Operating System Organization

CS 239, Spring 1995

May 30, 1995

What is the Best Way To Design An Operating System?

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

Decide on those, then design to match them

Important OS Software Characteristics

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

Some OS Organizations

- Monolithic organizations
- Layered organizations
- Kernel designs
- Microkernels
- Object-Oriented Organizations

Note that OS components can be organized these ways

Monolithic OS Design

Build OS as single combined module

Hopefully using data abstraction, compartmentalization of 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
+ Little planning required
+ Potentially good performance
– Hard to extend and change
– Eventually becomes extremely complex
– Eventually performance becomes poor
– Highly prone to bugs

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, 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

Sort of like 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

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 careful initial design choices
– 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
+ Which easily divides system into pieces
+ Good hooks for security

– Can be a limiting model

– Must watch for performance problems

More On 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

Some Important Microkernel Designs

- Mach
- Amoeba
- Plan 9
- Chorus
- 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

- Send 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

Essentially a capability

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 NetMsgServer

Mach's NetMsgServer

User-level capability-based networking daemon

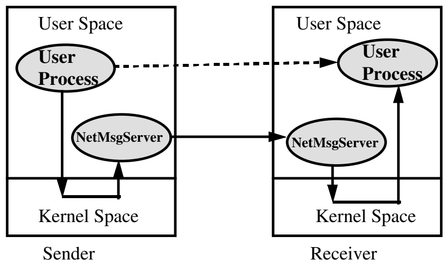
Handles naming and transport for messages

Provides network-wide name service for ports

Handles out-of-line data in messages

Messages sent to off-node ports go through this server

NetMsgServer in Action



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?

For that matter, can Mach run programs for other popular systems?

Mach and Unix Emulation

Mach typically runs a user-level server that offers Unix emulation

Either provides Unix system call semantics internally

Or translates it to Mach primitives

Multithreaded single server

But how do you get the system calls to the server?

The Unix Emulation Library

Special library linked to Unix applications

Translates Unix system calls into Mach messages

Which are sent to ports belonging to the Unix emulation server

What Do Other Microkernels Offer as Basic Services?

- Amoeba
- Plan 9
- Chorus

Amoeba's Basic Primitives

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

Amoeba's Basic Concept

Provide connected distributed computing environments

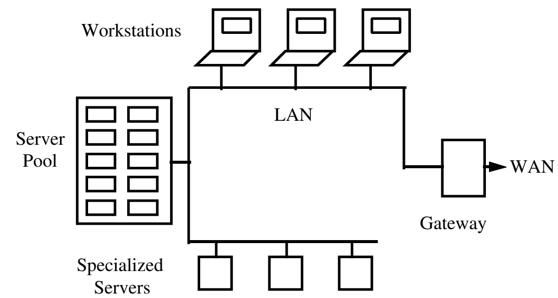
Users sit at workstations

Amoeba also controls processor pool for their use

And special servers for special tasks

Gateway machines for off-LAN communications

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

Like most RPC mechanisms, uses stubs to perform actual calls

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 Basic Primitives

- Terminals
- CPU servers
- Files systems
- Channels

Basic Concept in Plan 9

Everything's a file system (almost)

Processes

Files

IPC

Devices

And only a few operations are required for files

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, or 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

Chorus Basic Primitives

- Unique Identifiers
- Actors
- Threads
- Messages
- Ports and Port Groups
- Segments and Regions
- Capabilities and Protection Identifiers

Chorus Unique Identifiers

Chorus is an object oriented system

All objects have unique identifiers

An identifier service provides single-use, globally unique identifiers (across entire world)

UIs are 64 bit data structures

Any site can use UI to find an object

Local identifiers for purely local objects

Chorus Actors

Collection of Chorus resources

Defines protected address space in which threads can share those resources

An actor is tied to a single site

With no migration

Actor's resources are designated by local identifiers

Chorus Threads

Much like everyone else's threads

A thread belongs to a single actor

If their actor is running on a multiprocessor, true concurrency permitted

Threads within an actor can use shared memory primitives

Chorus Messages

Contiguous byte string logically copied from sender to receiver's address space

Depending on style of usage, various optimizations are allowed (e.g., copy-on-write)

Messages are addressed to ports

Chorus provides RPC and async messaging

Chorus Ports

Represent both address and queue

A port is attached to specific actor

But multiple threads can access a port

Ports can migrate between actors

Taking unconsumed messages with them

Ports are addressed with local or unique identifiers

Chorus Port Groups

- Collections of ports that can serve as message destinations
- Ports in a port group can belong to different actors
  - On different sites
- Ports can be members of multiple port groups

Chorus Segments and Regions

- Segments are pieces of memory
  - Either static or dynamic
- System actors called mappers manage segments
- Actor address spaces are divided into regions
  - Regions contain portions of a segment
- Regions' memory protected by access rights

Chorus Capabilities and Protection Identifiers

- Unique identifiers can be augmented with a server-dependent key to form a capability
- Capabilities used to securely access services
- The Chorus Nucleus stamps messages with protection identifiers
- Protection identifiers can be used for authentication of messages