The Performance of $\mu$-Kernel-Based Systems

Liedtke et al.
presented by: Ryan O’Connor <rjo@cs.ubc.ca>

October 7th, 2009
By this time (1997) the OS research community had virtually abandoned research on pure $\mu$-kernels. 

- due primarily to the poor performance of first-generation $\mu$-kernels

Many researchers believe that $\mu$-kernel abstraction layer is either too low or too high

- **too low**: research should focus on *extensible-kernels*, allowing users to safely add new functionality to a monolithic kernel
- **too high**: kernels should export an interface resembling a hardware architecture

**Claim**: pure $\mu$-kernels are not fundamentally flawed, they just haven’t been done right yet
L4 and L^4Linux

the penalty for using a pure μ-kernel can be kept between 5% and 10%

μ-kernel abstractions are efficient enough to motivate their use in user applications
L4 exports a pure $\mu$-kernel interface (threads, address spaces, and IPC). In addition it supports:

- user-level paging, and recursive construction of address spaces
- translation of hardware interrupts to IPC messages
- fast context switches on small address-spaces using Pentium segmentation trick
L$^4$Linux is a Linux Single Server, an L4 process that provides a unix 'personality' for user applications. When designing L$^4$Linux, the authors made the following design decisions:

- all modifications to linux should be restricted to the architecture-dependent code
- no linux-specific modifications should be made to L4
- L$^4$Linux should be binary-compatible
Linux Server
- runs as a single L4 task and handles system calls and page faults
- maps and manages all physical memory

Interrupts
- top halves of interrupt handlers run as separate L4 threads
- one L4 thread runs as bottom half to all top halves
- all interrupt threads run at a priority above the Linux server thread to avoid concurrent execution

Linux User Processes
- each user process is an L4 task
- Linux server specifies itself as the pager for the process, and maps specific signal and emulation code into the process
System Calls

- modeled as IPC between user process and Linux server
- implemented by modifying libc, however an exception handler is needed to emulate the native syscalls trap for statically linked applications
- mapping kernel into user processes did not allow for the Pentium segmentation trick, so physical copyin and copyout operations were used instead

Signalling

- Linux delivers signals by directly manipulating the user process stack, which is not allowed in L4
- instead an additional signal-handler thread is added to each user process to manipulate the user process stack

Scheduling

- all threads are scheduled by L4’s scheduler
to evaluate performance of L4Linux, the authors compared it to native Linux and MkLinux (Linux single server on OSF-Mach)

- syscalls are over twice as slow in L4 compared to native, however MkLinux is close to 10 times as slow, even when co-located
- under AIM benchmark, L^4 Linux is 8% slower than native Linux on average
- under AIM benchmark, MkLinux is 29% slower than native Linux on average, when co-located
Ok, so we can run Linux 10% slower (just like the Mach paper claimed), now what?

- Pipes and RPC can be implemented faster, with more bandwidth using L4 directly.
- VM operations can be done faster when using L4 (bypassing Linux kernel)
- User-level paging allows for cache-partitioning for real-time applications
PCT and Grafting

Protected control transfer allows a thread to cross address spaces through special gates

- PCT was dismissed since IPC required just 7 more cycles
- Grafting addresses executing extensions in kernel address space
- "still and open question"
L³Linux incurs a 5% to 10% penalty "in a practical scenario"
using L4, user processes may gain additional performance in some areas (IPC and VM operations)
L4 can allow special applications with special requirements to co-exist with normal OSes.
Questions

- Why did their L^4 Linux require so much code (8,500 LOC vs 4,500 LOC)?
- How does L4 compare to hypervisors?
- Is L4 more or less relevant in today’s multicore world?
- Security?

Liedtke et al. presented by: Ryan O’Connor <rjo@cs.ubc.ca>