CS 202
Advanced Operating Systems

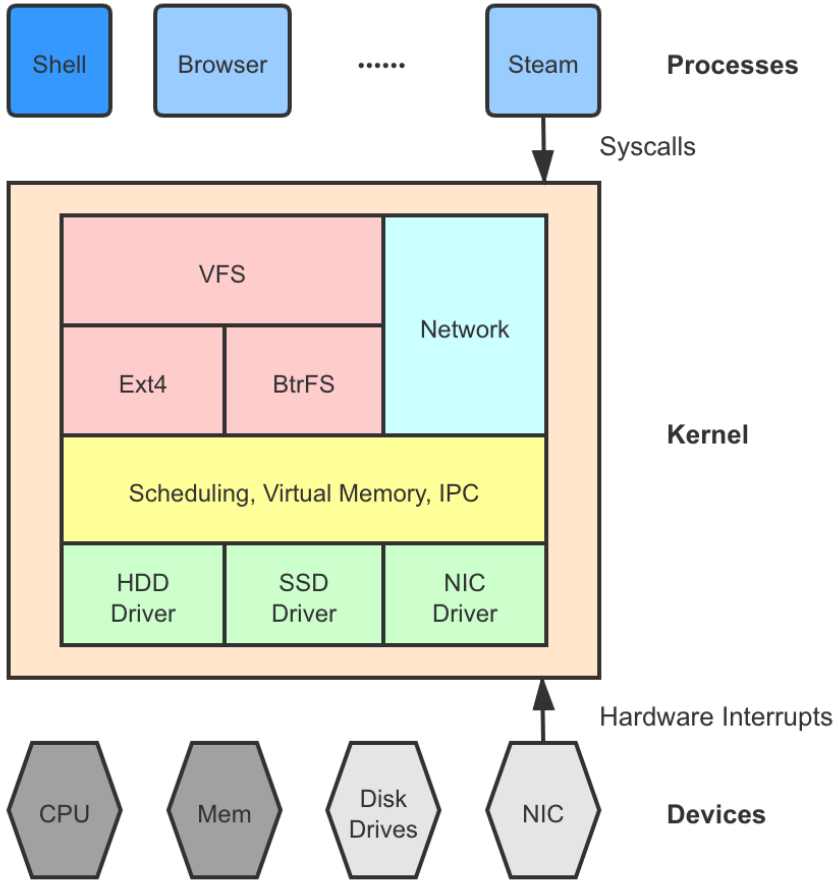
Winter 26

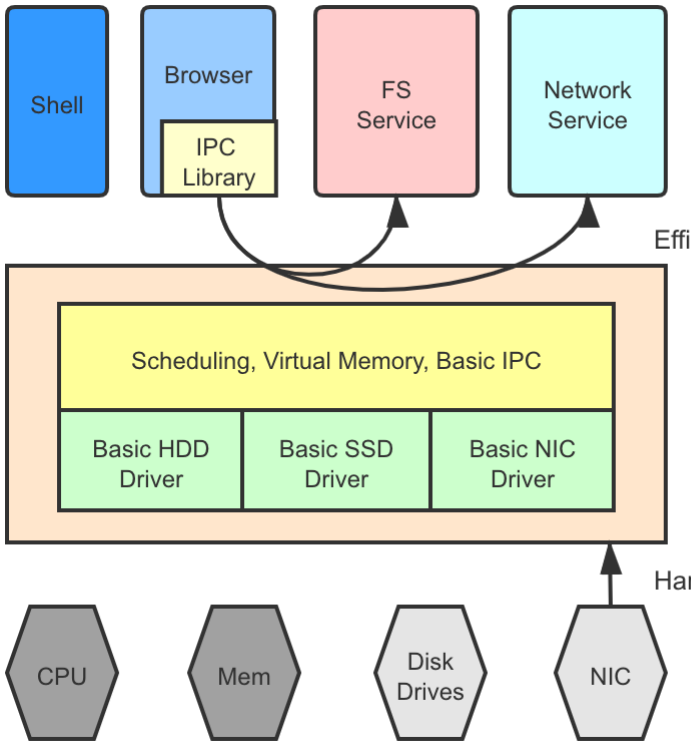
Lecture 3: OS Architecture

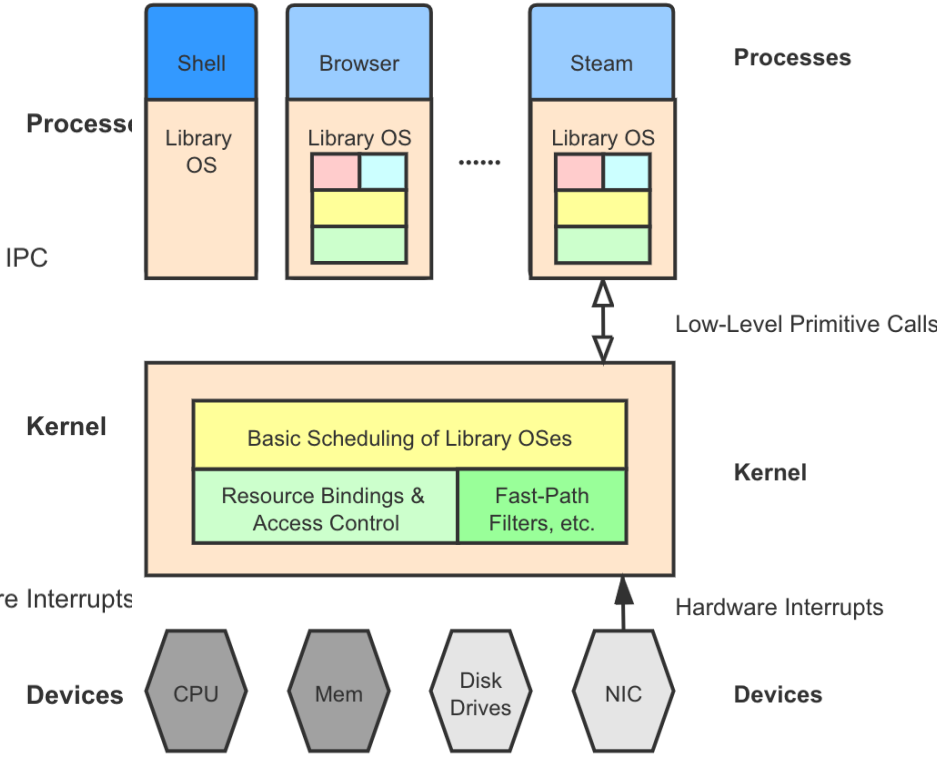
Instructor: Chengyu Song

Typical OS architectures

- Monolithic
  - UNIX, Linux, Windows
- Micro-kernel
  - L4, Mach, QNX
- LibraryOS
  - exokernel, graphene

