Operating System Kernels

Ken Birman
(borrowing some content from Peter Sirokman)

A short history of kernels

- Early kernel: a library of device drivers, support for threads (QNX)
- Monolithic kernels: Unix, VMS, OS 360...
  - Unstructured but fast...
  - Over time, became very large
  - Eventually, DLLs helped on size
- Pure microkernels: Mach, Amoeba, Chorus...
  - OS as a kind of application
- Impure microkernels: Modern Windows OS
  - Microkernel optimized to support a single OS
  - VMM support for Unix on Windows and vice versa

The great µ-kernel debate

- How big does it need to be?
  - With a µ-kernel protection-boundary crossing forces us to
    - Change memory-map
    - Flush TLB (unless tagged)
  - With a macro-kernel we lose structural protection benefits and fault-containment
- Debate raged during early 1980's

Summary of First Paper

- The Performance of µ-Kernel-Based Systems
  (Hartig et al. 16th SOSP, Oct 1997)
  - Evaluates the L4 microkernel as a basis for a full operating system
  - Ports Linux to run on top of L4 and compares performance to native Linux and Linux running on the Mach microkernel
  - Explores the extensibility of the L4 microkernel

Summary of Second Paper

- The Flux OSKit: A Substrate for Kernel and Language Research (Ford et al. 16th SOSP, 1997)
  - Describes a set of OS components designed to be used to build custom operating systems
  - Includes existing code simply using “glue code”
  - Describes projects that have successfully used the OSKit

In perspective?

- L4 seeks to validate idea that a µ-kernel can support a full OS without terrible cost penalty
  - Opened the door to architectures like the Windows one
- Flux argues that we can get desired structural benefit in a toolkit and that runtime µ-kernel structure isn't needed
Microkernels
- An operating system kernel that provides minimal services
- Usually has some concept of threads or processes, address spaces, and interprocess communication (IPC)
- Might not have a file system, device drivers, or network stack

Monolithic and Micro-kernels

Microkernels: Pro
- Flexibility: allows multiple choices for any service not implemented in the microkernel
- Modular design, easier to change
- Stability:
  - Smaller kernel means it is easier to debug
  - User level services can be restarted if they fail
- More memory protection

Microkernel: Con
- Performance
  - Requires more context switches
    - Each “system call” must switch to the kernel and then to another user level process
    - Context switches are expensive
    - State must be saved and restored
    - TLB is flushed

Context Switches

Paper Goals
- Is it possible to build an OS on a Microkernel that performs well?
  - Goal is to prove that it is
  - Port Linux to run on top of L4 (a microkernel)
  - Compare performance of L4Linux to native Linux
  - Since L4Linux is a “complete” operating system, it is representative of microkernel operating systems
**More Paper Goals**

- Is this actually useful? Is the microkernel extensible?
- Implemented a second memory manager optimized for real-time applications to run alongside Linux on L4
- Implemented an alternative IPC for applications that used L4 directly (requires modifying the application)

---

**The L4 Microkernel**

- Operations:
  - The kernel starts with one address space, which is essentially physical memory
  - A process can grant, map, or unmap pages of size $2^n$ from its own virtual address space
  - Some user level processes are pagers and do memory management (and possibly virtual memory) for other processes using these primitives.

---

**The L4 Microkernel (continued)**

- Provides communication between address spaces (inter-process communication or IPC)
- Page faults and interrupts are forwarded by the kernel to the user process responsible for them (i.e. pagers and device drivers)
- On an exception, the kernel transfers control back to the thread's own exception handler

---

**L4Linux**

- Linux source has two cleanly separated parts
  - Architecture dependent
  - Architecture independent
- In L4Linux
  - Architecture dependent code is replaced by L4
  - Architecture independent part is unchanged
  - L4 not specifically modified to support Linux

---

**L4Linux (continued)**

- Linux kernel as L4 user service
  - Runs as an L4 thread in a single L4 address space
  - Creates L4 threads for its user processes
  - Maps parts of its address space to user process threads (using L4 primitives)
  - Acts as pager thread for its user threads
  - Has its own logical page table
  - Multiplexes its own single thread (to avoid having to change Linux source code)

---

**L4Linux – System Calls**

- The statically linked and the shared C libraries are modified
  - System calls in the library call the kernel using L4 IPC
- For unmodified native Linux applications there is a "trampoline"
  - The application traps to the kernel as normal
  - The kernel bounces control to a user-level exception handler
  - The handler calls the modified shared library
A note on TLBs

- Translation Lookaside Buffer (TLB) caches page table lookups
- On context switch, TLB needs to be flushed
- A tagged TLB tags each entry with an address space label, avoiding flushes
- A Pentium CPU can emulate a tagged TLB for small address spaces

Microkernel Cons Revisited

- A significant portion of the performance penalty of using a microkernel comes from the added work to reload the page table into the TLB on every context switch
- Since L4 runs in a small address space, it runs with a simulated tagged TLB
- Thus, the TLB is not flushed on every context switch
- Note that some pages will still be evicted — but not as many

Performance – Compatibility

- L4Linux is binary compatible with native Linux from the applications point of view.

