Microkernel Evolution:
Designs and Aspects

Jim Huang (黃敬群) <**jserv**@0xlab.org>

Mar 27, 2013 / JuluOSDev

Rights to copy

creative commons
COMMONS DEED

Attribution – ShareAlike 3.0

Corrections, suggestions, contributions and translations are welcome!

Latest update: Apr 17, 2013

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

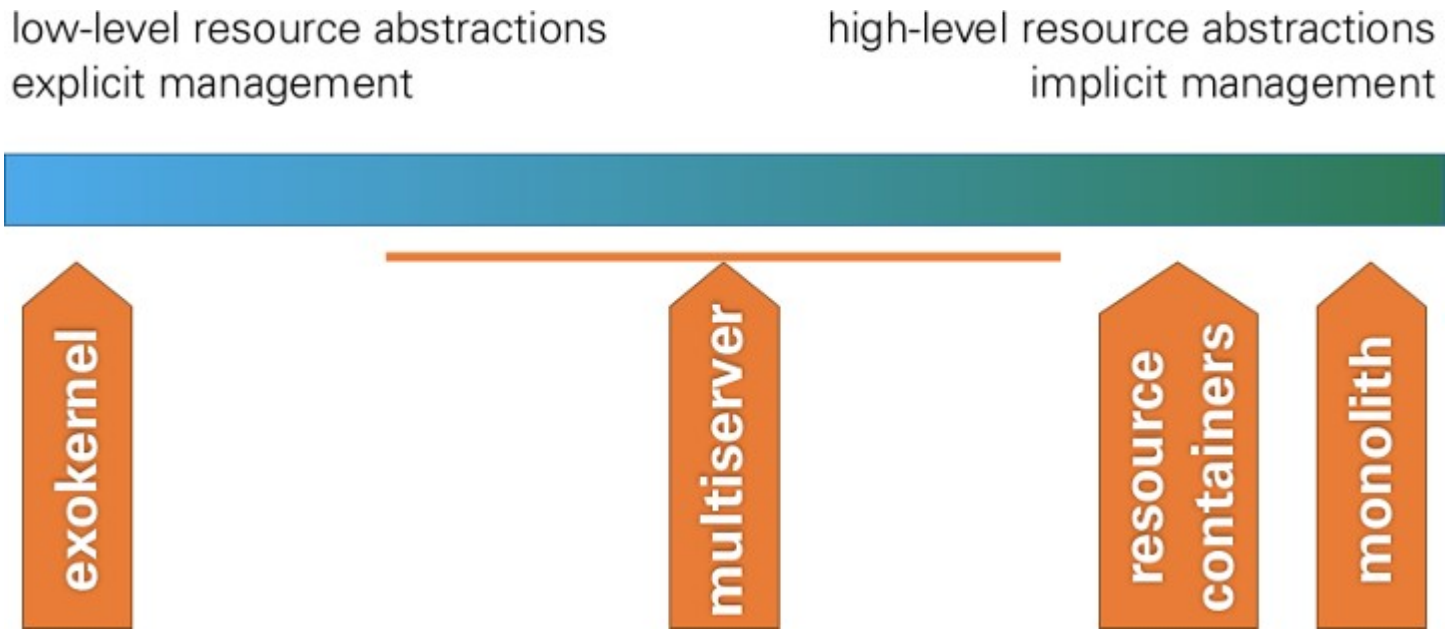
Agenda

- Kernel style!
- 3 Generations of Microkernel
  - Mach
  - L4
  - seL4, Fiasco.OC, NOVA
- Adopting microkernels: Mobile Virtualization

Kernel style!

Types of Kernel Designs

- Monolithic kernel
- Microkernel
- Hybrid Kernel
- Exokernel
- Virtual Machine / Hypervisor

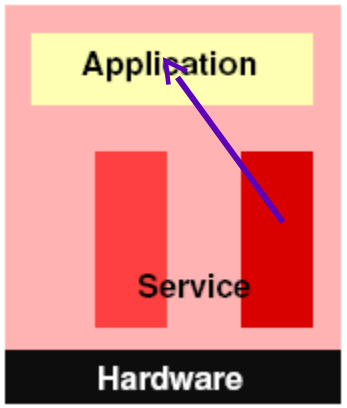


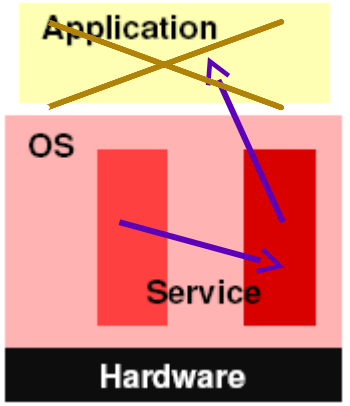
Source: Michael Roitzsch, TU Dresden

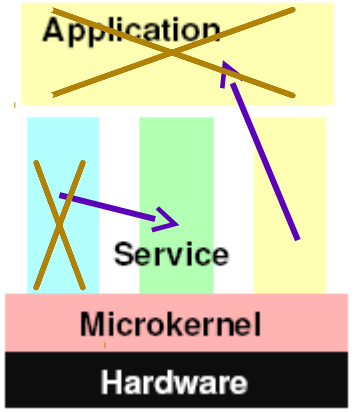
Monolithic Kernel

- All OS services operate in kernel space
- Good performance
  - less context-switch, TLB flush
- Disadvantages
  - Dependencies between system component
  - Complex & huge (millions(!) of lines of code)
  - Larger size makes it hard to maintain
- Examples: MULTICS, Unix, FreeBSD, Linux

TCB (Trusted Computing Base)







System	traditional embedded	Linux/ Windows	Microkernel based
TCB	all code	100,000 LoC	10,000 LoC

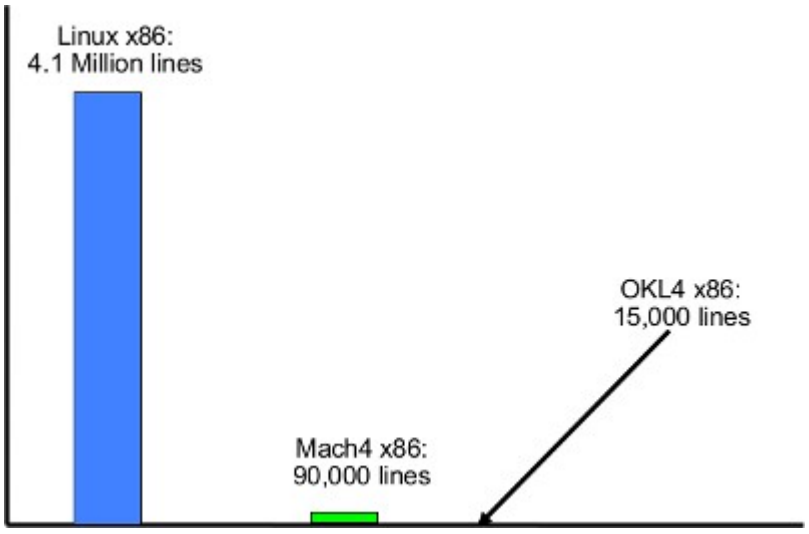
Diagram from Kashin Lin (NEWS Lab)

Case Study: Bugs inside big kernels

- Drivers cause 85% of Windows XP crashes.
  - Michael M. Swift, Brian N. Bershad, Henry M. Levy: “Improving the Reliability of Commodity Operating Systems”, SOSP 2003
- Error rate in Linux drivers is 3x (maximum: 10x)
  - Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson R. Engler: “An Empirical Study of Operating System Errors”, SOSP 2001
- Causes for driver bugs
  - 23% programming error
  - 38% mismatch regarding device specification
  - 39% OS-driver-interface misconceptions
  - Leonid Ryzhyk, Peter Chubb, Ihor Kuz and Gernot Heiser: “Dingo: Taming device drivers”, EuroSys 2009

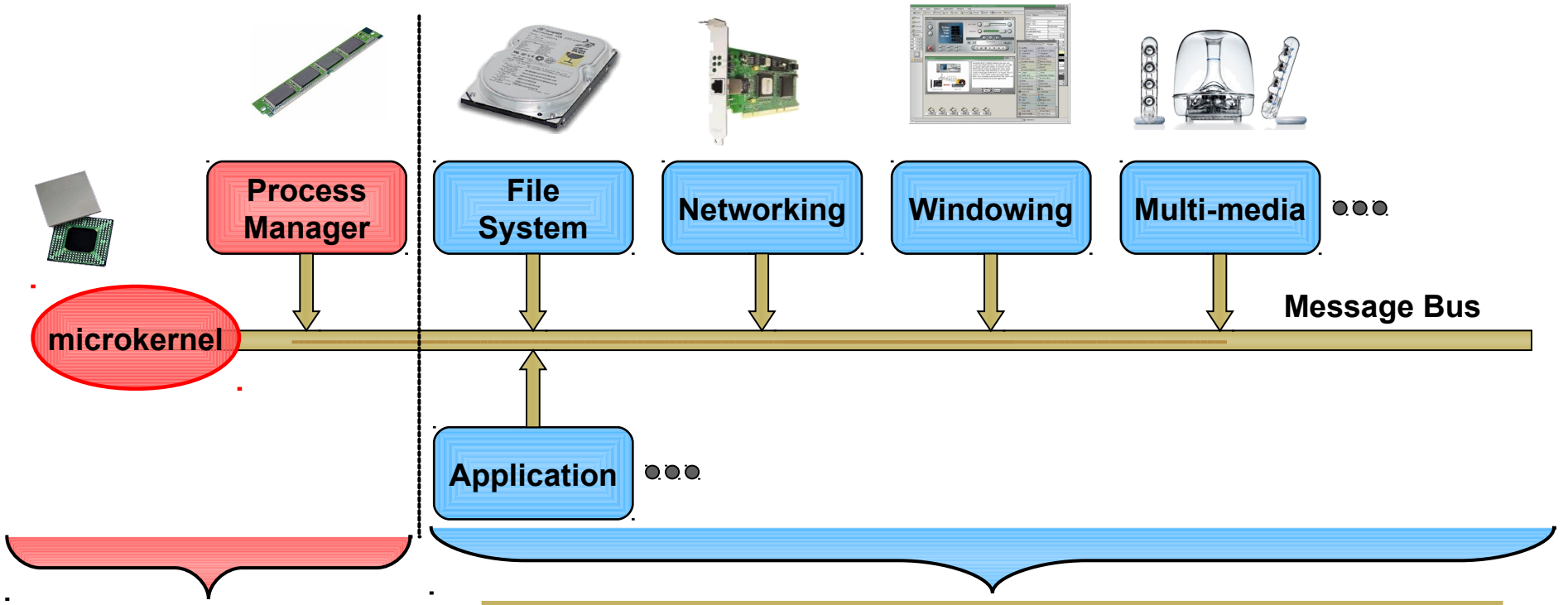
Microkernel

- Minimalist approach
  - IPC, virtual memory, thread scheduling
- Put the rest into user space
  - Device drivers, networking, file system, user interface
- Disadvantages
  - Lots of system calls and context switches
- Examples: Mach, L4, QNX, MINIX, IBM K42



Microkernel

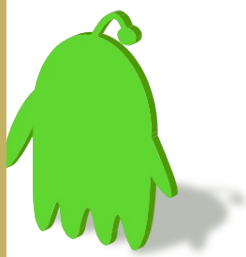
- Put the rest into user space
  - Device drivers, networking, file system, user interface



Microkernel + Process Manager are the only trusted components

Applications and Drivers

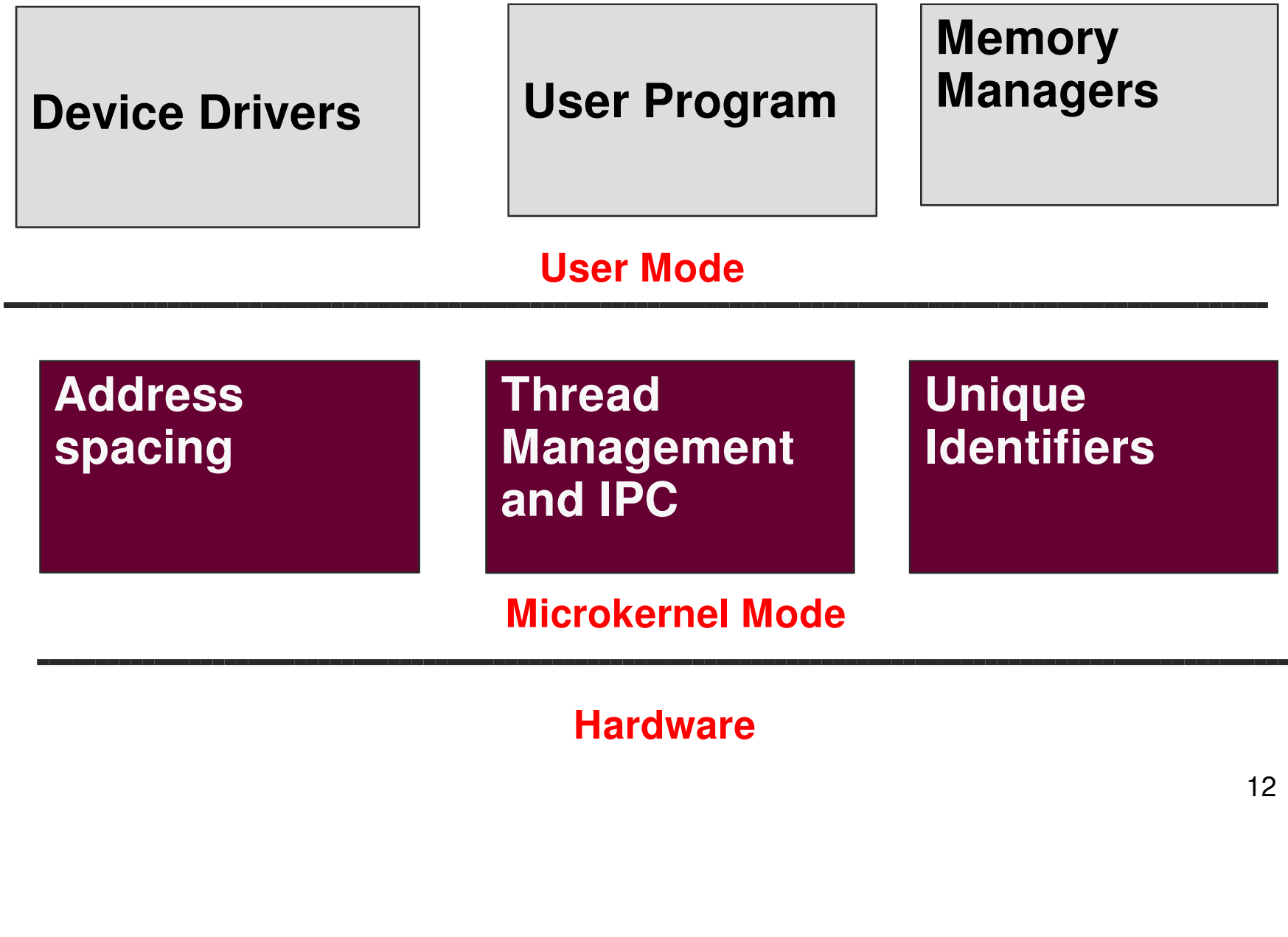
- Are processes which plug into a message bus
  Reside in their own memory-protected address space
- Have a well defined message interface
- Cannot corrupt other software components
- Can be started, stopped and upgraded on the fly



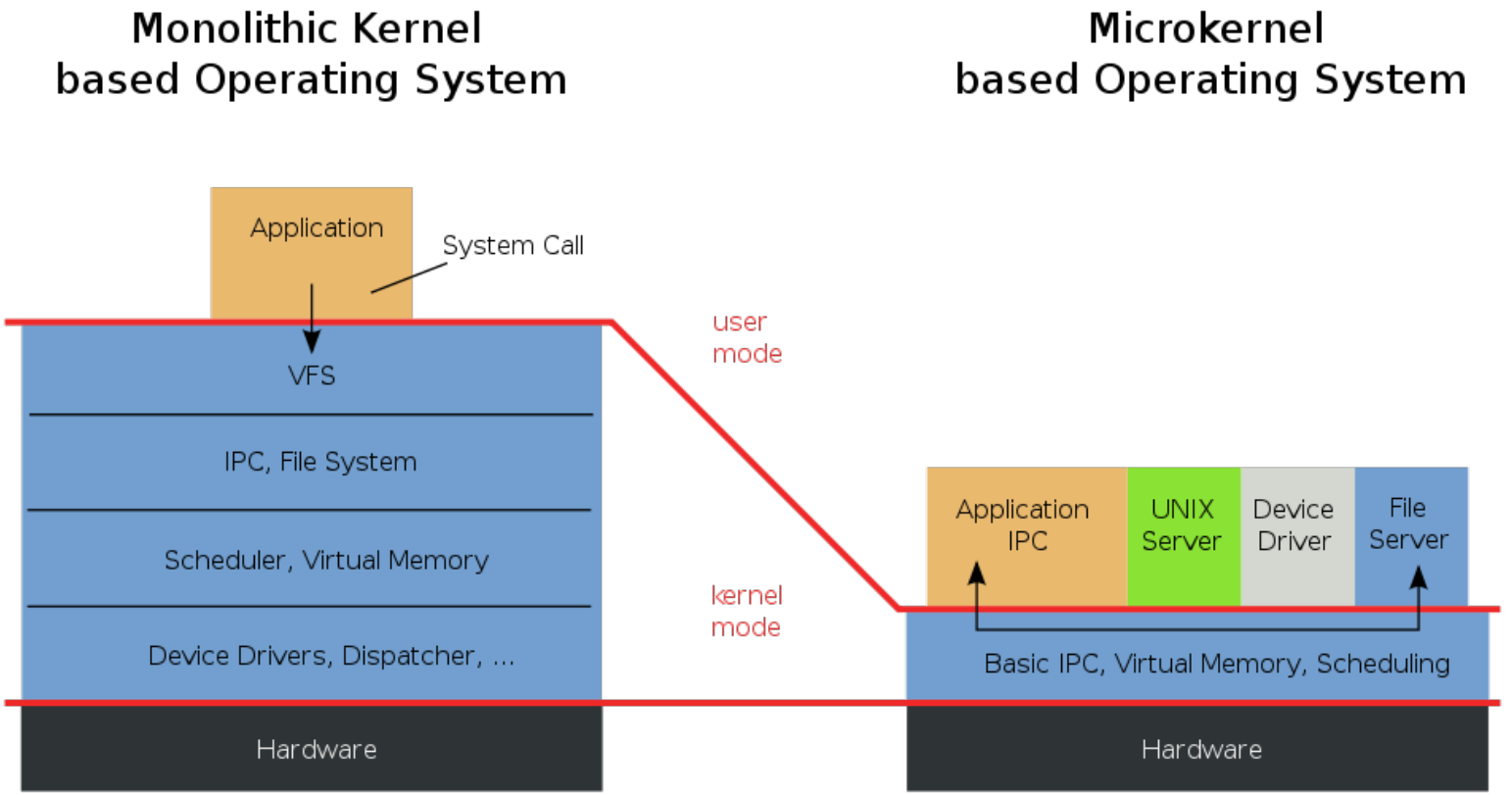
Microkernel: Definitions

- 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

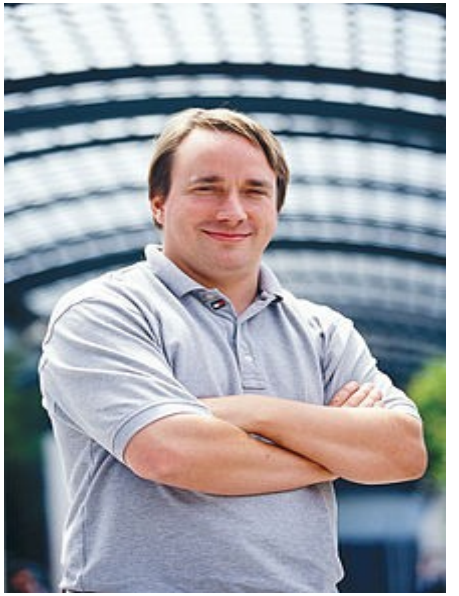


Monolithic vs. Microkernel



Tanenbaum-Torvalds Debate

- Andrew Tanenbaum
  - Author of Minix, Amoeba, Globe
  - Many books on OS and distributed system
  - One of the most influential fellow in OS research



- Linus Torvalds
  - Chief architect of the Linux kernel
  - Most influential fellow in open source

round 1: “Linux is obsolete” (1992)

- Historical context – early 1990s:
  - AT&T USL vs BSDi lawsuit.
  - GNU kernel was not out yet.
- Andrew:
  - “Linux is obsolete”
  - “The limitations of Minix are partly due to my being a prof.”
  - w.r.t OS design, the debate is over. “Microkernels have won.” and “Linux is a giant step back into the 1970”.
  - “writing a new operating system that is closely tied to any particular piece of hardware, especially a weird one like the Intel line, is basically wrong.”
  - “Designing a monolithic kernel in 1991 is a fundamental error. Be thankful you are not my student. You would not get a high grade for such a design :-)”

Linus' response

- *On "for me MINIX is a hobby"*: "You use this as an excuse for the limitations of minix? Sorry, but you loose: I've got more excuses than you have, and linux still beats the pants of minix in almost all areas. Look at who makes money off minix, and who gives linux out for free"
- *On "Minix is a micro-kernel design, Linux is a monolithic style sys":* "If this was the only criterion for the "goodness" of a kernel, you'd be right. What you don't mention is that minix doesn't do the micro-kernel thing very well, and has problems with real multitasking (in the kernel). If I had made an OS that had problems with a multithreading filesystem, I wouldn't be so fast to condemn others: in fact, I'd do my damndest to make others forget about the fiasco."
- "Portability is for people who cannot write new programs": "I agree that portability is a good thing: but only where it actually has some meaning. There is no idea in trying to make an operating system overly portable: adhering to a portable API is good enough. The very idea of an operating system is to use the hardware features, and hide them behind a layer of

round 2: Minix 3 (2006)

- Context: “Can we make OS reliable and secure?” & Slashdot
- Andrew:
  - “Be sure brain is in gear before engaging mouth”
  - Encourage people to try Minix3 before making comments about uKernel
- Linus:
  - “The real reason people do microkernels .. and make up new reasons for why they are better is that the *concept* sounds so good. It sounded good to me too!”
  - “...real progress is made. Not by people who are afraid of the complexity. Real progress is made by people who are too ignorant to even realize that there **is** complexity, and how things are "supposed" to work, and they just do it their own way.
  - Over-thinking things is never how you do anything real.”

