Predictable Interrupt Management and Scheduling in the Composite Component-based System

Gabriel Parmer and Richard West

Computer Science Department
Boston University
Boston, MA 02215
{gabep1, richwest}@cs.bu.edu

December 2, 2008
Motivation, Goals, and Challenges

Customizable/extensible base system upon which wide range of systems can be built

- with application-specific services and abstractions

System dependability challenges

- code complexity
  - complicate testing and verification
  - focus on fault tolerance
- services provided by 3rd party, untrusted developers
  - schedulers, synchronization policies, ... 

Can we provide a canonical base system that is predictable and dependable?
System policies and abstractions defined in separate components (schedulers, synchronization policies, etc.) at user-level in their own protection domains

Constrain scope of impact of component faults

Focus on

- application-specific system composition
- system-provided dependability
Simple Composite System

- Task 1 (T1)
- Task 2 (T2)
- Network (N)
- Demultiplexer (DM)
- Network Device
- Deferrable Server (DS)
- Fixed Priority, Round Robin (FPRR)
- Timer

- components related via \textit{dependency}
Correctness of RT systems dependent on tasks’ temporal behaviors

**Goal:** Define system scheduling policies in components
- control task and interrupt execution
- maintain accurate execution accountability
- must be efficient and predictable

**Challenge:** increased overhead of scheduler invocation
- inter-protection domain communication
Composite Scheduler Components

Composite kernel provides trusted communication between components

- scheduling external to kernel

Composite includes functions to

- create a hierarchy of schedulers
- grant control of specific threads to certain schedulers

Schedulers multiplex CPU via

\texttt{cos\_switch\_thread(thd\_id, \ldots)}
Common design: threads in protection domains that communicate with each other via IPC

- each IPC includes scheduling decisions
- user-level component scheduler → significant overhead, potentially increased WCET of a critical path
Composite: component invocation via migrating thread model

- A thread, \( \tau \) executing in component A invokes B via the kernel and continues executing in B
  - \( \tau \) is charged for execution in A and B
- IPC does not require scheduling decision by design
Interrupt Execution Management

Scheduling of interrupts

- interrupts promoted to threads
- run interrupt thread, or currently executing thread?
  - scheduling decision needed
- interrupt thread finishes execution
  - another scheduling decision needed

Possibility of significant overhead with user-level scheduling

- lessen effective utilization
- increase interrupt response time
Composite’s mechanism for asynchronous “upwards” execution

- **brands**
  - a path of components to guide the asynchronous execution
  - a priority/urgency with which to schedule a corresponding upcall

- **upcalls**
  - thread, associated with a brand
  - conducts the asynchronous execution along path recorded in brand

Interrupts are *branded*, which associates them with a brand, and attempts to execute the brand’s upcall
Avoiding Scheduler Invocations

Scheduling decision required when interrupt is branded
- shared data-structure between schedulers and kernel
  - scheduler posts urgency/importance of threads with kernel
  - kernel publishes CPU usage information to scheduler (cycles)
- when upcall attempted for a brand
  - kernel compares urgency/importance of current thread with brand
  - switch to upcall thread if higher urgency/importance
When upcall completes execution

- *common case:* immediately switch back to preempted thread

- *rare case:* possibly when a scheduling decision involving upcall has been made
User-Level Scheduler Synchronization

Schedulers must synchronize around critical sections
- mechanism must be efficient, predictable, policy neutral

Kernel-provided semaphores, mutexes, locks???
- but these primitives *rely* on a scheduler
  1. $\tau_1$ acquires lock to shared run-queues →
  2. $\tau_1$ is preempted →
  3. $\tau_2$ attempts to take lock: contention →
  4. kernel must know which thread to switch to →
  5. invoke scheduler that synchronizes around run-queues →
  6. *livelock*

Allow schedulers to disable/enable interrupts???
- not with untrusted/malicious/errant schedulers
Uncontested critical section access

- user-level solution using lock-free synchronization and restartable atomic sections
  - sections of assembly: if preempted in section, instruction pointer reset to start of section
  - can model atomic instructions

Contention for critical section

- library and kernel supplied wait-free synchronization
- higher-priority thread, $H$, “helps” lower priority thread through critical section, which then switches back to $H$
Microbenchmarks

- Thread migration efficient?
- Thread switching support efficient?
- Brands/upcall costs vs. scheduler invocations for interrupts

