CPU Inheritance Scheduling

Bryan Ford    Sai Susarla

Computer Systems Laboratory
Department of Computer Science
University of Utah

flux@cs.utah.edu
http://www.cs.utah.edu/projects/flux/

October 30, 1996
Key Concepts

Threads schedule *each other* by donating the CPU using a *directed yield* primitive.

One *root scheduler thread* per processor sources all CPU time.

Kernel *dispatcher* manages threads, events, and CPU donation without making any scheduling policy decisions.
The Dispatcher

Implements thread sleep, wakeup, schedule, etc.

Runs in the context of currently running thread.

Has no notion of thread priority, CPU usage, clocks, or timers.

Dispatcher wakes a scheduler thread when:

- Scheduler’s client blocks.

- Event of interest to the scheduler occurs.
Scheduling Example

Scheduler

CPU
donation

Running
thread

Scheduler
thread

Ready
queues

scheduling
requests

Port

Waiting thread

App 1

App 2

Ready
threads
The schedule() operation

schedule(thread, port, sensitivity)

Sensitivity levels:

- **ON_BLOCK**: Wake the scheduler *any* time its client thread blocks.

- **ON_SWITCH**: Wake the scheduler only when a different client is requesting the CPU.

- **ON_CONFLICT**: Wake the scheduler only when *two or more* clients are runnable at the same time.
Implicit Donation

Works like `schedule()`, except done implicitly; e.g.:

- Thread attempting to lock a held mutex donates to current owner
- Client thread donates to server thread for the duration of an RPC

Analogous to priority inheritance in traditional systems.
Multiprocessor Scheduling

Scheduler

CPU 0

CPU 1

Ready queues

Scheduler threads

App 1

App 2
Benefits

- Hierarchical, stackable scheduling policies
- Application-specific scheduling policies
- Modular CPU usage control
- Automatic priority inheritance
- Accurate CPU usage accounting
- Naturally extends to multiprocessors
- Supports processor affinity policies and scheduler activations
Prototype Implementation

Implemented as a fancy threads package in a BSD process.

Schedulers implemented:

- Fixed priority round-robin and FIFO
- Rate monotonic
- Lottery
Scheduling Hierarchy

Root Scheduler
Fixed-priority

Real-time Scheduler
Rate-monotonic

Timesharing Class
Lottery scheduling

Web browser
Lottery scheduling

Background
Round-robin

FIFO Scheduler
Non-preemptive

Java applet threads

Cooperating threads

RR1
RR2
RM1
RM2
LS1
JAVA1
JAVA2
FIFO1
FIFO2
Results

Three measures:

- Scheduling behavior (correctness)
- Overhead
- Implementation complexity
Multi-policy Scheduling Behavior

- Rate-monotonic thread 1 (50%)
- Rate monotonic thread 2 (25%)
- Lottery thread (Interactive - bursty)
- Round-robin thread 1 (Insatiable)
- Round-robin thread 2 (Insatiable)
Modular Control of CPU Usage

- Applet thread 1
- Applet thread 2
- FIFO thread 1
- FIFO thread 2
- Round-robin thread 1
- Round-robin thread 2

Relative CPU time allocation (percent) vs. Time (clock ticks)
Real-time Scheduling Behavior

CPU donation on mutex contention
No CPU donation

Number of occurrences

Mutex lock latency for real-time thread (clock ticks)
Performance

• Dispatcher overhead
  – Base cost
  – Sensitivity to hierarchy depth

• Context switching overhead
  – Number of additional context switches
  – Cost of context switches
Dispatcher Micro-benchmarks

<table>
<thead>
<tr>
<th>Scheduling Hierarchy Depth</th>
<th>Dispatch Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root scheduler only</td>
<td>8.0</td>
</tr>
<tr>
<td>2-level scheduling</td>
<td>11.2</td>
</tr>
<tr>
<td>3-level scheduling</td>
<td>14.0</td>
</tr>
<tr>
<td>4-level scheduling</td>
<td>16.2</td>
</tr>
<tr>
<td>8-level scheduling</td>
<td>24.4</td>
</tr>
</tbody>
</table>
Context switch overhead

