L4 Microkernel :: Design Overview

Jim Huang (黃敬群) <**jserv**@0xlab.org>

Developer, 0xlab

July 17, 2012 / JuluOSDev

June 11, 2012 / CSIE, CSIE

Rights to copy

contact@0xlab.org

Corrections, suggestions, contributions and translations are welcome!

Latest update: July 17, 2012

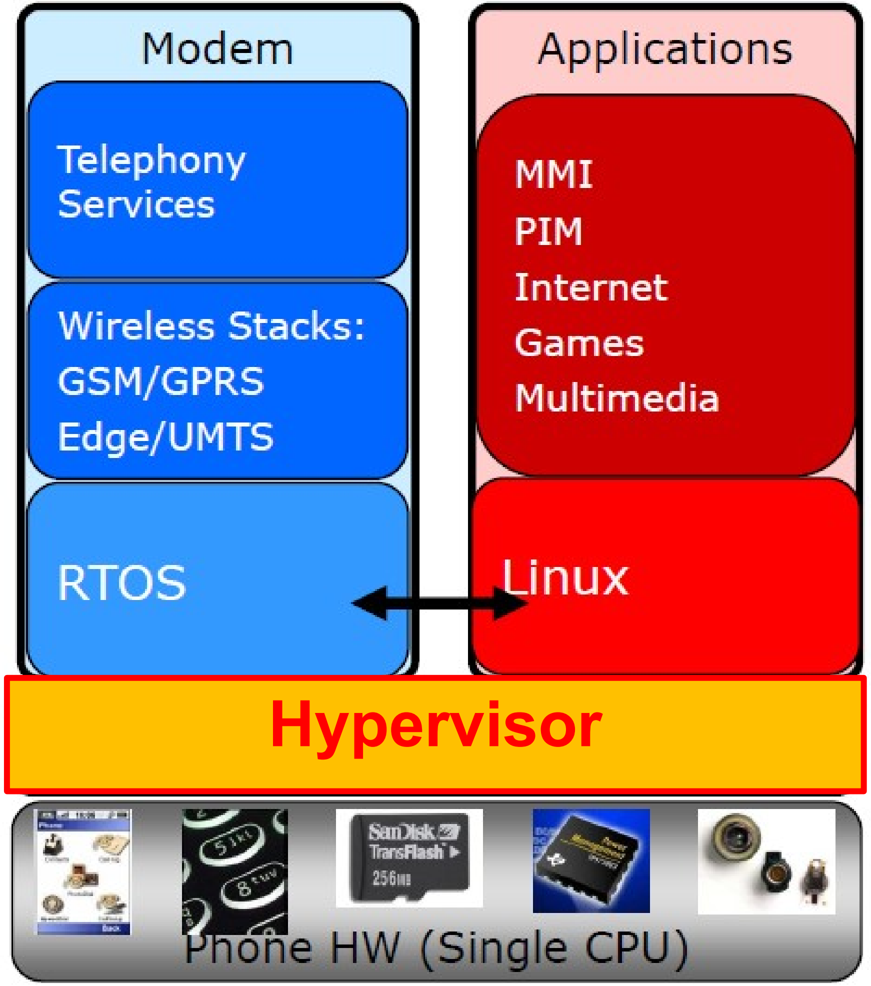
On μ-Kernel Construction

Jochen Liedtke (1953-2001)

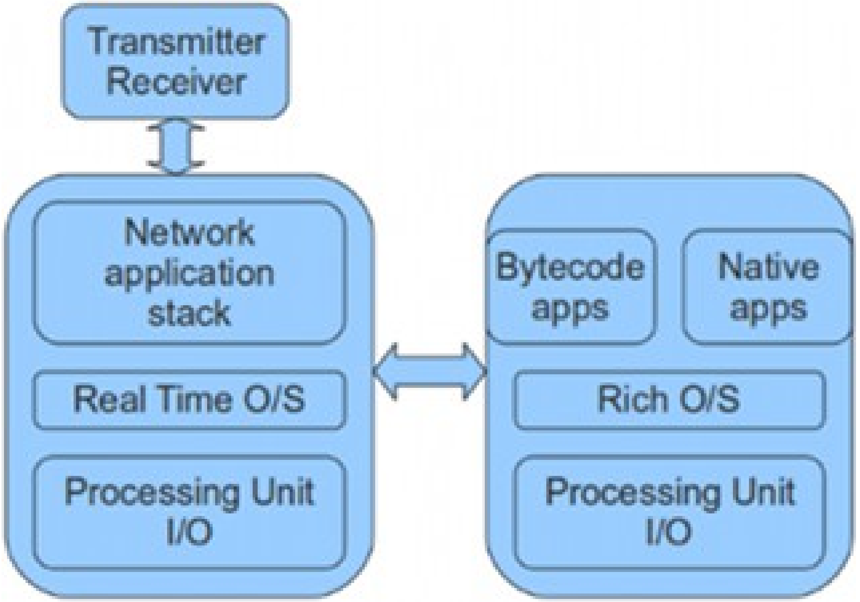
15th ACM Symposium on Operating System Principles (1995)

Use Case: Low-cost 3G Handset

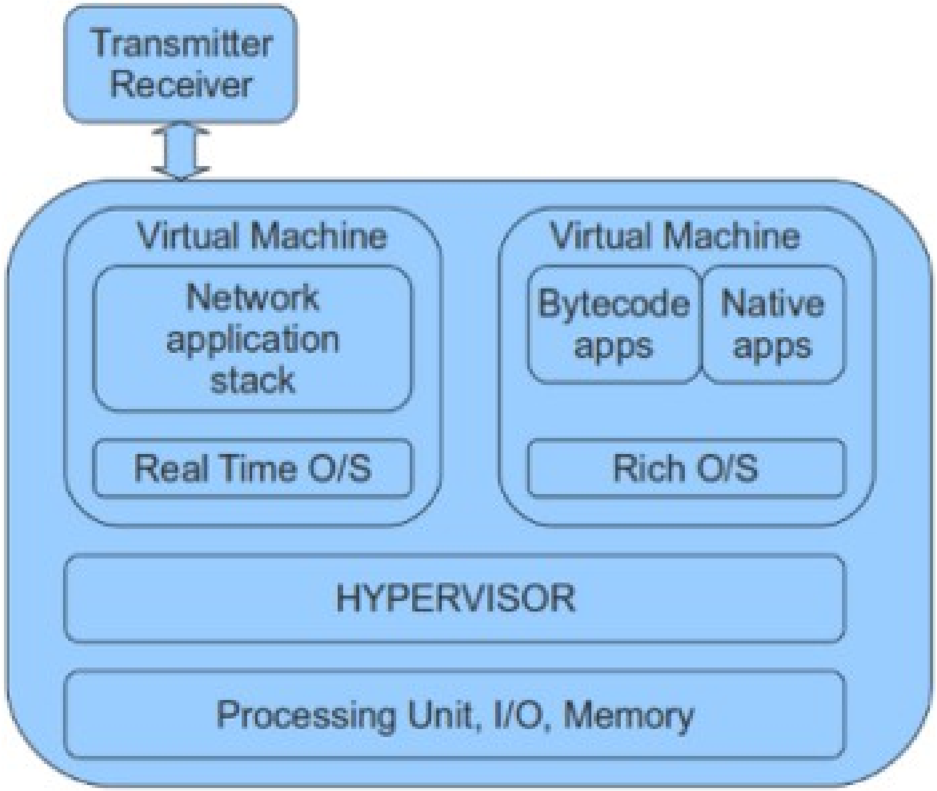
- Mobile Handsets
  - Major applications runs on Linux
  - 3G Modem software stack runs on RTOS domain
- Virtualization in multimedia Devices
  - Reduces BOM (bill of materials)
  - Enables the Reusability of legacy code/applications
  - Reduces the system development time
- Instrumentation, Automation
  - Run RTOS for Measurement and analysis
  - Run a GPOS for Graphical Interface



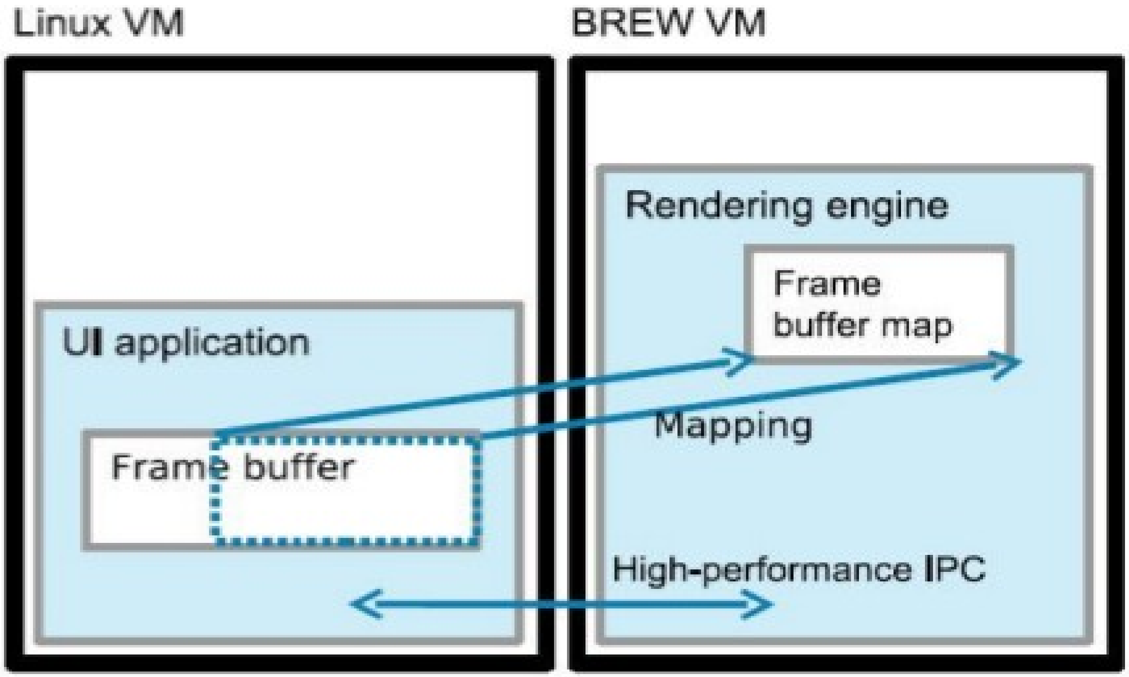
original mobile phone:
two CPUs required



with Virtualization: single chip



- Evoke’s UI functionalities including the touch screen is owned by the Linux apps while video rendering uses a rendering engine running on BREW.
- When a user requests a BREW app, Linux communciates with BREW in the other VM to start up the app. The BREW obtains access to the screen by using a frame buffer from a shared-memory mapping.



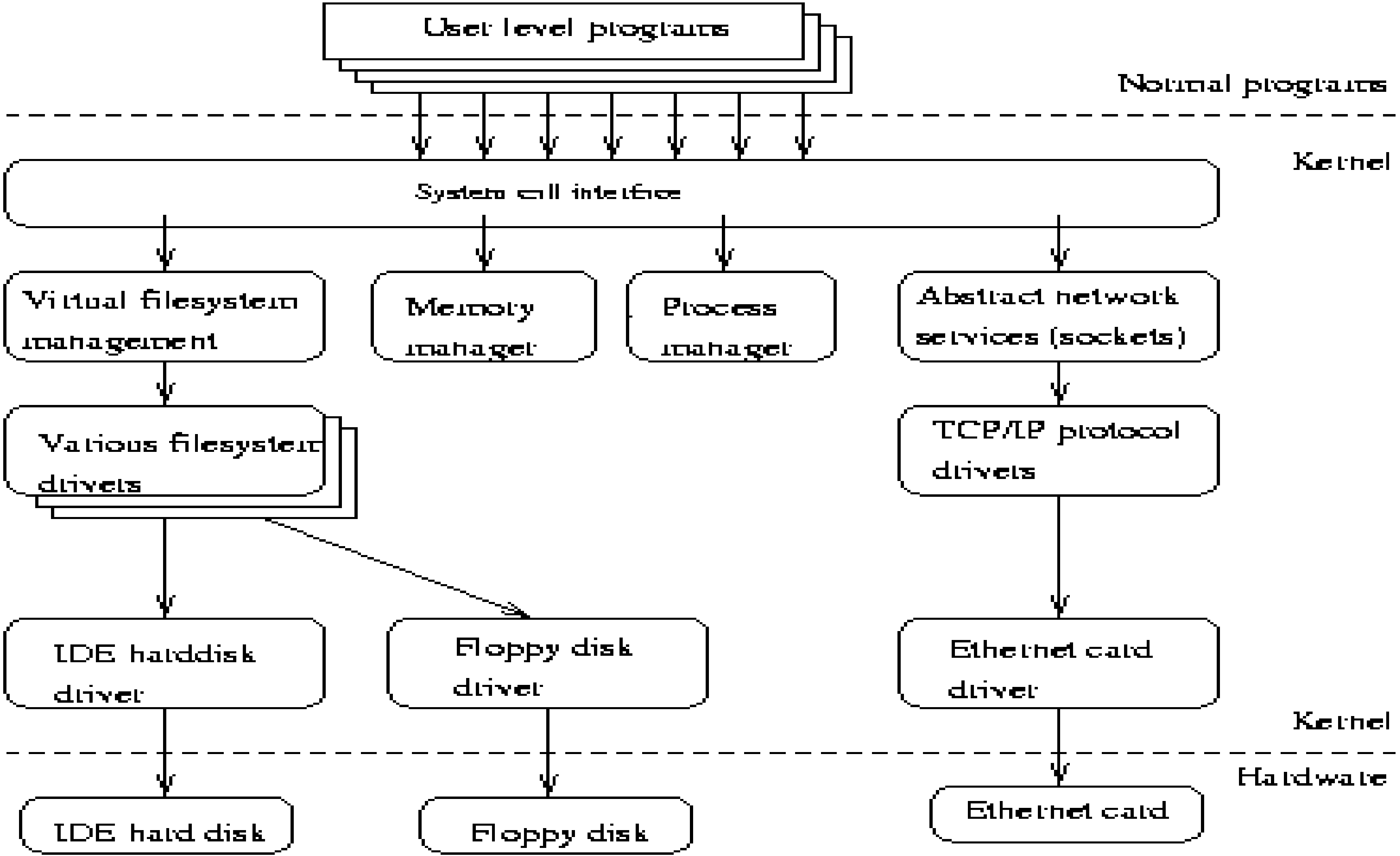
Agenda

- Myths of Microkernel
- Characteristics of 2nd generation microkernel
  - memory, thread, IPC management
- Toward 3rd generation microkernel
- Real-world Deployment

