Operating Systems Architecture, Part II

CS239
Advanced Operating Systems
October 6, 1997
Dr. Richard Guy

Lecture Source Materials

- Microkernel Performance
  - On µ-Kernel Construction, Liedtke, SOSP95.
  - The Performance of µ-Kernel-Based Systems, by Hartig, et al, in SOSP, 1997.
- 64-bit architectures
  - Opal: A single Address Space System for 64-bit Architectures, by Chase, et al, IEEE WWOS92.
  - Sharing and Protection in a Single Address Space OS, by Chase, et al, ACM TOCS 11/94.

Source materials, continued

- Extensibility
  - Microkernels Meet Recursive Virtual Machines, by Ford, et al, in OSDI, 1996.
  - The Flux OSKit: A Substrate for Kernel and Language Research, by Ford et al, in SOSP97.
  - Extensibility, Safety and Performance in the SPIN Operating System, by Bershad, SOSP95.

Source materials, continued

- Exokernel: An Operating System Architecture for Application-Level Resource Management, by Engler, et al, in SOSP, 1995.
- Application Performance and Flexibility on Exokernel Systems, Kaashoek, et al, SOSP97.

Outline

- Microkernels, revisited
- Operating System Extensibility
- 64-bit Architectures

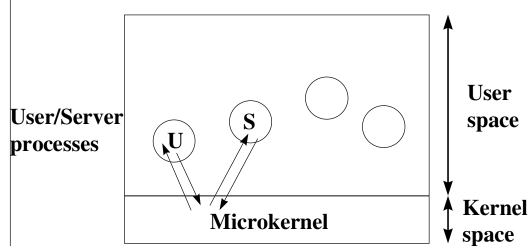
Are Microkernels Dead?

- “The operating systems research community has almost completely abandoned research on system architectures that are based on pure µ-kernels, i.e., kernels that provide only address spaces, threads, and IPC…”, by Hartig et al, in SOSP97.

Why Haven't µ-kernels Won?

- Typical microkernels require extensive OS/user boundary crossings for basic work
  - each crossing is fairly expensive
  - so, many microkernels have poor performance
- "Microkernel" grows out of its micro-ness
- Are microkernels a bad idea, or did the early designers make bad choices?

Generic Microkernel Model



The Problem With µ-kernels

- Microkernels should be small
- But they must also be powerful
- So what goes in your microkernel?
  - the fundamental decision in microkernel OS design
- Must choose minimal, yet sufficient, primitives

Microkernel Design - A Balancing Act

If you put too much in a microkernel
- It's no longer "micro," and you lose all advantages

If you put too little in a microkernel
- It lacks sufficient power, and performance suffers

The fundamental problem in microkernel design - kernel crossings are expensive

How To Optimize Microkernel Performance?

- Minimize what must be in the microkernel
- Make those primitives really fast
- e.g., Liedtke's L3 microkernel
  - "On μ-kernel Construction," SOSP95
  - follow-up work in SOSP97 with L4

Liedtke's Minimal Primitives

- Support for recursive construction of address spaces
  - With kernel-provided protected address space at the base
- Threads
- IPC
- Unique identifier support

L3 Address Space Support

- Supports three primitives
  1). Grant - give pages to another address space
  2). Map - share pages with another address space
  3). Flush - take back mapped or granted pages
- Not in microkernel -
  – Memory management
  – Paging
- I/O treated as address space

L3 Thread Support

- L3 thread is an activity executing inside an address space
- Characterized as a set of registers
  – including at least IP, SP, and state information
- Microkernel maintains association of thread to address space
  – otherwise, threads might unsafely cross address spaces

L3 Microkernel IPC Support

- Kernel provides communications between address spaces
  – by transferring messages between them
- Other IPC abstractions built on top of messages
- Messages can be used as an abstraction for hardware interrupts, too

L3 Support For Unique Identifiers

- Microkernel must reliably tag unique identifiers onto something
  – Threads, tasks, or channels
- Otherwise, prohibited operations can masquerade as legal operations

Can A Microkernel Supporting These Perform Well?

- Well, L3 is *very* fast
- Careful implementation of kernel-user task switches allows L3 to switch in less than 3 µsecs (primarily cost of entering/leaving kernel mode)
- Liedtke also argues that other poor switching and caching effects are avoidable

L4Linux Performance

System	Time	Cycles
Linux	1.68µs	223
L4Linux	3.95µs	526
MkLinux (kernel)	15.41µs	2050
MkLinux (user)	110.60µs	14710

getpid( ) system call on 133MHz Pentium

What's the Cost of Fast Microkernels?

