Supporting Time Sensitive Application in a Commodity OS

Ashvin Goel, Luca, Charles, Jim, Jonathan Walpole
Oregon Graduate Institute, Portland
Presented by Emalayan
Agenda

• Problem Definition
• Design Objectives
• Solution - TSL
  ‒ Firm Timers
  ‒ Preemptable kernel
  ‒ Efficient scheduler design
• Results
• System overhead analysis
• Conclusion
Problem Identification

Real Time systems:
- Must meet an exact deadline
- Failure to meet deadline means system failure
- E.g. (car engine, air crafts)

Time Sensitive Applications:
- Application with real-time requirements
- Examples: A/V media, soft modems
- Emphasis on:
  - Timing – e.g. periodic execution with low jitter
Problem Identification

- Time sensitivity constraint of Commodity operating systems.
- Real-time systems not good at throughput applications. Limited throughput of Real-time systems.
- General purpose OS should offer both Time Sensitivity and high throughput.
Design Objectives

• Providing efficient support for time-sensitive applications in a commodity OS without considerably degrading the performance of traditional throughput oriented applications.
Execution Sequence – In Commodity OS

- Timer Latency – depends on the clock frequency
- Pre-emption latency – time taken to interrupt kernel’s current activity and enable the scheduler
- Scheduling Latency – Time taken to schedule the event
Solution – Time Sensitive Linux

- A system with following future
  - Very low timer latency
  - Responsive Kernel with Fine-Grained pre-emptibility
  - CPU scheduling algorithm
Timer Latency

• Different type of timers
  – Periodic timers
    • High frequency increase interrupt overhead.
    • Low frequency yield poor performance for real time.
  – One shot timers
    • Interrupts only when needed
    • Cost of re-programming
    • High cost of interrupts – used frequently
  – Soft timers
    • Cost of polling for the event
    • May have delays due to polling
  – Firm Timers
    • Combine all the advantages of above timers
    • Low over head
Firm timers

- Hybrid approach
- Combination of one shot timer and soft timer
- Uses APIC one shot timers in Intel Pentium

Taken from http://web.cecs.pdx.edu/~walpole/class/cs533/winter2008/home.html
Function of Firm Timers

- Poll for timer expiry
- System call
- Reprogram One-shot timer and fire the event
- Overshoot Parameter

Time-Sensitive Application ready For execution

One-shot timer expiry

Timer Latency
Fine Grained Preemptible Kernel

• Bottleneck
  – Length of non-preemptible sections

• Solutions
  – Explicit insertion of preemption points
  – Any time preemption
    • Using spin locks for shared data
  – Anytime preemption by means of acquiring and releasing locks (Robert Love’s lock breaking kernel)
## Fine Grained Preemptible Kernel

<table>
<thead>
<tr>
<th>Methods</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit preemption</td>
<td>- Fine grained preemptibility</td>
<td>- Explicit points should be inserted in system calls</td>
</tr>
<tr>
<td>Any time preemption (using spin locks)</td>
<td>- Easy Implementation</td>
<td>- Poor performance with large shared data</td>
</tr>
<tr>
<td></td>
<td>- Independent of system call paths</td>
<td></td>
</tr>
<tr>
<td>Robert Love’s lock breaking kernel (acquire/ release / re acquire)</td>
<td>- Fine grained preemptibility</td>
<td>- Modification to the existing kernel ???</td>
</tr>
</tbody>
</table>
CPU scheduling

• Proportion based CPU scheduling
  – Each task is allocated a fixed proportion of time
  – Provides Temporal Protection
    • If a task over runs only the task itself will suffer possible consequences
  – Difficult to determine proportion

• Priority based CPU scheduling
  – Real time priorities are assigned to time sensitive tasks based on application needs
TSL scheduling model

• According to the application needs Real-time priorities are given
• Based on Highest locking priority protocol (HLP) to cope with priority inversion
• if(Only fixed priority tasks are accessing X Server)
  {schedule X Server with maximum priority}
else
  {schedule the proportion-priority task with high priority}
Results

• Setup:
  – Linux 2.4.16 (using Firm timers mentioned earlier)
  – Robert Love’s lock-breaking preemptible kernel patch
  – Proportion-period scheduler
  – 1.5 GHz Intel P4
  – 512 MB RAM
Results

• Micro Benchmarks
  – Evaluated timer latency and pre-emption latency
  – Using nanosleep()
  – Measured sleeping time
  – Maximum latency < 1ms
    • Average linux kernel latency > 15 ms
Real time application latency - Mplayer

Audio/video skew on Linux and TSL with user-level CPU load
Real time application latency - Mplayer

