History of Operating Systems
In ENIAC, human labor is the operating systems
Evolution of Operating Systems

- **Phase 1:** Hardware is expensive, humans are cheap
  - User at console
  - Single-user systems, batching systems, multi-programming systems
  - Single user systems
    - OS = loader + libraries of common subroutines
    - Problem: low utilization of expensive components

- **Phase 2:** Hardware is cheap, humans are expensive
  - Time sharing: Users use cheap terminals and share servers

- **Phase 3:** Hardware is very cheap, humans are very expensive
  - Personal computing: One system per user
  - Distributed computing: lots of systems per user

- **Phase 4:** Ubiquitous computing
  - Mobile computing
  - Embedded computing
  - Internet-of-Things
Open Shop Model

- Key OSs
  - IBM 701
  - TX-0

- Each user was allocated a minimum 15-min slot
  - 10 min for setting up equipment
  - 5 min for computation

- Wasted computer time of IBM 701 was $146,000 per month in 1954 dollars!
Batch Systems

- Emergence of fast input device
  - Magnetic tapes

- Operators collect punch cards and transfer them to magnetic tapes using satellite computers

- How can you improve batch process based on this technical advancement?
Batch Systems

- Using multiple tape readers makes computers work even at tape preparation time
- OS = loader + libraries of common subroutines
Multiprogramming

- Technical advancement in early 1960s
  - Large core memories
  - Secondary storage (drums) with random access
  - Data channels
  - Hardware interrupts
- Multiprogramming is a natural consequence considering these technologies
- Spooling was first used in this era
  - Simultaneous Peripheral Operation On-Line
  - For output spooling
Multiprogramming

- IBM MFT - ‘Multiprogramming with Fixed Tasks’
  - fixed partition boundaries
- IBM MVT - ‘Multiprogramming with Variable Tasks’
  - dynamic partition sizes and positions
  - OS scheduled jobs to minimize memory wastage
- Demand paging
  - First used in ATLAS OS
  - Core memory of 16 K and a drum of 96 K words
  - Core store is divided into 512 word pages
  - Drums and magnetic tapes also use 512 word fixed blocks
- ATLAS first used supervisor calls known as “extracodes”
Complications by Multiprogramming

- Jobs had to run at different places in memory.
- Inefficient use of expensive memory.
- One job could interfere with another e.g. overwrite other jobs memory.
- Jobs could clash over use of I/O devices.
Solutions to Complications

- Relocating linkers and loaders
- Hardware relocation base and limit registers
  - Read/write/execute protected access
- OS controlled access to I/O devices
- OS finally got complex

Core memory, the origin of “core dump”
Time Sharing

- John McCarthy at MIT proposed original idea of time sharing in 1959
- CTSS (compatible time-sharing system) developed at MIT in 1962 was the first time sharing OS
  - Multilevel feedback queue scheduling
  - Notion of background and foreground processes
- Multics File System (released in 1969)
  - Concept of general purpose file system
  - Hierarchical file systems for private and shared files
UNIX

- Developed by Dennis Ritchie and Ken Thompson at Bell Labs for PDP-11 in 1969
- Efficient design
- Extendible interface
- Publicly open source code
- De-facto standard time sharing operating system
Concurrent Programming

- What are required for concurrent programming?
  - Communications
  - Synchronizations

- THE system developed by Dijkstra in 1968
  - Semaphores for synchronization and communication

- Complexity became a serious problem

- RC 4000 developed by Per Brinch Hansen in 1969
  - Separation of mechanism and policy
Personal Computing

- Personal computer emerged in 1970s
- Xerox PARC was the leader in personal computing
  - Bit-mapped display
  - Mouse
  - Laser printer
  - Ethernet
  - Pilot OS
  - Star User Interface
    - Metaphor of office desk
Design of Modern OSs
Operating System Design Paradigms

- Monolithic kernel
- Microkernel
- Hybrid Kernel
- Exokernel
- Virtual Machines
Monolithic Kernel