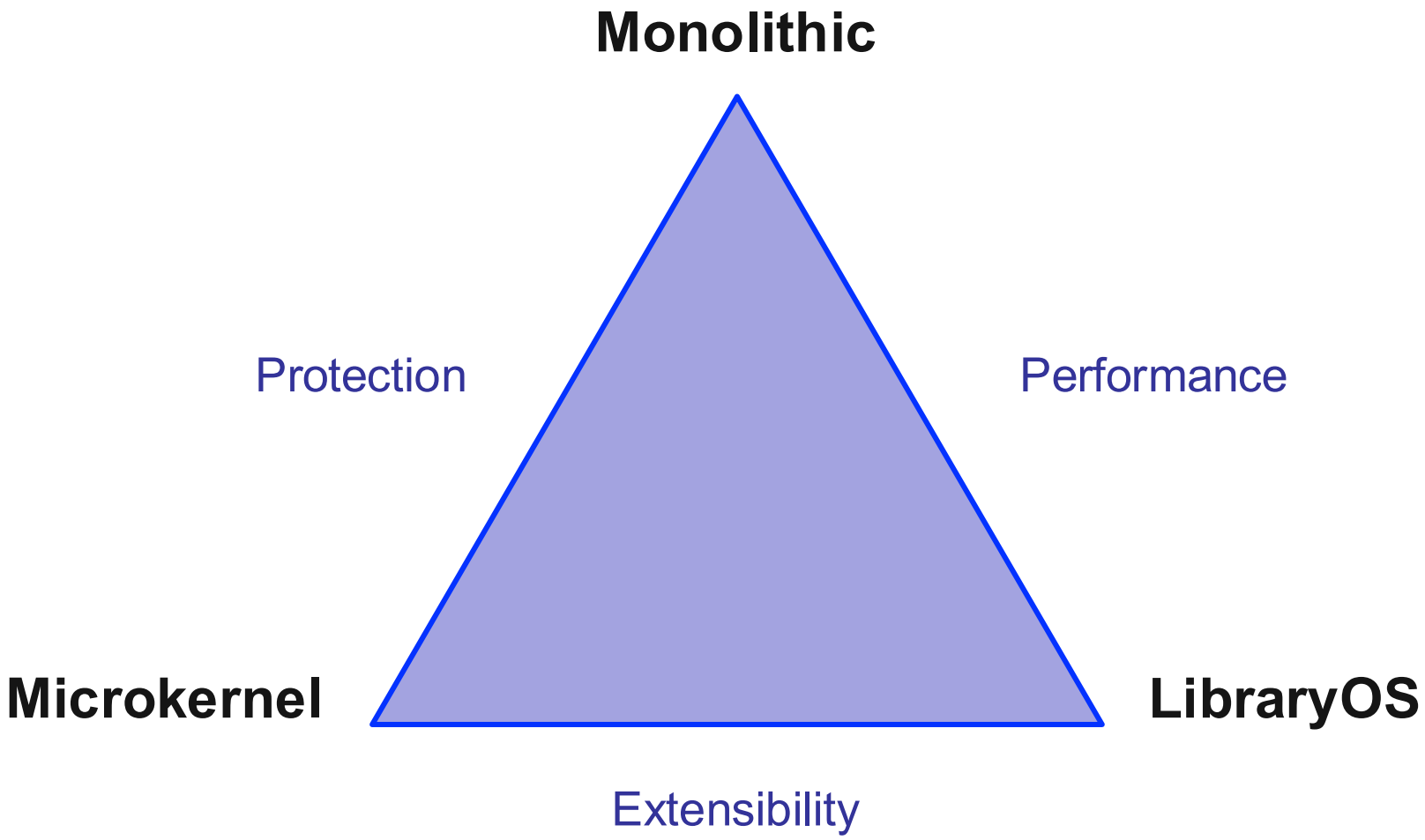






Figures from https://www.josehu.com/technical/2021/05/24/os-kernel-models.html

Trade-offs of OS architectures



Trade-offs of OS architectures

- Monolithic kernels
  - Good performance
  - Okay protections (has isolation, too big to be bug free, hard to verify)
  - Bad extensibility
- Microkernels
  - Very good protection (isolation, small, verifiable)
  - Okay extensibility
  - Bad performance
- LibraryOS
  - Very good extensibility
  - Good performance
  - Bad protection (no isolation between U/libOS, exposes everything)

