OS Structures

——from Monolithic kernal to Microkernal

Presenter:

Zhenhong GUO, Shihan QIAO, Zitao LIAO

Introduction

What is OS Structure?

- Determines how OS is built, manages resources, serves applications
- Key trade-offs: Performance vs. Stability vs. Flexibility
- Today's Focus:
  - UNIX: Practical Monolithic Kernel
  - THE: Rigorous Layered System
  - Microkernal and Mach as an example

UNIX – The Practical Monolith

Design Philosophy

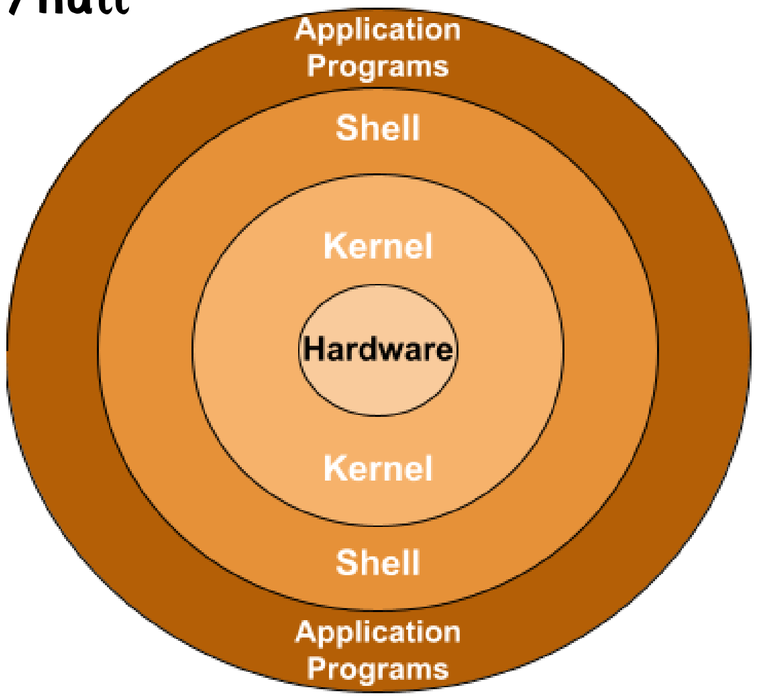
- "Everything is a file" – unified abstraction
- "Mechanism, not policy" – provide building blocks
- "Small is beautiful" – compose simple tools
- Programmer-friendly above all

"Do one thing, and do it well."

UNIX File System Structure

Unified Storage Abstraction

- File Types:
  - Regular files: unstructured byte streams
  - Directories: (filename → i-number) mappings
  - Special files: hardware as files in /dev ⟶ "/dev/null"
- Hierarchical Structure:
  - Tree organization with path names
  - mount seamlessly integrates volumes
- i-node Implementation:
  - Metadata container for each file
  - Name/data separation enables hard links
  - All links have equal status



UNIX Process Management

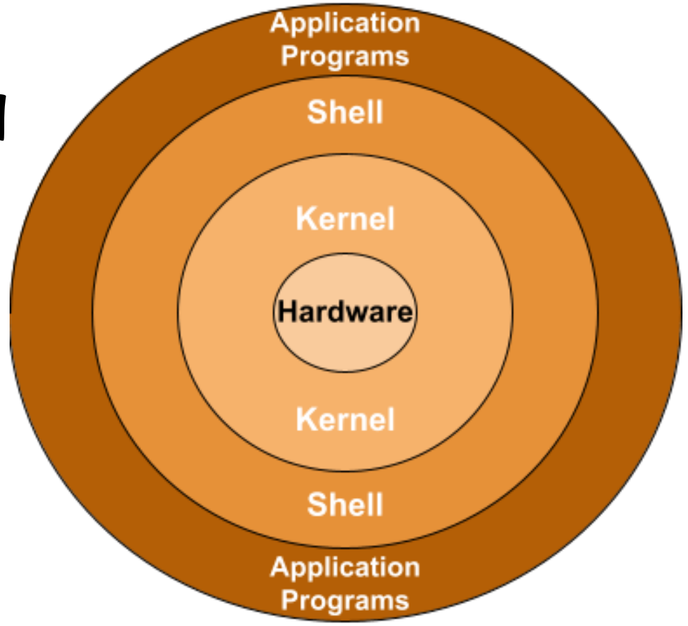
Powerful Primitives

- Process Creation:
  - fork: only way to create processes
  - exec: load new program image
  - wait: parent waits for child
  - exit: terminate process
- Inter-Process Communication:
  - pipe: creates kernel buffer
  - IPC as file operations
  - Foundation for filters: cmd1 | cmd2 | cmd3

UNIX Shell – The User Interface Glue

User-Level Strategy Layer

- I/O Redirection:
  - < input, > output via file descriptor manipulation
- Pipes: Connect process inputs/outputs
- Background Jobs: & for asynchronous execution
- Key Insight: Powerful interface can be implemented entirely in user space



UNIX Structure Summary

Monolithic Kernel with Unified Abstractions

- **Strengths:**
  - **High performance**
  - **Practical and flexible**
  - **Excellent programmer experience**
- **Weakness:**
  - **Tight coupling → single component failure can crash system**
- **Legacy: Foundation for modern practical OS design**

THE - The Rigorous Layered System

Design Philosophy

- Correctness by Construction
  - Build right, don't debug later
- Layered Abstraction
  - Clear separation of concerns
  - Each layer builds on lower abstractions
- Discrete Reasoning
  - Model concurrency as sequential processes + synchronization points

