CS202 – Advanced Operating Systems

OS Structures

January 22, 2025

Check your understanding

- True or False: a process can use more memory than the physical memory of the computer?
  - Yes, any process memory not in physical memory is in swap
- True or False: A virtual address can be resolved to a physical address without using the process's page tables
  - Yes, can have a "hit" in the TLB
  - TLB is a cache of virtual page to physical frame mappings
- Where are memory mappings for each process stored?
  - In the process's own Page Table and TLB

Operating System Organization

- The bigger conversation…
- In the 70s and 80s, OS design started emerging as a discipline
- How should the OS be structured?
  - Why does it matter? What can be accomplished by a good/bad structure?
- For time sharing, its clear we need a separate OS and user space
  - Do we need further structure?

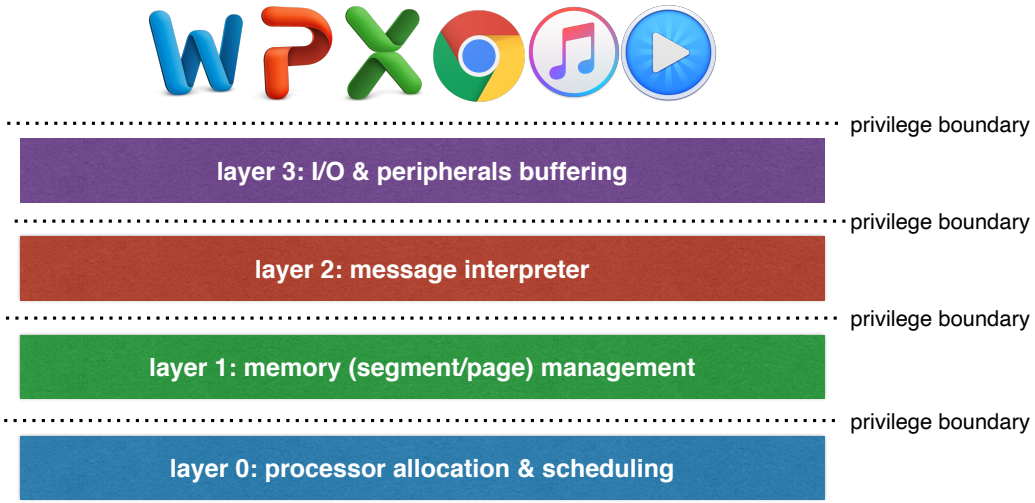
Why is the structure of an OS important?

- Protection
  - User from user *and* system from user *and* system from system
- Performance
  - Does the structure facilitate good performance?
- Flexibility/Extensibility
  - Can we adapt the OS to the application?
- Scalability
  - Performance overhead increases when more resources are used
- Agility
  - Adapt quickly to application needs and resources
- Responsiveness
  - How quickly it reacts to external events?

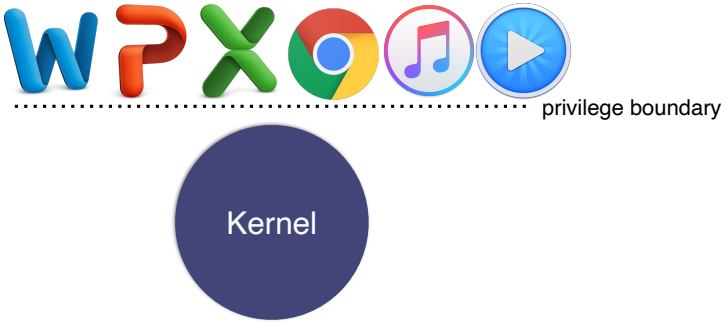
An earlier conversation

THE v.s. Hydra

THE



Hydra



Extensibility

- What do we mean by extensibility?
  - Flexible to add new features/functionalities
  - Good efficiency
  - Good security
- Can you give a few examples?
  - Device drivers
  - Browser plugins/extensions

Extensibility context

- Traditional OSes provide a standard…
  - Set of abstractions
    - Processes, threads, VM, Files, IPC
    - Reachable through syscalls
  - Resource allocation and management
  - Protection and security
- Industry complaining of large OS overheads
  - Researchers were doing customized extensions
  - Research community started asking how to provide customization
    - Need to do so while maintaining security and good performance

Is extensibility really important?

- What are the arguments in the Exokernel paper?
  - OS does not perform well for specific applications
    - End to end argument in system design
- What examples of applications do they list?
- Is it an implementation or abstraction issue?
  - Both? Abstractions overly general, and implementations are fixed
  - Protection and management interfere with performance and flexibility

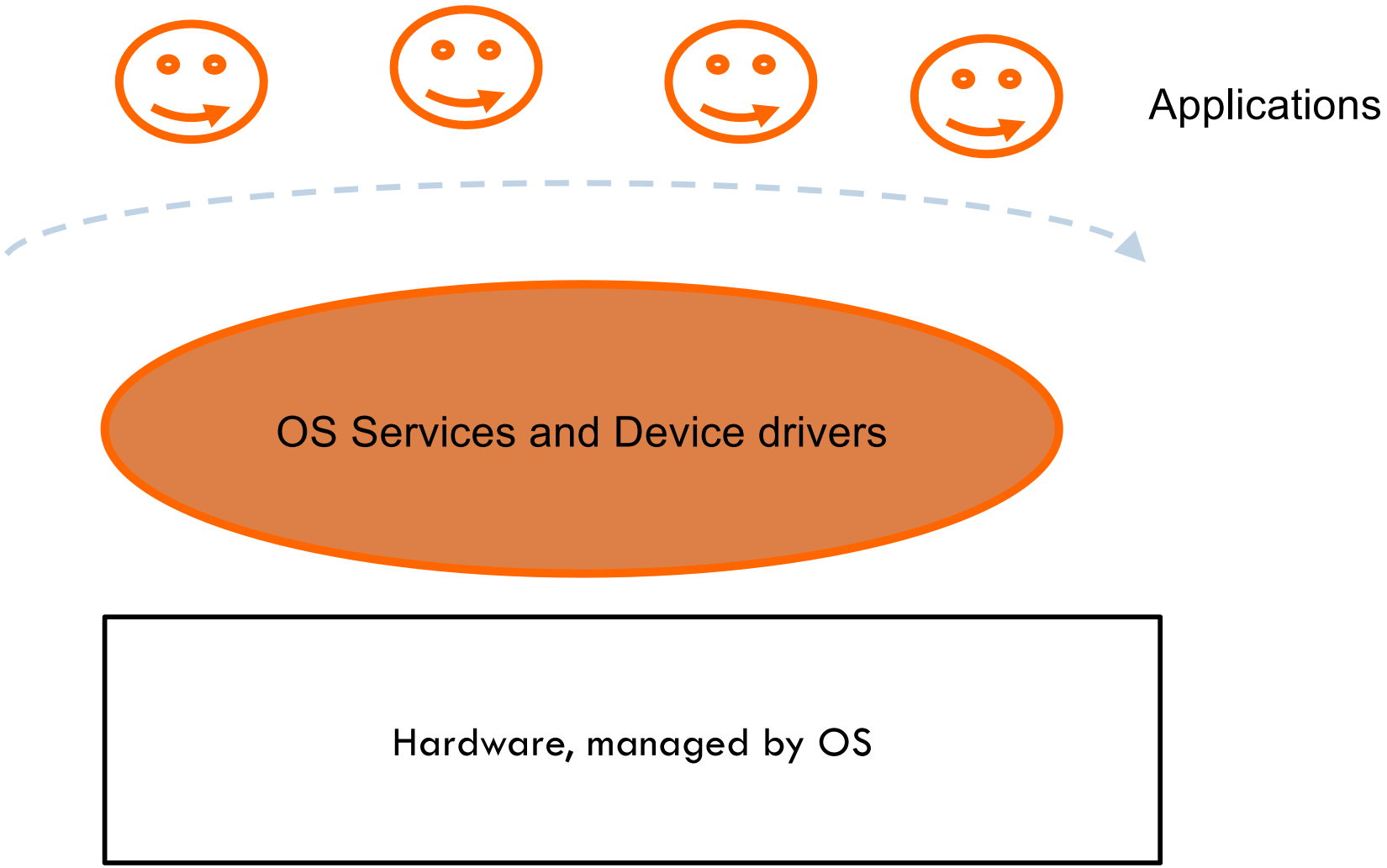
How expensive are border crossings?

