CLASSIC OPERATING SYSTEMS: UNIX AND MACH

CS6410

Ken Birman

Unifying question for today

- What shoul … dern operating system?
  - Unix (n
  - Mach: Refocus the whole system on memory segments and sharing, message passing, c
  - Windows (not included): End user will program against .NET framework. Role of OS is to make .NET fast

> Simple process, file and stream abstractions. Often used directly by application developer or end-user.

> Mach hosts standard operating systems over these abstractions. The core system layer aims at a developer who works mostly on componentized CORBA-style applications.

> … so OS should use the hardware as efficiently as possible – end user will rarely if ever "see" the Win32/Win64 API! Offer powerful complete functionality to reduce frequency of "domain crossings"

Implicit claims?

- Unix: Operating systems were inelegant, batch-oriented, expensive to use. *New personal computing systems demand a new style of* OS.
- Mach: *Everything has become componentized, distributed.* Mach reimagines the OS for new needs.
- Windows: What matters more are end-users who work with IDEs and need to create applications integrated with powerful packages. Unix and Mach? Too low level. Focus on making OS fast, powerful.

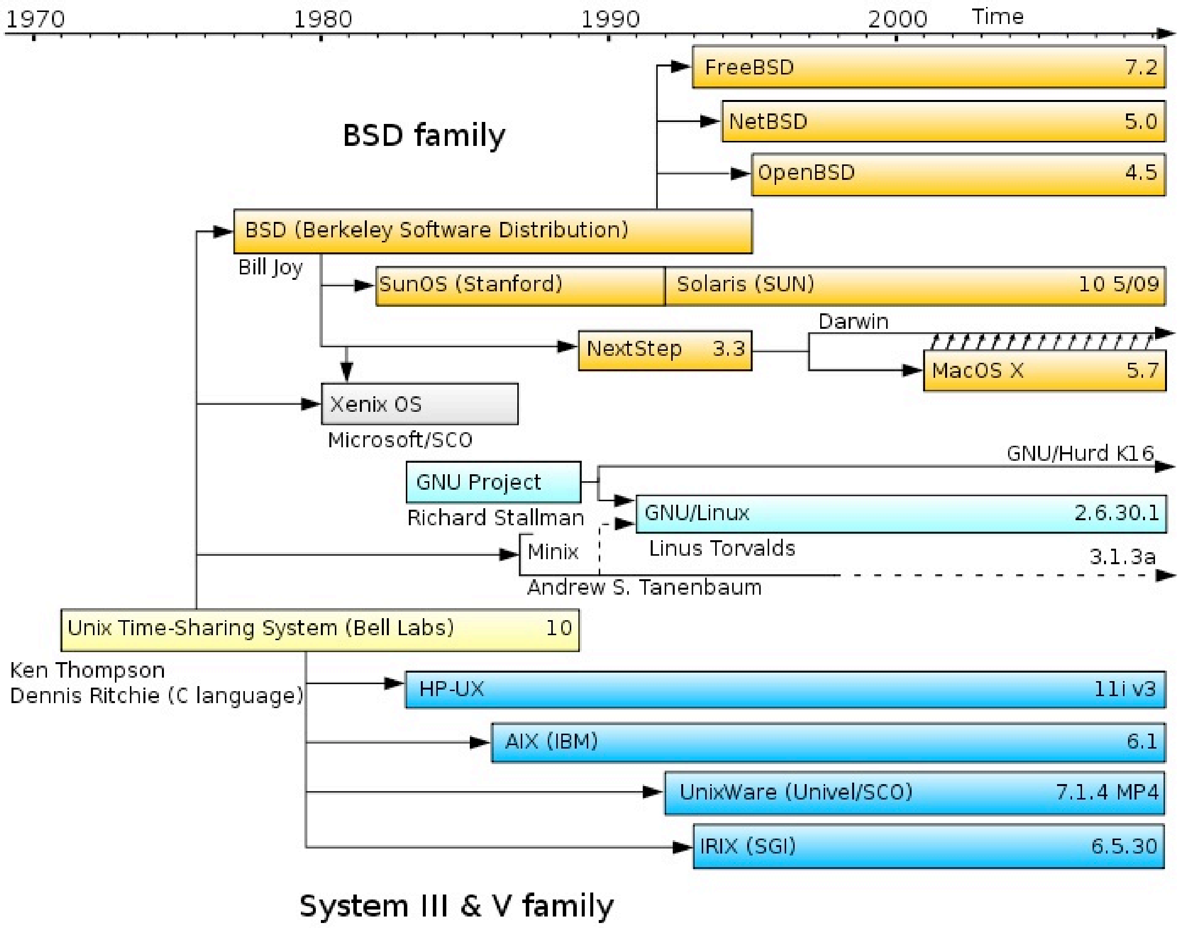
The UNIX Time-Sharing System Dennis Ritchie and Ken Thompson

