Micro-Kernels

Pr. Olivier Gruber

Université Joseph Fourier

Laboratoire d'Informatique de Grenoble

Olivier.Gruber@imag.fr

Acknowledgements and References

- This lecture includes material courtesy of:
  - Dr. Renaud Lachaize, UJF
  - Prof. Gernot Heiser, UNSW

- References:
  - Research publications cited in the slides
- Reference Books:
  - A. Tanenbaum, Modern Operating Systems (3rd Edition), Prentice Hall, 2007
  - A. Silberschatz et al, Operating System Concepts (9th Edition), Wiley, 2013

Traditional Kernels

- Traditional kernels
  - Windows or Linux
  - Kernel is a single large binary, executing in kernel mode
  - Usually load/unload modules such as drivers or file systems
- Pros
  - Considered trusted – have been around and used for a long time
  - Performance – tight integration, kernel mode
  - Extensibility – if you can dream it, you can implement it

Traditional Kernels

- Cons
  - Large code base
    - Millions of lines of code, a large part is driver code
    - Linux: 70% of the code is driver code
  - Hard to understand, maintain, tune, or shrink
    - Complex and therefore buggy (5-10 bugs per KLOC)
  - Bugs often cause a complete system failure (panic – blue screen)
    - Linux: 70% of kernel failures are due to drivers
    - Windows: 85% of failures are attributed to drivers

Traditional Kernels

- Efforts to fix the driver problem
  - Many research efforts with tools to improve driver trustworthiness
  - Revisiting driver framework and security mechanisms
- Efforts to tackle complexity through modularity
  - Relatively small, self-contained components
  - Well-defined interfaces
- Does not work well with monolithic kernels
  - Because all kernel code executes in privileged mode
  - Interfaces cannot be enforced
  - Faults cannot be contained
  - And performance ultimately takes priority over modularity

Traditional Kernels

- Academic study of the Linux kernel evolutions (2002)
  - Looked at size and coupling of kernel “modules”
  - Coupling: interdependency via global variables
  - Result 1: Module size grows linearly with version number
  - Result 2: Interdependency grows exponentially with version number!
- Intel study on Linux kernel releases (2009)
  - 12% performance drop over the last ten releases
  - Quoting Linus Torvalds (creator of Linux):
    - “We are getting bloated and huge. Yes, it’s a problem.”
    - “Our instruction cache footprint is scary. [...] And whenever we add a new feature, it only gets worse.”
- What about other kernels?
  - There are no reasons to believe that other kernels are different

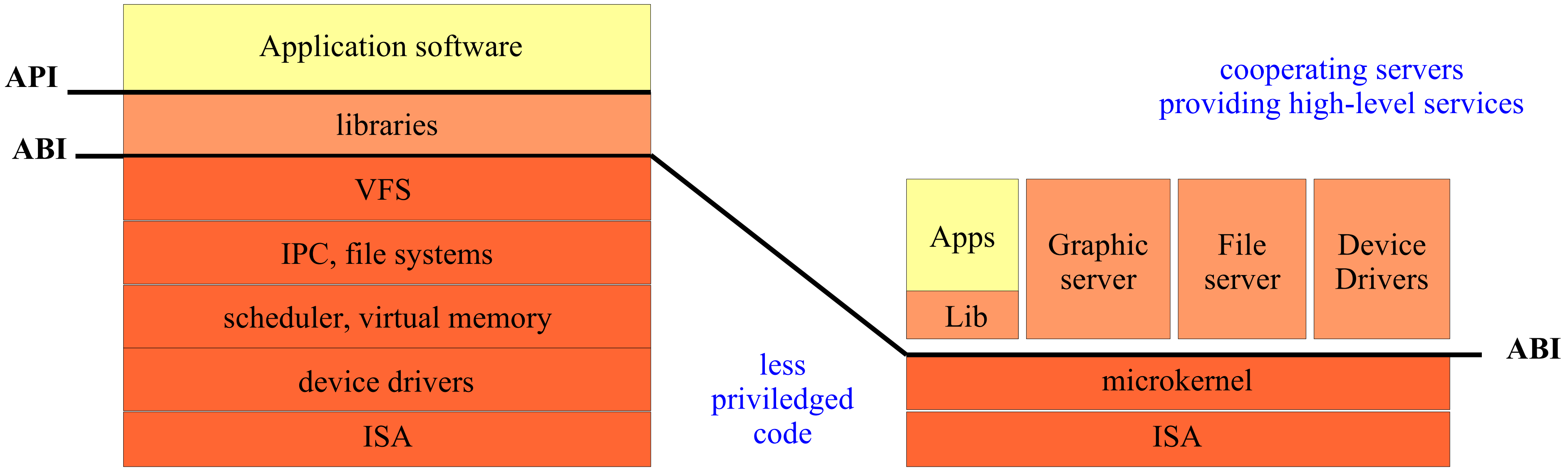
Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and
A. Jefferson Offutt. Maintainability of the Linux kernel. IEE Proceedings:
Software, 149:18–23, 2002.

Introducing Microkernels

- Core idea
  - Improve kernel maintainability and trustworthiness
  - Minimizing the Trusted Code Base (TCB) running in privileged mode
  - Seminal ideas can be traced to Brinch Hansen’s Nucleus (1970)
- Main principles
  - Small (privileged) kernel providing core functionality
  - Most OS services provided by user-level servers
  - Servers communicate via messages

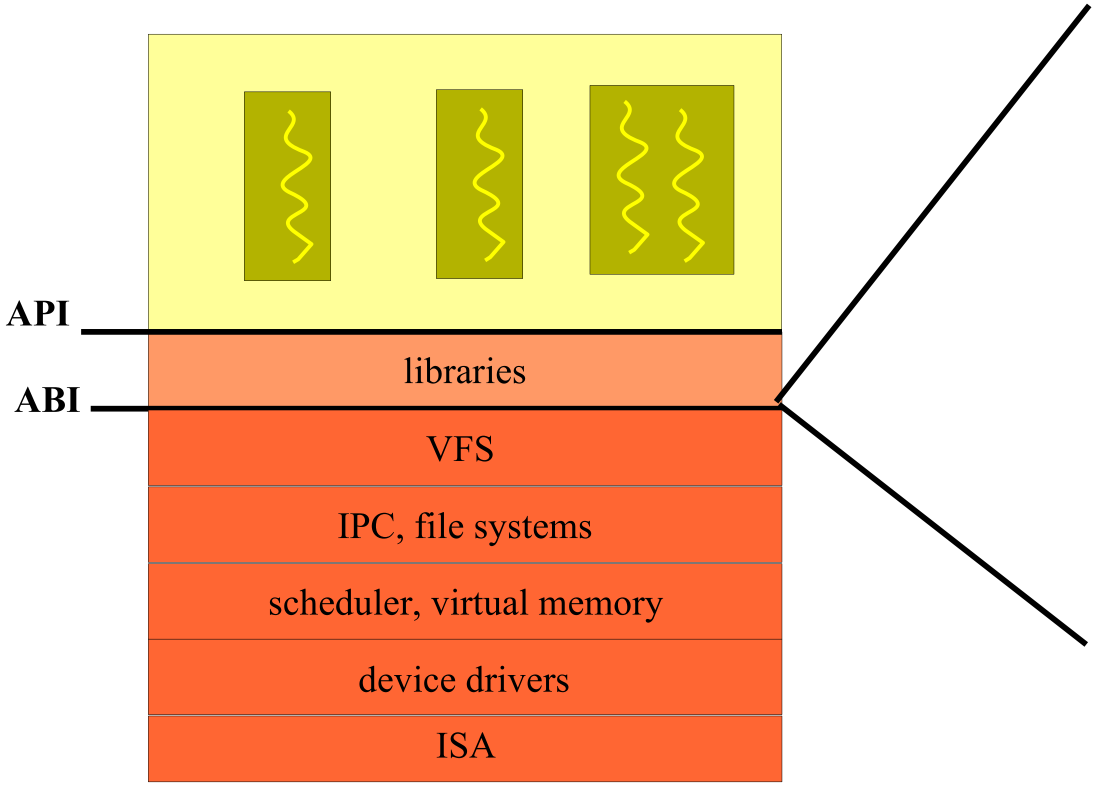
Operating System Architecture

- Microkernels
  - Minimal kernel (TCB) with only few and low-level mechanisms
  - Promote a modular design of cooperative user-level servers
  - Cooperation relies on Inter-Process Communications (IPC)



Operating System Architecture

- Linux/Unix Kernel Architecture
  - The pieces and the rationales…



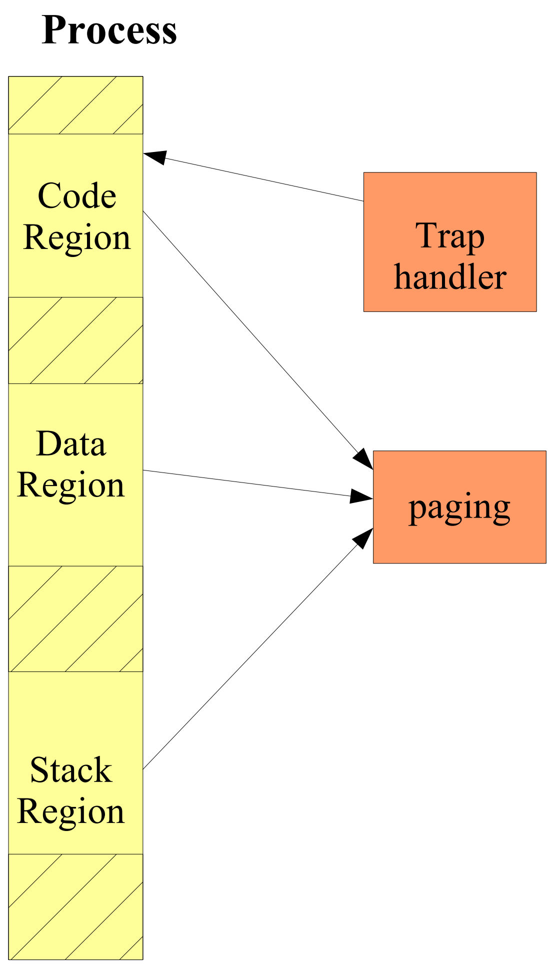
Processes
- Memory (sbrk)
- Signals (exceptions)
- Fork/Exec

Threads
- Create/Destruct
- Synchronization

Streams
- Sockets (tcpip, udp, local)
- Files (open/read/write/close/ioctl)
- Devices

Processes

- Processes as address spaces
  - A virtualized memory, isolated by hardware
  - Trap handler generates signals
- Mapped memory regions
  - Paging memory regions
- Management
  - Create by forking (duplicate process)
  - Kill by pid
  - Exec (loader like Elf loader or script loader)



→ Paging and loaders require file system support in the Kernel

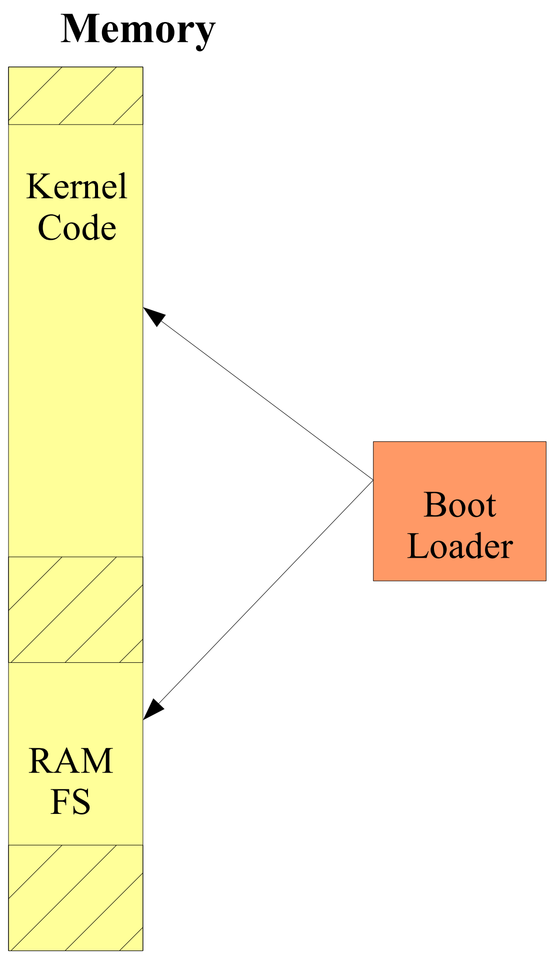
Boot Process

- First process
  - Traditionally /init
  - From the root partition /
  - Once the kernel has booted
- Boot loader
  - Loads the kernel in memory
  - From /boot

⇨ Boot loader requires file system support as well

⇨ BIOS loads the boot loader
So it needs file system support

⇨ And therefore everyone needs device drivers



Network and Thread Support

- Before the early 80's
  - No network support
  - No kernel threads
- Today's kernels
  - Light-weight processes
  - Known to the scheduler
  - Generic scheduling
  - Full network support

$\Rightarrow$ What are the rationales for kernel support?

Network and Thread Support

- Green threads
  - Can be entirely user-level library
  - Fast synchronization
  - Application-specific scheduling is possible
  - Requires cooperative scheduling
  - Blocking or non-blocking I/Os → Requires non-blocking I/Os from the kernel...

    And what about paging?
- Kernel threads
  - Light-weight processes
  - Known to the scheduler
  - Generic scheduling

Network and Thread Support

- Network stack
  - Device drivers for various NICs
  - Full IP – ARP – ICMP – UDP – TCPIP support
  - Supports socket streams
- Kernel support
  - Sockest are known to the Clib API
  - Sockets are known to the ABI ⇨ Why?

