Kernel Thread Scheduling in Real-Time Linux for Wearable Computers

Dongwook Kang†, Woojoong Lee‡, and Chanik Park‡
†Department of Computer Science & Engineering, POSTECH, S. Korea
‡Embedded Software Research Division, ETRI, Daejeon, S. Korea.
‡Department of Computer Science & Engineering, POSTECH, S. Korea

ETRI Journal, June 2007
Outline

- Introduction
- Scheduling Problems of Kernel Threads
- Dynamic Kernel Thread Scheduling for Real-Time Systems
- Experimental Evaluation
  - Implementation in Linux 2.6
  - Performance Evaluation
- Conclusion
Wearable computers select Linux
- reliability, security, and flexibility
- restricted real-time support capability
  - Ingo Molnar’s real-time preemption
    - O(1) and Completely Fair Scheduler
  - the threaded interrupt
    - handling in the process context
    - assign priorities to each interrupt statically
    - priority inversion problem
Introduction

Fig. 1. The problem of IRQ thread of RT-preempt patch.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Name</th>
<th>Default priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct group</td>
<td>IRQ thread</td>
<td>Real-time 40–50</td>
</tr>
<tr>
<td></td>
<td>ksoftirqd</td>
<td>Real-time 1</td>
</tr>
<tr>
<td>Indirect group</td>
<td>pdflush</td>
<td>Nice 0</td>
</tr>
<tr>
<td></td>
<td>kswapd</td>
<td>Nice 0</td>
</tr>
<tr>
<td>System kernel thread group</td>
<td>keventd</td>
<td>Nice -5</td>
</tr>
<tr>
<td>Worker kernel thread group</td>
<td>aio</td>
<td>Nice -5</td>
</tr>
<tr>
<td></td>
<td>kthread</td>
<td>Nice -5</td>
</tr>
<tr>
<td></td>
<td>khelper</td>
<td>Nice -5</td>
</tr>
</tbody>
</table>

Table 1. Classification of kernel threads.
Introduction

- **Direct or indirect group**: the response time of real-time tasks is affected by the execution delay of the kernel thread

- **System kernel thread group**: The kernel threads invoked when the current system state exceeds a system threshold configured for optimal throughputs

- **Worker kernel threads group**: the kernel threads serving work queues Do job asynchronously with real-time tasks
Scheduling Problems of Kernel Threads

- IRQ Thread and ksoftirqd
  - top-halves
    - of interrupts are handled in the interrupt context with a higher priority than all tasks
  - bottom-halves
    - are handled by a kernel thread, ksoftirqd

- With Ingo Molnar's real-time preemption patch
  - top-halves of interrupts are handled by an IRQ thread
  - allow real-time tasks to preempt interrupt handlers
  - priority inversion problem
Scheduling Problems of Kernel Threads

- **Pdflush**
  - writes back dirty pages to disks to control the dirty ratio of the system
  - If resources are below the threshold…
    - deadly chain

- **kswapd**
  - swaps out the least recently used pages to maintain a number of free pages
  - If starvation…
Scheduling Problems of Kernel Threads

- Worker Kernel Threads
  - based on work queues
    - the normal tasks insert work structures to the worker kernel threads
    - the worker kernel thread sequentially handles the work structures
  - Execution delays affect the performance of real-time tasks
    - When pdflush or a normal task tries to create an additional pdflush and if the kthread is starved
Dynamic Kernel Thread Scheduling for Real-Time Systems

- For solving the problems with pdflush..., etc.
  - a dynamic kernel thread scheduling algorithm
  - weighted average priority inheritance protocol (PIP)

1. \[ \text{prio}(k_i) = \begin{cases} 
\text{default\_prio}(k_i) & \text{if } |R_i| = 0, \\
\text{max\_prio}(R_i) & \text{if } |R_i| \neq 0.
\end{cases} \] (1)