- Background of authors at Bell Labs
  - Both won Turing Awards in 1983

- Dennis Ritchie
  - Key developer of The C Programming Lanuage, Unix, and Multics
- Ken Thompson
  - Key developer of the B programming lanuage, Unix, Multics, and Plan 9
  - Also QED, ed, UTF-8

Unix slides based on Hakim's Fall 2011 materials
Mach slides based on materials on the CMU website

The UNIX Time-Sharing System
Dennis Ritchie and Ken Thompson



The UNIX Time-Sharing System
Dennis Ritchie and Ken Thompson

- Classic system and paper
  - described almost entirely in 10 pages

- Key idea
  - elegant combination: a few concepts that fit together well
  - Instead of a perfect specialized API for each kind of device or abstraction, the API is deliberately small

System features

- Time-sharing system
- Hierarchical file system
- Device-independent I/O
- Shell-based, tty user interface
- Filter-based, record-less processing paradigm

- Major early innovations: "fork" system call for process creation, file I/O via a single subsystem, pipes, I/O redirection to support chains

Version 3 Unix

- 1969: Version 1 ran PDP-7
- 1971: Version 3 Ran on PDP-11's
  - Costing as little as $40k!
- < 50 KB
- 2 man-years

  to write
- Written in C

PDP-7

PDP-11

File System

- Ordinary files (uninterpreted)
- Directories (protected ordinary files)
- Special files (I/O)

Uniform I/O Model

- open, close, read, write, seek
  - Uniform calls eliminates differences between devices
  - Two categories of files: character (or byte) stream and block I/O, typically 512 bytes per block
- other system calls
  - close, status, chmod, mkdir, ln
- One way to “talk to the device” more directly
  - ioctl, a grab-bag of special functionality
- lowest level data type is raw bytes, not “records”

Directories

- root directory
- path names
- rooted tree
- current working directory
- back link to parent
- multiple links to ordinary files

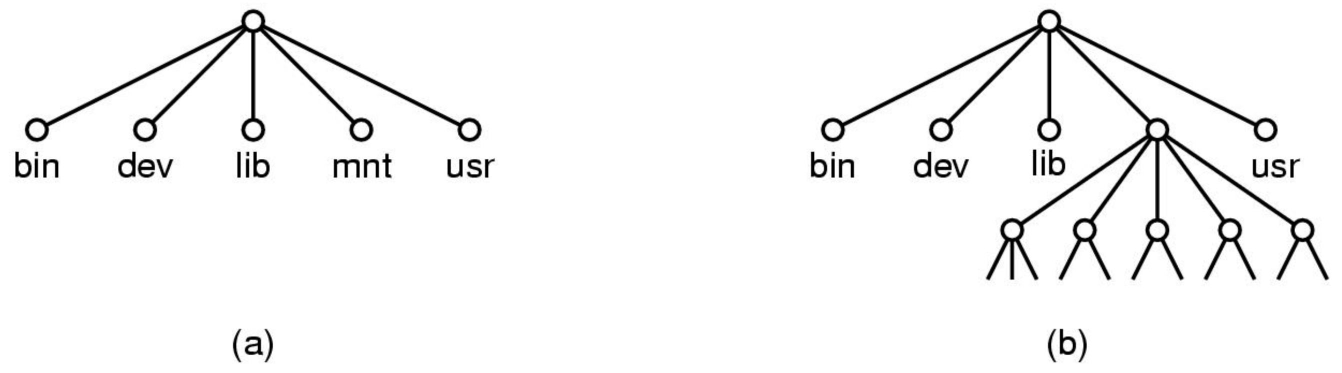
Special Files

- Uniform I/O model
  - Each device associated with at least one file
  - But read or write of file results in activation of device