Exokernel

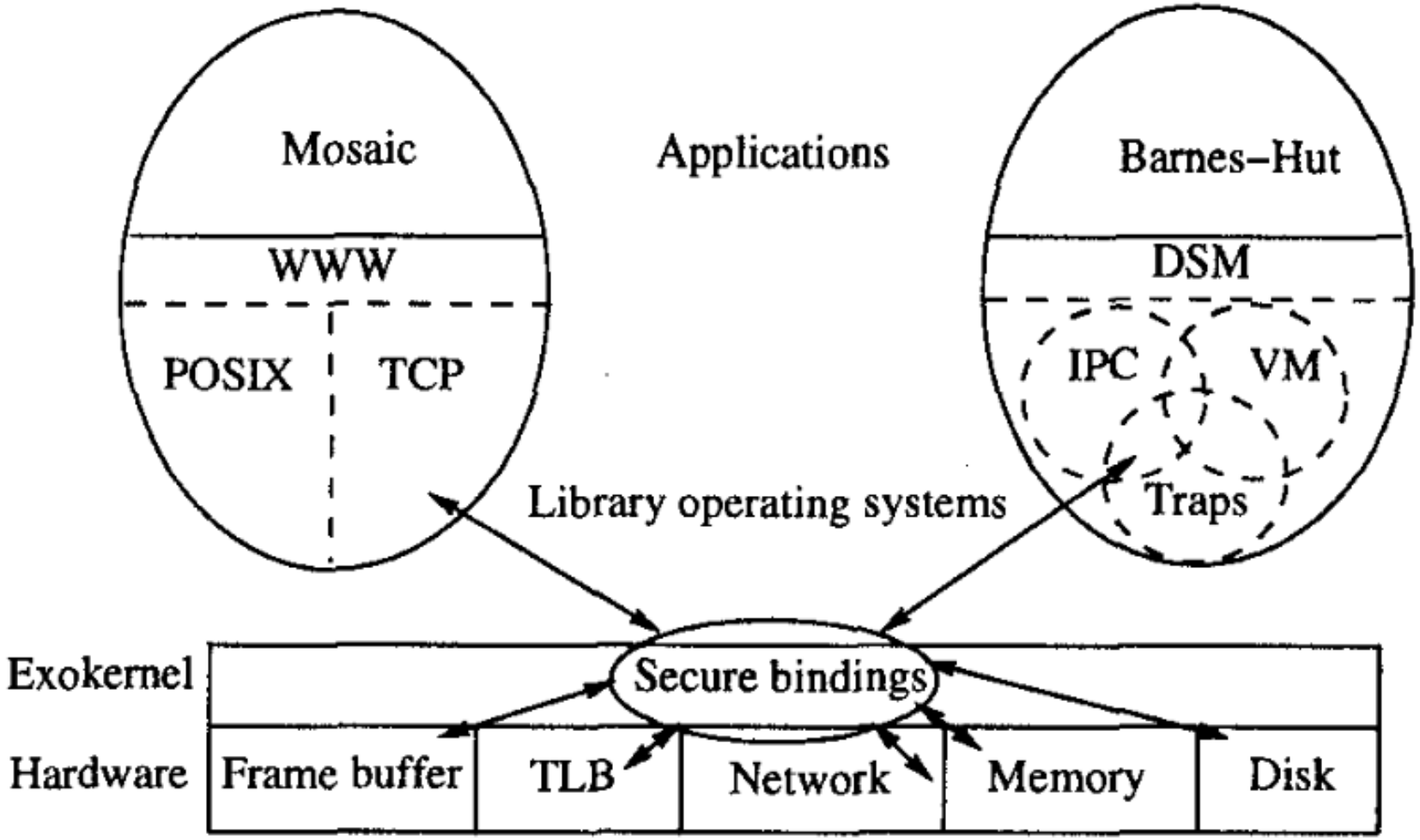
- Motivations: fixed high-level abstractions are hurting applications
  - Fixed high-level abstractions hurt application performance by denying domain-specific optimizations
    - » Scheduler, cache eviction policy
    - » Evidences: Cao et al. [1 O] measured that application-controlled file caching can reduce application running time by as much as 45%.
  - Fixed high-level abstractions discourages changes to existing abstractions and limit the functionality of applications
    - » Syscalls rarely change, if ever
    - » Evidences: few research ideas have been adopted
  - Fixed high-level abstractions hide information from applications and prevent it from making good decisions
    - » Low-level exceptions, interrupts, raw device I/O
    - » Evidences: database implementors must struggle to emulate random-access record storageon top of tile systems [47].