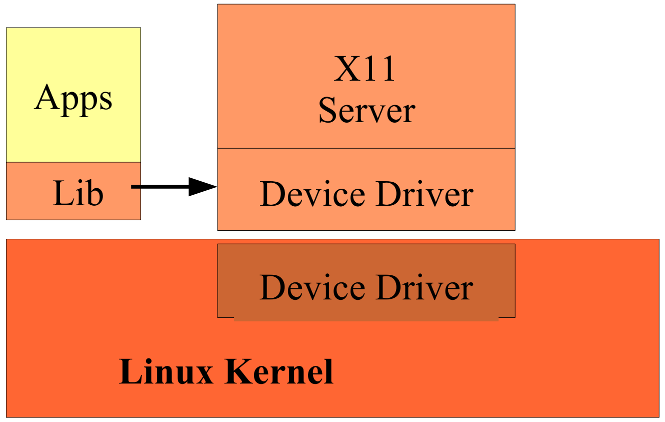
Why not just a block device?

Graphic Support

- X11 Server
  - User-space graphic server
  - User-space device drivers
- Basic kernel support
  - Basic generic device drivers
  - Like a framebuffer originally
  - Early graphics ~ gray-scale pixels

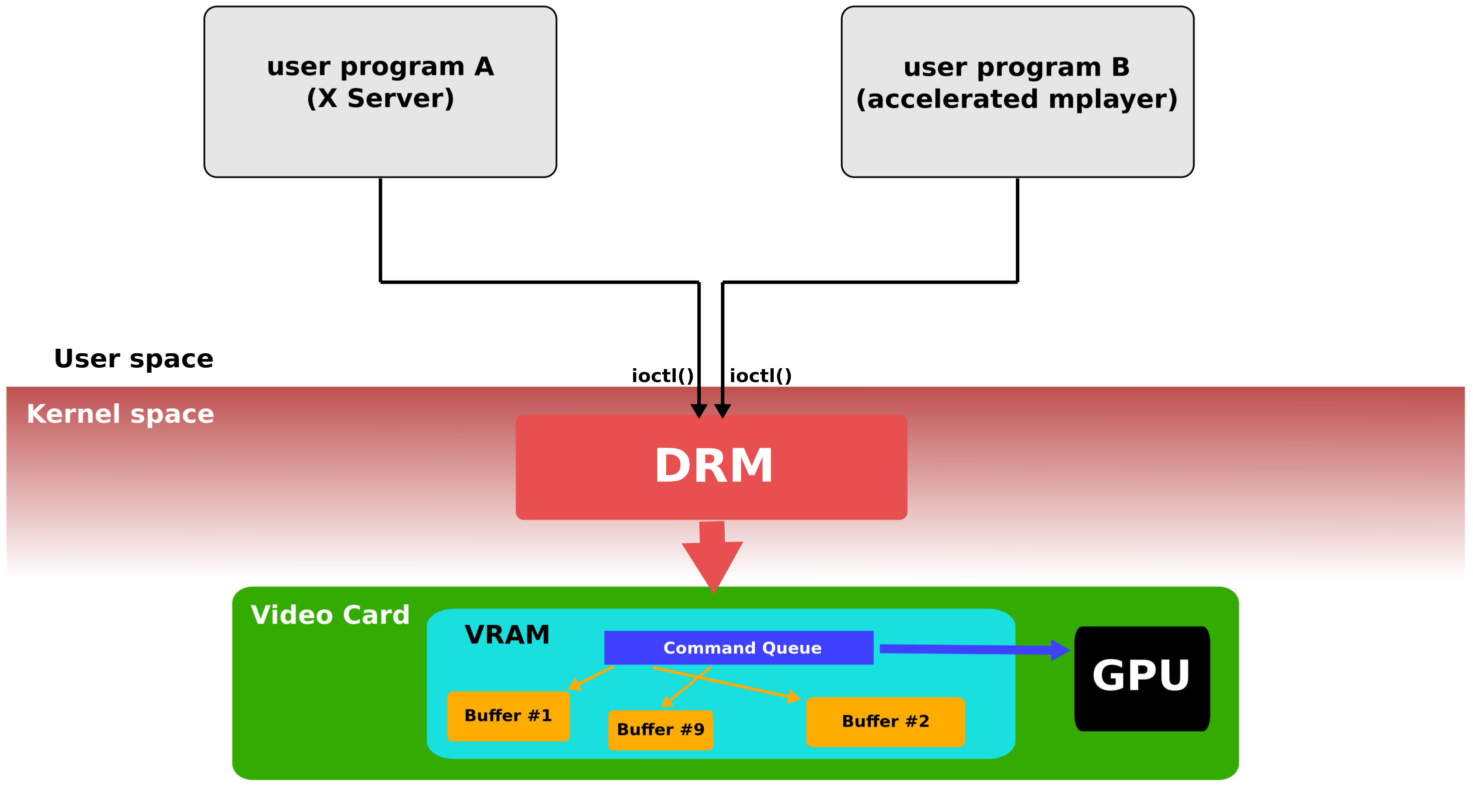
video memory → mapped file
mode change → ioctl

2D Acceleration?
bitblit → ioctl



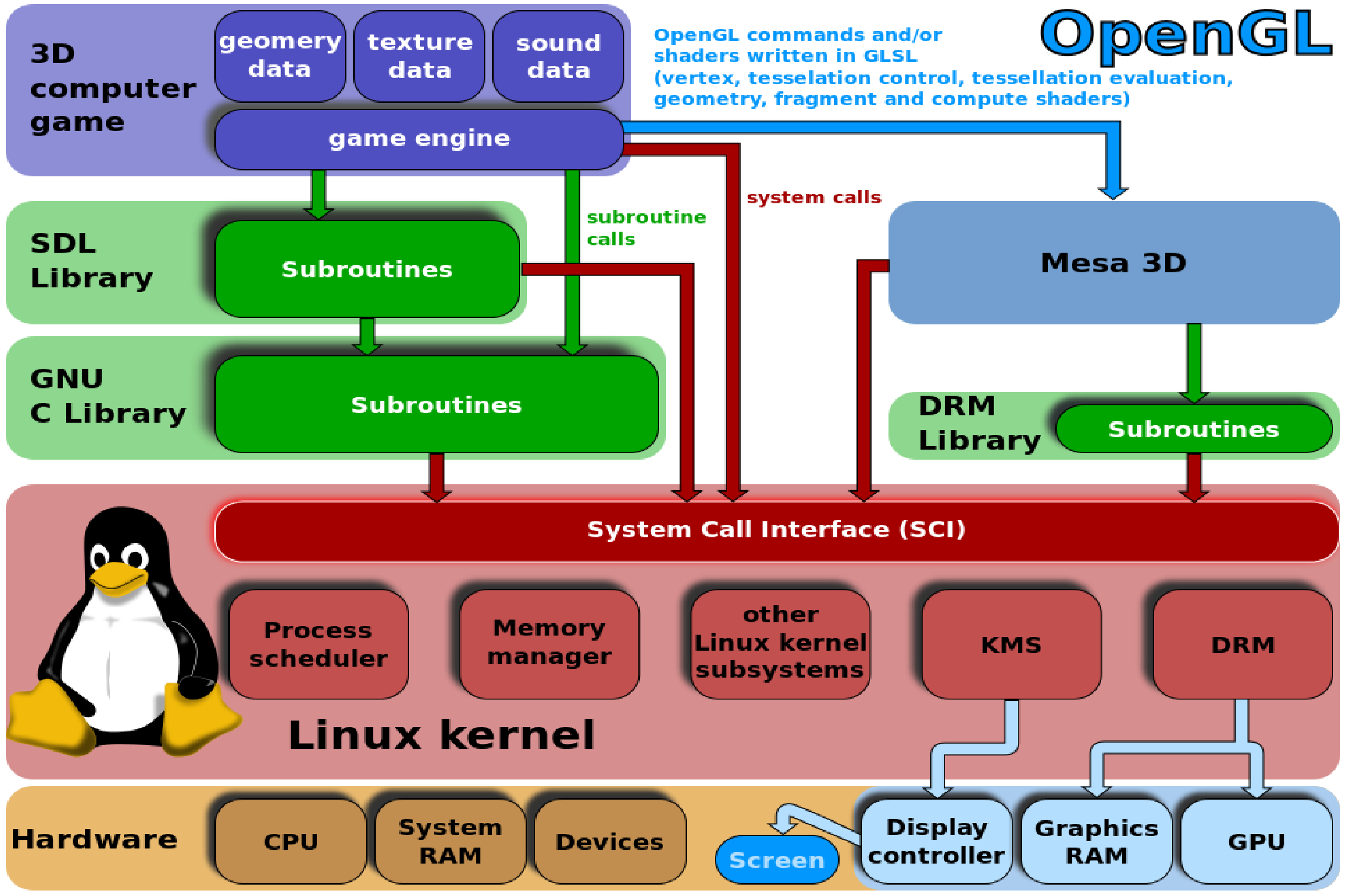
Graphic Support

- Direct Rendering Infrastructure (DRI)
  - Direct Rendering Manager



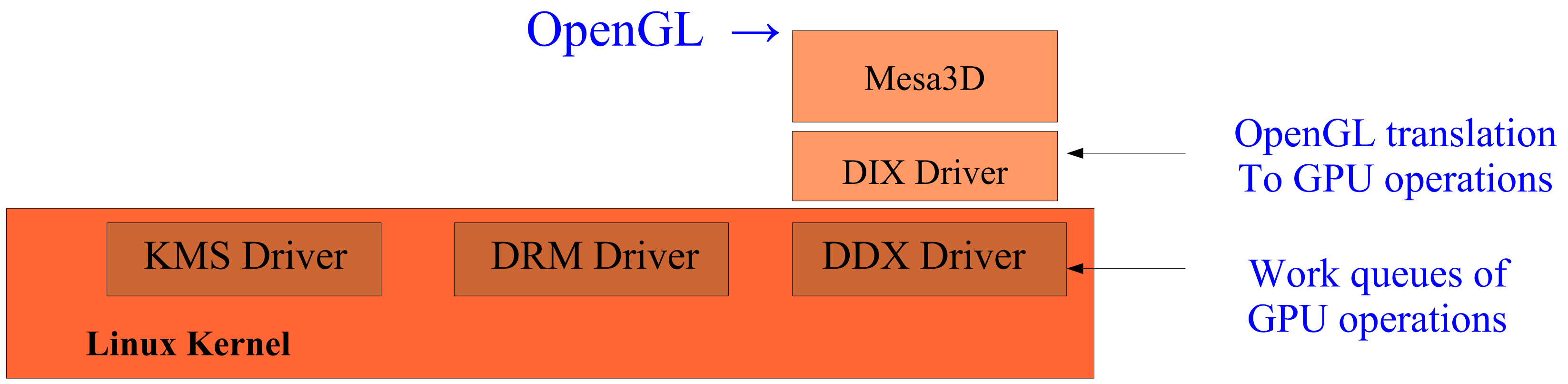
Graphic Support

- Direct Rendering Infrastructure (DRI)



Graphic Support

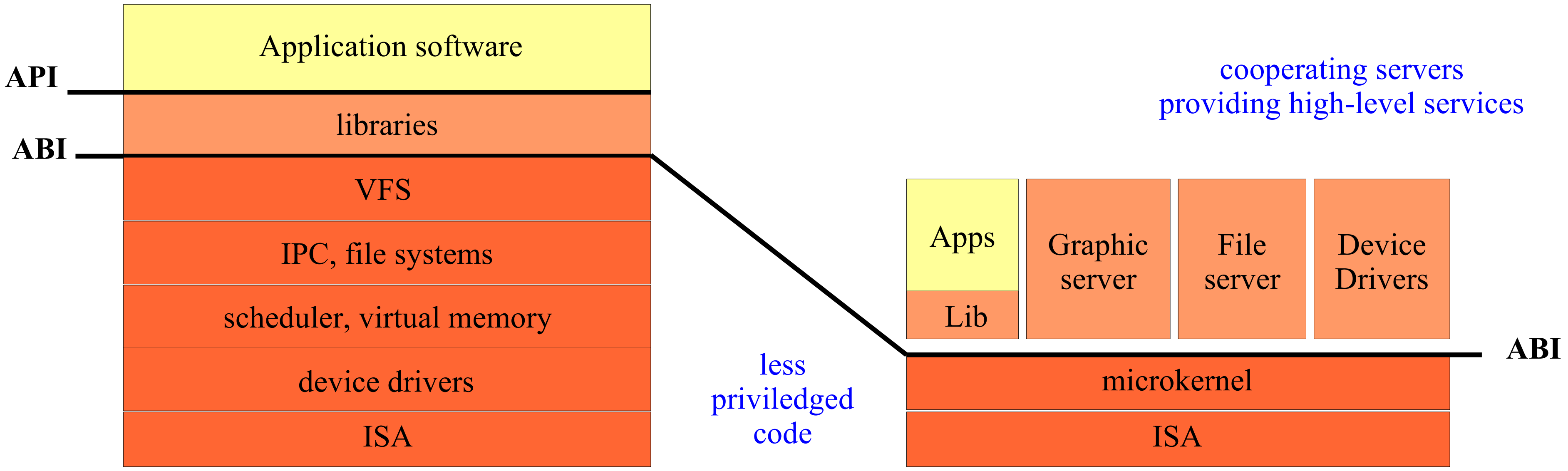
- Mesa3D
  - Open-source OpenGL
  - Simplified overview...



So now...
why do we have full network support
inside the kernel?

