What’s Wrong with the Operating System Interface?

Collin Lee and John Ousterhout
Goals for the OS Interface

- More convenient abstractions than hardware interface
- Manage shared resources
- Provide near-hardware performance
Evolution of Threads

- **Multi-tasking (1970s)**
  - 1 thread per process.
  - Threads used to model concurrency between processes.

- **Multi-threading (1990s)**
  - 1 thread per concurrent task (within a process).
  - Threads used to model concurrency within a process (application).

- **Multi-Core (2000s)**
  - 1 thread per desired parallel execution (core).
  - Threads used to request resources for parallel execution.

- Designed to provide logical concurrency for long tasks.

- NOT designed to schedule parallel execution.
  - (Though this is how we use it today.)
Faster Threads for Short Tasks

- Today we have lower latency tasks.

- Threads aren’t fast enough to model concurrency for short tasks.
  - Threads were designed for long running task.
  - Thread creation 5.76µs; condition signal 4.14µs.

- Dynamic thread creation is impractical.
  - Most apps use a fixed number of threads or thread pool.
  - Most applications limit threads to number of expected cores.

- Why are kernel threads so slow?

- Kernel Bypass for Threads (User Space Threading)?
  - Go Routines? Arachne?
  - Are there issues with user space threading?
  - How does the current kernel interface restricting us?
Core Aware Threading

- Threading model not designed to express parallelism.
  - Designed for concurrency and co-opted for parallelism for multi-core.

- Applications today are blindly requesting parallelism.
  - Applications use threads as a proxy for cores.
  - No way for applications to know what resources (cores) are actually available to it.

- Blind thread allocation leads to inefficient resource utilization.
  - Underutilization (# threads < # cores) or inefficiency (# threads > #cores)

- Separate mechanisms for concurrency and parallelism?
  - We don’t really use threads for concurrency anymore anyway.

- New Core Request API?
  - Provide way for applications to ask for cores and and know when they are revoked.
  - On the other hand, does an application actually want to deal with cores?
### Does Virtual Memory Make Sense?

<table>
<thead>
<tr>
<th></th>
<th>1960’s</th>
<th>2018</th>
<th>Change</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk latency</td>
<td>80 ms</td>
<td>5 ms</td>
<td>16x</td>
<td>10 µs</td>
</tr>
<tr>
<td>Disk transfer rate</td>
<td>250 KB/s</td>
<td>200 MB/s</td>
<td>800x</td>
<td>2 GB/s</td>
</tr>
<tr>
<td>Memory size</td>
<td>256 KB</td>
<td>64 GB</td>
<td>250,000x</td>
<td>64 GB</td>
</tr>
<tr>
<td>Time to replace all of memory (random)</td>
<td>6.4 s</td>
<td>22 hrs</td>
<td>12500x</td>
<td>160 s</td>
</tr>
<tr>
<td>Time to replace all of memory (sequential)</td>
<td>1 s</td>
<td>320 s</td>
<td>320x</td>
<td>32 s</td>
</tr>
</tbody>
</table>

Can’t afford to page out unless I/O for a long time
Does Paging Make Sense?

- **How did we end up with 4 KB pages?**
  - Flexible management of physical memory: noncontiguous allocation
  - Flexible management of virtual address space (e.g., sharing)
  - Efficient paging (granular)

- **4 KB pages: too small**
  - TLB misses double cost of cache miss

- **1 GB pages:**
  - Too large: not enough independently manageable units?
  - Too large: internal fragmentation
  - Too small: still can’t map all of memory in TLB
● For example:
  ▪ Fixed number of segment descriptors (~100?) for each virtual address space
  ▪ Always loaded in “TLB”: no misses, ever

● Potential problems:
  ▪ Will there be enough independently shareable units?
    ● How many mmapped files might be open at once?
  ▪ Must do additions and subtractions during memory translation

● Compromise: Mark Hill’s proposal
  ▪ One segment per process, plus the usual paging
  ▪ Assumption: huge applications typically have one large memory region (e.g., RAMCloud log)
Kernel_Bypass_Networking++

- Kernel networking has been found to be too slow.
  - Designed to handle slow networks and packet loss.
  - 100B RAMCloud network read: 24µs (kernel UDP) vs 5µs (kernel bypass).

- Solution was to move the transport into user space (kernel bypass).
  - The kernel is only involved during initial setup and not on the data path.
  - Allows for customized, simpler, faster network transports.

- Problem: Possible interference between user space transports
  - Having multiple transports can cause avoidable contention on the network.
    - Should each app have its own transport?
    - Can applications share a transport? How do you deal with protection?

- New Kernel Shared “Network Resource” State API?
  - Provide a way for transports to share state about the network.