Exokernel

- Core idea: **Separating Protection from Management**
  - Exokernel: provides protection
  - LibraryOS: implements management and high-level abstractions

- Solution: securely multiplexes available hardware resources through a low-level interface
  - Low-level: even more flexible / aggressive than microkernel
  - No emulation: direct exposing hardware (including naming): much lower overhead than virtual machine monitors
  - Securely: trust but verify

Exokernel: architecture



Exokernel: Design Princincple

- **Securely expose hardware**
  - An exokernel strives to safely export all privileged instructions, hardware DMA capabilities, and machine resources
- **Expose allocation**
  - An exokernel should allow library operating systems to request specific physical resources
- **Expose names**
  - An exokernel should export physical names
- **Expose Revocation**
  - An exokemel should utilize a visible resource revocation protocol so that an well-behaved libraryOS can perform effective application-level resource management
  - All revocations, including CPU (i.e., explicit context switch)
- **Arbitration**: same as other OS

Exokernel: secure bindings

- Tracks resource ownership
  - Allocation to libraryOS at bind time
  - Free to use afterwards, no more authorization
- Protection (ownership) checks are simple operations performed by the kernel
  - Hardware mechanisms (if available, e.g., capability tokens)
    - » Memory: TLB, page tables, DMA
  - Software caching (bookkeeping)
  - Downloadable application code
    - » Packet filter
- Allows protection without understanding

Exokernel: secure binding example (1)

- Virtual-to-Physical address translation
  - Protecting which physical pages can be accessed by a libraryOS
- TLB-based
  - LibraryOS decide the virt-to-phy mapping
  - LibraryOS decide the TLB replacement policy
  - Binding presented to exokernel
  - Exokernel checks ownership, if passed, installs/removes entries in hardware TLB
  - LibraryOS can access physical memory freely
- Page-table based
  - More challenging, Xen, hardware virtualization

Exokernel: secure binding example (2)

- Packet filtering
  - Handle packet distributing (who handles?) and processing (e.g., firewall)
  - In-kernel processing is much faster than polling / querying
- Manage packet handling from library (app policy)
  - Packet filter code provided by library (app)
  - In the form of downloadable code
  - Binding (code) presented to exokernel
  - Exokernel verifies the code
  - Exokernel installs app's packet filter code (e.g., to hardware NIC)

Exokernel: hypotheses

- The design and implementation of exokernel is built upon four hypotheses that must be validated through empirical evaluation
  - Exokemels can be very efficient
  - Low-level, secure multiplexing of hardware resources can be implemented efficiently
  - Traditional operating system abstractions can be implemented efficiently at application level
  - Applications can create special-purpose implementations of these abstractions
- Q: how to design experiments?

