Stack Architecture and Flat Memory
For Faster Syscalls
By: Eric LaForest

Background:
Switching between user and kernel mode can be expensive due to TLB flushes and saving processor state. This overhead negatively impacts fine-grained systems such as microkernel OSes.

Premise:
Use a simpler memory and processor architectures to improve the performance of mode switches.

1. Address Space

0 \rightarrow 2^n

RAM \quad I/O

Disk

Key Points:
- Flat address space: no virtual memory
- Memory-mapped I/O for disk, cycle counter, and console
- Address space after physical RAM is mapped to disk by kernel

2. Stack Architecture Summary

<table>
<thead>
<tr>
<th>data</th>
<th>MEM</th>
<th>adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>A</td>
<td>RS</td>
</tr>
<tr>
<td>IR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key Points:
- Stacks are non-addressable and on-chip
- All calculations done on top of Data Stack
- Memory load/store from top of Data Stack using Address Register
- Subroutine return addresses held in Return Stack
- Data can be moved between stacks
- Code is not position-independent (branches are absolute)

3. Virtualization

<table>
<thead>
<tr>
<th>data</th>
<th>MEM</th>
<th>adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>UB</td>
<td>TPC</td>
</tr>
<tr>
<td>DS</td>
<td>A</td>
<td>RS</td>
</tr>
<tr>
<td>IR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key Points:
- Memory load/store outside of memory bounds will cause a trap
- Return To User (RTU) privileged instruction to enter User Mode
- Executing RTU in User Mode causes a trap, used for syscalls

4. State After A Trap

<table>
<thead>
<tr>
<th>data</th>
<th>MEM</th>
<th>adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>(UB)</td>
<td>A</td>
<td>(IR)</td>
</tr>
<tr>
<td>(LB)</td>
<td></td>
<td>(PC)</td>
</tr>
<tr>
<td>(TPC)</td>
<td></td>
<td>Super</td>
</tr>
<tr>
<td>0</td>
<td>2^n</td>
<td>TPC</td>
</tr>
</tbody>
</table>

Key Points:
- Trap to Supervisor Mode executes in two cycles
- Memory bounds set to maximum range to make traps impossible
- Return to User Mode is the exact reverse process
- No memory traffic other than an instruction fetch

5. Access To Memory

Two ways for a process to access data from outside its bounds:

Trap:
The process attempts to directly read/write the data, causing a trap to kernel which decides whether to complete the operation or deny access to memory.

Syscall:
The process places a syscall number on the Data Stack and executes a Return To User (RTU) instruction, causing a trap to kernel.

Tests

g getpid(): have a process get its Process ID from its header
byte read: read one byte from a cached disk block
(Linux reads a byte, Stack reads an int)

6. Test Results

Linux results from imbench 3.0-a7-1 on kernel 2.6.20.6 on 2.2GHz AMD Athlon™ 64 with warm cache.

Stack results from cycle-accurate simulator running a simple kernel.

<table>
<thead>
<tr>
<th>Test</th>
<th>Linux</th>
<th>Stack</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid() trap:</td>
<td>N/A</td>
<td>98</td>
<td>3.22</td>
</tr>
<tr>
<td>getpid() syscall:</td>
<td>316</td>
<td>81</td>
<td>3.90</td>
</tr>
<tr>
<td>byte read trap:</td>
<td>N/A</td>
<td>105</td>
<td>5.87</td>
</tr>
<tr>
<td>byte read syscall:</td>
<td>616</td>
<td>N/A*</td>
<td></td>
</tr>
</tbody>
</table>

*Stack syscall reads entire block, trap returns one buffered byte

7. Conclusions

- A stack architecture and flat memory can improve syscall performance.
- Performance speedup is not the expected order of magnitude as most of the cycles (~70) are spent saving/restoring state and checking permissions.
- However, Linux was tested in ideal conditions (no TLB misses)
- Finally: improved performance on much simpler hardware than x86.

Further Work

- Simplifying stack trap mechanism: don't copy LB/UB to stacks on trap, let the kernel remember it per process.
- Extend trap mechanism to subroutine calls.
- Alternatively, remove initial trap checks by reducing source of traps to one method only (call, RTU, or mem. trap).
- Managing flat, non-virtual memory by using cheap cross-domain calls to dynamically generated code (fast IPC).