Reliability, Security, and Complexity

- Andrew:
  - Reliability & Security have now become more important than performance for most users. Systems in military and aerospace uses uKernel.
  - Complexity:

    Shared data structure is bad and hard to get right. “you want to avoid shared data structures as much as possible”. “That's what object-oriented programming is all about” (*thousands* of bugs have been found in the Linux kernel in this area alone)

    Distributed algorithm is only a problem for multiple machines. Complexity is manageable, we implemented it.

- Linus:
  - Reliability & Security: “The fact that each individual piece is simple and secure does not make the aggregate either simple **or** secure”, “when one node goes down, often the rest comes down too.”
  - Complexity:

    Shared data structure is good, beside performance benefit, makes it easier to develop system: “in the absense of a shared state, you have a hell of a lot of problems trying to make any decision that spans more than one entity in the system.”

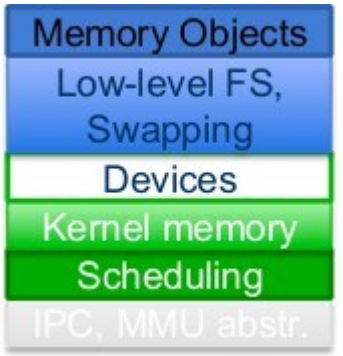
    uKernel requires distributed algorithms which are difficult to get right.. “whenever you compare the speed of development of a microkernel and a traditional kernel, the traditional kernel wins. By a huge amount, too.”

3 Generations of Microkernel

- Mach (1985-1994)
  - replace pipes with IPC (more general)
  - improved stability (vs monolithic kernels)
  - poor performance
- L3 & L4 (1990-2001)
  - order of magnitude improvement in IPC performance
    - written in assembly, sacrificed CPU portability
    - only synchronus IPC (build async on top of sync)
  - very small kernel: more functions moved to userspace
- seL4, Fiasco.OC, Coyotos, NOVA (2000-)
  - platform independence
  - verification, security, multiple CPUs, etc.

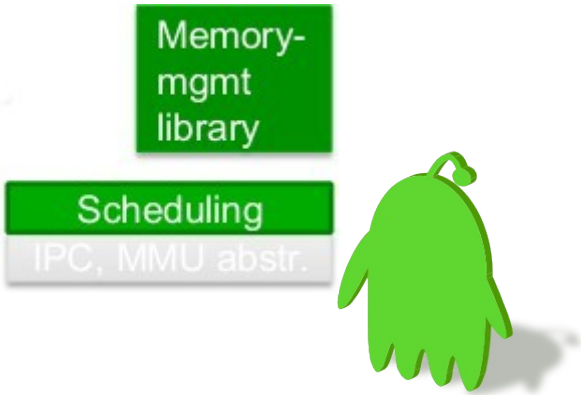
3 Generations of Microkernel

- Generation 1: Mach (1985-1994)



- Generation 2: L3 & L4 (1990-2001)

- Generation 3: seL4, Fiasco.OC, NOVA (2000-)



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
- Examples: Windows NT, NetWare, BeOS, Darwin (MacOS X), DragonFly BSD
  - Darwin is built around XNU, a hybrid kernel that combines the Mach 3 microkernel, various elements of BSD (including the process model, network stack, and virtual file system), and an object-oriented device driver API called I/O Kit.

Exokernel

- Follows end-to-end principle ("Library" OS)
  - Extremely minimal
  - Fewest hardware abstractions as possible
  - Just allocates physical resources to apps
- Disadvantages
  - More work for application developers
- Examples: MIT Exokernel, Nemesis

“Worse Is Better", Richard P. Gabriel

	New Jersey style [UNIX, Bell Labs]	MIT style [Multics]
Simplicity	No.1 consideration Implementation > Interface	Interface > Implementation
Correctness	mostly	100%
Consistency	mostly	100%
Completeness	de facto	mostly

Microkernel: 1st Generation

Contrasting Reading

- Paper: The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System
  - SOSP 1987, Young et al
- Examples: CMU Mach (1985), Chorus (1987)

- Richard Rashid
  - Lead developer of Mach
  - Lead, Microsoft Research
- William Bolosky
  - Microsoft Research
- Avie Tevanian
  - former Apple CTO
- Brian Bershad
  - Professor, University of Washington

Mach

- 1st generation microkernel
  - OpenStep / Apple Darwin (XNU), GNU Hurd, IBM Workplace OS, OSF/1
- Derived from CMU Accent operating system
  - Accent has no ability to execute UNIX applications (1981)
  - Mach = BSD Unix system + Accent concepts
- Memory object
  - Mange system services like network paging and file system
- Memory via communication

UNIX was owned by AT&T which controlled the market. IBM, DEC, etc. got together and formed the OSF (Open Software Foundation). In an effort to conquer market share, OSF took the Mach 2.5 release and made it the OSF/1 system.

Mach failed because of performance

- Mach were notoriously noted for suffering from excessive performance limitations.
- Mach on a DEC-Station 5200/200 was found to endure peak degradations of up to 66% when compared to Ultrix running on the same hardware. [1]
- Mach-based OSF/1 is cited to perform on average at only half the performance level of monolithic OSF/1. [2]

[1] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In Proceedings of the 14th ACM Symposium on OS Principles, pages 120–133, Asheville, NC, USA, December 1993.
[2] Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, Cambridge, 1994.

Mach in a nutshell

- Simple, extensible communication kernel
  - "Everything is a pipe." – ports as secure communication channels
- Multiprocessor support
- Message passing by mapping
- Multi-server OS personality
- POSIX-compatibility
- Shortcomings
  - Performance
  - drivers still in the kernel

Mach Design Principles

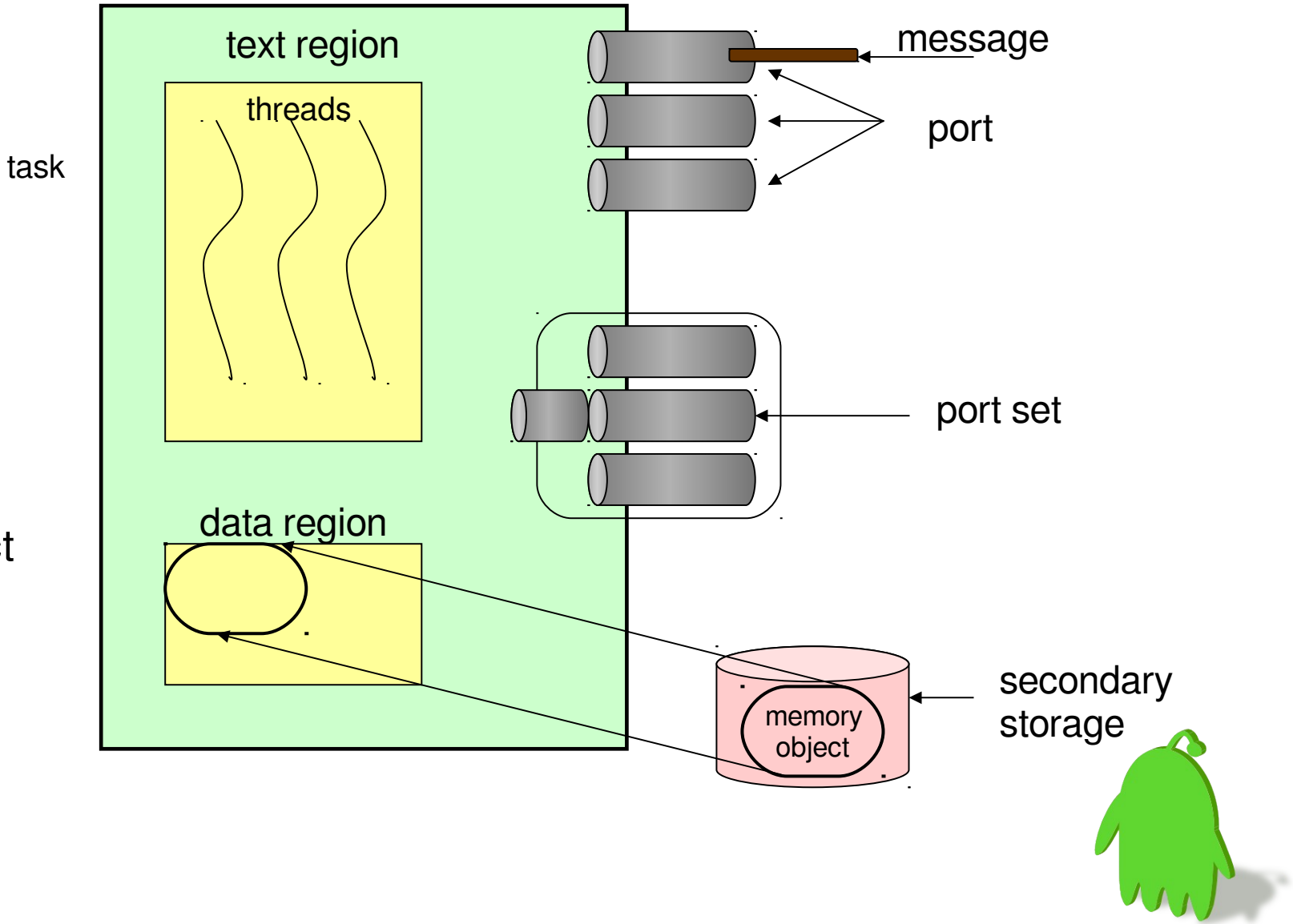
- Maintain BSD compatibility
  - Simple programmer interface
  - Easy portability
  - Extensive library of utilities / applications
  - Combine utilities via pipes
- In addition,
  - Diverse architectures
  - Varying network speed
  - Simple kernel
  - Distributed operation
  - Integrated memory management and IPC
  - Heterogeneous systems

Mach Abstractions

- Task
  - Basic unit of resource allocation
  - Virtual address space, communication capabilities
- Thread
  - Basic unit of computation
- Port
  - Communication channel for IPC
- Message
  - May contain port capabilities, pointers
- Memory Object

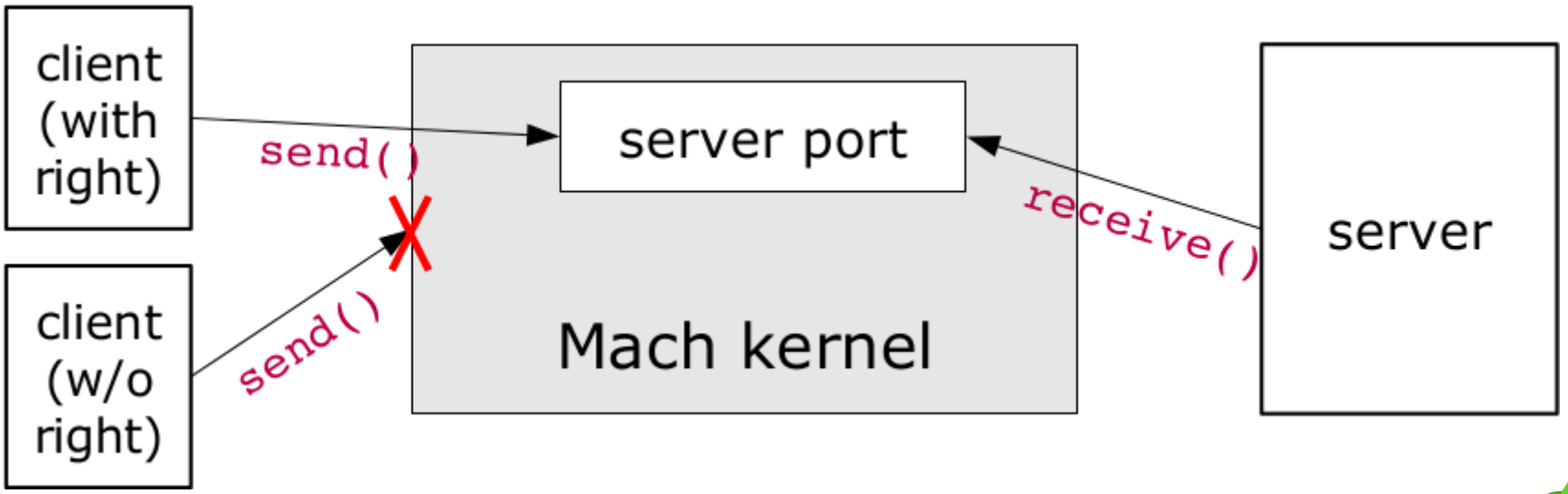
Mach Abstractions

- Task
- Thread
- Port
- Port set
- Message
- Memory object



Ports in Mach microkernel

- Dedicated kernel objects
- Applications hold send/recv rights for ports
- Kernel checks whether task owns sufficient rights before doing IPC



Memory Management and IPC

- Memory Management using IPC:
  - Memory object represented by port(s)
  - IPC messages are sent to those ports to request operation on the object
  - Memory objects can be remote → kernel caches the contents
- IPC using memory-management techniques:
  - Pass message by moving pointers to shared memory objects
  - Virtual-memory remapping to transfer large contents (virtual copy or copy-on-write)

External Memory Management

- No kernel-based file system
  - Kernel is just a cache manager
- Memory object
  - As known as “paging object”
- Page
  - Task that implements memory object

Benefits from such design

- Keypoint: consistent network shared memory
- Examples:
  - Each client maps X with shared pager
  - Use primitives to tell kernel cache what to do:
    - Locking
    - Flushing

Problems of External Memory Management

- External data manager failure looks like communication failure
  - timeouts are needed
- Opportunities for data manager to deadlock on itself

Performance Impacts

- Does not prohibit caching
- Reduce number of copies of data occupying memory
  - Copy-to-use, copy-to-kernel
  - More memory for caching
- “compiling a small program cached in memory…is twice as fast”
- I/O operations reduced by a factor of 10
- Context switch overhead
  - Cost of kernel overhead can be up to 800 cycles.
- Address Space Switches
  - Expensive Page Table and Segment Switch Overhead
  - Untagged TLB = Bad performance

Microkernel: 2nd Generation

Contrasting Reading

- Paper: On μ-Kernel Construction
  - 15th Symposium on Operating System Principles (1995)
  - Microkernels can have adequate performance if they are architecture-dependent.
- Jochen Liedtke (1953-2001)
  - worked on Eumel, L3, L4
  - worked at IBM Research and GMD (German National Research Center for IT)

”Our vision is a microkernel technology that can be and is used advantageously for constructing any general or customized operating system”. [Liedtke et al 2001]

What are going into Kernel?

- Only determined by function, not performance.
- Key issue is protection.
  - Principle of independence
    - Servers can't stomp on each other.
  - Principle of integrity
    - Communication channels can't be interfered with.
- Usually the following go into the kernel:
  - Address spaces
  - IPC
  - Basic scheduling

L4: the 2nd Generation

- Similar to Mach
  - Started from scratch, rather than monolithic
  - But even more minimal
- minimality principle for L4:

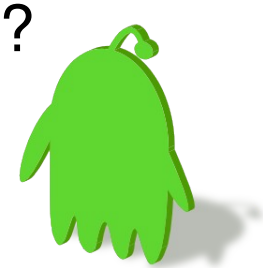
  A concept is tolerated inside the microkernel only if moving it outside the kernel, *i.e.*, permitting competing implementations, would prevent the implementation of the system's required functionality.
- Tasks, threads, IPC
  - Contains only 13 system calls. 7 of them are for IPC
  - Uses only 12k of memory

Recall: Mach kernel

- API Size: 140 functions
  - Asynchronus IPC
  - Threads
  - Scheduling
  - Memory management
  - Resource access permissions
  - Device drivers (in some variants)
- All other functions are implemented outside the kernel.

Performance Issues of Mach kernel

- 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?



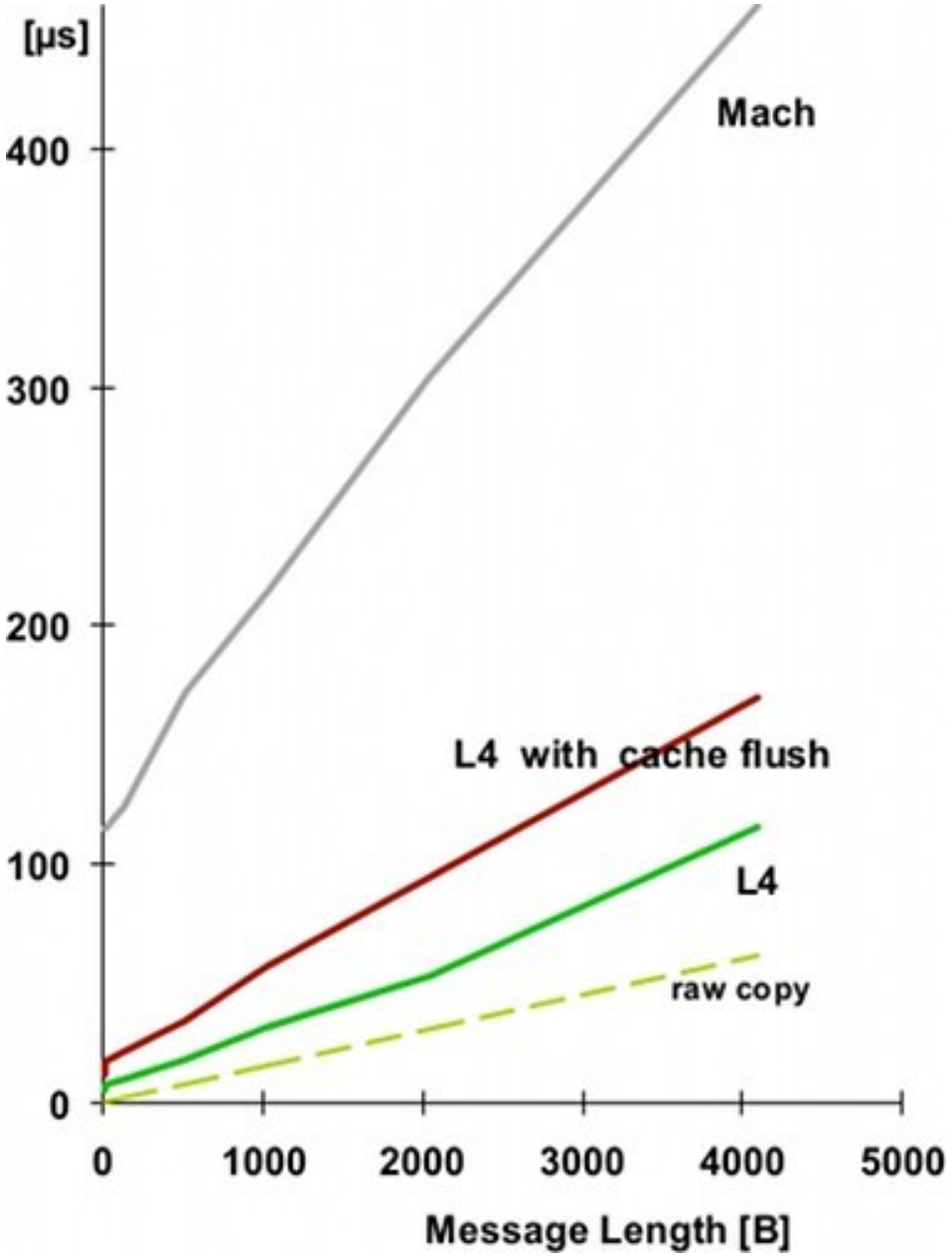
Performance Issues for 1st Generation

- First-generation microkernels were slow
- Reason: Poor design [Liedtke SOSP'95]
  - complex API
  - Too many features
  - Poor design and implementation
  - Large cache footprint ⇒ memory-bandwidth limited
- L4 is fast due to small cache footprint
  - 10–14 I-cache lines
  - 8 D-cache lines
  - Small cache footprint ⇒ CPU limited

Designs taken by L4 over Mach

- API Size: 7 functions (vs 140 for Mach3)
  - Synchronous IPC
  - Threads
  - Scheduling
  - Memory management

IPC Performance of L4



- “Radical” approach
- Strict minimality
- From-scratch design
- Fast primitives

Typical code size of 2nd Microkernel

- Source code (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
x86	seL4	74k	98k
ARMv6	seL4	64k	112k