THE Hierarchical Structure

Six Logical Layers

Level	Responsibility	Abstraction Provided
0	Processor Allocation	Multiple Processes
1	Memory/Storage Control	Virtual Memory (Segments)
2	Operator Console	Private Console
3	I/O Management	Logical Communication
4	User Programs	Application Environment
5	System Operator	Human User

THE Process Management & Synchronization

Sequential Processes + Semaphores

- Sequential Processes:
  - All system activities divided into processes
  - Execute with undefined speed ratios
- Synchronization Primitives:
  - Semaphores: Special integer variables
  - P-operation(Proberen): P(sem) - potential delay
  - V-operation(Verhogen): V(sem) - remove barrier
- Two Usage Patterns:
  - Mutual exclusion semaphores (critical sections)
  - Private semaphores (condition synchronization)

THE Verification Methodology

Building Flawless Systems

- A Priori Proof:
  - Formal verification before coding
  - Prove absence of deadlocks
- Layered Testing:
  - Build and test bottom-up
  - Assume lower layers are perfect
- Exhaustive Testing:
  - Finite "relevant states" at each layer
  - Test programs generate all states
- Result: Guaranteed flawless system delivery

THE Structure Summary

Conceptual Masterpiece

- Contributions:
  - Layered architecture
  - Sequential processes
  - Synchronization primitives (semaphores)
  - Verifiable design methodology
- Influence:
  - Foundation for software engineering
  - Precursor to microkernel ideas
  - High-reliability systems design

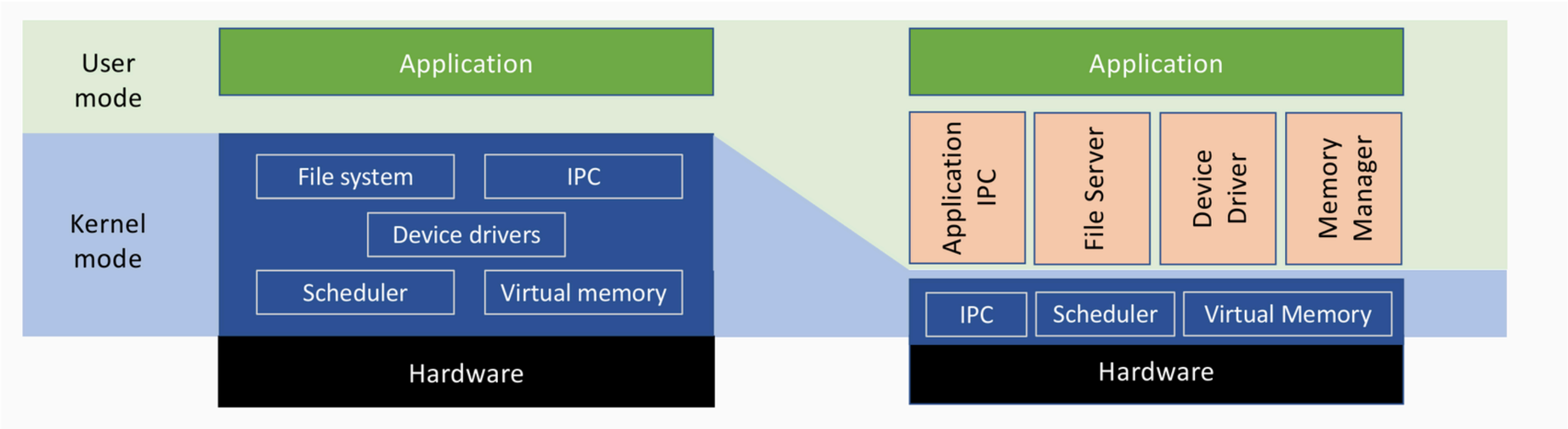
From Monolithic to Modular

- UNIX: Efficient but coupled
- THE: Modular but static
- The Next Evolution:Combine THE's isolation with UNIX-like performance

Pure Microkernel

- The kernel's functionality has been streamlined to the minimum necessary set, typically only providing three core abstractions: Address Spaces, Threads, IPC-Inter-Process Communication

- All service requests must be completed through the IPC mechanism provided by the microkernel, which is the biggest difference between it and the monolithic kernel



Pure Microkernel

Example - Read File:

- monolithic kernel: System Call → Switch from user mode to kernel mode → Execute the kernel code → Switch back to user mode

- microkernel:
  - Application: Send a message containing the file path and operation to the file system server via IPC
  - Microkernel: Copies/maps the message from the user's memory area to the memory area of the file system server
  - File system server: Sends a "Read specified sector" request message to the hard drive server via IPC
  - Hard disk drive server: Completes data reading and passes the results back to the file system server via IPC
  - File system server: Transmits the final file data back to the original application via IPC

Pure Microkernel

Features of microkernel:

- Isolation: The file system and drivers run as separate processes in the user space. If the driver crashes, the kernel remains unaffected and the system remains stable
- Communication Method: The only interaction method between the client and the server is IPC (rather than direct function calls)
- Data is copied between the user space and the kernel space more frequently → the running speed is slow
- Components must strictly follow the IPC interface and message format → insufficient flexibility

Background of Mach

- Compability between old and new
- Environment – large, complex, expensive to maintain
  - additional syscalls and options
  - less than uniform access to different resources

	Old	New
CPU architecture	CISC	RISC
Memory Architecture	uniprocessor	multiprocessor
I/O organizations	buses	networks

