CS 423
Operating System Design: Scheduling in Linux

Professor Adam Bates
Spring 2017

Goals for Today

- Learning Objective:
  - Understand inner workings of modern OS schedulers
- Announcements, etc:
  - MP1 deadline is **Feb 19.**
  - MP2 is coming out on **Feb 21st**.
  - Important concepts for MP2 will be covered in class Friday.

Reminder: Please put away devices at the start of class

What Are Scheduling Goals?

- What are the goals of a scheduler?
- Linux Scheduler's Goals:
  - Generate illusion of concurrency
  - Maximize resource utilization (e.g., mix CPU and I/O bound processes appropriately)
  - 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
  - Support Real-Time (RT) applications

Early Linux Schedulers

- Linux 1.2: circular queue w/ round-robin policy.
  - Simple and minimal.
  - Did not meet many of the aforementioned goals

- Linux 2.2: 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
```

Why 2 RT mechanisms?

Two Fundamental Mechanisms…

- Prioritization
- Resource partitioning

Prioritization

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

Partitioning

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

Linux 2.4 Scheduler

- 2.4: O(N) scheduler.
  - 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.

Linux 2.6 Scheduler

- O(1) scheduler
- Tasks are indexed according to their priority [0,139]
  - Real-time [0, 99]
  - Non-real-time [100, 139]

SCHED_NORMAL

- Used for non real-time processes
- Complex heuristic to balance the needs of I/O and CPU centric applications
- Processes start at 120 by default
  - Static priority
    - 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

SCHED_NORMAL

- Used for non real-time processes
- Complex heuristic to balance the needs of I/O and CPU centric applications
- Processes start at 120 by default
    - S

Static Priority: Handles assigned task priorities

Dynamic Priority: Favors interactive tasks

Combined, these mechanisms govern CPU access in the SCHED_NORMAL scheduler.

    - 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

SCHED_NORMAL Heuristic

How does a static priority translate to real CPU access?

if (static priority < 120)
    Quantum = 20 (140 – static priority)

else
    Quantum = 5 (140 – static priority)

(in ms)

Higher priority → Larger quantum

SCHED_NORMAL Heuristic