- Procedure call:
  - Save some general-purpose registers and jump
- Mode switch:
  - Trap or call gate overhead
    - Nowadays syscall/sysreturn
  - Switch to kernel stack
  - Switch some segment registers
  - 100s of ns
- Context switch:
  - Change address space
  - This could be expensive; flush TLB, …
  - Few microsecs

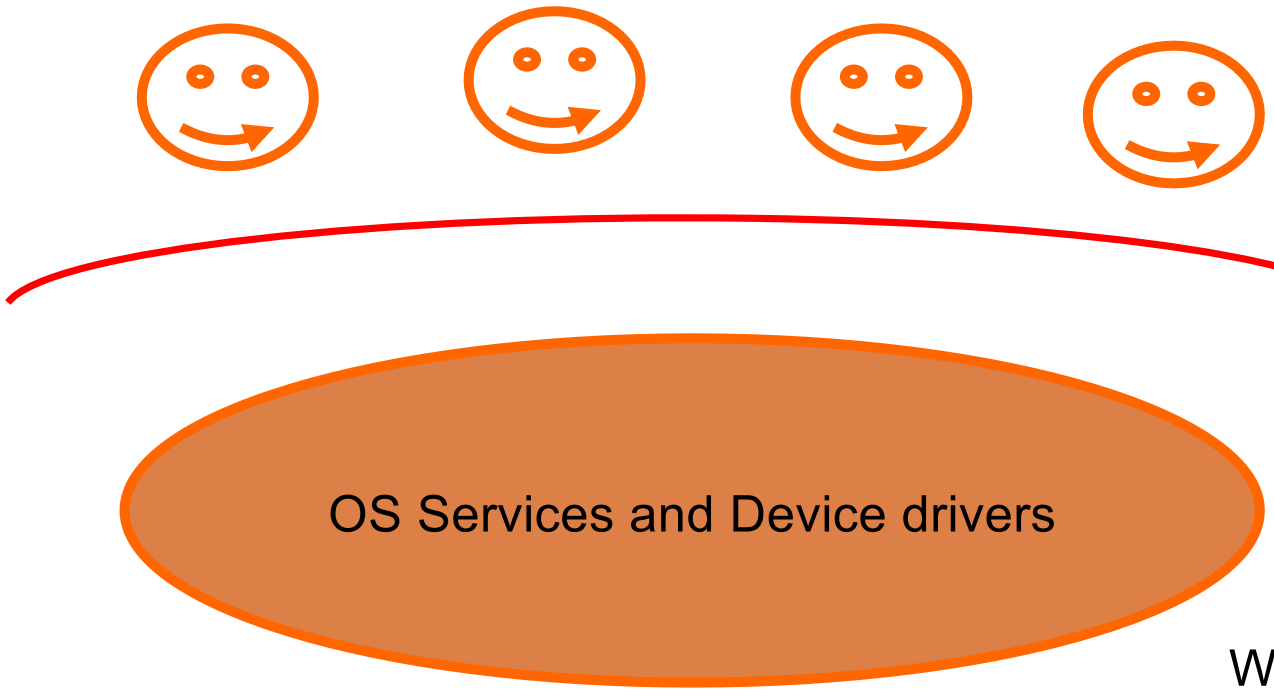
OS structures and extensibility

- Library OS
  - Customized (somewhat) to the application
- Monolithic Kernel
  - with isolated (custom) extensions
- Microkernel
  - Run external servers for chosen functionality

OS as library (DOS-like)

