Micro-Kernels
Part One

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.”

There are no reasons to believe that it is any different for other kernels

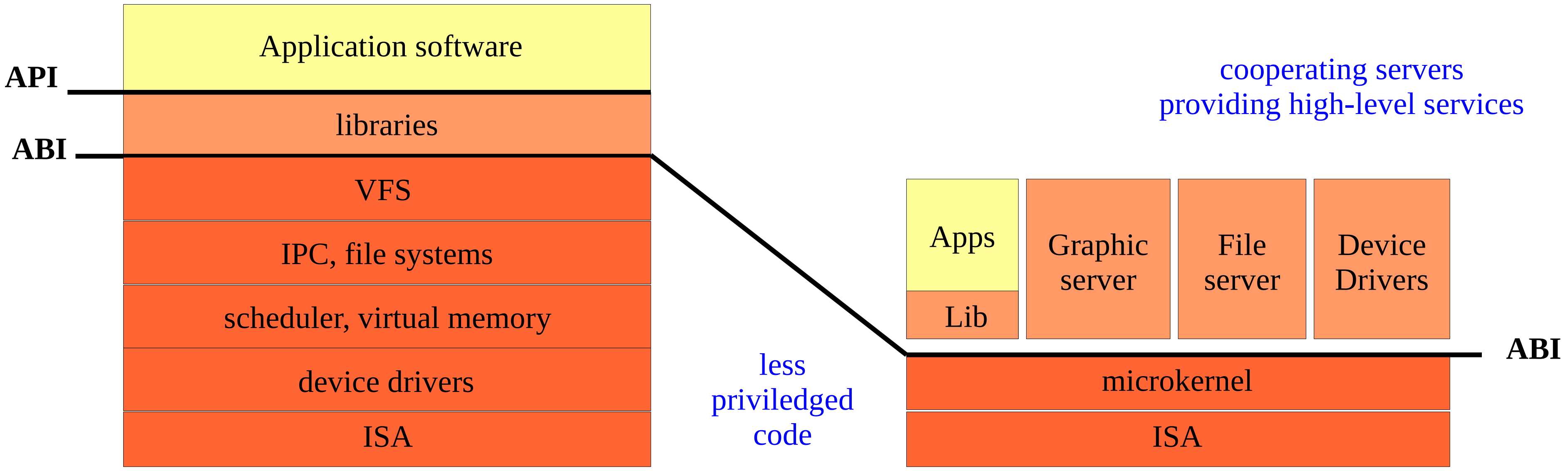
Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. IEE Proceedings: Software, 149:18–23, 2002.

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)



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

Questions – Your mind at work...

Warm-up discussion:

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

Let's apply it to sockets:

socket_t accept(port_t port);

socket_t connect(host_t host, port_t port);

int **read**(socket_t sock, uint8_t *bytes, uint32_t nbytes);

int **write**(socket_t sock, uint8_t *bytes, uint32_t nbytes);

void **close**(socket_t sock);

Questions – Your mind at work...

Assuming the following socket API as a starting point:

socket_t accept(port_t port);

socket_t connect(host_t host, port_t port);

int **read**(socket_t sock, uint8_t *bytes, uint32_t nbytes);

int **write**(socket_t sock, uint8_t *bytes, uint32_t nbytes);

void **close**(socket_t sock);

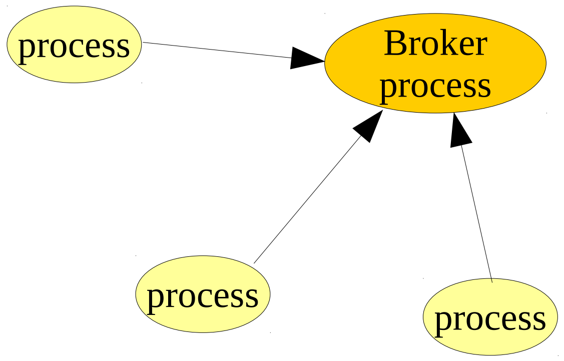
=> Propose a design for socket-based IPCs

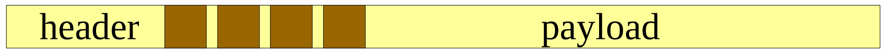
- Pay particular attention to buffer management
- Discuss virtual vs physical addresses
- Discuss ownership
- Discuss thread safety
- Discuss L1/L2 cache management