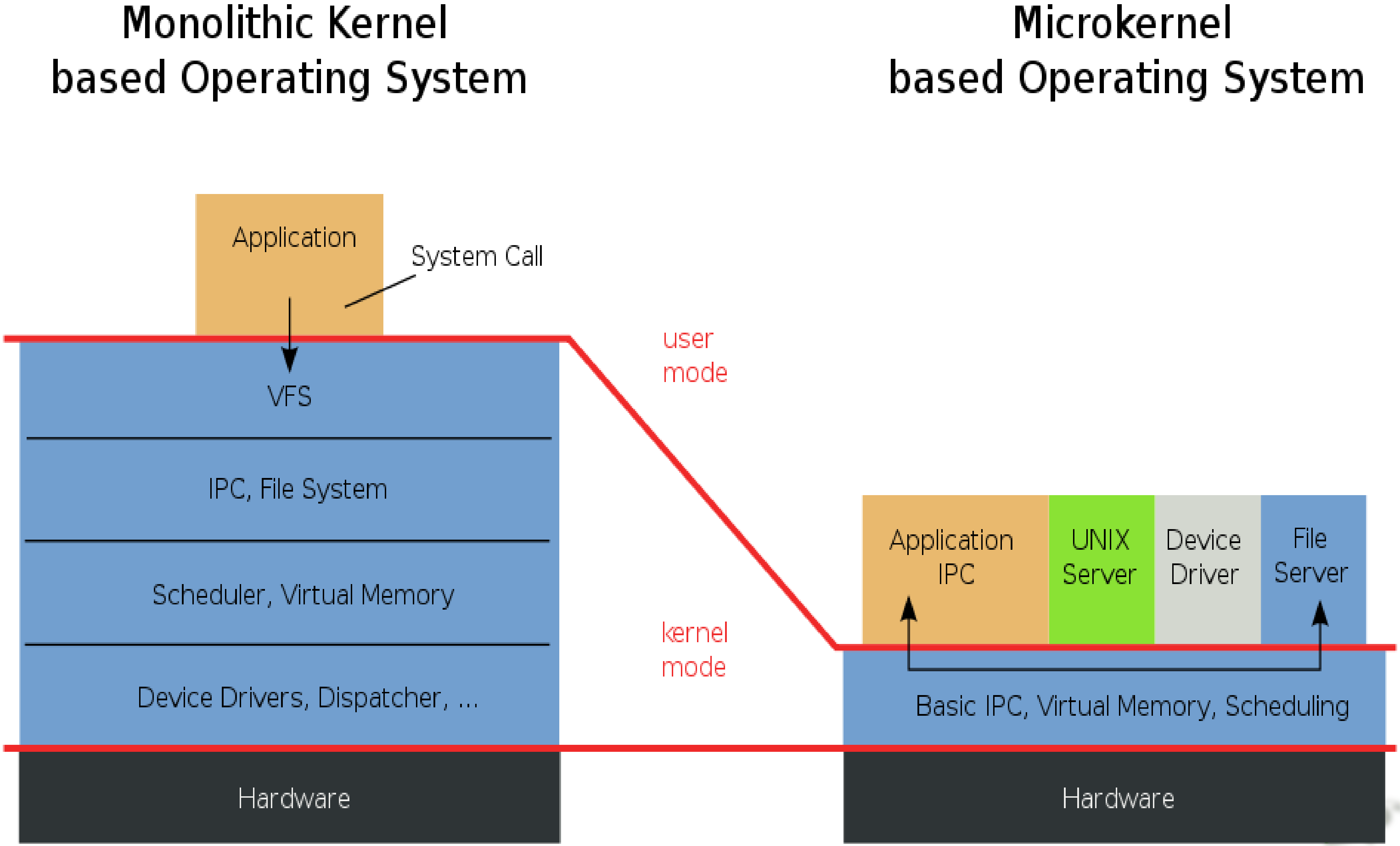
Myths of Microkernel

Definition of Kernel

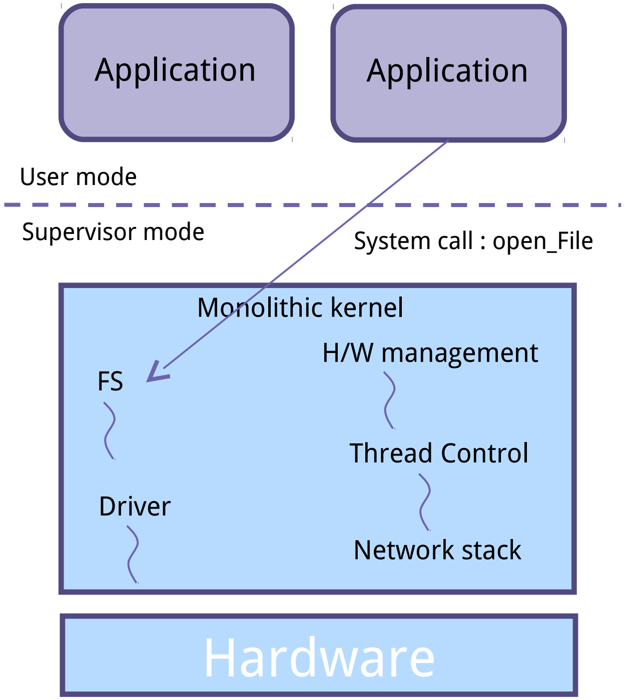
- The fundamental part of an Operating System.
- Responsible for providing secure access to the machine's hardware for various programs.
- Responsible for deciding when and how long a program can use a certain hardware (multiplexing).

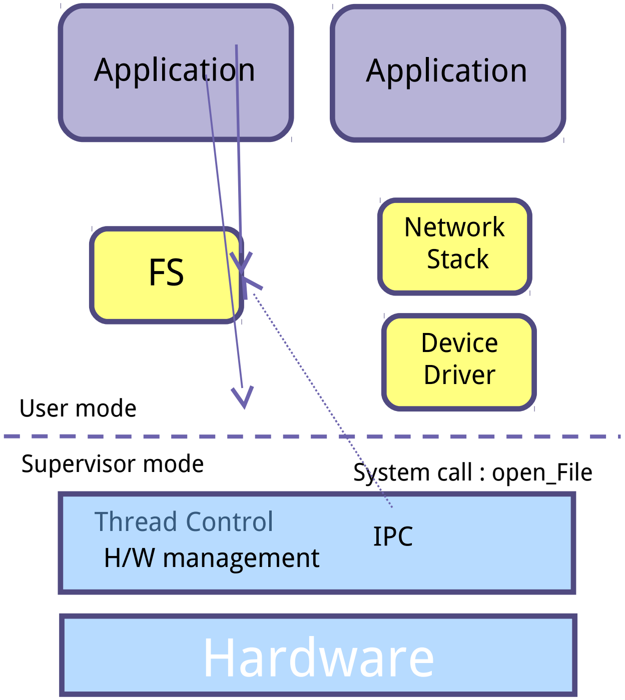


Monolithic vs. Microkernel



Monolithic vs. Microkernel





Hybrid Kernel

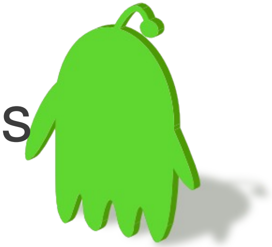
- Combine the best of both worlds
  - Speed and simple design of a monolithic kernel
  - Modularity and stability of a microkernel
- Still similar to a monolithic kernel
  - Disadvantages still apply here
- Example: Windows NT, BeOS, DragonFlyBSD

Exokernel

- Follows end-to-end principle
  - Extremely minimal
  - Fewest hardware abstractions as possible
  - Just allocates physical resources to apps
- Old name(s): picokernel, nanokernel
- Example: MIT Exokernel, Nemesis, ExOS

Kernel Comparison

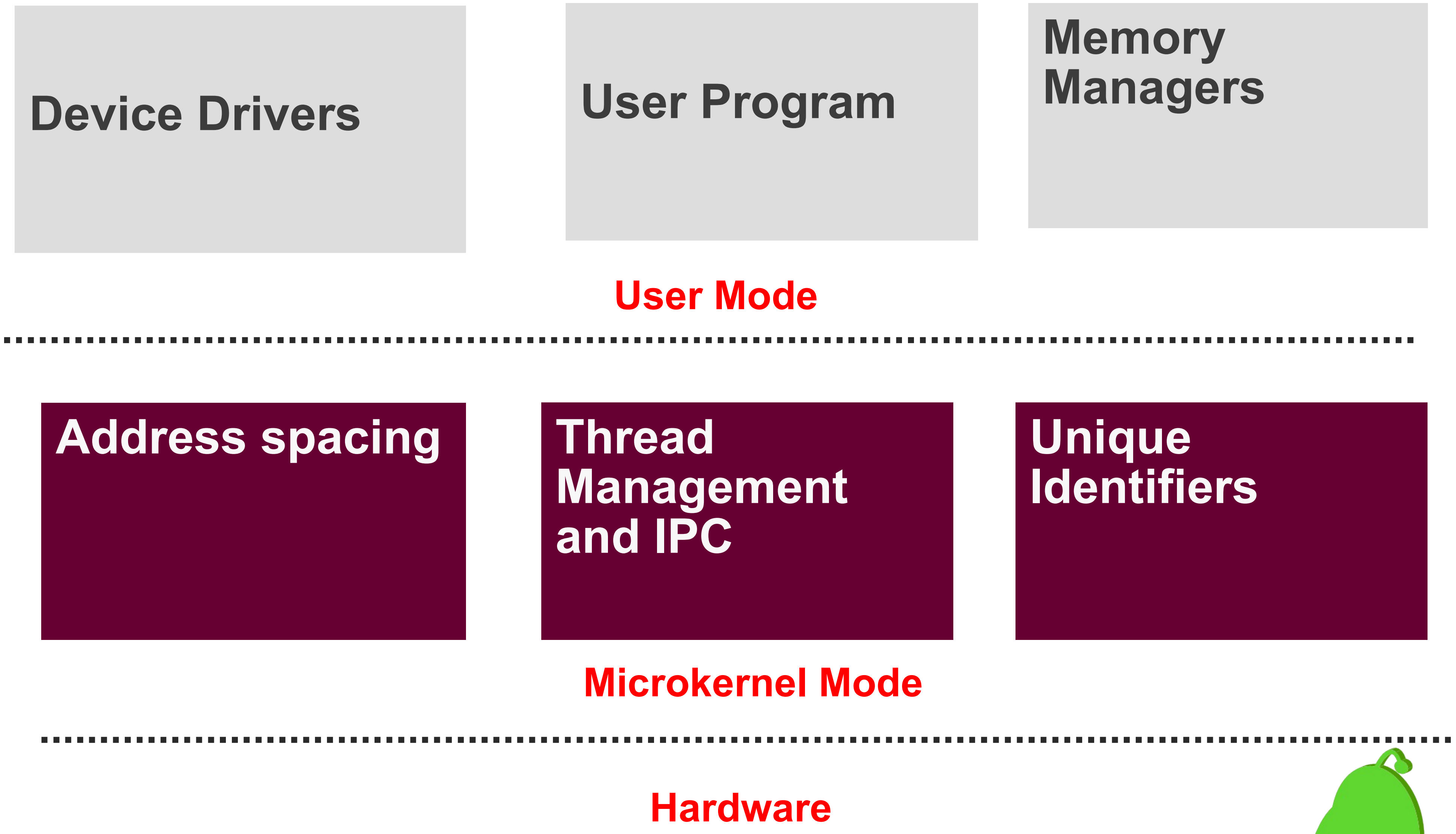
- Monolithic kernels
  - Advantages: performance
  - Disadvantages: difficult to debug and maintain
- **Microkernels**
  - **Advantages: more reliable and secure**
  - **Disadvantages: more overhead**
- Hybrid Kernels
  - Advantages: benefits of monolithic and microkernels
  - Disadvantages: same as monolithic kernels
- Exokernels
  - Advantages: minimal and simple
  - Disadvantages: more work for application developers



Definition of Microkernel

- A kernel technique that provides only the minimum OS services.
  - Address Spacing
  - Inter-process Communication (IPC)
  - Thread Management
  - Unique Identifiers
- All other services are done at user space independently.

Microkernel



Device Drivers

User Program

Memory Managers

User Mode

Address spacing

Thread Management and IPC

Unique Identifiers

Microkernel Mode

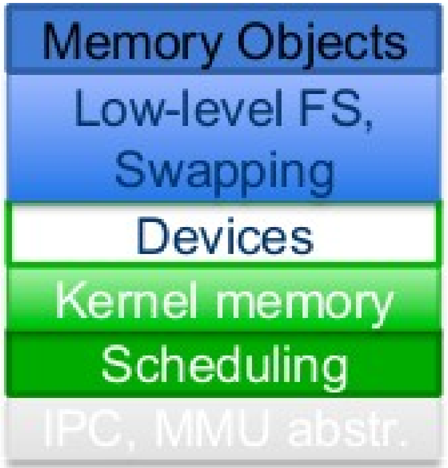
Hardware

Microkernel Advantage

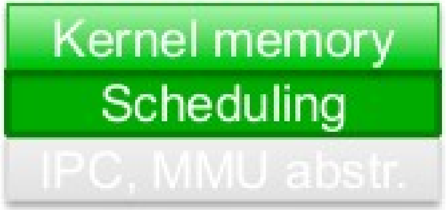
- A clear microkernel interface enforces a more modular system structure
- Servers can use the mechanisms provided by the microkernel like any other user program.
- So server malfunction is as isolated as any other user program’s malfunction
- The system is more flexible and tailorable. Different strategies and APIs, implemented by different severs, can coexist in the system

3 Generations of Microkernel

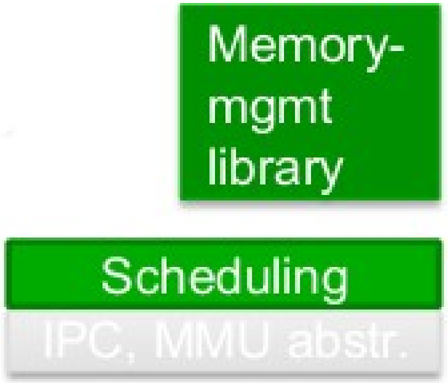
- Mach, Chorus (1985-1994)
  - replace pipes with IPC (more general)
  - improved stability (vs monolithic kernels)
  - poor performance



- L3 & L4 (1990-2001)
  - Large improvements in IPC performance
  - Written in assembly, poor portability
  - only synchronus IPC (build async on top of sync)
  - very small kernel: more functions moved to userspace



- seL4, Coyotos, Nova (2000-present)
  - platform independence
  - verification, security, multiple CPUs, etc.



1st Generation: Chorus Nucleus

- Supervisor
  - Dispatches traps, interrupts, and exceptions delivered by hardware.
- Real Time Executive
  - Controls allocation of processes and provides preemptive scheduling
- Virtual Memory Manager
  - Manipulates VM hardware and memory resources.
- IPC
  - Provides message Exchanging and Remote Procedure Calls (RPC).

1st Generation: CMU Mach

- Asynchronous IPC
- Threads
- Scheduling
- Memory management
- Resource access permissions
- Device drivers (in some variants)

  (All other functions are implemented outside kernel.)