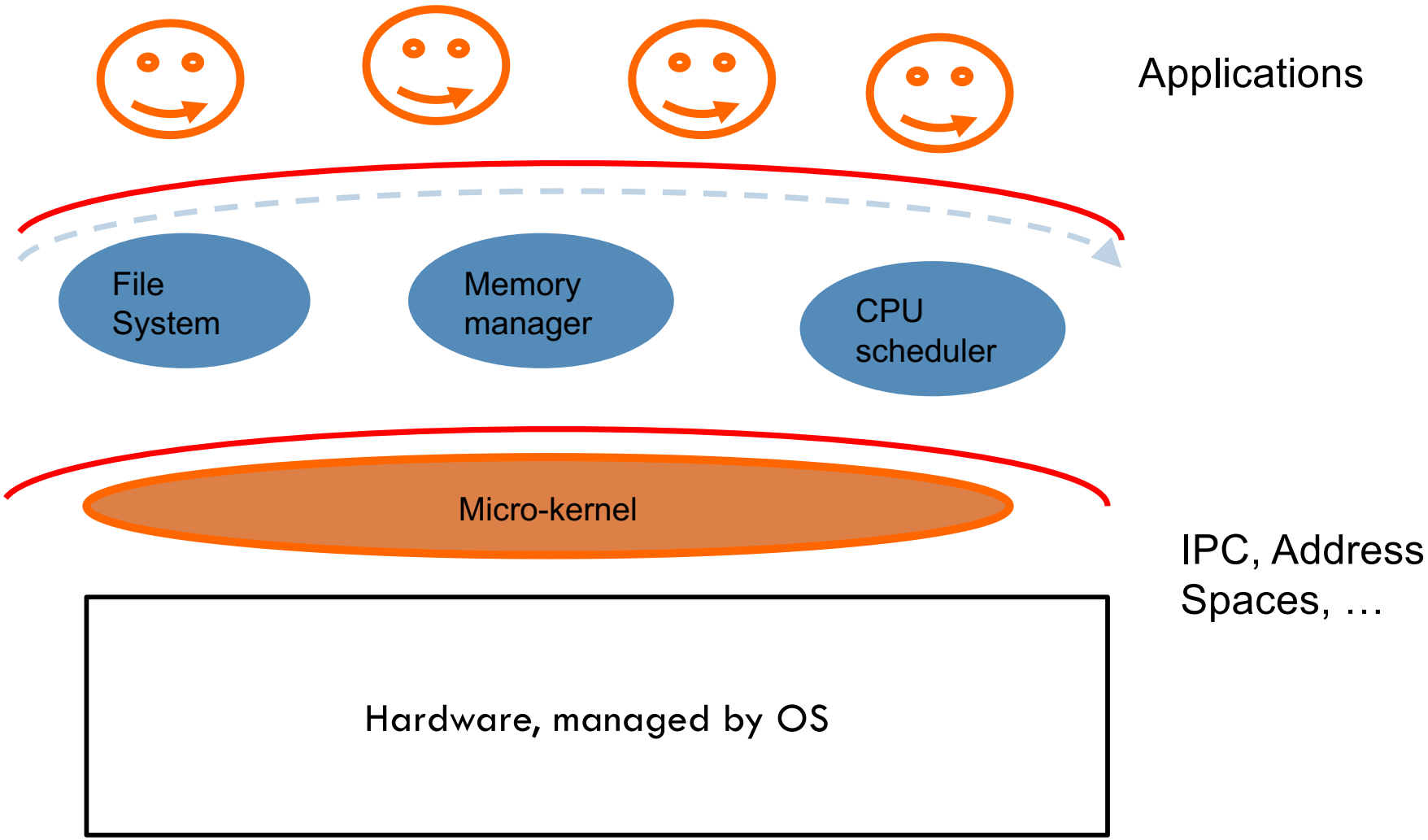


Monolithic Kernel

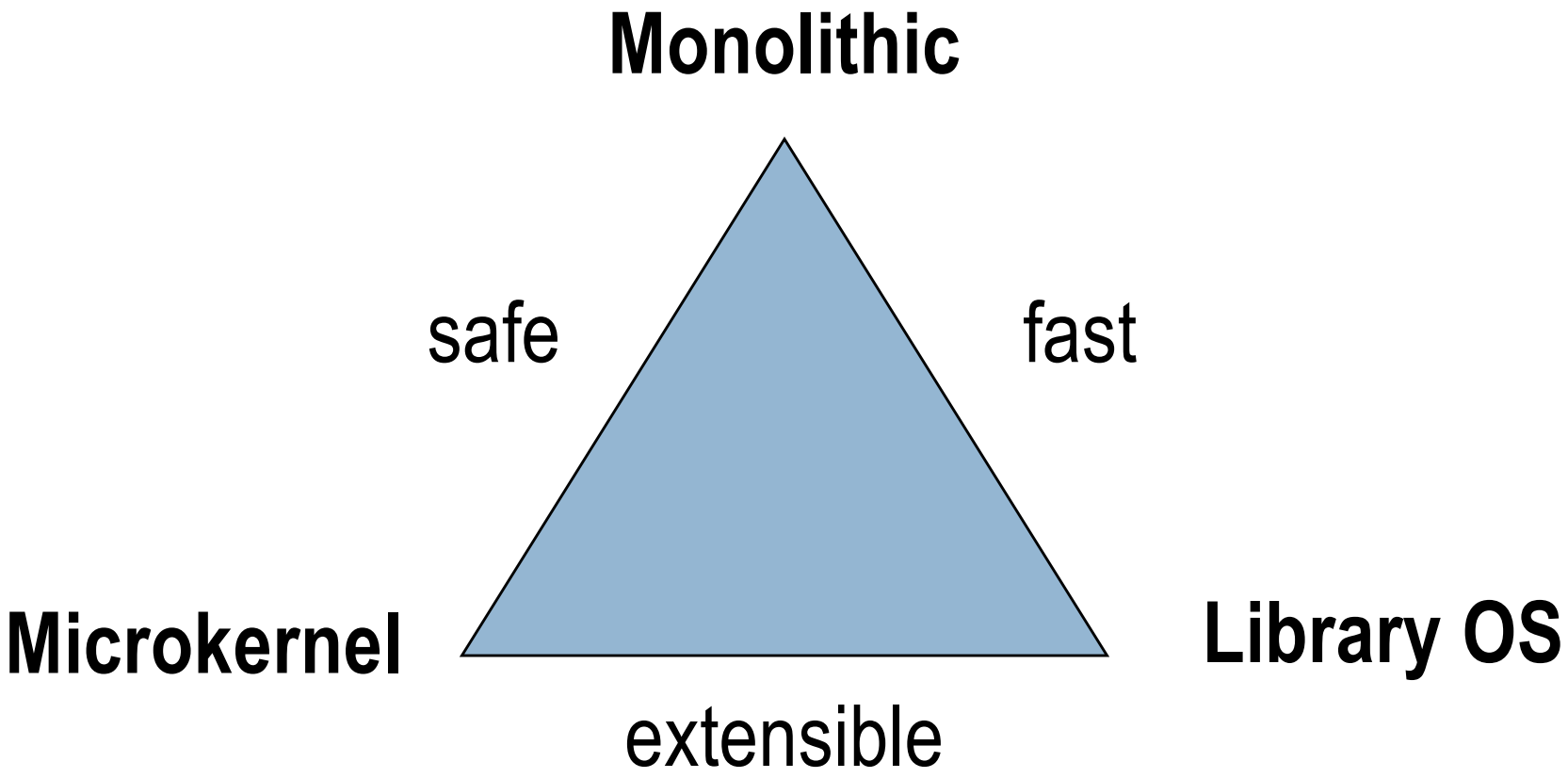


What is the difference?

