Operating Systems Architecture

CS239
Advanced Operating Systems
October 5, 2000
Dr. Richard Guy

Outline

- Part I: Why Study OS?
  - Operating Systems vs. Architecture
- Part II: What's the best way to build an OS?
  - Organizing operating systems
  - Some microkernel examples
- Part III: What's just over the horizon?
  - New architectures for extensibility
  - 64-bit address space architectures

Why Operating Systems?

Why do operating systems exist?

Are they really essential?

Part I: Why study?

Don't we already know how to build a pretty good operating system? Haven't the past 40 years covered all (or most) of the interesting ground?

Part I Source Materials

- Ousterhout, Why Aren't Operating Systems Getting Faster as Fast as Hardware?, 1990.
- Anderson et al, The Interaction of Architecture and Operating System Design, 1991.
- Rosenblum et al, The Impact of Architectural Trends on O.S. Performance, 1995.

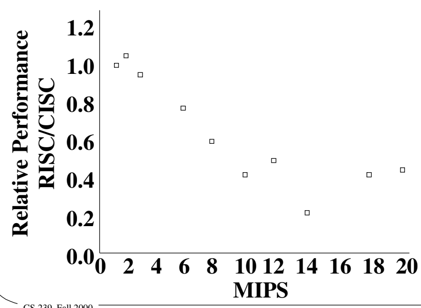
OS vs. Architecture

- "The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' (I found it!) but 'That's funny …'" -- Isaac Asimov
- … apparent OS/HW performance disparity
  - Is it really true?
  - If so, why?
  - and then, what could/should be done about it?

Ousterhout's Study

- One macrobenchmark, many microbench
  - MAB; kernel entry/exit, context switching, select, block copy, file cache read, open/close, file create/delete.
  - Six OSs on ten hardware architectures

Uninterpreted Results



Interpretation

- Hardware Issues
  - Memory bandwidth of CISC is 3-4x RISC
    - CPU rate increase >> mem/bus rate increase
  - Context switching cost of RISC is 2x > CISC
- Software Issues
  - Disk I/O speed increases lagging CPU speed
    - decouple file system perf. from disk perf.
    - non-volatile memory as buffer; log-struct. FS

Anderson et al

- CISC→RISC debate; other arch. trends
  - move to simple, directly-executed instructions
  - open-architecture (implementation visible)
  - function migration from hardware to software
  - performance must justify cost of feature
- OS trends
  - Unix OS interface/services is major standard
  - Unix changing under the covers in big ways

Independent Evolution

- Simulation/measurement guide h/w designs but frequently overlook the OS
  - early tracing tools unable to trace OS itself
  - microcode tools: 50% of memory refs to OS
  - OS behavior quite different from workloads
- Traditional Unix & apps no longer "right"
- OS optimized to outdated h/w features/perf.
- Critical low-level services moved m/c→s/w

Study overview

- Tested four OS primitives on 5 machines
  - MHz range: 11-25; relative perf. range: 0.5-6.7
  - primitives extensively used, and even more so
- Studied higher level services
  - IPC (cross-machine, local, syscalls/ints, copies)
  - Virtual Memory (faults, TLBs, page tables)
  - Threads/Multiprocessing (user-level threads)
- cf. Tables 2, 3, 5, 6, 7

Study Conclusions

- OSs are being decomposed into kernelized structures with independent servers executing in separate address spaces
- OSs are requiring more use of memory mgt, when memory management is more difficult
- OSs are moving towards support of fine-grained multithreaded application programs
- WARNING: bottlenecks on the horizon

Rosenblum et al

- Used full hardware simulation to analyze OS perf. on 1994, projected 1998 hardware
- Studied uni- and multi-processor perf. with programming, DB, and engineering loads
- Main results
  - I/O subsystem biggest bottleneck; getting worse
  - memory system: kernel stalled 50%; improving
  - synchronization costs increasingly important

Part II: OS Organization

What is the best way to organize/structure/design an operating system?

Part I Source Materials

- Singhal, Ch. 1: Operating System Overview
- Tanenbaum, Ch. 7: Amoeba [SEL Reserve]
- Tanenbaum, Ch. 8: Mach [SEL Reserve]
- Pike, Plan 9 from Bell Labs [CS239 Web]
- Stallings, Ch.2:Windows NT [SEL Reserve]

Question Assumptions

- at least one way to organize OS software
- more than one way to organize
- different ways can be effectively compared
  - requires comparison metric(s)
  - metrics must be valid for all subjects studied

OS Organization Issues

- Important OS software characteristics
- Typical OS components
- Component interactions
- Survey of OS organization structures used
  - description
  - pros/cons

Important OS Software Characteristics

- Correctness
- Simplicity
- Power & completeness
- Performance
- Extensibility & portability
- Suitability for distributed & parallel systems
- Compatibility with existing systems
- Security
- Fault tolerance

Typical OS Components

- Process/thread support
- Memory management
- I/O and device support
- Interprocess communications
- Security
- File system support
- Fault tolerance and recovery support

Putting the Pieces Together

- Many of these functions interact
- The totality of all these functions is highly complex
- How do we organize the overall operating system software to meet the goals?