- All OS services operate in kernel space
- Good performance

Disadvantages
- Dependencies between system component
- Complex & huge (millions(!) of lines of code)
- Larger size makes it hard to maintain

Examples
- Multics, Unix, BSD, Linux
Microkernel

- Minimalist approach
  - IPC, virtual memory, thread scheduling
- Put the rest into user space
  - Device drivers, networking, file system, user interface
- More stable with less services in kernel space
- Disadvantages
  - Lots of system calls and context switches
  - Slow inter-process communication
- Examples
  - Mach, L4, AmigaOS, Minix, K42
Hybrid Kernels

- Combine the best of both worlds
  - Speed and simple design of a monolithic kernel
  - Modularity and stability of a microkernel
- Still similar to a monolithic kernel
  - Disadvantages still apply here
- Examples
  - Windows NT, NetWare, BeOS
Exokernel

- Follows end-to-end principle
  - Extremely minimal
  - Fewest hardware abstractions as possible
  - Just allocates physical resources to apps

- Disadvantages
  - More work for application developers

- Examples
  - Nemesis
  - ExOS
The Mach System
How Mach Started

- Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University.
  - Communication system and philosophy are derived from Accent.
- Unlike Accent, Mach is:
  - Able to execute UNIX Applications.
  - Not tied to any one architecture.
How Mach Started (Continued)

- Mach code was first developed inside 4.2BSD
  - Mach components replaced BSD ones as they were completed
- Mach 3 moves BSD code outside kernel
  - Microkernel
  - Allows replacement of BSD with another OS
- Or, the simultaneous execution of multiple operating-system interfaces on top of the microkernel
Goals of Mach

- **Compatibility with UNIX**
  - Mach is compatible with UNIX 4.3BSD

- **Support diverse architectures**
  - Varying number of processors (to thousands)
  - Varying degrees of shared memory access

- **Simplified kernel structure**
  - Small number of abstractions
  - Minimize code within the kernel
  - Make the code powerful enough that all other features can be implemented at user level
Mach’s Benefits

- Simple kernel structure and abstractions
  - General enough to allow other operating systems to be implemented on top of Mach
  - Avoids having too many competing ways to perform the same task

- Example of this simplification:
  - All requests to the kernel, and all data movement among processes, are handled through one communication mechanism
    - Mach is able to provide system wide protection by protecting the communications mechanism
    - Optimizing this communications path can increase performance, and is simpler than optimizing several paths
Mach’s Primitive Abstractions

- Task (Execution Environment)
- Thread (Unit of Execution)
- Port (Object Reference Mechanism)
- Port Set
- Message (Thread Communication)
- Memory Object (Source of Memory)
Mach’s Primitive Abstractions

Figure B.2  Mach’s basic abstractions.
Mach Primitives - Tasks

- A **task** is an execution environment that provides the basic unit of resource allocation
  - Virtual address space
  - Protected access to system resources via ports
  - A task can contain 1 or more threads
  - States: **Running, Suspended** (explained on next slide)

- An operation on a task affects all threads in a task
  - Suspending a task suspends all the threads in it
  - Task and thread suspensions are separate, independent mechanisms
    - Resuming a thread in a suspended task does not resume the task

- A **task** can be thought of as a traditional process that does not have an instruction pointer or a register set. (but the task does nothing without 1+ threads)
Mach Primitives - Threads

- A thread is the basic unit of execution
  - Must run in a task (which provides the address space)
  - All threads within a task share the tasks’ resources
    - Ports
    - Memory

- States:
  - **Running:**
    - Thread is executing
    - Waiting to be given a CPU
    - A thread is considered to be running even if it is blocked within the kernel (a page fault, etc.)
  - **Suspended:**
    - Thread is not executing
    - Not waiting to be given a CPU
    - The thread can resume only if it is returned to the “running” state
A **port** is the basic object reference mechanism in Mach