- Advantage: Uniform naming and protection model
  - File and device I/O are as similar as possible
  - File and device names have the same syntax and meaning, can pass as arguments to programs
  - Same protection mechanism as regular files

Removable File System

- Tree-structured
- *Mount*'ed on an ordinary file
  - Mount replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume)
  - After mount, virtually no distinction between files on permanent media or removable media

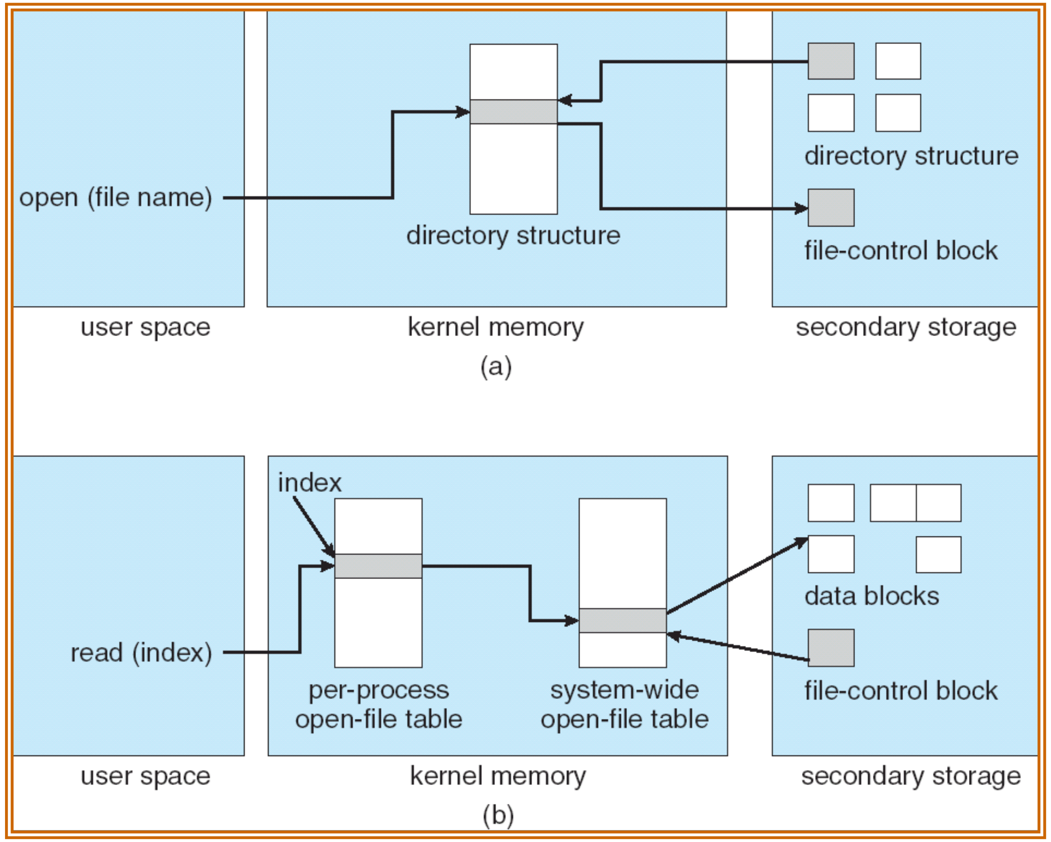


Protection

- User-world, RWX bits
- set-user-id bit
- super user is just special user id

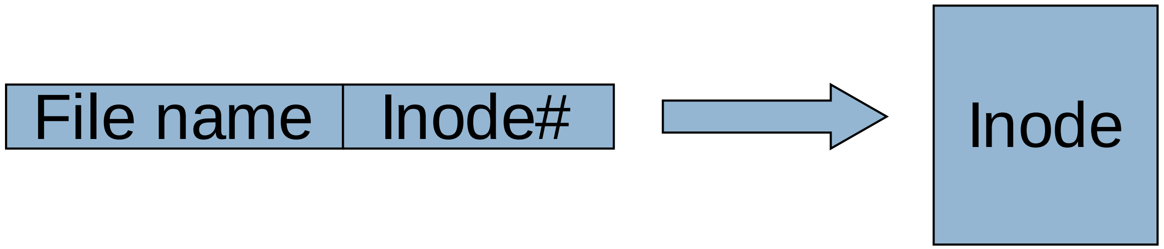
File System Implementation

- System table of i-numbers (i-list)
- i-nodes
- path names (directory is just a special file!)
- mount table
- buffered data
- write-behind



