Introducing the Chrono-kernel: Kernel Privilege for the People

Larry Woodman, Ulrich Drepper, Richard Jones, and Daniel Bristol de Oliveira

Abstract

We have retrofit Linux with the kElevate mechanism in an x86_64 prototype, and we have demonstrated the same technique is viable on an ARM64 proof of concept.

Approach: Instead of resorting to kernel hypervisors or custom operating system construction, the goal of this project is to marry the convenience of application programming on Linux with the full power of the kernel's mode of execution.

A Chrono-kernel is a system that offers first-class support for application threads to access supervisor execution mode. We retrofit Linux with a new mechanism, kElevate, turning it into a Chrono-kernel. kElevate allows any set of application threads to toggle into the supervisor mode of hardware execution in ~10 nanoseconds. We call these “elevated” threads.

Deployment: kElevate has wide applicability. It runs baremetal, in containers, and in virtualization. It can be invoked from any language via the syscall interface.

We have retrofitted Linux with the kElevate mechanism in an x86_64 prototype, and we have demonstrated the same technique is viable on an ARM64 proof of concept.

Context: Like Kernel Modules, elevated threads are unlimited in their power to change the system, so they are similarly protected behind supervisor access. Unlike Kernel Modules, elevated threads are built to run from the application context. Elevated applications use standard build processes. ABI compatibility may be maintained if desired. When running in kernel privilege, an elevated thread can make structural changes to Linux. It can interpose on system structures like the interrupt descriptor table, to place core system components and code.

We show baremetal latency plots demonstrating that a server using the kElevate mechanism improves significantly when using shortcuts directly into the kernel, and in parallel modifies the low-level internal interfaces of the kernel, in addition to the usual syscall API, utilizing the kernel as a library. We demonstrate using this to shortcut standard syscall paths as well as to perform more aggressive shortcircuiting deep in the kernel.

We have many more tasks than we have developers to implement them! We are looking for help from both types of developers: application and kernel. We are looking for clear kernel bound latency/throughput/real-time optimization objectives and kernel hackers who would be interested in helping us get through our to-do lists or bring their own perspectives to the work. If this sounds interesting to you please contact Tommy (tommyu@bu.edu).

Microbenchmark Experiments

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Linux</th>
<th>kElevate (thread), short</th>
<th>kElevate (thread), long</th>
<th>Average Latency</th>
<th>Percent Throughput</th>
<th>Average Latency (short)</th>
<th>Percent Throughput (short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid()</td>
<td>4.87s</td>
<td>1.70s</td>
<td>1.51s</td>
<td>11.2%</td>
<td>98.8%</td>
<td>11.2%</td>
<td>98.8%</td>
</tr>
<tr>
<td>kread[read/c]</td>
<td>5.50s</td>
<td>12.9%</td>
<td>1.51s</td>
<td>11.2%</td>
<td>98.8%</td>
<td>11.2%</td>
<td>98.8%</td>
</tr>
<tr>
<td>topbsd[read/c]</td>
<td>6.45s</td>
<td>32.4%</td>
<td>1.28s</td>
<td>24.7%</td>
<td>75.3%</td>
<td>24.7%</td>
<td>75.3%</td>
</tr>
</tbody>
</table>

Not Yet Peer Reviewed

We show baremetal latency plots demonstrating that a server using the kElevate mechanism improves significantly when using shortcuts directly into the kernel, and in parallel modifies the low-level internal interfaces of the kernel, in addition to the usual syscall API, utilizing the kernel as a library. We demonstrate using this to shortcut standard syscall paths as well as to perform more aggressive shortcircuiting deep in the kernel.