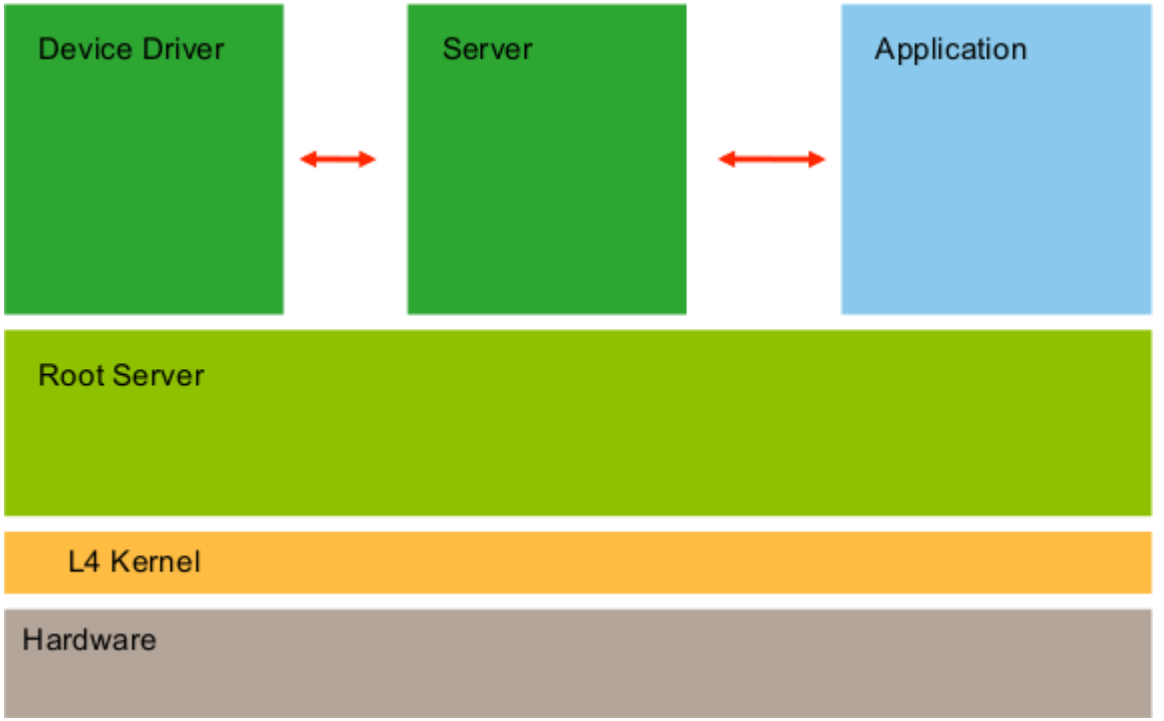
L4 Abstractions, Mechanisms, Concepts

- 3 basic Abstractions
  - Address spaces (or virtual MMUs) — for protection
  - Threads (or virtual CPUs) — for execution
  - Capabilities (for naming and access control) — from OKL4 2.1
  - Time (for scheduling) — removed recently
- 2 basic Mechanisms
  - Message-passing communication (IPC)
  - Mapping memory to address spaces
- Other core concepts
  - Root task — Removed in OKL4 2.2
  - Exceptions

L4 based Systems

(each variant differs in some parts)

- L4 kernel
  - variants
- Device Driver
- Server
- Application



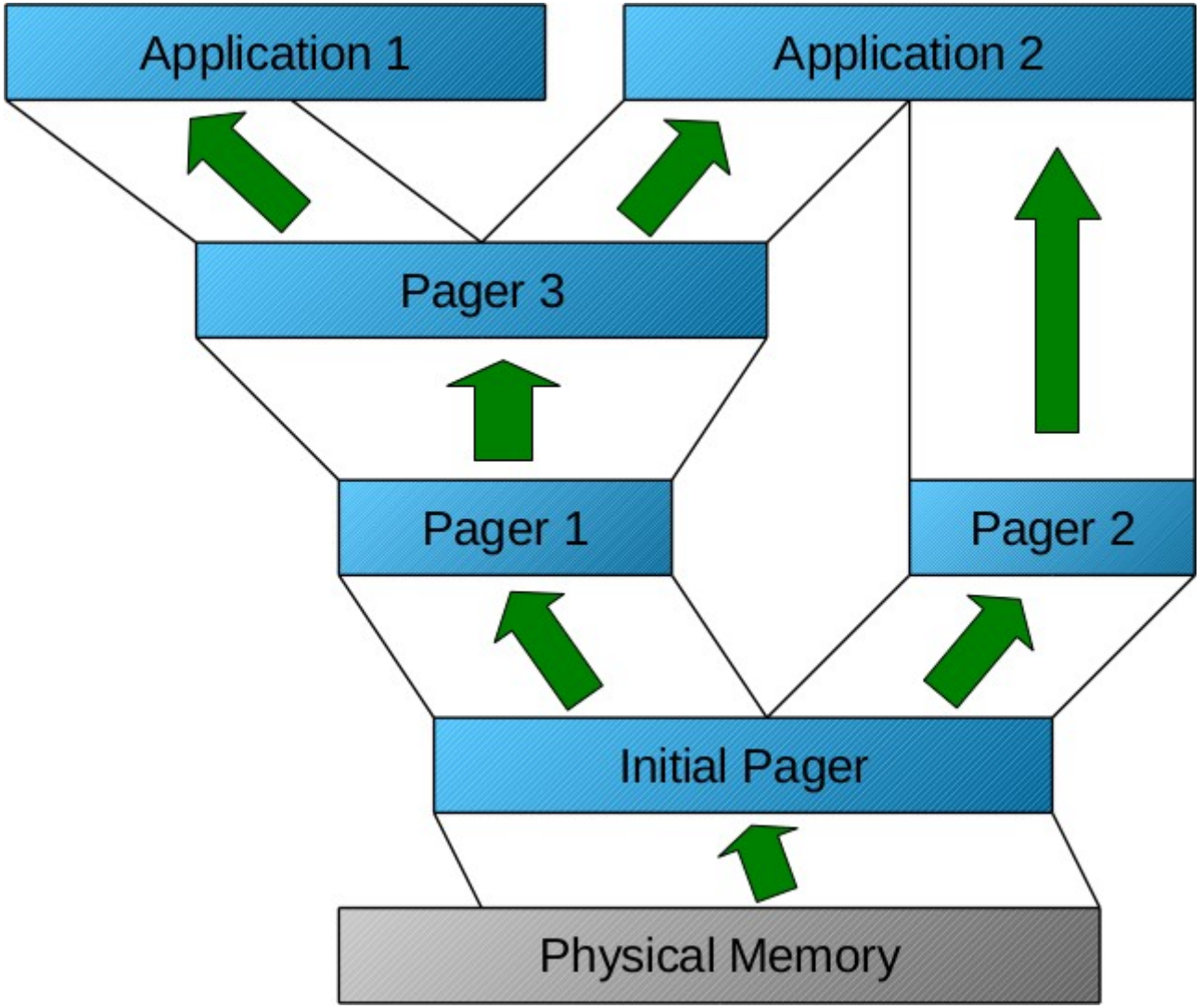
Abstractions: Address Spaces

- Definition: A mapping which associates each virtual page to a physical page. (Liedtke)
- microkernel has to hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible
- microkernel concept of address spaces must be tamed, but must permit the implementation of arbitrary protection schemes on top of microkernel

Abstractions: Address Spaces (AS)

- Address space is unit of protection
  - Initially empty
  - Populated by mapping in frames
- Mapping performed by privileged MapControl() syscall
  - Can only be called from root task
  - Also used for revoking mappings (unmap operation)
- Root task *[removed]*
  - Initial AS created at boot time, controlling system resources
  - Privileged system calls can only be performed from root task
  - privileged syscalls identified by names ending in “Control

Abstractions: Address Spaces - Recursive [removed in OKL4]



Abstractions: Address Spaces

- L4 provides 3 operations:
  - Grant

    owner of an address space can grant any of its pages to another space
  - Map

    owner of an address space can map any of its pages to another space
  - Flush

    owner of an address space can flush any of its pages

Abstractions: Address Spaces - I/O

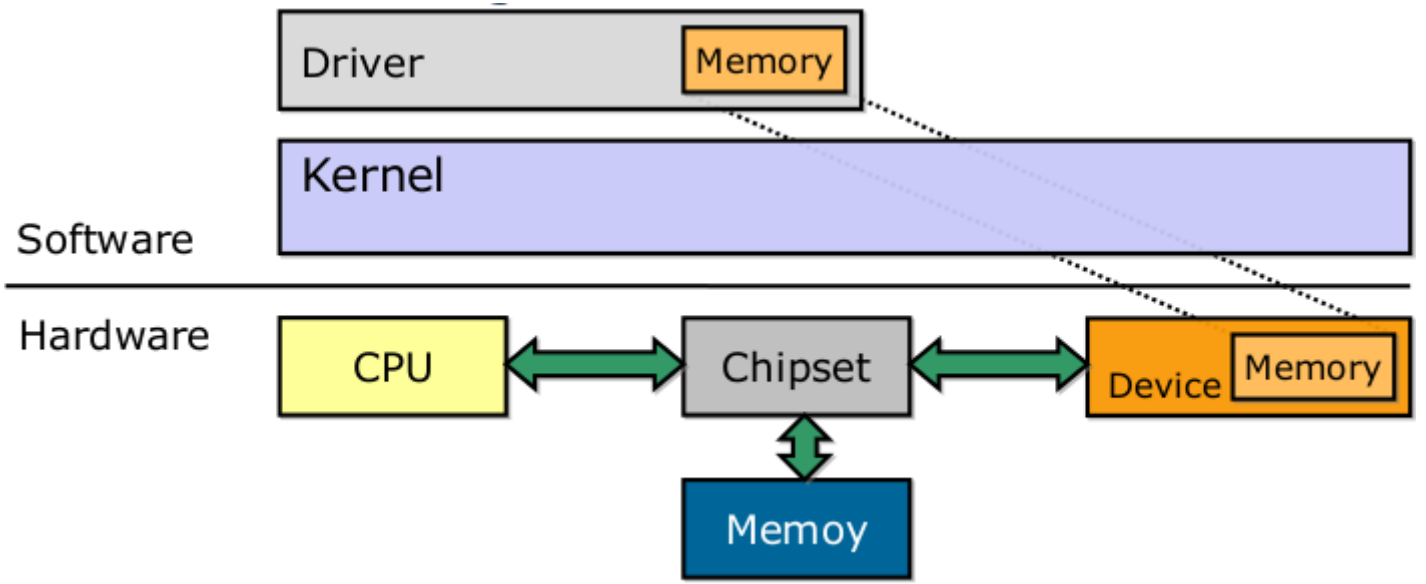
- I/O ports treated as address space
- As address space, can be mapped and unmapped
- Hardware interrupts are handled by user-level processes. The L4 kernel will send a message via IPC.

Abstractions: Address Spaces - I/O

- Recall that I/O can be done either with special “ports”, or memory-mapped.
- Incorporated into the address space
  - Natural for memory-mapped I/O (RISC series)
  - Also works on I/O ports (x86 permits control per port, but no mapping)
- Do not confuse memory-mapped I/O with memory-mapped files.

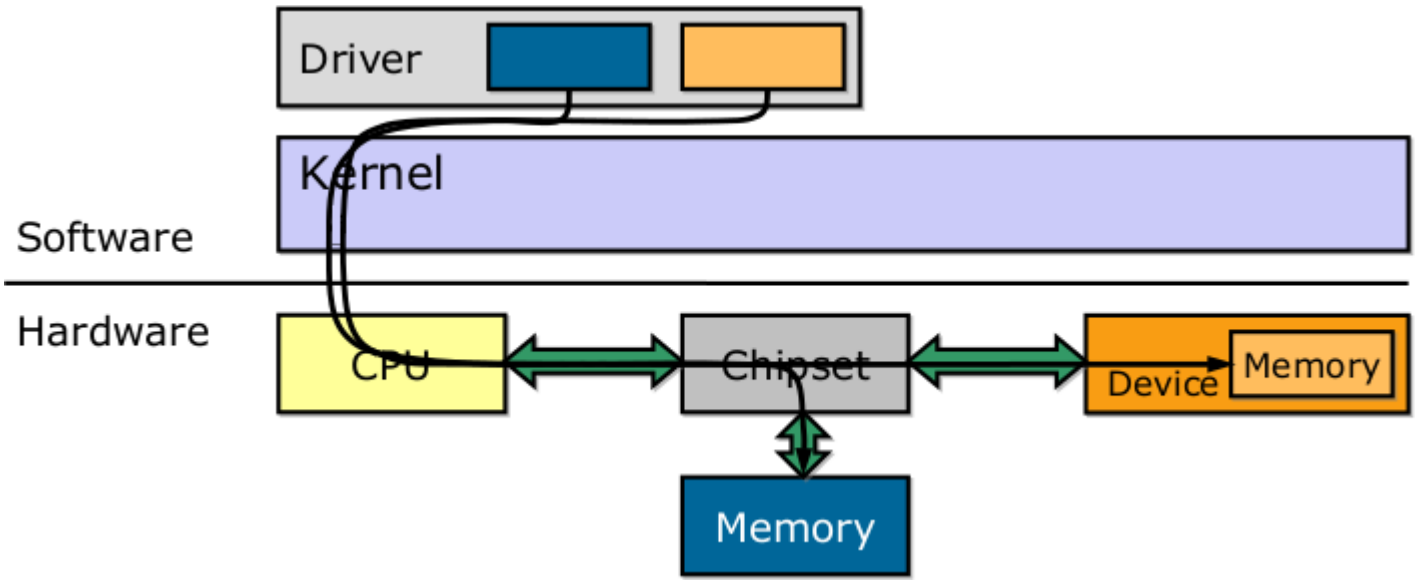
Abstractions: AS - I/O Memory

- Devices often contain on-chip memory (NIC, graphcis, etc)
- Instead of accessing through I/O ports, drivers can map this memory into their address space just like normal RAM
  - no need for special instructions
  - increased flexibility by using underlying virtual memory management



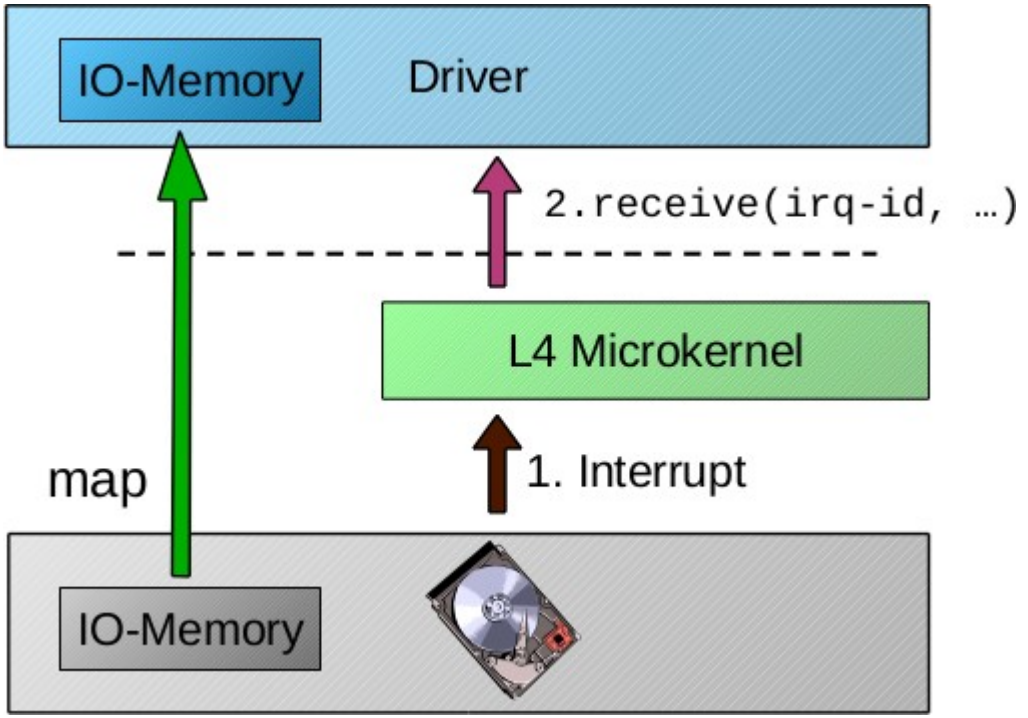
Abstractions: AS - I/O Memory

- Device memory looks just like physical memory
- Chipset needs to
  - map I/O memory to exclusive address ranges
  - distinguish physical and I/O memory access

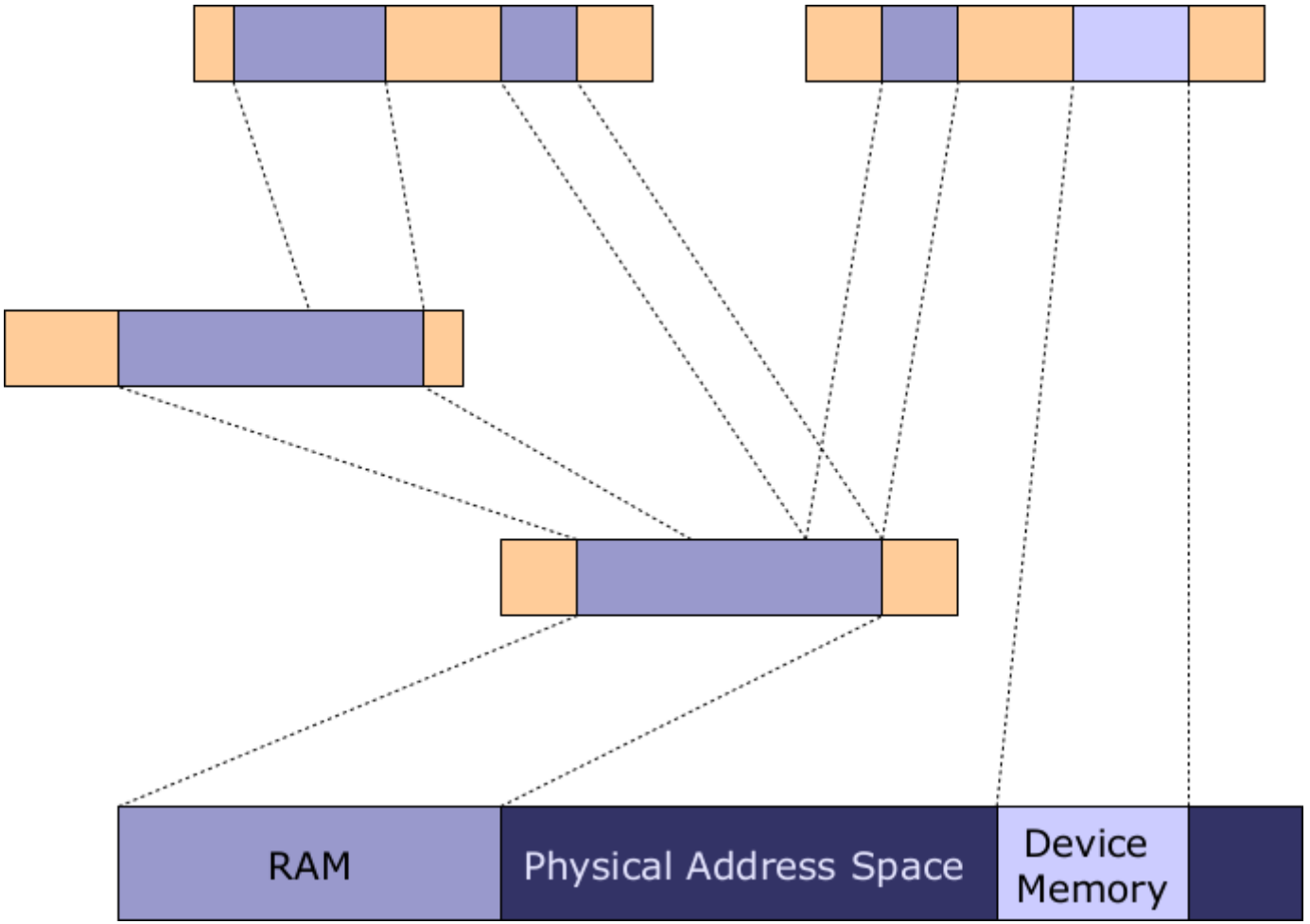


Abstractions: AS - Device Driver

- Hardware interrupts: mapped to IPC
- I/O memory & I/O ports: mapped via flexpages



Abstractions: Address Spaces - Recursive

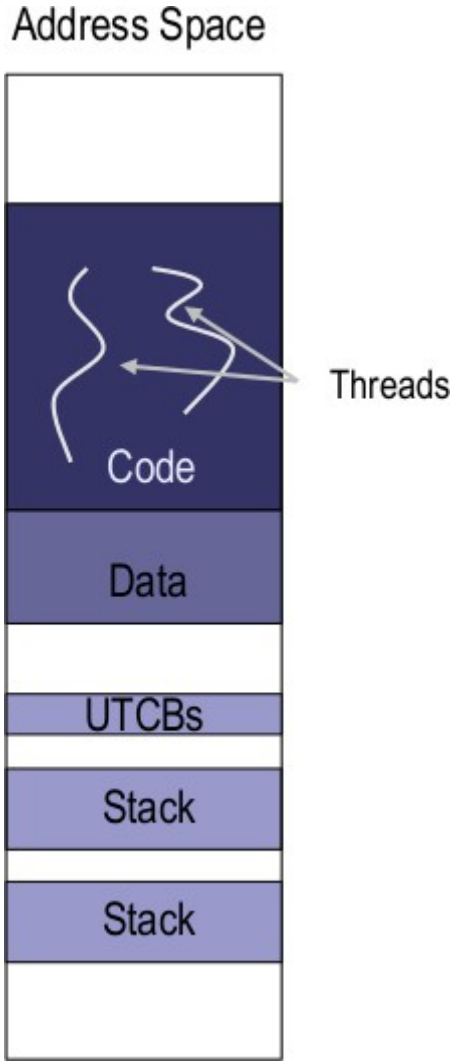


Abstractions: Threads

- Thread is unit of execution
  - Kernel-scheduled
- Thread is addressable unit for IPC
  - Thread capability used for addressing and establishing send rights
  - Called Thread-ID for backward compatibility
- Threads managed by user-level servers
  - Creation, destruction, association with address space
- Thread attributes:
  - Scheduling parameters (time slice, priority)
  - Unique ID (hidden from userland)
    - referenced via thread capability (local name)
  - Address space
  - Page-fault and exception handler

Abstractions: Threads

- Thread
  - Implemented as kernel object
- Properties managed by the kernel:
  - Instruction Pointer (EIP)
  - Stack (ESP)
  - Registers
  - User-level TCB
- User-level applications need to
  - allocate stack memory
  - provide memory for application binary
  - find entry point



Abstractions: Time

- Used for scheduling times slices
  - Thread has fixed-length time slice for preemption
  - Time slices allocated from (finite or infinite) time quantum
    - Notification when exceeded
- In earlier L4 versions also used for IPC timeouts
  - Removed in OKL4
- Future versions remove time completely from the kernel
  - If scheduling (including timer management) is completely exported to user level