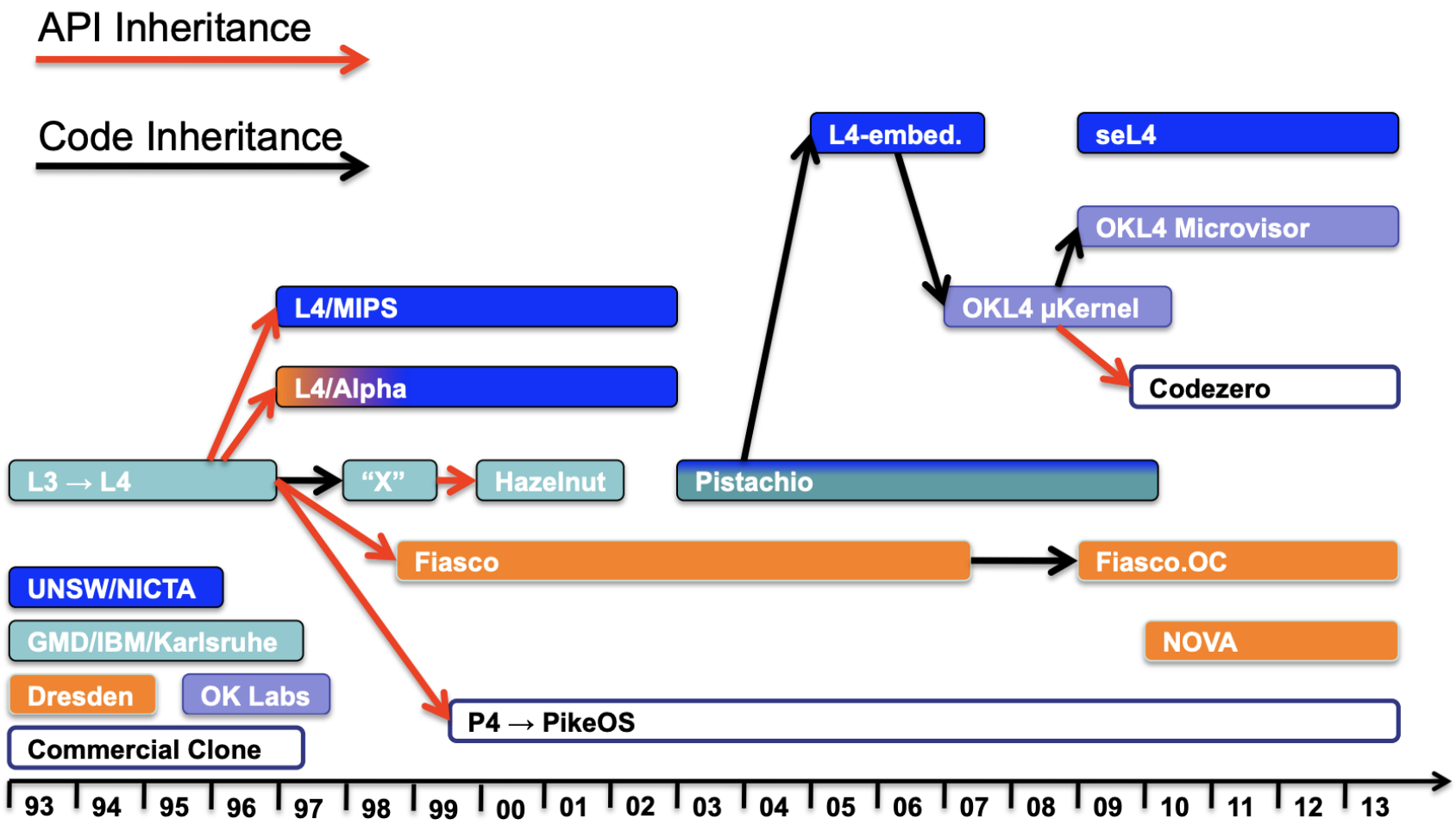
L4 microkernel

- Conventional belief
  - μ-kernel is a nice concept, but
  - They are (1) inherently inefficient and (2) not sufficiently flexible
- L4 (SOSP 95): microkernels can be high performance
  - Slowness is an implementation issue rather they design
  - Microkernel aims to provide only necessary abstractions
  - Address space/process (memory), threads (exec), and IPC (comm)
  - Minimal overhead IPC by exploiting hardware features
  - Isolate applications and drivers in user-space

L4 microkernel family

- L3 to seL4 (SOSP 2013): subsequent evolution of the design of L4-based microkernel systems
  - Evolution of microkernels from L4 predecessor (SOSP 1993) to the ”security-enchanced” L4 (seL4) from UNSW
  - Presents design decisions and reasoning on key functionality, such as IPC and resource management
  - Successful OS with different offshoot distributions
- Commercially successful
  - OKLabs OKL4 shipped over 1.5 billion installations by 2012
    - » Mostly Qualcomm wireless modems
    - » But also player in automotive and airborne entertainment systems
  - Used in the secure enclave processor on Apple’s A7 chips
    - » All iOS devices have it! 100s of millions

L4 family tree



L4: how to build good microkernel

- Principle: **only put in anything that if moved out prohibits functionality**
  - A concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality [Liedtke, 1995].
- Assumptions
  - We require security/protection
  - We require a page-based virtual memory
  - Subsystems should be isolated from one another
  - Two subsystems should be able to communicate without involving a third

