Course Introduction and Operating Systems Architecture CS239 Advanced Operating Systems March 31, 1997

CS239 - Advanced Operating Systems

Pre-requisites - CS111

Dr. Peter Reiher & Dr. Richard Guy

E-mail: reiher@cs.ucla.edu
rguy@cs.ucla.edu

Grading

- Mid-term - 25%
- Paper - 25%
- Final exam - 50%

Tests

Probably closed book

Probably a combination of short answer and essay questions

Midterm in the 5th week

Paper

Paper describing some important recent research on some operating system topic

Around 12-15 pages

1 paragraph paper proposals due at the end of the 4th week of classes

Submitted by email

Reading Materials

Textbook - **Advanced Concepts in Operating Systems**, by Singhal and Shivaratri

Supplemental papers assigned weekly

Most papers will be posted on WWW

http://www.cs.ucla.edu/reiher/CS239_spring97.html

Other papers will be put on reserve in archives

Background books - Tannenbaum's OS text and Silberschatz & Galvin's OS text

Office Hours

Formal hours set up if students need them

In 3732J Boelter Hall

Usually available by appointment

Both teachers will hold office hours

Usually only one teacher in given week

Co-Teaching Plans

Lectures delivered by one of the two teachers

Typically, lectures for a particular week will be given by one of the two teachers

Grading will be handled jointly

Class Outline

- OS architecture
- Distributed operating systems
- Advanced file systems
- Performance and measurement issues
- Fault tolerance and failure recovery
- Security
- OS's for multiprocessors and databases

Course Schedule

- Monday, March 31 - First day of class
- Friday, April 25 - Paper proposal due
- Wednesday, April 30 - Midterm
- Wednesday, June 4 - Papers due
- Friday, June 13 (11:30AM) - Final exam

Outline

- Organizing operating systems
- Some microkernel examples
- New architectures for extensibility
- 64-bit address space architectures

Operating System Organization

What is the best way to design an operating system?

Put another way, what are the important software characteristics of an OS?

Can operating systems be made safe and extensible, <u>and</u> perform well?

Important OS Software Characteristics

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

What's In an Operating System?

It varies, but typically:

- – 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?

Some OS Organizations

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

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 communications 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
1). Kernel OS services
2). Non-kernel OS services
Move certain OS functionality outside kernel
Like 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 part of OS

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

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 a dirty word in OS design

Why?

The Problem With Microkernels

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

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
  But doesn't force OO at higher levels
Microkernel concentrates on communications facilities
Much concern with parallel/distributed systems

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 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

Mach Task and Thread Scheduling

Very flexible

Controllable by kernel or user-level programs

Threads of single task can execute in parallel

- Either on single processor or on multiple processors

User-level scheduling can extend to multiprocessor scheduling

Mach Ports

Basic Mach object reference mechanism

- A 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 in Mach

Protected by kernel

Of bounded length

Sending message to a full queue can cause abort or blocking

Port Rights

Mechanism by which tasks control who may talk to their ports

Kernel prevents messages being sent to a port unless the sender has its port rights

Port rights also control which single task receives on a port

Port Sets

A group of ports sharing a common message queue

A thread can receive messages from a port set

- Thus servicing multiple ports

Messages are tagged with the actual port

A port can be a member of at most one port set

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

E.g., 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 provides connected distributed computing environments

Users sit at workstations controlled by Amoeba

Amoeba also controls processor pool for their use

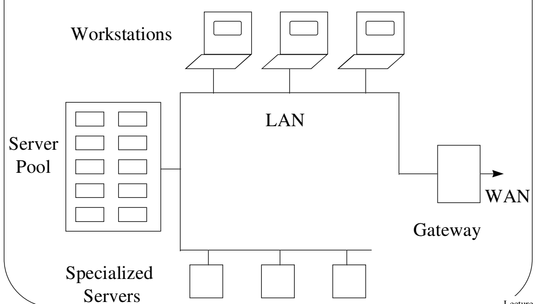
And special servers for special tasks

Gateway machines for off-LAN communications

Amoeba’s Basic Primitives

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

Amoeba Diagram



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

Amoeba Remote Procedure Call

Fundamental Amoeba IPC mechanism

Amoeba RPC is thread-to-thread

Microkernel handles on/off machine invocation of RPC

Amoeba Ports

Amoeba RPC addresses are called *ports*

  But they're not like Mach ports

Amoeba ports are simply addresses

  Randomly chosen 48 bit addresses

  Chosen by owning thread

Address-to-location mapping by broadcast

  With caching to minimize broadcast

Plan 9

Everything in Plan 9 is a file system (almost)

