S	21.8	0.0	0:04.61	caja
19205	yikait2	20	0	1614484	120616	9976	S	8.9	0.1	0:07.60	Xorg
19842	setroub+	20	0	386100	69152	11732	S	3.3	0.1	0:01.78	setroubleshootd
1641	fastx	20	0	1394068	167020	13684	S	2.0	0.1	750:05.04	node
9	root	20	0	0	0	0	S	1.0	0.0	147:30.96	rcu_sched
20225	waf	20	0	164476	2612	1596	R	1.0	0.0	0:00.27	top
48902	root	20	0	1607636	79088	6472	S	0.7	0.1	19:52.92	salt-minion
41	root	20	0	0	0	0	S	0.3	0.0	23:05.85	ksoftirqd/6
69	root	rt	0	0	0	0	S	0.3	0.0	2:17.31	watchdog/12
708	root	20	0	278464	132776	131612	S	0.3	0.1	4:56.77	systemd-journal
1009	root	20	0	0	0	0	S	0.3	0.0	12:09.11	xfsaild/dm-5
1632	root	20	0	584156	22028	6692	S	0.3	0.0	2:48.24	tuned
1667	root	20	0	966076	91396	4896	S	0.3	0.1	102:22.28	f2b/server
15832	root	20	0	0	0	0	S	0.3	0.0	0:00.02	kworker/5:1
19204	yikait2	20	0	161412	10788	1492	S	0.3	0.0	0:00.12	FastX monitor 1
19859	waf	20	0	169568	2704	1244	S	0.3	0.0	0:00.03	sshd
46084	bjzhang2	20	0	1090328	61924	16540	S	0.3	0.0	0:25.44	node
1	root	20	0	194340	7404	4232	S	0.0	0.0	19:09.92	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:08.38	kthreadd
4	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H

PR: System Priority:
- rt == real-time
- [0, 39] == non-real-time

NI: “Nice” Value
- [-20, 19] == niceness

Static Priority CPU Translation

Description	Static priority	Nice value	Base time quantum
Highest static priority	100	-20	800 ms
High static priority	110	-10	600 ms
Default static priority	120	0	100 ms
Low static priority	130	+10	50 ms
Lowest static priority	139	+19	5 ms

Dynamic Priority

bonus = min (10, (avg. sleep time / 100) ms)

- avg. sleep time is 0 => bonus is 0
- avg. sleep time is 100 ms => bonus is 1
- avg. sleep time is 1000 ms => bonus is 10
- avg. sleep time is 1500 ms => bonus is 10
- Your bonus increases as you sleep more.

dynamic priority =

max (100, min (static priority – bonus + 5, 139))

Completely Fair Scheduler (Linux)

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Completely Fair Scheduler (CFS)

- **Basic Idea:**
  - **Virtual Runtime (vruntime)**: When a process runs it accumulates “virtual time.”
    - If priority is high, virtual time accumulates slowly.
    - If priority is low, virtual time accumulates quickly.
  - Virtual Runtime is a “catch up” policy — task with smallest amount of virtual time gets to run next.

Completely Fair Scheduler (CFS)

- ★ Scheduler **maintains a red-black tree** where nodes are ordered according to received virtual execution time.
- ★ Node with smallest virtual received execution time is picked next.
- ★ Priorities determine accumulation rate of virtual execution time.
- ★ Higher priority ⇒ slower accumulation rate.

CFS - Example

- ★ **Setup:**
  - Three tasks A, B, C accumulate virtual time at a rate of 1, 2, and 3, respectively.

- ★ **Q:** What is the expected share of the CPU that each gets?