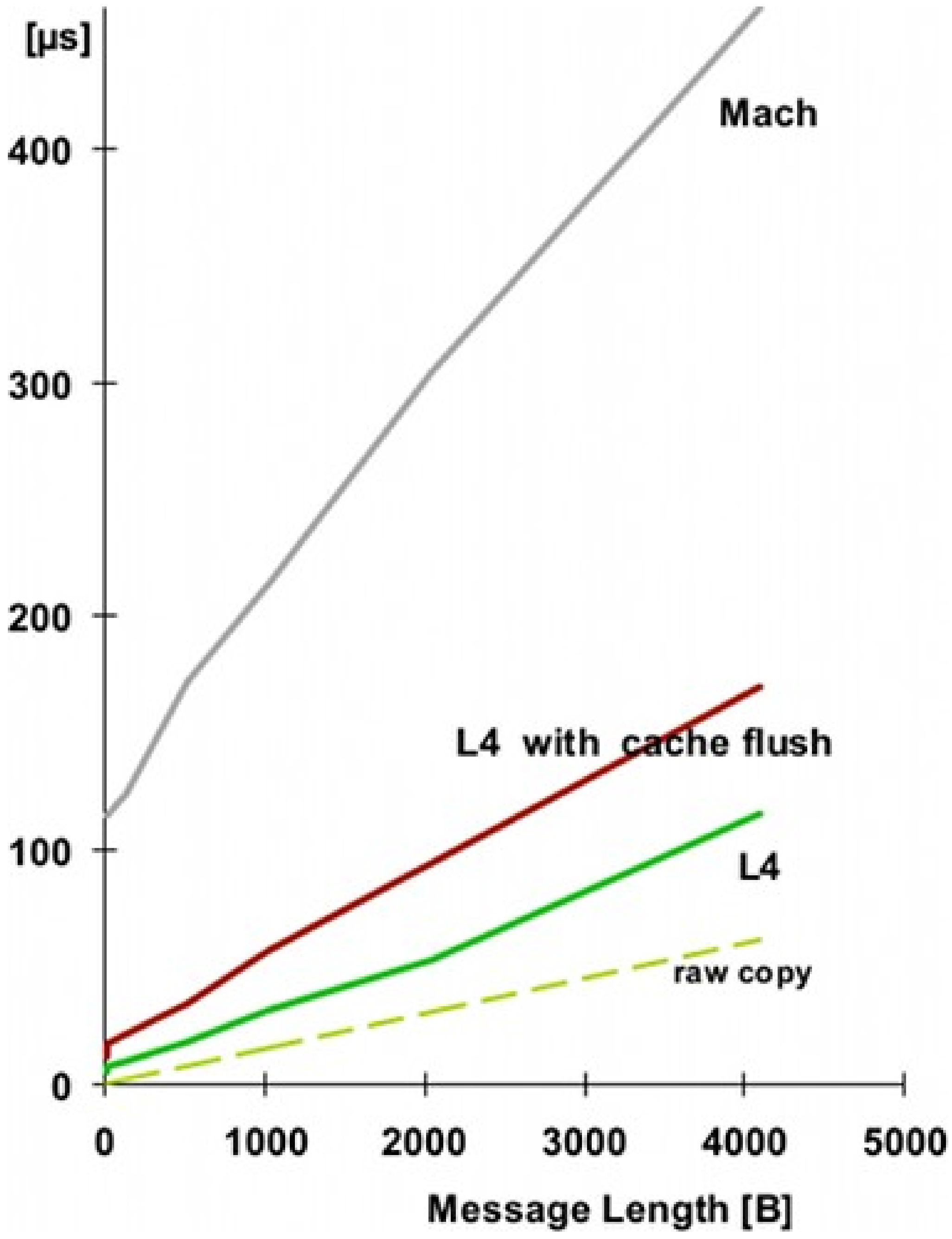
- API Size of Mach 3: 140 functions

Mach microkernel performance issues

- Checking resource access permissions on system calls.
  - Single user machines do not need to do this.
- Cache misses
  - Critical sections were too large.
- Asynchronus IPC
  - Most calls only need synchronus IPC.
  - Synchronous IPC can be faster than asynchronous.
  - Asynchronous IPC can be built on top of synchronous.
- Virtual memory
  - How to prevent key processes from being paged out?

2nd Generation: L4

- “Radical” approach
- [Liedtke’93, Liedtke ‘95]:
- Strict minimality
- From-scratch design
- Fast primitives



3rd Generation: seL4

- [Elphinstone et al 2007, Klein et al 2009]
- Security-oriented design
  - capability-based access control
  - strong isolation
- Hardware resources subject to user-defined policies
  - including kernel memory (no kernel heap)
  - except time
  - “Microhypervisor" concept
- Designed for formal verification

Classical L4 microkernel functionality

- Threads
- Scheduling
- Memory management
- (All other functions are implemented outside kernel)

- API size of L4: 7 functions
  - Compare to 140 functions for Mach3

L4 Mimnimality Principle

- A concept is tolerated inside the microkernel only if moving it outside the kernel, *i.e.*, permitting competing implementations, would prevent the implementation of the system's required functionality.
- Fred Books on *conceptual integrity* [Mythical Man Month]
  - UNIX : Everything is a file
  - Mach : IPC generalizes files
  - L4 : Can it be put outside the kernel?

L4 Kernel size

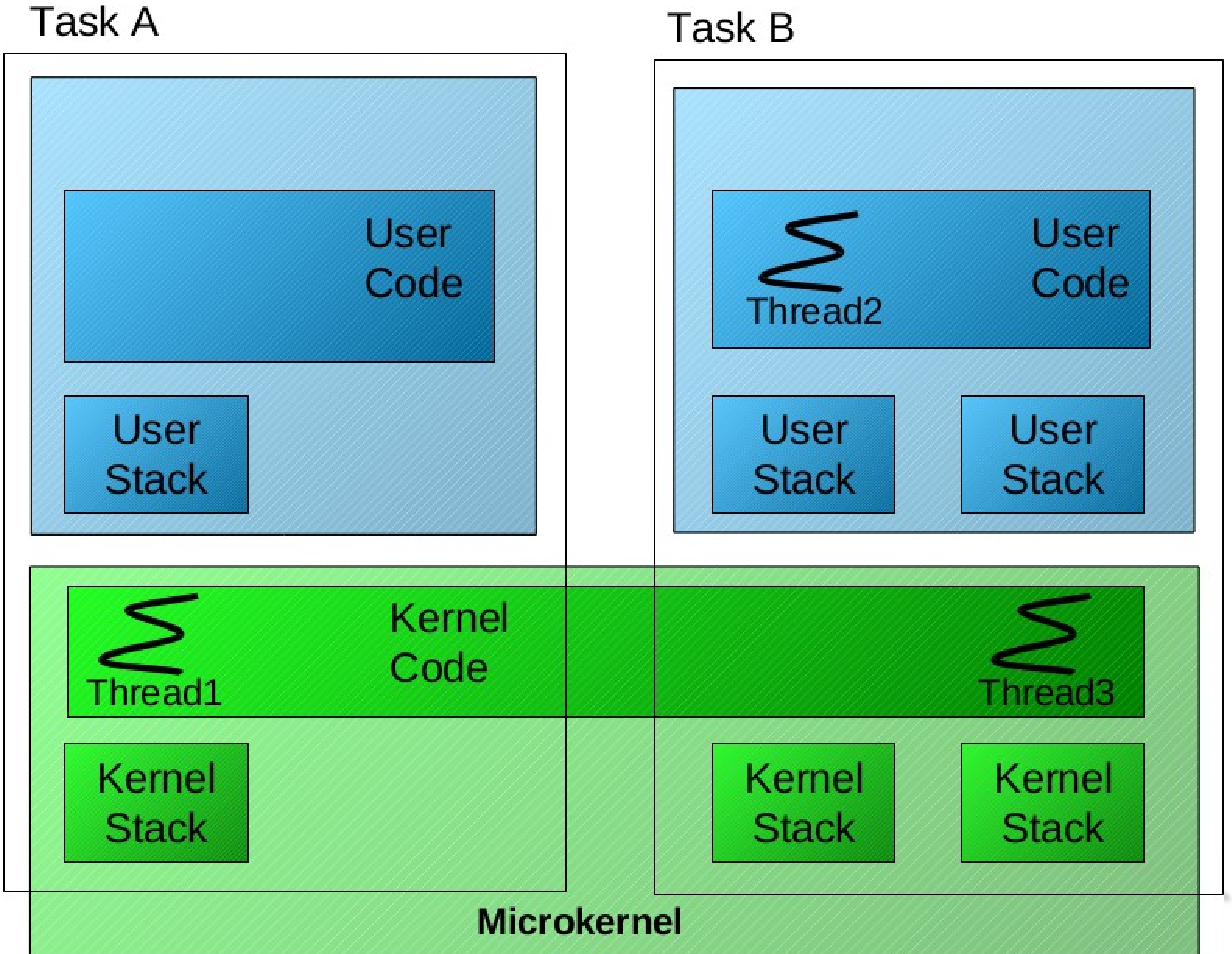
- Line of Code in OKL4
  - ~9k LOC architecture-independent
  - 0.5–6k LOC architecture/platform-specific
- Memory footprint kernel (not aggressively minimized):
  - Using gcc (poor code density on RISC/EPIC architectures)

Architecture	Version	Text	Total
X86	L4Ka	52k	98k
Itanium	L4Ka	173k	417k
ARM	OKL4	48k	78k
PPC-32	L4Ka	41k	135k
PPC-64	L4Ka	60k	205k
MIPS-64	NICTA	61k	100k

What properties do we expect from Kernel?

- Every system call terminates
- No exceptions thrown
- No arithmetic problems (e.g., overflow, divide by zero)
- No null pointer de-references
- No ill-typed pointer de-references
- No memory leaks
- No buffer overflows
- No unchecked user arguments
- Code injection attacks are impossible
- Well-formed data structures
- Correct book-keeping
- No two objects overlap in memory

Characteristics of second generation microkernel: memory, thread, IPC management


Task A
Task B
User Code
User Stack
Thread2
User Code
User Stack
User Stack
Kernel Code
Thread1
Thread3
Kernel Stack
Kernel Stack
Kernel Stack
Microkernel

Threads

- Represent unit of execution
  - Execute user code (application)
  - Execute kernel code (system calls, page faults, interrupts, exceptions)
- Subject to scheduling
  - Quasi-parallel execution on one CPU
  - Parallel execution on multiple CPUs
  - Voluntarily switch to another thread possible
  - Preemptive scheduling by the kernel according to certain parameters
- Associated with an address space
  - Executes code in one task at one point in time

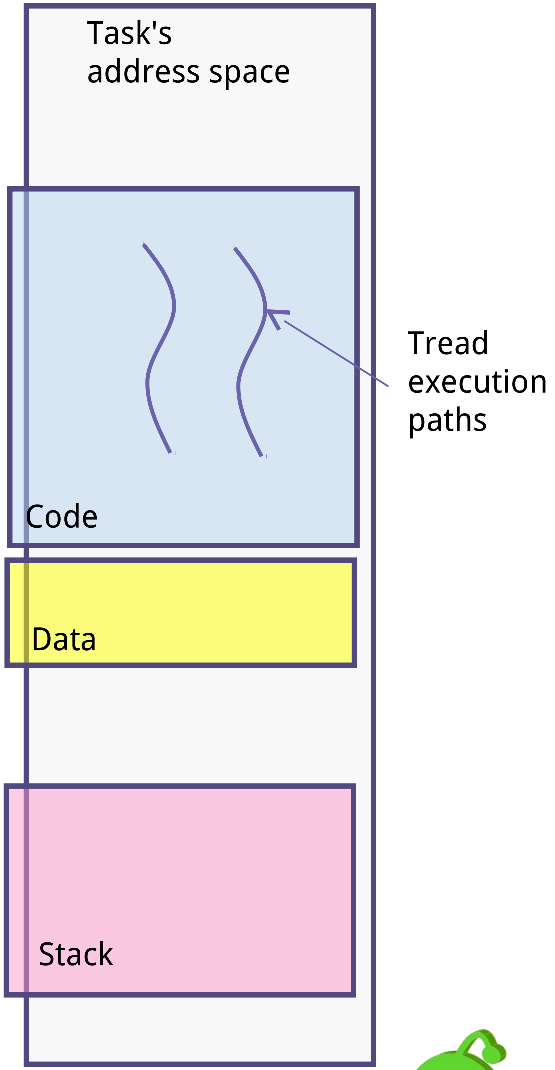
    (Migration allows threads move to another task)
  - Several threads can execute in one task

Tasks

- Represent domain of protection and isolation
- Container for code, data and resources
- Address space: capabilities + memory pages
- management operations:
  - Map: share page with other address space
  - Grant: give page to other address space
  - Unmap: revoke previously mapped page

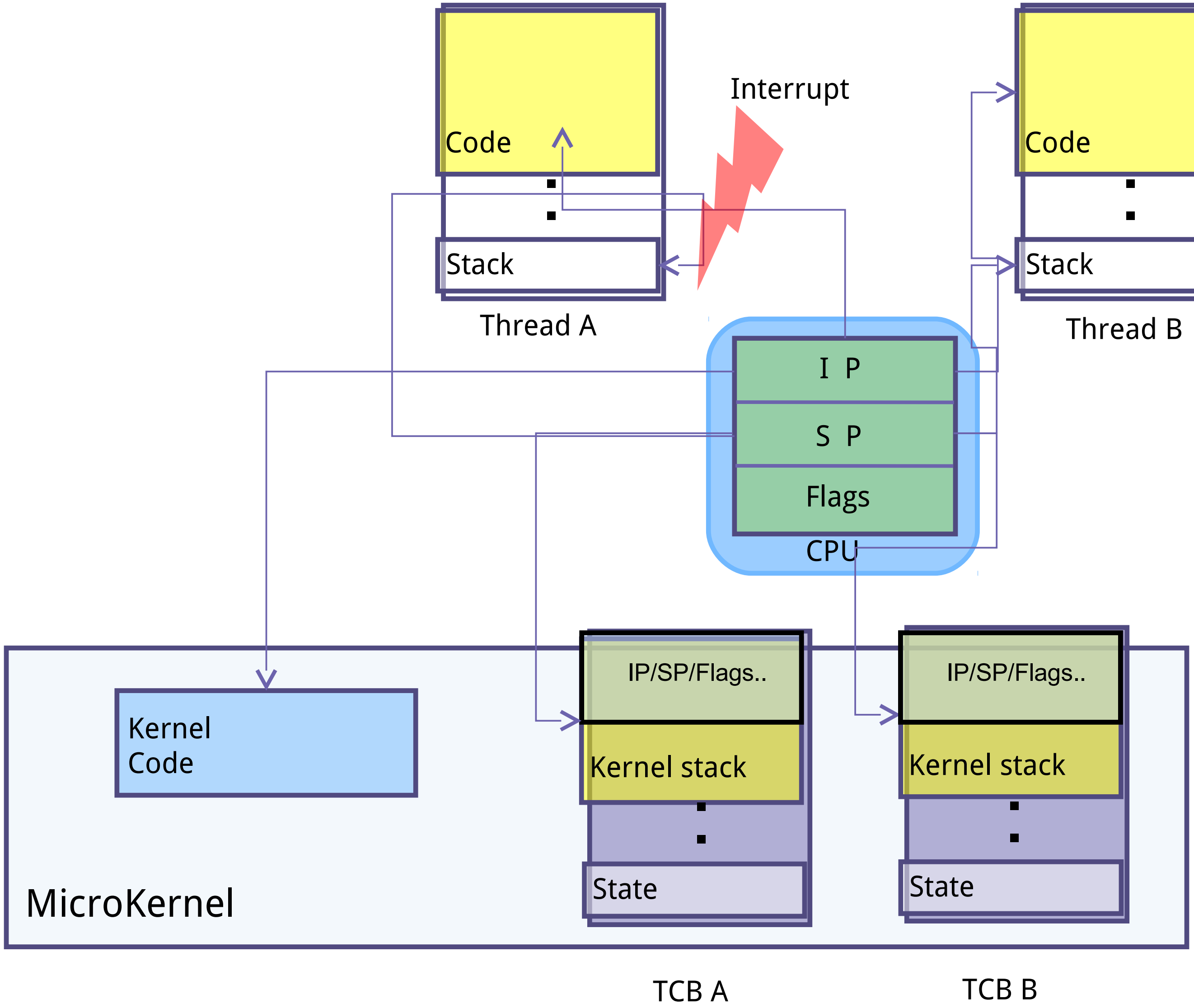
L4 uniprocessor microkernel

- Thread
  - Abstraction and unit of execution
  - Identified by thread ID
  - Consist of
    - Instruction pointer
    - Stack
    - Registers, flags…
      - Thread state
  - L4 manages (preserve) only IP, SP and registers