Mechanism: IPC

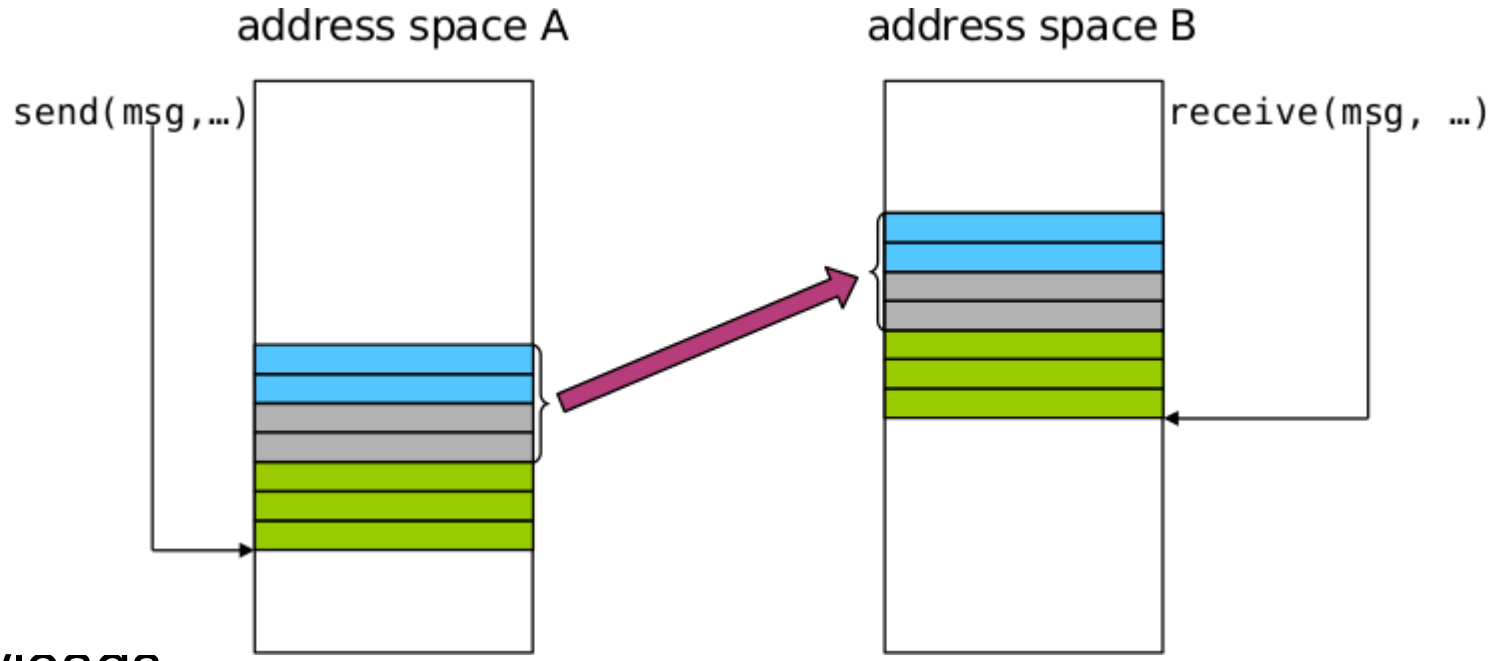
- Synchronous message-passing operation
- Data copied directly from sender to receiver
  - Short messages passed in registers
  - Long messages copied by kernel (semi-)asynchronously [OKL4]
- Can be blocking or polling (fail if partner not ready)
- Asynchronous notification variant
  - No data transfer, only sets notification bit in receiver
  - Receiver can wait (block) or poll
- In earlier L4 versions [removed in OKL4]:
  - IPC also used for mapping
  - long synchronous messages

IPC: Inter-Process Communication

- Definition: Exchange of data between 2 process.
  - IPC is one way communication
  - RPC (remote procedure call) is round trip communication
- The microkernel handles message transfers between threads.
- Grant and Map operations rely on IPC.

Mechanism: IPC

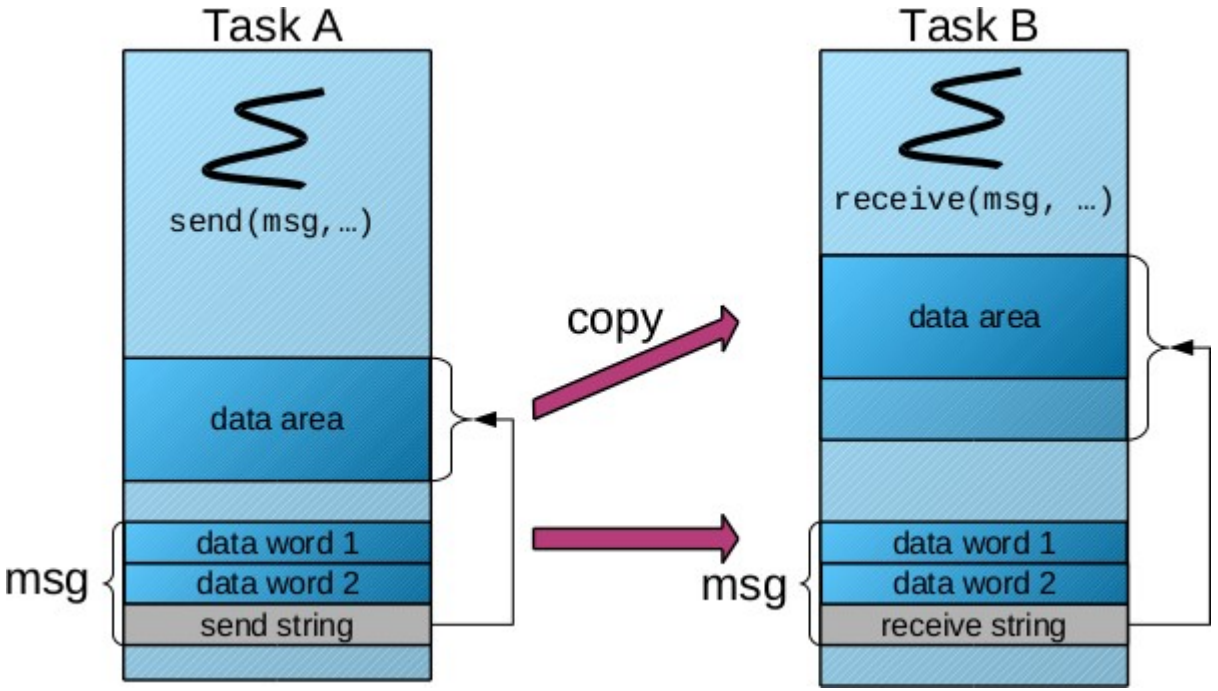
- synchronous (no buffering)



- Payloads
  - registers only (short IPC), fast
  - strings (long IPC)
  - access rights (“mappings”)
  - Faults
  - interrupts

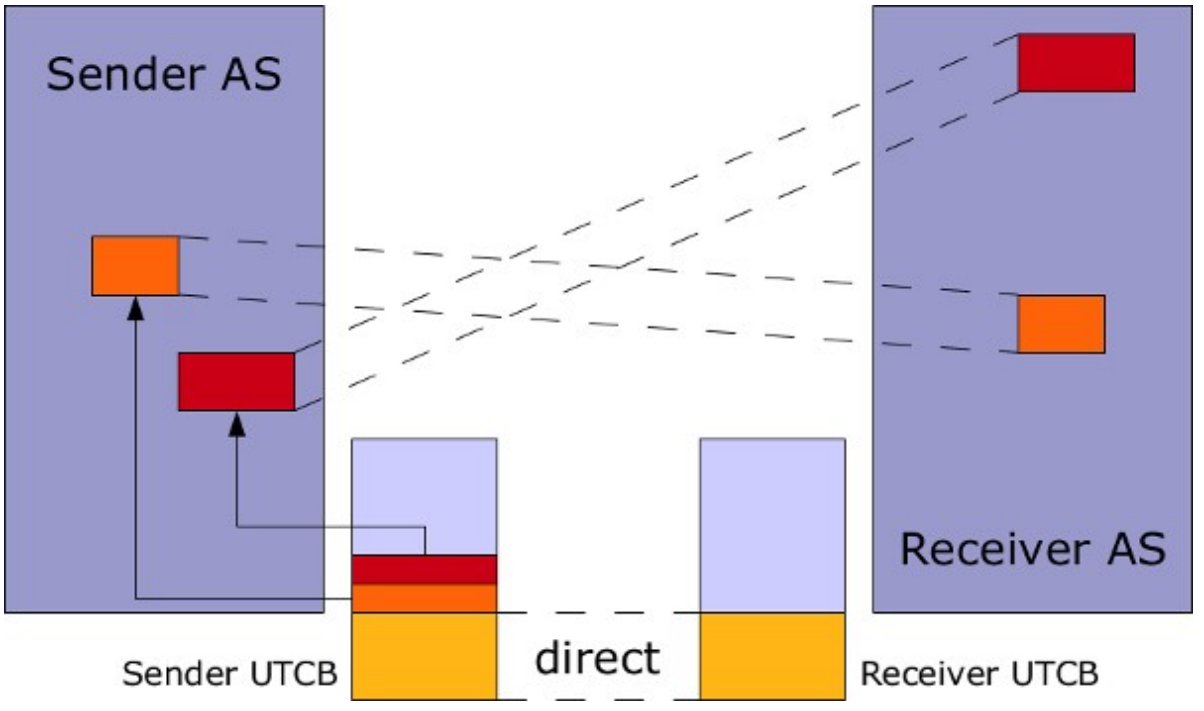
Mechanism: IPC - Copy Data

- Direct and indirect data copy
- UTCB (User-level Thread Control Block) message
  (special area; shared memory between user and kernel)
- Special case: register-only message
- Page faults during user-level memory access possible



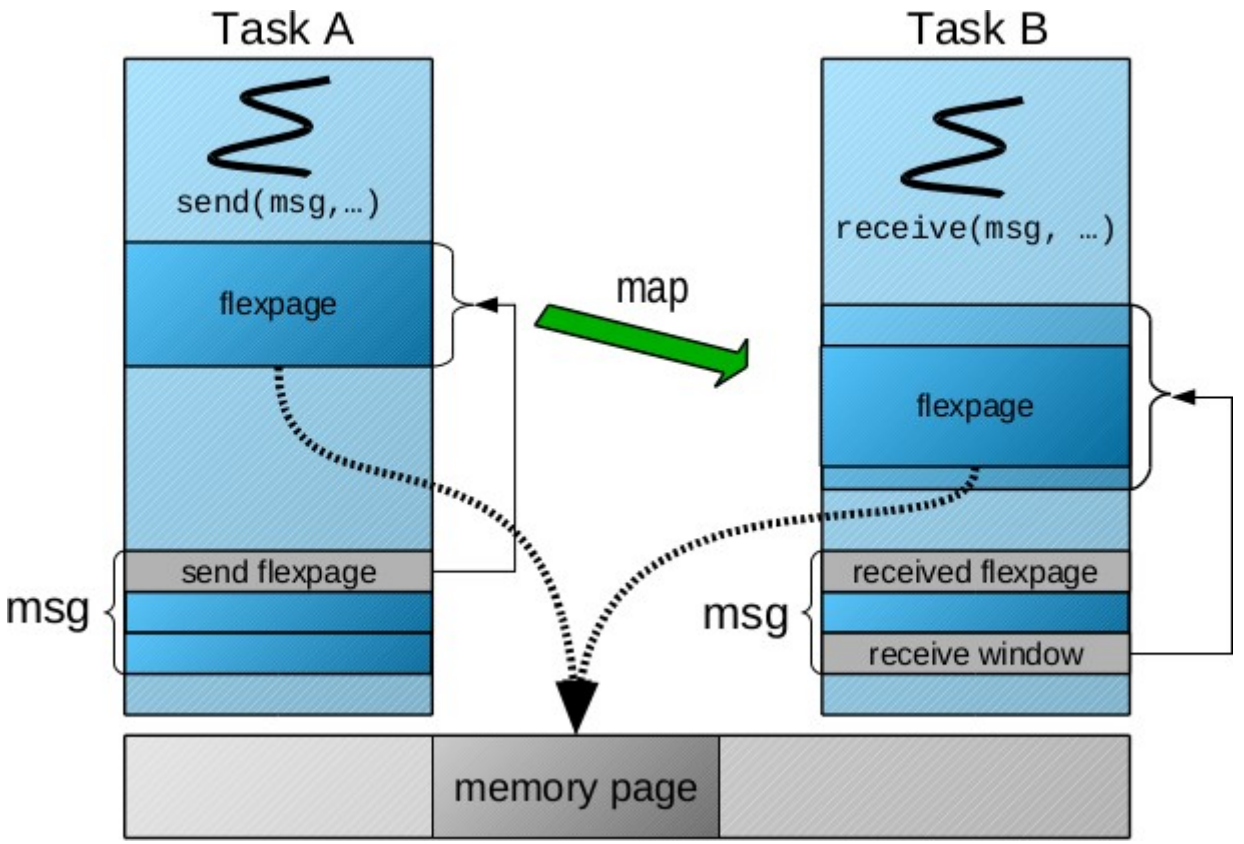
Mechanism: IPC - Direct/Indirect Copy

- UTCB: set of “virtual” registers
- Message Registers
  - Message Registers System call parameters
  - IPC: direct copy to receiver
- Buffer Registers
  - Receive flexpage descriptors
- Thread Control Registers
  - Thread-private data
  - Preserved, not copied



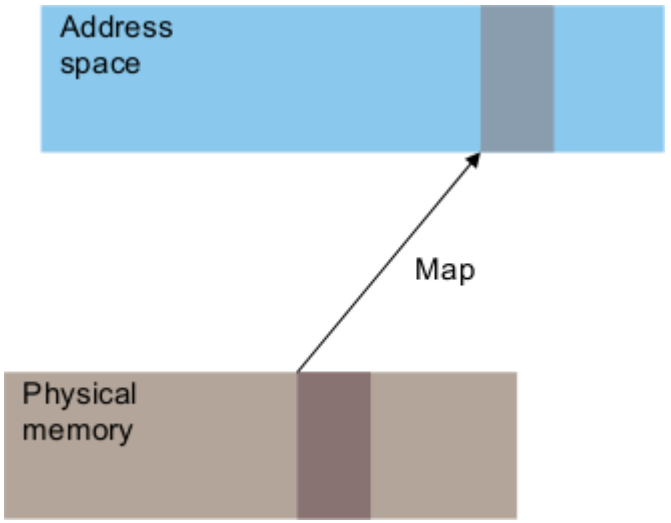
Message: Map Reference

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



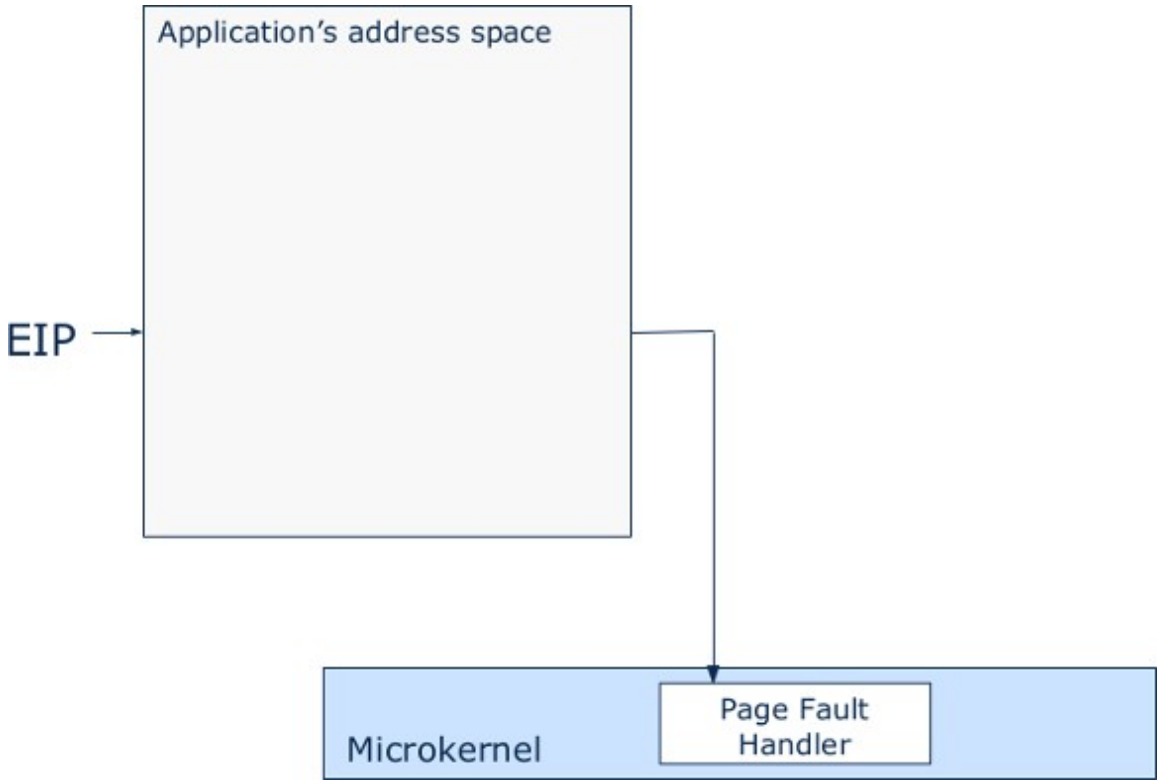
Mechanism: Mapping

- Create a mapping from a physical frame to a page in an address space
  - Privileged syscall MapControl
  - unprivileged in newer L4 (access control via memory caps)
- Typically done in response to page fault
  - VM server acting as pager
  - can pre-map, of course
- Also used for mapping device
  - VM server acting as pager
  - can pre-map, of course



Page Fault Handling

- Page fault exception is caught by kernel page fault handler
- No management of user memory in kernel
- Invoke user-level memory management
  - Pager

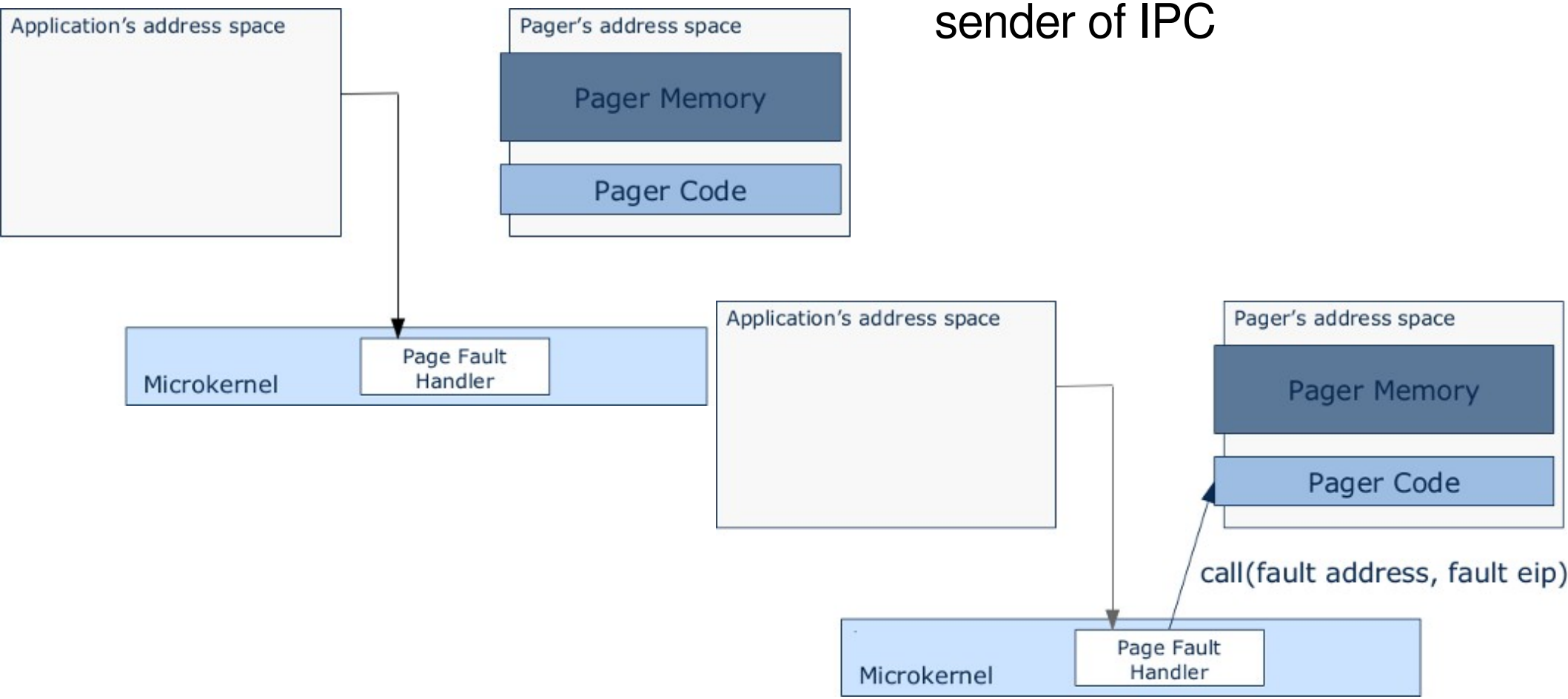


Pager

- Pager is the thread invoked on page fault
- Each thread has a (potentially different) pager assigned