- Non-portability - getting stuff to run that fast inherently involves using special characteristics of particular hardware
- Is this terrible?
  - only if the microkernel is large and complex
  - if not, porting to new architecture isn't too bad
  - especially since some porting required, anyway

Recent Research In Operating System Organizations

- Extensibility research
  - Spin
  - Exokernel
  - Fluke
- 64-bit address space architectures
  - Opal

Extensibility In Operating Systems

- Microkernels and layers systems promised easy extensibility
- But extensibility hasn't been so easy
- Why is extensibility hard?
- What can designers do to make it easier?

Why Is Extensibility Required?

- Hard to design one system that does everything well
  - Special OSs for databases and real time systems
  - Some propose special Java-supporting OS
- Things change, and systems must change with them
  - Five years ago, who knew network servers would need to deliver Web pages?
  - Today's OSs not equipped for multimedia

Why Is Extensibility Hard?

- What can(not) be extended in an OS?
- How can you design an OS that can be safely extended?
- How can extensions to an OS run sufficiently fast?

What Can Be Extended

- Most operating systems allow some extensions
  - e.g., new device drivers, new file systems
  - sometimes new schedulers or pagers
- Most systems don't allow everything to be extended
- Deciding what can and can't be extended is tricky

Safe Extensions

- How can one user (process) extend a system without endangering others?
  - worry about error or malice
- If the extension is in kernel mode, can't it do anything?
- If it isn't in kernel mode, can it perform well enough?

Extensibility and Performance

- The real trick is having both
  - without sacrificing safety
- Many extensions are at a very low level
  - they are invoked frequently
  - and must work quickly

Spin

- An extensible OS developed at the University of Washington
- Goals: extensibility and customization
- Approach is to allow users to download code into the kernel
- Suitable performance achieved by making the cost of using extensions comparable to a procedure call

The Spin Philosophy

- Provide a set of core operating system services
- Allow extension of system services by downloading new code <u>into</u> the kernel
- Ensure safety of extensions by using a type-safe language and strong interfaces between system services

Core Spin Services

- Extensible memory management
- Extensible thread management
- A few others
- A few trusted internals (e.g., device drivers) written in C

Spin Extensions

- Written in Modula-3
  - a type-safe language
  - also supports strong interfaces and automatic memory management
- Typical extension makes calls on interfaces of the core services
- Since both extension and core services are in kernel space, potentially very fast

Spin Capabilities

- Kernel resources in Spin are referenced by capabilities
- A capability is an unforgeable reference to a resource
  - system object
  - interface
  - collection of interfaces
- Capabilities in Spin are Modula-3 pointers
  - Safety ensured at compile time

Spin Protection Domains

- Set of accessible names available to an execution context
- Unlike most systems, not provided by virtual addressing
- Protected by the language at compile time

Extending Spin

- Many extensions possible
  - Monitoring
  - Hints to system algorithms
  - Replacement of scheduler, etc.
- Extensions defined in terms of *events* and *handlers*
  - Event announces change in system state
  - Handler receives message
  - Extensions register handlers to match to events

Spin Performance

- Highly variable, of course
- But typically a little faster to vastly faster
  - Than either monolithic or micro-kernels
- For example, user-level thread Fork/Join
  - DEC OSF/1 1230 microseconds
  - Mach 338 microseconds
  - Spin 85 microseconds

The Exokernel

- Extensible OS being developed at MIT
- Fundamental idea is to expose very low level details of hardware to users
- Provide libraries to support their typical needs

The Exokernel Philosophy

- OSs typically do two basic things
  - multiplex hardware
  - provide useful abstractions of hardware
- Multiplexing hardware is good
- Providing abstractions of hardware is bad
  - no abstraction is always right
  - abstractions cost performance

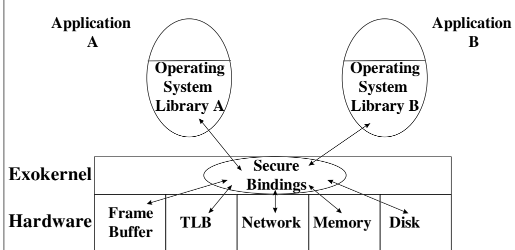
What's So Bad About Abstractions?