I-node Table

- short, unique name that points at file info.
- allows simple & efficient fsck
- cannot handle accounting issues



Many devices fit the block model

- Disks
- Drums
- Tape drives
- USB storage

- Early version of the ethernet interface was presented as a kind of block device (seek disabled)

- But many devices used IOCTL operations heavily

Processes and images

- text, data & stack segments
- process swapping
- pid = fork()
- pipes
- exec(file, arg1, ..., argn)
- pid = wait()
- exit(status)

Easy to create pipelines

- A “pipe” is a process-to-process data stream, could be implemented via bounded buffers, TCP, etc
- One process can write on a connection that another reads, allowing chains of commands

```
% cat *.txt | grep foo | wc
```

- In combination with an easily programmable shell scripting model, very powerful!

The Shell

- cmd arg1 … argn
- stdio & I/O redirection
- filters & pipes
- multi-tasking from a single shell
- shell is just a program

- Trivial to implement in shell
  - Redirection, background processes, cmd files, etc

Traps

- Hardware interrupts
- Software signals
- Trap to system routine

Perspective

- Not designed to meet predefined objective
- Goal: create a comfortable environment to explore machine and operating system
- Other goals
  - Programmer convenience
  - Elegance of design
  - Self-maintaining

Perspective

- But had many problems too. Here are a few:
  - Weak, rather permissive security model
  - File names too short and file system damaged on crash
  - Didn't plan for threads and never supported them well
  - "Select" system call and handling of "signals" was ugly and out of character w.r.t. other features
  - Hard to add dynamic libraries (poor handling of processes with lots of "segments")
  - Shared memory and mapped files fit model poorly
- ...in effect, the initial simplicity was at least partly because of some serious limitations!

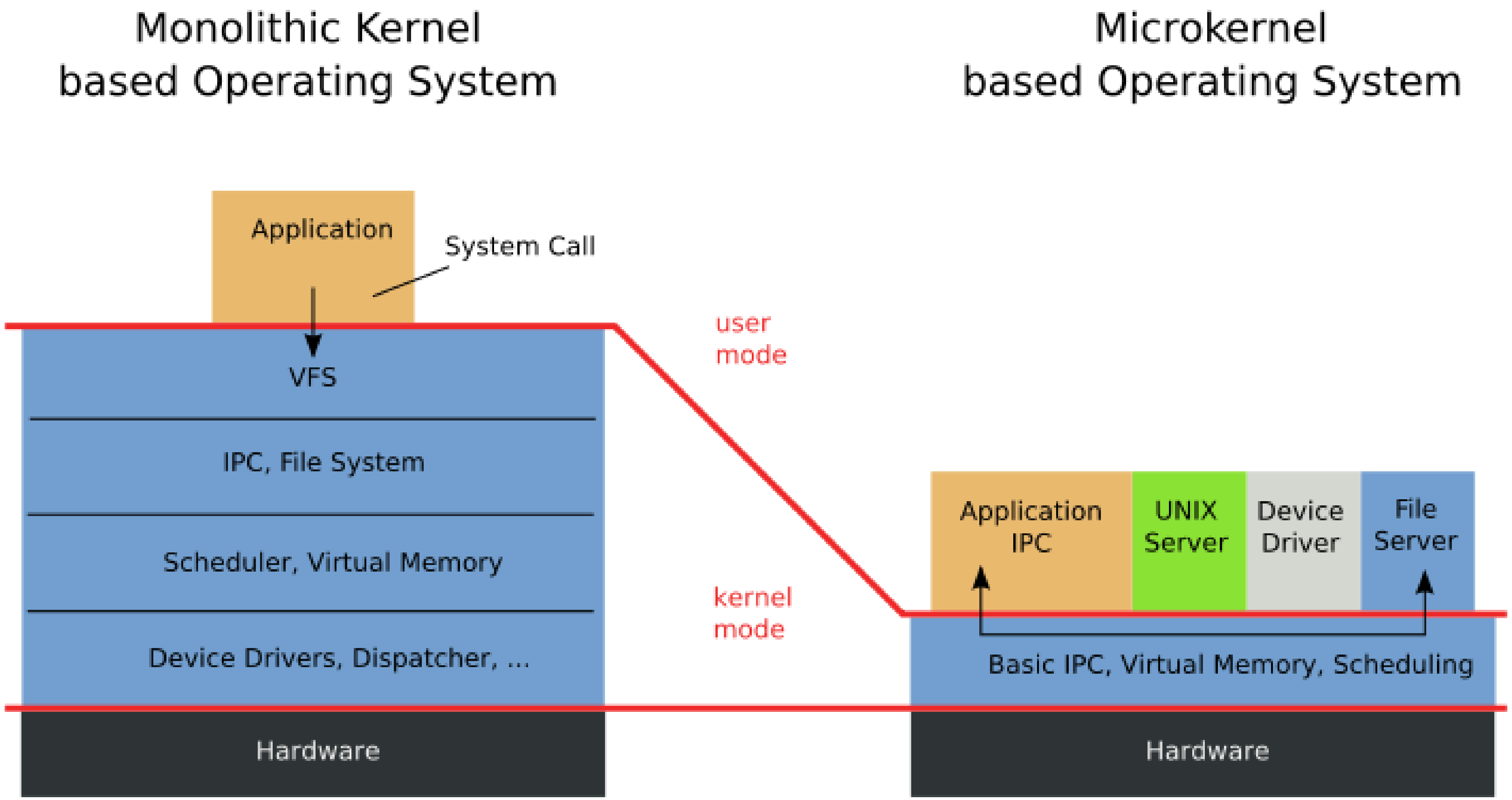
Even so, Unix has staying power!