- It is a kernel-protected communication channel
  - Communication: sending messages to ports
  - A message is queued at the destination port if no thread can receive it
  - Ports are protected by *port rights* *(required to send message)*

- The programmer invokes an operation on an object by *sending* a message to a port associated with that object

- The object being represented by a port receives the messages

- A **port set** is a group of ports sharing a common message queue
  - A thread can receive messages for a port set, and thus *service multiple ports*
    - Each received message identifies the individual port (within the set) that it was received from; the receiver can use this to identify the object referred to by the message
Ports (Continued)

- A port is a protected, bounded queue within the kernel. If full, a sender may abort the send, wait for a slot to become available, or have the kernel deliver the message for it.

- Allocate a new port for a task
  - Task given all access rights to the port
  - Port name is returned

- Deallocate rights to a port
  - If task is destroyed that is receiving
    - Destroy port, all other sending to that port (potentially) notified

- Ports created by the kernel for a new task:
  - Task_self – handle’s the task’s kernel calls.
  - Task_notify – receives notification messages for the task (eg. If a port is closed)

- *Mach ports can be transferred only in messages
Port Security

- Mach ensures security by requiring that message senders and receivers have rights
  - A port name
  - A capability (send or receive) on that port
  - Only one task with receive rights to any given port
    - Allows IPC to be used for synchronization (passing a resource with messages)
  - Many tasks may have send rights

- When an object is created
  - New port to represent the object
  - Creator obtains the access rights
    - Rights can be given out by the creator, and are passed in messages
      - If the holder of a receive right sends that right in a message, the receiver of the message gains the right and the sender loses it.

- A task may allocate ports
  - To allow access to any objects it owns, or for communication

- The destruction of either a port or the holder of the receive right causes
  - Revocation of all rights to that port
  - Tasks holding send rights can be notified
Mach Primitives - Messages

- A message is the basic method of communication between threads in Mach
  - Typed data object(s)
    - Actual data
    - A pointer to out-of-line data
    - Port rights
      - Passing port rights in messages is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)
Mach Primitives - Messages

Figure B.5  Mach messages.
The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel’s computer:

1. Mach’s kernel IPC to the local NetMsgServer
2. Local NetMsgServer to remote NetMsgServer by an appropriate protocol
3. Remote NetMsgServer uses that kernel’s IPC to send the message to the correct destination task

As a security precaution, a port value provided in an add request must match that in the remove request for a thread to ask for a port name to be removed from the database.
Figure B.7  Network IPC forwarding by NetMsgServer.
A memory object is a source of memory; tasks may access it by mapping portions (or the entire object) into their address spaces.

- May be managed by a user-mode external memory manager.
  - Example: a file managed by a file server.

- A memory object can be any object for which memory-mapped access makes sense.
Mach Primitives – Memory Objects

- A secondary-storage object is usually mapped into the virtual address space of a task
- Thread access -> fault: kernel sends a memory object data request message to the memory object’s port
  - The thread is placed in wait state until the memory manager either
    - Returns the page in a memory object data provided call
    - Returns an appropriate error to the kernel
  - …Meaning memory objects can be created and serviced by non-kernel tasks
- The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage.
Memory and IPC Integration

- **IPC in Memory**
  - Object is represented by a port (or ports), and IPC messages are sent to this port to request operations.

- **Memory in IPC**
  - Where possible, Mach passes messages by moving pointers to shared memory objects, rather than by copying the object itself.
C Threads Package

- Create a new thread within a task
  - Runs concurrently with the calling thread; calling thread receives a thread ID
- Destroy the calling thread, and return a value to the creating thread
- Wait for a specific thread to terminate before allowing the calling thread to continue
  - This is a synchronization tool
- Yield use of a processor; increases efficiency
- Mutual Exclusion in C Threads:
  - `mutex alloc`
  - `mutex free`
  - `mutex lock`
  - `mutex unlock`
General synchronization without busy waiting can be achieved through the use of condition variables, which can be used to implement a monitor.