L4 uniprocessor micro kernel

- Thread switch

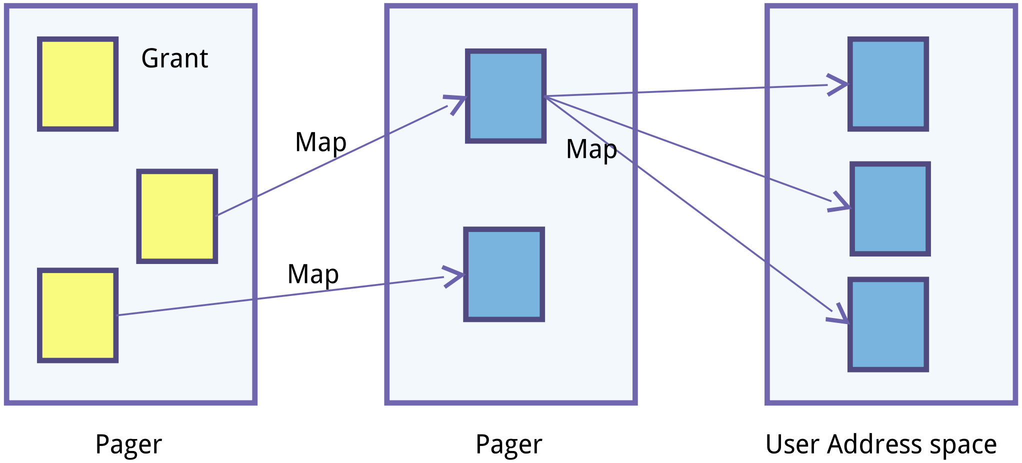


L4 uniprocessor micro kernel

- Scheduling
  - Scheduling implemented by kernel, based on priorities
  - Timeslice donation

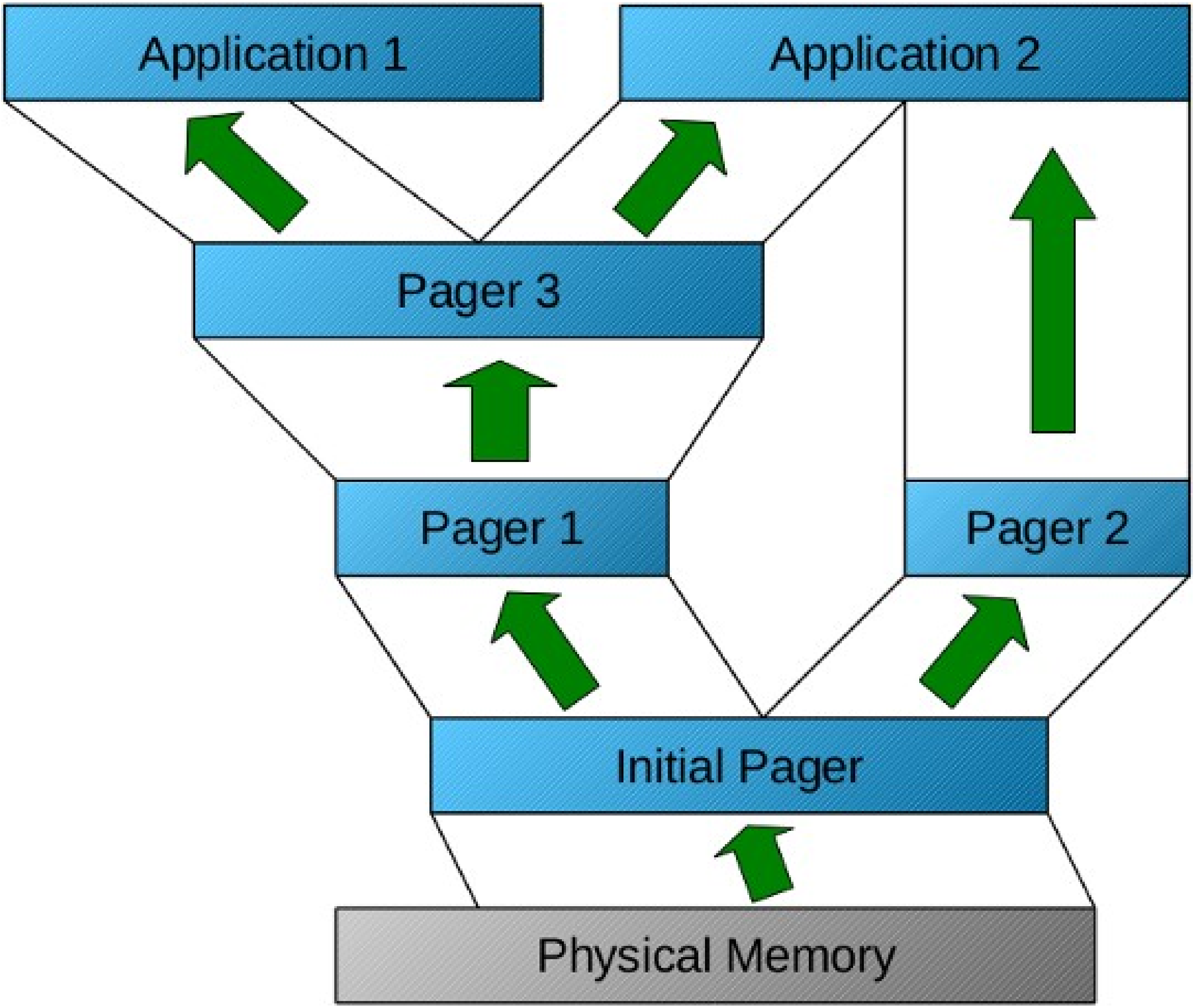
Address Space

- 3 management operations
  - Map/Unmap
    - Share/revoke page with other address space
  - Grant
    - give page to other address space
  - Flush
    - The owner of an address space can ***flush*** any of its pages.



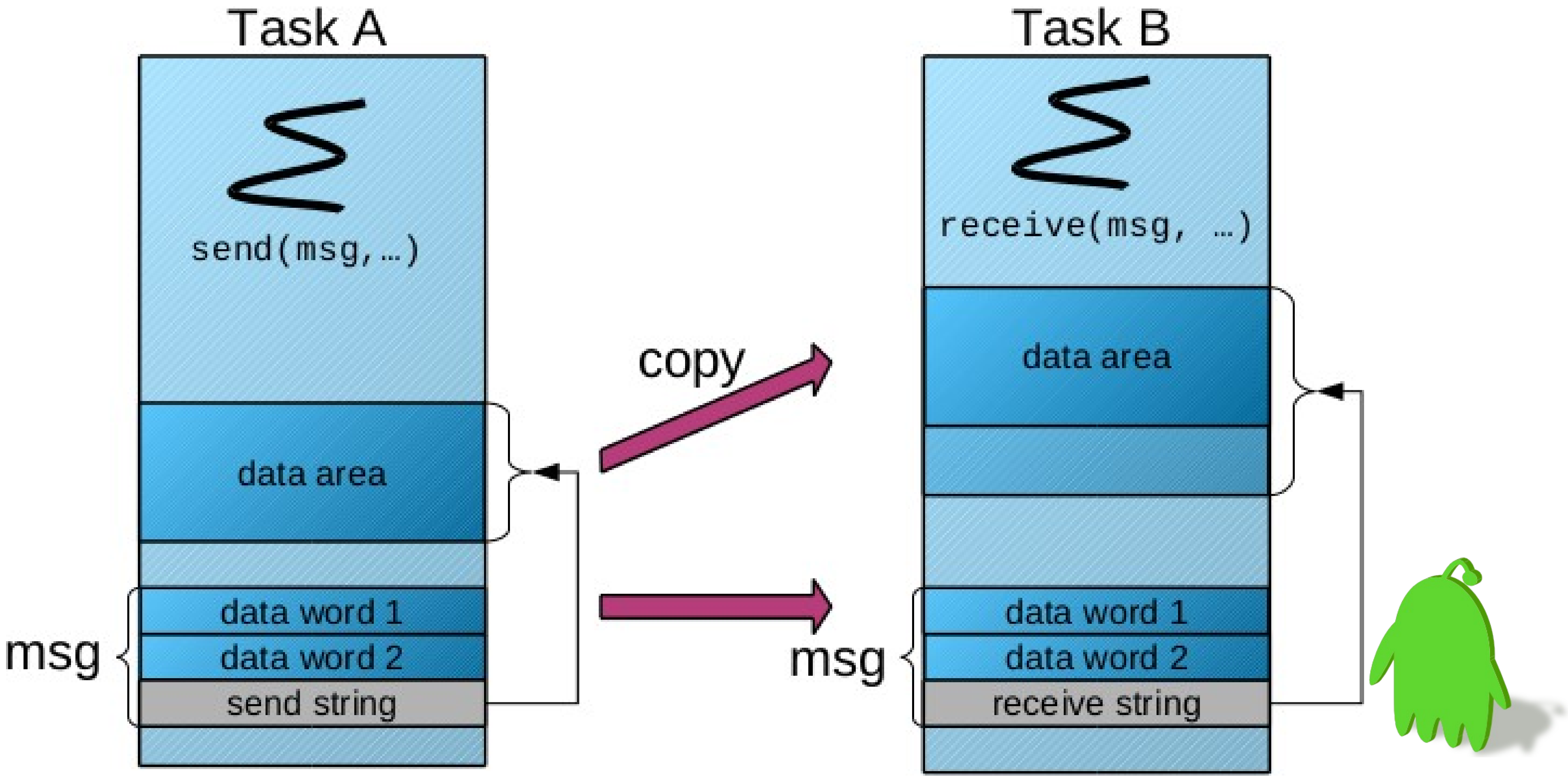
Recursive Address Space

(abandoned by seL4)



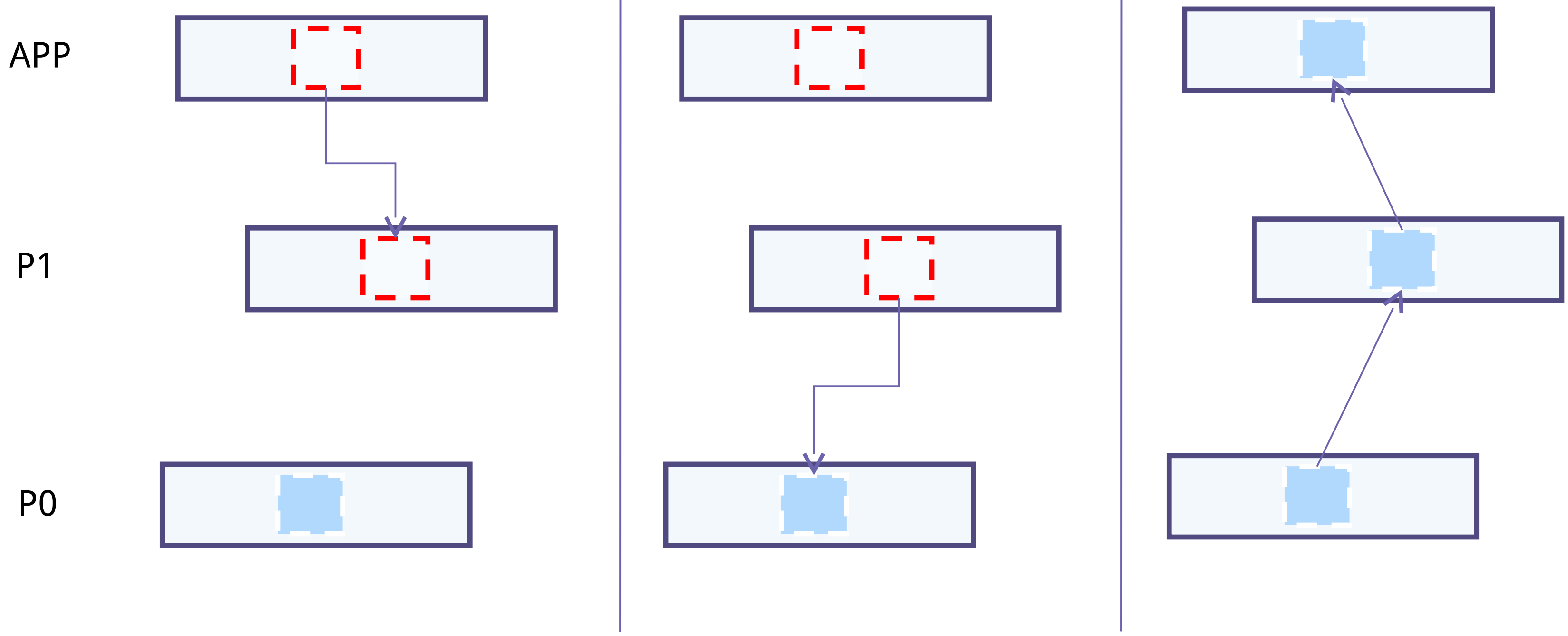
Messages: Copy Data

- Direct and indirect data copy
- UTCB message (special area)
- Special case: register-only message
- Pagefaults during user-level memory access possible



Page Fault Handling

- Page Faults are mapped to IPC
  - Pager is special thread that receives page faults
  - Page fault IPC cannot trigger another page fault
- Kernel receives the flexpage from pager and inserts mapping into page table of application
- Other faults normally terminate threads