Mach: A New Kernel Foundation For UNIX Development

Mike Accetta, Robert Baron, William Bolosky, David Golub,
Richard Rashid, Avadis Tevanian and Michael Young
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pa. 15213

Mach: A System Software Kernel

Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi,
Robert Baron, Alessandro Forin, David Golub, Michael Jones

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Mach (/maːk/)

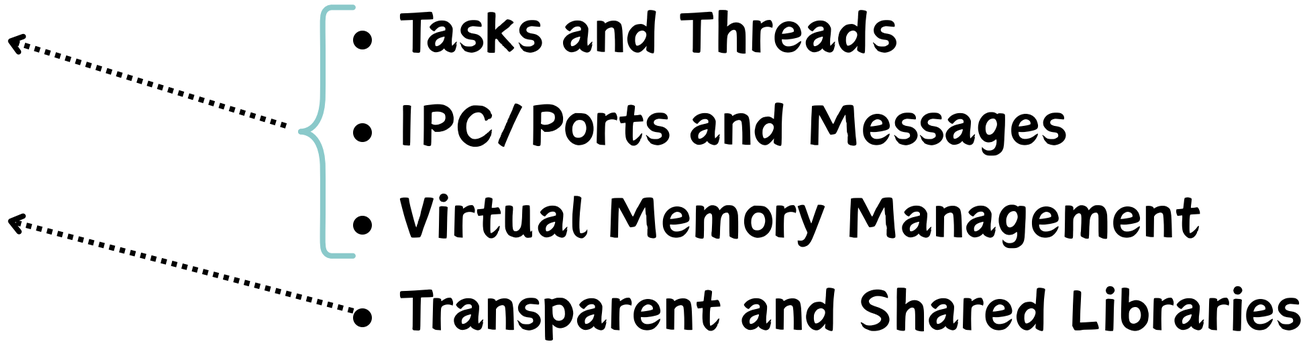
is a kernel developed at Carnegie Mellon University by Richard Rashid and Avie Tevanian to support operating system research, primarily distributed and parallel computing. Mach is often considered one of the earliest examples of a microkernel.

Name

The developers had to bike to lunch through rainy Pittsburgh's mud puddles, and Tevanian joked the word "muck" could form a backronym for their **M**ulti-**U**ser (or **M**ultiprocessor **U**niversal) **C**ommunication **K**ernel. Italian CMU engineer Dario Giuse[5] later asked project leader Rick Rashid about the project's current title and received "MUCK" as the answer, though not spelled out but just pronounced: IPA: [mʌk] which he, according to the Italian alphabet, wrote like Mach. Rashid liked Giuse's spelling "Mach" so much that it prevailed.[6]:103

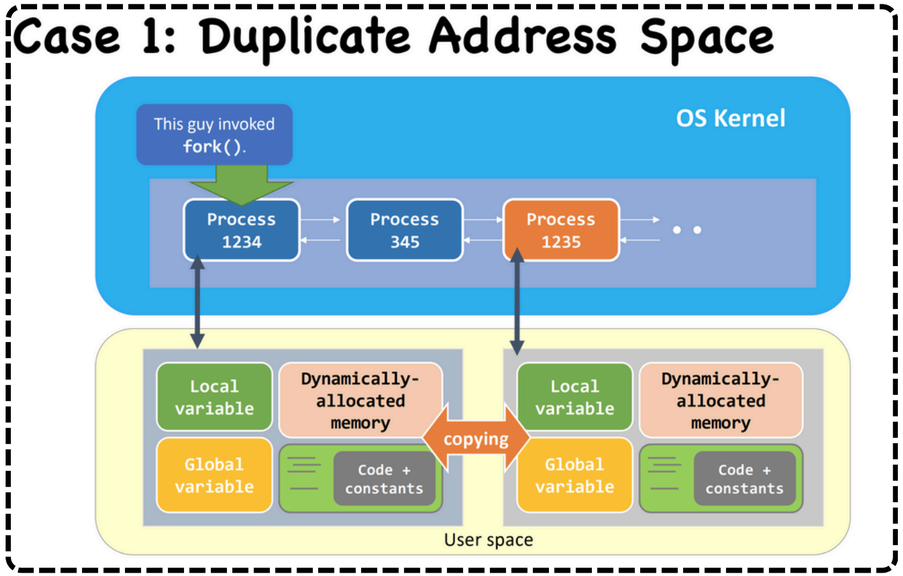
Design Goals and Key Features

- operate on both uniprocessors and multiprocessors
- support for binary compatibility with existing operating system environments
- provide basics for implementation of various operating system environments



1. Task and Threads

- UNIX: (process)
  - High cost
  - Resource waste
  - Poor parallelism



- Mach:
- Task → resources
  - virtual address space
  - a set of port rights
- Thread → execution
  - basic unit of computation

- Create a new task (child process)
- Create a thread in the new task
- Share the memory space of the parent task using copy-on-write technology

1. Task and Threads

- allows multiple threads to exist (execute) within a single task
- On tightly coupled shared memory multiprocessors, multiple threads may execute in parallel
- mechanism and strategy are separated