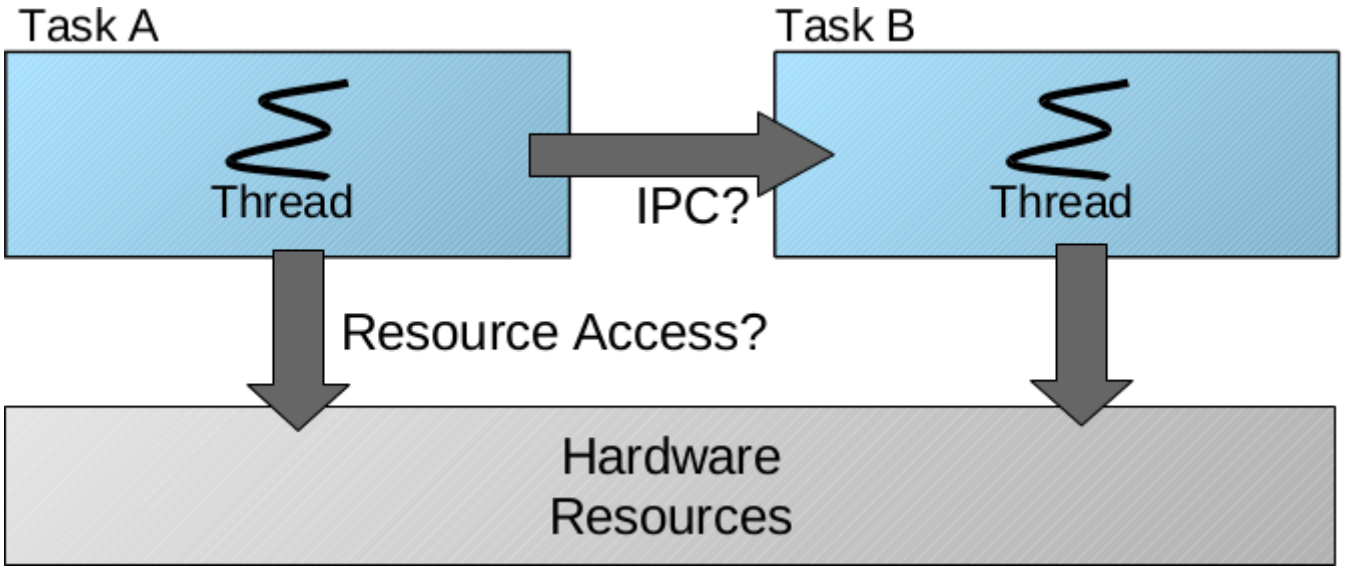
Pager Invocation

- Communication with pager thread using IPC
- Kernel page fault handler sets up IPC to pager
- Pager sees faulting thread as sender of IPC



Communication and Resource Control

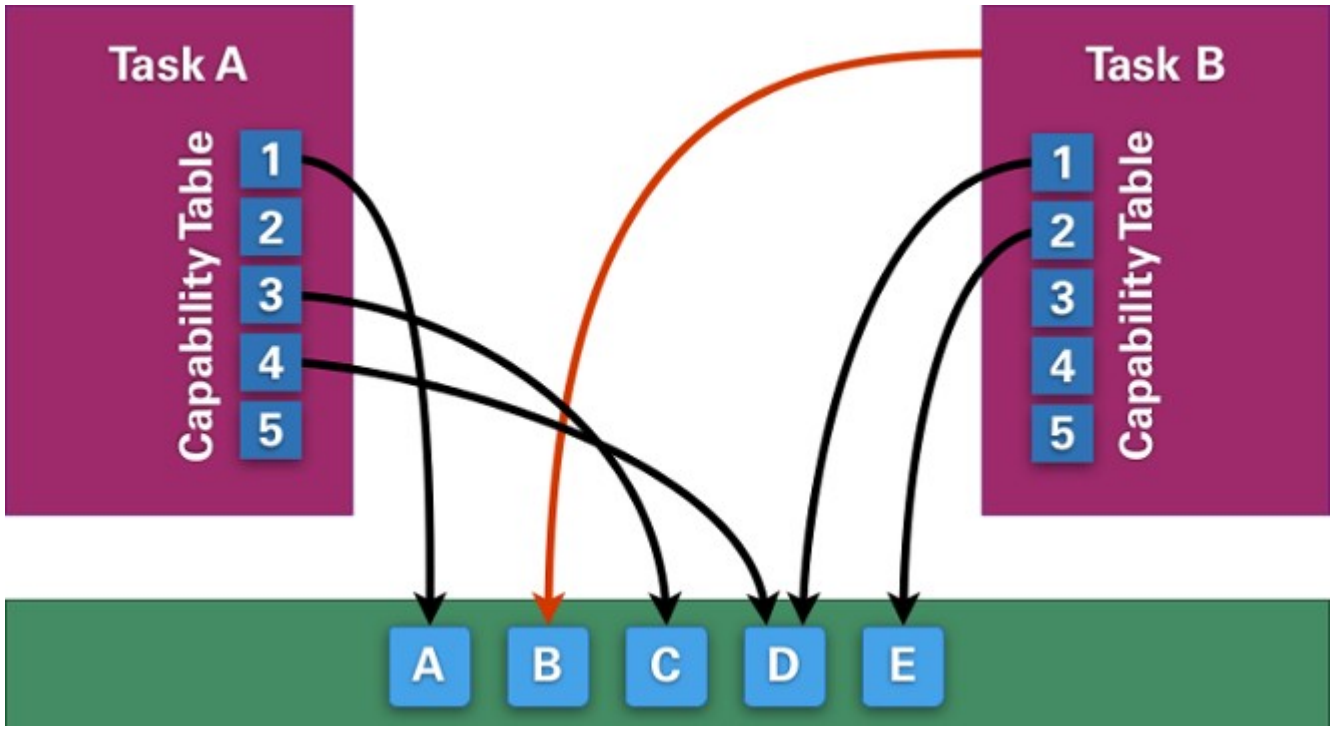
- Need to control who can send data to whom
  - Security and isolation
  - Access to resources
- Approaches
  - IPC redirection/introspection
  - Central vs. Distributed policy and mechanism
  - ACL-based vs. capability-based



Abstractions: Capabilities

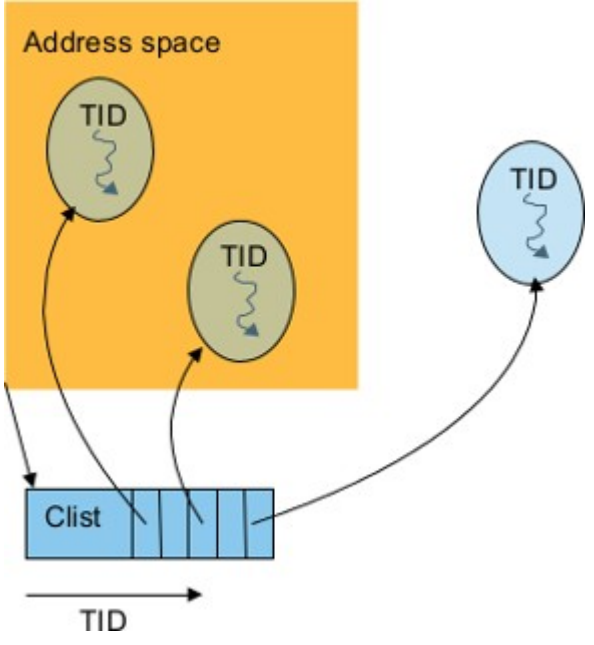
(in recent L4 implementations)

- actors in the system are objects
  - objects have local state and behavior
- capabilities are references to objects
  - any object interaction requires a capability
  - unseparable and unforgeable combination of reference and access right



Abstractions: Capabilities

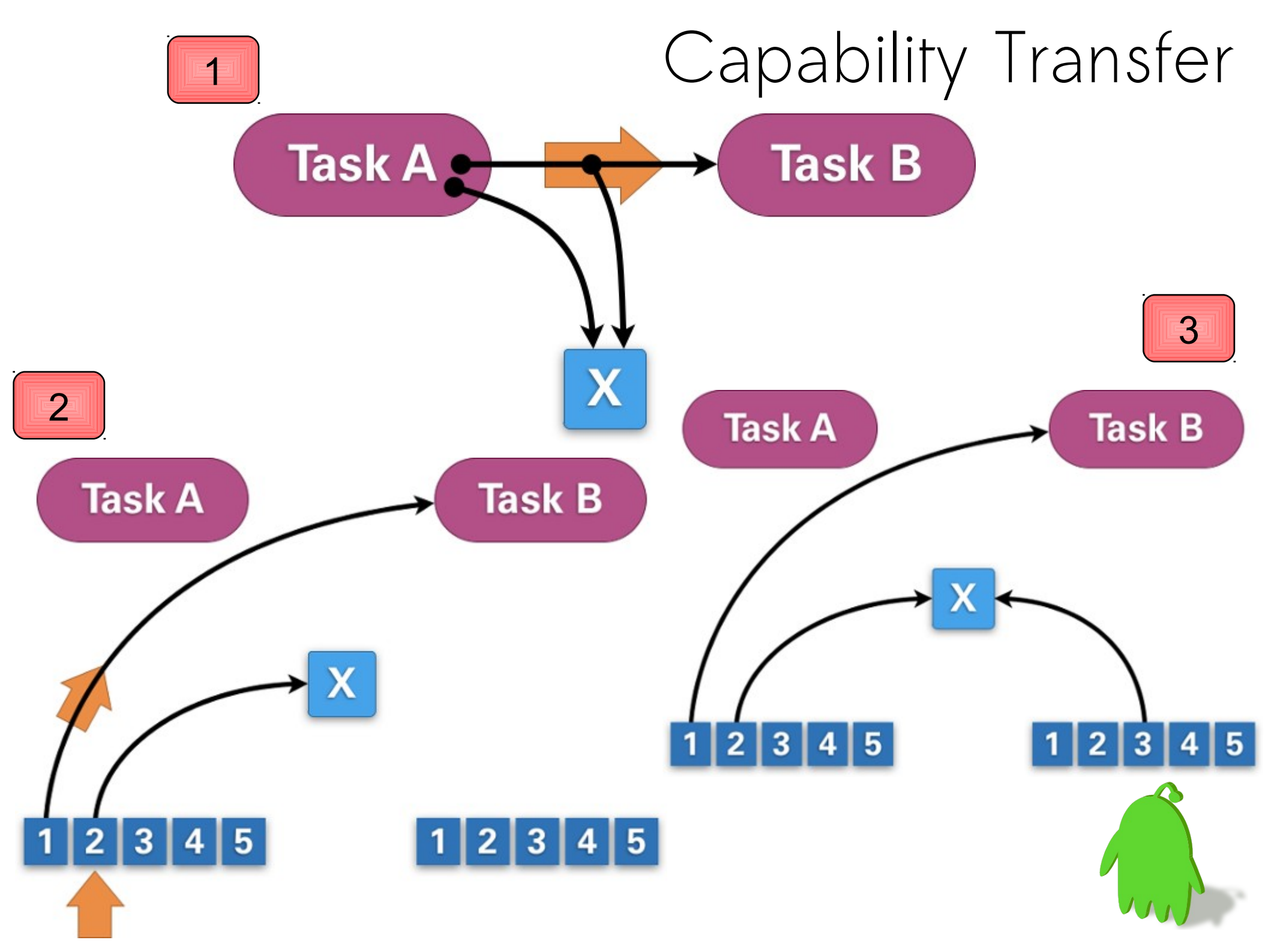
- Capabilities reference threads
  - actual cap word (TID) is index into per-address-space capability list (Clist)
- Capability conveys privilege
  - Right to send message to thread
  - May also convey rights to other operations on thread
- Capabilities are local names for global resources



Abstractions: Capabilities

- If a thread has access to a capability, it canmap this capability to another thread
- Mapping / not mapping of capabilities used for implementing access control
- Abstraction for mapping: flexpage
- Flexpages describe mapping
  - location and size of resource
  - receiver's rights (read-only, mappable)
  - type (memory, I/O, communication capability)

Capability Transfer

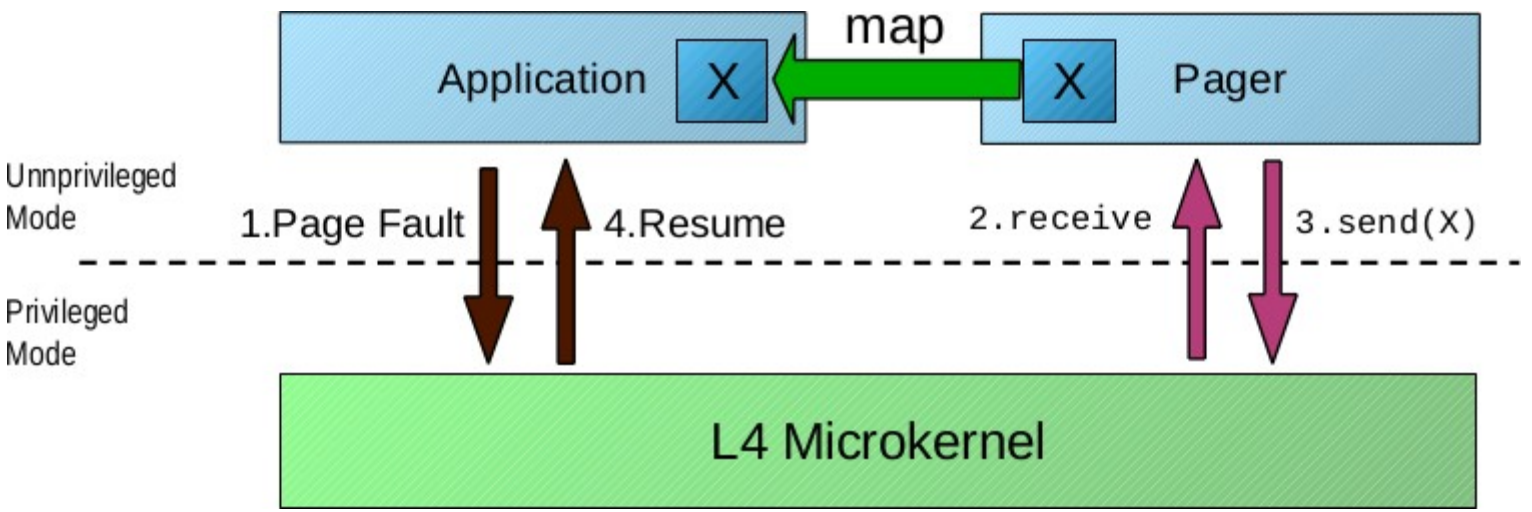


Mechanism: Exception Handling

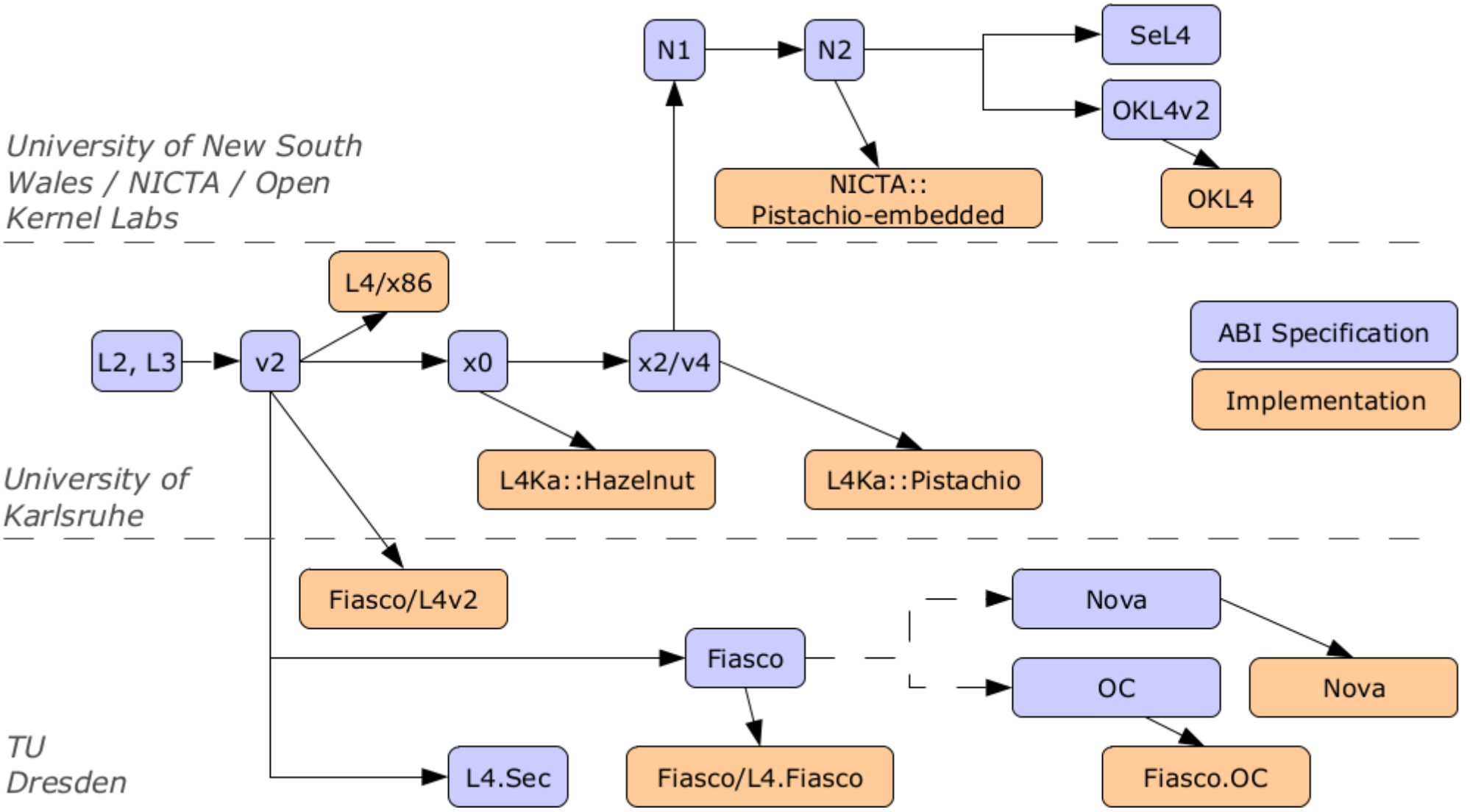
- Interrupts
  - Modelled as hardware “thread” sending messages
  - Received by registered (user-level) interrupt-handler thread
  - Interrupt acknowledged by handler via syscall (optionally waiting for next)
  - Timer interrupt handled in-kernel
- Page Faults
  - Kernel fakes IPC message from faulting thread to its pager
  - Pager requests root task to set up a mapping
  - Pager replies to faulting client, message intercepted by kernel
- Other Exceptions
  - Kernel fakes IPC message from exceptor thread to its exception handler
  - Exception handler may reply with message specifying new IP, SP
  - Can be signal handler, emulation code, stub for IPCing to server, ...

Page Fault and Pager

- 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



L4 Family (incomplete)



Source: Microkernel-based Operating Systems – Introduction, Carsten Weinhold, TU Dresden (2012)

L4 API vs. ABI

- L4 project was originally established by Jochen Liedtke in the 1990s and is actively researched by the L4Ka team at the University of Karlsruhe in collaboration with NICTA / University of New South Wales and the Dresden University of Technology.
- L4 is defined by a platform-independent1 API and a platform-dependent ABI

Background: Liedtke's involvement

- Jochen Liedtke and his colleagues continued research on L4 and microkernel based systems in general.
- In 1996, Liedtke started to work at IBM's Thomas J. Watson Research Center.
- In 1999, Liedtke took over the Systems Architecture Group at the University of Karlsruhe, Germany
  - L4Ka::Hazelnut

Commercial L4: from NICTA to OKLabs

- L4::Pistachio microkernel was originally developed at Karlsruhe University. NICTA had ported it to a number of architectures, including ARM, had optimized it for use in resource-constrained embedded systems.
- In 2004, Qualcomm engaged NICTA in a consulting arrangement to deploy L4 on Qualcomm's wireless communication chips.
- The engagement with Qualcomm grew to a volume where it was too significant a development/engineering effort to be done inside the research organization.
  - Commercized!

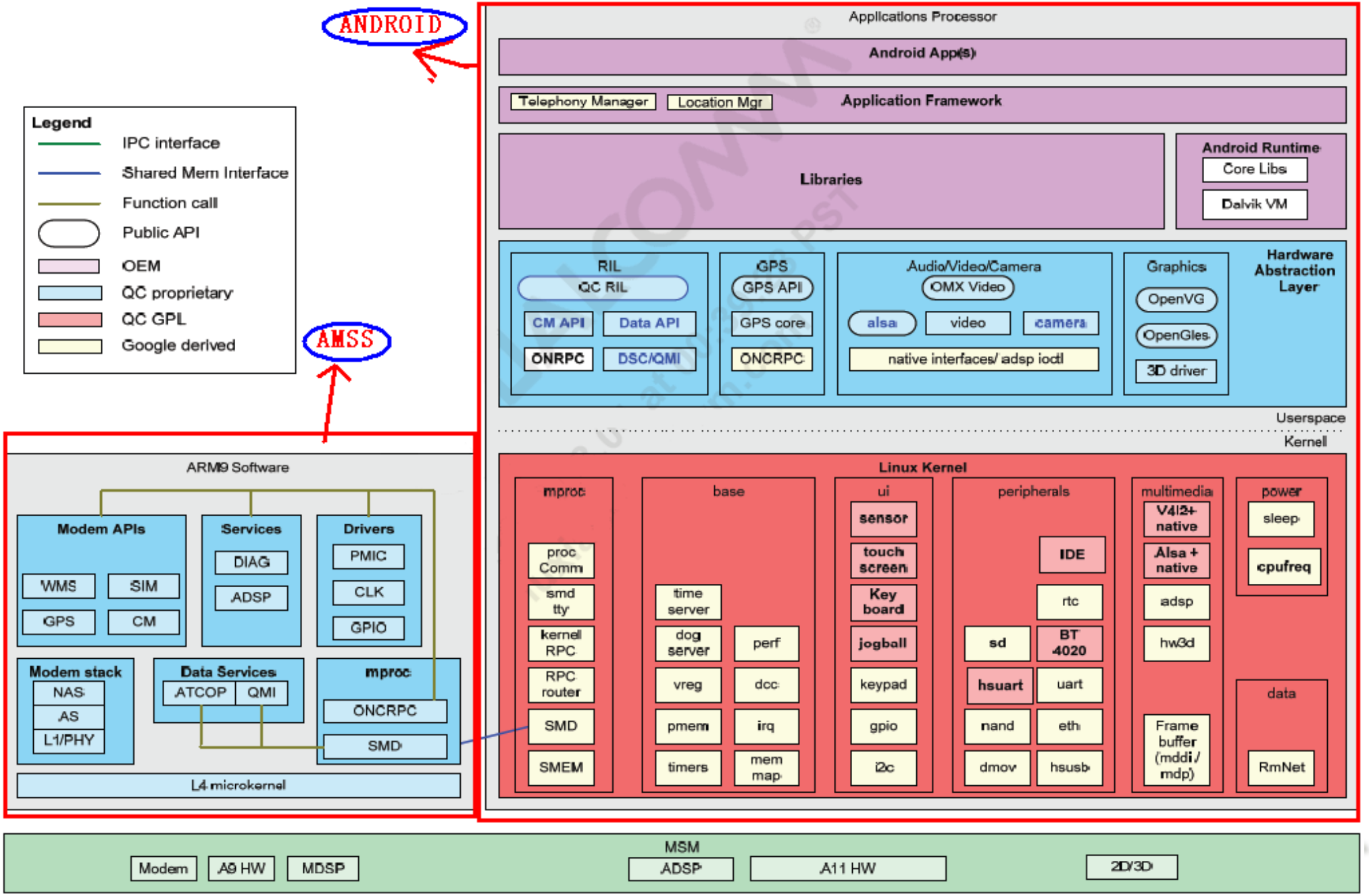
Source: http://microkerneldude.wordpress.com/2012/10/02/giving-it-away-part-2-on-microkernels-and-the-national-interes/

AMSS

- AMSS = Advanced Mobile Subscriber Software, developed by Qualcomm
  - RTOS runs atop the baseband processor.
  - based on L4 microkernel
  - Real world verified commercial microkernel
- Outside of some fixed functionality and off-die RF, the baseband really is just a CPU, a big DSP or two, and collection of management tasks that run as software - GPS, GLONASS, talking with the USIM, being compliant with the pages and pages of 3GPP specifications, etc.