Performance – The Competitors

- Mach 3.0
  - A “first generation” microkernel
  - Developed at CMU
  - Originally had the BSD kernel inside it
- L4
  - A “second generation” microkernel
  - Designed from scratch

Performance – Benchmarks

- Compared the following systems
  - Native Linux
  - L4Linux
  - MkLinux (in-kernel)
    - Linux ported to run inside the Mach microkernel
  - MkLinux (user)
    - Linux ported to run as a user process on top of the Mach microkernel

Performance - Microbenchmarks
**Performance - Macrobenchmarks**

- AIM Benchmark Suite VII simulates “different application loads” using “Load Mix Modeling”.
  - This benchmark has fallen out of favor but included various compilation tasks
  - Tasks are more representative of development in a systems lab than production OS in a web farm or data center

**Performance - Analysis**

- L4Linux is 5% - 10% slower than native for macrobenchmarks
- User mode MkLinux is 49% slower (averaged over all loads)
- In-kernel MkLinux is 29% slower (averaged over all loads)
- Co-location of kernel is not enough for good performance

**So What?**

- If performance suffers, there must be other benefits – Extensibility
  - While Linux pipes in L4Linux are slower than in native Linux, pipes implemented using the bare L4 interface are faster
  - Certain primitive virtual-memory options are faster using the L4 interface than in native Linux
  - Cache partitioning allows L4Linux to run concurrently with a real-time system with better timing predictability than native Linux

**Microkernel Con: Revisited Again**

- The Linux kernel was essentially unmodified
- Results from “extensibility” show that improvements can be made (e.g. pipes)
- If the entire OS were optimized to take advantage of L4, performance would probably improve
- Goal Demonstrated

**Flux OS**

- Research group wanted to experiment with microkernel designs
- Decided that existing microkernels (Mach) were too inflexible to be modified
- Decided to write their own from scratch
- In order to avoid having it become inflexible, built it in modules
- Invented an operating system building kit!
The Flux OSKit

- Writing Operating Systems is hard:
  - Relevant OSs have lots of functionality:
    - File system
    - Network Stack
    - Debugging
  - Large parts of OS not relevant to specific research
  - Not cost effective for small groups

Adapting Existing Code

- Many OS projects attempt to leverage existing code
- Difficult
  - Many parts of operating systems are interdependent
  - E.g. File system depends on a specific memory management technique
  - E.g. Virtual memory depends on the file system
  - Hard to separate components

Separating OS Components

OSKit

- OSKit is not an operating system
- OSKit is a set of operating system components
- OSKit components are designed to be as self-sufficient as possible
- OSKit components can be used to build a custom operating system – pick and choose the parts you want – customize the parts you want

Diagram of OSKit

Example OS using OSKit
Another Example OS

OSKit Components

- Bootstrapping
  - Provides a standard for boot loaders and operating systems
- Kernel support library
  - Make accessing hardware easier
  - Architecture specific
  - E.g. on x86, helps initialize page translation tables, set up interrupt vector table, and interrupt handlers

More OSKit Components

- Memory Management Library
  - Supports low level features
  - Allows tracking of memory by various traits, such as alignment or size
- Minimal C Library
  - Designed to minimize dependencies
  - Results in lower functionality and performance
  - E.g. standard I/O functions don’t use buffering

Even More OSKit Components

- Debugging Support
  - Can be debugged using GDB over the serial port
  - Debugging memory allocation library
- Device Drivers
  - Taken from existing systems (Linux, FreeBSD)
  - Mostly unmodified, but encapsulated by “glue” code – this makes it easy to port updates

Two more OSKit Components

- Network Stack
  - Taken from FreeBSD and “encapsulated” using glue code
- File System
  - Taken from NetBSD and “encapsulated” using glue code