How does a static priority translate to CPU access?

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

SCHED_NORMAL Heuristic

How does a dynamic priority adjust CPU access?

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.

Max priority # is still 139

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

Min priority # is still 100

(Bonus is subtracted to increase priority)

SCHED_NORMAL Heuristic

How does a dynamic priority adjust CPU access?

bo

What's the problem with this (or any) heuristic?

- Your bonus increases as you sleep more.

Max priority is still 100

dynamic priority =

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

Min priority is still 100

(Bonus is subtracted to increase priority)

Completely Fair Scheduler

- Merged into the 2.6.23 release of the Linux kernel and is the default scheduler.
- 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

Completely Fair Scheduler

- Merged into the 2.6.23 release of the Linux kernel and is the default scheduler
- Sc ... are
  or ... e
- Nc ... is
  pi

Property of CFS: If all task's virtual clocks run at exactly the same speed, they will all get the same amount of time on the CPU.

How does CFS account for I/O-intensive tasks?

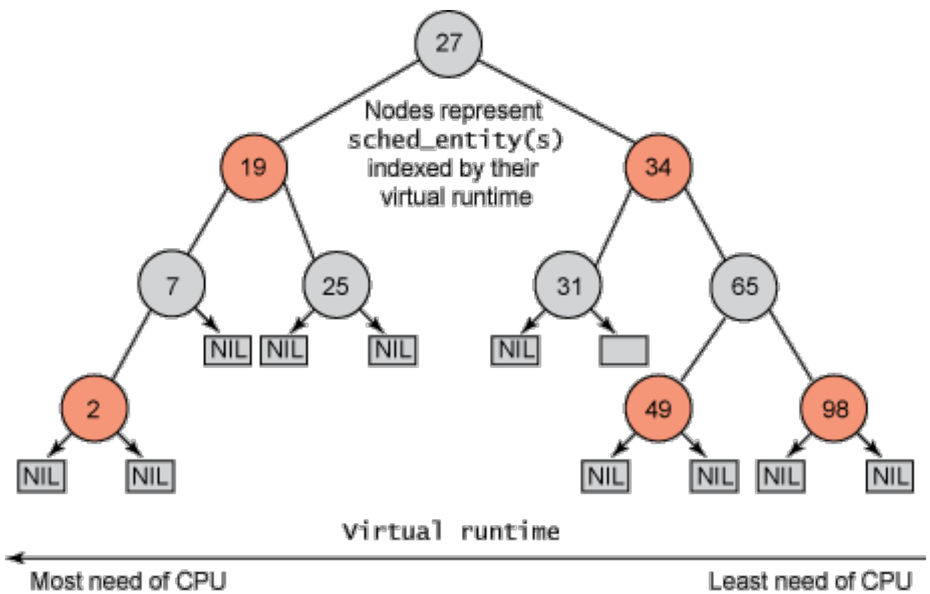
- Priorities determine accumulation rate of virtual execution time
  - Higher priority → slower accumulation rate

Example

- Three tasks accumulate virtual execution time at a rate of 1, 2, and 3, respectively.
- What is the expected share of the CPU that each gets?

Red-Black Trees

- CFS dispenses with a run queue and instead maintains a time-ordered **red-black tree**. Why?

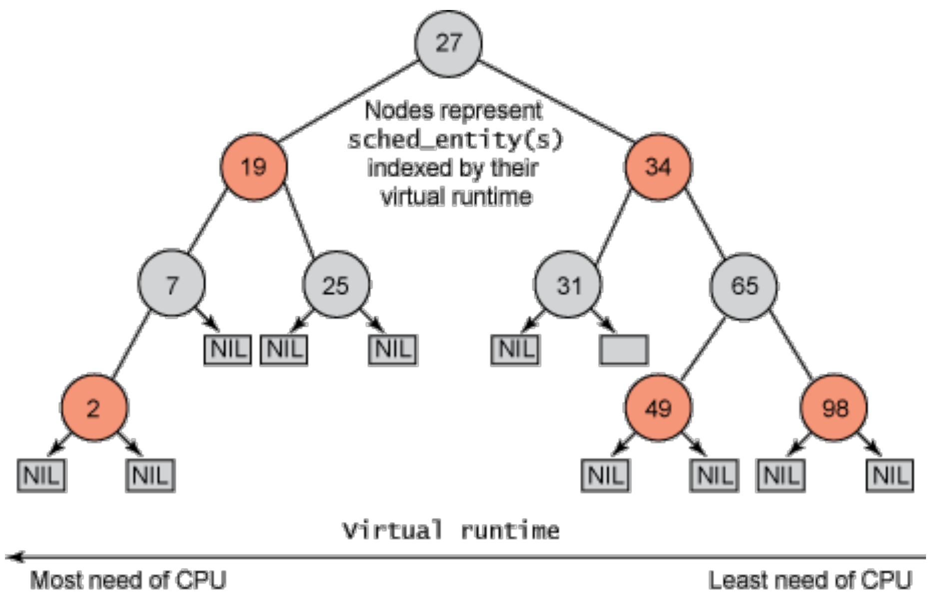


An RB tree is a BST w/ the constraints:

1. Each node is red or black
2. Root node is black
3. All leaves (NIL) are black
4. If node is red, both children are black
5. Every path from a given node to its descendent NIL leaves contains the same number of black nodes

Red-Black Trees

- CFS dispenses with a run queue and instead maintains a time-ordered **red-black tree**. Why?



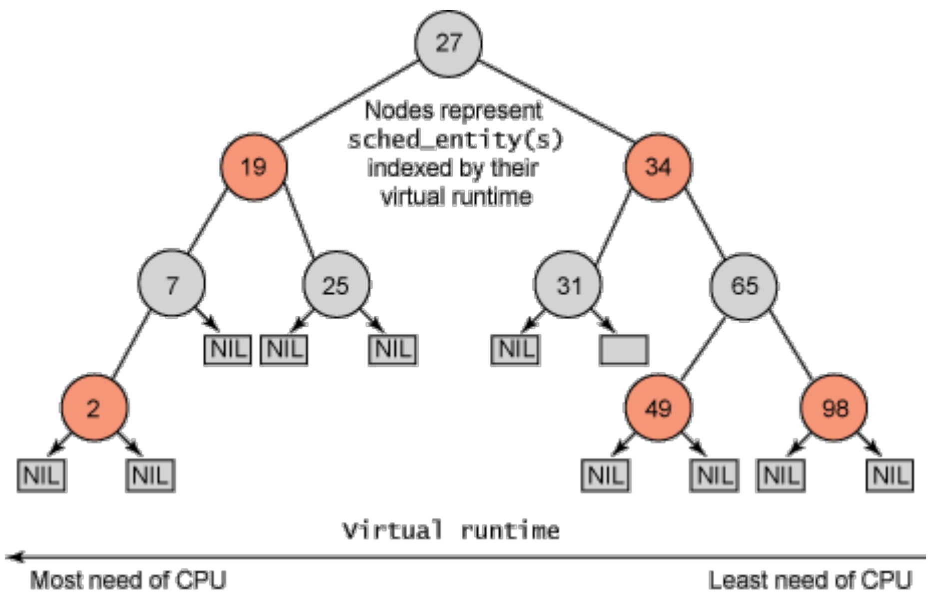
An RB tree is a BST w/ the constraints:

1. Each node is red or black
2. Root node is black
3. All leaves (NIL) are black
4. If node is red, both children are black