2. \[ \text{prio}(k_i) = \begin{cases} 
\text{default\_prio}(k_i) & \text{if } |R_i| = 0, \\
\min \left( \sum_{r_j \in R_i} \text{prio}(r_j), \frac{\sum_{r_j \in R_i} \text{prio}(r_j)}{|R_i|}, \text{max\_prio}(R_i) \right) & \text{if } |R_i| \neq 0.
\end{cases} \] (2)
Dynamic Kernel Thread Scheduling for Real-Time Systems

- IRQ thread and ksoftirqd
  - A real-time task has a relation with them
    - since it requests an I/O job until the request is fulfilled
  - Whenever a relation is created or terminated,
    - recalculated using (2)

\[
prio(k_i) = \begin{cases} 
  \text{default\_prio}(k_i) & \text{if } |R_i| = 0, \\
  \min \left( w_i \cdot \frac{\sum_{r_j \in R_i} \text{prio}(r_j)}{|R_i|}, \max\_\text{prio}(R_i) \right) & \text{if } |R_i| \neq 0.
\end{cases}
\]

(2)
Dynamic Kernel Thread Scheduling for Real-Time Systems

- pdflush
  - a real-time task is maintained in $R_{pdflush}$ during which the dirty pages written by the real-time task exist in the page cache
  - apply the relative importance
    - if a real-time task opens and writes to three different files, then the task is inserted to $R_{pdflush}$ three times
  - This mechanism is efficient
    - because the dirty pages are managed in the inode unit and are written back in the inode unit as well
Dynamic Kernel Thread Scheduling for Real-Time Systems

- **kswapd**
  - real-time tasks whose pages are in the LRU cache are the elements of $R_{kswapd}$
  - **complex**:
    - too many pages in LRU cache
    - A pages may be used by multiple user
    - assume that each real-time task has the same number of pages in the LRU cache

- **rt_page_ratio**
  - the ratio of LRU cache pages used by all real-time tasks
Dynamic Kernel Thread Scheduling for Real-Time Systems

- **Kswapd**
  
  \[
  \text{prio}(k_{\text{kswapd}}) = \begin{cases} 
  \text{default}\_\text{prio}(k_{\text{kswapd}}) & \text{if } |R_{\text{kswapd}}| = 0, \\
  \min \left( \frac{w_{\text{kswapd}} \sum_{r_j \in R_{\text{kswapd}}} \text{prio}(r_j)}{|R_{\text{kswapd}}|} \cdot rt\_\text{page}\_\text{ratio}, \right) & \text{if } |R_{\text{kswapd}}| \neq 0. 
  \end{cases}
  \]

- **Worker Kernel Threads**
  
  - real-time tasks related to a worker kernel thread \( R_{\text{worker}} \)
    - begins when the real-time tasks insert a work structure to the work queue
    - the relation ends when handling of the work structure is completed
    - In both conditions, priority need to be recalculated
Experimental Evaluation

- Implementation in Linux 2.6
  - For recalculation, some information is maintained in each kernel thread
    - the default priority of ki, Ri, current average and maximum priority of Ri, and the number of real-time tasks in Ri …
    - as **member variables in TCB**
  - The average priority can be calculated from
    - the previous average priority
    - the number of tasks in Ri,
    - the priority of the real-time tasks newly added to or removed from Ri
Experimental Evaluation

- IRQ Thread and ksoftirqd
  - add the real-time task to Rirq right before this task adds the I/O job to the I/O request queue and is blocked, and we remove it right after the I/O job is finished and the task is woken up

- pdflush
  - Some variables are added to the inode structure
    - the number of related real-time tasks
    - the average priority
Experimental Evaluation

- **kswapd**
  - rt_page_ratio
    - each page descriptor
      - has the information of *which kinds of tasks* are using the corresponding page
      - use a *redundant bit of the flags* member variable of the page descriptor

- **Worker Kernel Threads**
  - A member variable is added to the work structure
    - the priority of a real-time task
Experimental Evaluation

- Performance Evaluation
  - OS
    - Linux 2.6.15 which was applied with the *Ingo Molnar's real-time preemption patch* of version rt16.
    - the Ubuntu 5.10 Linux distribution with NPTL 2.3.5. The run level was set to 3.
  - machine
    - 2 GHz Intel Pentium IV CPU
    - 256 MB DDR SDRAM
    - IBM IC35L080AVVA07-0 IDE disk
  - weight values: IRQ thread 1.2, otherwise 0.8
  - trial-and-error method
Experimental Evaluation