Operating System Architecture

- Microkernels
  - Minimal kernel (TCB) with only few and low-level mechanisms
  - Promote a modular design of cooperative user-level servers
  - Cooperation relies on Inter-Process Communications (IPC)



Microkernel Core Concepts

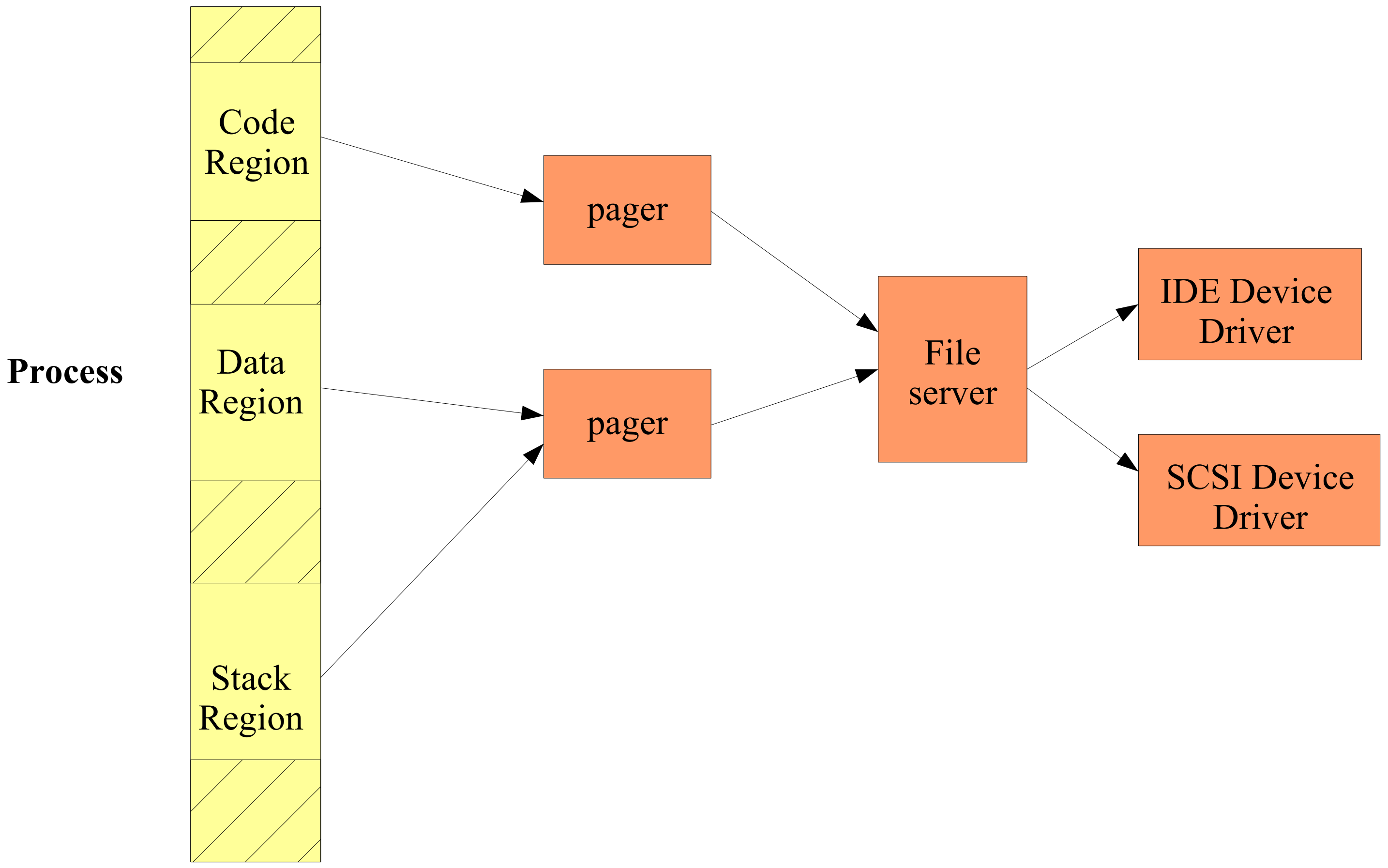
- Processes as address spaces
  - A virtualized memory, isolated by hardware
  - Mapping memory regions, managed by user-level pagers
- Threads in address spaces
  - Regular concept of threads (intra-process stacks of frames)
  - Microkernel is responsible for scheduling threads
  - Exceptions (traps) as IPCs
- Inter-Process Communication (IPC)
  - Message-based mechanism for inter-process communication
  - Sole medium of cooperation: send and receive messages

Microkernel Core Concepts

- Capabilities
  - Tokens representing priviledges
  - The *right* to execute an action on an entity
  - Verified at runtime
- Different microkernels use capabilities
  - KeyKOS (‘85), Mach (’87), EROS (‘99)
  - OKL4 V2.1 (’08): first cap-based L4 kernel
  - SeL4 ('09): first proven microkernel

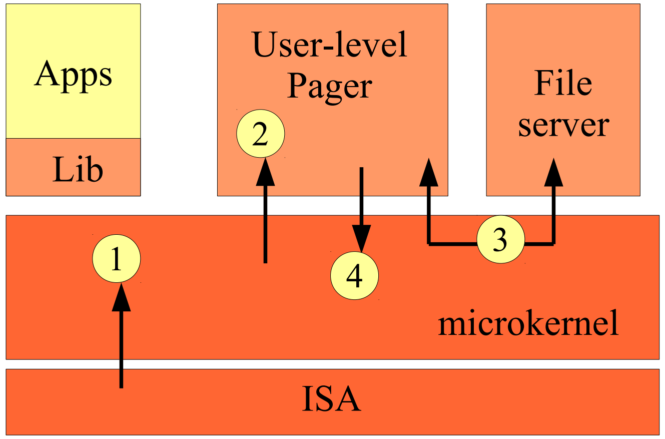
Microkernel – Processes

- Processes as address spaces
  - A virtualized memory, isolated by hardware
  - Mapping memory regions, managed by user-level pagers



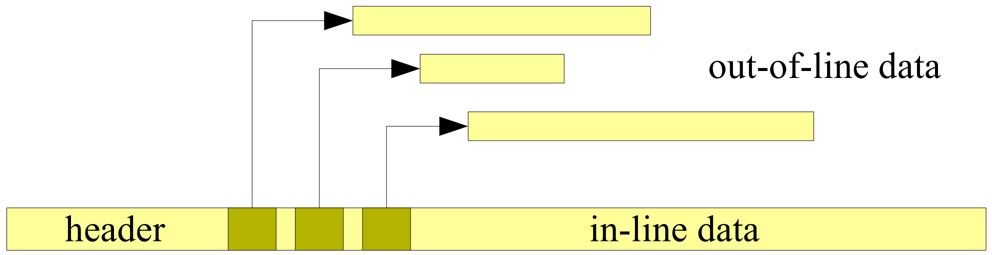
User-Level Pagers

- Principle
  - The microkernel manages the page table and/or TLB
  - Relies on an external user-level server for paging
- Overview
  - (1) a page fault occurs
  - (2) Upcalls (IPC) the corresponding external pager
  - (3) External pager finds the page on disk
  - (4) Provide the page to the kernel



Microkernels - IPC

- Inter-Process Communications
  - Between end points (either ports or threads)
  - Payload as both in-line data and out-of-line data
  - One receiver, multiple senders
  - Blocking or non-blocking operations
  - Receiving on multiple ports with one thread is usually possible
- IPC Broker
  - How do you get a remote port?

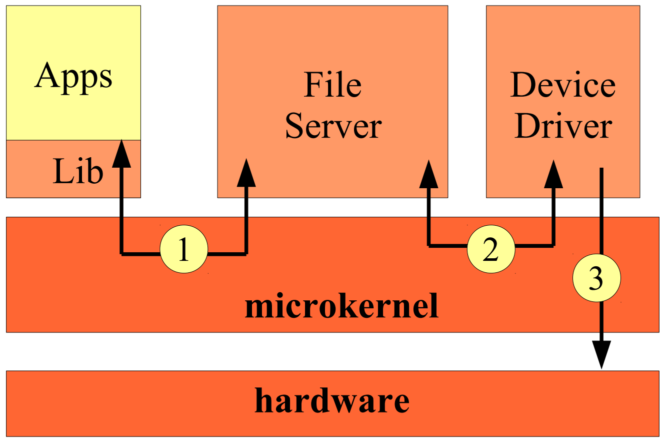


Microkernels - Threads

- *Regular* threads
  - Execution flow, local to a process
  - Usually associated capabilities
  - Usually execute at a certain priority level
  - Uncaught exceptions redirected as messages to a handler end-point
  - Often has thread-local variables
- *Regular scheduling*
  - Microkernels embeds one or more scheduling policies
  - Often based on priorities and time slices
  - Executing thread can yield
  - Executing thread can send or receive messages
  - Synchronization mechanisms

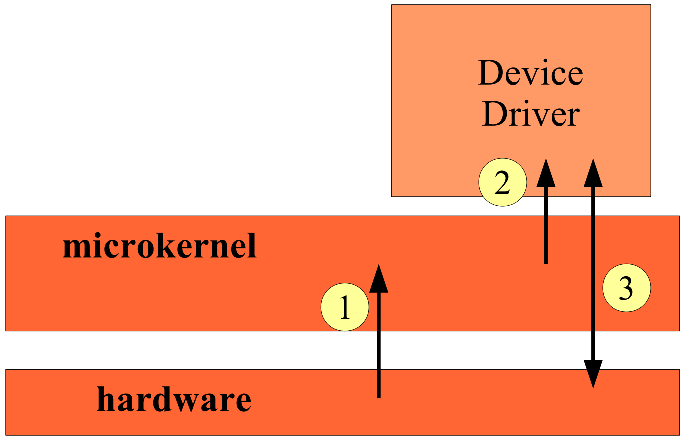
Microkernels – Threads

- Discussing regular I/O
  - Application reads or writes a file or a socket
  - Translates to an IPC to the file server
- File Server
  - Manages the file cache (blocks are available or not)
  - IPC to the device driver to read missing blocks
- Device driver
  - Issues the I/O request to the device
  - Using memory-mapped I/O ports
  - Configured interrupts



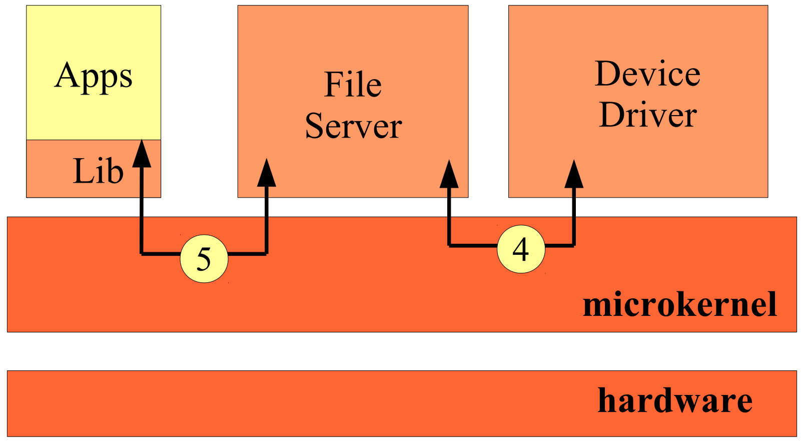
External Device Drivers

- Interrupt handling
  - The microkernel captures the interrupts (1)
  - But it does not handle the interrupts
  - It forwards it to the concerned driver
  - Generating an IPC (2)
- The device driver handles the interrup (3)
  - Potentially dialoging with the device through I/O ports
  - Interrupt may indicate the end of the DMA transfer



External Device Drivers

- Interrupt handling (cont…)
  - The device driver notifies the file server (4)
  - The file server caches the block and notifies the application (5)
  - The library unblocks the calling thread, the data being available



Design Session

Warm-up discussion:

What are the differences between these three?
- a specification
- a design
- an implementation

Let's get to work:

1) What is the difference between a syscall and an IPC, if any?
2) Why do you need a service broker? Design one.
3) How do you create a process?
4) How do you map/unmap regions in a process?
5) How do you wrap the MMU into an IPC protocol with a pager?
6) How do you design malloc/free?
7) How do you create threads?
8) How do you manage thread stacks?
9) Specify send and receive syscalls
   - format of the messages, fixed or variable?
   - memory management, avoiding copies as much as possible
   - does the sender block if the receiver isn't ready?
   - does the receiver block if there is no message
   - IPC end points? threads? Mailboxes or ports ?
10) Design send and receive syscalls

Bonus: How do you bootstrap?

Discussing Microkernels

- Some spectacular “failures”
  - Most notorious: IBM Workspace OS
  - Also the GNU Hurd
- Recently spectacular “successes“
  - Apple Mac-OS
  - GreenHills (world leader in RTOS)
  - Nicta (1.5 billions smart phones use L4)

IBM Workplace OS

- Unify IBM’s operating systems (DOS, OS/2, AIX, OS/400, 390, ...)
  - Cost saving but also multiple personalities on the same hardware
  - Based on Mach 3.0 from Carnegie Mellon University (CMU)
    - Across a wide range of hardware environments
    - PDAs, desktops, massively parallel machines
  - With concurrent OS personalities
    - Sharing personality neutral services (PSNs)
    - Applications could use services from multiple OSes
- Very ambitious project (budget and manpower)
  - Plagued by internal problems
  - Too much focus on portability, not enough on personalities
  - Microkernel performance not mentioned as one of the main issues

Discussing Performance

- Nevertheless, disappointing first generation
  - By 1991, an IPC overhead bottomed at 115 µs on a 486-DX50
  - Conventional Unix system call was at roughly 20 µs
  - About 10 times faster (syscall = 2 IPCs)
  - But it is hard to discuss and appreciate micro-benchmark numbers
- Overall results
  - Some applications performed as well or slightly better on microkernels
  - Some showed peak degradation of 50% to 66%
  - Clearly depends on how much applications use system calls

Discussing Performance

- Analysis
  - At least 73% of the measured penalty was attributed to IPC overhead
  - 10% was attributed to multi-processor provision in microkernel that was not present in the uniprocessor Unix measured
  - 17% were unattributed
- Conclusions
  - IPC-based architecture was flawed from a performance perspective
    - Mach and Chorus microkernels reintegrated performance-crucial servers
    - Both had limited commercial successes
  - IPC performance is believed to be the main rationale for the commercial failure of microkernels

Discussing Performance

- Which applications were used?
  - Regular applications as sed, egrep, yacc, gcc or compress
  - Are they really representative?
- What would be the case for Open Office?
  - More CPU intensive and less I/O bound
  - For instance, graphics are already IPC-based (with X11)
- Was 50-60% degradation a real showstopper?
  - It was considered as one, by researchers themselves
    - Industry was not to accept something researchers found a failure
  - What about Moore's law
    - Hardware was twice faster every year

Discussing Commercial Failure

- The overhead can't justify commercial failure
  - Did you notice that Java failed in its early years because it was slow?
  - What about newer versions of Microsoft Windows?
- Really missing a killer reason for adoption
  - Flexibility, modularity, and freedom of choice are not obvious market growth opportunity
  - Forgot the natural resistance to change bottom layers of complex critical software stacks
- Who was to pay for porting existing OSes on microkernels?
- **Who was to rewrite all device drivers?**

Discussing Commercial Failure

- We have a chicken and egg problem
  - New applications requiring a specific OS platform is a commercial No-No
  - Without new applications, microkernel were not seen as an advantage
- Traditional Unix systems evolved proposing new system calls
  - Memory mapped files
  - Pinning pages in memory
  - Efficient signals for virtual memory protection violations
- Middleware phenomenon versus multiple personalities
  - We saw cross-porting of much libraries and file-systems
  - We saw cross-porting of applications
  - We also saw distributed integration
- Hypervisor phenomenon
  - Multiplexing operating systems

Stepping Back...