Lightweight

Parallelism

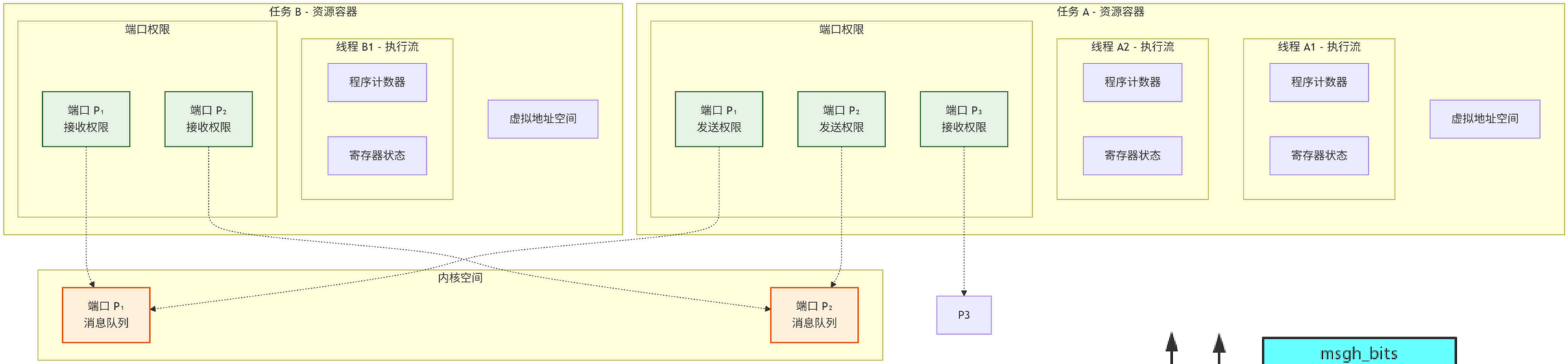
Scalability

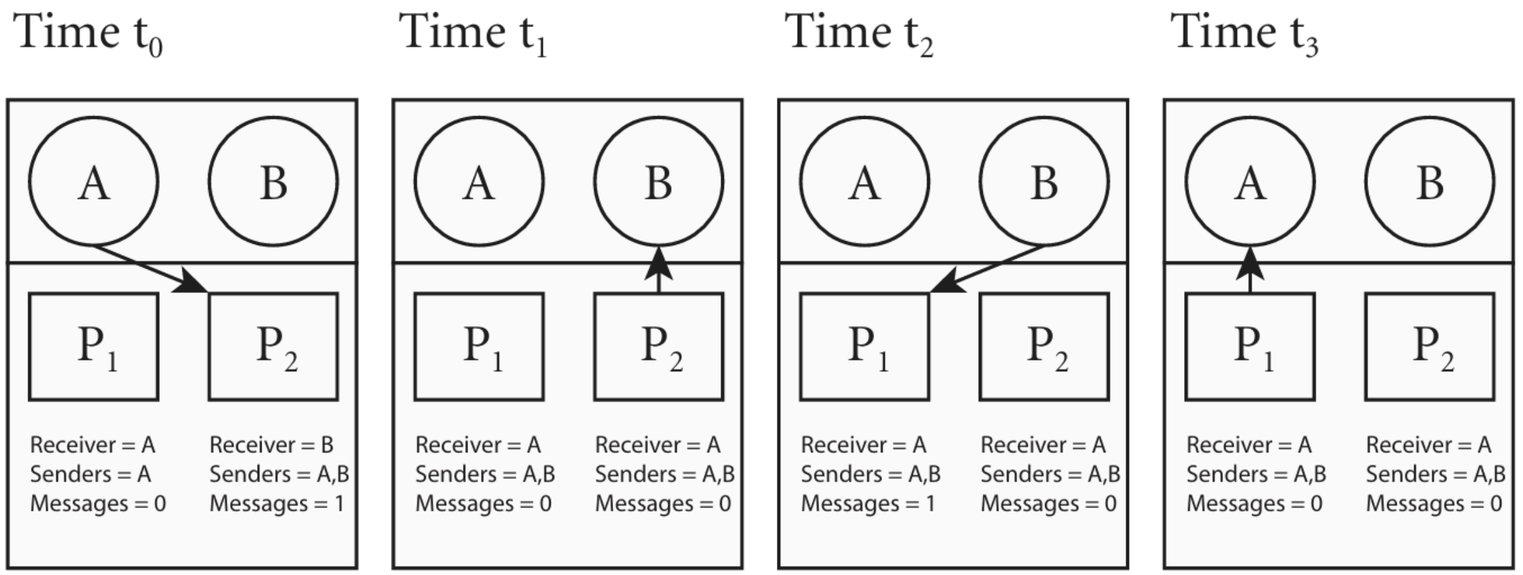
2. Interprocess Communication

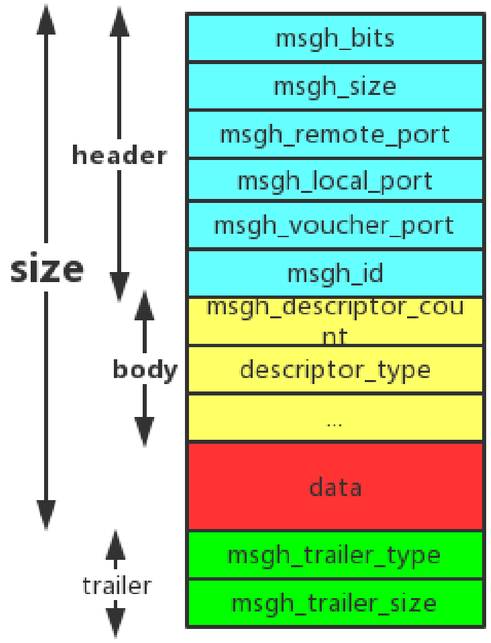
- Port:
  - logically a finite length queue of messages sent by a task
  - Ports may have any number of senders but only one receiver
  - Access to a port is granted by receiving a message containing a port capability (to either send or receive).
- Message:
  - consists of a fixed length header and a variable size collection of typed data objects.
  - may contain both port capabilities and/or embedded pointers as long as both are properly typed.
  - A single message may transfer up to the entire address space of a task.