L4: Address Space Management

- **Idea**: support recursive construction of address spaces outside the kernel (i.e., delegating management to userspace)
- After boot: only one address space σ_0 which essentially represents the physical memory and is controlled by the first subsystem S_0
  - **Grant**: The owner of an address space can grant any of its pages to another space, provided the recipient agrees.
  - **Map**: The owner of an address space can map any of its pages into another address space, provided the recipient agrees.
  - **Flush**: The owner of an address space can flush any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly from the flusher.
- Handles I/O too
  - Memory mapped I/O, port

L4: Threads and IPC

- Threads represent execution – used as unique identifiers
  - What's in the Thread's abstraction?
    - A set of registers, including at least an instruction pointer, a stack pointer
    - A state information
    - An address space
- End point for IPC (messages)
- Interrupts are IPC messages from kernel
  - Microkernel turns hardware interrupts to thread events

L4: Flexibility?

- Memory Manager
- Pager
- Multimedia Resource Allocator
- Device Driver
- Second Level Cache and TLB
- Remote Communication
- Unix Server

L4: Performance, Facts & Rumors

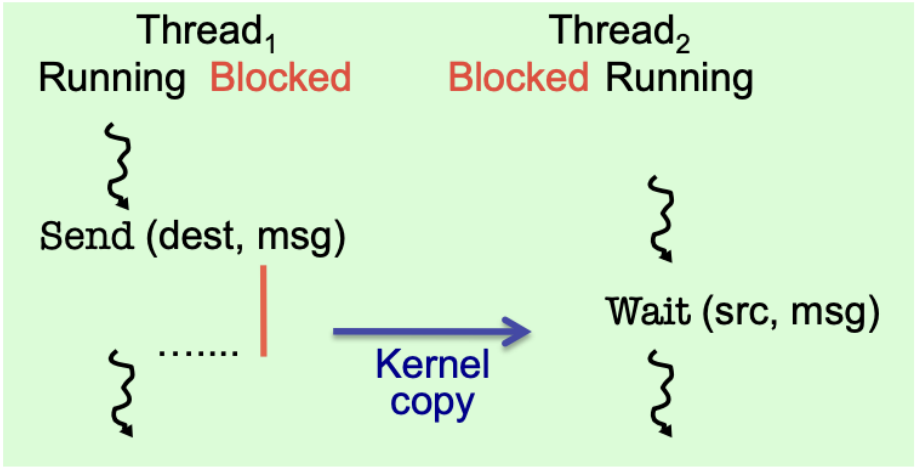
- Context switch overhead
  - Kernel-user switches
    - Mach (old generation μ-kernel): 18 μs (900 cycles) for a mode switch
    - Theoretical lower bound: 107 cycles
    - L3: 123 to 180 cycles (15 cycle overhead), mostly less than 3 μs
  - Address space switches
    - Mostly a TLB issue
    - Properly constructed address-space switches are not very expensive, less than 50 cycles on modern processors (in 1995)
    - Now?
      - TLB entries are tagged with ASID, multiple levels of TLB, page table entries are cached too
  - Thread Switches and IPC
    - With clever use of hardware (e.g., segment vs paging), IPC can be fast

L4: Portability?!

- Old µ-kernels were built machine-independently on top of a small hardware-dependent layer
  - Cannot take advantage of specific hardware
  - Cannot take precautions to circumvent or avoid performance problems of specific hardware
  - The additional layer per se costs performance
- To be efficient, a µ-kernel must be **inherently non-portable**. It should be treated like an optimized code generator, tailored specifically to the processor's architecture (TLB, cache, and register structures)