5. Every path from a given node to its descendent NIL leaves contains the same number of black nodes

Takeaway: In an RB Tree, the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.

Red-Black Trees

- CFS dispenses with a run queue and instead maintains a time-ordered **red-black tree**. Why?

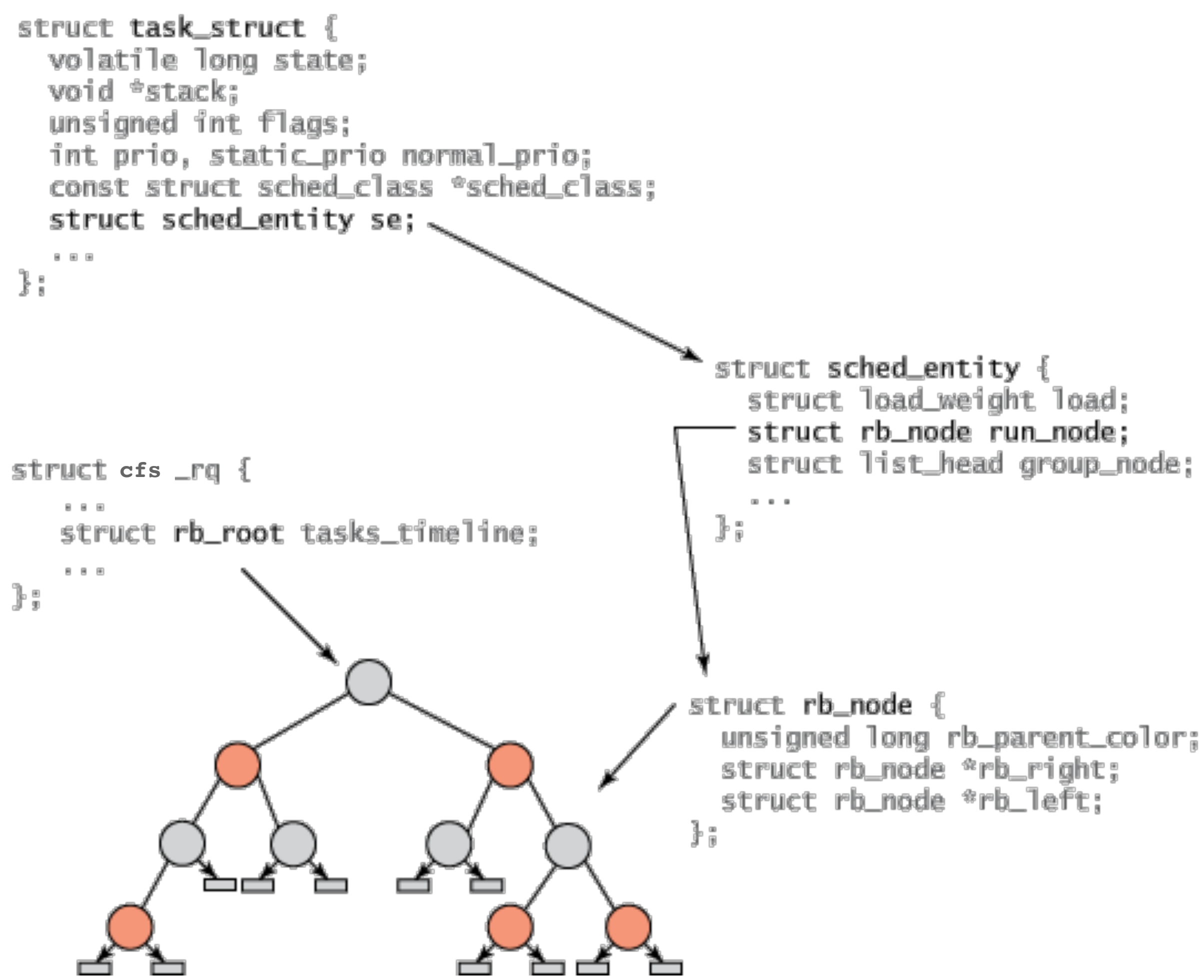


Benefits over run queue:

- O(1) access to leftmost node (lowest virtual time).
- O(log n) insert
- O(log n) delete
- self-balancing

RBT Structure Hierarchy

Like the kernel linked list (see MP1 Q&A), the data struct contains the node struct.



How/when to preempt?

- Kernel sets the need_resched flag (per-process var) 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())

Other scheduling policies

- What if you want to maximize throughput?

Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!

Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!
- What if you want to meet all deadlines?

Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!
- What if you want to meet all deadlines?
  - Earliest deadline first!
  - Problem?

Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!
- What if you want to meet all deadlines?
  - Earliest deadline first!
  - Problem?
  - Works only if you are not "overloaded". If the total amount of work is more than capacity, a domino effect occurs as you always choose the task with the nearest deadline (that you have the least chance of finishing by the deadline), so you may miss a lot of deadlines!

EDF Domino Effect

- Problem:
  - It is Monday. You have a homework due tomorrow (Tuesday), a homework due Wednesday, and a homework due Thursday
  - It takes on average 1.5 days to finish a homework.
- Question: What is your best (scheduling) policy?

EDF Domino Effect

- Problem:
  - It is Monday. You have a homework due tomorrow (Tuesday), a homework due Wednesday, and a homework due Thursday
  - It takes on average 1.5 days to finish a homework.
- Question: What is your best (scheduling) policy?
  - You could instead skip tomorrow's homework and work on the next two, finishing them by their deadlines
  - Note that EDF is bad: It always forces you to work on the next deadline, but you have only one day between deadlines which is not enough to finish a 1.5 day homework – you might not complete any of the three homeworks!