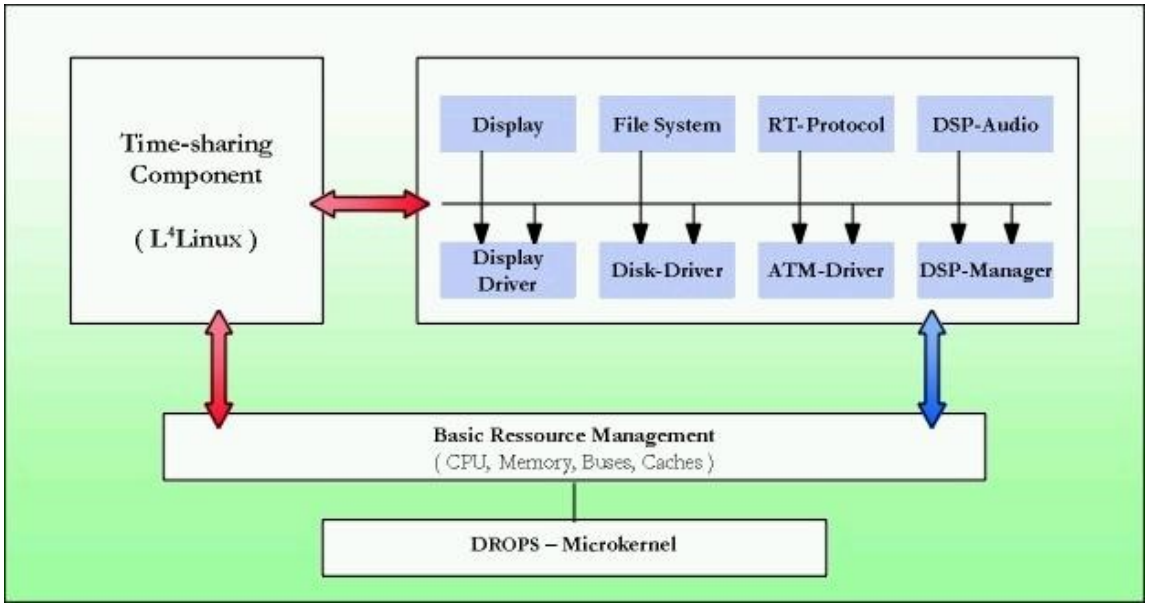
AMSS

Source:
http://blog.csdn.net/thinkandchange/article/details/8090527



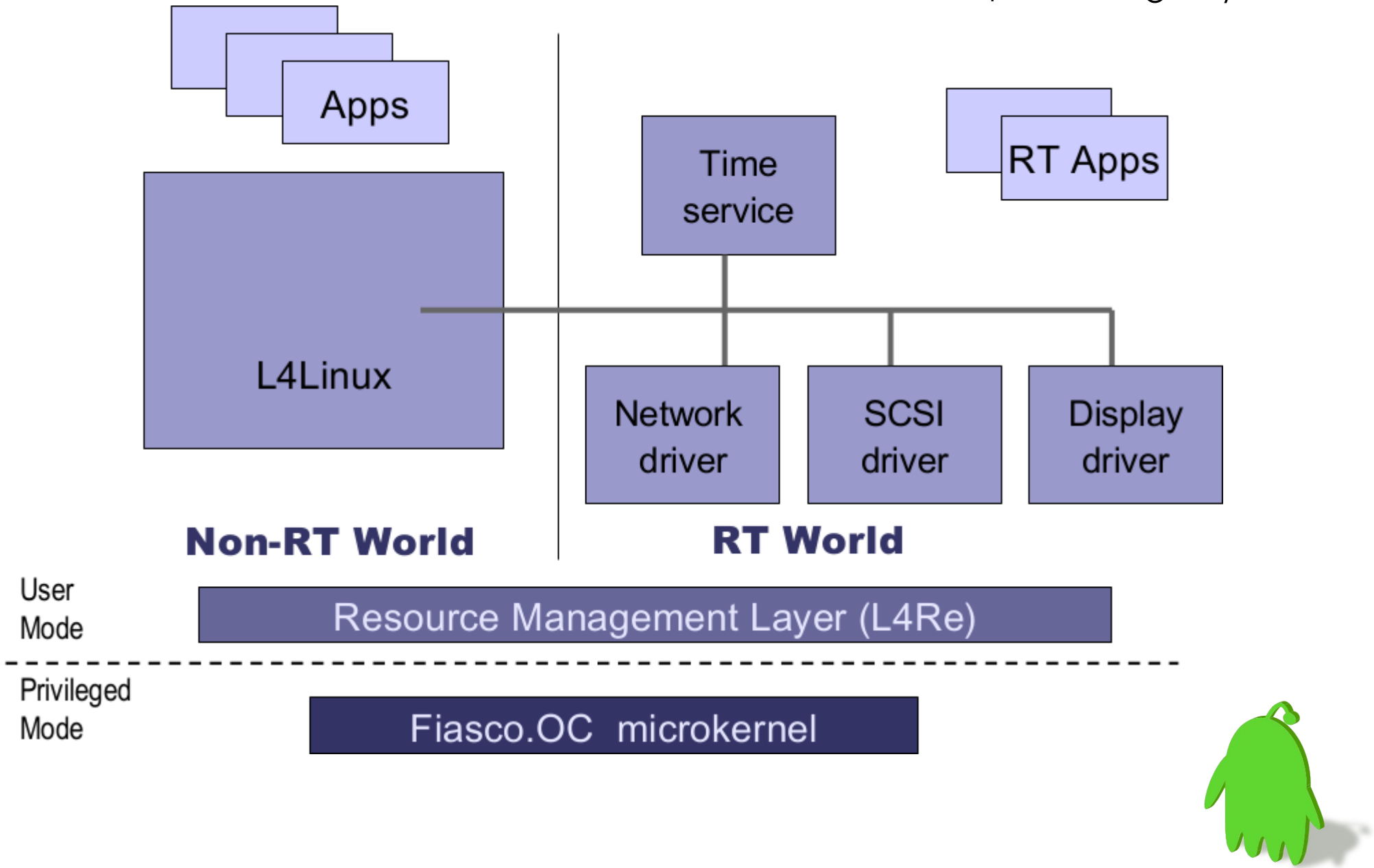
L4 Family: Fiasco

- Design by TU Dresden
- Preemptible real-time kernel
- Used for DROPS project (GPL)
  - Dresden Real-Time Operating Systems Project
  - Find design techniques for the construction of distributed real time operating systems
  - Every component guarantees a certain level of service



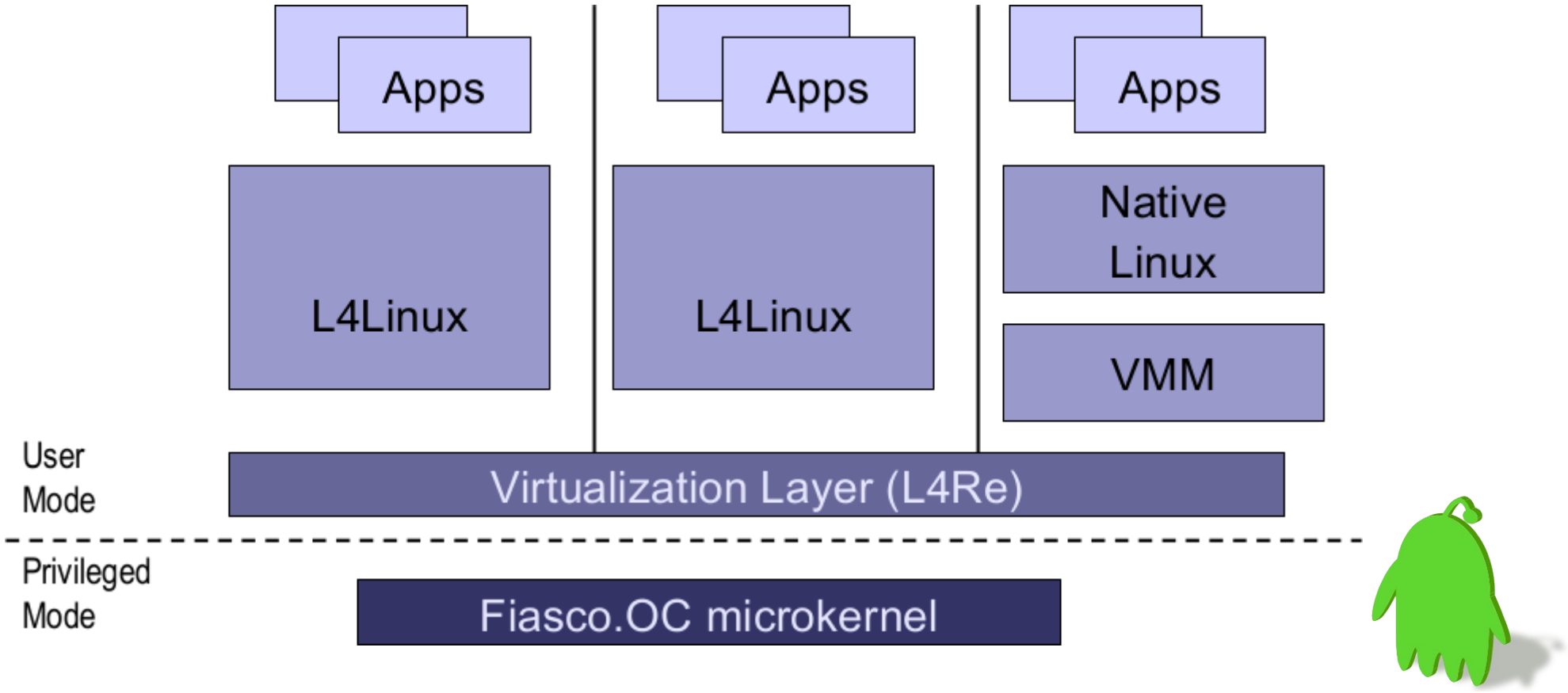
DROPS

(Dresden Real-Time Operating System)



Virtual Machines based on L4Re

- Isolate not only processes, but also complete
- Operating Systems (compartments)
  - "Server consolidation"



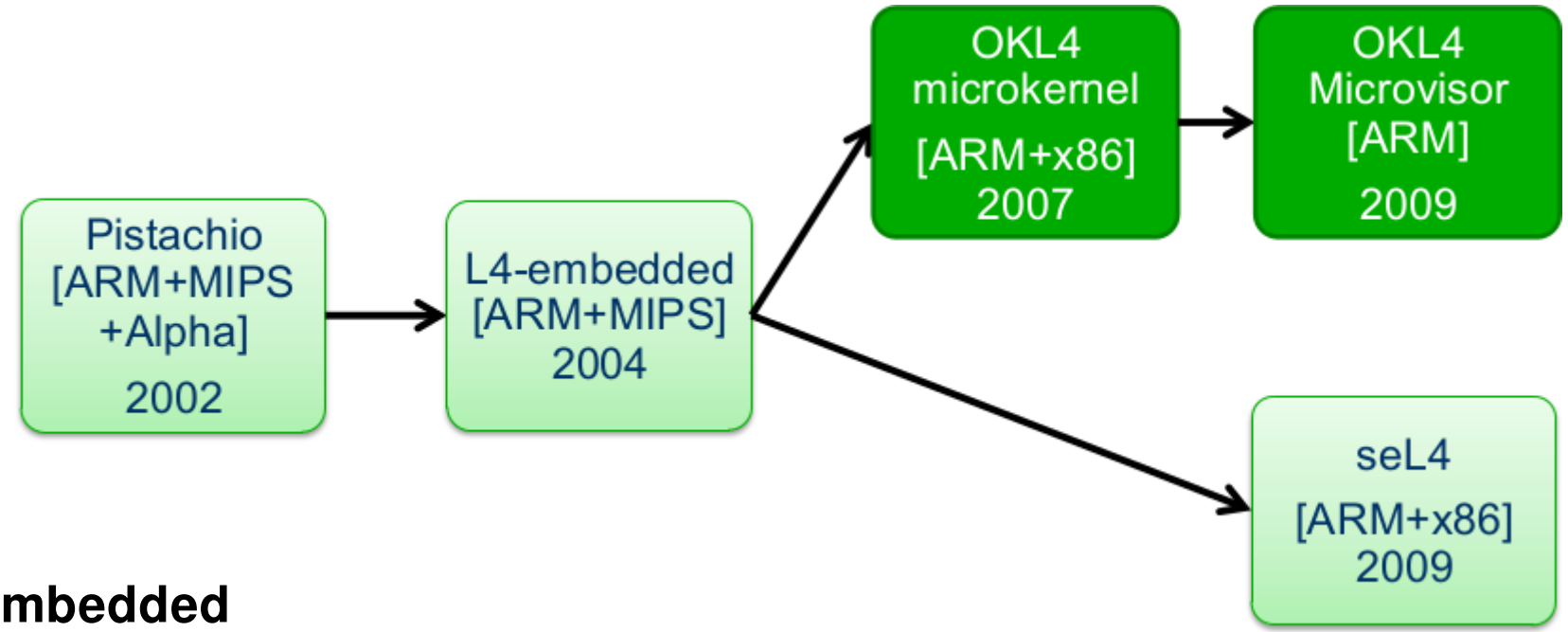
L4 Family: Pistachio

- Pistachio was designed by L4Ka team and NICTA EROPS Group
  - L4 API V X.2
- NICTA::Pistachio-embedded
  - Designed by UNSW and NICTA EROPS Group
  - Based on L4Ka::Pistachio, designed for embedded
  - keep almost all system calls short

L4 Family: OKL4

- Design by OKLab (Open Kernel Labs)
  - old source available
- OKLab spun out from NICTA
- Based on NICTA::Pistachio-embedded
- Commercial deployment
  - Adopted by Qualcomm for CDMA chipsets

L4 Implementations by UNSW/NICTA



L4-embedded

- Fast context-switching on ARMv5
  - context switching without cache flush on virtually-addressed caches
  - 55-cycle IPC on Xscale
  - virtualized Linux faster than native
- Event-based kernel (single kernel stack)
- Removed IPC timeouts, “long” IPC
- Introduced asynchronous notifications

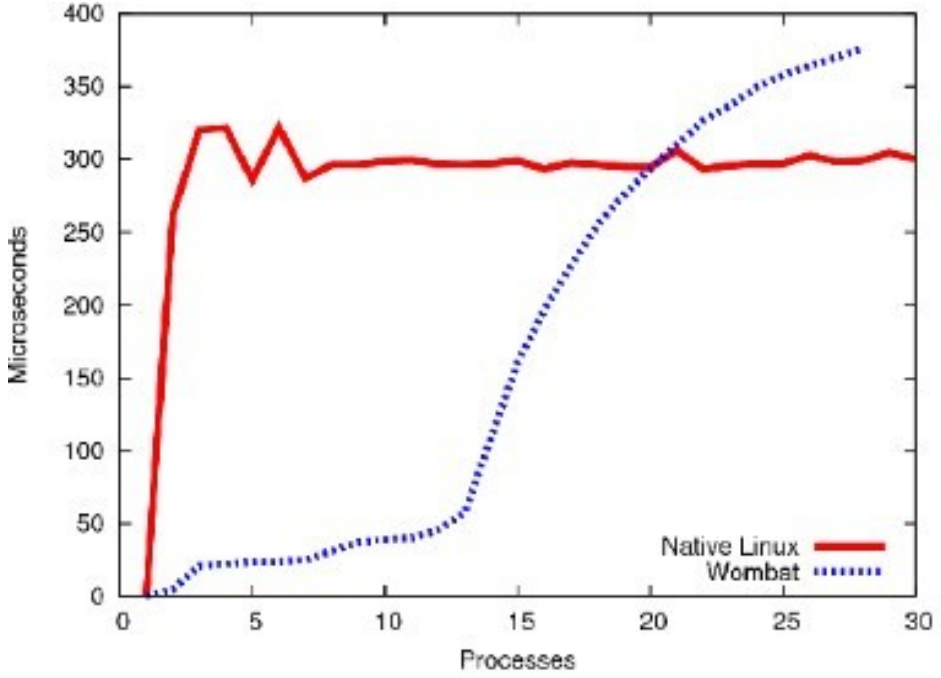
OKL4

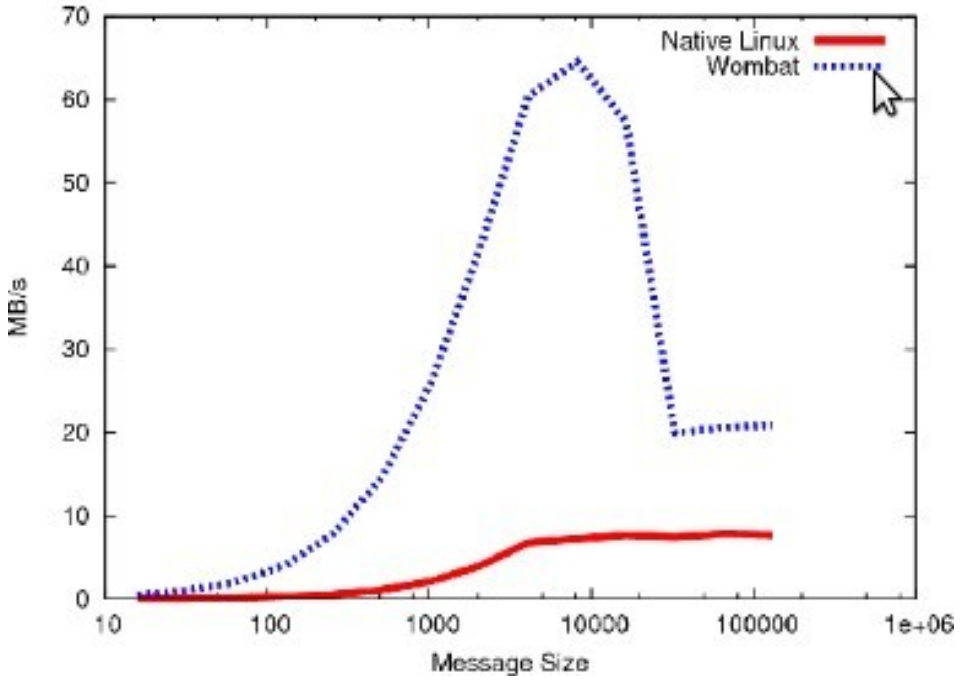
- Dumped recursive address-space model
  - reduced kernel complexity
  - First L4 kernel with capability-based access control