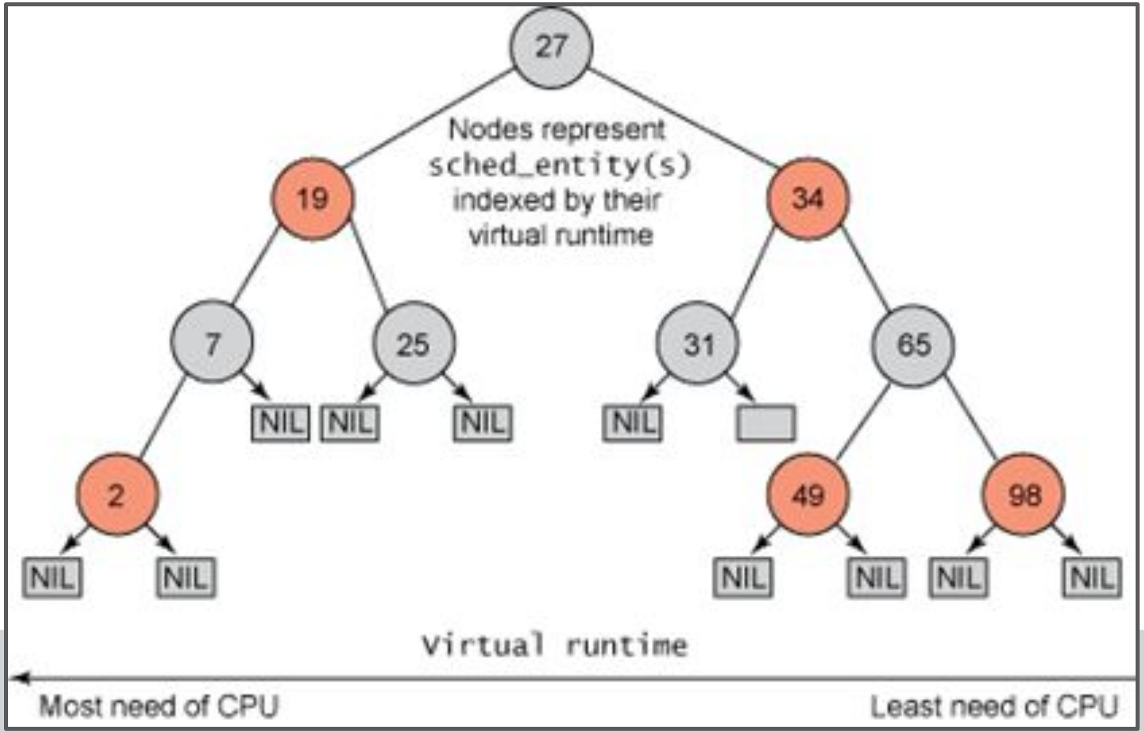
```
Q00: - => {A:0, B:0, C:0}
Q01: A => {A:1, B:0, C:0}
Q02: B => {A:1, B:2, C:0}
Q03: C => {A:1, B:2, C:3}
Q04: A => {A:2, B:2, C:3}
Q05: B => {A:2, B:4, C:3}
Q06: A => {A:3, B:4, C:3}
Q07: A => {A:4, B:4, C:3}
Q08: C => {A:4, B:4, C:6}
Q09: A => {A:5, B:4, C:6}
Q10: B => {A:5, B:6, C:6}
Q11: A => {A:6, B:6, C:6}
```

```
Q00: - => {A:0, B:0, C:0}
Q01: A => {A:1, B:0, C:0}
Q02: B => {A:1, B:2, C:0}
Q03: C => {A:1, B:2, C:3}
Q04: A => {A:2, B:2, C:3}
Q05: B => {A:2, B:4, C:3}
Q06: A => {A:3, B:4, C:3}
Q07: A => {A:4, B:4, C:3}
Q08: C => {A:4, B:4, C:6}
Q09: A => {A:5, B:4, C:6}
Q10: B => {A:5, B:6, C:6}
Q11: A => {A:6, B:6, C:6}
```