Microkernel



More simply



Summary

- Library OS
  - Good performance and extensibility
  - Bad protection
- Monolithic kernels
  - Good performance and protection
  - Bad extensibility
- Microkernels
  - Very good protection
  - Good extensibility
  - Could have bad performance (L4 as counterexample)

Extensibility in Library OSes

- Exokernel (SOSP 1995): safely exports machine resources
  - Kernel only multiplexes hardware resources (Aegis)
  - Higher-level abstractions in Library OS (ExOS)
  - Secure binding, Visible resource revocation, Abort
  - Apps link with the LibOS of their choice
- Graphene (EuroSys 2014): library OS for multiple processes
  - From the traditional operating system
  - For enclave-based processing (e.g., Intel TDX and AMD SEV-SEP)

EXOKERNEL

Motivation for Exokernels

- Traditional centralized resource management cannot be specialized, extended or replaced
- Privileged software must be used by all applications
- Fixed high level abstractions too costly for good efficiency
- Exokernel as an end-to-end argument

Exokernel Philosophy

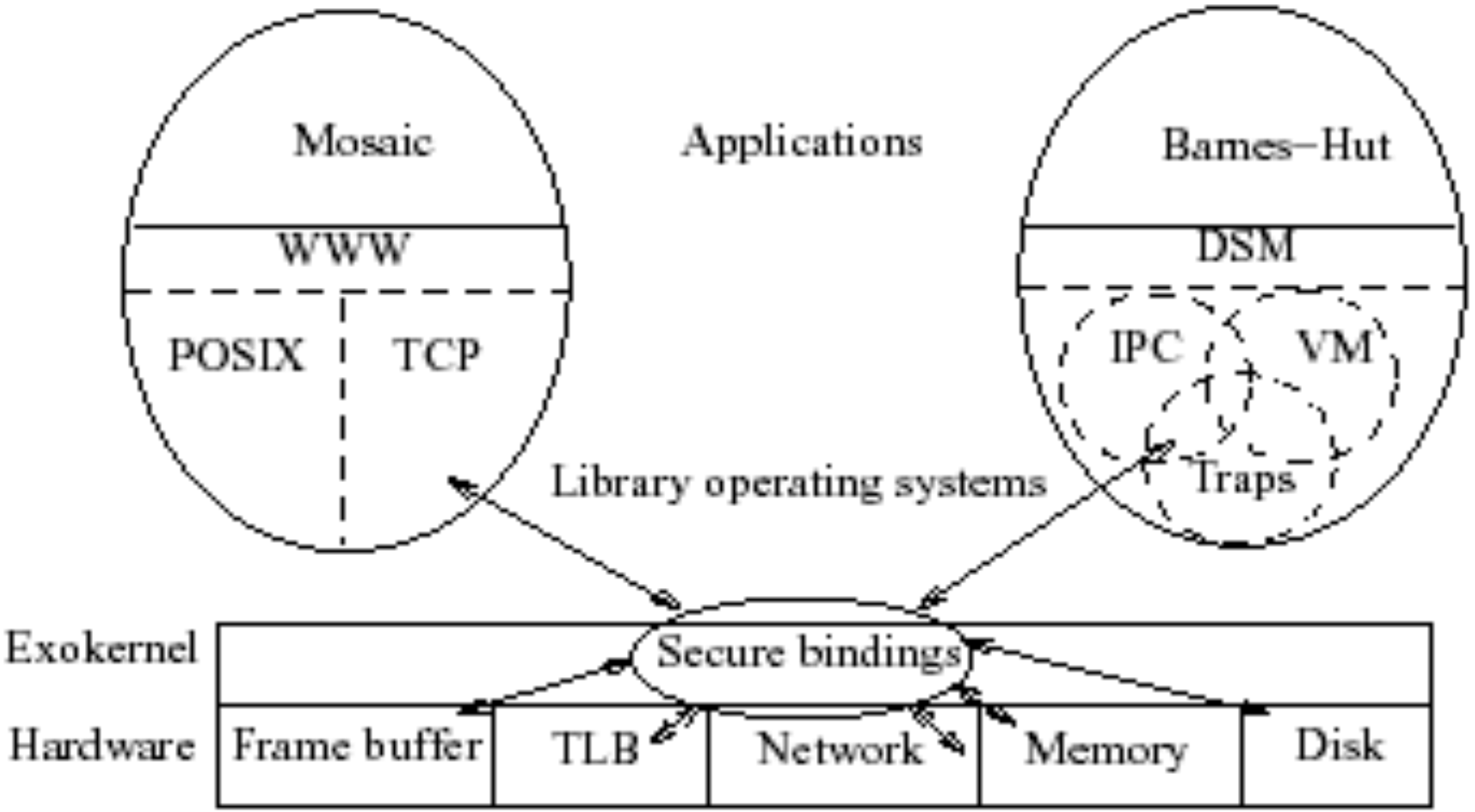
- Expose hardware to library OS
  - Not even mechanisms are implemented by exokernel
    - They argue that mechanism is policy
- Exokernel worried only about protection not resource management

Design Principles

- Track resource ownership
  - Subdivide some resources
- Ensure protection by guarding resource usage
  - Check ownership
- Revoke access to resources
- Expose hardware, allocation, names and revocation

- Basically, validate binding, then let library manage the resource

Exokernel Architecture



Core Concept: Secure Bindings

- Decouple protection from use
- Allocation to library OS at bind time
- Protection checks are simple operations performed by the kernel
  - Do you have a capability (token) that says you can operate on this resource (like a file descriptor)
- Allows protection without understanding
- Operationally – set of primitives needed for applications to control use of (subset of) resources (e.g., subset of physical pages)

Secure Binding Example (1)

- TLB Miss Processing
- TLBs
  - Cache of virtual-to-physical page mappings
- Manage TLB entries from library (app policy)
  - Virtual-to-physical mapping done by library (app)
    - E.g., App-specific policy for TLB replacement
  - Binding presented to exokernel
  - Exokernel puts app's mapping in hardware TLB
  - Library OS (app) can then manage and use caching policy without exokernel intervention

Secure Binding Example (2)

- Packet filtering
- Packet filters
  - Handle packet processing (e.g., firewall)
- Manage packet handling from library (app policy)
  - Packet filter code provided by library (app)
    - E.g., App-specific policy for packet handling
  - Binding presented to exokernel
  - Exokernel installs app's packet filter code
- Rather naïve (hopeful) about ability to insert code into the kernel without causing issues (more later)