a kernel-controlled and secure manner

insufficient flexibility







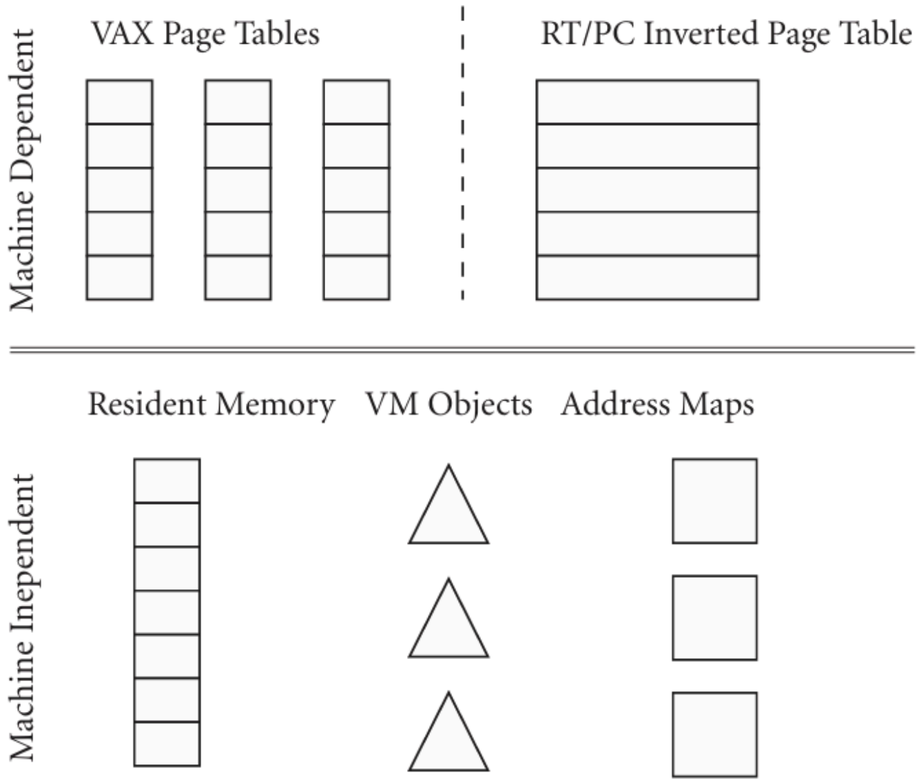
3. Virtual Memory Management

Machine Dependent section

- merely executes commands
- provides interfaces
  - validate()
  - invalidate()
  - protect()