- Abstractions deny applications the advantages of domain-specific optimizations
- Abstractions discourage changes to existing implementation
- New abstractions can only be built on top of old ones, which can be clumsy

So Down With Abstractions!

- The Exokernel philosophy suggests the only kernel OS functionality should relate to hardware multiplexing
- All abstractions should be provided in libraries at the user level
- Key is to design good, cheap hardware access facilities

Basic Exokernel Architecture



How Is This Different Than Microkernels?

- It's the most extreme microkernel possible
  - Short of none
- ALL the exokernel does is multiplex hardware
- Actually, it's not quite no abstractions, but it's very, very few
- Compare bare (multiplexed) hardware to Liedtke's address spaces + threads + IPC + unique IDs

Services Provided by the Exokernel

- Tracking ownership of resources
- Guarding all resource usage or binding points
- Revoking access to resources

How Are These Services Provided?

- *Secure bindings* allow library operating systems to bind to machine resources
- *Visible revocation* allows library operating systems to participate in resource revocation
- *Abort protocols* allow exokernel to break bindings of uncooperative library operating systems

What Doesn't the Exokernel Do?

- It does not manage resources
  - No attempt to control who does what with the hardware
    - **IF** they have obtained control of it properly
- No imposition of higher-level abstractions
  - No signal or RPC semantics

Exokernel Performance

- An exokernel architecture can be very, very fast
  - No abstractions to get in the way of exactly what needs doing
- Question - how smart does the programmer have to be for it to go that fast?

Key Unanswered Questions in Exokernel

- Have you simply bumped all problems of OS design from the kernel to user space?
- Can ordinary mortals (or even highly skilled application programmers) write OS-type code well enough to actually get performance gains?
- Is this stuff at all portable?

Fluke

- Another extensible operating system from the University of Utah
- Concept is to marry virtual machines with stackable systems
- Add extensibility by stacking new layers (of virtual machines) on top of a minimal virtual machine

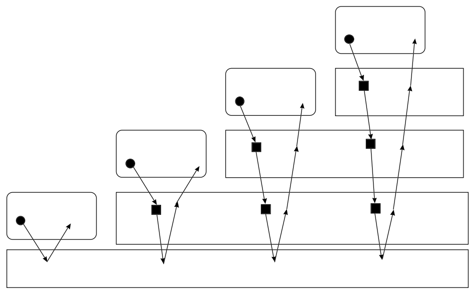
More on Virtual Machines

- Virtual machine operating systems provide higher level systems software the illusion of their own machine
- In principle, the virtual machine need not match the architecture of the physical machine
  - for fast operations, it should, most of the time

The Fluke Architecture

- A microkernel that provides a process model "machine"
- Ability to stack more virtual machines on top of the base one
- With efficient execution of both instructions and most system calls

The Stacking Problem



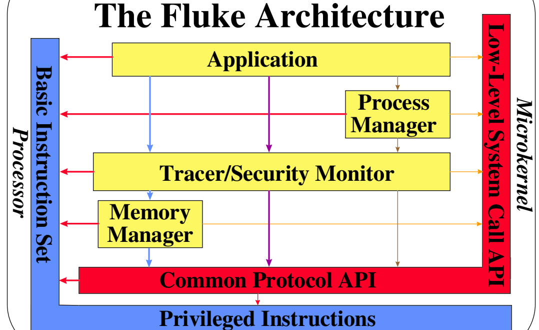
The Stacking Problem in the Virtual Machine Context

- If it’s truly a virtual machine, you may need to virtualize **<u>every</u> <u>instruction</u>**
- This is extermely expensive
- How do you get around the problem in Fluke?

Fluke’s Answers to Stacking Costs

- Don’t virtualize every instruction
  - only privileged instructions and system call
  - machine code mostly executes untouched
- Provide access at all levels to lowest level virtual machine system calls
  - if all you want is what the microkernel provides, skip going through the other layers

The Fluke Architecture



The Fluke Instruction Set

- A commonly used subset of the x86 instruction set
- Most compilers produce these instructions
- Privileged instructions (e.g., virtual memory controls) go through the microkernel
- If you use an unsupported x86 instruction, all bets are off

The Fluke Low-Level API

- Part of the microkernel
- A set of low-level objects and system calls
- Careful design maintains virtualizability of these calls
  - so any level of process can call them directly

What's in the Fluke μ-kernel?

