CS 423
Operating System Design: The Linux Scheduler

Professor Adam Bates
Spring 2017
Goals for Today

- **Learning Objective:**
  - Understand inner workings of modern OS schedulers

- **Announcements, etc:**
  - vSphere... can someone check and see if it's working yet?
  - MP1 deadline extended due to vSphere chaos... **Feb 20.**
  - Syscall Walkthrough slides are fixed.
  - Instructions for Special Topics sign-up coming out today

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).
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]
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
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
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
  - 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

Static Priority: Handles assigned task priorities
Dynamic Priority: Favors interactive tasks
Combined, these mechanisms govern CPU access in the SCHED_NORMAL scheduler.
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
### How does a static priority translate to CPU access?

<table>
<thead>
<tr>
<th>Description</th>
<th>Static priority</th>
<th>Nice value</th>
<th>Base time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest static priority</td>
<td>100</td>
<td>-20</td>
<td>800 ms</td>
</tr>
<tr>
<td>High static priority</td>
<td>110</td>
<td>-10</td>
<td>600 ms</td>
</tr>
<tr>
<td>Default static priority</td>
<td>120</td>
<td>0</td>
<td>100 ms</td>
</tr>
<tr>
<td>Low static priority</td>
<td>130</td>
<td>+10</td>
<td>50 ms</td>
</tr>
<tr>
<td>Lowest static priority</td>
<td>139</td>
<td>+19</td>
<td>5 ms</td>
</tr>
</tbody>
</table>
How does a dynamic priority adjust CPU access?

\[
\text{bonus} = \min \left( 10, \left( \frac{\text{avg. sleep time}}{100} \right) \text{ ms} \right)
\]

- \text{avg. sleep time is 0 } \Rightarrow \text{ bonus is 0}
- \text{avg. sleep time is 100 ms } \Rightarrow \text{ bonus is 1}
- \text{avg. sleep time is 1000 ms } \Rightarrow \text{ bonus is 10}
- \text{avg. sleep time is 1500 ms } \Rightarrow \text{ bonus is 10}
- \text{Your bonus increases as you sleep more.}

\[
\text{dynamic priority} = \max \left( 100, \min \left( \text{static priority} - \text{bonus} + 5, 139 \right) \right)
\]

\text{Min priority # is still 100} \quad \text{(Bonus is subtracted to increase priority)}

\text{Max priority # is still 139}
How does a dynamic priority adjust CPU access?

$$\text{bonus} = \min \left(10, \frac{\text{avg. sleep time}}{100}\right) \text{ ms}$$

- $\text{avg. sleep time is 0} \Rightarrow \text{bonus is 0}$
- $\text{avg. sleep time is 100 ms} \Rightarrow \text{bonus is 1}$
- $\text{avg. sleep time is 1000 ms} \Rightarrow \text{bonus is 10}$
- $\text{avg. sleep time is 1500 ms} \Rightarrow \text{bonus is 10}$

- Your bonus increases as you sleep more.

$$\text{dynamic priority} = \max (100, \min (\text{static priority} - \text{bonus} + 5, 139))$$

- $\text{Max priority is still 100}$
- $\text{Min priority is still 100}$
- (Bonus is subtracted to increase priority)

**SCHED_NORM**AL Heuristic
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.
- 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.

Property of CFS: If all tasks’ 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?
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?
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
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.
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
Like the kernel linked list (see MP1 Q&A), the data struct contains the node struct.

```c
struct task_struct {
    volatile long state;
    void *stack;
    unsigned int flags;
    int prio, static_prio normal_prio;
    const struct sched_class *sched_class;
    struct sched_entity se;
};
```

```c
struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
};
```

```c
struct cfs_rq {
    ...
    struct rb_root tasks_timeline;
};
```

```c
struct rb_node {
    unsigned long rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};
```
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!
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?
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!
How/when to preempt?

- Kernel sets the \texttt{need\_resched} flag (per-process var) at various locations
  - \texttt{scheduler\_tick()}, a process used up its timeslice
  - \texttt{try\_to\_wake\_up()}, higher-priority process awaken
- Kernel checks \texttt{need\_resched} at certain points, if safe, \texttt{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 \texttt{schedule()}
  - A task in the kernel blocks (which results in a call to \texttt{schedule()} )