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware RPC Costs (U/K transitions, page tbls)</td>
<td>1110</td>
</tr>
<tr>
<td>Linux Pipe RPC</td>
<td>15367</td>
</tr>
<tr>
<td>Composite Component Invocation</td>
<td>1620</td>
</tr>
<tr>
<td>Linux Thread Switch</td>
<td>1903</td>
</tr>
<tr>
<td>Composite Base Thread Switch</td>
<td>529</td>
</tr>
<tr>
<td>Composite Thread Switch w/ FPRR</td>
<td>976</td>
</tr>
<tr>
<td>Composite Brand w/ Upcall Execution</td>
<td>3442</td>
</tr>
<tr>
<td>Composite Brand w/ Scheduler Invocations</td>
<td>9410</td>
</tr>
</tbody>
</table>

1 2.4 Ghz Pentium IV, Linux 2.6.22
Case Study: Predictable Interrupt Scheduling

**Goal:** predictable interrupt scheduling

Predictable task execution
- control interrupt interference with tasks
- tasks have dependencies on interrupts!
  - a NIC interrupt executes on behalf of task that will receive that packet

Intelligent scheduling of interrupts
- demultiplexing component inspects packet contents
- brands specific to the contents of the packet
  - interrupts for different tasks are scheduled differently
All interrupts executed at highest priority

- italic nodes are schedulers
- dotted lines represent dependencies
Linux-Style Interrupt Scheduling II

Task 1 & 2 interrupt (prio 0)
Task 1 processing (prio 1)
Task 2 processing (prio 2)

Parmer, West, BU CS
Scheduling in Composite
Interrupts are executed at the priority above their associated task, but below other tasks of higher priority.
Priority Differentiation for Tasks and Interrupts II

Parmer, West, BU CS Scheduling in Composite 20/33
Interrupts are executed at a higher priority than tasks, but in an aperiodic (deferrable) server.
Threaded Interrupts with Deferrable Server II

Parmer, West, BU CS

Scheduling in Composite
Interrupts are executed at the priority above their associated task, but below other tasks of higher priority, and are executed in deferrable servers with allocations commensurate with desired task progress.
Task 1 interrupt (prio 0, ds 5/20)
Task 1 processing (prio 1)
Task 2 interrupt (prio 2, ds 2/20)
Task 2 processing (prio 3)
### Overall Comparison

<table>
<thead>
<tr>
<th></th>
<th>Task 1 (higher prio.)</th>
<th>Task 2 (lower prio.)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest Priority Interrupts</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Interrupt Thread Prioritized Interrupts</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Differentiated Service</td>
<td>10</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

Cumulative Packets Processed (x10^9)

Parmer, West, BU CS Scheduling in Composite
Related Work

L4, Scheduler Activations (inc. K42), middleware-scheduling, CPU inheritance scheduling, . . .

No previous work including all of the following

1. define complete scheduling behavior of all execution in system
2. schedulers don’t have to be trusted
3. efficient even in presence of frequent interrupts
Conclusions

Component-based schedulers enable
- application-specific system behavior
- high confidence system/fault tolerance

User-level component-defined scheduling policies
- can precisely control the system’s temporal behavior
- can maintain accurate accounting information even for interrupt execution
(d) Packets/Sec in Stream 2 Sent, Stream 1 Constant at 48800

Task 1 & 2 interrupt (prio 0, ds 7/20)
Task 1 processing (prio 1)
Task 2 processing (prio 2)
Task 1 interrupt (prio 0, ds 5/20)
Task 1 processing (prio 1)
Task 2 interrupt (prio 2, ds 2/20)
Task 2 processing (prio 3)
Composite Thread Migration

Thread 0

Task 1 (T1)  Task 2 (T2)

Network (N)

Demultiplexer (DM)

Network Device

Deferrable Server (DS)

Fixed Priority, Round Robin (FPRR)

Timer

Thread 0’s Invocation Stack

T1

N

FPRR
1. Thread\textsubscript{0} calls scheduler (FPRR) to create a brand to be executed from DM.
1. Thread\_0 calls scheduler (FPRR) to create a brand to be executed from DM

2. \texttt{cos\_brand\_cntl(BRAND\_CREATE, DM)} in FPRR
Thread\(_1\) in the Demultiplexer wishes to cause an asynchronous upcall.
1. Thread\(_1\) in the Demultiplexer wishes to cause an asynchronous upcall
2. Thread\(_1\) executes \texttt{cos\_brand\_upcall(Brand, arg1, arg2)}