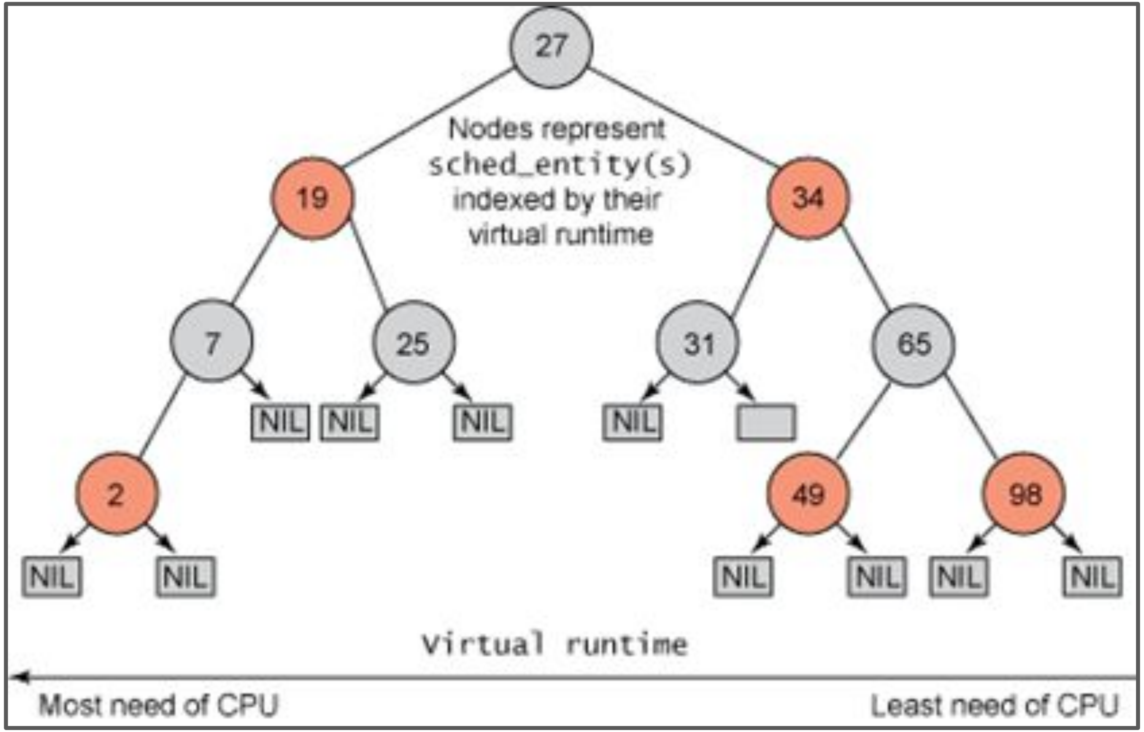
```
A: 6 quantum
B: 3 quantum
C: 2 quantum
```

CFS - Example

★ **Scheduler Implementation:** CFS does not work with a queue and instead maintains a time-ordered **red-black tree**.



CFS - Example



- ★ **O(1)** to maintain access to the left-most node.
- ★ **O(lg(n))** insert and delete operations
- ★

Completely Fair Scheduler (CFS)

One problem with picking the lowest vruntime to run next arises with jobs that have gone to sleep for a long period of time.

Example: Imagine two processes, A and B, one of which (A) runs continuously, and the other (B) which has gone to sleep for a long period of time (ex: 10 seconds). When B wakes up, its vruntime will be 10 seconds behind A's, and thus (if we're not careful), B will now monopolize the CPU for the next 10 seconds while it catches up, effectively starving A.

Scheduling Preemption

- Kernel sets the **need_resched** flag (per-process variable) at various locations
  - `scheduler_tick()`, a process used up its timeslice
  - `try_to_wake_up()`, higher-priority process awaken
- Kernel checks **need_resched** at certain points, if safe, **schedule()** will be invoked

- User preemption
  - Return to user space from a system call or an interrupt handler
- Kernel preemption
  - A task in the kernel explicitly calls schedule()
  - A task in the kernel blocks (which results in a call to schedule())