GRAPHENE

Graphene Impact

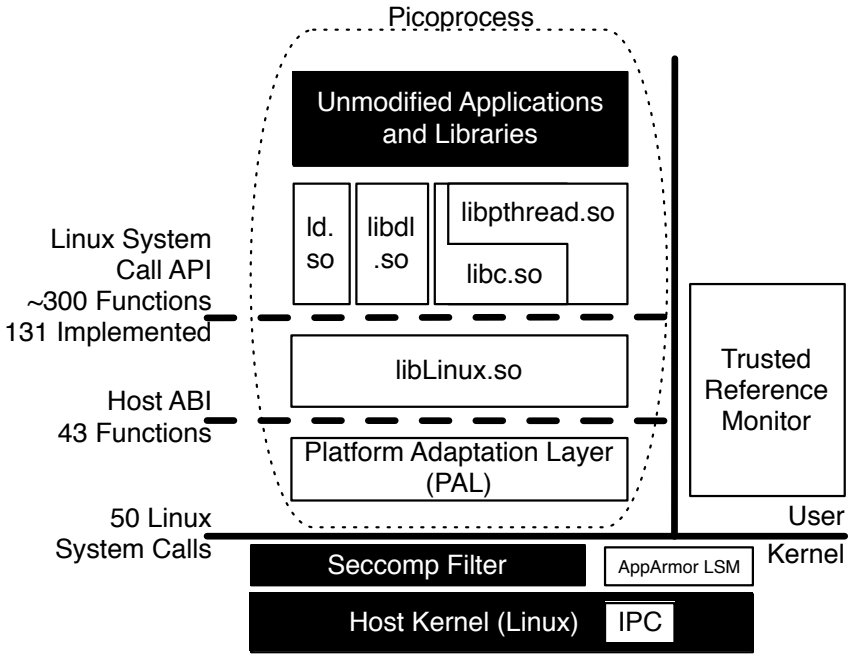
- Intel Labs recognized the potential of Graphene portability for use in confidential computing
  - SGX Enclaves – Published in USENIX ATC 2017
- Community support has created an open-source system – called Gramine
  - https://gramineproject.io
- Deployments include Azure cloud
- Joined the Confidential Computing Alliance

Goal: Lightweight Application Isolation

- By 2014, using virtualization to isolate applications has become a mature technology
  - But, it is rather expensive
  - Each VM has its own independent operating system
- Rather than run an entire virtual machine, this paper advocates providing applications (one or more processes) with their own library OS
  - Can be customized to the application
  - Contrast with container systems, which are still managed by the host OS with application-independent support

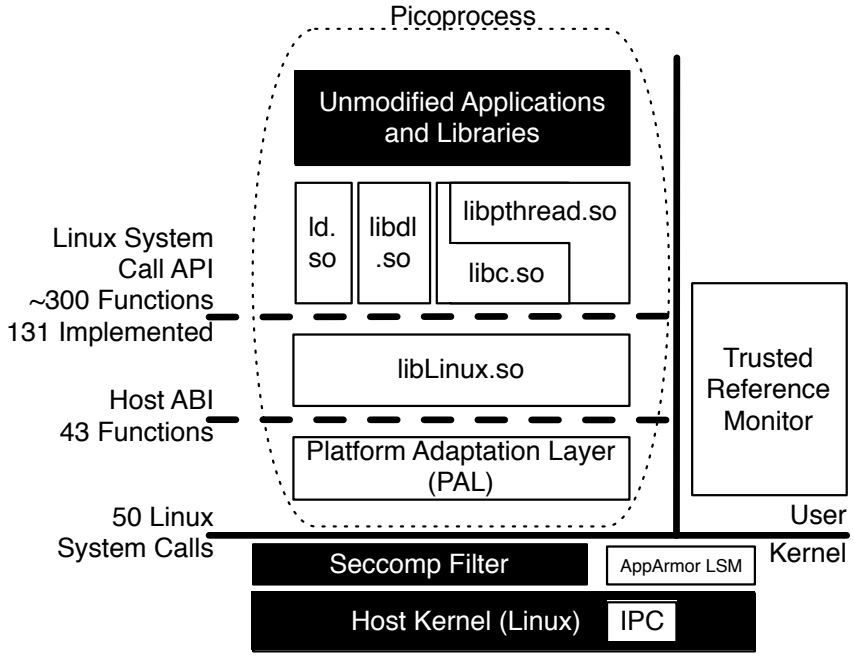
Graphene: Linux Library OS

- Graphene, a Linux library OS, which supports real-world, multi-process applications



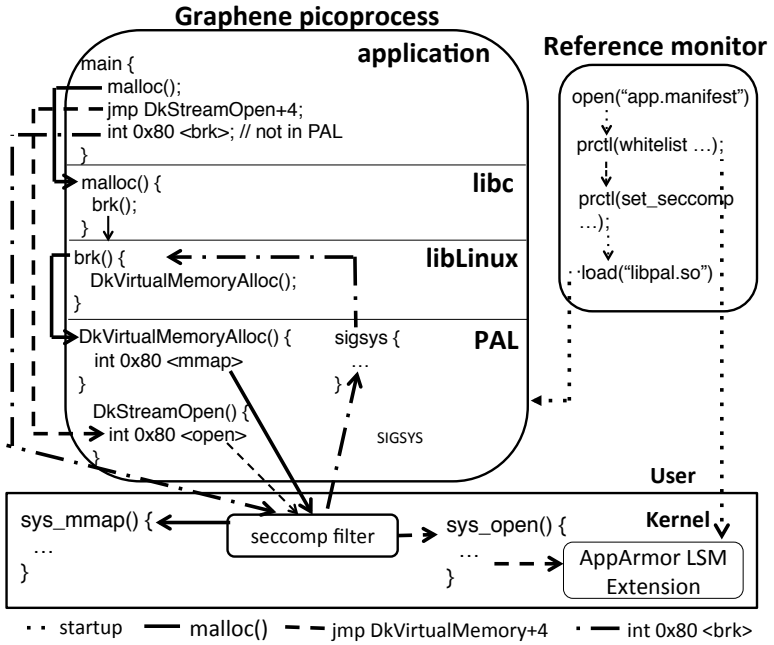
Graphene: Linux Library OS

- Applications use libLinux for syscalls, which are abstracted to the underlying OS using a Platform Adaptation Layer (PAL)