- What about our numbers and our understanding of them?
  - Numbers clearly showed that IPC was the crux of the problem
  - At about 100 µs on a 486-DX50, it seemed an unacceptable overhead
  - The conclusion was that approach was flawed and doomed
- Was that correct? Well... not really
  - Naive and uninterpreted measurements can be misleading
  - Performance measurements on complex systems are not trivial
- Reality
  - It was the implementation of the kernels that was at fault
    - The code was borrowed from monolithic kernel
    - It was improved incrementally (usually considered a good idea)
  - But the approach proved to be at the heart of the problem
    - Reducing a large kernel does not produce an efficient micro-kernel

Microkernel – Second Generation

- Case study: L4 microkernel
  - Developed at GMD in 1995
  - Provides a minimal set of abstractions
    - Like the first generation
  - But argues non-portable implementations
    - For real performance, implementations must be processor dependent
    - So writing microkernel code is not about writing portable code
    - Even between 486 and Pentium!

Jochen Liedtke, *Towards Real Microkernels*
Communications of the ACM, Vol 39, No 9, 1996

Jochen Liedtke, *Improving IPC by Kernel Design*,
SIGOPS'93

Introducing L4 Microkernel

- L4 Abstractions
  - Task: an address space with threads
  - Inter-Process Communications (IPC)
- L4 Design
  - Only 7 system calls while Mach has 140 system calls
  - Footprint of 12KB while Mach footprint is 300KB
- L4 Performance (on 486-DX50)
  - 8-byte IPC
    - **5 µs** for L4 against 100 µs for Mach
    - L4 IPC (2*5µs) is twice as fast as Unix system call
  - 512-byte IPC
    - **18 µs** for L4 againts 172 µs for Mach
    - Only twice slower than a Unix system call (2*18µs versus 20µs)

From L3 to L4 – Starting Fresh

- Principles and methods
  - IPC performance is the master design issue
  - All design decisions require a performance decision
  - If something performs poorly, look for new techniques
  - Synergetic effects have to be taken into considerations
  - The design has to cover all levels from architecture down to coding
  - The design has to be made on a concrete basis (hardware capabilities)
  - The design has to aim at a concrete performance goal
  - Only accept a concept in the kernel if it cannot be implemented above

Improving IPC performance

- Concrete design
  - 486-based hardware family
- Intel 486-DX50
  - Clock rate 50MHz
  - On-chip cache of 8KB
  - Typical instruction takes 1 or 2 cycles, assuming cache hits
  - The MMU translates 32-bit virtual addresses
    - Using 4KB pages
    - With read-write protections
    - TLB has 32 entries
  - TLB is flushed on switching address space

IPC Performance Objective

- Simple scenario – Zero-byte message
  - Two threads A and B in two address spaces
  - Sending a zero-byte message, from thread A to thread B already waiting
- Rough estimate
  - Down to 20 instructions
    - Ignoring parameter passing and testing
  - These 20 instructions are about 127 cycles
    - Out of which 107 cycles are consumed to enter/leave kernel mode
  - 486 MMU is flushed on address space switch
    - So we need to add at least 5 TLB misses
    - One TLB miss is about 9 cycles
  - So this means a lower bound of 172 cycles (about 3.5µs)
    - Not realistic, but a lower bound
    - Since we assumed no write or code prefetch delays or cache misses

IPC Performance

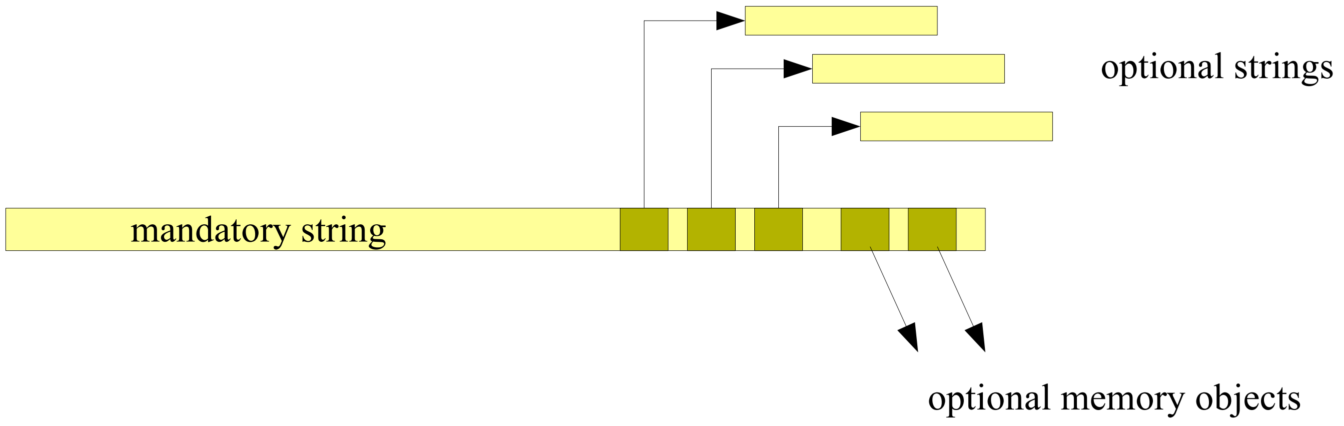
- A realistic target was 350 cycles, about 7µs
  - Reach **250 cycles**, **5µs**
  - **T=5µs** will be used as a basic unit for discussing performance
- How was it done?

IPC Performance

- Reducing system calls
  - System calls are expensive
    - 107 cycles for entering and leaving kernel, about 2µs
    - So about 40% T
  - Traditional RPC
    - Two send and receive IPC calls on both client and server side
    - Results in 4 enter-leave kernel-mode instructions
  - Combining IPCs
    - IPC call combines sending a message and receiving the reply
    - IPC call combines replying and receiving next message
    - We are down to only 2 enter-leave instructions

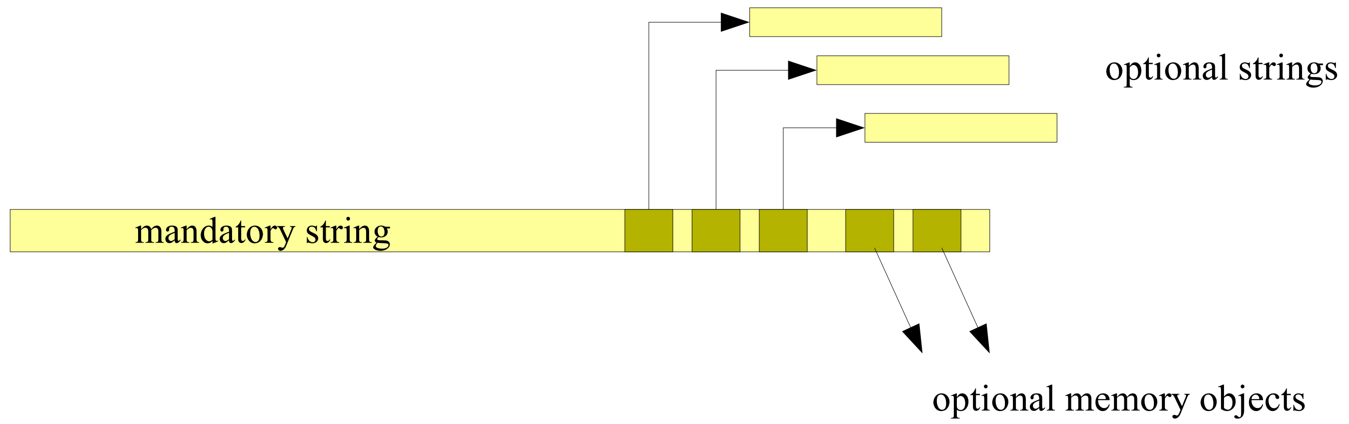
IPC Performance

- Overhead of system calls and address space switch
  - About 60% T, which suggests to have complex message à la Mach
  - Sending larger messages reduce the relative overhead
- Message format
  - With mandatory and optional strings
  - And optional memory objects
  - Provides a gather-scatter approach



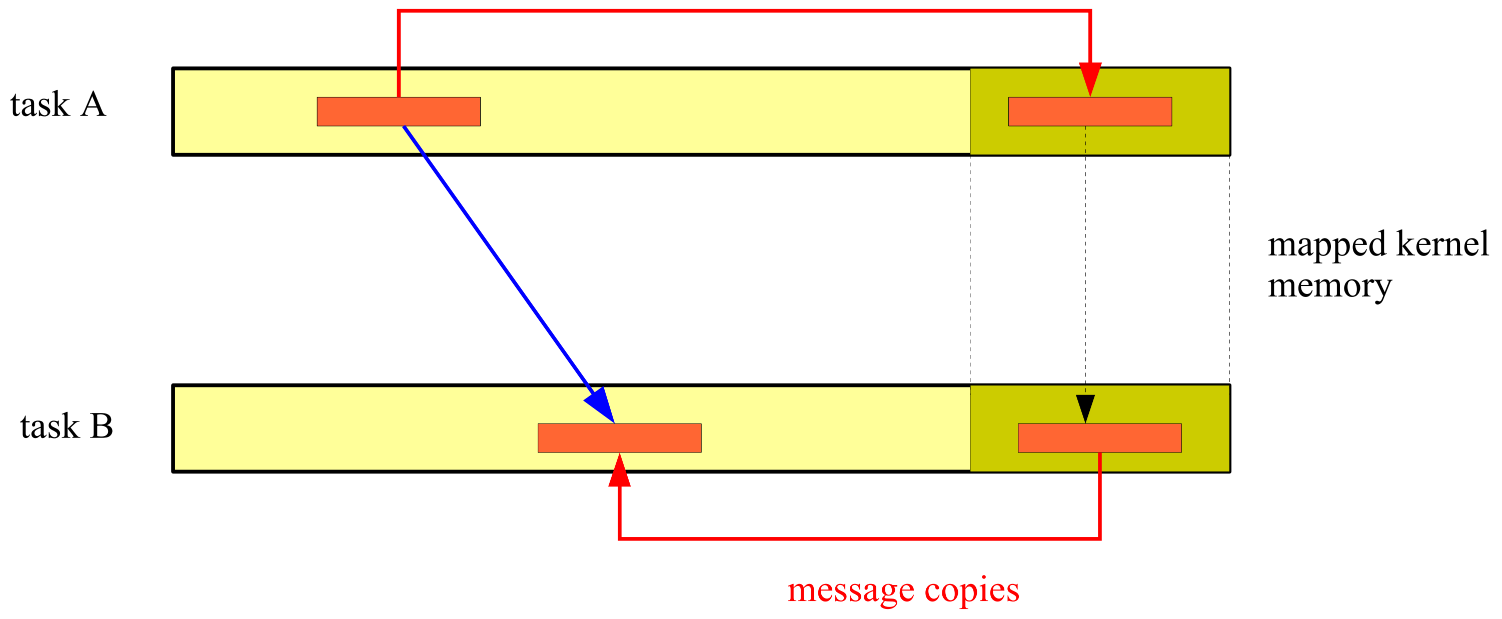
IPC Performance

- Optional memory objects
  - Passed by playing with address space mappings
- Optional strings
  - Passed through copies between address spaces
  - But traditional application-level copies are avoided
  - Message is constructed using the address of language variables



IPC Performance

- Traditional design
  - Costly copies, from sender to kernel and from kernel to receiver
- Overhead
  - For n-byte message, about 20+0.75n cycles
  - 8-byte message is about 26 cycles, about 0.5µs, about 10% T
  - 512-byte message is about 404 cycles, about 8µs, about 160% T