- Today's Linux systems are far more comprehensive yet the core simplicity of Unix API remains a very powerful force
- Struggle to keep things simple has helped keep O/S developers from making the system specialized in every way, hard to understand
- Even with modern extensions, Unix has a simplicity that contrasts with Windows .NET API... Win32 is really designed as an internal layer that libraries invoke, but that normal users never encounter.

Linux gave rise to a (brief) μ-Kernel trend

- Even at outset we wanted to support many versions of Unix in one "box" and later, Windows and IBM operating systems too
  - A question of cost, but also of developer preference
  - Each platform has its merits
- Led to a research push: build a micro-kernel, then host the desired O/S as a customization layer on it
  - NOT the same as a virtual machine architecture!
  - In a μ-Kernel, the hosted O/S is an "application", whereas a VM mimics hardware and runs the real O/S

Microkernel vs. Monolithic Systems



Source: http://en.wikipedia.org/wiki/File:OS-structure.svg

Mach: Intended as a grown-up μ-Kernel

- CMU Accent operating system
  - No ability to execute UNIX applications
  - Single Hardware architecture
- BSD Unix system + Accent concepts
- Mach

OpenStep
GNU Hurd
XNU
OSF/1
Darwin
Mac OS X

Professor at Rochester, then CMU. Now Microsoft VP Research

Design Principles

Maintain BSD Compatibility

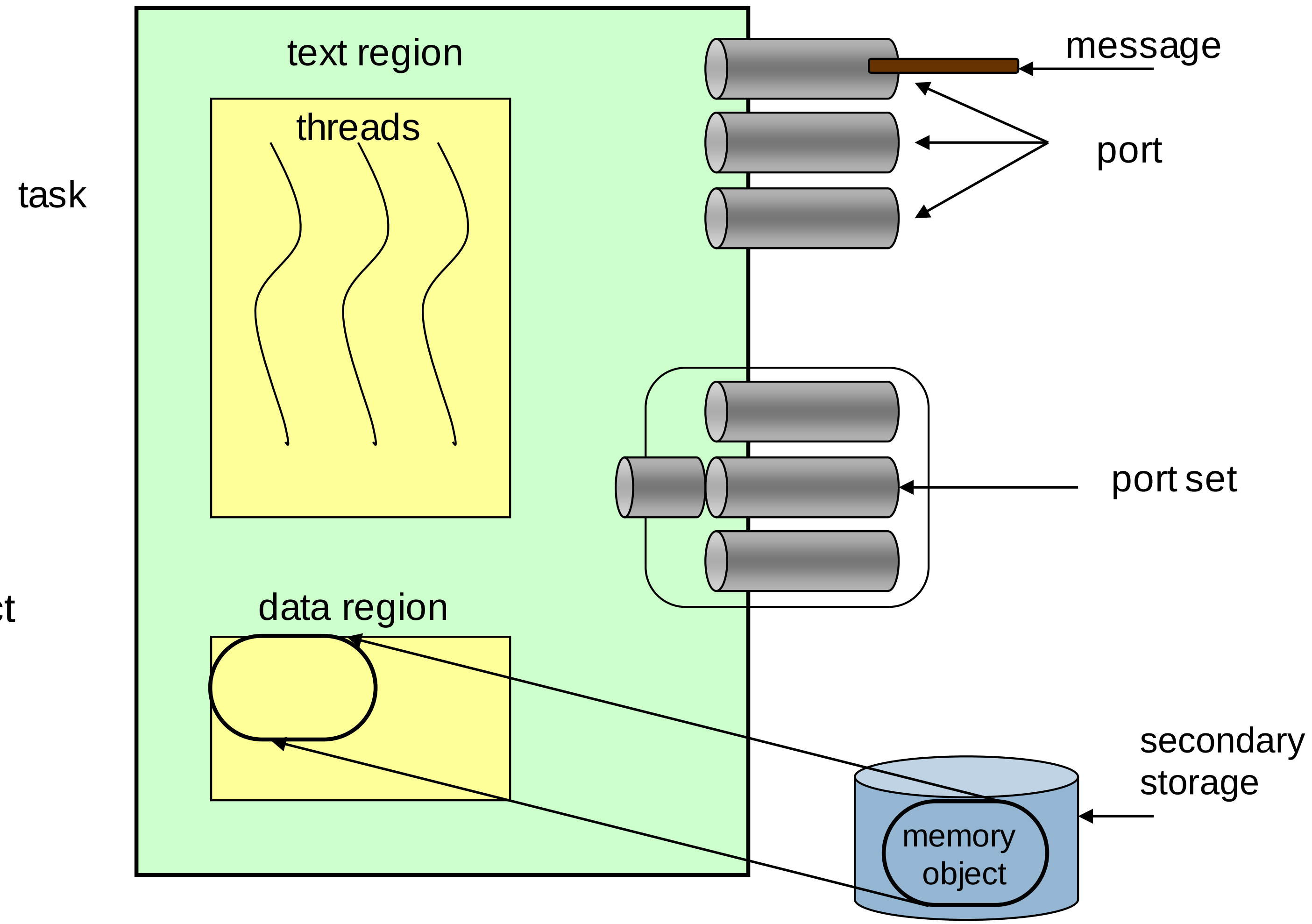
- Simple programmer interface
- Easy portability
- Extensive library of utilities/applications
- Combine utilities via pipes

PLUS

- Diverse architectures.
- Varying network speed
- Simple kernel
- Distributed operation