- IRQ Thread and ksoftirqd

```c
sleep_usec = 0; tsc1 = gettsc();
for (i = 0; i < sampling; i++) {
    tsc1 += (sleep_usec * CPU_CLOCK);
    memcpy(buf2, buf1, MEMCPY_LEN * 1024);
    write(fd, buf2, WRITE_LEN);
    tsc2 = gettsc();
    sleep_usec =
        WRITE_PERIOD - (tsc2 - tsc1) / CPU_CLOCK;
    tsc1 = gettsc();
    if (sleep_usec > 0) usleep(sleep_usec);
    else sleep_usec = 0;
}
```

Fig. 2. Essential code snippet of the disk write task used for experiments with IRQ threads.

Fig. 3. Response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.15).

Fig. 4. Response time of the disk write task on Linux with RT-preempt patch and weighted average PIP (Linux 2.6.15).
Experimental Evaluation

- IRQ Thread and ksoftirqd

Fig. 5. Response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.18).

Fig. 6. Cumulative rate of response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.15).
Experimental Evaluation

- pdflush

```
sleep_usec = 0; tsc1 = gettsc();
for (i = 0; i < sampling; i++) {
    tsc1 += (sleep_usec * CPU_CLOCK);
    memcpy(buf2, buf1, MEMCPY_LEN * 1024);
    write(fd, buf2, WRITE_LEN);
    tsc2 = gettsc();
    sleep_usec =
        WRITE_PERIOD - (tsc2 - tsc1) / CPU_CLOCK;
    tsc1 = gettsc();
    if (sleep_usec > 0) usleep(sleep_usec);
    else sleep_usec = 0;
}
```

Fig. 7. Essential code snippet of the disk write task used for experiments with pdflush.

Fig. 8. Architecture of the FPGA module.

Fig. 9. Response time of the disk write task executed with a CPU-bound task on Linux with RT-preempt patch and weighted average PIP.
Experimental Evaluation

- pdflush

Fig. 10. Dirty ratio with and without applying weighted average PIP.

Fig. 11. Cumulative rate of response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.15).
Experimental Evaluation

- kswapd

```c
sleep_usec = 0; tsc1 = gettsc();
for (i = 0; i < sampling; i++) {
    tsc1 += (sleep_usec * CPU_CLOCK);
    memcpy(buf2, buf1, MEMCPY_LEN * 1024);
    offset = rand() % READ_FILE_SIZE;
    lseek(fd, offset, SEEK_SET);
    read(fd, buf, READ_LEN);
    tsc2 = gettsc();
    sleep_usec =
        WRITE_PERIOD - (tsc2 - tsc1) / CPU_CLOCK;
    tsc1 = gettsc();
    if (sleep_usec > 0) usleep(sleep_usec);
    else sleep_usec = 0;
}
```

Fig. 12. Essential code snippet of the disk read task used for experiments about kswapd.

Fig. 13. Response time of disk read task executed with a CPU-bound task on Linux with RT-preempt patch.

Fig. 14. Response time of disk read task executed with a CPU-bound task on Linux with RT-preempt patch and weighted average PIP.
Experimental Evaluation

- **kswapd**

Fig. 15. The number of free pages with and without applying weighted average PIP to kswapd.

Fig. 16. Cumulative rate of response time of the disk readtask on Linux with RT-preempt patch (Linux 2.6.15).
Experimental Evaluation

- Worker Kernel Threads

Fig. 17. Response time of the disk write task executed with kthread on Linux with RT-preempt patch.

Fig. 18. Response time of the disk write task executed with kthread on Linux with RT-preempt patch and weighted average PIP.
Experimental Evaluation

- Lmbench

Table 2. Latencies for processor/process activities.