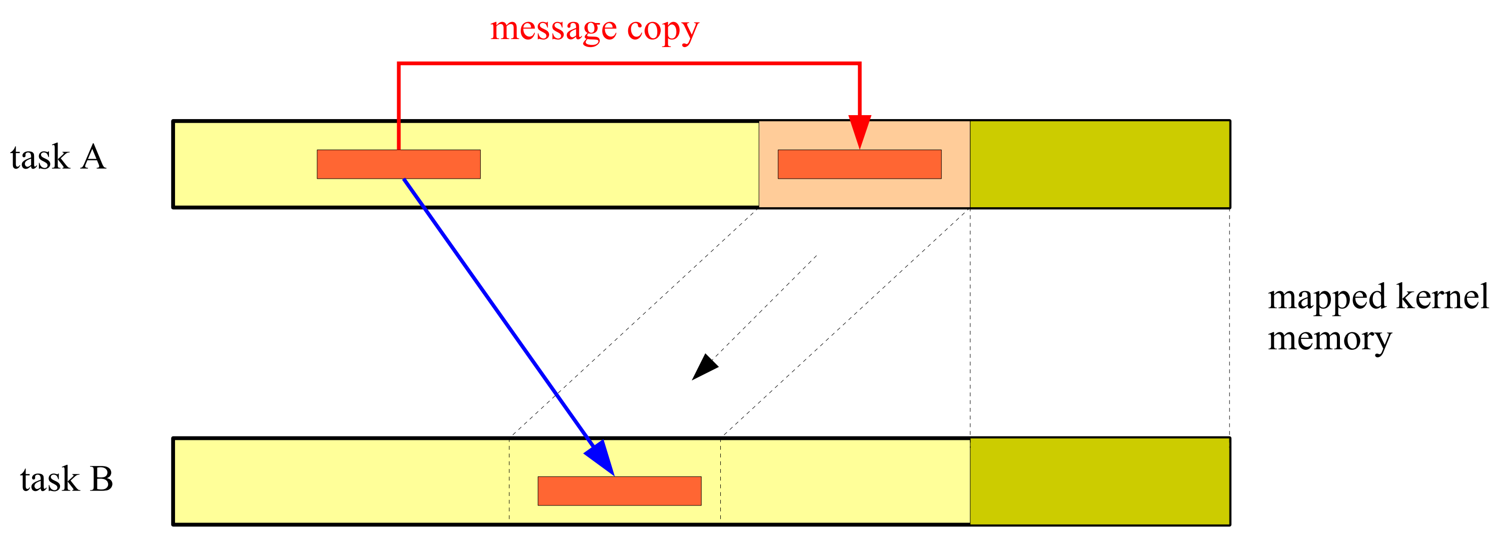
IPC Performance – Lightweight RPC

- Lightweight RPC
  - Based on user-level shared buffers
  - Requires an explicit open of connections to allocate buffers
  - One copy in sender address space from message to the shared buffer
- Does not scale well
  - For server tasks with many clients
  - Shared buffers with large messages may become a scarce resource
- Security breach
  - Receiver can't ensure the legality of the message
  - Since the sender may change it after the receive
  - Constitute hidden channels without proper IPC controls

Bershad et al, Lightweight Remote Procedure Call,
ACM Symposium on Operating System Principles, Dec 1989, pp 102-113

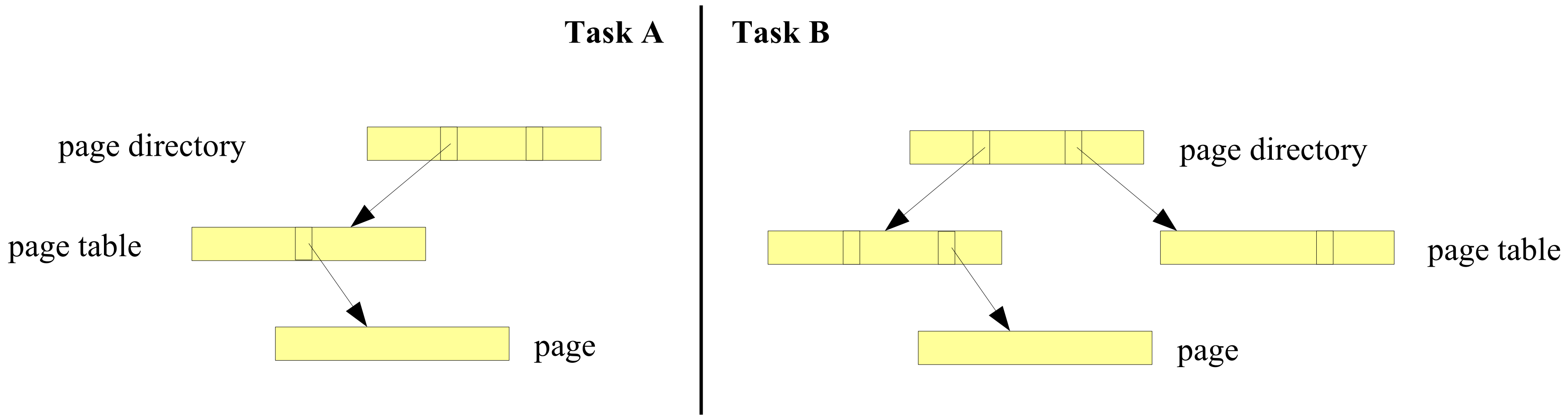
IPC Performance – Single-Copy Design

- Direct transfers by temporary mappings
  - Map the necessary virtual pages
    - Temporary mapping in task A, only accessible to the kernel
    - Copy only once the message
  - Unmap the temporary mapping



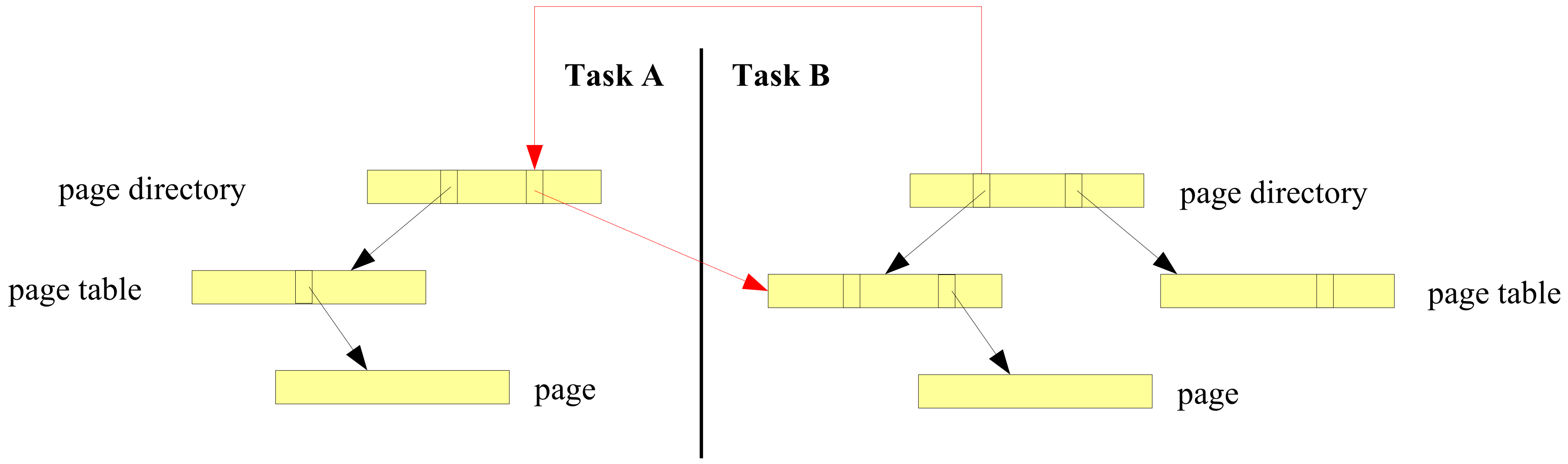
IPC Performance – Single-Copy Design

- Tempory mapping design
  - Two challenges
    - Temporary mapping must be fast
    - Design must work with multiple threads in the same address space
  - Performance on 486 hardware
    - 486 MMU uses an architected two-level page table
    - Page directory is the first level, hold 1024 entries
    - Page tables are the second level, each also holds 1024 entries to pages



IPC Performance – Single-Copy Design

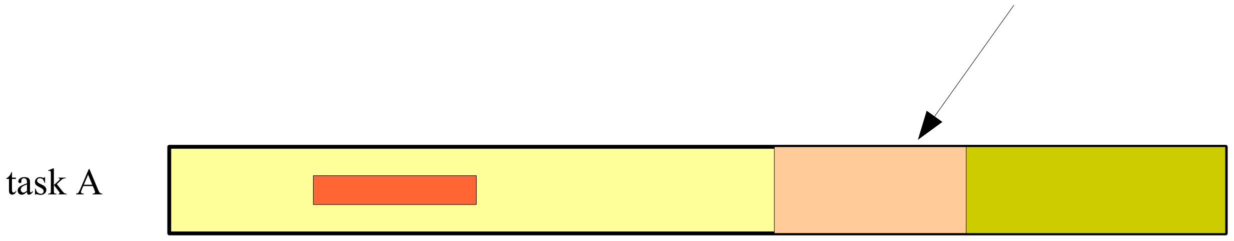
- Tempory mapping design
  - Two-word-copy mapping on 486 hardware
  - Each page table corresponds to 4MB
  - To map aligned 4MB, we need one word copy between page directories
  - Messages are restricted to 255 strings and always map to 8MB region

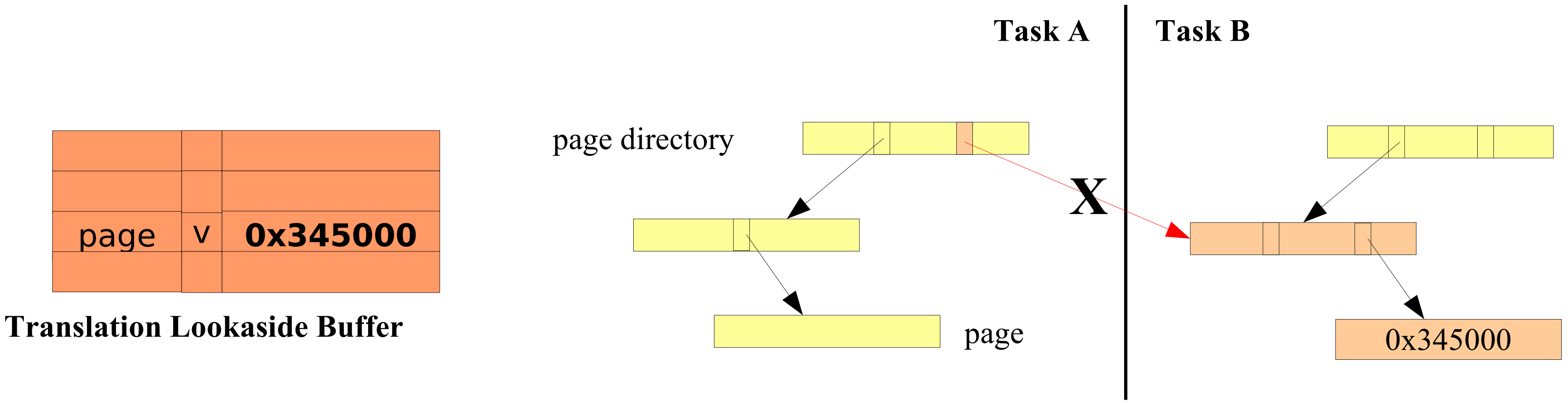


IPC Performance – Single-Copy Design

- Removing the temporary mapping
  - Cutting the entry in the page directory is not enough
  - We need to ensure that we have a *IPC-clean* TLB:

An **IPC-clean TLB** has no entries related to any ***IPC-mapped region***



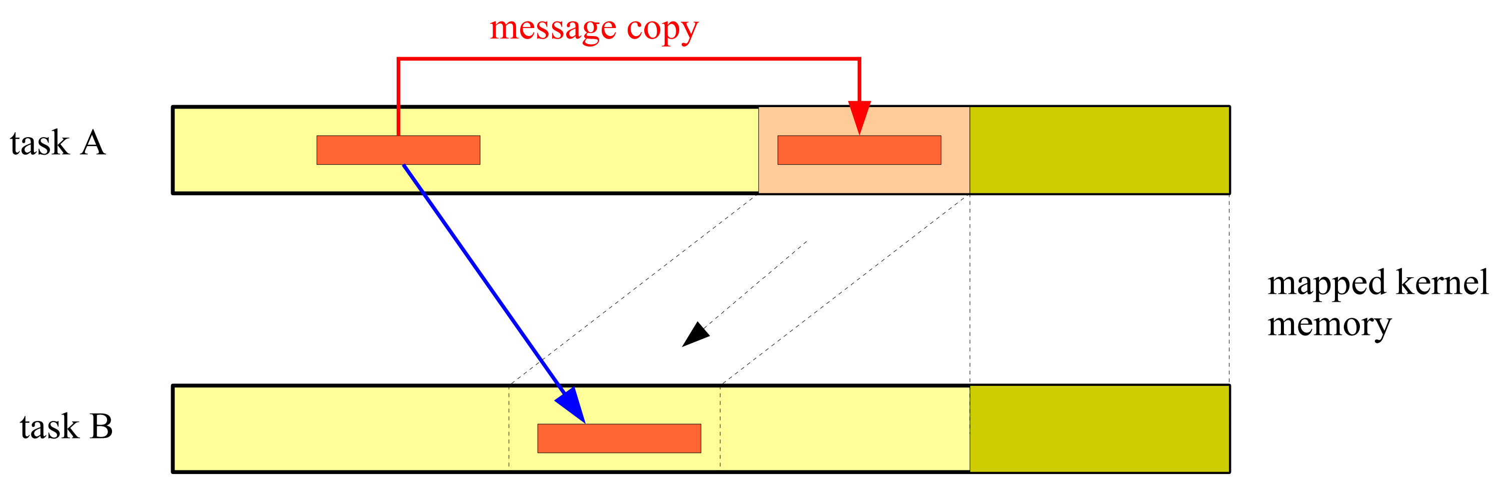


IPC Performance – IPC-Clean TLB

- Ensuring an IPC-clean TLB
  - Flushing is too expensive
  - Flushing an entire TLB induces a too great overhead
  - Flushing individual pages for 4MB regions is too slow
- A solution with one thread per address space
  - When starting a task
    - The TLB is *IPC-clean* since it is empty
  - When switching to a thread in a task
    - The TLB is flushed on address space switch
    - Therefore the TLB *IPC-clean*
  - What about during the IPC?