• In prototype, measure what proportion of context switches are to scheduler threads (i.e., extra)

• On a real OS, measure rate of context switches in various work loads

• Project slowdown in two OSs, based on expected rate and speed of context switches
## Context Switches for Simple Tests

<table>
<thead>
<tr>
<th></th>
<th>Client/Server</th>
<th>Parallel Database</th>
<th>Real-time</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM1</td>
<td>57</td>
<td></td>
<td>322</td>
<td>101</td>
</tr>
<tr>
<td>RM2</td>
<td>19</td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>RM3</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1</td>
<td>25</td>
<td></td>
<td>622</td>
<td>17</td>
</tr>
<tr>
<td>JAVA1</td>
<td>46</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO1</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR1</td>
<td>114</td>
<td>238</td>
<td>249</td>
<td>7</td>
</tr>
<tr>
<td>RR2</td>
<td>3</td>
<td>242</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>RR3</td>
<td></td>
<td>234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR4</td>
<td></td>
<td>243</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User invocations</td>
<td>492</td>
<td>957</td>
<td>1193</td>
<td>165</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Client/Server</th>
<th>Parallel Database</th>
<th>Real-time</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root scheduler</td>
<td>262</td>
<td>956</td>
<td>1237</td>
<td>142</td>
</tr>
<tr>
<td>Rate monotonic</td>
<td>43</td>
<td></td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>Lottery scheduler</td>
<td>30</td>
<td></td>
<td>57</td>
<td>3</td>
</tr>
<tr>
<td>Applet scheduler</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO scheduler</td>
<td>1</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Round-robin sched</td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Scheduler invoc.</td>
<td>346</td>
<td>956</td>
<td>1303</td>
<td>218</td>
</tr>
<tr>
<td>Total csw</td>
<td>838</td>
<td>1913</td>
<td>2496</td>
<td>383</td>
</tr>
<tr>
<td>Scheduler %</td>
<td>41%</td>
<td>50%</td>
<td>52%</td>
<td>56%</td>
</tr>
</tbody>
</table>
## Statistics for Common Applications

<table>
<thead>
<tr>
<th></th>
<th>gzip</th>
<th>gcc</th>
<th>tar</th>
<th>configure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time (sec)</td>
<td>26.4</td>
<td>35.3</td>
<td>9.6</td>
<td>26.0</td>
</tr>
<tr>
<td>Context switches/sec</td>
<td>11</td>
<td>32</td>
<td>81</td>
<td>202</td>
</tr>
<tr>
<td>Traps/sec</td>
<td>10</td>
<td>562</td>
<td>22</td>
<td>3470</td>
</tr>
<tr>
<td>System calls/sec</td>
<td>23</td>
<td>651</td>
<td>517</td>
<td>1807</td>
</tr>
<tr>
<td>Device interrupts/sec</td>
<td>427</td>
<td>509</td>
<td>3337</td>
<td>1055</td>
</tr>
</tbody>
</table>
Code Complexity

- **Dispatcher:**
  550 raw, 160 lines of semicolons

- **Example schedulers:**
  each is 100–200 semicolons
Related Work

Existing multi-policy systems:

- Multi-class systems: Mach, NT
- Aegis Exokernel
Related Work

Existing hierarchical scheduling policies:

- KeyKOS meters
- Lottery/stride scheduling
- Start-time Fair Queuing (SFQ)

CPU inheritance scheduling is not a policy.
Status

Works, but needs to be tried in a real OS

Fluke kernel implementation in progress

Source for prototype will be available from the OSDI and Flux project web pages:
http://www.cs.utah.edu/projects/flux/
Conclusion

CPU inheritance scheduling:

- Provides flexible CPU scheduling, and supports many existing policies and mechanisms

- Is efficient enough for common uses

- Is straightforward to implement (in user mode)

- Supports the Fluke nested process model