=> Apply it to handling traps

Microkernels - Broker

- IPC challenge
  - How does it all start?
  - How do you get a remote port?
- A broker is necessary
  - Requires an initial port, given to all processes
  - A port can be bound to a name
- Messages must be able to contain ports
  - Ports may be a simple global identity (e.g. IP and port number) or a capability
  - Port marshalling impacts the message format





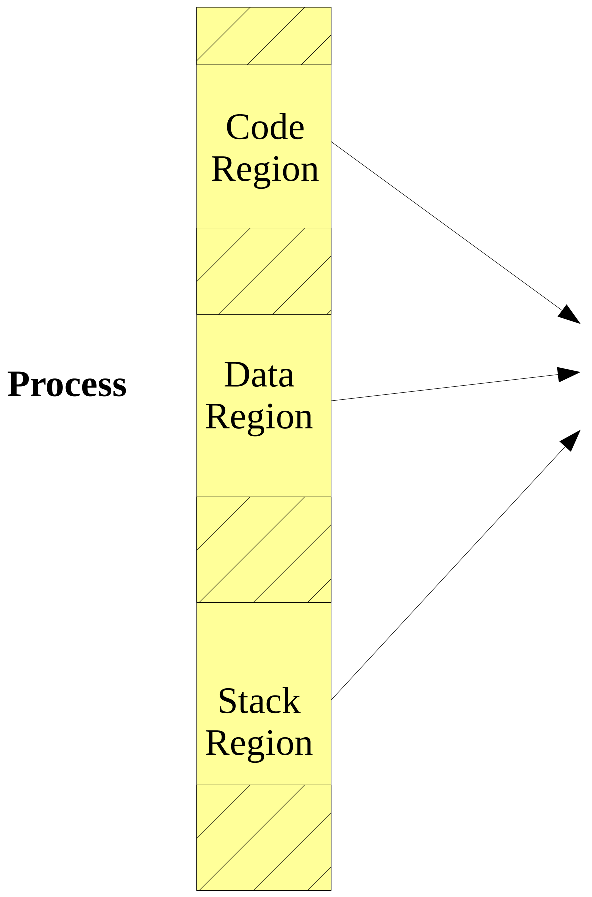
embedded ports

Microkernels – Capabilities

- Capabilities
  - Tokens representing priviledges
  - The *right* to execute an action on an entity
    - A right to send or receive messages
    - A right to map a page to a process
    - A right to create a process or a thread
  - 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