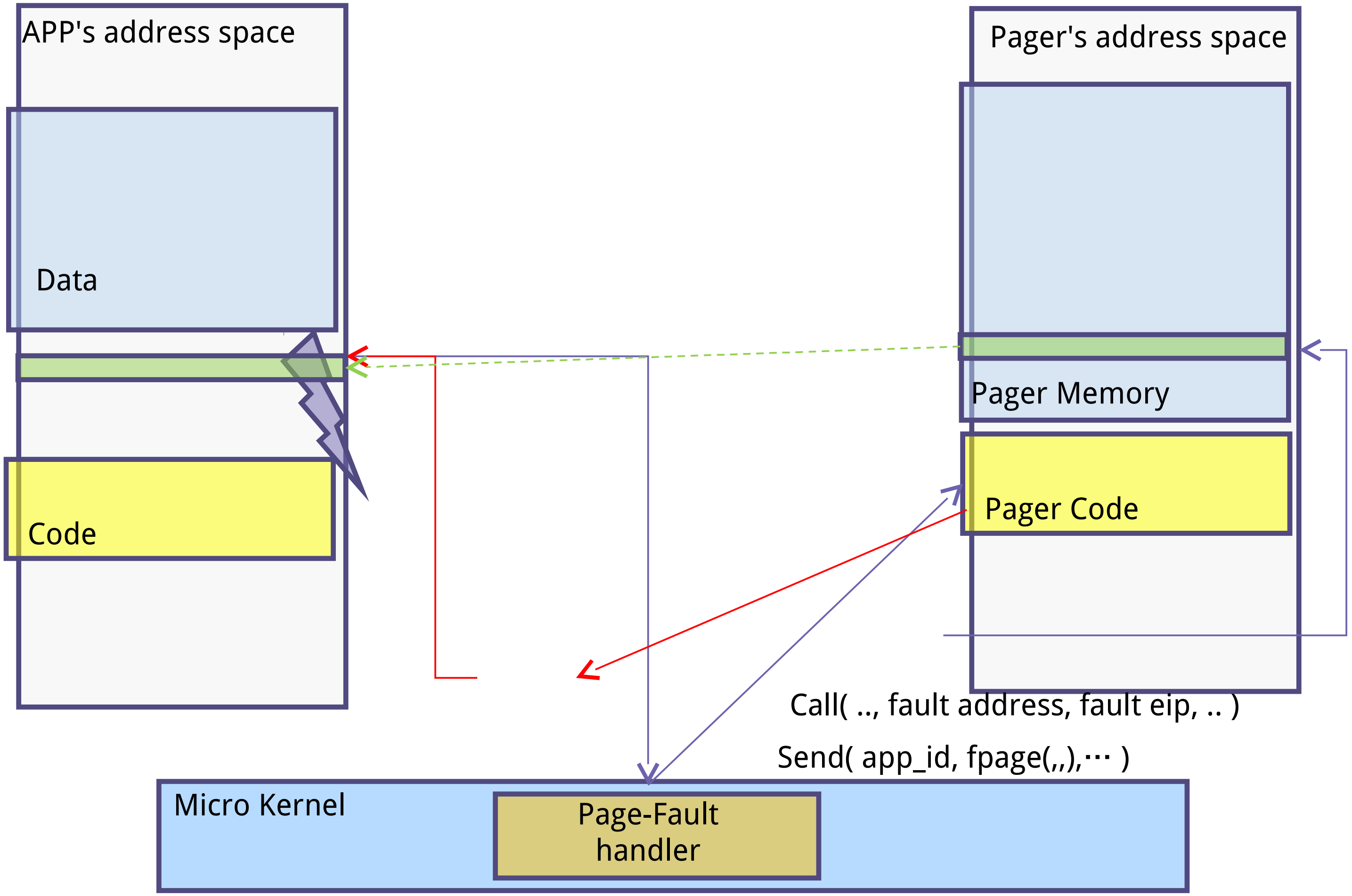


App Fault

P1 touches its own page and faults

P0 maps then P1

Page Fault Handling

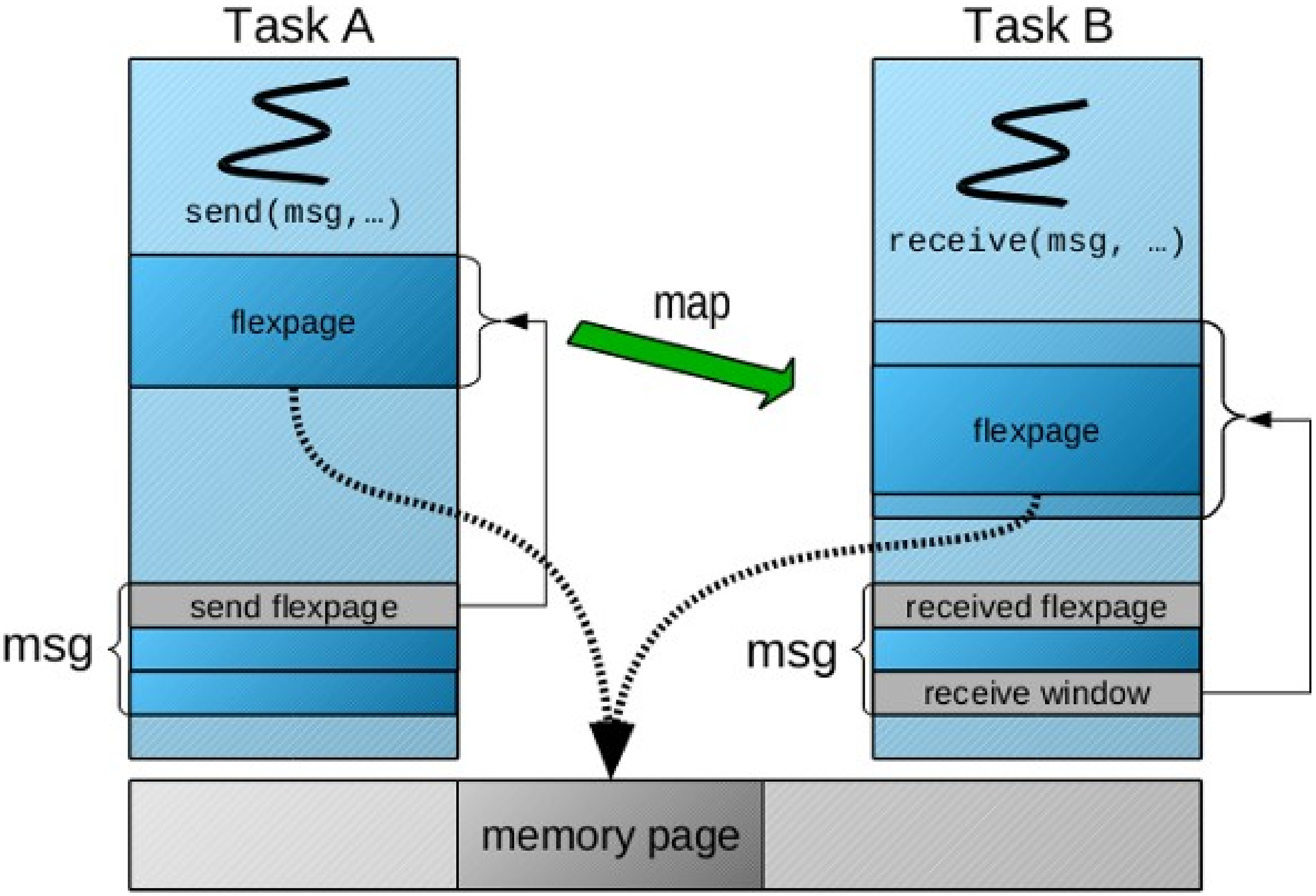


Call(.., fault address, fault eip, ..)

Send(app_id, fpage(,,),…)

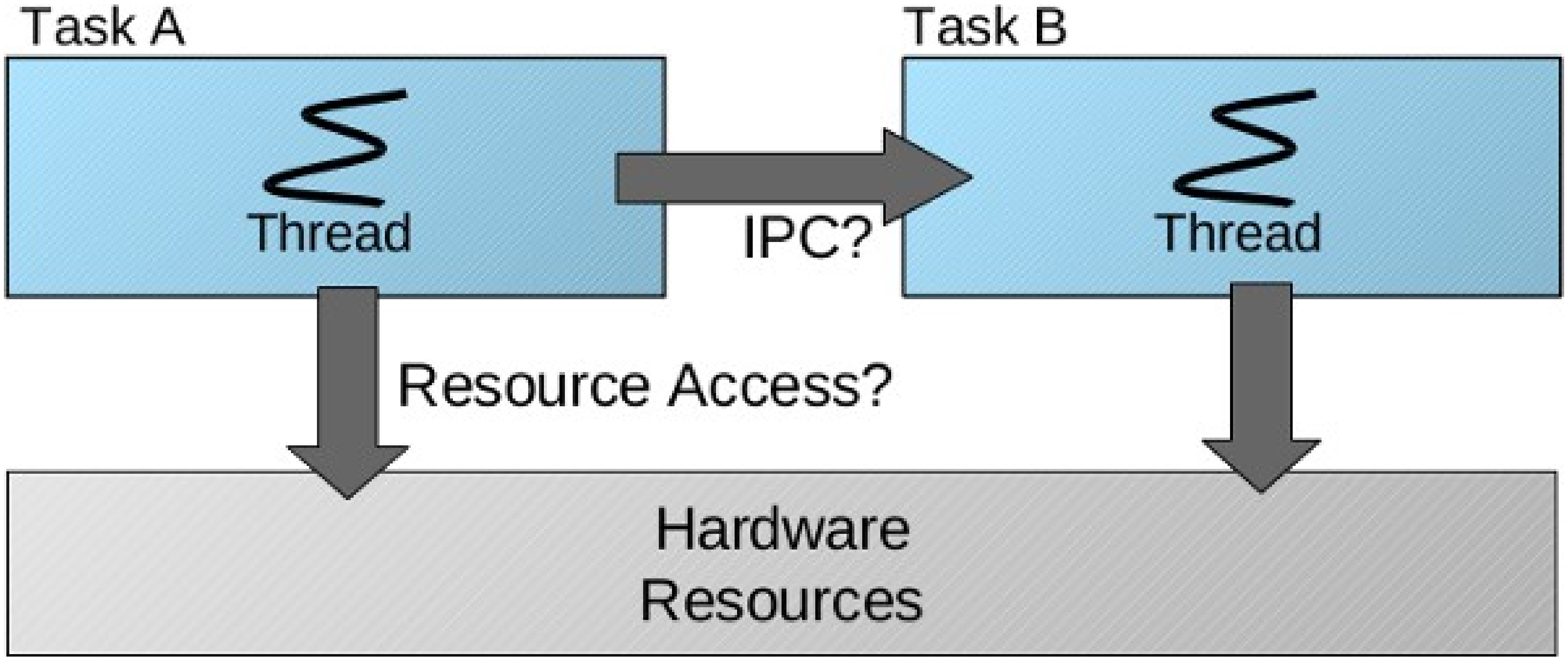
Messages: Map Reference

- Used to transfer memory pages and capabilities
- Kernel manipulates page tables
- Used to implement the map/grant operations



Communications & Resource Control

- Need to control who can send data to whom
  - Security and isolation
  - Access to resources
- Approaches
  - IPC-redirection/introspection
  - Central vs. Distributed policy and mechanism
  - ACL-based vs. capability-based



Toward 3rd generation microkernel

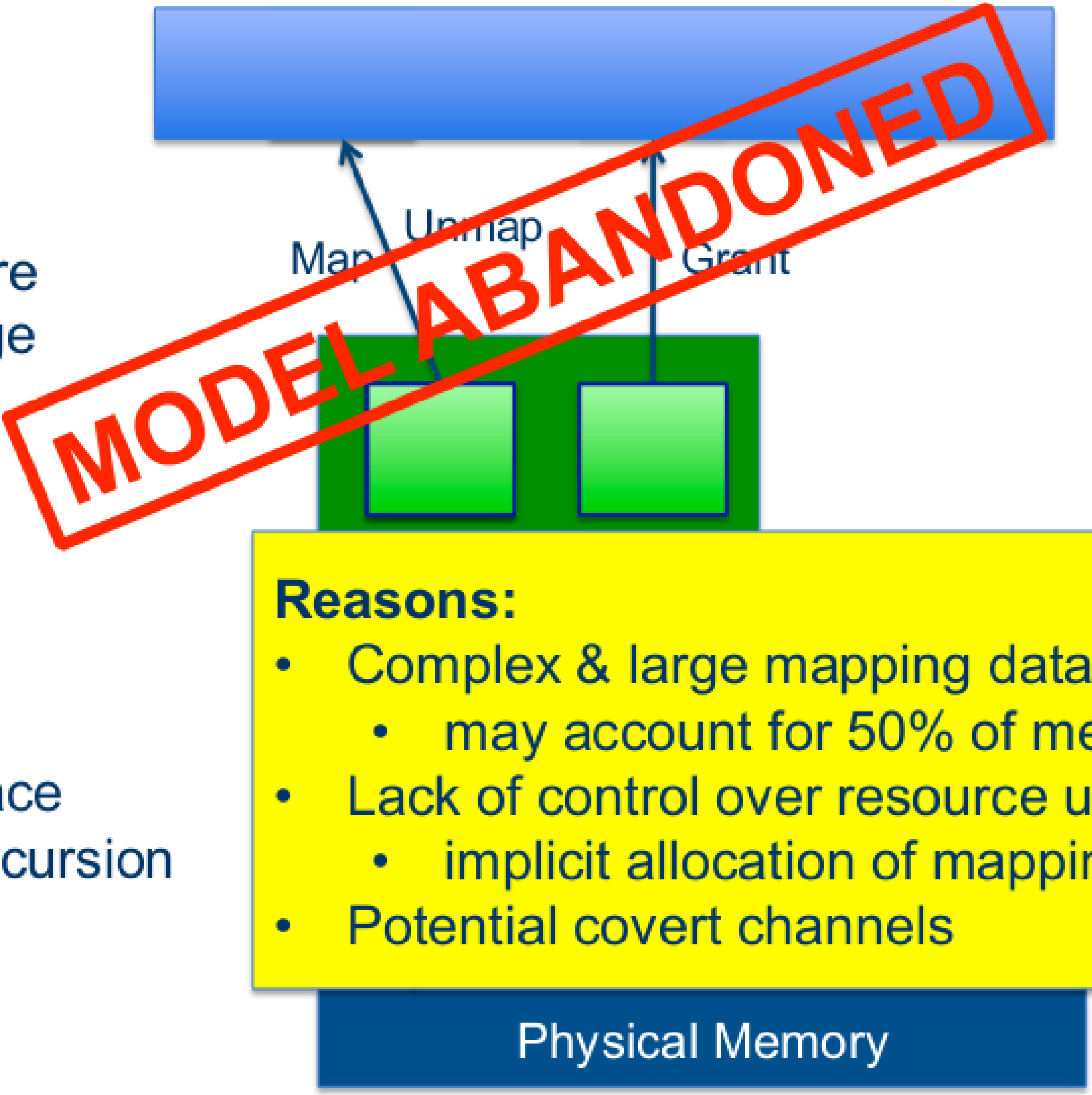
Unsolved Problems in original L4

- L4 solved performance issue [Härtig et al, SOSP'97]
  - "... but left a number of security issues unsolved"
- Problems addressed by seL4: ad-hoc approach to protection and resource management
  - Global thread name space → covert channels
  - Threads as IPC targets → insufficient encapsulation
  - Single kernel memory pool → DoS attacks
  - Insufficient delegation of authority → limited flexibility, performance

Traditional L4: Recursive Address Spaces

- Mappings are page → page

- Magic initial address space to anchor recursion



Reasons:

