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)

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

Thread 1:

Thread 2:

Thread A:

Thread B:
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

Thread 1:

Thread 2:

Thread A:

Thread B:

Thread A Downgraded to P2

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?
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

```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
```
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
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

/* 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

- **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

<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>
<tr>
<td>PID</td>
<td>USER</td>
<td>PR</td>
<td>NI</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>----</td>
<td>-----</td>
</tr>
<tr>
<td>1559</td>
<td>apache</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>1542</td>
<td>apache</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>1318</td>
<td>apache</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>1487</td>
<td>apache</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>1552</td>
<td>apache</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>root</td>
<td>0 -20</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>root</td>
<td>0 -20</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>root</td>
<td>rt</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>root</td>
<td>rt</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>root</td>
<td>rt</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>root</td>
<td>rt</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>root</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>root</td>
<td>0 -20</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>root</td>
<td>0 -20</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

91-divoc static web server
<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>19314</td>
<td>yikait2</td>
<td>20</td>
<td>0</td>
<td>1024332</td>
<td>45260</td>
<td>17212</td>
<td>S</td>
<td>21.8</td>
<td>0.0</td>
<td>0:04:61</td>
<td>caja</td>
</tr>
<tr>
<td>19205</td>
<td>yikait2</td>
<td>20</td>
<td>0</td>
<td>1614848</td>
<td>120616</td>
<td>9976</td>
<td>S</td>
<td>8.9</td>
<td>0.1</td>
<td>0:07:60</td>
<td>Xorg</td>
</tr>
<tr>
<td>19842</td>
<td>setroub+</td>
<td>20</td>
<td>0</td>
<td>386100</td>
<td>69152</td>
<td>11732</td>
<td>S</td>
<td>3.3</td>
<td>0.1</td>
<td>0:01:78</td>
<td>setroubleshootd</td>
</tr>
<tr>
<td>1641</td>
<td>fastx</td>
<td>20</td>
<td>0</td>
<td>1394068</td>
<td>167020</td>
<td>13684</td>
<td>S</td>
<td>2.0</td>
<td>0.1</td>
<td>750:05:04</td>
<td>node</td>
</tr>
<tr>
<td>9</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>1.0</td>
<td>0.0</td>
<td>147:30:96</td>
<td>rcu_sched</td>
</tr>
<tr>
<td>20225</td>
<td>waf</td>
<td>20</td>
<td>0</td>
<td>164476</td>
<td>2612</td>
<td>1596</td>
<td>R</td>
<td>1.0</td>
<td>0.0</td>
<td>0:00:27</td>
<td>top</td>
</tr>
<tr>
<td>48902</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>1607636</td>
<td>79088</td>
<td>6472</td>
<td>S</td>
<td>0.7</td>
<td>0.1</td>
<td>19:52:92</td>
<td>salt-minion</td>
</tr>
<tr>
<td>41</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>23:05:85</td>
<td>ksoftirqd/6</td>
</tr>
<tr>
<td>69</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>2:17:31</td>
<td>watchdog/12</td>
</tr>
<tr>
<td>708</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>278464</td>
<td>132776</td>
<td>131612</td>
<td>S</td>
<td>0.3</td>
<td>0.1</td>
<td>4:56:77</td>
<td>systemd-journal</td>
</tr>
<tr>
<td>1009</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>12:09:11</td>
<td>xfsaid/dm-5</td>
</tr>
<tr>
<td>1632</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>584156</td>
<td>22028</td>
<td>6692</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>2:48:24</td>
<td>tuned</td>
</tr>
<tr>
<td>1667</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>966076</td>
<td>91396</td>
<td>4896</td>
<td>S</td>
<td>0.3</td>
<td>0.1</td>
<td>102:22:28</td>
<td>f2b/server</td>
</tr>
<tr>
<td>15832</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>0:00:02</td>
<td>kworker/5:1</td>
</tr>
<tr>
<td>19204</td>
<td>yikait2</td>
<td>20</td>
<td>0</td>
<td>161412</td>
<td>10788</td>
<td>1492</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>0:00:12</td>
<td>FastX monitor 1</td>
</tr>
<tr>
<td>19859</td>
<td>waf</td>
<td>20</td>
<td>0</td>
<td>169568</td>
<td>2704</td>
<td>1244</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>0:00:03</td>
<td>sshd</td>
</tr>
<tr>
<td>46084</td>
<td>bjzhang2</td>
<td>20</td>
<td>0</td>
<td>1090328</td>
<td>61924</td>
<td>16540</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>0:25:44</td>
<td>node</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>194340</td>
<td>7404</td>
<td>4232</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>19:09:92</td>
<td>systemd</td>
</tr>
<tr>
<td>2</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:08:38</td>
<td>kthreadd</td>
</tr>
<tr>
<td>4</td>
<td>root</td>
<td>0</td>
<td>-20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:00</td>
<td>kworker/0:0H</td>
</tr>
<tr>
<td>5</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:33:07</td>
<td>ksoftirqd/9</td>
</tr>
</tbody>
</table>

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

**NI: “Nice” Value**
- [-20, 19] == niceness
### Static Priority CPU Translation

<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>
Dynamic Priority

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

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

\[
\text{dynamic priority} = \\
\max (100, \min (\text{static priority} - \text{bonus} + 5, 139))
\]
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 $\Rightarrow$ 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\}
\[
\begin{align*}
Q00: & \quad - \Rightarrow \{A:0, B:0, C:0\} & \text{A: 6 quantum} \\
Q01: & \quad A \Rightarrow \{A:1, B:0, C:0\} \quad \text{B: 3 quantum} \\
Q02: & \quad B \Rightarrow \{A:1, B:2, C:0\} \quad \text{C: 2 quantum} \\
Q03: & \quad C \Rightarrow \{A:1, B:2, C:3\} \\
Q04: & \quad A \Rightarrow \{A:2, B:2, C:3\} \\
Q05: & \quad B \Rightarrow \{A:2, B:4, C:3\} \\
Q06: & \quad A \Rightarrow \{A:3, B:4, C:3\} \\
Q07: & \quad A \Rightarrow \{A:4, B:4, C:3\} \\
Q08: & \quad C \Rightarrow \{A:4, B:4, C:6\} \\
Q09: & \quad A \Rightarrow \{A:5, B:4, C:6\} \\
Q10: & \quad B \Rightarrow \{A:5, B:6, C:6\} \\
Q11: & \quad A \Rightarrow \{A:6, B:6, C:6\}
\end{align*}
\]
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.
- \( O(\log(n)) \) insert and delete operations.

Nodes represent `sched_entity(s)` indexed by their virtual runtime.

Virtual runtime: Most need of CPU to Least need of CPU.
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()` )