L4: IPC Evolvement

L4 Synchronous IPC

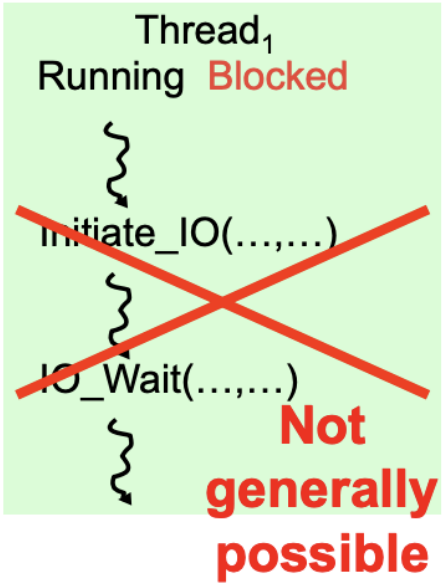


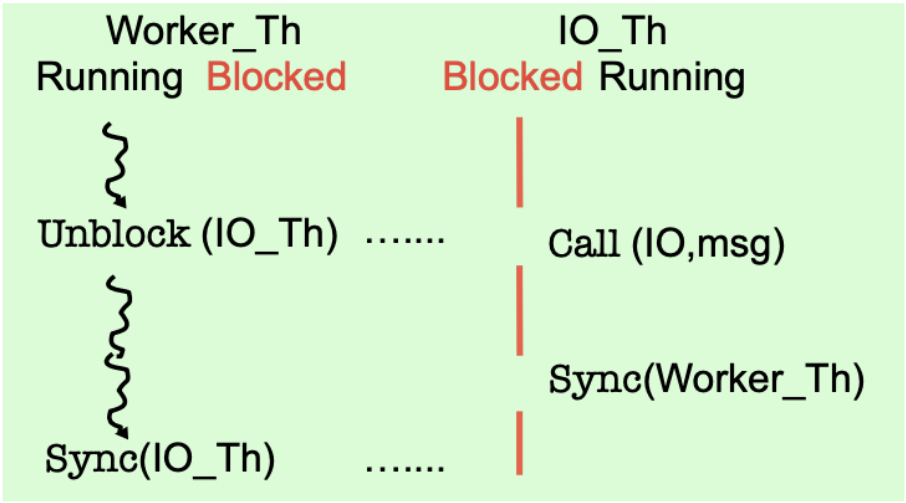
Rendezvous model

Kernel executes in sender's context
- copies memory data directly to receiver (single-copy)
- leaves message registers unchanged during context switch (zero copy)

L4: IPC Evolvement

Synchronous IPC Issues

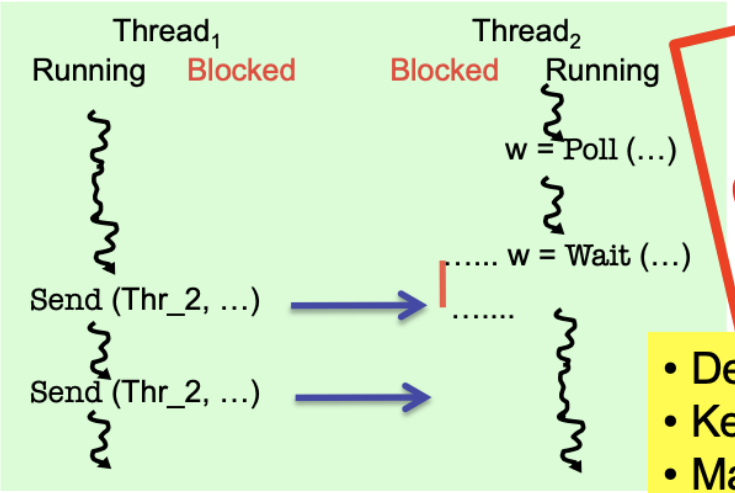




- Sync IPC forces multi-threaded code!
- Also poor choice for multi-core

L4: IPC Evolvement

Asynchronous Notifications



Sync IPC complemented with async

- Delivers few bits (destructively)
- Kernel only buffers single word
- Maps well to interrupts, exceptions

- Thread can wait for synchronous and asynchronous messages concurrently

L4: IPC Evolvement