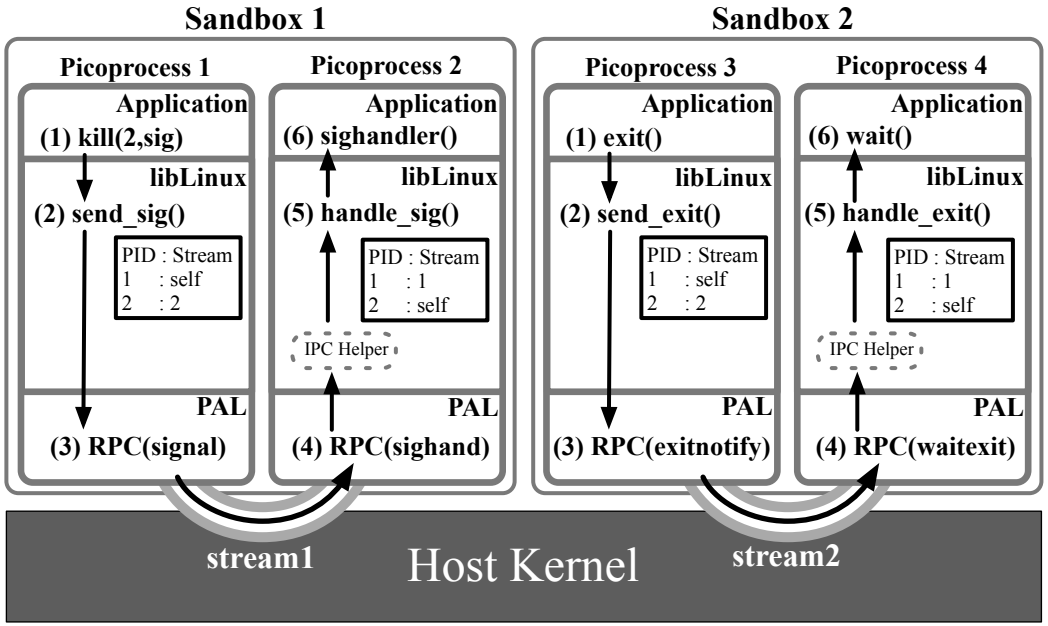
Implements and Restricts Syscalls

- A Graphene application requests OS services in three ways.
  - First, malloc (libc) invokes brk (libLinux) and mmap (PAL).
  - Second, the application jumps to an address in PAL, which is permissible. The LSM checks the open system call.
  - Third, invokes the syscall brk (w/ int 0x80), which is reflected to libLinux.



Supports Multiple "Processes"

- Two pairs of Graphene picoprocesses in different sandboxes coordinate signaling and process ID management.
  - Picoprocess 1 signals picoprocess 2 by sending a signal RPC over stream 1, and the signal is ultimately delivered using a library implementation of the sigaction interface.
  - Picoprocess 4 waits on an exitnotify RPC from picoprocess 3 over stream 2.



Performance

- Portability is the main focus
- Performance is OK, but not awesome
- But, if the application is not super-performance critical, Graphene makes it easy to deploy

	Execution time (s), +/- Conf. Interval, % Overhead								
gcc/make	**Linux**		**KVM**			**Graphene + RM**			
bzip2	2.57	.00	2.70	.00	5 %	2.70	.00	5 %	
bzip2 -j4	1.00	.00	1.09	.00	9 %	1.08	.02	8 %	
libLinux	7.23	.00	7.55	.00	4 %	8.64	.00	20 %	
libLinux -j4	1.95	.00	2.03	.00	4 %	2.54	.00	30 %	
gcc	24.74	.02	26.80	.02	8 %	31.84	.00	29 %	
Ap. Bnch	Avg Throughput (MB/s), +/- Conf. Interval, % Overhead								
Apache	**Linux**		**KVM**			**Graphene + RM**			
25 conc	5.73	.25	4.84	.03	18 %	4.02	.00	43 %	
50 conc	5.57	.28	4.80	.06	16 %	4.01	.00	39 %	
100 conc	5.87	.20	4.80	.03	22 %	3.98	.00	47 %	
lighttpd	**Linux**		**KVM**			**Graphene + RM**			
25 conc	6.66	.01	6.46	.03	3 %	5.65	.00	18 %	
50 conc	6.65	.13	6.41	.02	4 %	4.79	.00	39 %	
100 conc	6.69	.01	6.39	.03	5 %	4.56	.01	47 %	
	Execution Time (s), +/- Conf. Interval, % Overhead								
bash	**Linux**		**KVM**			**Graphene + RM**			
Unix utils	0.87	.00	1.10	.01	26 %	2.01	.00	134 %	
Unixbench	0.55	.00	0.55	.00	0 %	1.49	.00	192 %	

Table 5. Application benchmark execution times in a native Linux process, a process inside a KVM virtual machine, and a Graphene picoprocess with reference monitoring (+RM).