- Integrated memory management and IPC
- Heterogeneous systems

System Components

- Task
- Thread
- Port
- Port set
- Message
- Memory object



Memory Management and IPC

- Memory Management using IPC:
  - Memory object represented by port(s)
  - IPC messages are sent to those ports to request operation on the object
  - Memory objects can be remote → kernel caches the contents

- IPC using memory-management techniques:
  - Pass message by moving pointers to shared memory objects
  - Virtual-memory remapping to transfer large contents
    (virtual copy or copy-on-write)

Mach innovations

- Extremely sophisticated use of VM hardware
  - Extensive sharing of pages with various read/write mode settings depending on situation
  - Unlike a Unix process, Mach “task” had an assemblage of segments and pages constructed very dynamically
  - Most abstractions were mapped to these basic VM ideas, which also support all forms of Mach IPC

Process Management
Basic Structure

- Tasks/Threads
- Synchronization primitives:
  - Mach IPC:
    - Processes exchanging messages at rendezvous points
    - Wait/signal associated with semaphores can be implemented using IPC
    - High priority event-notification used to deliver exceptions, signals
  - Thread-level synchronization using thread start/stop calls

Process Management

C Thread package

- User-level thread library built on top of Mach primitives
- Influenced POSIX P Threads standard
- Thread-control:
  - Create/Destroy a thread
  - Wait for a specific thread to terminate then continue the calling thread
  - Yield
- Mutual exclusion using spinlocks
- Condition Variables (wait, signal)