Audio video skew on Linux and on TSL with kernel CPU Load

Latency measured by audio/video synchronization skew
Real time application latency - Proportion period scheduler

<table>
<thead>
<tr>
<th>No Load</th>
<th>File System Load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max Proportion Deviation</td>
</tr>
<tr>
<td>Thread 1</td>
<td>0.3% (~25 μs)</td>
</tr>
<tr>
<td>Proportion: 40%, 3276.8 μs</td>
<td>Period: 8192 μs</td>
</tr>
<tr>
<td>Thread 2</td>
<td>0.7% (~3 μs)</td>
</tr>
<tr>
<td>Proportion: 20%, 102.4 μs</td>
<td>Period: 512 μs</td>
</tr>
</tbody>
</table>

Deviation in proportion and period when two process are run under the proportion period scheduler in TSL
System overhead analysis – preemption checks

• Memory Access
  – TSL overhead .42 +/- .18%

• Fork
  – TSL overhead .53 +/- .06%

• File System Access
  – TSL no significant overhead

• Overhead of checking for preemption points in TSL low
System overhead analysis – Firm Timers

Figure 6: Overhead of firm timers in TSL as compared to standard Linux with 20 timer processes.

Figure 7: Overhead of firm timers in TSL as compared to standard Linux with 50 timer processes.
Conclusion

• TSL provides ideal real time support by means of
  – Firm Timers, Pre-emptable kernel, Proportion based scheduler.

• TSL can be used in both time sensitive and throughput oriented application

• TSL needs further investigation
  – Interrupt service time
  – Network processing latency
  – Fine grained accounting time
THANK YOU
Questions

1. They talk about releasing and re-acquiring kernel locks to allow pre-emption of kernel threads. Wouldn't this mean that a whole lot of assumptions that have been made may no longer hold when the lock is acquired again and need to be verified. What is the overhead of such checks and how much reprogramming of the OS do they require?

2. For proportion based scheduling they need to know the time required for correct execution. While this may be ok in an RTOS world, how does it translate for a general purpose operating system?
Questions

3. The Soft Timer paper by Aron and Druschel already dictates the use of the hardware timer to provide an upper bound on delay. Why do the authors seem to make the greater claim, when their contribution is actually limited to replacing the periodic timer with a frequently reprogrammed one-shot timer?

4. One of the ideas described in this paper, one-shot timers, has been implemented in the Linux kernel. However, this feature was not implemented for the sake of real-time guarantees. It was implemented to save power by avoiding unnecessary wakeups. Does the implementation of one-shot timers (called "tickless system" in the Linux kernel) also grant the benefits described in this paper, or not?
Questions

6. There is further complexity added when they combine explicit preemption and the preemptible kernel approach by releasing and reacquiring spin-locks at strategic points. This takes relatively simple code and makes it hard to verify. The implementation effort could be even more difficult in a bigger OS where the kernel has much shared data. How would this system be modified to support multi-core machines?
Questions

7. How can prove that the soft timer is approximate to the event happens to the system? How to predict the event?

8. If there is a malicious process that request the resource all the time, does it mean that the TSL always allocate it to the highest priority?

9. Nowadays, the CPU is much faster than the time this paper written, and therefore high frequency interrupt of timer is allowed (1ms now). So do you think we still need such kind of work to support more time-sensitive applications?
Questions

10. The author claimed that traditional commodity kernels disable preemption for the entire period of time when a thread in the kernel. Why most kernels forbid these preemption? What is trade off between allow or disable preemption inside the kernels?
Questions

11. Priority Inversion
The authors note that many applications many require soft real-time scheduling and must interact with other processes. The authors claim that the HLP protocol fixes their X server problem. Is this actually the case? will this cause the X server to run at real-time priority to do work for non real-time processes?

Feedback Scheduler (section 4.4.2)
Their scheduler relies on an application specific metric to indicate progress, yet they do not discuss this at length. Their example of a bounded buffer as a metric is also a bit dubious as it seems to 'know' that the 'progress' of two processes is related. Do we buy this?

Timer Overshoot
Timer Overshoots allow the OS to impose a hard deadline on the scheduling of an event. The window between the actual event deadline and the timer overshot is a window in which soft timers may be used to schedule an event. This guarantees that an event will always be delivered a little later that scheduled. Why didn't the authors attempt to deliver events a little earlier as well?
Questions

12. I wonder if there can be a pure one-shot or soft timer mechanism, which doesn’t get involved with periodic timer, being able to handle all events well?

13. It is said in the paper that a real-rate scheduler uses an application specific progress rate metric to assign correct allocations to tasks. Will this notion conflicts with the goal of TSL that runs on a general commodity OS?