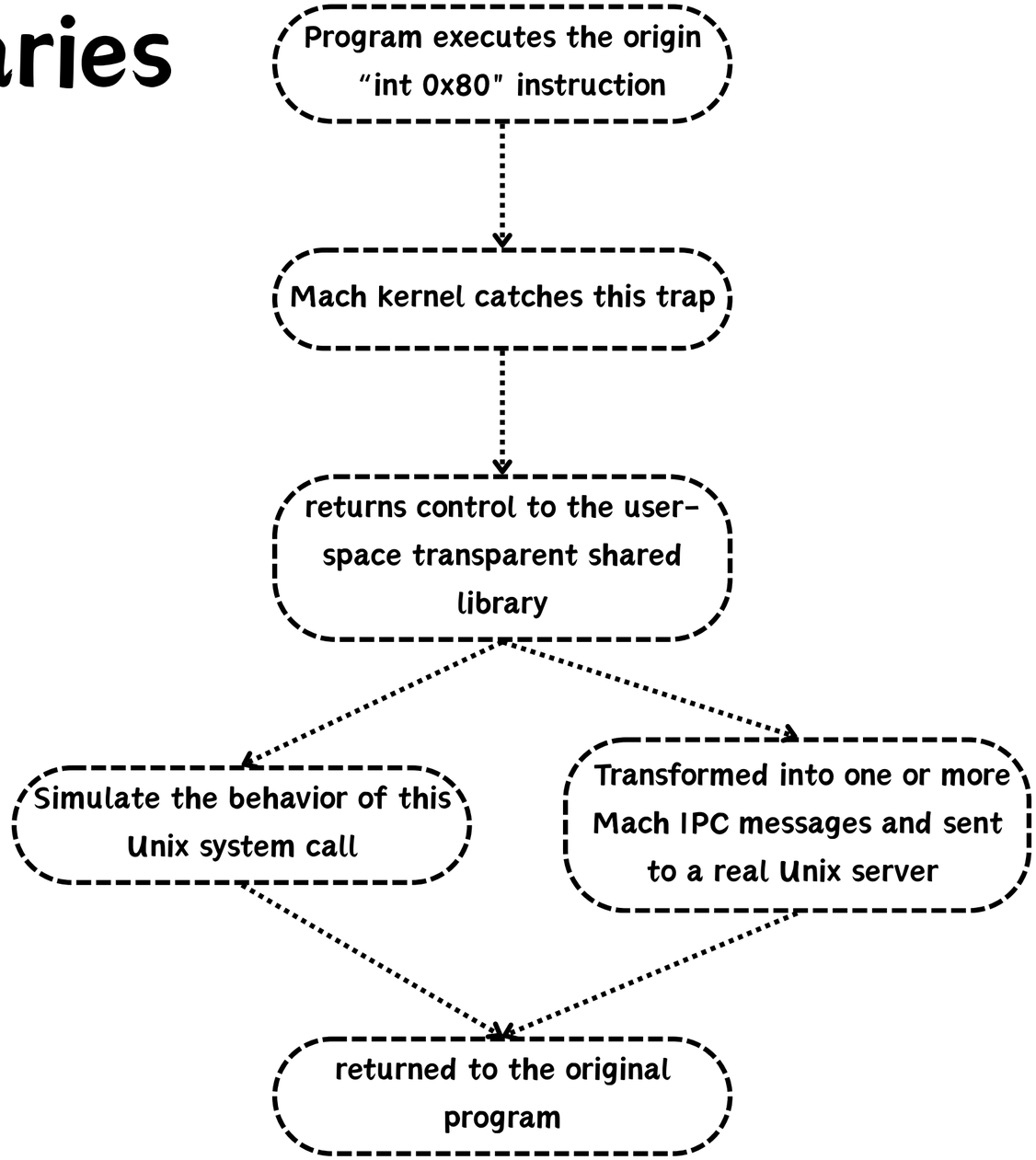
Machine Independent section

- design strategies and make decisions
  - address maps
  - share maps
  - VM objects



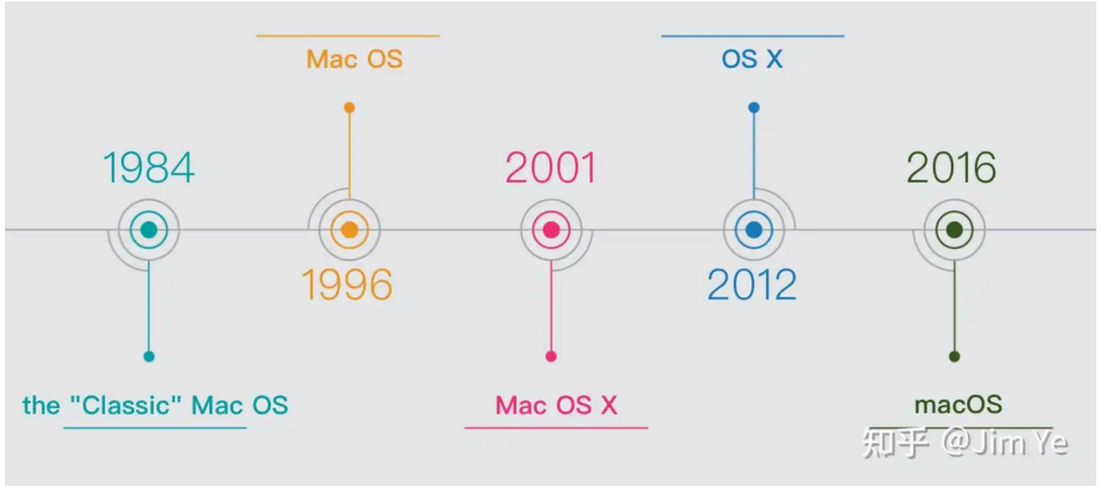
4. Transparent Shared Libraries

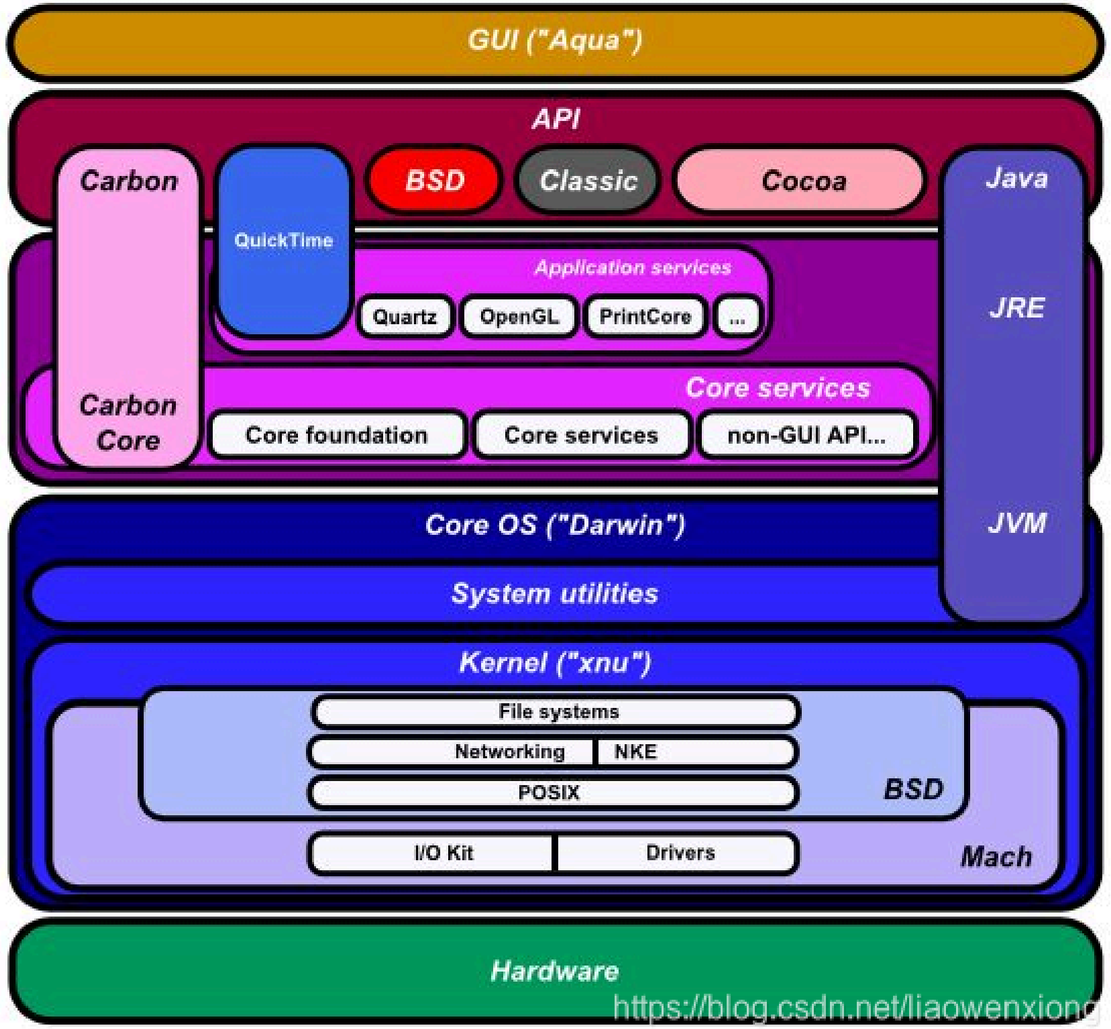
- A transparent shared library is a code library
- can intercept system calls made by that program
- can be used for a variety of purposes, such as:
  - binary compatibility with non-Mach OS environ ments,
  - support for multiple OS environments (e.g. Unix 4.3, Unix V.4),
  - debugging and monitoring
  - network redirection of OS traps