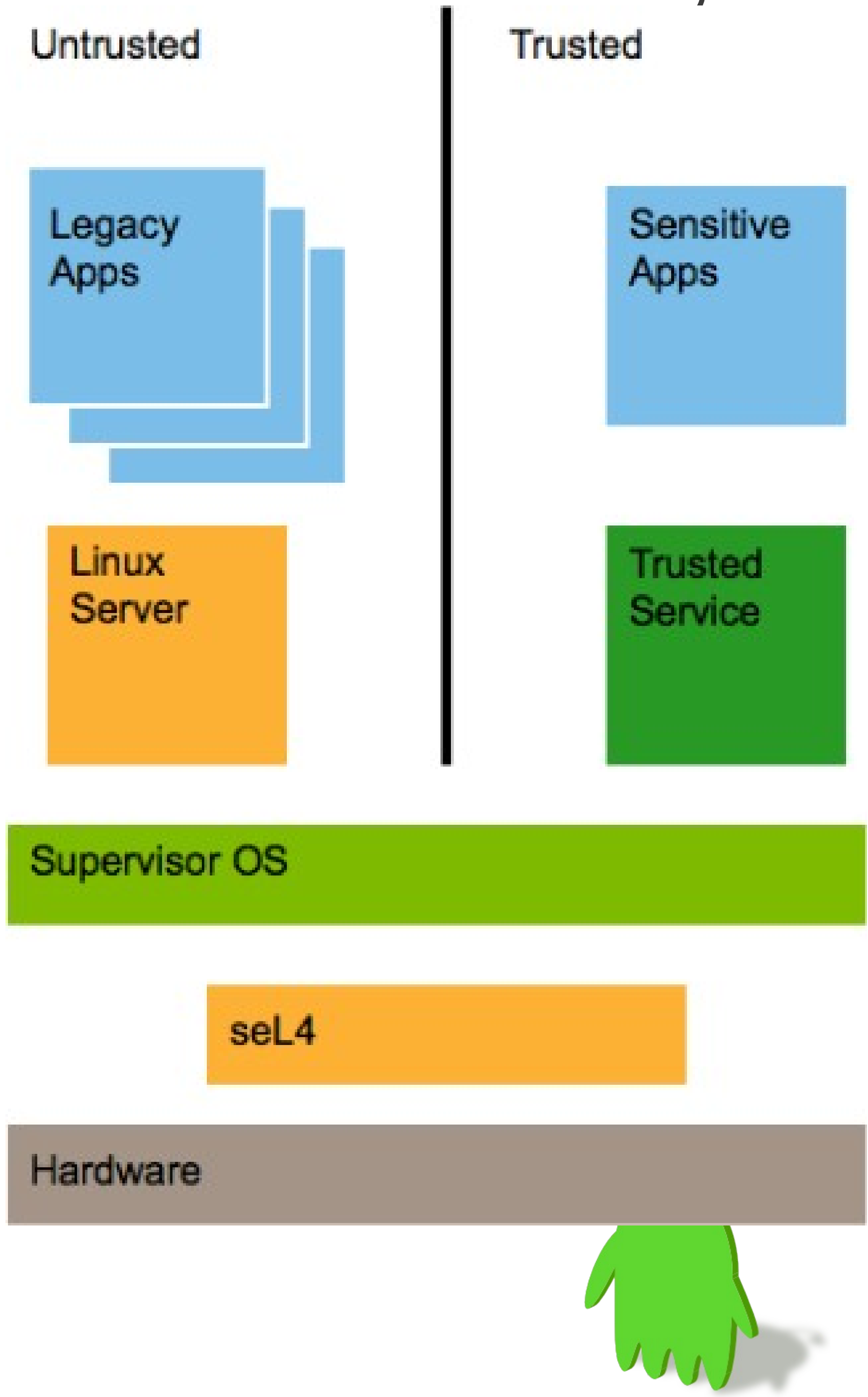
- Complex & large mapping database
  - may account for 50% of memory use!
- Lack of control over resource use
  - implicit allocation of mapping nodes
- Potential covert channels

How seL4 solves problem by designs

- Isolation: Memory management is user-level responsibility
  - Kernel never allocates memory (post-boot)
  - Kernel objects controlled by user-mode servers
- Performance: Memory management is fully delegatable
  - Supports hierarchical system design
  - Enabled by capability-based access control
- Realtime: "Incremental consistency" design pattern
  - Fast transitions between consistent states
  - Restartable operations with progress guarantee
- Verification: No concurrency in the kernel
  - Interrupts never enabled in kernel
  - Interruption points to bound latencies
  - Clustered multikernel design for multicores

seL4 in the first sight

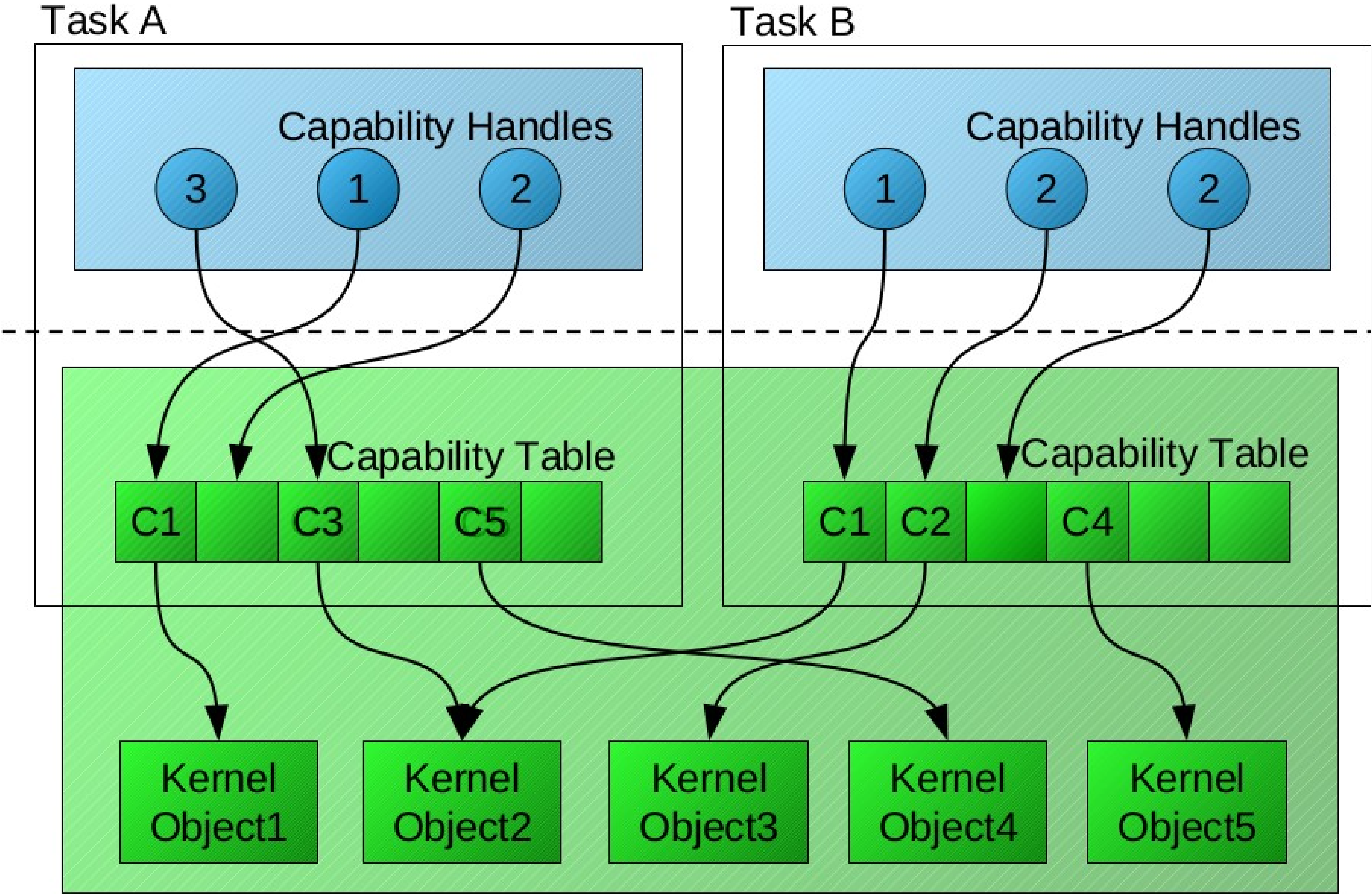
- Formal verification
  - Functional correctness
  - Security/safety properties
- No kernel heap: all memory left after boot is handed to userland
  - Resource manager can delegate to subsystems
  - Operations requiring memory explicitly provide memory to kernel
- Result: strong isolation of subsystems and high performance
  - Operate within delegated resources
  - No interference



Move to Capability based design

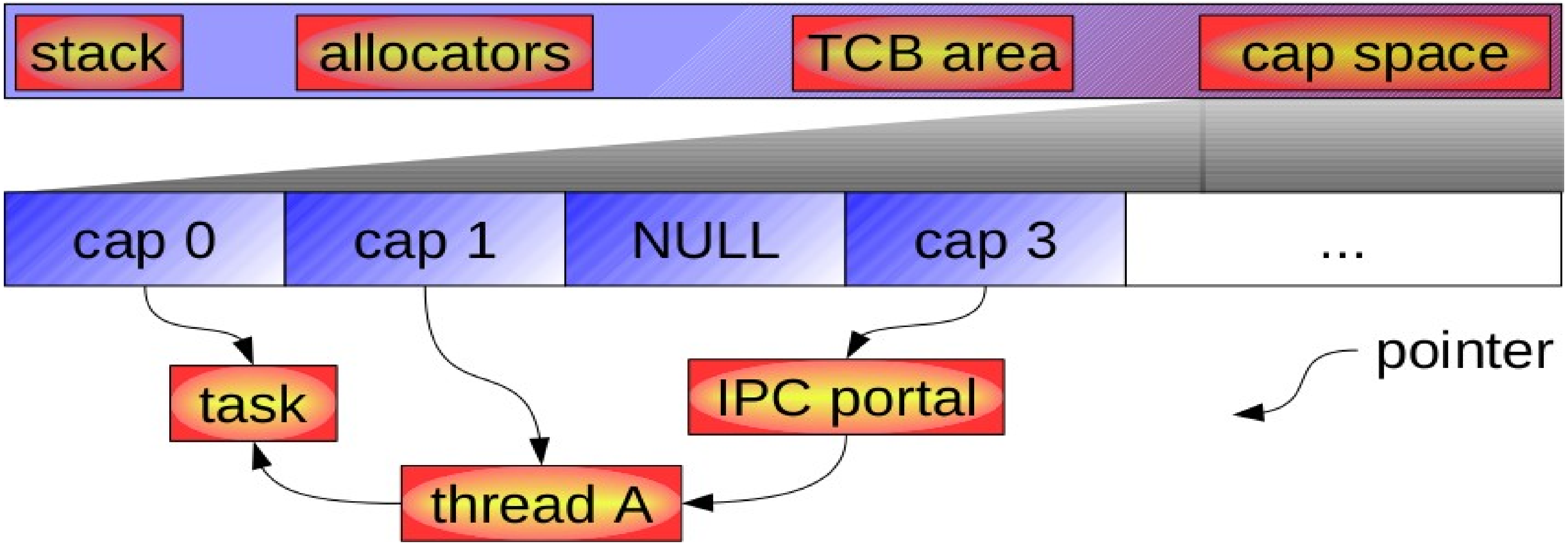
- Don't need global names (task/thread IDs)
  - Names (or IDs) are only valid within a task and have no meaning elsewhere
- Kernel objects are referenced through local IDs, comparable to POSIX file descriptors or handles
- Creating a new (kernel) object returns an index into a task-local table, where in turn the pointer to the object is stored
- Kernel protects this capability table, therefore unforgeable

Capabilities



Capability space

- In-kernel memory table with pointers to kernel objects
- Sending a message to thread A merely requires the sender to have a capability to the portal cap
- Sender does not know which thread/task will receive it
- Receiver does not know who sent it (in general)
- Separation of subsystems, combinable, independent

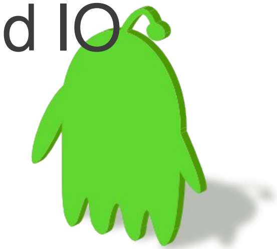


Capabilities

- Kernel objects represent resources and communication channels
- Capability
  - Reference to kernel object
  - Associated with access rights
  - Can be mapped from task to another task
- Capability table is task-local data structure inside the kernel
  - Similar to page table
  - Valid entries contain capabilities
- Capability handle is index number to reference entry into capability table
  - Similar to file handle of POSIX
- Mapping capabilities establishes a new valid entry into the capability table

Importance of Capabilities

- Everything is a file → Everything is a capability
- Object capabilities
  - Tasks, threads, IPC portals, factories, semaphores
  - Handles/pointers to kernel objects, can be created, delegated and destroyed
- Memory capabilities
  - Resembles virtual memory pages
  - Sending (mapping) a memory capability established shared memory between sender and receiver
- IO capabilities
  - Abstraction for access to IO ports, delegating IO caps allows the receiving Task/Address space to access denoted IO ports

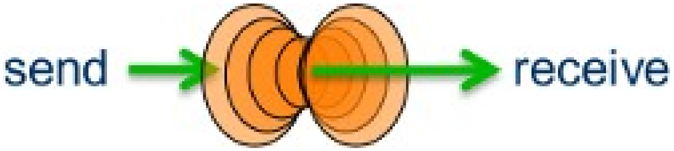


seL4 concpts

- Capabilities (Caps)
  - mediate access
- Kernel objects:
  - Threads (thread-control blocks, TCBs)
  - Address spaces (page table objects, PDs, Pts)
  - IPC endpoints (EPs, AsyncEPs)
  - Capability spaces (Cnodes)
  - Frames
  - Interrupt objects
  - Untyped memory
- System calls
  - Send, Wait (and variants)
  - Yield

Revised IPC

- OS services provided by (protected) user-level server processes
  - invoked by IPC
- seL4 IPC uses a handshake through endpoints:
  - Transfer points without storage capacity
  - Message must be transferred instantly
    - One partner may have to block
    - Single copy user → user by kernel
- Two endpoint types:
  - Synchronous (Endpoint)
  - asynchronous (AsyncEP)



L4 Revisions

L4 History: V2 API

- Original version by Jochen Liedtke (GMD) » 93–95
  - “Version 2” API
  - i486 assembler
  - IPC 20 times faster than Mach [SOSP 93, 95]
  - Proprietary code base (GMD)
- Other L4 V2 implementations:
  - L4/MIPS64: assembler + C (UNSW) 95–97
    - Fastest kernel on single-issue CPU (100 cycles on MIPS R4600)
    - Open source (GPL)
  - L4/Alpha: PAL + C (Dresden/UNSW), 95–97
    - First released SMP version (UNSW)
    - Open source (GPL)
  - Fiasco (Pentium): C++ (Dresden), 97–99, ongoing development
    - Open source (GPL)

L4 History: X.1 API

- Experimental “Version X” API
  - Improved hardware abstraction
  - Various experimental features (performance, security, generality)
  - Portability experiments
- Implementations
  - Pentium: assembler, Liedtke (IBM), 97–98
    - Proprietary
  - *Hazelnut (Pentium+ARM), C, Liedtke et al (Karlsruhe), 98–99*
    - Open source (GPL)

L4 History: X.2/V4 API

- “Version 4” (X.2) API, 02
  - Portability, API improvements
- L4Ka::Pistachio, C++ (plus assembler “fast path”)
  - x86, PPC-32, Itanium (Karlsruhe), 02–03
    - Fastest ever kernel (36 cycles on Itanium, NICTA/UNSW)
  - MIPS64, Alpha (NICTA/UNSW), 03
    - Same performance as V2 kernel (100 cycles single issue)
  - ARM, PPC-64 (NICTA/UNSW), x86-64 (Karlsruhe), 03–04
  - Open source (BSD license)

Real-world Deployment: Virtualization drives performance improvements

L4Linux

where virtualization comes from

- Linux source has two cleanly separated parts
  - Architecture dependent
  - Architecture independent
- In L4Linux
  - Architecture dependent code is modified for L4
  - Architecture independent part is unchanged
  - L4 not specifically modified to support Linux