L4 MICROKERNELS

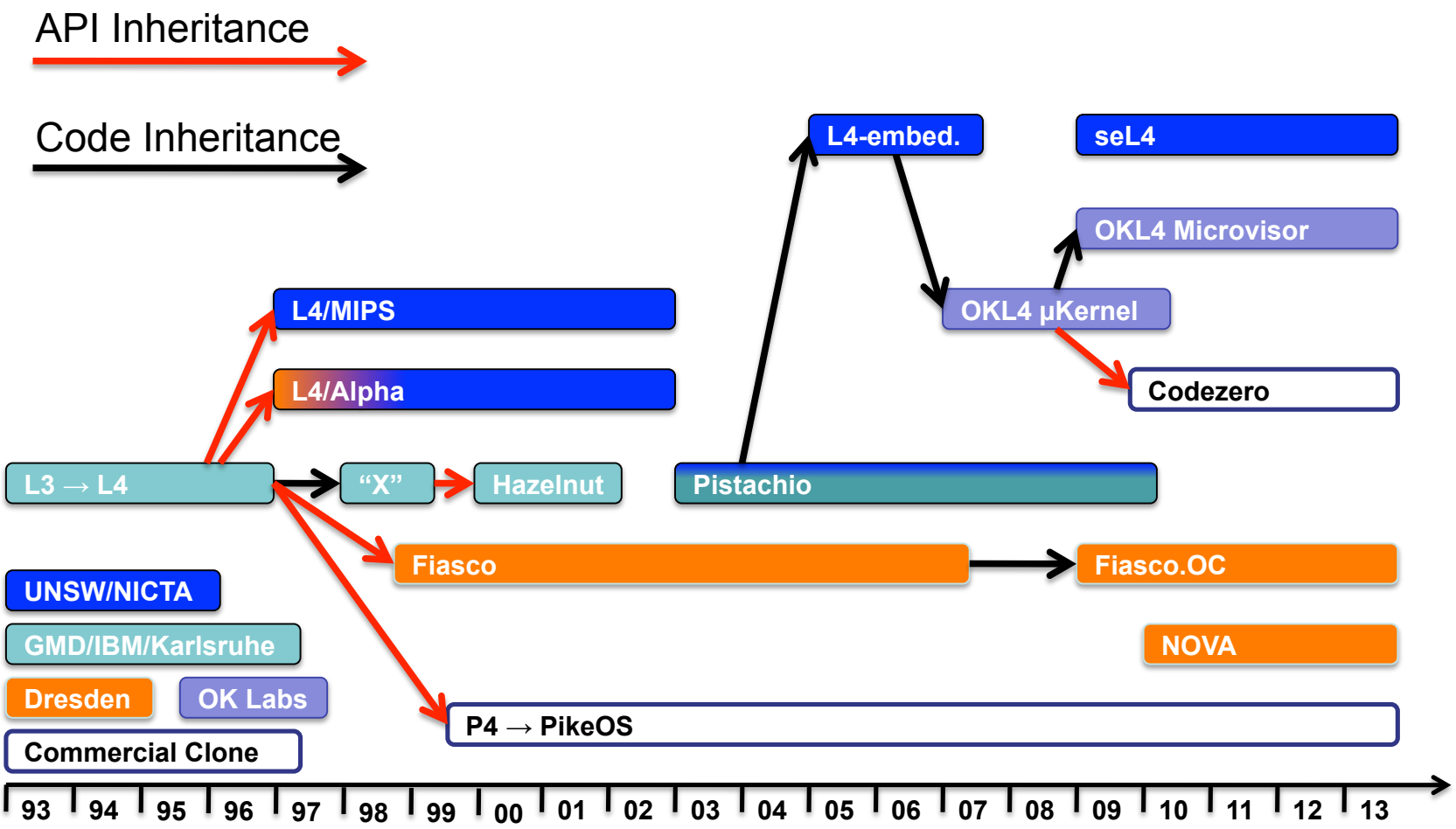
Microkernel systems

- L4 (SOSP 1995): microkernels can be high performance
  - 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
- 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

L4 microkernel family

- 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



Big picture overview

- Conventional wisdom at the time (1995) was:
  - Microkernels offer nice abstractions and should be flexible
  - …but are inherently low performance due to high cost of border crossings and IPC
  - …because they are inefficient, they are inflexible
- This paper refutes the performance argument
  - Main takeaway: its an implementation issue
    - Identifies reasons for low performance and shows by construction that they are not inherent to microkernels
      - 10-20x improvement in performance over Mach
- Several insights on how microkernels should (and shouldn't) be built
  - E.g., Microkernels should not be portable

Paper argues for the following

- 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].
- Assumes:
  - 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

Abstractions provided by L4

- Address spaces (to support protection/separation)
  - Grant, Map, Flush
  - Handling I/O
- Threads and IPC
  - Threads represent execution – used as unique identifiers
  - End point for IPC (messages)
  - Interrupts are IPC messages from kernel
    - Microkernel turns hardware interrupts to thread events

Paper argues for the following

- What is the optimal performance for critical operations
  - Dependent on the hardware
- E.g., user-kernel mode switches
  - Why do we do these?
  - Mach (older µ-kernel design) takes 18 µs for a mode switch
  - But, Intel hardware only requires 107 cycles (2 µs at 50MHz)
  - Where does the other time go?
  - Can we minimize code to support mode switching?
    - L3 could do null mode switching in 15 cycles
- Paying attention to every cycle is required at the level of design

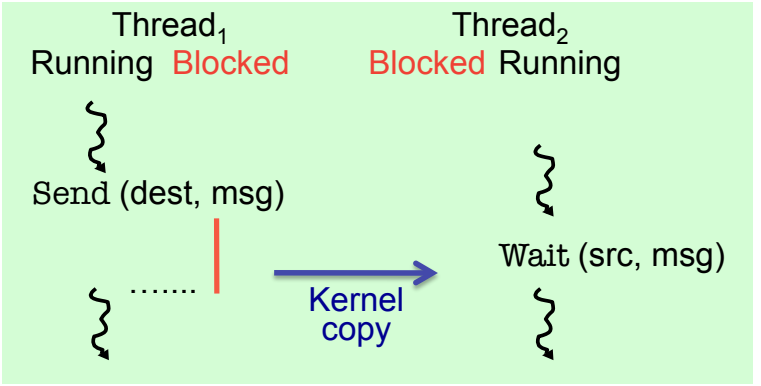
Abstractions evolved: IPC

- From synchronous IPC to asynchronous IPC

L4 Synchronous IPC

NICTA

Rendezvous model



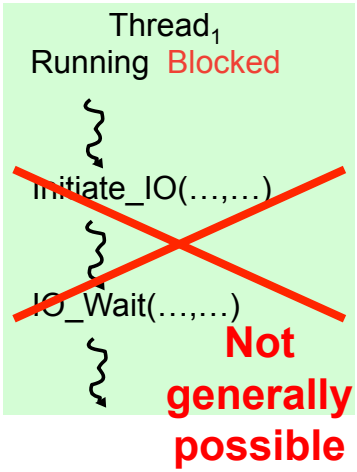
Kernel executes in sender's context
- copies memory data directly to receiver (single-copy)
- leaves message registers unchanged during context switch (zero copy)