Common OS Organizations

- Monolithic
- Virtual machine
- Layered designs
- Kernel designs
- Microkernels
- Object-Oriented

Note that individual OS components can be organized these ways

Monolithic OS Design

- Build OS as single combined module
  - Hopefully using data abstraction, compartmentalized function, etc.
- OS lives in its own, single address space
- Examples
  - DOS
  - early Unix systems
  - most VFS file systems

Pros/Cons of Monolithic OS Organization

+ Highly adaptable (at first . . .)
+ Little planning required
+ Potentially good performance
– Hard to extend and change
– Eventually becomes extremely complex
– Eventually performance becomes poor
– Highly prone to bugs

Virtual Machine Organizations

- A base operating system provides services in a very generic way
- One or more other operating systems live on top of the base system
  - Using the services it provides
  - To offer different views of system to users
- Examples - IBM's VM/370, the Java interpreter

Pros/Cons of Virtual Machine Organizations

+ Allows multiple OS personalities on a single machine
+ Good OS development environment
+ Can provide good portability of applications
– Significant performance problems
– Especially if more than 2 layers
– Lacking in flexibility

Layered OS Design

- Design tiny innermost layer of software
- Next layer out provides more functionality
  - Using services provided by inner layer
- Continue adding layers until all functionality required has been provided
- Examples
  - Multics
  - Fluke
  - layered file systems and comm. protocols

Pros/Cons of Layered Organization

+ More structured and extensible
+ Easy model
– Layer crossing can be expensive
– In some cases, multiple layers unnecessary

Kernel OS Designs

- Similar to layers, but only two OS layers
  - Kernel OS services
  - Non-kernel OS services
- Move certain functionality outside kernel
  - file systems, libraries
- Unlike virtual machines, kernel doesn't stand alone
- Examples - Most modern Unix systems

Pros/Cons of Kernel OS Organization

+ Many advantages of layering, without disadvantage of too many layers
+ Easier to demonstrate correctness
– Not as general as layering
– Offers no organizing principle for other parts of OS, user services
– Kernels tend to grow to monoliths

Microkernel OS Design

- Like kernels, only less so
- Try to include only small set of required services in the microkernel
- Moves even more out of innermost OS part
  – Like parts of VM, IPC, paging, etc.
- Examples - Mach, Amoeba, Plan 9, Windows NT, Chorus

Pros/Cons of Microkernel Organization

+ Those of kernels, plus:
+ Minimizes code for most important OS services
+ Offers model for entire system
– Microkernels tend to grow into kernels
– Requires very careful initial design choices
– Serious danger of bad performance

Object Oriented OS Design

- Design internals of OS as set of privileged objects, using OO methods
- Sometimes extended into application space
- Tends to lead to client/server style of computing
- Examples
  – Mach (internally)
  – Spring (totally)

Pros/Cons of Object Oriented OS Organization

+ Offers organizational model for entire system
+ Easily divides system into pieces
+ Good hooks for security
– Can be a limiting model
– Must watch for performance problems

Not widely used yet

Some Important Microkernel Designs

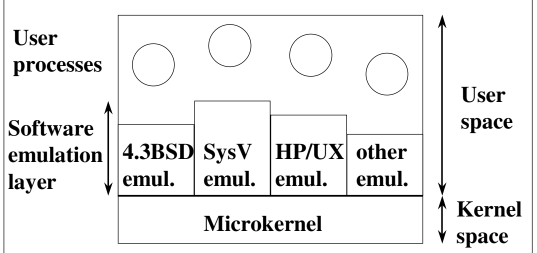
Micro-ness is in the eye of the beholder

- Mach
- Amoeba
- Plan 9
- Windows NT

Mach

- Mach didn’t start life as a microkernel
  – became one in Mach 3.0
- Object-oriented internally
  – doesn’t force OO at higher levels
- Microkernel focus is on communications facilities
- Much concern with parallel/distributed systems

Mach Model



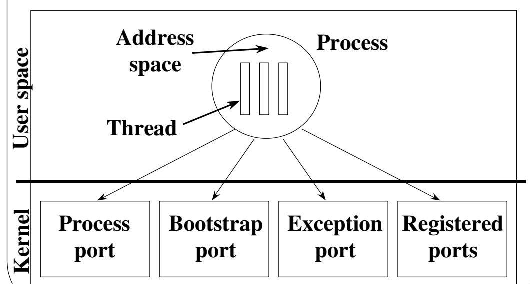
What's In the Mach Microkernel?

