CS 423
Operating System Design: Scheduling in Linux

Tianyin Xu (MIC)

* Thanks for Prof. Adam Bates for the slides.
Midterm time is changed

• 3/12 -- right before the Spring Break
  • I hope I will be physically proctoring!
• I haven’t decided the format yet.
  • May end up doing the standard (boring) paper tests
  • Let me know if you have good suggestions!
• Everything before 3/12 would be covered
  • Yes, memory management!
“CPU scheduling is not planning; there is not an optimal solution. Rather CPU scheduling is about balancing goals and making difficult tradeoffs.”

-- Joseph T. Meehean
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
## Multi-Level Feedback Queue

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time Slice (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
</tr>
</tbody>
</table>

Round Robin Queues:
- New or I/O Bound Task
- Time Slice Expiration
Why is MLFQ 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?

• Yes, SJF – the assumption is to know which is the “shortest..”
  • It’s just very hard to know in advance.
  • Sometimes processes/threads could try to game (we will see an example).
Why is MLFQ a good design?

• The Key Idea
  • Dynamically adjusting the priority level based on observing the behavior of the processes/threads

• Basic Design
  • When a job enters the system, it is placed at the highest priority (the topmost queue).
  • If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  • If a job gives up the CPU before the time slice is up, it stays at the same priority level.
Why is MLFQ a good design?

- **The Key Idea**
  - Dynamically adjusting the priority level based on observing the behavior of the processes/threads

- **Basic Design**
  - When a job enters the system, it is placed at the highest priority (the topmost queue).
  - If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.
because it doesn’t know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process.
Starvation?

- Jack has a way to game the scheduler!
Starvation?

- Jack has a way to game the scheduler!
Priority Boost

- After some time period $S$, move all the jobs in the system to the topmost queue
• 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).
Sounds perfect?

- 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?
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).

```c
/* 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
```
Two Fundamental Mechanisms...

- Prioritization
- Resource partitioning
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
  - 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?

\[
\begin{align*}
\text{if } & (\text{static priority} < 120) \\
\text{Quantum} & = 20 \ (140 - \text{static priority})
\end{align*}
\]

\[
\text{else} \\
\text{Quantum} & = 5 \ (140 - \text{static priority})
\]

(in ms)

Higher priority $\rightarrow$ 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?

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))

Min priority # is still 100
(Bonus is subtracted to increase priority)

Max priority # is still 139
How does a dynamic priority adjust CPU access?

- Your bonus increases as you sleep more.

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

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

Max priority is still 100

Min priority is still 100

(Bonus is subtracted to increase priority)
Completely Fair Scheduler

- **Goal:** Fairly divide a CPU evenly among all competing processes with a clean implementation
- Merged into the 2.6.23 release of the Linux kernel and is the default scheduler.
- Created by Ingo Molnar in a short burst of creativity which led to a 100K kernel patch developed in 62 hours.

**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.
- It is a “catch up” policy — task with smallest amount of virtual time gets to run next.
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

- 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 \(\rightarrow\) slower accumulation rate.

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?
Example

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

Strategy: **How many quantums required for all clocks to be equal?**
- Least common multiple is 6
- To reach VT=6...
  - A is scheduled 6 times
  - B is scheduled 3 times
  - C is scheduled 2 times.
- $6+3+2 = 11$
- A => 6/11 of CPU time
- B => 3/11 of CPU time
- C => 2/11 of CPU time

<table>
<thead>
<tr>
<th>Q</th>
<th>Outcome</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q01</td>
<td>A =&gt;</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q02</td>
<td>B =&gt;</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Q03</td>
<td>C =&gt;</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Q04</td>
<td>A =&gt;</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Q05</td>
<td>B =&gt;</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Q06</td>
<td>A =&gt;</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Q07</td>
<td>A =&gt;</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Q08</td>
<td>C =&gt;</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Q09</td>
<td>A =&gt;</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Q10</td>
<td>B =&gt;</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q11</td>
<td>A =&gt;</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
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
One problem with picking the lowest vruntime to run next arises with jobs that have gone to sleep for a long period of time. 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 (say, 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.

What’s the solution? 😊
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() )
A Note on CPU Affinity

We’ve had lots of great (abstraction-violating) questions about how multiprocessor scheduling works in practice…

• To answer, consider *CPU Affinity* — scheduling a process to stay on the same CPU as long as possible
  
  • Benefits?
  
  • Soft Affinity — Natural occurs through efficient scheduling
    
    • Present in O(1) onward, absent in O(N)
  
  • Hard Affinity — Explicit request to scheduler made through system calls (Linux 2.5+)
Multi-Processor Scheduling

• CPU affinity would seem to necessitate a **multi-queue** approach to scheduling... but how?

• **Asymmetric Multiprocessing (AMP):** One processor (e.g., CPU 0) handles all scheduling decisions and I/O processing, other processes execute only user code.

• **Symmetric Multiprocessing (SMP):** Each processor is self-scheduling. Could work with a single queue, but also works with private queues.
  
  • Potential problems?
SMP Load Balancing

• SMP systems require load balancing to keep the workload evenly distributed across all processors.

• Two general approaches:
  
  • **Push Migration**: Task routinely checks the load on each processor and redistributes tasks between processors if imbalance is detected.
  
  • **Pull Migration**: Idle processor can actively pull waiting tasks from a busy processor.
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!