L4Linux

where virtualization comes from

- Linux kernel as L4 user service
  - Runs as an L4 thread in a single L4 address space
  - Creates L4 threads for its user processes
  - Maps parts of its address space to user process threads (using L4 primitives)
  - Acts as pager thread for its user threads
  - Has its own logical page table
  - Multiplexes its own single thread (to avoid having to change Linux source code)

L4Linux

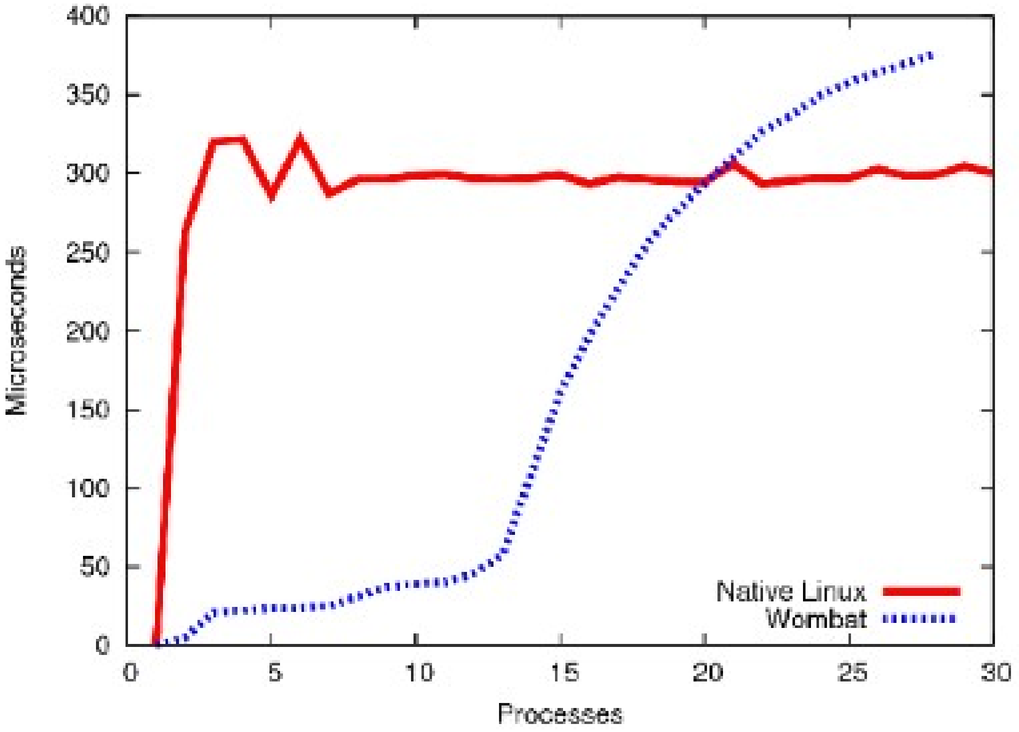
where virtualization comes from

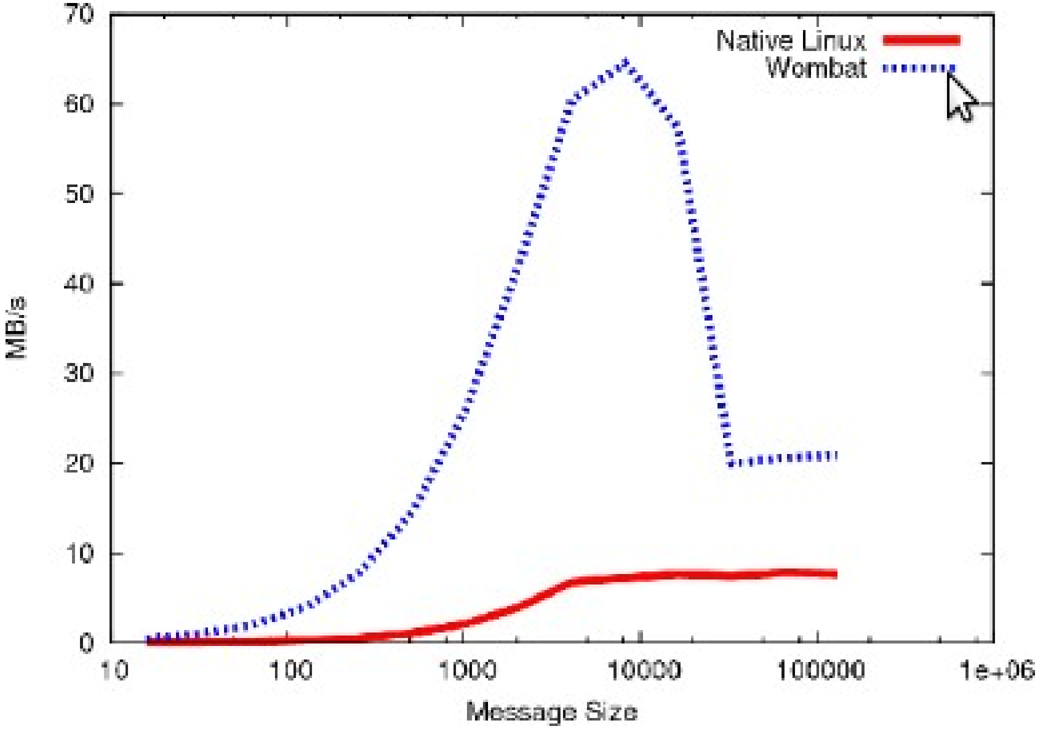
- The statically linked and shared C libraries are modified
  - Systems calls in the lib call the Linux kernel using IPC
- For unmodified native Linux applications, there is a “trampoline”
  - The application traps
  - Control bounces to a user-level exception handler
  - The handler calls the modified shared library
  - Binary compatible

Performance is not acceptable!

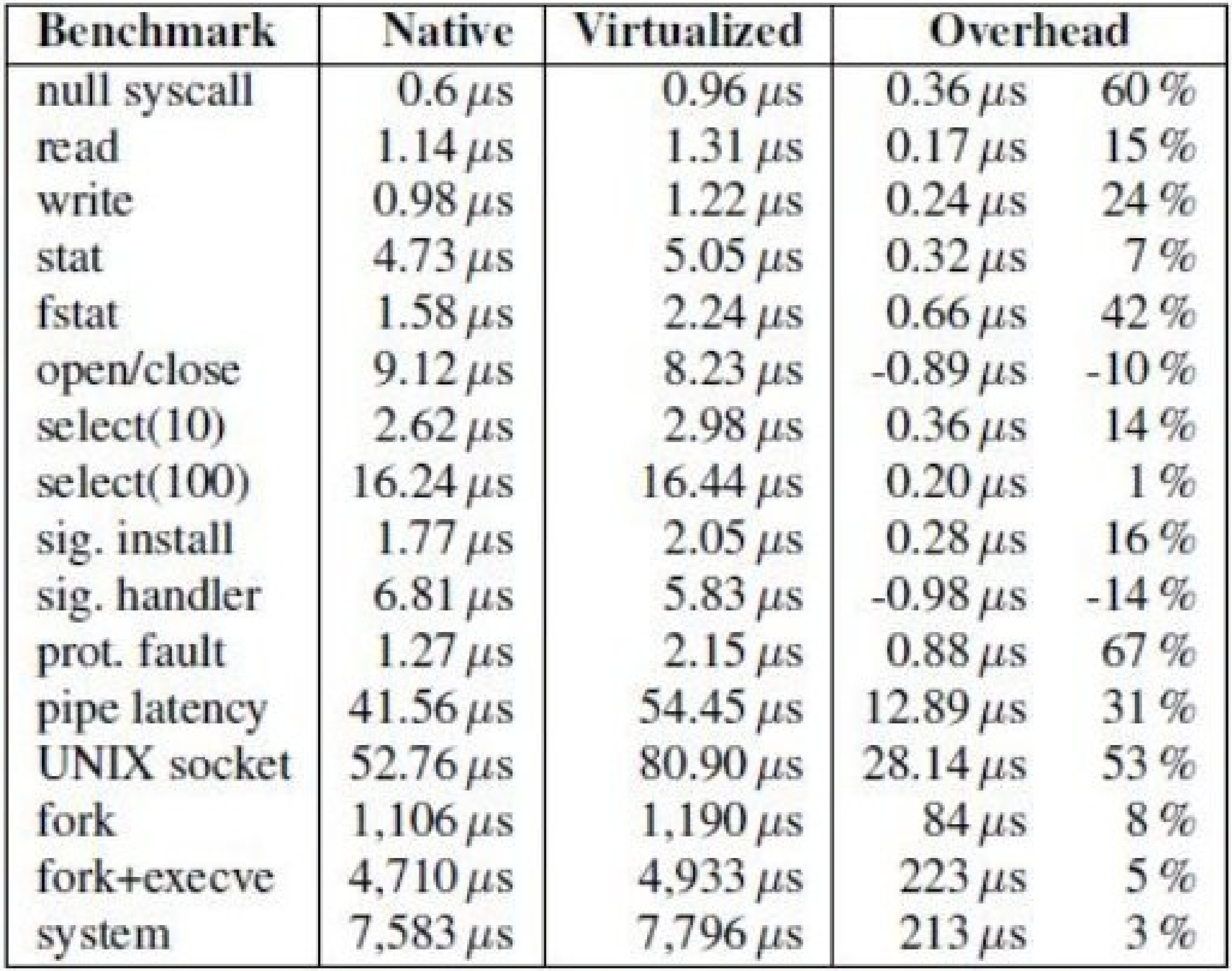
- L4Linux [Härtig et al., SOSP'97]
  - 5–10% overhead on macro-BMs
  - 6–7% overhead on kernel compile
- MkLinux (Linux on Mach):
  - 27% overhead on kernel compile
  - 17% overhead with Linux in kernel

NICTA L4 / OKL4



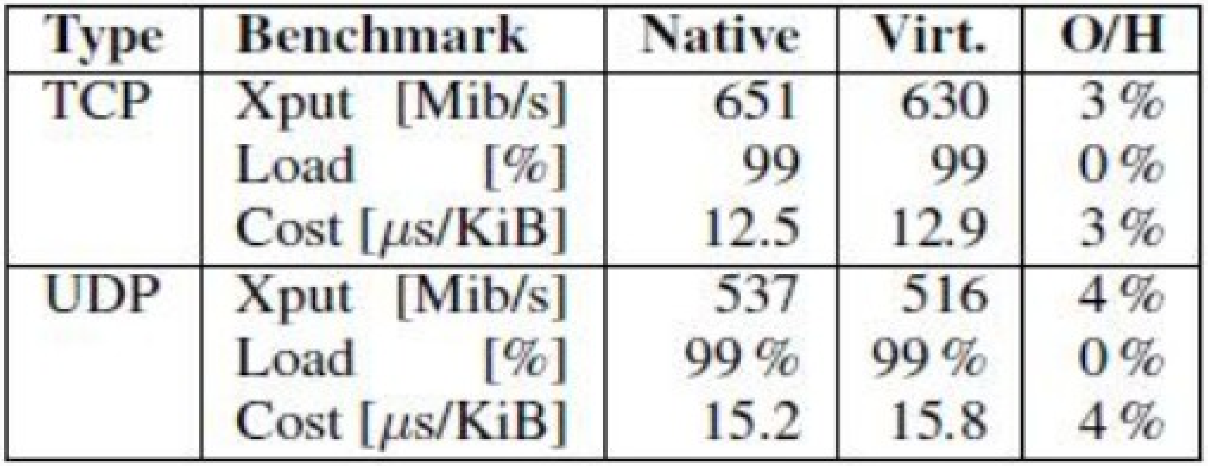


- L4 implementations on embedded processors
  - ARM, MIPS
- Wombat: portable virtualized Linux for embedded systems
- ARMv4/v5 thanks to fast context-switching tricks



Benchmark	Native	Virtualized	Overhead	
null syscall	0.6 µs	0.96 µs	0.36 µs	60 %
read	1.14 µs	1.31 µs	0.17 µs	15 %
write	0.98 µs	1.22 µs	0.24 µs	24 %
stat	4.73 µs	5.05 µs	0.32 µs	7 %
fstat	1.58 µs	2.24 µs	0.66 µs	42 %
open/close	9.12 µs	8.23 µs	-0.89 µs	-10 %
select(10)	2.62 µs	2.98 µs	0.36 µs	14 %
select(100)	16.24 µs	16.44 µs	0.20 µs	1 %
sig. install	1.77 µs	2.05 µs	0.28 µs	16 %
sig. handler	6.81 µs	5.83 µs	-0.98 µs	-14 %
prot. fault	1.27 µs	2.15 µs	0.88 µs	67 %
pipe latency	41.56 µs	54.45 µs	12.89 µs	31 %
UNIX socket	52.76 µs	80.90 µs	28.14 µs	53 %
fork	1,106 µs	1,190 µs	84 µs	8 %
fork+execve	4,710 µs	4,933 µs	223 µs	5 %
system	7,583 µs	7,796 µs	213 µs	3 %