OKL4 Microvisor

- Removed synchronous IPC
- Removed kernel-scheduled threads

L4 Family: OKL4





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

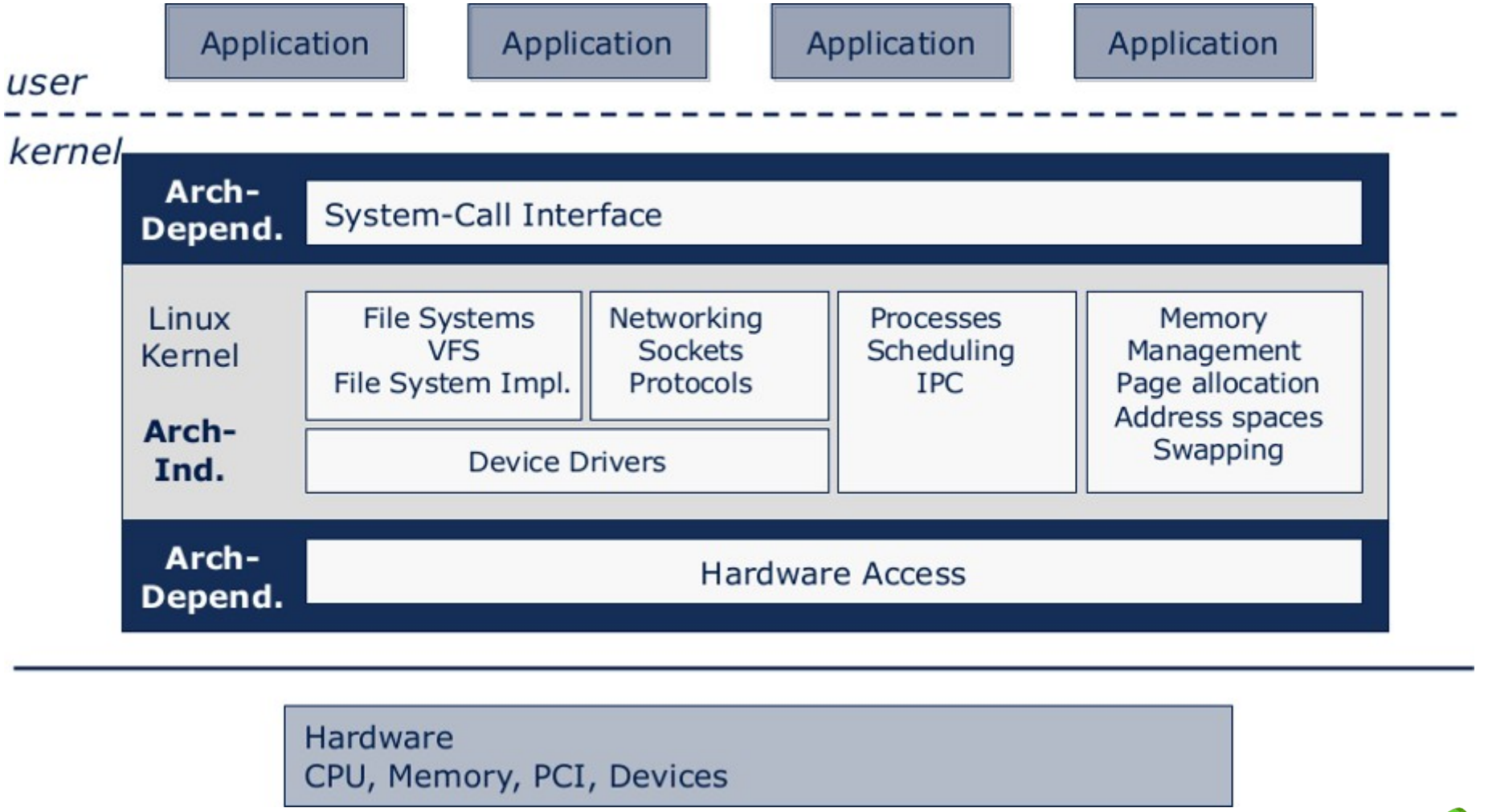
Bring Linux to L4

- Some portions of Linux are architecture dependent:
  - Interrupts
  - Low-level device drivers (DMA)
  - Methods for interaction with use processes.
  - Switching between kernel contexts
  - Copyin/copyout between kernel and user space
  - Mapping/unmapping for address space
  - System calls

L4Linux

- Is a paravirtualized Linux first presented at SOSP'97 running on the original L4 kernel.
  - L4Linux predates the x86 virtualization hype
  - L4Linux 2.4 first version to run on L4Env
  - L4Linux 2.6 uses Fiasco.OC's paravirtualization features
- Current status
  - based on Linux 3.8
  - x86 and ARM support
  - SMP

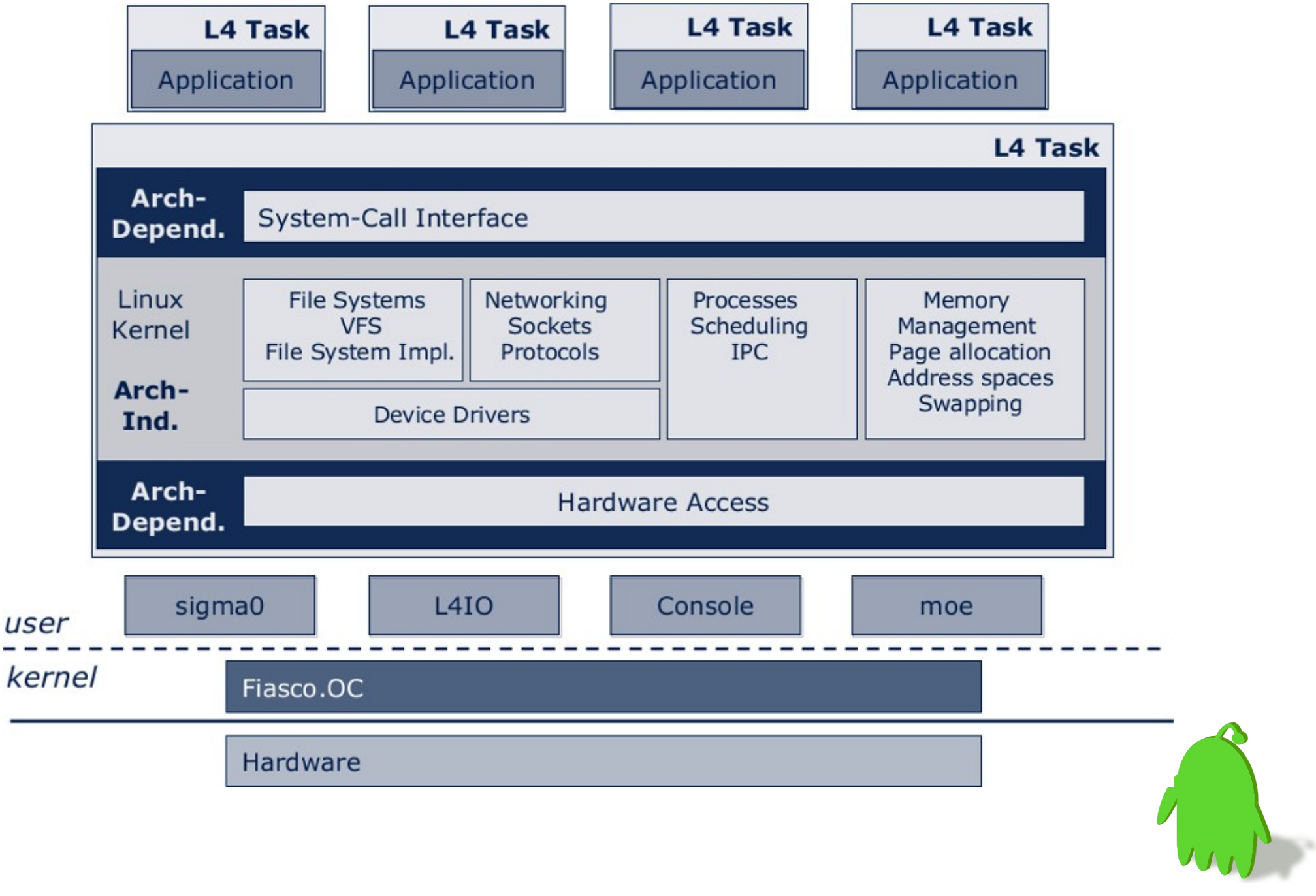
L4Linux Architecture



L4Linux

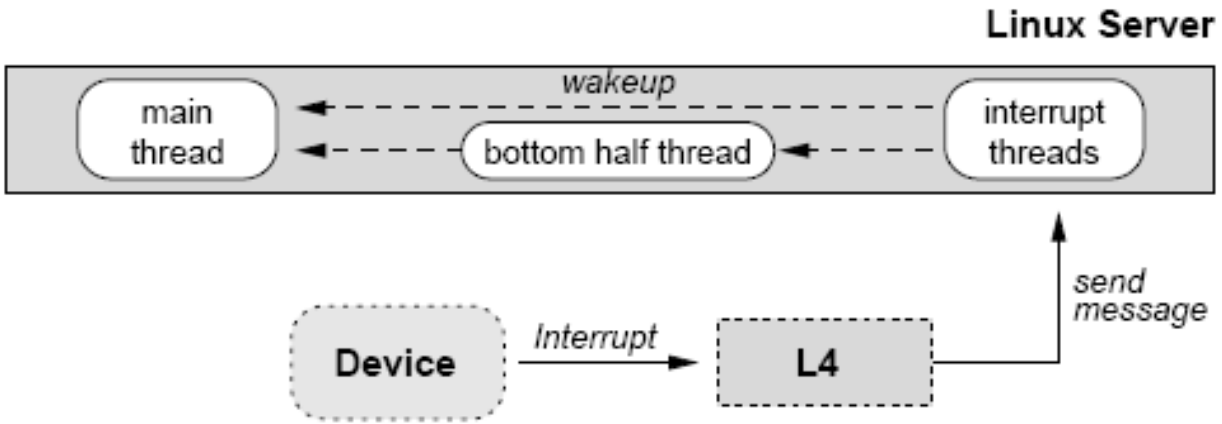
- Linux kernel as L4 user service
  - Runs as an L4 thread in a single L4 address space
  - Creates L4 threads for its user processes
  - Maps parts of its address space to user process threads (using L4 primitives)
  - Acts as pager thread for its user threads
  - Has its own logical page table
  -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

L4Linux


L4 Task
Application
L4 Task
Application
L4 Task
Application
L4 Task
Application
L4 Task
Arch-Depend.
System-Call Interface
Linux Kernel
Arch-Ind.
File Systems
VFS
File System Impl.
Networking
Sockets
Protocols
Processes
Scheduling
IPC
Memory
Management
Page allocation
Address spaces
Swapping
Device Drivers
Arch-Depend.
Hardware Access
sigma0
L4IO
Console
moe
user
kernel
Fiasco.OC
Hardware

L4Linux: Interrupt Handling

- All interrupt handlers are mapped to messages.
- The Linux server contains threads that do nothing but wait for interrupt messages.
- Interrupt threads have a higher priority than the main thread.

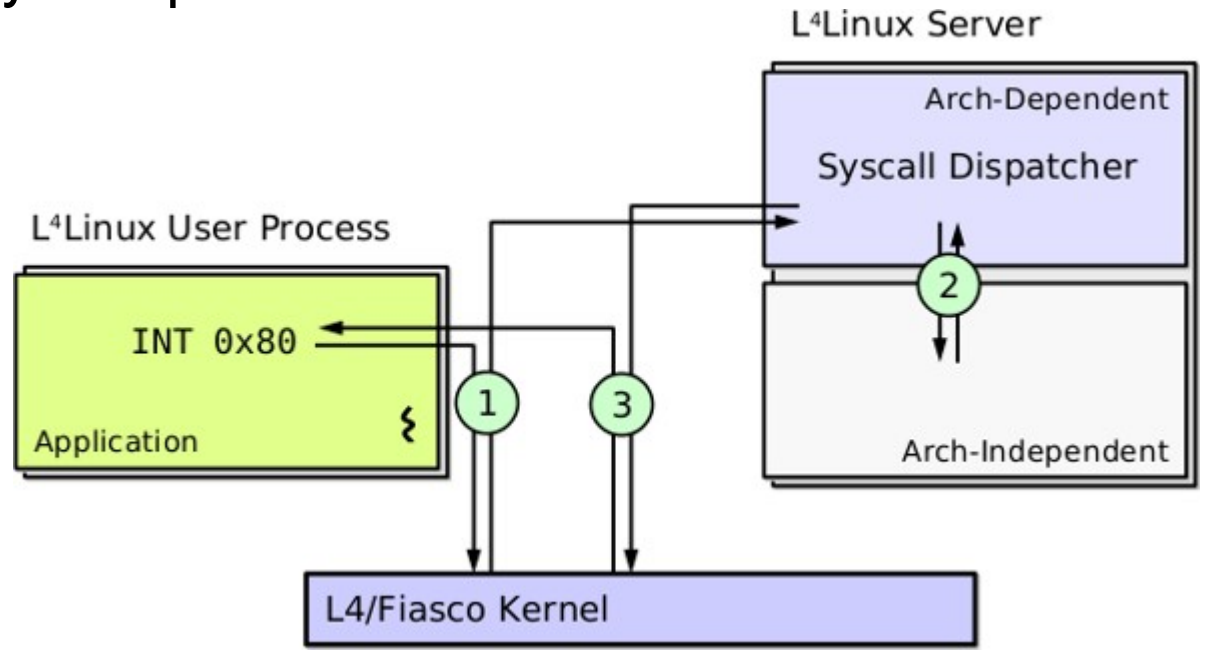


L4Linux: User Process

- Each different user process is implemented as a different L4 task: Has it's own address space and threads.
- Linux Server is the pager for these processes. Any fault by the user-level processes is sent by RPC from the L4 kernel to the Server.

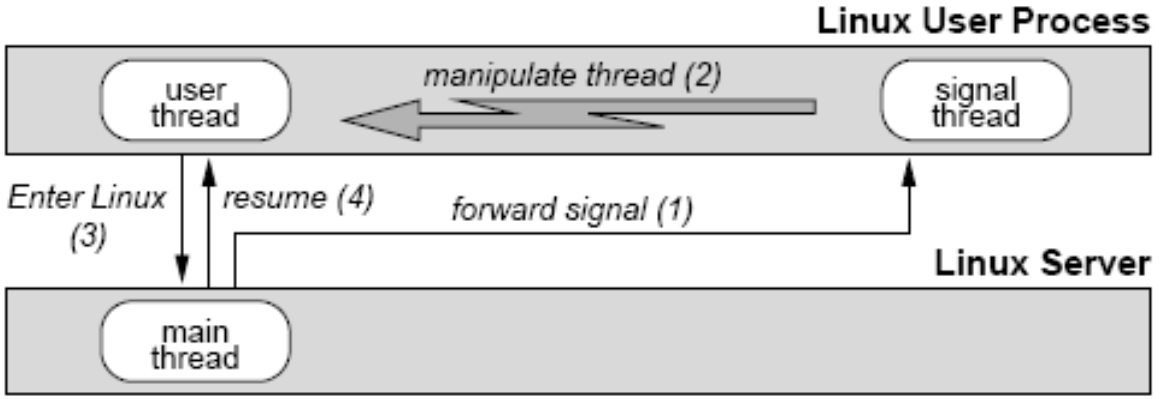
L4Linux: System Calls

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



L4Linux: Signal

- Each user-level process has an additional thread for signal handling.
- Main server thread cannot directly manipulate user process stack, so it sends a message for the signal handling thread, telling the user thread to save it's state and enter Linux



L4Linux: Scheduling

- All thread scheduling is down by the L4 kernel
- Linux server's schedule() routine is only used for multiplexing it's single thread.
- After each system call, if no other system call is pending, it simply resumes the user process thread and sleeps.

TLB

- A Translation Look-aside Buffer (TLB) caches page table lookups
- On context switch, TLB needs to be flushed
- A tagged TLB tags each entry with an address space label, avoiding flushes
- Pentium-class CPU can emulate a tagged TLB for small address spaces

Small Memory Space

- In order to reduce TLB conflicts, L4Linux has a special library to customize code and data for communicating with the Linux Server
- The emulation library and signal thread are mapped close to the application, instead of default high-memory area.

Benchmark Matrix

- Compared the following systems
  - Native Linux
  - L4Linux

    8.3% slower; Only 6.8% slower at maximum load.
  - MkLinux (in-kernel)
    - Linux ported to run inside Mach
    - 29% slower
  - MkLinux (user)
    - Linux ported to run as a user process on top of Mach
    - 49% slower

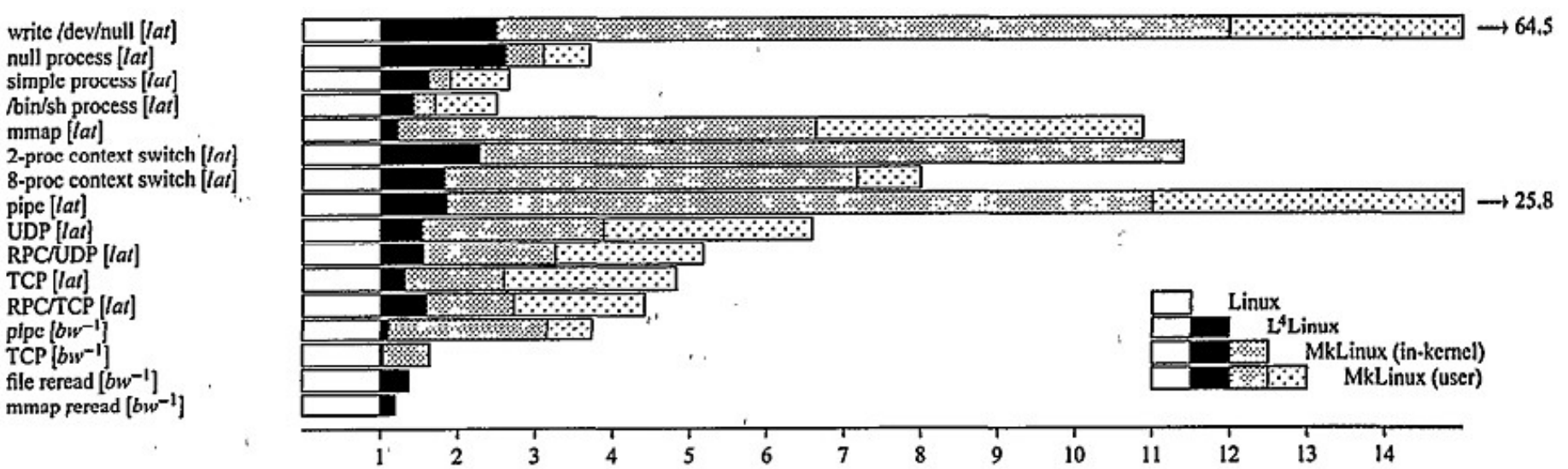


Figure 6: *Imbench results, normalized to native Linux.* These are presented as slowdowns: a shorter bar is a better result. [*lat*] is a latency measurement, [bw^{-1}] the inverse of a bandwidth one. Hardware is a 133 MHz Pentium.

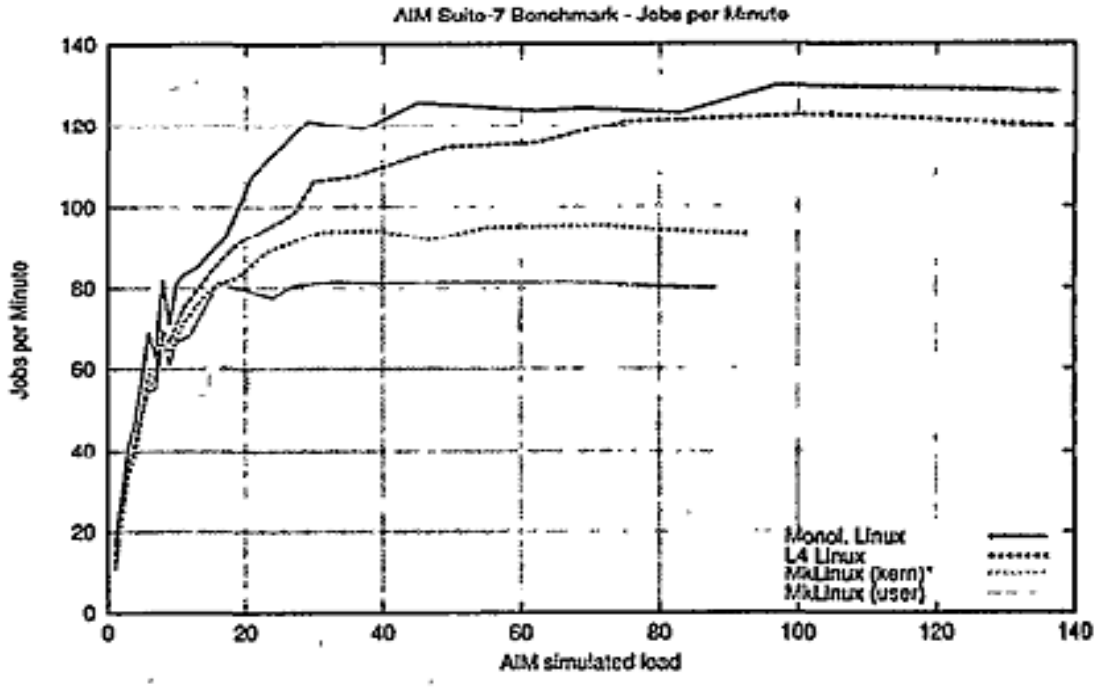
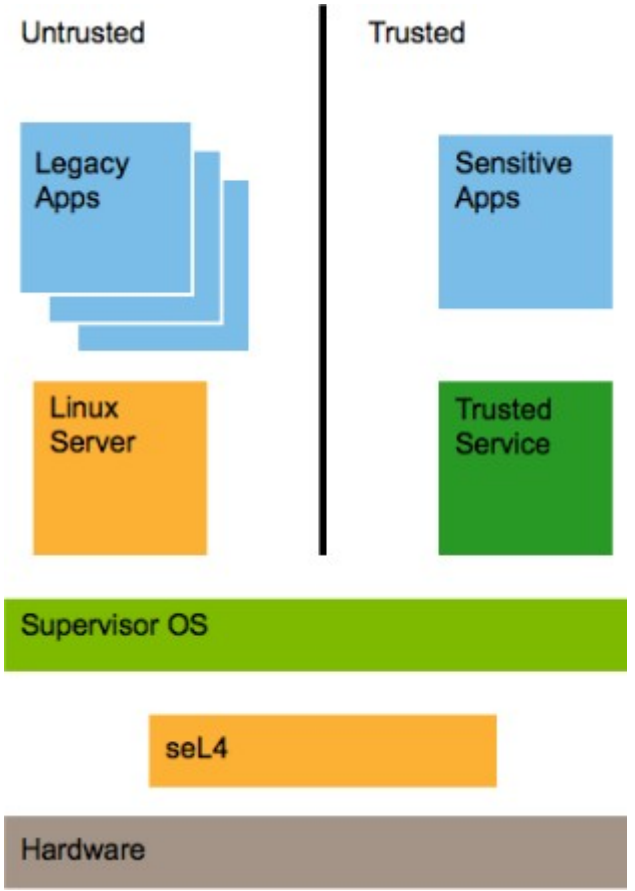


Figure 9: *AIM Multiuser Benchmark Suite VII.* Jobs completed per minute depending on AIM load units. (133 MHz Pentium)

Microkernel: 3rd Generation

Contrasting Readings

- seL4: Formal Verification of an Operating-System Kernel
  - Gerwin Klein et alia
  - SOSP'09 Best Paper
- Goals
  - General-purpose
  - Formal verification
    - Functional correctness
    - Security/safety properties
  - High performance