- Address spaces
  - a very lightweight process/thread mechanism
- Thread objects
- Capabilities
  - References to kernel objects
  - Wrapped inside reference objects
  - Only the kernel sees the actual capability
- Scheduling primitives

Building Useful Services in Fluke

- Microkernel services too minimal for many purposes
- Higher level services can be built on top of microkernel
  - At any level above the microkernel
- Must match interfaces defined for Fluke, to ensure they allow interposition

Conclusions on OS Extensibility

- OS extensibility is still an open issue
- Highly differing, extreme positions guarantee fun times at workshops attended by proponents of various options
- Dig into the literature and decide for yourself!

64-Bit Address Space Machines

- An entirely different challenge for operating system architecture
- A fundamental assumption about our hardware is in the process of changing
- Should we change the OS to match it?

The Challenge of Wide Virtual Address Spaces

- When virtual memory was created, 16 or 32 bit addresses spaces were available
- Reasonable size for one process
  - but maybe not for all processes on a machine
  - and certainly not for all processes <u>ever</u> on a machine

Wide Address Space Architectures

- Computer architects can now give us 64 bit virtual addresses
- A 64 bit address space, consumed at 100 megabytes per second, lasts 5000 years
  - orders of magnitude beyond any process' needs
  - 40 bits can address a terabyte

Do We Care?

- Should OS designers care about wide address spaces?
- Well, what can we do with them?
- One possible answer
  - put all processes in the same address space
  - maybe all processes for all time?

Traditional Addressing Architectures

- Each process gets its own complete 32-bit address space
- Only that process can correctly "issue" its addresses
- Virtual memory hardware + OS software supports switching of virtual memories for different processes
- IPC must usually go through the OS

Implications of Single Shared Address Space

- IPC is trivial
  - Everybody agrees on the addresses of everything
  - Shared memory, RPC are thus simple
- Separation of concepts of address space and protection domain
- *Uniform-address space*

Address Space and Protection Domain

- A process has a protection domain
  - The data that cannot be touched by other processes
- And an address space
  - The addresses it can generate and access
- In standard systems, these concepts are merged

Separating the Concepts

- These concepts are potentially orthogonal
- Just because you can issue an address doesn't mean you can access it
- (Though clearly to access an address you must be able to issue it)
- Existing hardware can support this separation

Context-Independent Addressing

- Addresses mean the same thing in any execution context
- So a given address always refers to the same piece of data
- Key concept of uniform-address systems
- Allows many OS optimizations and improvements

Uniform-Addressing Allows Easy Sharing

- Any process can issue any address
  - So any data can be shared
- All that's required is changing protection to permit desired sharing
- Suggests programming methods that make wider use of sharing

The Opal System

- New OS using uniform-addressing
- Developed at U. of Washington
- Not intended as slight alteration to existing Unix system
- Other uniform-address operating systems are under development

Protection Mechanisms for Uniform-Addressing

- Protection domains are assigned portions of the address space
- They can allow other protection domains to access them:
  - Read-only
  - With transferable access permissions
  - With system-enforced page-level locking

Program Execution in Uniform-Access Memory

- Executing a program creates a new protection domain
- The new domain is assigned an unused portion of the virtual address space
- But it may also get access to used portions
- E.g., a segment containing the required executable image

Virtual Segments

- Global address space is divided into segments
  - Each composed of variable number of contiguous virtual pages
- Domains can only access segments they *attach* to
- Attempting to access unattached segment causes a segment fault

Persistent Memory in Opal

- *Persistent* segments exist even when attached to no current domain
- *Recoverable* segments are permanently stored
  - And can thus survive crashes
- All Opal segments can be persistent and recoverable
- Pointers can thus live forever on disk

Code Modules in Opal

- Executable code stored in *modules*
  - independent of protection domains
- *Pure modules* can be easily shared
  - essentially static
- Can get benefit of dynamic loading without run-time linking

Address Space Reclamation

- Trivial in non-uniform-address systems
- Tricky in uniform-address systems
- Problem akin to reclaiming inodes in the presence of (multiple) hard links
- But even if segments improperly reclaimed, only trusting domains can be hurt

Meta-level Summary Comments

- When exploring a new paradigm, push it to the (believed) limits
- Try separating joint concepts, or splitting a previously "atomic" idea into multiple parts