- **Condition_alloc**
- **Condition_free**
- **Condition_wait** unlocks the associated mutex variable
  - Blocks the thread until a condition signal is executed on the condition variable
    - Mutex variable is then locked, and the thread continues
- **condition_signal**
  - Does not guarantee that the condition still holds when the unblocked thread finally returns from its condition wait call, so the awakened thread must loop, executing the condition wait routine until it is unblocked and the condition holds.
Mach’s CPU Scheduler

- The CPU scheduler for a thread-based multiprocessor OS is more complex than a process-based one
  - Generally more threads in a multithreaded system than there are processes in a multitasking system
  - Tracking multiple processors is difficult
- Mach uses a simple policy to keep the scheduler manageable:
  - Only threads are scheduled, so no knowledge of tasks is needed in the scheduler
  - All threads compete equally for resources, including time
- Each thread has a priority number ranging from 0 through 127, which is based on its CPU usage
  - A thread that recently used the CPU for a long time has the lowest priority
  - Priority places the thread in one of 32 global run queues
    - These queues are searched in priority order for waiting threads when a processor becomes idle
  - Mach also keeps per-processor, or local, run queues
    - Used for threads that are bound to an individual processor
Mach’s CPU Scheduler (Continued)

- Each processor consults the run queues to select the next thread to run
  - Threads in the local run queue have priority over those in the global queues
- Run queues are locked when they are modified to avoid corruption
- Mach maintains a list of idle processors to serve the global run queue
- Mach varies the size of the interrupt timing inversely with the total number of threads in the system
  - More threads? Shorter time between interrupts
- No need to interrupt if fewer threads than processors...
- Disruptions come in two varieties:
  - Internally generated exceptions
    - Asynchronously generated disruptions of a thread or task
  - External interrupts
    - Unusual conditions during a thread’s execution
Memory

- The virtual address space of a task is generally sparse
  - Holes of unallocated space
- Mach makes no attempt to compress the address space
- A task may fail (crash) if it has no room for a requested region in its address space
  - Memory is cheap
- Page-table space is used for only currently allocated regions
- Page fault:
  - Kernel checks to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry
    - Reduced memory-storage requirements
    - Simpler address-space maintenance
Memory Managers

Responsible for the consistency of the contents of a memory object mapped by tasks on different machines

- Useful for:
  - Maintaining consistency of secondary storage for threads on multiple processors
  - Controlling the order of operations on secondary storage, to enforce consistency constraints demanded by database management systems

- Insufficient in cases where:
  - A task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object
Shared Memory

- All threads in a task share that task’s memory, so no formal shared-memory facility is needed within a task.
- Parent task can declare memory regions that can be inherited by its children for reading and writing.
  - All changes are made to the same copy.
  - Threads handle synchronization.
Present and Future of OSs
Distributed OSs

- It is cheaper to buy multiple ordinary computers than buy a super computer

- Basic philosophy
  - Let’s group multiple computers and use them like one

- Single system image
  - Although it consists of multiple nodes, it appears to users and applications as a single-node

- Kernel must control local resource while providing global services by collaborating with other nodes
Fig. 1-14. General structure of a multicomputer operating system.
Fig. 1-19. General structure of a network operating system.
Message Passing VS Shared Memory

![Diagram showing the difference between message passing and shared memory architectures.](image-url)
Distributed File Systems

- Provide single file system from multiple storage nodes

- Design goals
  - Access transparency
  - Location transparency
  - Concurrency transparency
  - Failure transparency
  - Heterogeneity
  - Scalability
Mobile-Cloud Computing

- Combination of cloud computing, mobile computing and wireless networks
- To bring rich computational resources to mobile users, network operators
IoT Operating Systems

- Collaboration among intelligent things
- Tiny low-cost operating system design
- What will come?