Problems in 2nd Generations

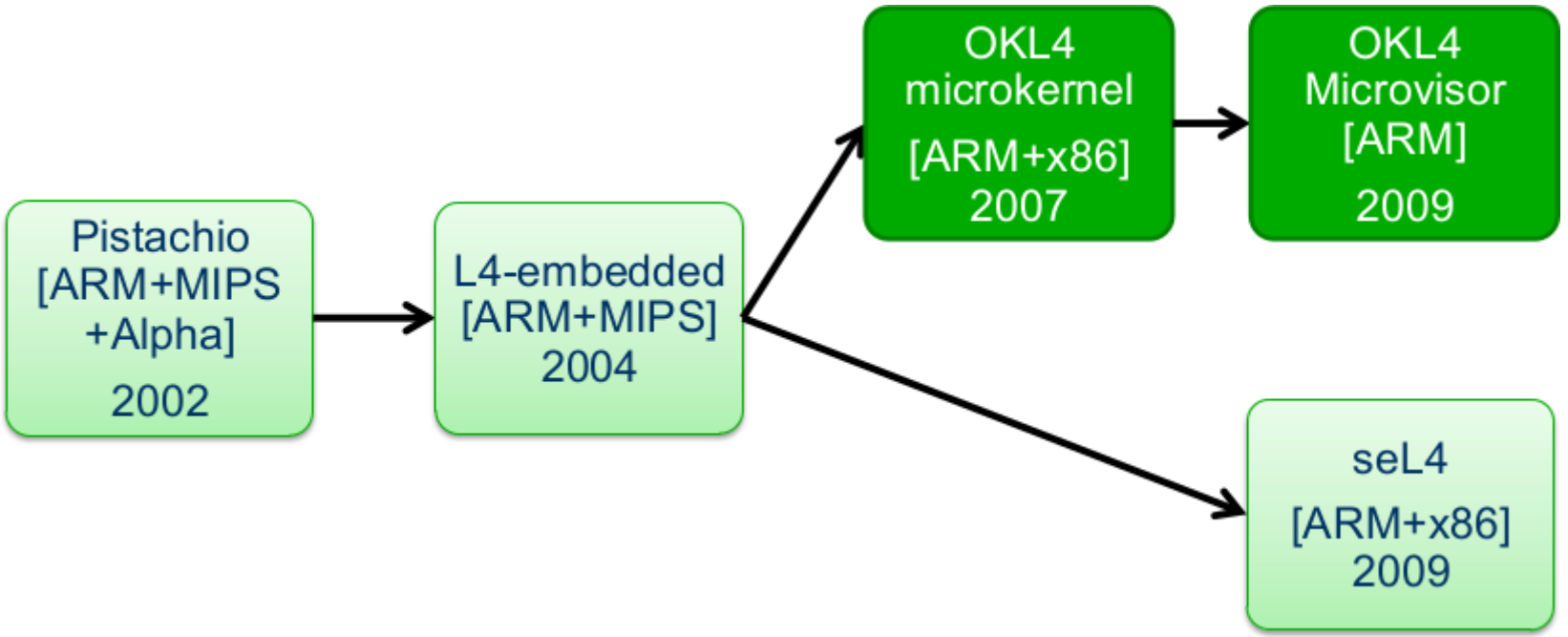
- microkernel needs memory for its abstractions
  - tasks: page tables
  - threads: kernel-TCB
  - capability tables
  - IPC wait queues
  - mapping database
  - kernel memory is limited
  - opens the possibility of DoS attacks

Ideas to solve 2nd's Problems

- memory management policy should not be in the kernel
- account all memory to the application it is
- needed for (directly or indirectly)
- kernel provides memory control mechanism
  - exception for bootstrapping:

    initial kernel memory is managed by kernel
- untyped memory in seL4 (3rd Generation)
  - all physical memory unused after bootstrap is represented by untyped memory capabilities

Moving from 2nd to 3rd Generation



OKL4

- Dumped recursive address-space model
  - reduced kernel complexity
  - First L4 kernel with capability-based access control

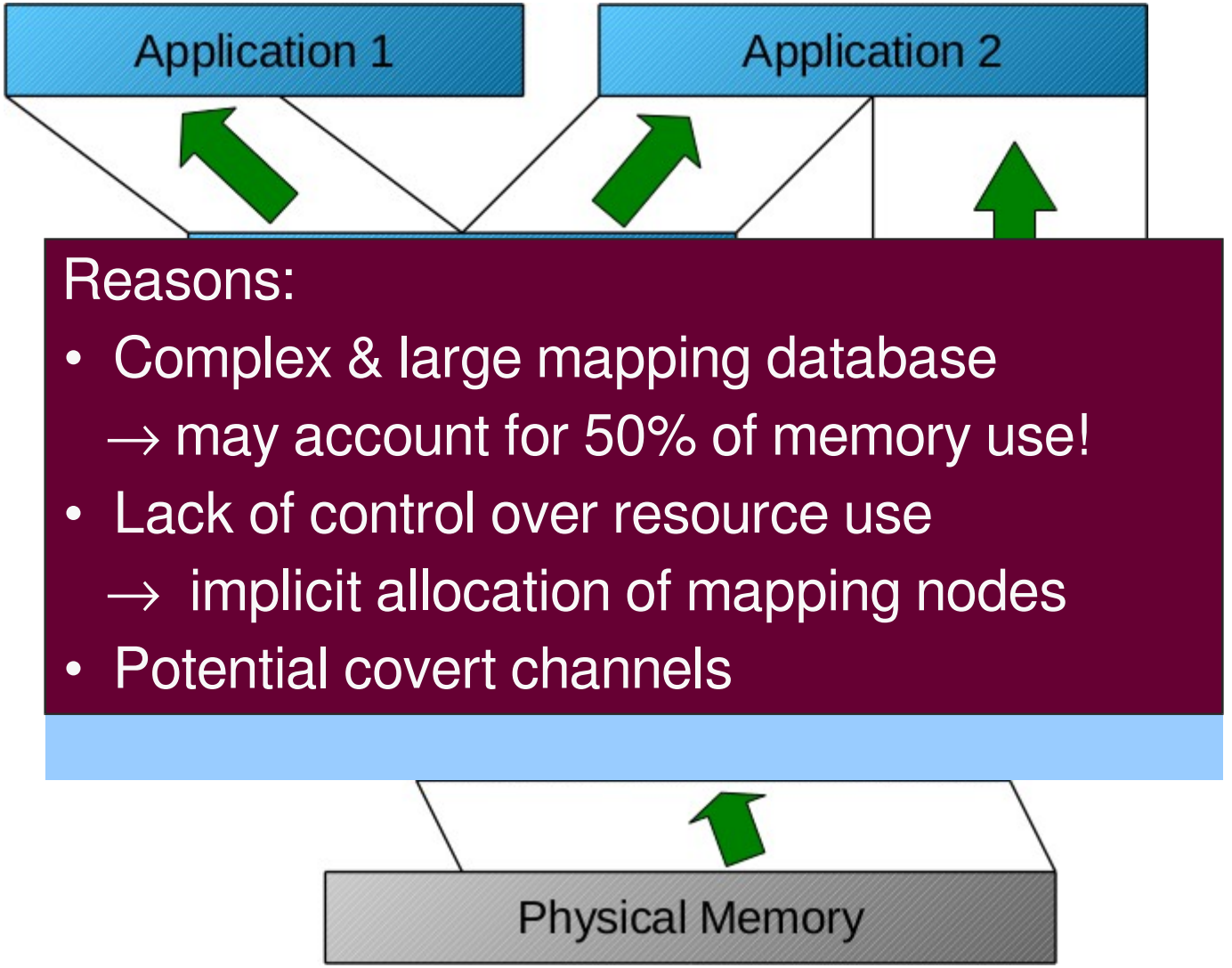
OKL4 Microvisor

- Removed synchronous IPC
- Removed kernel-scheduled threads

seL4

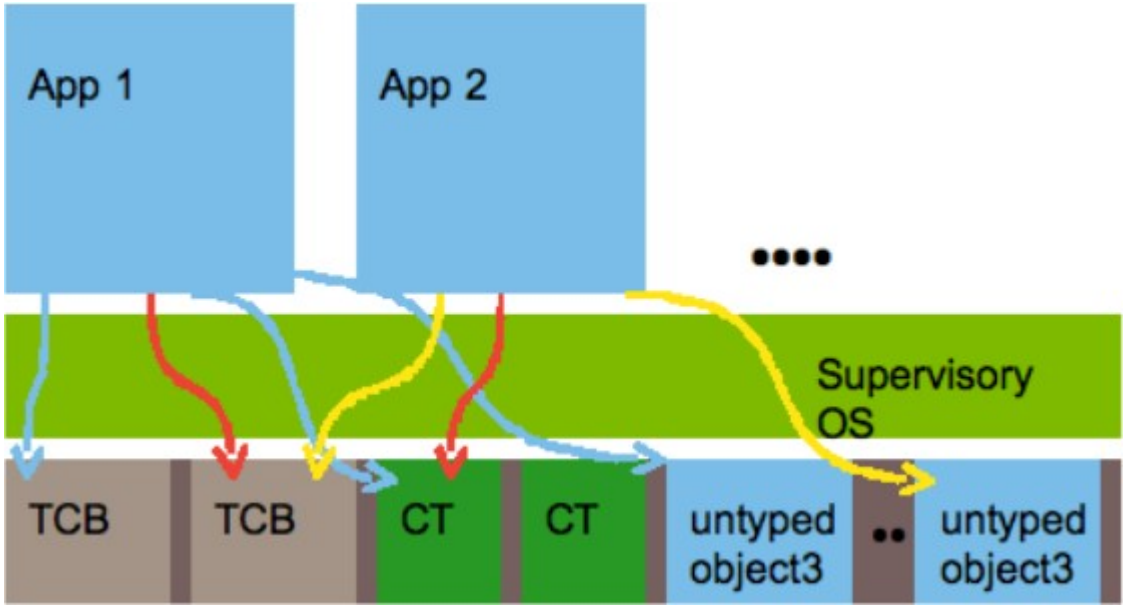
- All memory management at user level
  - no kernel heap!
- Formal proof of functional correctness
- Performance on par with fastest kernels
  - <200 cycle IPC on ARM11 without assembler fastpath

Why Recursive Address Spaces are removed?



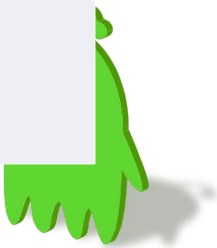
seL4 Novelty: Kernel Resource Management

- 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
  - Operate within delegated resources
  - No interference



Conceptual Difference with original L4

Original L4, Liedtke	seL4	OKL4 Microvisor
Abstractions	Yes, but abstractions are quite different	
Threads	thread	virtual CPU
Address Spaces	address space	virtual MMU
Synchronous IPC	sync IPC + async notify	virtual IRQ (async)
Rich msg struct	No	
Unique thread ID	No, has capabilities	
Virtual TCB array	No	
Per-thread kernel stack	No, kernel event	



Adopting Microkernels: Mobile Virtualization

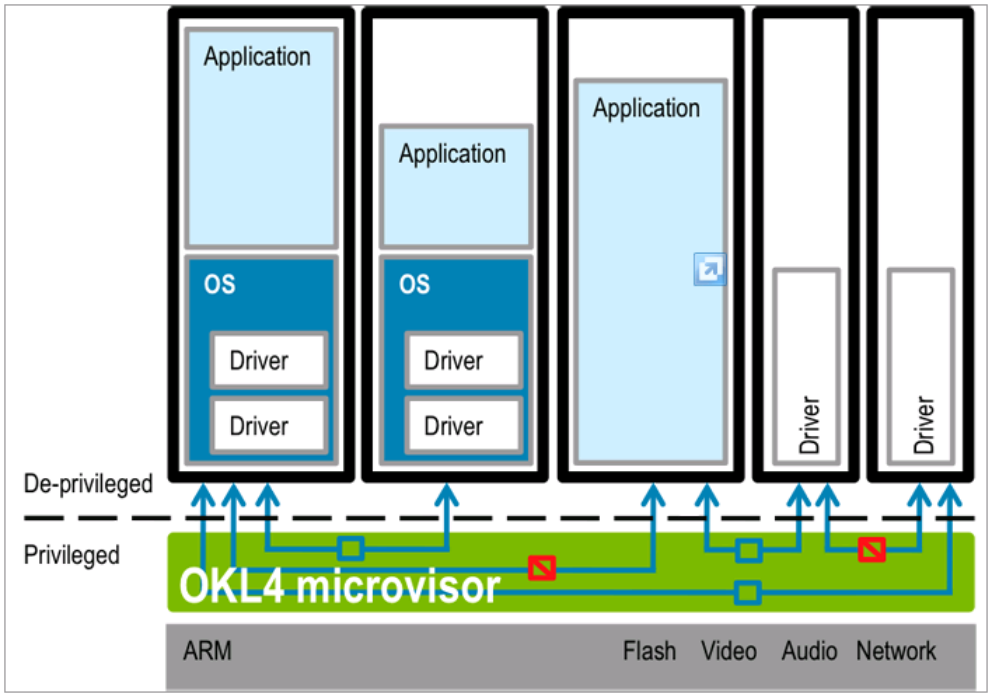
Mobile Virtualization

- Why mobile virtualization?
  - Current and next-generation smartphones increasingly resemble desktop computers in terms of computing power, memory, and storage
- OKL4 Microvisor
  - API optimized for low-overhead virtualization
  - Eliminated:
    - recursive address spaces
    - Synchronous IPC
    - Kernel-scheduled threads
  - API closely models hardware:
    - vCPU, vMMU, vIRQ + “channels” (FIFOs)
    - Capabilities for resource control

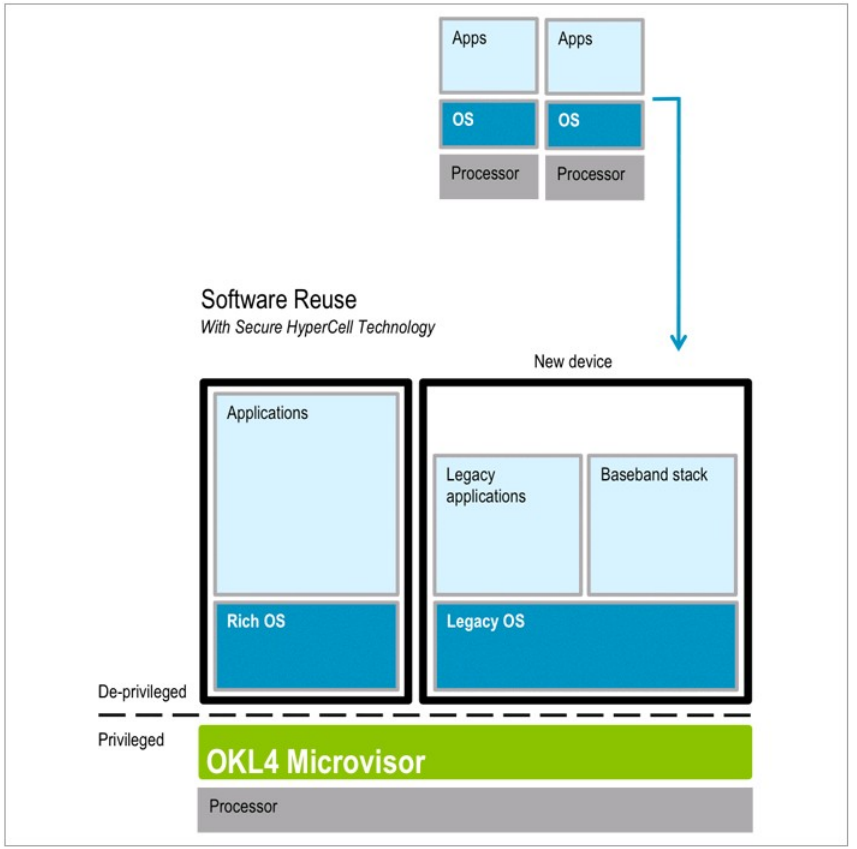
OKL4 Microvisor: Benefits and Capabilities

- Virtualization
- Resource management
- Lightweight components
- Real-time capability and low-performance overhead
- Small memory footprint
- High-performance IPC between secure cells
- Minimal trusted computing base(TCB)
- Single core and multicore support
- Ready-to-integrate guest OS support with available OS support packages (paravirtualizing guest Oses)

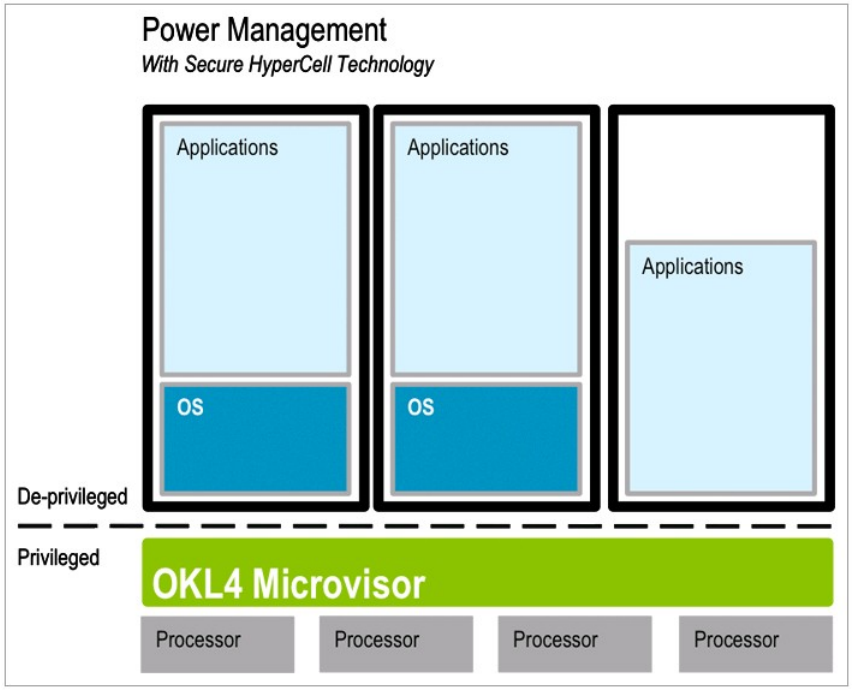
Use Cases



Each secure cell in the system offers isolation from software in other cells



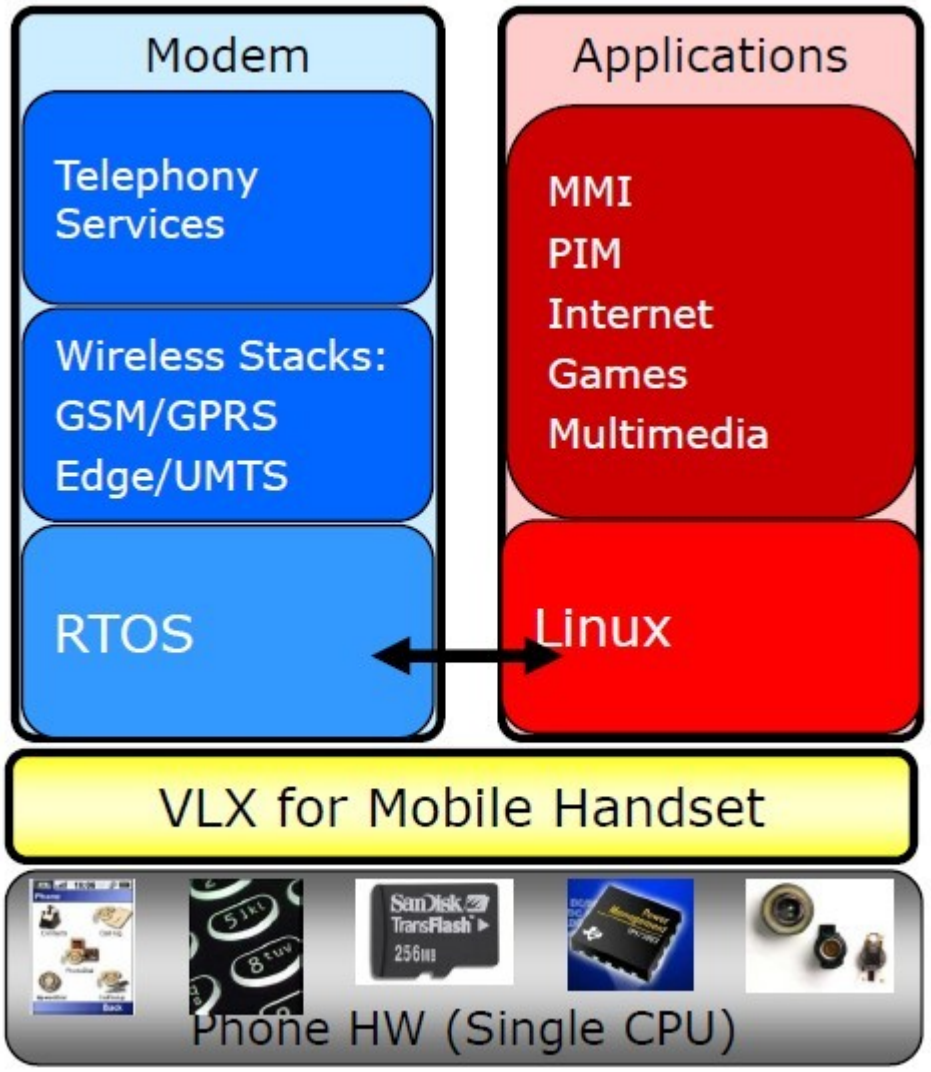
Existing software components can be reused in new designs



Microvisor tames the complexity of dispatching multi-OS workloads across multiple physical CPUs

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 Re-usability of legacy code/applications
  - Reduces the system development time
- Instrumentation, Automation
  - Run RTOS for Measurement and analysis
  - Run a GPOS for Graphical Interface



Reference

- Wikipedia - http://en.wikipedia.org/wiki/L4_microkernel
- Microkernels, Arun Krishnamurthy, University of Central Florida
- Microkernel-based Operating Systems – Introduction, Carsten Weinhold, TU Dresden
- Threads, Michael Roitzsch, TU Dresden
- Virtualization, Julian Stecklina, TU Dresden

http://0xlab.org