Process Management
CPU Scheduler

- Only threads are scheduled
- Dynamic thread priority number (0 – 127)
  - based on the exponential average of its CPU usage.
- 32 global run queues + per processor local queues (ex. driver thread)
- No Central dispatcher
  - Processors consult run queues to select next thread
  - List of idle processors
- Thread time quantum varies inversely with total number of threads, but constant over the entire system

Process Management
Exception Handling

- Implemented via RPC messages
- Exception handling granularities:
  - Per thread (for error handling)
  - Per task (for debuggers)
- Emulate BSD style signals
  - Supports execution of BSD programs
  - Not suitable for multi-threaded environment

Interprocess Communication
Ports + messages

- Allow location independence + communication security
- Sender/Receiver must have *rights* (port name + send or receive *capability*)
- Ports:
  - Protected bounded queue in the kernel
  - System Calls:
    - Allocate new port in task, give the task all access rights
    - Deallocate task's access rights to a port
    - Get port status
    - Create backup port
  - Port sets: Solves a problem with Unix "select"

Interprocess Communication

Ports + messages

- Messages:
  - Header + typed data objects
  - Header: destination port name, reply port name, message length
  - In-line data: simple types, port rights
  - Out-of-line data: pointers
    - Via virtual-memory management
    - Copy-on-write
  - Sparse virtual memory

Interprocess Communication
Ports + messages

- NetMsgServer:
  - user-level capability-based networking daemon
  - used when receiver port is not on the kernel's computer
  - Forward messages between hosts
  - Provides primitive network-wide name service
- Mach 3.0 NORMA IPC
- Syncronization using IPC:
  - Used in threads in the same task
  - Port used as synchronization variable
  - Receive message → wait
  - Send message → signal

Memory Management

- Memory Object
  - Used to manage secondary storage (files, pipes, ...), or data mapped into virtual memory
  - Backed by user-level memory managers
- Standard system calls for virtual memory functionality
- User-level Memory Managers:
  - Memory can be paged by user-written memory managers
  - No assumption are made by Mach about memory objects contents
  - Kernel calls to support external memory manager
- Mach default memory manager

Memory Management
Shared memory

- Shared memory provides reduced complexity and enhanced performance
  - Fast IPC
  - Reduced overhead in file management
- Mach provides facilities to maintain memory consistency on different machines

Programmer Interface

- System-call level
  - Emulation libraries and servers
  - Upcalls made to libraries in task address space, or server
- C Threads package
  - C language interface to Mach threads primitives
  - Not suitable for NORMA systems
- Interface/Stub generator (*MIG*) for RPC calls

Mach versus Unix

- Imagine a threaded program with multiple input sources (I/O streams) and also events like timeouts, mouse-clicks, asynchronous I/O completions, etc.

- In Unix, need a messy select-based central loop.
- With Mach, a port-group can handle this in a very elegant and general way. But forces you to code directly against the Mach API if the *rest* of your program would use the Unix API

Mach Microkernel
summary

- Simple kernel abstractions
  - Hard work is that they use them in such varied ways
  - Optimizing to exploit hardware to the max while also matching patterns of use took simple things and made them remarkably complex
  - Even the simple Mach “task” (process) model is very sophisticated compared to Unix
- Bottom line: an O/S focused on communication facilities
- System Calls:
  - IPC, Task/Thread/Port, Virtual memory, Mach 3 NORMA IPC

Mach Microkernel

summary

- User level
  - Most use was actually Unix on Mach, not pure Mach
  - Mach team build several major servers
    - Memory Managers
    - NetMsgServer
    - NetMemServer
    - FileServer
  - OS Servers/Emulation libraries
  - C Threads user-level thread management package

Big picture questions to ask

- *Unix focuses on a very simple process + I/O model*
- *Mach focused on a very basic / general VM model, then uses it to support Unix, Windows, and "native" services*

- If Mach mostly is a VM infrastructure, was this the best way to do that? If Linux needed to extend Unix, was Unix simplicity as much of a win as people say?

- Did Mach exhbit a mismatch of goals: a solution (fancy paging) in search of a platform using those features?

- Fate of Mach: The system lived on and became Apple OS/X, and some ideas are still present in Windows, notably treating files as VM segments