- Tasks & Threads
- Ports and Port Sets
- Messages
- Memory Objects
- Device Support
- Multiprocessor/Distributed Support

Mach Tasks

- An execution environment providing basic unit of resource allocation
- Contains
  – Virtual address space
  – Port set
  – One or more threads

Mach Task Model



Mach Threads

- Basic unit of Mach execution
- Runs in context of one task
- All threads in one task share its resources
- Unix process similar to Mach task with single thread

Task and Thread Scheduling

- Very flexible
- Controllable by kernel or user-level programs
- Threads of single task can execute in parallel
  – on single processor
  – multiple processors
- User-level scheduling can extend to multiprocessor scheduling

Mach Ports

- Basic Mach object reference mechanism
  - kernel-protected communication channel
- Tasks communicate by sending messages to ports
- Threads in receiving tasks pull messages off a queue
- Ports are location independent
- Port queues protected by kernel; bounded

Mach Messages

- Typed collection of data objects
  - Unlimited size
- Sent to particular port
- May contain actual data or pointer to data
- Port rights may be passed in a message
- Kernel inspects messages for particular data types (like port rights)

Mach Memory Objects

- A source of memory accessible by tasks
- May be managed by user-mode external memory manager
  - a file managed by a file server
- Accessed by messages through a port
- Kernel manages physical memory as cache of contents of memory objects

Mach Device Support

- Devices represented by ports
- Messages control the device and its data transfer
- Actual device driver outside the kernel in an external object

Mach Multiprocessor and Distributed System Support

- Messages and ports can extend across processor/machine boundaries
  - Location transparent entities
- Kernel manages distributed hardware
- Per-processor data structures, but also structures shared across the processors
- Intermachine messages handled by a server that knows about network details

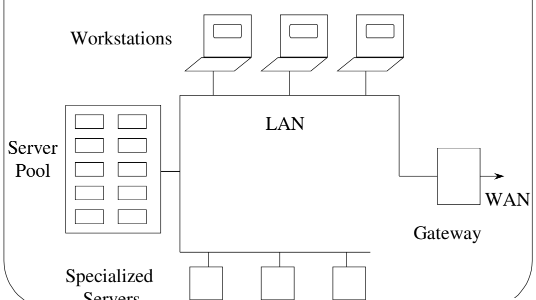
Mach and User Interfaces

- Mach was built largely for the community that uses Unix
- Unix programs don’t know about ports, messages, threads, and tasks
- So how do Unix programs run under Mach?
- Mach typically runs a user-level server that offers Unix emulation
- Either provides Unix system call semantics internally or translates it to Mach primitives

Amoeba

- Amoeba presents transparent distributed computing environment (a la timesharing)
- Major components
  – processor pools
  – server machines
  – X-terminals
  – gateway servers for off-LAN communications
- Microkernel runs everywhere

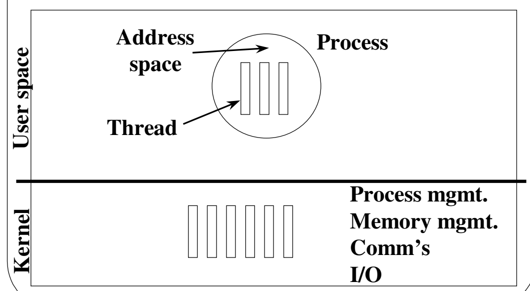
Amoeba Diagram



Amoeba's Basic Primitives

- Processes
- Threads
- Low level memory management
- RPC
- I/O

Amoeba Software Model



Amoeba Processes

- Similar to Mach processes
- Process has multiple threads
  – But a single shared address space
- Thread scheduling by microkernel

Amoeba Memory Management

- Amoeba microkernel supports concept of *segments*
- A segment is a set of memory blocks
- Segments can be mapped in/out of address spaces

Remote Procedure Call

- Fundamental Amoeba IPC mechanism
- Amoeba RPC is thread-to-thread
- Microkernel handles on/off machine invocation of RPC

Plan 9

- Everything in Plan 9 is a file system (almost)
  - Processes
  - Files
  - IPC
  - Devices
- Only a few operations are required for files
- Text-based interface

Plan 9 Basic Primitives

- Terminals
- CPU servers
- File systems
- Channels

File Systems in Plan 9

- File systems consist of a hierarchical tree
- Can be persistent or temporary
- Can represent simple or complex entities
- Can be implemented:
  - in the kernel as a driver
  - as a user level process
  - by remote servers