Future development & relation with Mac OS X

- Lightweight but inefficient
- 1985 – The NeXT team founded by Steve Jobs used the Mach kernel and part of the BSD codebase to create the NeXTSTEP operating system. (XNU kernel)
- 1997 – Apple bought NeXT, and NeXTSTEP became the basis of Mac OS X.
- 2001 – Mac OS X v 10.0 was released





The second-generation microkernel L4

- The main change in L4 is IPC

	Pure Microkernel	L4
Communication Mode	Support synchronous and asynchronous IPC	Only supports synchronous IPC, remove the message queue
Message Passing	Sender user memory -[copy]-> Kernel buffer -[copy]-> Receiver user memory	Sender register ---[context switch]--->Receiver register (large data blocks → Map&granting mechanism)
Memory Management	Kernel contains complex virtual memory objects, port collections, etc	Kernel only provide three atomic primitives: Grant, Map, and Unmap. A user-level Pager server is responsible for all complex policies
Small-Address-Space Optimization	Each time the context is switched to another address space, the TLB usually needs to be refreshed	Give small spaces a unique label, and when switching between different small address spaces, the TLB can continue to retain the previous cache records

Transplant the Linux operating system to run on the L4 microkernel

- Transplant the entire Linux kernel as a huge user-level service (L4Linux) onto the L4 microkernel, and compare its performance with the native Linux and systems based on the first-generation microkernel (MkLinux/Mach) to evaluate the actual performance loss of L4

- Demonstrate the flexibility and scalability of the L4 architecture

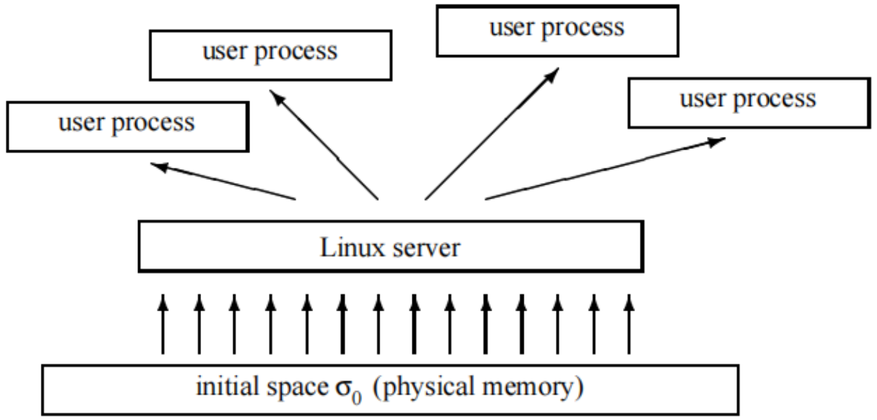
(1) Memory management

- Linux server requests memory resources from the underlying pager, usually handled by σ0 (the initial address space), which maps the available physical memory one-to-one to the server's address space, the Linux server then acts as the pager for user processes

- The actual hardware page tables are stored internally in L4, and user-level processes cannot directly access them. Therefore, the Linux server must maintain additional logical page tables in its own address space