IPC Performance – IPC-Clean TLB

- During the IPC:
  - (1) map the IPC region
  - (2) do the message copy
  - (3) unmap IPC region
  - (4) switch to receiver thread
- In step (4), the TLB is IPC-clean
  - We switch address space (since we have one thread per task)
  - So we flushed the TLB, so it is IPC-clean



IPC Performance – IPC-Clean TLB

- Discussing mutliple threads
  - One 4MB region per thread may be difficult to get (only 1024 entries)
  - Use only one IPC-mapped region, shared by all threads
- First design
  - Synchronization
    - Only one thread can use the IPC-mapped region
    - Mutual exclusion only during the send part of the IPC
  - Mapping management
    - Restore the correct mapping upon thread switching
    - Since each thread has potentially its own mapping for the IPC-mapped region

IPC Performance – IPC-Clean TLB

- Scheduler modifications
  - When scheduling between threads
    - Always invalidate the IPC region in the page table (no access allowed)
  - Between threads in different address spaces
    - Flush the TLB (as before)
- Pager modification
  - Upon any page fault in the IPC region
  - Restores the correct mapping for the current thread
    - At page fault, we known the address and faulting thread
    - Since a given thread may be in only one IPC at a time
    - We know the IPC region to restore

IPC Performance – First Milestone

- Key design points
  - Used Rendez-Vous
  - Combined send and receive (half the number of system calls)
  - Only one copy through temporary virtual memory mapping
  - Required to change the scheduler to ensure IPC-clean TLB
    - Switch to receiver upon rendez-vous
    - Thread-aware temporary mapping, with pager support
- Now what?
  - Managing timeouts and wakeups…

IPC Performance – Timeouts & Wakeups

- Timeouts may be specified on each IPC
  - A timeout of τ means that the thread is awakened if message transfer has not started τ ms after invoking the operation
- Timeouts require a wakeup queue
  - For each IPC, insert the thread in the wakeup queue
  - Upon IPC completion, remove the thread from the wakeup queue
- Timer interrupt is expensive
  - Must scan the wakeup queue
  - Test for the expiration of each timeout value
  - For expired timeouts, awake the corresponding thread
    - Remove it from the wakeup queue
    - Insert it in the ready queue

IPC Performance – Timeouts & Wakeups

- Fast wakeup queue operations
  - Simple and fast design is an array
    - Indexed by the thread number
    - Holding the wakeup time, computed from timeout values
  - Insert/remove from the wakeup queue is then extremely fast
    - Just an array store (with zero or a wakeup time)
- But the timeout management is costly
  - Timer interrupt handler must scan the array
  - For 16K entries
    - This would require 2ms for each timer interrupt
    - On-chip cache would be flooded
    - Not acceptable
  - Why so many threads?
    - Remember, blocking IPC tend to require many threads

IPC Performance – Timeouts & Wakeups

- A better-balanced design
  - Use N lists for the wakeup queue
  - A thread with a timeout τ is inserted in the (τ mod N) list
- Timer interrupt handler
  - For k threads, on average, scans k/n entries
- Optimization to reduce scanning time further
  - Divide each of the N wakeup queues
  - Create a short-timeout and a long-timeout wakeup queue
  - Insert in the short-timeout queue threads whose timeout approaches

IPC Performance – Timeouts & Wakeups

- For n=8 lists
  - A wakeup granularity of 4 ms (check timeout list 250 times per second)
  - With 400 threads
    - We instpect at most 250*400/8= 12500 inspected entries per second
    - We spend less than **1%** of processor cycles in scheduling
  - At this rate of 12500 entries inspected per second
    - This would require at least 12500 ipc per second
    - Would mean at least 6% of the CPU time used in pure IPC
  - In practice
    - Since 75% of the IPC have concrete timeout values others than 0 and ∞
    - This scenario would require 50,000 IPC/sec
    - At 50000 IPC/sec, IPCs would take **25%** of the processor time
  - Scheduling overhead is therefore about 4% of that total IPC time
    - 4% = 1/25 (the 1% and 25% processor cycles above)

IPC Performance – Timeouts & Wakeups

- Long Timeouts and wakeups
  - Even though timeouts may be limited to 2^{31} ms, about 24 days
  - Corresponding wakeup times will be longer
  - 64bit quantities are too expensive
  - Using too many registers on a 32bit processor
- Use base+offset to represent a wakeup time
  - Manage time offsets always smaller than 2^{24} ms
  - Every 4.5 hours (2^{24} ms), the kernel updates the base
- Wakeup lists managed with a base and an offset
  - Split wakeups between those in the current window and later wakeups
  - On the time interrupt, only scan wakeups in the current window (2^{24} ms)

IPC Performance – Second Milestone

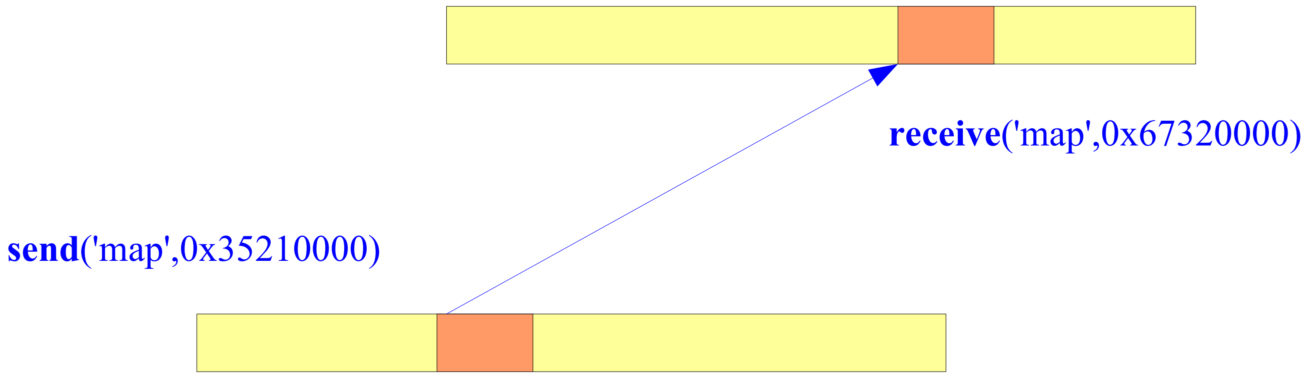
- Key design points
  - Used Rendez-Vous
  - Combined send and receive (half the number of system calls)
  - Only one copy through temporary virtual memory mapping
  - Required to change the scheduler to ensure IPC-clean TLB
    - Switch to receiver upon rendez-vous
    - Thread-aware temporary mapping, with pager support
  - Efficient management of wakeup queues
- Now what?
  - Address spaces...

L4 Address Spaces

- History
  - First generation microkernels had limited flexibility
    - Provided only the concept of external pagers, with mapped regions
    - Replacement policy were not accessible
  - L4 approach
    - Only mechanisms in the kernel
    - Powerful enough to build policies outside the kernel
- Core ideas
  - Recursive construction of address spaces through mappings
    - An *initial space* represents the physical memory
  - Address space owners
    - Threads executing in that address space
  - Only three operations by owners
    - ***Grant*** or ***Map*** a page to another address space
    - ***Demap*** a page from other address spaces

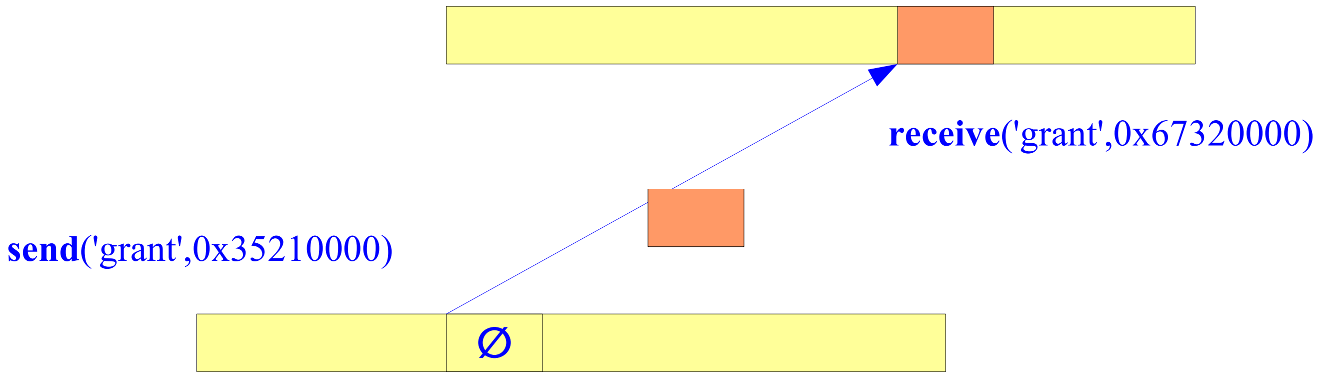
L4 Address Spaces

- **Map** IPC between owner threads
  - send('map',virtual @)
  - receive('map',virtual @)
- Semantics
  - Applies to a page
  - Sender retains the page in its own address space
  - Receiver accepts the page mapping, chooses the address



L4 Address Spaces

- **Grant** IPC between owner threads
  - send('grant',virtual @)
  - receive('grant',virtual @)
- Semantics
  - Applies to a page
  - Sender **does not retain** the page in its own address space
  - Receiver accepts the page mapping, chooses the address



L4 Address Spaces

- Memory mappings

$\sigma : V \rightarrow R \cup \{\emptyset\}$

where V is the set of virtual pages
R is the set of available real pages (physical)
$\emptyset$ is the nil page that cannot be accessed

Initial mapping representing the physical memory

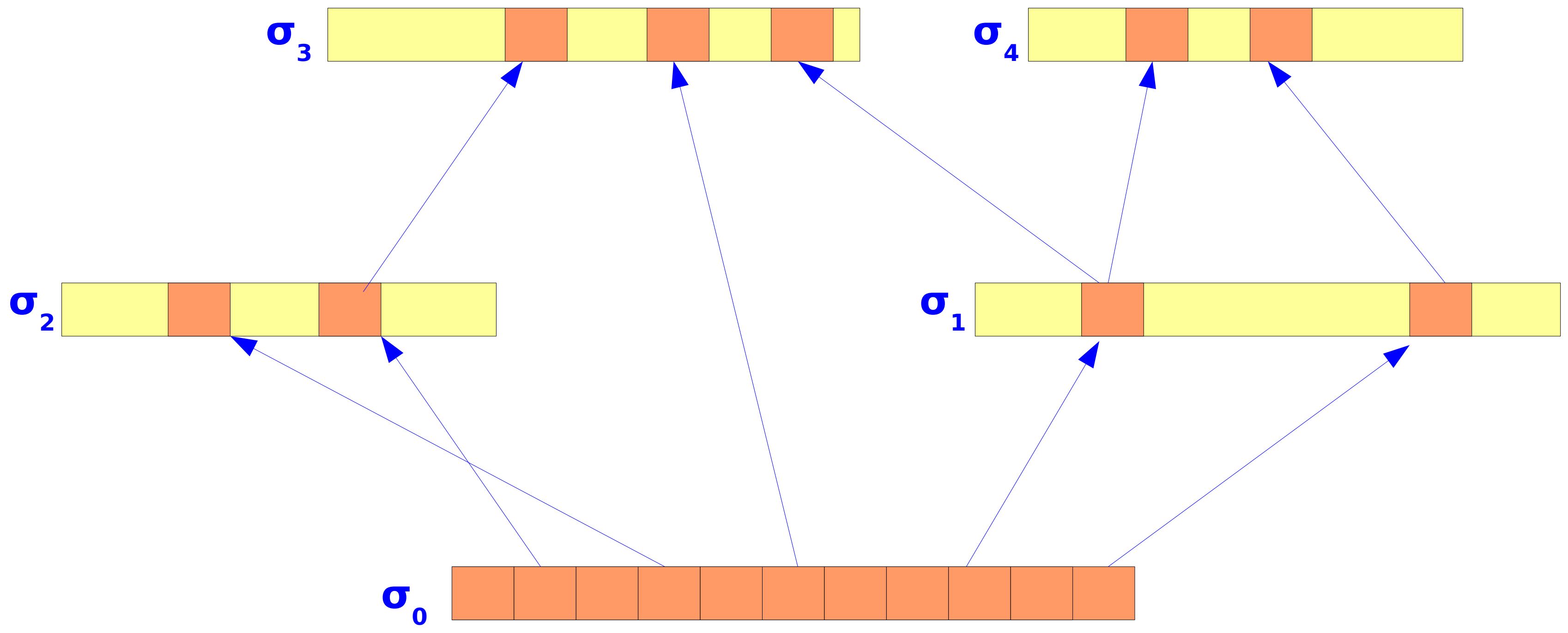
$\sigma_0(v) = v$

Map operation: $\sigma_j(v') = \sigma_i(v)$

Grant operation: $\sigma_j(v') = \sigma_i(v)$
$\sigma_i(v) = \emptyset$

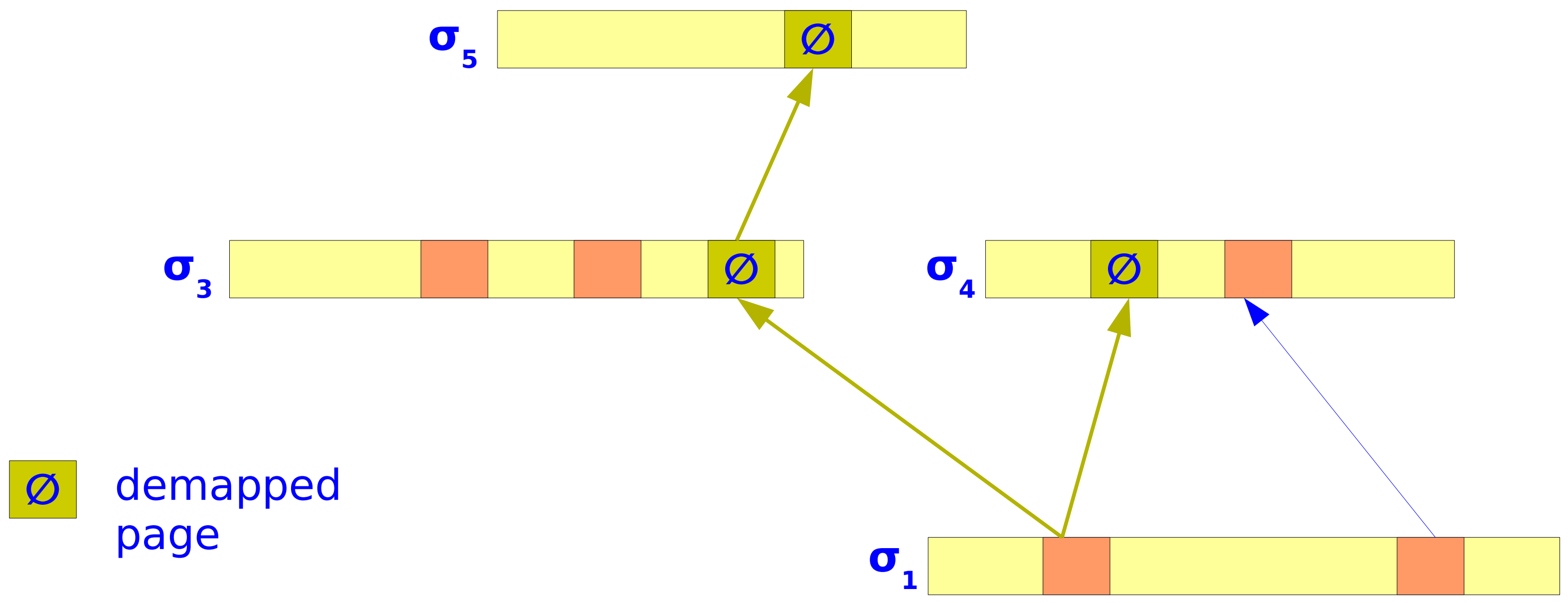
L4 Address Spaces

- A hierarchy of address spaces
  - Each owner may map, to other address spaces, pages that it owns