LmBench shows near native performance with OKL4 3.0 on ARMv7 target

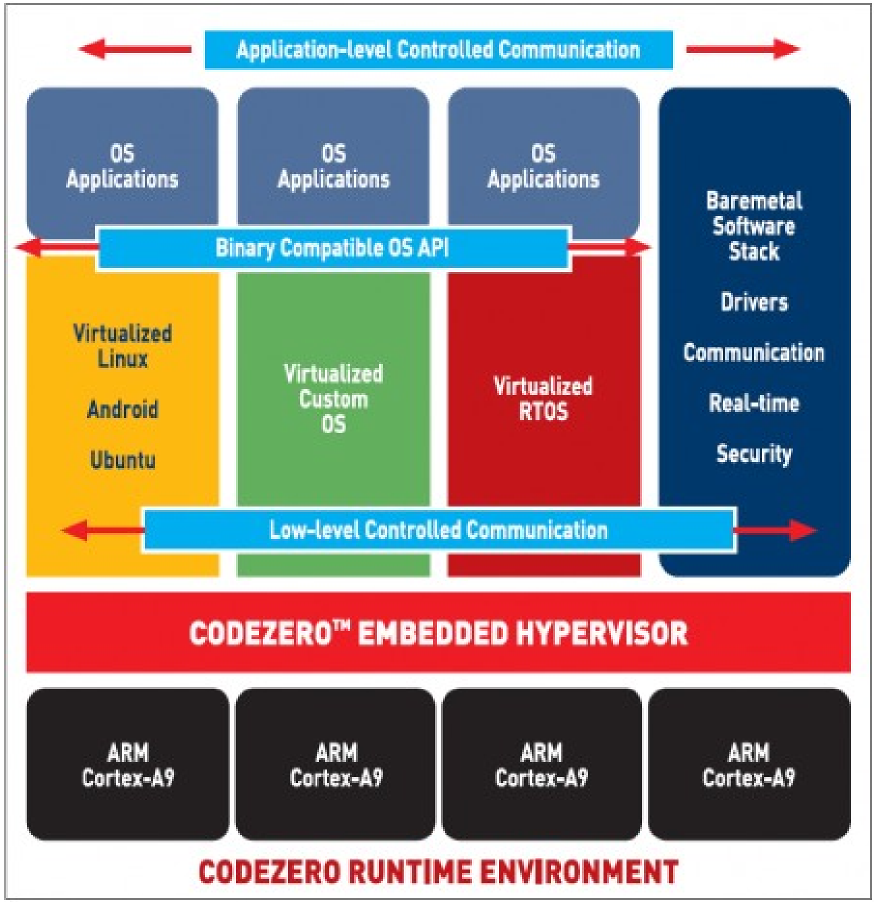


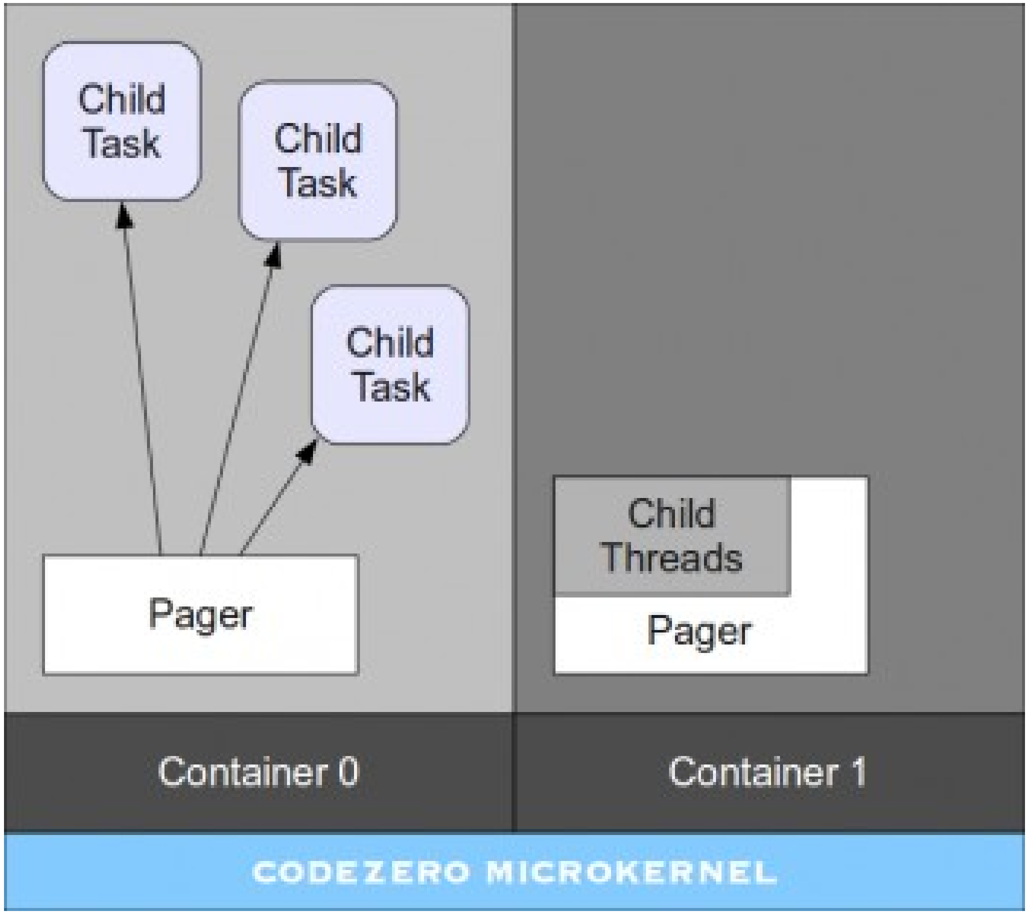
Type	Benchmark	Native	Virt.	O/H
TCP	Xput [Mib/s]	651	630	3 %
	Load [%]	99	99	0 %
	Cost [µs/KiB]	12.5	12.9	3 %
UDP	Xput [Mib/s]	537	516	4 %
	Load [%]	99 %	99 %	0 %
	Cost [µs/KiB]	15.2	15.8	4 %

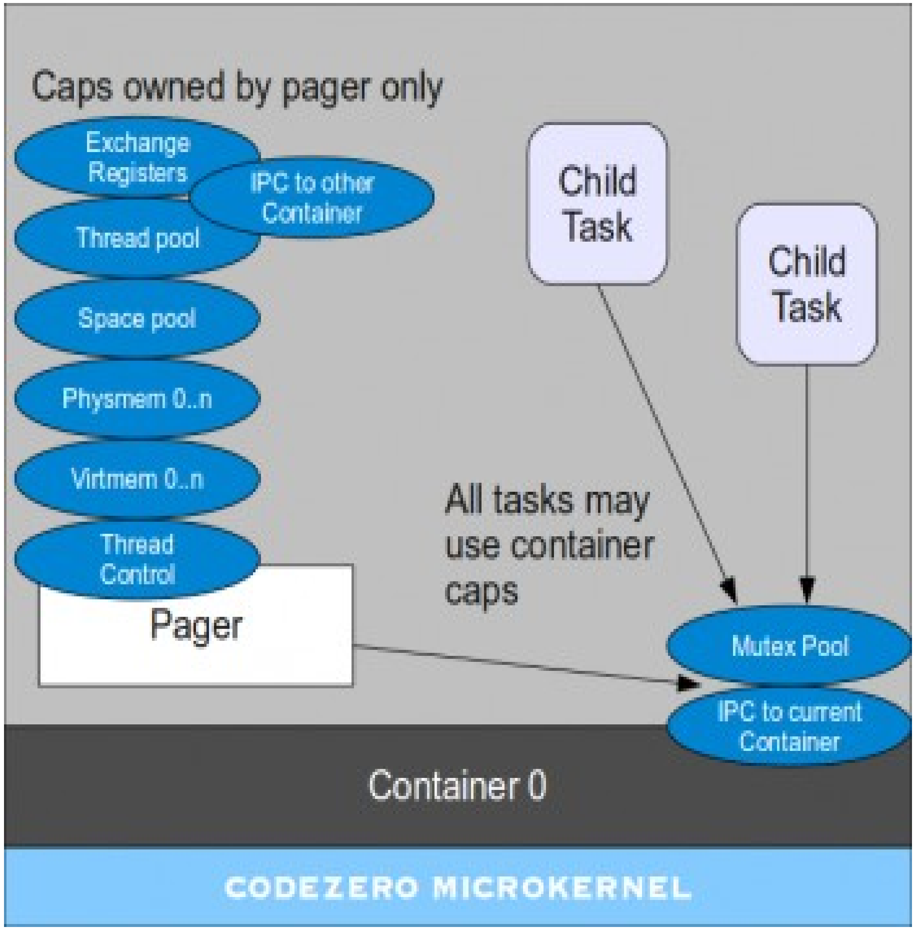
NetPerf
fully-loaded CPU and the throughput degradation of the virtualized is only 3% and 4%.

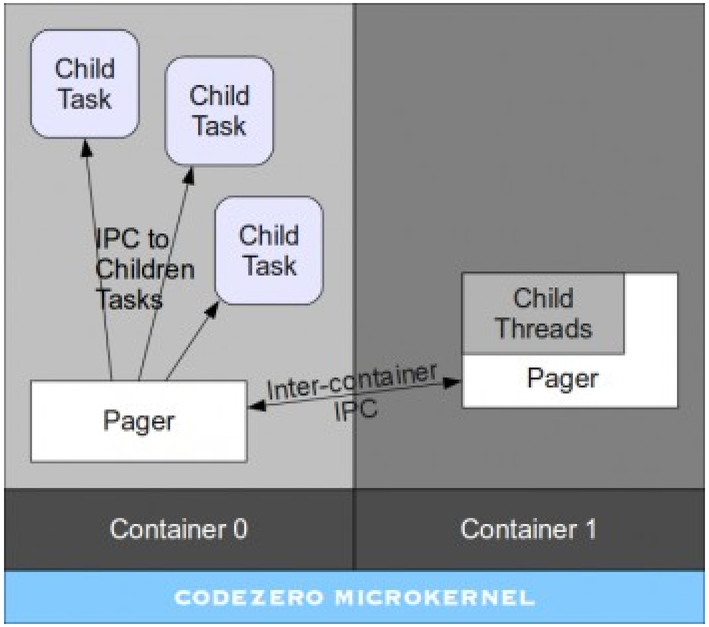
Codezero hypervisor

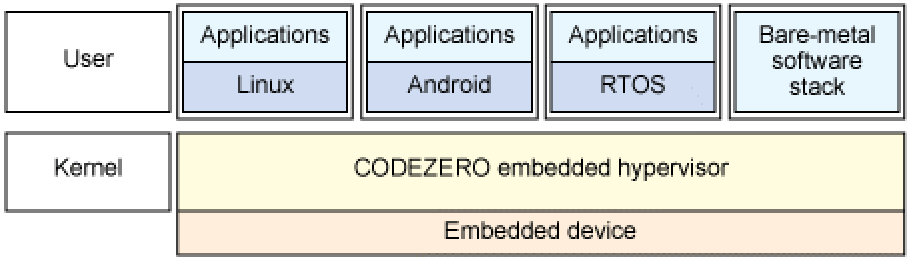
- Optimized for latest ARM cores (Cortex-A9/A15)
- L4 microkernel based design, written from scratch
- Capability based dynamic resource management
- Container oriented driver model: no modifications required for Linux











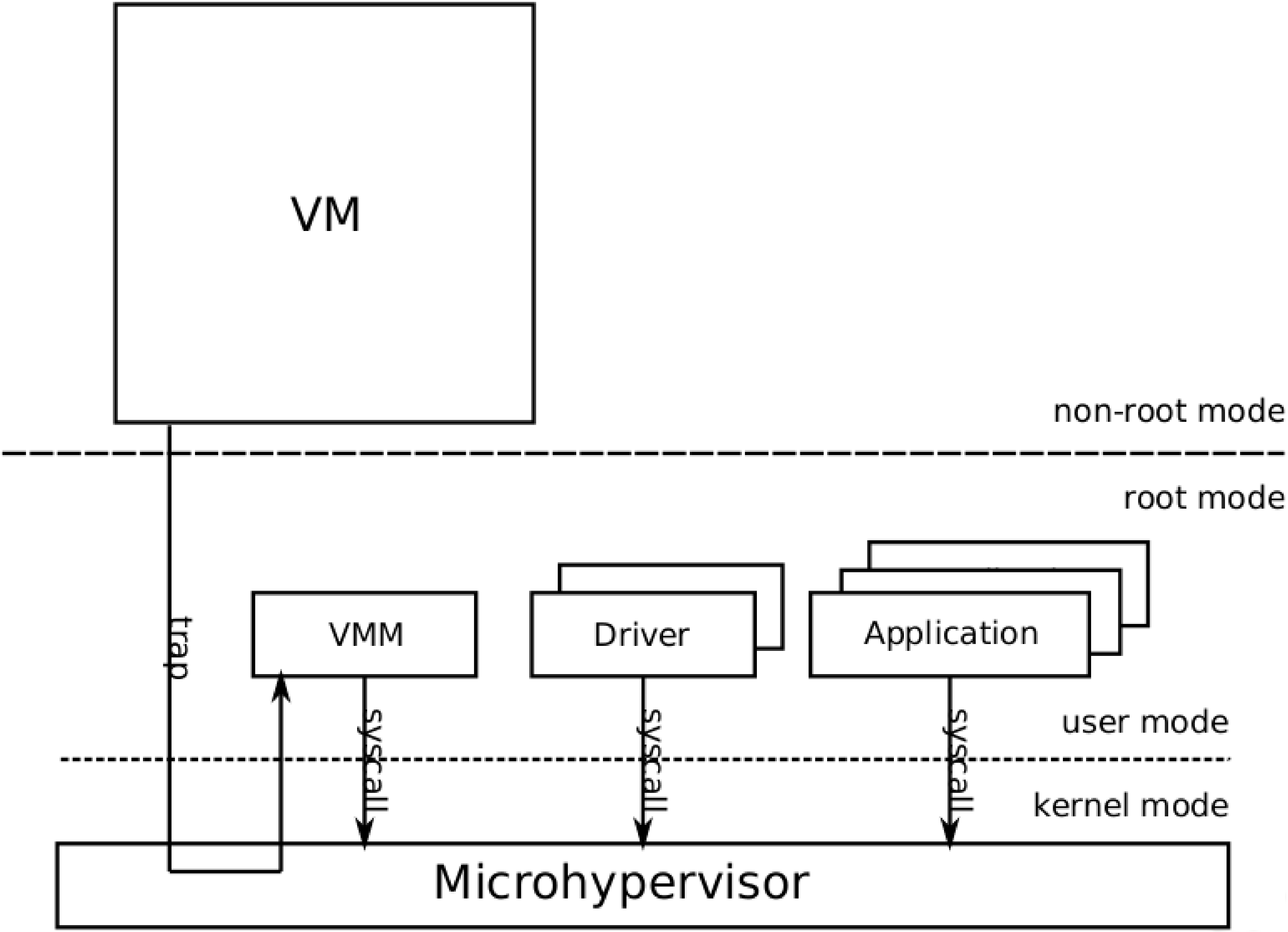
User	Applications Linux	Applications Android	Applications RTOS	Bare-metal software stack
Kernel	CODEZERO embedded hypervisor			
	Embedded device			

Micro-hypervisor

- Microvisor – OKL4 4.0
- Research projects such as NOVA, Coyotos, and seL4
- Aided by virtualizable ISA

- Microhypervisor
  - the “kernel” part
  - provides isolation
  - mechanisms, no policies
  - enables safe access to virtualization features to userspace

- VMM
  - the “userland” part
  - CPU emulation
  - device emulation


VM
non-root mode
root mode
trap
VMM
Driver
Application
syscall
syscall
syscall
user mode
kernel mode
Microhypervisor

Advantage of NOA architecture: Reduce TCB of each VM

- Micro-hypervisor provides low-level protection domains
  - address spaces
  - virtual machines
- VM exits are relayed to VMM as IPC with selective guest state
- one VMM per guest in (root mode) userspace:
  - possibly specialized VMMs to reduce attack surface
  - only one generic VMM implemented

Adaptation/Optimizations

Learned from NICTA L4

- Process-orientation wastes RAM
  - Replaced by single-stack (event-driven) approach
- Virtual TCB array wastes VAS, TLB entries
  - without performance benefits on modern hardware
- Capabilities are better than thread UIDs
  - Provide uniform resource control model & avoid covert channels
- Also: IPC timeouts are useless
  - Replaced by block/poll bit
- Virtualization is essential
  - Re-think kernel abstractions

Generic parts in L4

- Memory management
- Page-fault handling
- IPC Path
- Mapping database
- Base of the kernel debugger
- Most code of L4 abstractions
    - Thread and address-space management

Processor-specific parts in L4

- Basic data types
- Processor abstraction
  - IRQ control, sleep-mode support
- Atomic operations
- Page tables
- Parts of L4 abstractions
  - Switch of CPU and FPU state
- CPU specific optimizations

Hotspot in performance view

- Processor modes
  - mapping to kernel mode and user mode, mode switches
- Processor state
  - context switches
- MMU/TLB
  - specific address-space/page-table code
- Caches
  - specific cache-consistency handling
  - Cache consistency must be maintained (critical for task switches)
- IRQ controller
  - abstract controller interface

Generic optimizations

- Optimized data structures and code
  - Minimize memory accesses
  - Minimize cache and TLB footprint
  - Minimize number of instructions for frequently used operations
- Optimizations often depend on knowledge of HW
  - Cache size / associativity
  - TLB size / features (e.g., supported page sizes)
  - Available instructions in the ISA

0xlab

http://0xlab.org