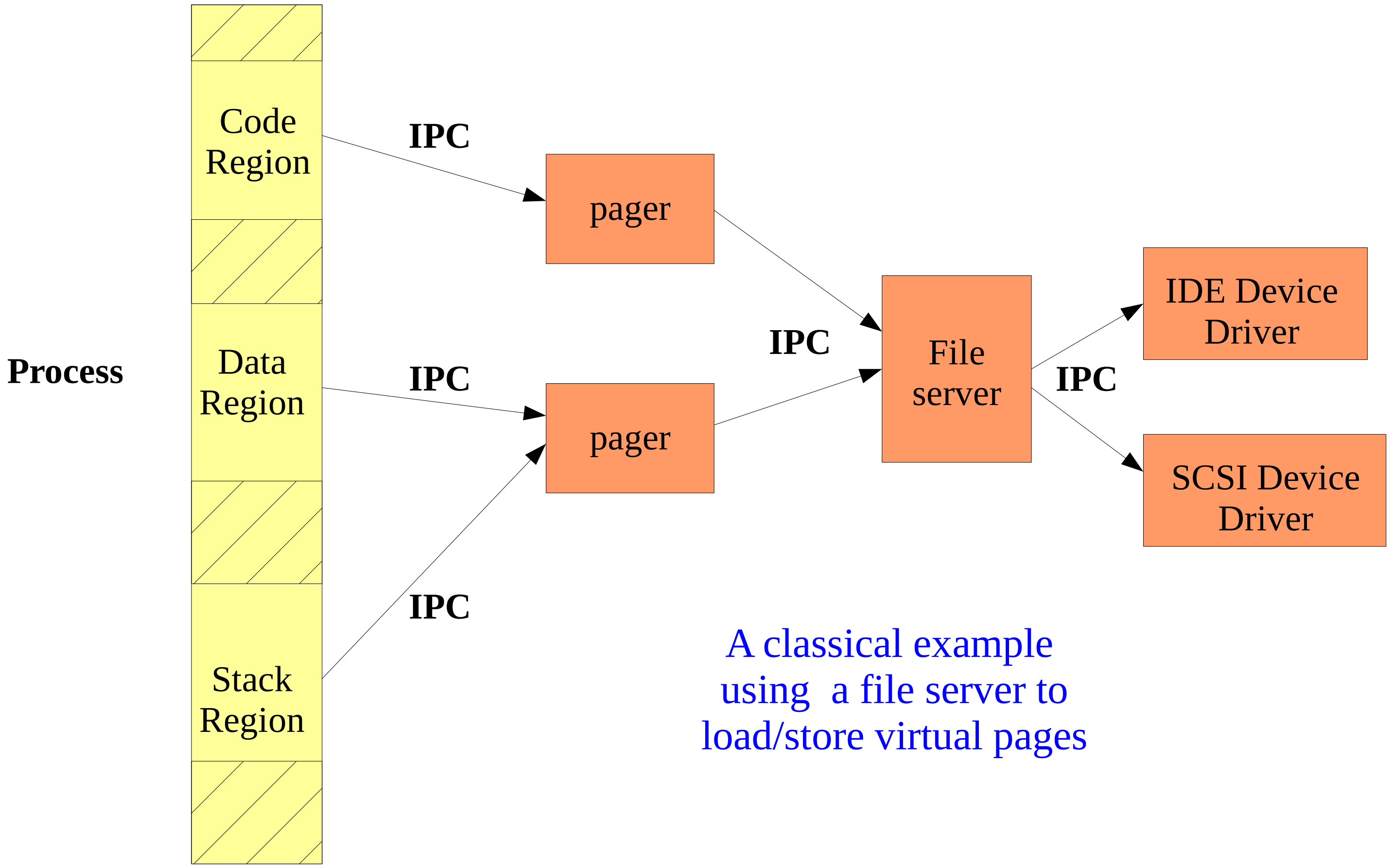
- Processes as address spaces
  - A virtualized memory, isolated by hardware
  - Mapped memory regions, managed by user-level pagers



Who is responsible for handling page faults within mapped regions?

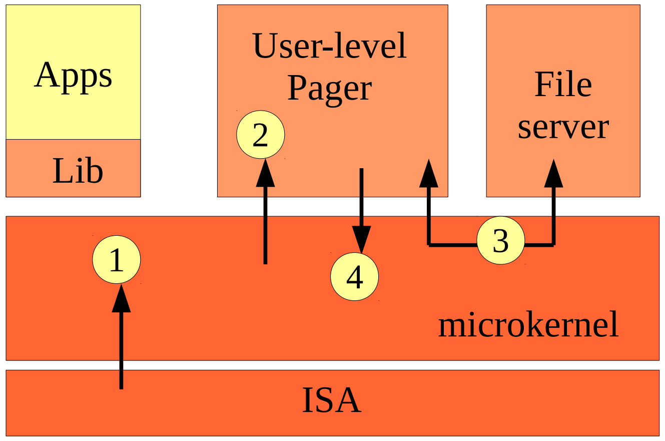
Microkernel – Processes

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



Microkernel – 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 → trap to the micro-kernel
  - (2) Upcalls (IPC) the corresponding external pager
  - (3) External pager finds the page on disk
  - (4) Provide the page to the kernel



Questions – Your mind at work...

Given our socket-based IPC design.

Assuming a libC, wrapping the file server, with the usual open/close/read/write calls.

Propose a design for the interactions between pagers and the kernel

- Relaying a page fault to the external pager

- Providing the faulted-in page to the kernel

Discuss the corresponding internals

- MMU management
- Scheduling

Questions – Your mind at work...

Assuming a libC, wrapping the file server, with the usual open/close/read/write calls.

Given our socket-based IPC design.

Given our design for the kernel/pager interactions.

Propose a design for the creation of a process from an ELF executable.

- How is a process created? Killed?
- How are threads created within a process?
- How are regions created and associated with external pagers?

Discuss the syscalls or messages that are required.