L4 Address Spaces

- Demap operation
  - An owner may demap one of its page
    - It retains the mapping of that page
  - For all *higher* address spaces that obtained that page
    - Directly or not from that owner
    - They loose that page



Microkernel Architecture

- L4 address space model

$\sigma : V \rightarrow R \cup \{\varnothing\}$

where V is the set of virtual pages
R is the set of available real pages (physical)
$\varnothing$ is the nil page that cannot be accessed

Initial mapping representing the physical memory

$\sigma_0(v) = v$

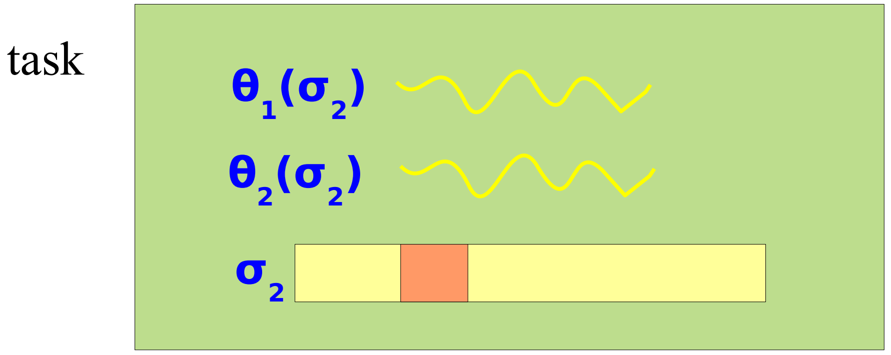
Map operation: $\sigma_j(v') = \sigma_i(v)$

Grant operation: $\sigma_j(v') = \sigma_i(v)$
$\sigma_i(v) = \varnothing$

Demap operation: $\forall$ j,v' such that $\sigma_j(v') = \sigma_i(v)$
$\sigma_j(v') = \varnothing$

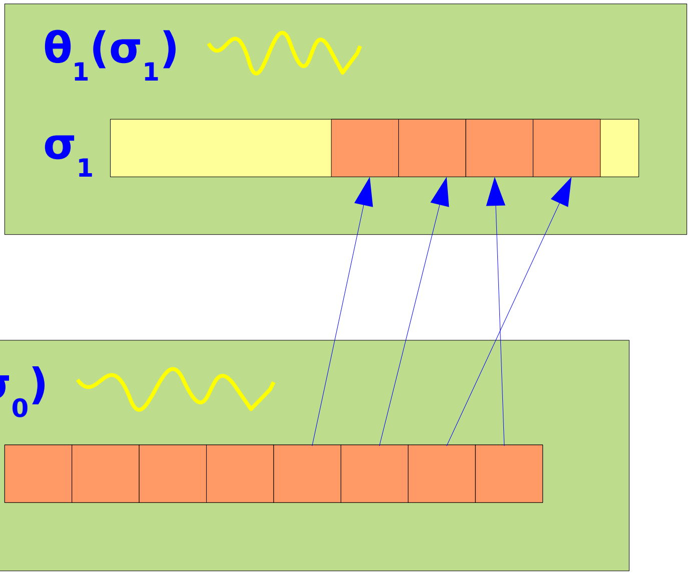
Microkernel Architecture

- L4 tasks
  - Single address space
    - A flat virtual address space (no segments)
  - One or more threads
    - Threads are called owners of the address space
      - Notation: $\boldsymbol{\theta_i(\sigma_j)}$ for a thread owning an address space
    - An address space is created with one owner
  - A pager
    - A thread to receive page fault traps
    - Traps are translated to IPC in L4



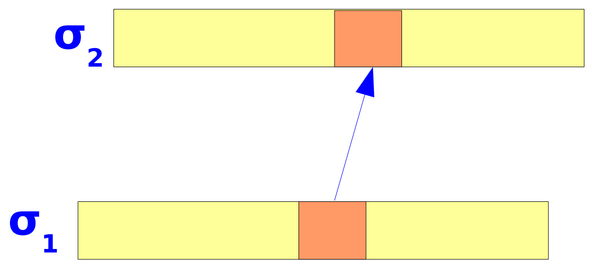
Microkernel Architecture

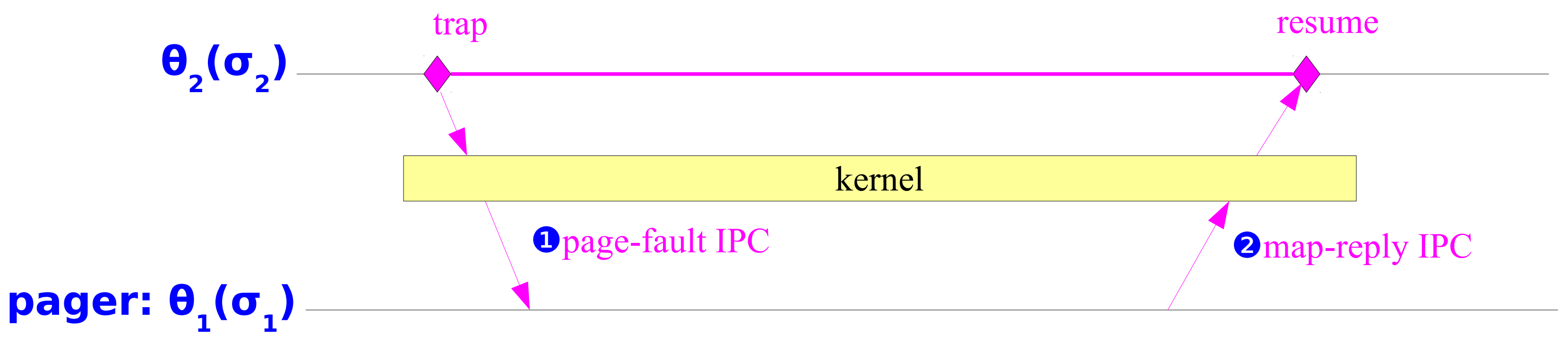
- Illustrating address spaces
  - Initial pager task
    - Simple mapping scheme
    - Maps each page only once, never demaps pages
  - Virtual memory pager
    - Requests a number of physical pages
    - That it manages as a cache



Microkernel Architecture

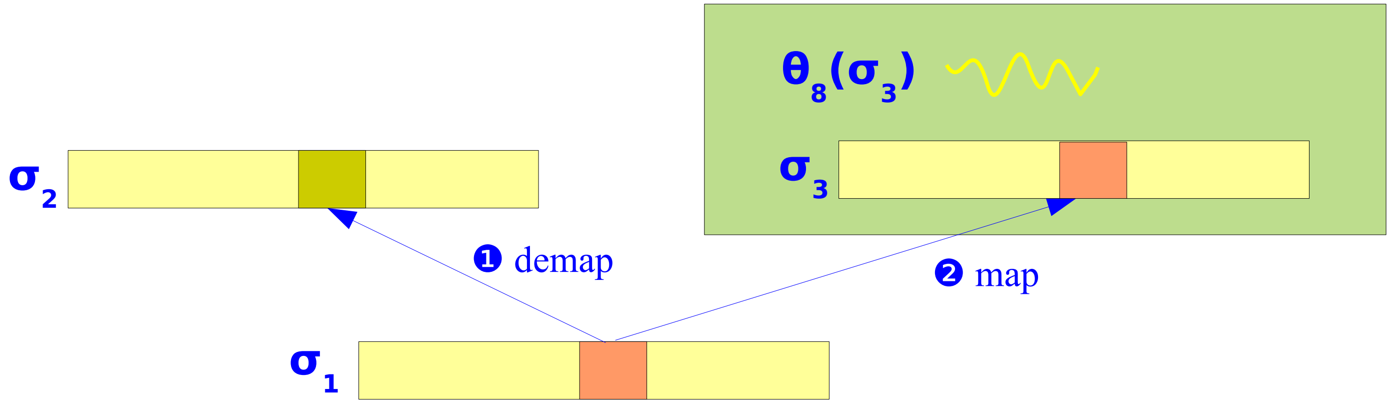
- Illustrating page faults
  - $\theta_2(\sigma_2)$ traps
    - Trap is translated into an IPC to the pager
    - Pager on thread $\theta_1(\sigma_1)$
  - Pager handles the page faults
    - Selects a free page in the cache
    - Loads the content of that page
    - Maps that page





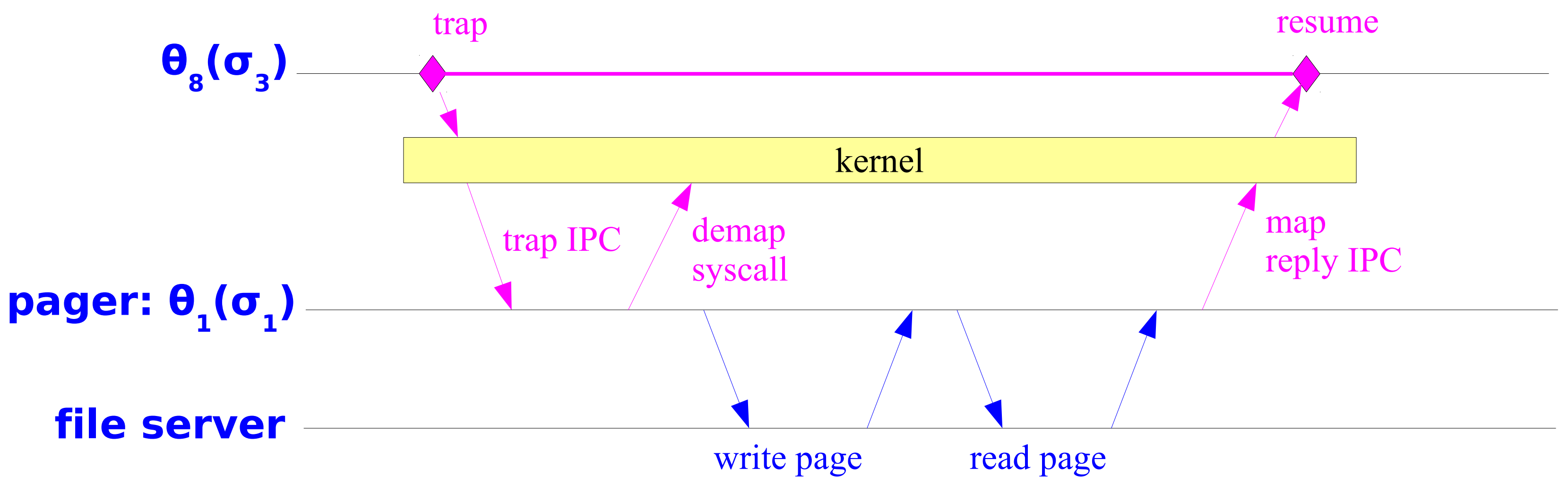
Microkernel Architecture

- Illustrating page replacement
  - $\theta_8(\sigma_3)$ traps
    - Trap is translated into an IPC to the pager
    - No more physical pages are available
  - Page replacement
    - Demap one of its managed page
    - Reuse that page to map it



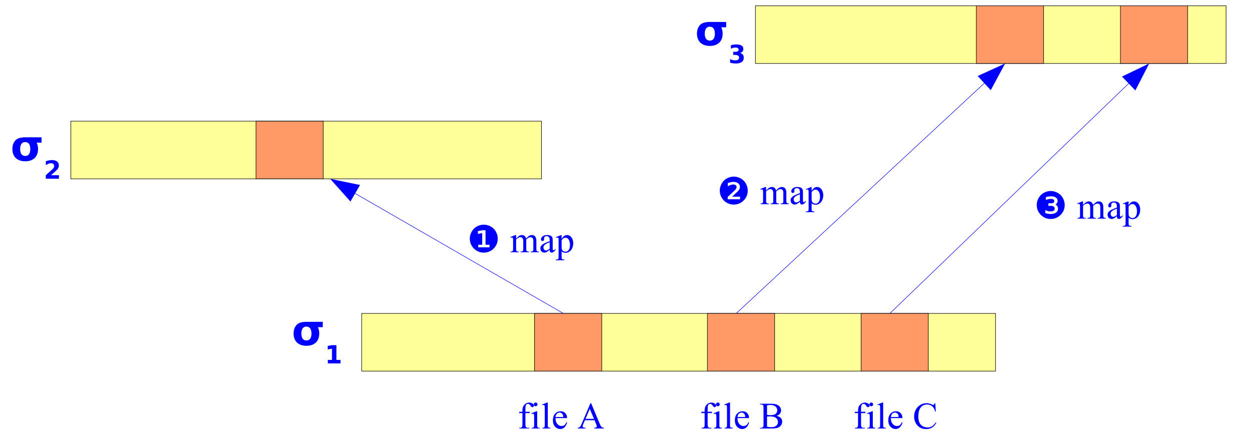
Microkernel Architecture

- Illustrating swapping
  - $\theta_8(\sigma_3)$ traps
    - Trap is translated into an IPC to the pager
  - Pager handles the page faults
    - Write out the cached page and then demap it
    - Read in the new content of the page
    - Maps the page



