Avoiding system call overhead via dedicated user and kernel CPUs
Josh Triplett, Philip W. Howard, Eric Wheeler, Jonathan Walpole

History
From batching...
Task A Task B Task C Task D ...
... where every task runs to completion...
... to timeslicing...
A1 A2 A3 A4
B1 B2 B3 B4
C1 C2 C3 C4
... where every task appears to have a CPU...
... to manycore...
A ...
B ...
...
Z ...
... where every task can have its own CPU

System call interface
But wait, we still timeslice two “tasks”...
A (user) A (kernel) A (user) A (kernel) ...
B (user) B (kernel) B (user) B (kernel) ...
... so give them each their own CPU!

Avoid privilege switch and cache pollution; preserve locality.

Architecture
Application writes syscall number and params into shared memory
syscall number result if complete args[0] ... args[5] syscall number ...

Syscall thread, in-kernel, calls syscall function directly
ret = sys_call_table[num](args[0], ..., args[5]);
Syscall thread writes result/errno, indicating completion
calls[i].ret = ret;

Features
Provides mechanism in kernel, keeps policy in userspace
• Syscalls executed in order—start more syscall threads if you want parallelism.
• Wait for the result, or don’t, the kernel doesn’t care.
• Just a standard thread calling a new syscall; no magic new thread type or scheduler
Non-invasive
• No changes to user code or existing kernel syscalls
• Implementation entirely within userspace syscall invocation and new syscall

Results
<table>
<thead>
<tr>
<th>syscall</th>
<th>before (ns)</th>
<th>after (ns)</th>
<th>% faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid</td>
<td>87</td>
<td>65</td>
<td>33%</td>
</tr>
<tr>
<td>close(open(/dev/null))</td>
<td>1583</td>
<td>828</td>
<td>91%</td>
</tr>
<tr>
<td>read(/dev/zero),write(/dev/null),4kB</td>
<td>741</td>
<td>352</td>
<td>110%</td>
</tr>
<tr>
<td>munmap(mmap(/dev/zero)),4kB</td>
<td>1398</td>
<td>392</td>
<td>257%</td>
</tr>
<tr>
<td>munmap(mmap(/dev/zero)),16MB</td>
<td>2611</td>
<td>675</td>
<td>287%</td>
</tr>
<tr>
<td>open,munmap(mmap(/dev/zero)),close,4kB</td>
<td>3089</td>
<td>699</td>
<td>432%</td>
</tr>
<tr>
<td>open,open,read(zero),write(null),close,4kB</td>
<td>3538</td>
<td>647</td>
<td>446%</td>
</tr>
<tr>
<td>open,open,read(zero),write(null),close,4MB</td>
<td>1627673</td>
<td>254423</td>
<td>540%</td>
</tr>
</tbody>
</table>

Future Work
Asynchronous pipelining

Data structure locality
• Dedicate kernel cores to specific functionality
• Avoid synchronization entirely through ownership of data

Related Work
FlexSC
• Modified Linux syscall interface; new kind of thread
• Targeted at server-class applications with many threads
• Custom threading library batches syscalls across threads
• Asynchronous only—cannot depend on syscall ordering

Corey, Tornado, Multikernel, Barrelfish
• Built from scratch, fundamental architectural differences
• Partition data to minimize sharing and increase locality

Syslets, Fibrils, other Linux proposals
• New kinds of threads, invasive kernel support
• Asynchronous only—cannot depend on syscall ordering

Exokernel, others
• Library OSes run at user level
• Reduced need for crossing protection domains

About the authors
From the Relativistic Programming group
Portland State University
http://wiki.cs.pdx.edu/rp/
Josh Triplett (josh@joshtriplett.org)
Phil Howard (pwh@cs.pdx.edu)
Eric Wheeler (ewheeler@ewheeler.net)
Jonathan Walpole (walpole@cs.pdx.edu)