OSKit Component Interfaces

- File System
  - open()
  - close()
  - read()
- Client OS Kernel
  - malloc()
- Memory Manager
  - malloc()
OSKit Implementation

- Libraries
  - To the developer, the OSKit appears as a set of libraries that can be linked to programs
  - Therefore, easy to use

Providing Separability

- Most operating systems are modular, but this does not make them separable into components
- Modules will assume and depend on the implementation specifics of other modules
- In OSKit components are wrapped in “glue code” to make them independent of other components

Glue Code

- Overridable functions
  - E.g. all device drivers use a function `fdev_mem_alloc` to allocate memory
  - The client OS (the OSKit user) must provide an implementation of this depending on the memory manager used by the OS being built
  - The default implementation uses the OSKit memory manager

What is this “glue code”?

- Interfaces
  - Interfaces use the COM standard
  - Like a Java object, a COM interface has known methods that can be invoked
  - The internal state is hidden
  - Each block device driver can implement a common COM interface, allowing all drivers to look the same to the file system

More “glue code”

- The file system must use block device drivers
- Yet the file system can’t know what the block device driver code will be
- Device drivers can return pointers to interfaces, which can be passed to the file system
- The file system is bound to a block device driver at run time
Execution Environment

- It is impossible to turn all components into black boxes that will automatically work in all environments.
- The absolute basic needs of a component, a file system for example, is abstracted as specified execution environment that the developer must follow.

—from “The execution environment specifies limitations on the use of the component.”
- Is the component reentrant?
- Must certain functions in the interface be synchronized?
- Can the execution of the component be interrupted?
- Example: While the file system is not designed to be used on a multiprocessor system, the execution environment can be satisfied using locks to synchronize its use.

Exposing the Implementation

- The OSKit provides abstract interfaces to its components.
- The OSKit also provides implementation specific interfaces to allow the user to have more control over the component.
- Key: these specialized interfaces are optional.
- E.g. the memory manager can be used as a simple malloc, or it can manipulate physical memory and the free list directly.
- Components can offer multiple COM interfaces to do this.

—from “interfaces presented by the OSKit are implemented as “glue code”.”
- This glue code makes calls to the imported legacy code, and makes modifications as needed to emulate the legacy code’s original environment.
- The glue code also accepts calls from the legacy code and translates them back to the interface offered.
- Thus once two components are encapsulated, their interfaces can be joined together seamlessly.

The Obligatory Benchmark

- Measured TCP bandwidth and latency.

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>FreeBSD</th>
<th>OSKit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sender:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>72.4</td>
<td>71.2</td>
<td>71.3</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>60.0</td>
<td>78.6</td>
<td>78.7</td>
</tr>
<tr>
<td>OSKit</td>
<td>56.4</td>
<td>68.3</td>
<td>68.2</td>
</tr>
</tbody>
</table>

Table 1: TCP bandwidth in MBit/s measured with tcp between two Pentium Pro 200MHz PCs connected by 100Mbps Ethernet.

—from “FreeBSD can use discontinuous buffers,…”

Bandwidth Analysis

—Figure 3: Structure of the tcp and rtcp example kernels.
Latency

<table>
<thead>
<tr>
<th>Client</th>
<th>Linux</th>
<th>FreeBSD</th>
<th>OSKit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>152</td>
<td>168</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>168</td>
<td>157</td>
<td>210</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>210</td>
<td>222</td>
</tr>
</tbody>
</table>

Table 2: TCP one-byte round-trip time in μsec measured with ping between two Pentium Pro 200MHz PCs connected by 100Mbps Ethernet.

Case Study 2: Standard ML

- SML is a functional programming language
- Goal: to model concurrency as continuations in high level programming languages
- This requires ML and its compiler to be able to manipulate context switching – difficult if not impossible on a standard OS
- MU/OS constructed by 2 people over a semester using OSKit
- Other projects with similar goals have not succeeded (at the time)
  - Fox project at CMU
  - Programming Principles group at Bell Labs

Other language based OSs

- SR – a language for writing concurrent programs
  - Other attempts abandoned
- Java/PC
  - Given a Java Virtual Machine and OSKit, took three weeks
  - Sun’s version took much longer to build since it was written mostly from scratch in Java

OSKit vs. Microkernel

- A Microkernel is an architecture for operating systems designed to be flexible
- OSKit is a tool for making operating systems
- OS-s built with OSKit may or may not be microkernel
- OSKit gives greater flexibility than a microkernel, since even microkernels force some concepts (threads, IPC) onto the overall system