<table>
<thead>
<tr>
<th></th>
<th>null call</th>
<th>null I/O</th>
<th>stat</th>
<th>opens</th>
<th>select TCP</th>
<th>sig inst</th>
<th>sig send</th>
<th>fork proc</th>
<th>exec proc</th>
<th>sh proc</th>
<th>(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>0.22</td>
<td>0.44</td>
<td>2.81</td>
<td>4.23</td>
<td>21</td>
<td>0.91</td>
<td>3.17</td>
<td>142</td>
<td>624</td>
<td>7906</td>
<td></td>
</tr>
<tr>
<td>dkts</td>
<td>0.22</td>
<td>0.44</td>
<td>2.82</td>
<td>4.24</td>
<td>21</td>
<td>0.91</td>
<td>3.19</td>
<td>148</td>
<td>627</td>
<td>7964</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Latencies for context switches.

<table>
<thead>
<tr>
<th></th>
<th>2p/0k</th>
<th>2p/16k</th>
<th>2p/64k</th>
<th>8p/16k</th>
<th>8p/64k</th>
<th>16p/16k</th>
<th>(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>2.38</td>
<td>3.58</td>
<td>6.66</td>
<td>4.43</td>
<td>32.3</td>
<td>9.74</td>
<td></td>
</tr>
<tr>
<td>dkts</td>
<td>2.39</td>
<td>3.55</td>
<td>6.65</td>
<td>4.43</td>
<td>31.9</td>
<td>8.84</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Latencies for file accesses and VM.

<table>
<thead>
<tr>
<th></th>
<th>0k file create</th>
<th>0k file delete</th>
<th>10k file create</th>
<th>10k file delete</th>
<th>mmap latency</th>
<th>prot fault</th>
<th>page fault</th>
<th>(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>28.9</td>
<td>12.4</td>
<td>85.4</td>
<td>31.2</td>
<td>980.0</td>
<td>1.312</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>dkts</td>
<td>29.1</td>
<td>12.4</td>
<td>90.8</td>
<td>31.2</td>
<td>983.0</td>
<td>1.309</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Latencies for local communications.

<table>
<thead>
<tr>
<th></th>
<th>2p/0k</th>
<th>pipe</th>
<th>AF UNIX</th>
<th>UDP</th>
<th>TCP</th>
<th>TCP conn</th>
<th>(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>2.380</td>
<td>7.674</td>
<td>14.0</td>
<td>20.5</td>
<td>21.8</td>
<td>77.4</td>
<td></td>
</tr>
<tr>
<td>dkts</td>
<td>2.390</td>
<td>7.715</td>
<td>14.0</td>
<td>20.3</td>
<td>21.7</td>
<td>77.1</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Memory latencies.

<table>
<thead>
<tr>
<th></th>
<th>L1 cache</th>
<th>L2 cache</th>
<th>Main mem</th>
<th>(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>0.999</td>
<td>9.215</td>
<td>116.0</td>
<td></td>
</tr>
<tr>
<td>dkts</td>
<td>0.999</td>
<td>9.222</td>
<td>116.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Bandwidths for local communications.

<table>
<thead>
<tr>
<th></th>
<th>pipe</th>
<th>AF UNIX</th>
<th>TCP</th>
<th>file reread</th>
<th>mmap reread</th>
<th>bcopy (libc)</th>
<th>bcopy (hand)</th>
<th>mem read</th>
<th>mem write</th>
<th>(MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>1448</td>
<td>1665</td>
<td>465</td>
<td>1313</td>
<td>1666.6</td>
<td>426.5</td>
<td>443.5</td>
<td>1658</td>
<td>662.4</td>
<td></td>
</tr>
<tr>
<td>dkts</td>
<td>1448</td>
<td>1683</td>
<td>478</td>
<td>1306</td>
<td>1661.9</td>
<td>424.1</td>
<td>443.5</td>
<td>1660</td>
<td>650.4</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- classify the kernel threads into three groups based on how they are associated with real-time tasks

- propose a new scheduling algorithm for all kernel threads using a weighted average PIP mechanism
  - the average priority
  - the weight value

- by experiment, the response time of real-time tasks was greatly reduced when compared to the current Linux system