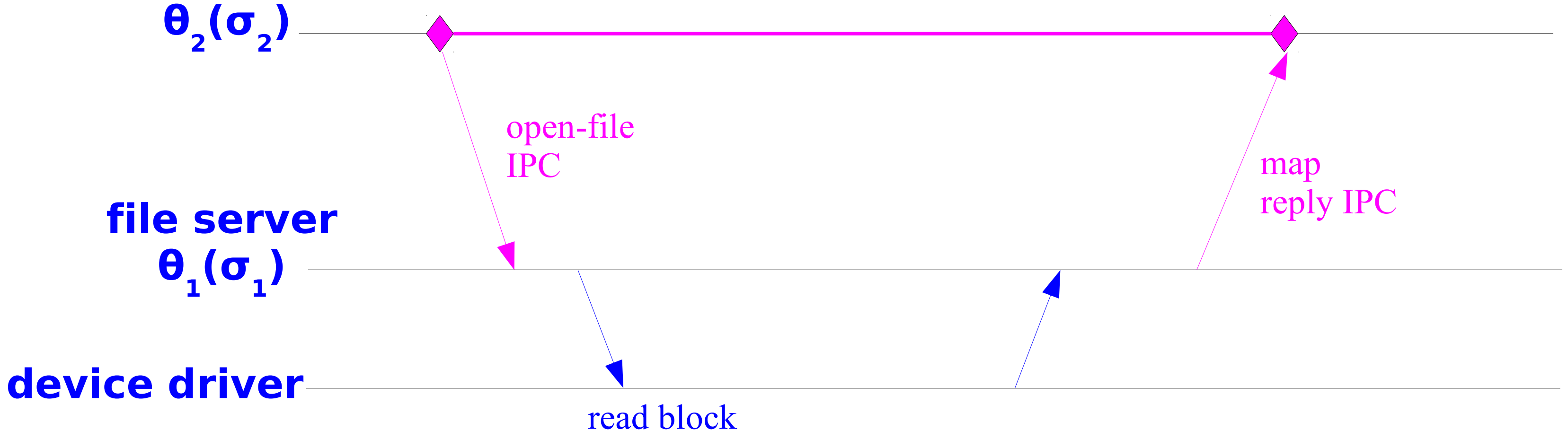
Microkernel Architecture

- Illustrating a simple file server
  - Manages a cache of physical pages
    - Use them as read-write buffers
  - For each client
    - Openned files and current position (I/O block)
    - Current block is mapped in client address space



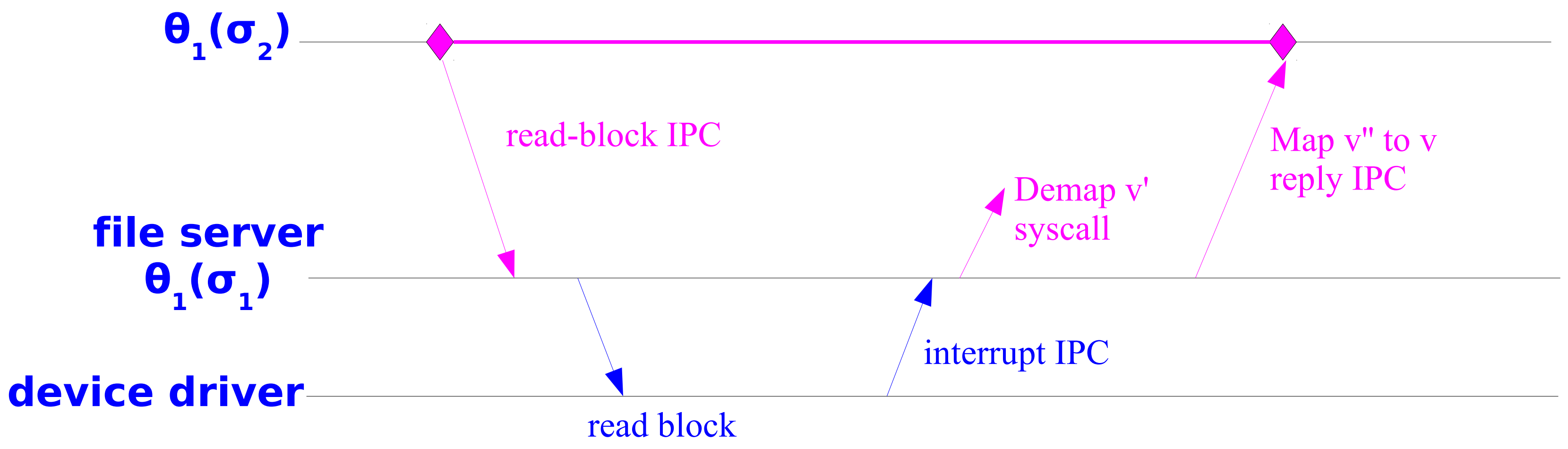
Microkernel Architecture

- Illustrating a simple file server
  - Opening a file (client)
    - Client allocates a buffer (a page v in virtual memory)
    - Calls the file server with a open-request IPC
    - Waits for a reply IPC, a map request: $\sigma_2(v) = \sigma_1(v')$
  - File server
    - Does an internal open of the file
    - Allocates a page v' in its cache and reads in the first block
    - Replies a map IPC, $\sigma_2(v) = \sigma_1(v')$



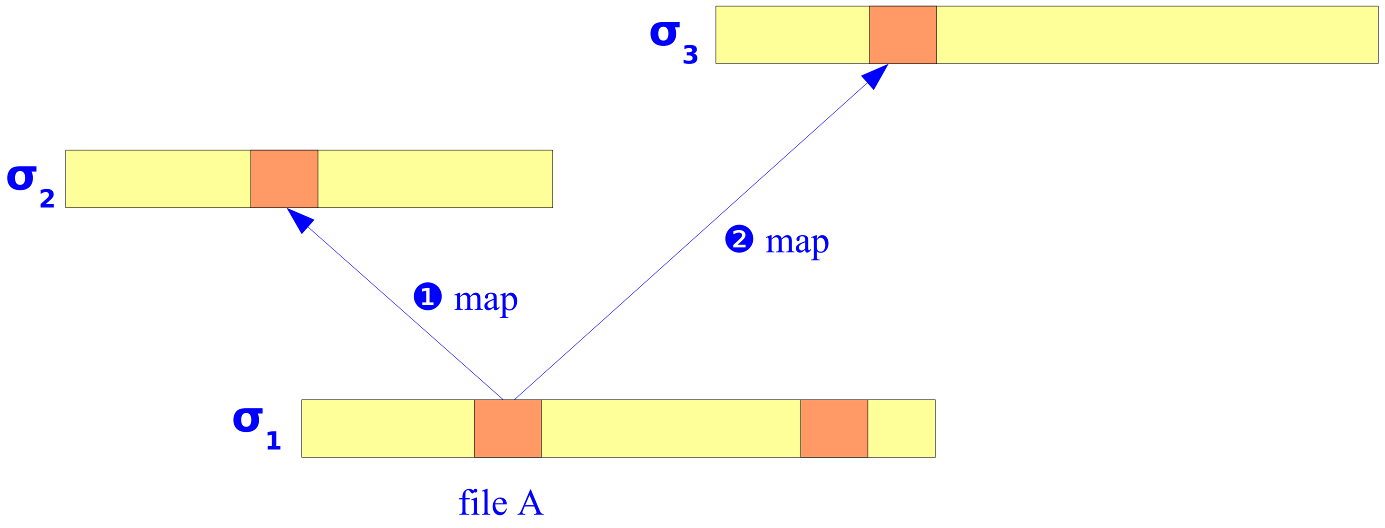
Microkernel Architecture

- Illustrating a file server
  - Read a block, given a block offset
    - Calls the file server with a read-block-at-offset IPC
    - Waits for a reply IPC, a map request **at the same client buffer v**
      - From $\sigma_2(v) = \sigma_1(v')$ to $\sigma_2(v) = \sigma_1(v'')$
  - File server
    - Allocates a new page v'' in its cache and initiate I/O
    - Demaps the previous page v' that held the previous current block
    - Replies a map IPC for the new page holding the new current block



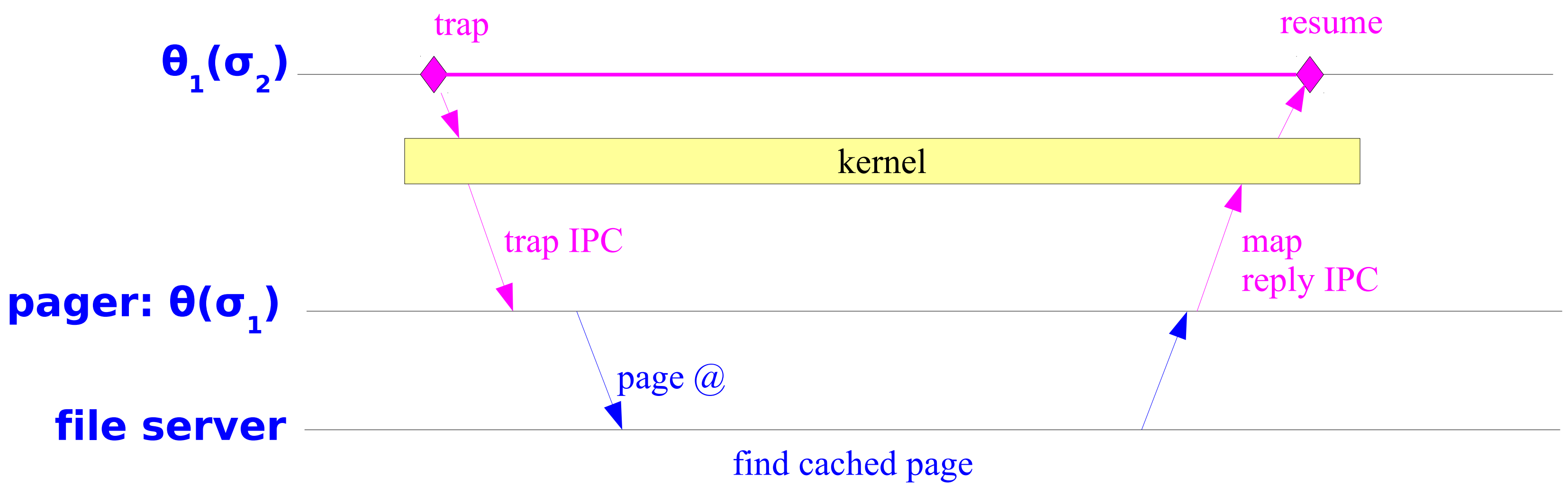
Microkernel Architecture

- Illustrating a file server
  - Concurrently opened files
    - Current blocks may be mapped in different clients
    - Always true for concurrently-openned files smaller than 4KB on most systems
  - Demap challenge
    - Demap system call unmaps $\sigma_i(v)$ from all clients
      - $\forall$ j,v' such that $\sigma_j(v') = \sigma_i(v) : \sigma_j(v') = \varnothing$
    - Page faults will occur, how do we re-establish the mapping?



Microkernel Architecture

- Illustrating a file server
  - A possible solution (others probably exist)
    - Specific address space structure
    - Reserve a file-system buffer region
  - File-system-aware pager
    - Handles the page fault if not in the file-system buffer region
    - Otherwise forwards the page trap to the file server
    - File server finds its corresponding page buffer and makes the map reply



References

- Architectural debates
  - Tanenbaum/Torvalds debate part 1 (1992)
    - http://www.oreilly.com/catalog/opensources/book/appa.html
  - Tanenbaum/Torvalds debate part 2 (2006)
    - http://www.cs.vu.nl/~ast/reliable-os/
    - http://www.coyotos.org/docs/misc/linus-rebuttal.html
- On micro-kernels versus virtual machine monitors
  - S. Hand et al., Are virtual machine monitors microkernels done right?,
  - Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X), 2005.
  - G. Heiser et al., Are virtual machine monitors microkernels done right? ACM Operating Systems Review, Volume 40, Issue 1, 2006.
  - T. Roscoe et al., Hype and virtue, Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI), 2007.

References

- Linux kernel software engineering study
  - Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. IEE Proceedings: Software, 149:18–23, 2002.
- Origins of microkernels
  - Per Brinch Hansen. The nucleus of a multiprogramming operating system. Communications of the ACM, 13:238–250, 1970.
  - W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system.Communications of the ACM, 17:337–345, 1974.

References

- Mach
  - M. Acetta et al., Mach: A new kernel foundation for UNIX development. Proceedings of the 1986 Summer USENIX Technical Conference, 1986.
  - David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an application program. In Proc. 1990 Summer USENIX Annual Technical Conference, pages 87–95, June 1990.
- L4
  - J. Liedtke, On µ-Kernel Construction. Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP), 1995.
  - J. Liedtke, Toward Real µ-kernels. Communications of the ACM, 39(9), pp. 70-77, 1996.
  - Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of µ-kernel-based systems. In Proc. Symp. on Operating Systems Principles, pages 66–77, St. Malo, France, October 1997.
  - Gerwin Klein et al. seL4: Formal verification of an OS kernel. Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT, USA, October, 2009