Sample Plan 9 File Systems

- Device file systems - Directory containing `data` and `ctl` file
- Process file systems - Directory containing files for memory, text, control, etc.
- Network interface file systems

Plan 9 Channels and Mounting

- A channel is a file descriptor
  - since a file can be anything, a channel is a general pointer to anything
- Plan 9 provides 9 primitives on channels
- Mounting is used to bring resources into a user's name space
- Users start with minimal name space, build it up as they go along

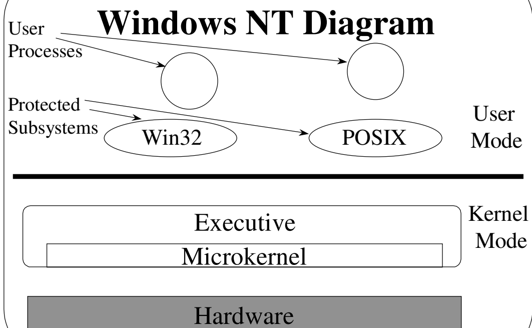
Typical User Operation in Plan 9

- User logs in to a terminal
  - provides bitmap display and input
- Minimal name space is set up on login
- Mounts used to build space
- Pooled CPU servers used for compute tasks
- Substantial caching used to make required files local

Windows NT

- More layered than some microkernel designs
- NT Microkernel provides base services
- Executive builds on base services via modules to provide user-level services
- User-level services used by
  - privileged subsystems (parts of OS)
  - true user programs

Windows NT Diagram



NT Microkernel

- Thread scheduling
- Process switching
- Exception and interrupt handling
- Multiprocessor synchronization
- Only NT part not preemptible or pageable
  - all other NT components runs in threads

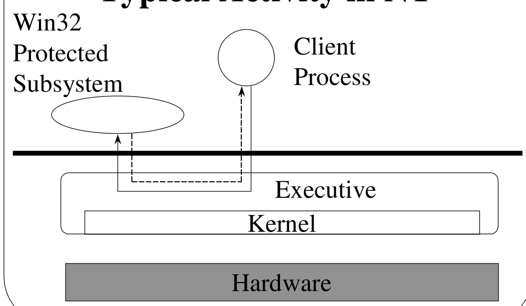
NT Executive

- Higher level services than microkernel
- Runs in kernel mode
  - but separate from the microkernel itself
  - ease of change and expansion
- Built of independent modules
  - all preemptible and pageable

NT Executive Modules

- Object manager
- Security reference monitor
- Process manager
- Local procedure call facility (a la RPC)
- Virtual memory manager
- I/O manager

Typical Activity in NT



Windows NT Threads

- Executable entity running in an address space
- Scheduled by kernel
- Handled by kernel's dispatcher
- Kernel works with stripped-down view of thread - *kernel thread object*
- Multiple process threads can execute on distinct processors--even Executive ones

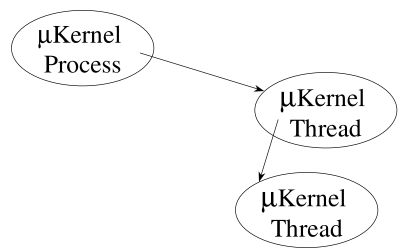
Microkernel Process Objects

- A microkernel proxy for the real process
- Microkernel's interface to the real process
- Contains pointers to the various resources owned by the process
  - e.g., threads and address spaces
- Alterable only by microkernel calls

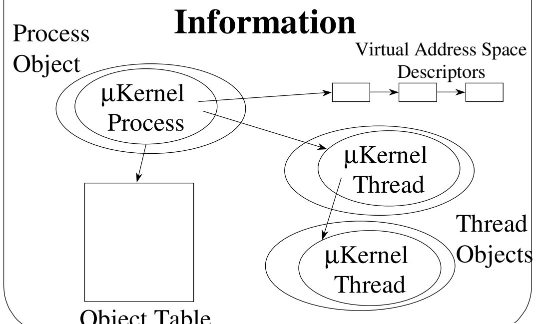
Microkernel Thread Objects

- As microkernel process objects are proxies for the real object, microkernel thread objects are proxies for the real thread
  - one per thread
- Contains minimal information about thread
  - Priorities, dispatching state
- Used by the microkernel for dispatching

Microkernel Process and Thread Object Diagram



Other Microkernel Process Information



More On Microkernels

- Microkernels were *the* research architecture of the 80s
- But few commercial systems of the 90s really use microkernels
- To some extent, “microkernel” is now a dirty word in OS design
- Why?