  Processes

  Files

  IPC

  Devices

And only a few operations are required for files

Plan 9 Basic Primitives

- Terminals
- CPU servers
- File systems
- Channels

File Systems in Plan 9

File systems consist of a hierarchical tree with no links

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

Which 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

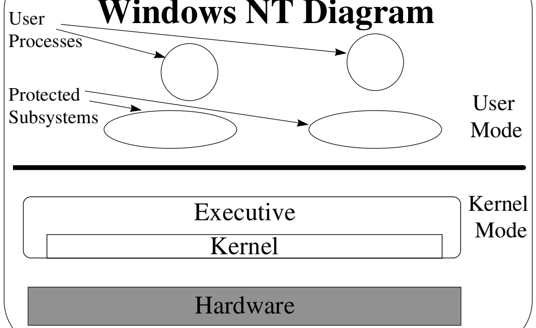
NT Microkernel provides base services

Executive builds on base services to provide user-level services

User-level services used by both privileged subsystems (parts of OS)

And true user programs

Windows NT Diagram



Windows NT Basic Primitives

- Basic object handling
- Dispatcher objects
- Control objects

Basic Object Handling Primitives

Primitives to create kernel objects

And manipulate them

Allows building of more complex objects for user-mode programs

Control Objects in Windows NT

- Kernel process object
- Asynchronous procedure call object
- Deferred procedure call object
- I/O subsystem objects
  – Interrupt object, e.g.

Dispatcher Objects in Windows NT

Handle scheduling and synchronization issues

- Kernel thread objects
- Kernel synchronization primitives
- Kernel timer

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*

Kernel Process Objects

Basically a kernel proxy for the real process

The kernel'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 kernel calls

Kernel Thread Objects

As kernel process objects are proxies for the real object, kernel thread objects are proxies for the real thread

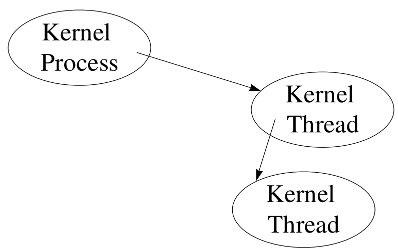
One per thread

Contains minimal information about a thread

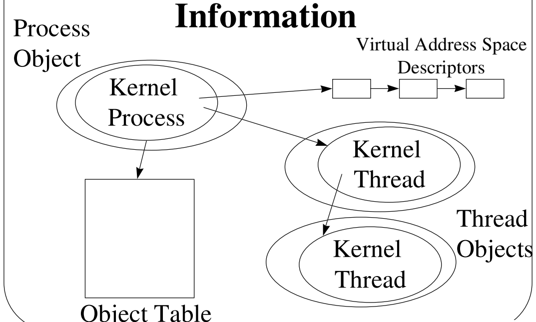
Priorities, dispatching state

Used by the kernel for dispatching

Kernel Process and Thread Object Diagram



Other Kernel Process Information



The Windows NT Executive

Typical microkernel approach to building total OS

Offers higher level services than the kernel

Runs in kernel mode

But separate from the kernel

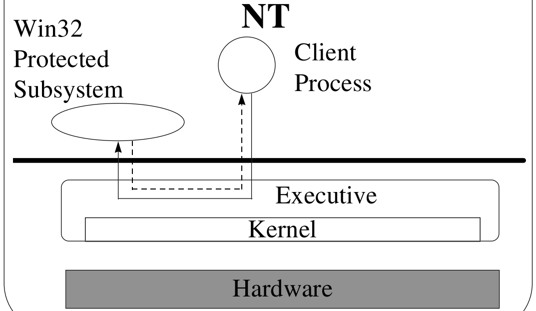
For ease of change and expansion

Built of independent modules

Windows NT Executive Modules

- Object manager
- Security reference monitor
- Process manager
- Local procedure call facility
- Virtual memory manager
- I/O manager

Typical Activity in Windows NT



So Why Haven't Microkernels Won?

Typical microkernels require lots of crossings of OS/user boundary to do anything useful

- Each crossing is pretty expensive
- So many microkernels perform like dogs

Or "microkernel" grows out of its micro-ness

- Question - are microkernels a bad idea, or did the early microkernel designers make bad choices?