Transplant the Linux operating system to run on the L4 microkernel

(1) Memory management



(2) Threads and synchronization mechanisms

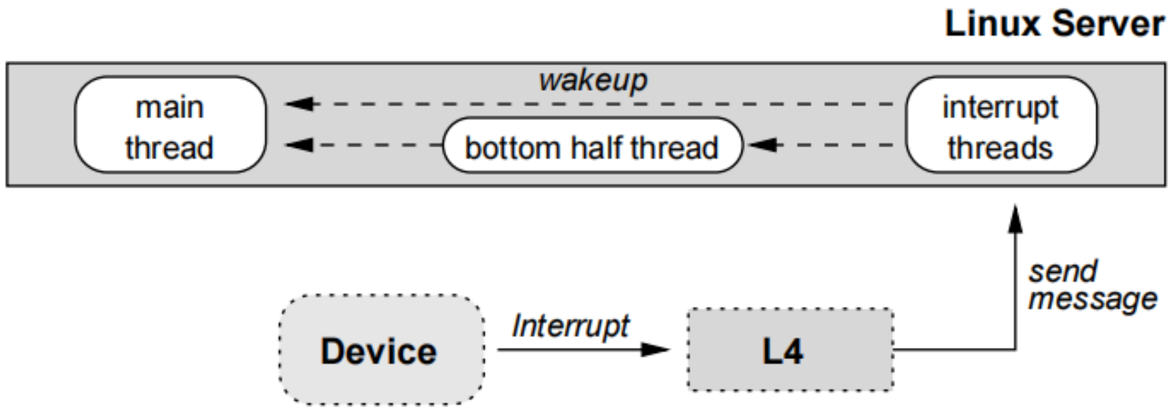
- Thread model: Single-threaded multiplexing
  - The entire L4Linux server uses one L4 thread to handle all the activities caused by system calls and page faults
  - Linux's multiplexing: Linux kernel uses its own mechanism internally to multiplex this thread, so as to handle multiple user requests simultaneously

Transplant the Linux operating system to run on the L4 microkernel

(2) Threads and synchronization mechanisms

- Synchronization mechanisms:
  - Linux synchronization: The original single-processor Linux kernel mainly achieves synchronization and protection of critical sections (key code segments) by disabling interrupts. When a thread enters the critical section, it uses interrupt disabling to ensure that other code does not interrupt it
  - L4's adaptability: The L4 microkernel design allows privileged user-level tasks to disable interrupts

(3) Interrupt handling



Transplant the Linux operating system to run on the L4 microkernel

(4) User Processes

- L4 microkernel is responsible for providing the underlying infrastructure (tasks and threads), the Linux server acts as the manager of the OS to control these basic resources
- Emulation Library
  - Originally, the Linux macro kernel could directly perform system calls, but when using the L4 microkernel, this simulation library is needed to convert system calls into IPC instructions for forwarding
  - All system calls (such as getpid, read) are captured by this library and forwarded to the Linux server for processing reasoning

Transplant the Linux operating system to run on the L4 microkernel

(5) Signal transmission

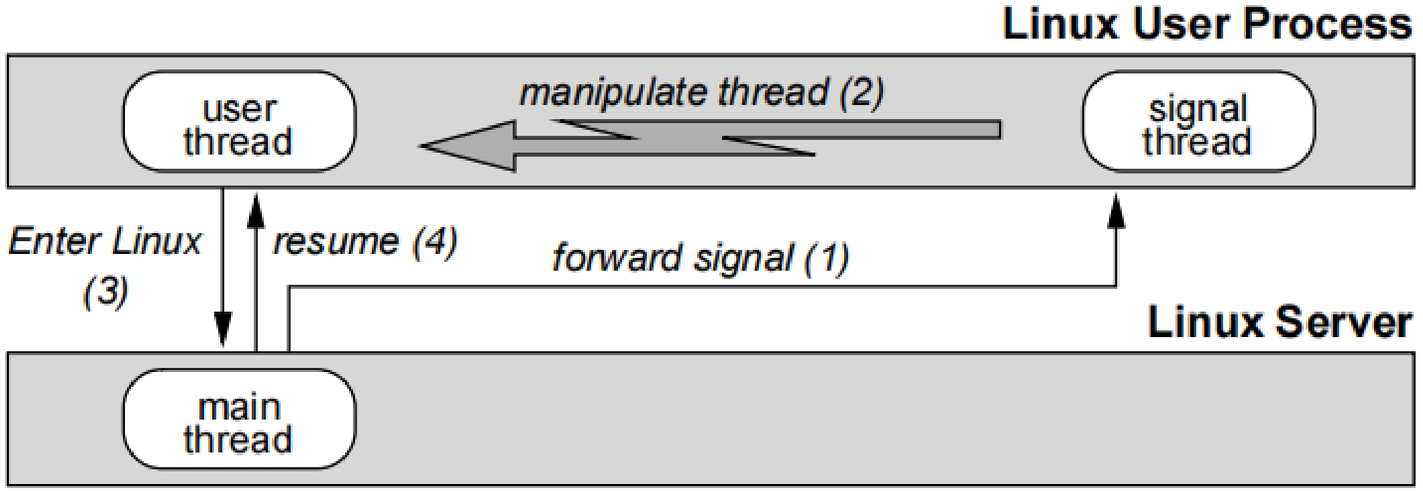


Figure 3: *Signal delivery in L⁴Linux.* Arrows denote IPC. Numbers in parentheses indicate the sequence of actions.

Transplant the Linux operating system to run on the L4 microkernel

Compatibility Performance -- Tests and Analyses of L4Linux

- The current L4Linux implementation is already quite close to the behavior of native Linux, even under heavy loads. The typical penalty ranges from 5% to 10%
- The performance of the underlying kernel is of vital importance
- Relying solely on co-location cannot solve the problems caused by the poor performance of the underlying kernel

Extensibility Performance -- Tests and Analyses of L4Linux

- L4 microkernel can achieve both compatibility and high performance simultaneously
- L4 architecture provides a strong foundation for running operating systems with completely different design paradigms on the same hardware
- L4 not only offers the inherent modularization and security (isolation) of a microkernel, but also achieves high performance, making it a practical and usable basic platform