Xen and Nooks

Robert Grimm
New York University
Agenda

- Altogether now: The three questions
- The (gory) details of Xen
  - We already covered Disco, so let’s focus on the details
- Nooks
  - The grand tour
Altogether Now: The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
The (Gory) Details of Xen
The Details of Xen

- Memory management
  - x86 has hardware-accessed page tables and TLB
  - Guest OS’s responsible for managing page tables
    - Provide machine memory (from their reservations)
    - Have direct read-only access
    - Defer to Xen for (batched) updates

- CPU
  - Guest OS’s run in otherwise unused ring 1
  - Privileged instructions need to be processed by Xen
    - Though, the x86 makes life a little difficult (how?)
  - Fast exception handler does not require Xen interaction
    - Must execute outside ring 0 (what does Xen need to do?)
The Details of Xen (cont.)

- Device I/O
  - Xen presents idealized device abstraction
    - Data is transferred through shared-memory buffer rings
    - Upcalls are delivered through event delivery mechanism
The Gory Details

- **Control transfer**
  - Hypercalls: synchronous software traps to VMM
  - Events: asynchronous, possibly batched upcalls to VMs

- **Data transfer through I/O rings**
  - Separate descriptors from actual data
    - Zero-copy transfer for data
  - Support batching and re-ordering
The Gory Details (cont.)

- Virtual memory
  - Remember: Guest OS’s manage page tables (not shadows)
    - Exposes names and allocation
  - Validated by types and reference counts
    - Page directory/table, local/global descriptor table, writable
  - Page directory and tables pinned
    - Cannot be swapped (why?)

- Physical memory
  - Controlled through balloon driver
    - Pins pages, which are then returned to VMM
  - Mapped into machine memory
    - Xen publishes machine-to-physical mapping (necessary for what?)
Nooks
An isolation and recovery service

- Manages kernel-space extensions
- Targeted at commodity kernels
  - Implemented in Linux, should be easily portable to other OSs
- Detects, removes, and restarts misbehaving extensions
  - But not malicious ones
Why Do We Need Safe Extensions for *Commodity* Kernels?

- Cost of failures continues to rise
  - Downtime of mission-critical systems
  - Staffing for help-desk
- Extensions are common-place
  - 70% of Linux code
  - 35,000 different drivers with 120,000 versions for Windows XP
- Extensions are leading cause of failures
  - 85% of failures for Windows XP
  - 7 times more bugs in drivers than in rest of kernel for Linux
Why Not Use X?

- Capabilities, segments
  - Need specialized hardware, no support for recovery
- Micro-, pico-, exo-kernels
  - No support for recovery, some performance concerns
- Transactions
  - Sloooooooooooowwwwwwww
- Type-safe languages and runtimes
  - Not backwards compatible
- Software fault isolation
  - No support for recovery
Why Not Use X? (cont.)

- Virtual machines
  - Still have drivers in VMM

- But lead us to important insight
  - Only virtualize interface between kernel and extensions
    - In other words, we don’t need to be perfect, just good enough.
Nooks Architecture

- Two principles
  - Design for fault resistance, not fault tolerance
  - Design for mistakes, not abuse

- Three goals
  - Isolation
  - Recovery
  - Backward compatibility
Nooks Architecture: Four Functions

- **Isolation**
  - Lightweight protection domains for extensions
    - What is the trust relationship?
  - Extension procedure call (XPC)

- **Interposition**
  - Wrappers for all kernel ↔ extension crossings
    - Manage control and data flow

- **Object-tracking**
  - List of kernel data structures modified by extension

- **Recovery**
  - Removal, restarting of extensions
Nooks Implementation

- Additional layer for Linux 2.4.18
  - Same privilege for all code (ring 0)
  - Memory protection through page tables

Compared to 2.4 million lines in Linux kernel

<table>
<thead>
<tr>
<th>Source Components</th>
<th># Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Management</td>
<td>1,882</td>
</tr>
<tr>
<td>Object Tracking</td>
<td>1,454</td>
</tr>
<tr>
<td>Extension Procedure Call</td>
<td>770</td>
</tr>
<tr>
<td>Wrappers</td>
<td>14,396</td>
</tr>
<tr>
<td>Recovery</td>
<td>1,136</td>
</tr>
<tr>
<td>Linux Kernel Changes</td>
<td>924</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>2,074</td>
</tr>
<tr>
<td><strong>Total number of lines of code</strong></td>
<td><strong>22,266</strong></td>
</tr>
</tbody>
</table>

Table 2: The number of non-comment lines of source code in Nooks.
Isolation

- Lightweight protection domains
  - Private memory structures for each extension
    - Heap, stacks, memory-mapped I/O regions, buffers
  - Different page tables for kernel and each extension (why?)
    - Kernel can read and write all memory
    - Each extension can only write its own memory

- XPC
  - Saves caller’s context, finds stack, changes page tables
  - May be deferred
    - Amortize cost over several logical transfers
Interposition & Wrappers

- **How to interpose?**
  - Bind extensions to wrappers instead of kernel functions
  - Explicitly interpose on extension initialization call
  - Replace function pointers with wrapped versions

- **What about kernel objects?**
  - Some are read only → done
  - Some are written by extensions
    - Non-performance-critical updates through XPC
    - Performance-critical updates on shadow copy, synchronized through a deferred XPC on next regular XPC
      - Call-by-what?
More on Wrappers: What They Do, How to Write Them

- Check parameters for validity
- Implement call-by-value-result for kernel objects
- Perform XPC

- Skeleton generated by tool
- Body written by hand

Figure 4: Control flow of extension and kernel wrappers.
Even More on Wrappers: Code Sharing

Figure 5: Code sharing among wrappers for different extensions.
Object Tracker

- Currently supports 43 types
  - E.g., tasklets, PCI devices, inodes
- Records addresses of all objects
  - Used for one XPC only: table attached to task structure
  - Used across several XPCs: hash table
- Keeps mapping between kernel and extension versions
- Tracks object lifetimes
  - Single XPC call
  - Explicit allocation and deallocation
  - Semantics of object (e.g., timer data structure)
Recovery

- Triggered by
  - Parameter validation, livelock detection, exceptions, signals

- Performed by
  - Recovery manager
    - Cleans up after extension
  - User-mode agent
    - Determines recovery policy

- Performed in several stages
  - Disable interrupts, unwind tasks, release resources, unload extension, reload and init extension, re-enable interrupts
Limitations

- Extensions run in kernel mode
  - May execute privileged instructions
  - May loop forever (but Nooks detects livelock)
- Parameter checking is incomplete
- Recovery safe only for dynamically loaded extensions
Evaluation

- The two eff's
  - Effectiveness (reliability)
  - Efficiency (performance)
Effectiveness

- Nooks prevents 99% of system crashes
- But catches only a limited number of non-fatal failures

Why the differences?
Efficiency

- XPC rate serves as performance indicator
  - Three broad categories
Efficiency (cont.)

- There’s more code to run
- The code runs more slowly
Discussion

- If only we had a software/tagged TLB…
- What about end-to-end benchmarking?
  - All/most drivers managed by Nooks
  - Typical application mix
    - Server/desktop environment
- How many wrappers is enough wrappers?
  - Remember the code-sharing slide…
- How general is Nooks?
  - Only one communication pattern is supported
    - Kernel ↔ extension, but not between extensions
- So, when should we use Nooks?