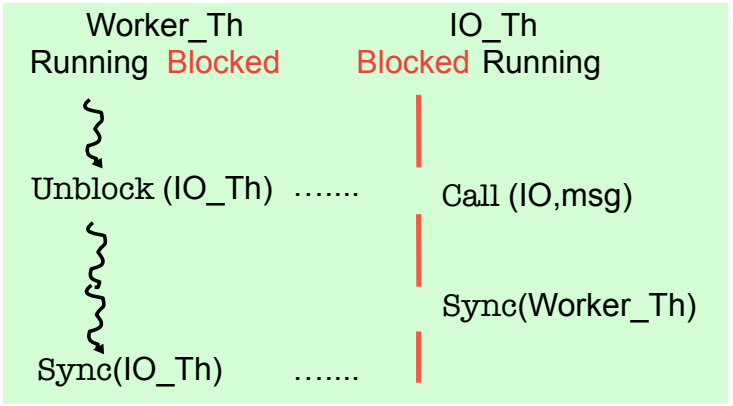
Abstractions evolved: IPC

- No method for one thread to wait for requests from clients and interrupts

Synchronous IPC Issues

NICTA

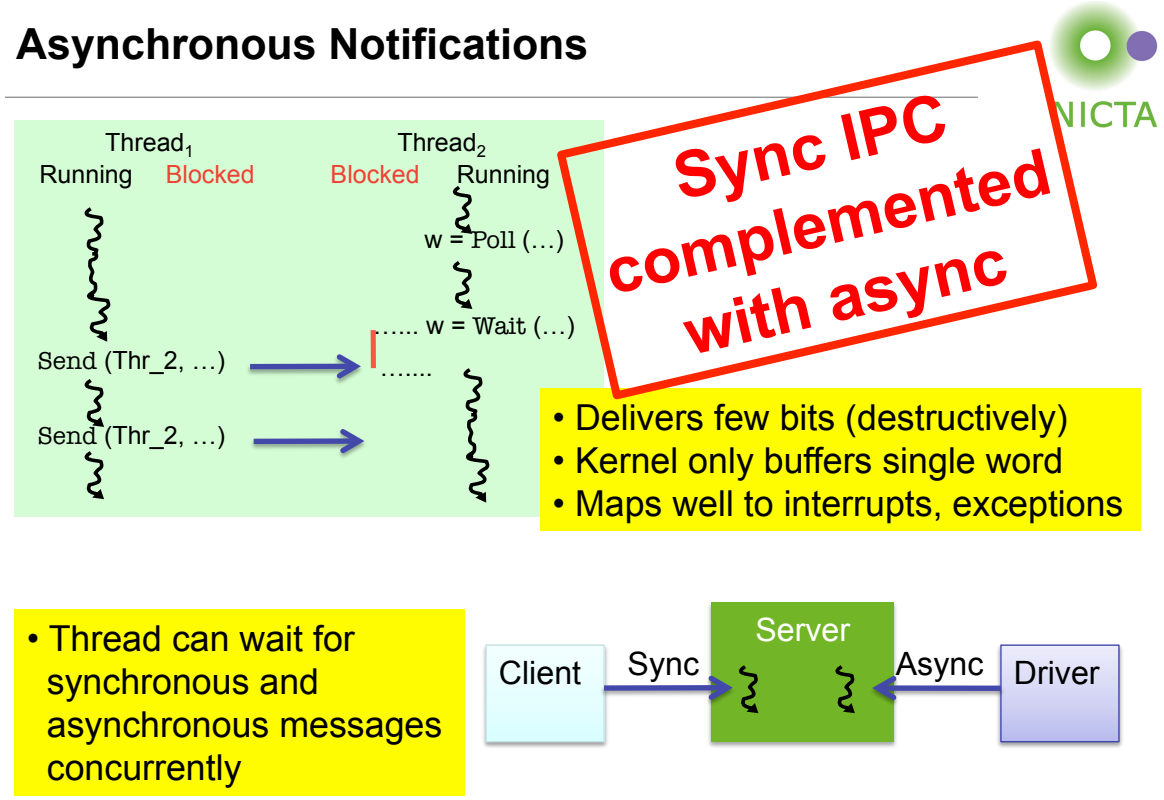




- Sync IPC forces multi-threaded code!
- Also poor choice for multi-core

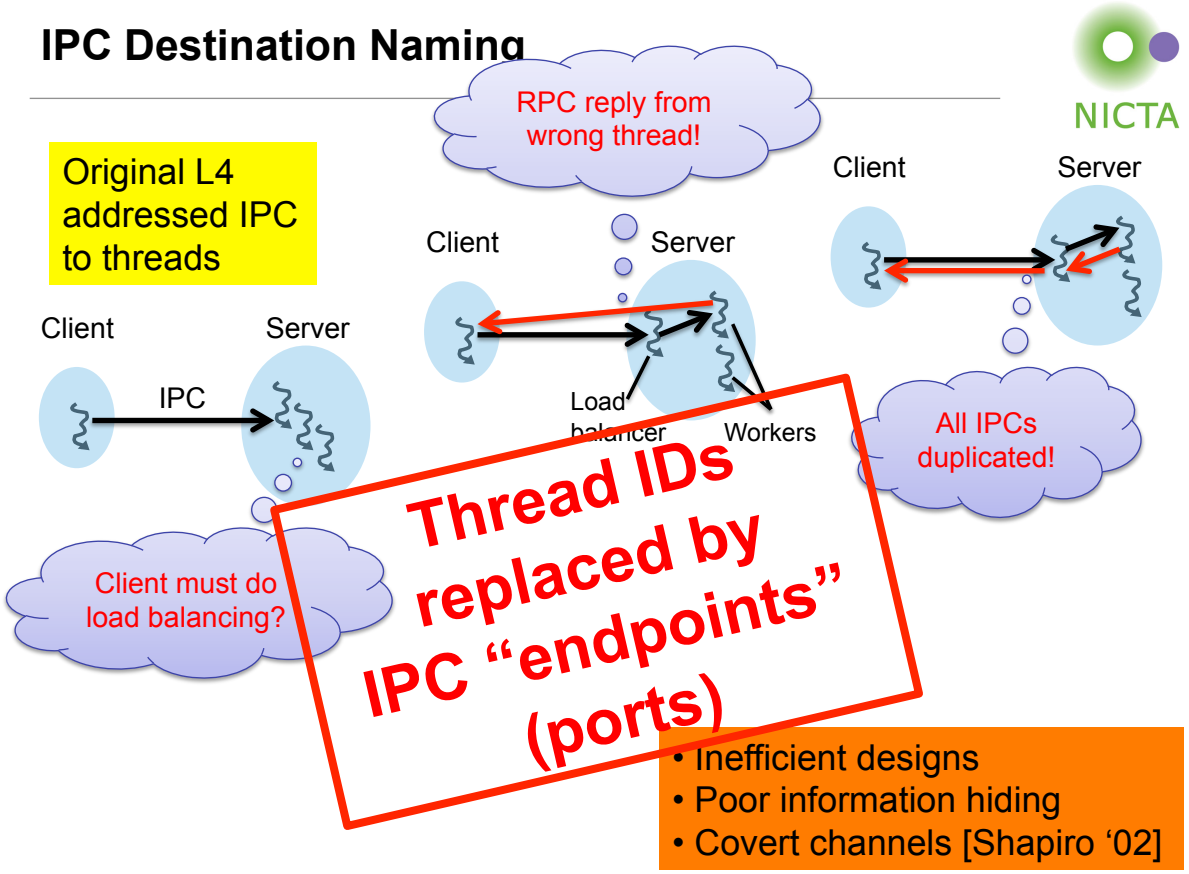
Abstractions evolved: IPC

- Asynchronous IPC allows a thread to wait for more than one



Abstractions evolved: Threads

- Using thread IDs as endpoint identifiers prevents useful types of communication patterns



Conclusions

- Today we had our first research discussion
  - On kernel structures and their implications
- Library OSes can be customized to applications
  - Run the OS in the application domain
  - Better performance and flexibility, but must manage protection
- Microkernels provide secure deployment of flexibility
  - Run application-specific services outside the base microkernel
  - Isolate services from each other, if desired
- Discuss research in the extensibility in monolithic systems next time

Questions