Microkernels – Multi-Threading

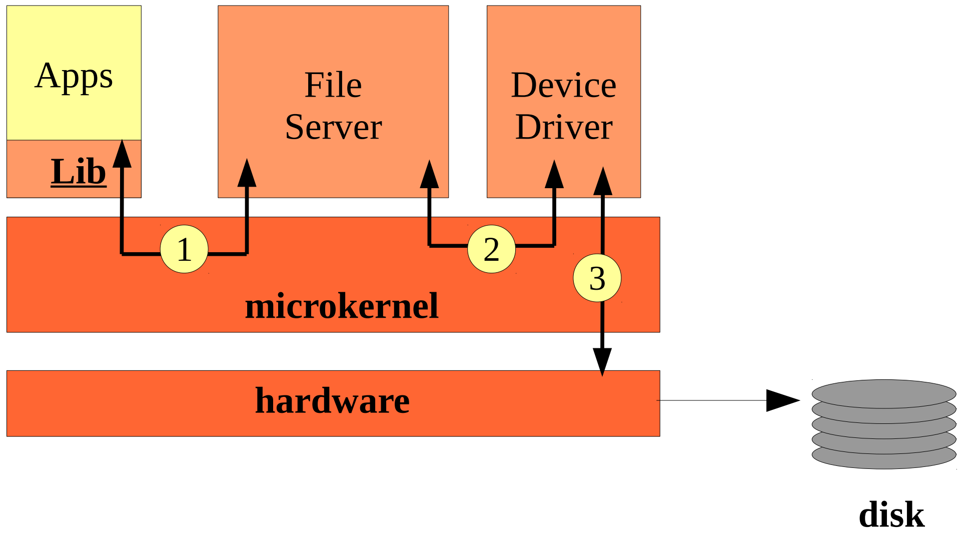
- *Quite regular* threads
  - Execution flow, local to a process
  - Often has thread-local variables
  - Usually execute at a certain priority level
- *Quite regular* scheduling
  - Microkernels embeds one or more scheduling policies
  - Often based on priorities and time slices
  - Executing thread can yield
  - Executing thread can send or receive messages (blocking or not)
  - Synchronization mechanisms are necessary

Microkernels – The libC

- Discussing the libC re-design
  - Providing a libC is of course possible
  - Providing the usual concept of streams (files, sockets)
  - With the typical open/read/write/close API
  - Potentially available as blocking and non-blocking

DO NOT CONFUSE WITH YOU IPC DESIGN!

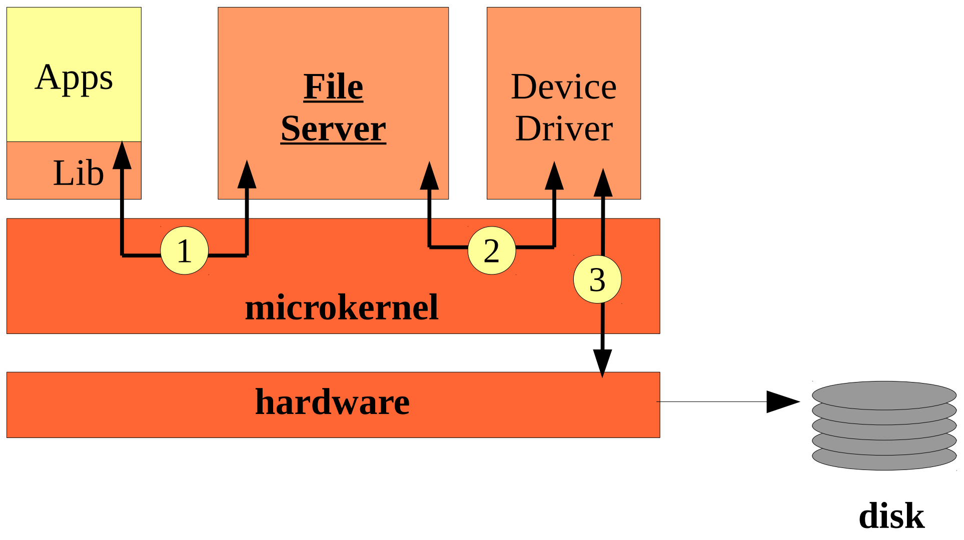
1 How do we implement these operations with IPC messages?



Microkernels – File Server

- File server process
  - Manages different file system types (ext2, ext3, etc.)
  - Manages the file cache (disk blocks are available or not)
  - Need to read missing blocks from disks
  - Need to write dirty blocks to disks
  - Need a block replacement strategy (a cache has a limited size)

2 How do we implement these operations with IPC messages?



Questions – Your mind at work...

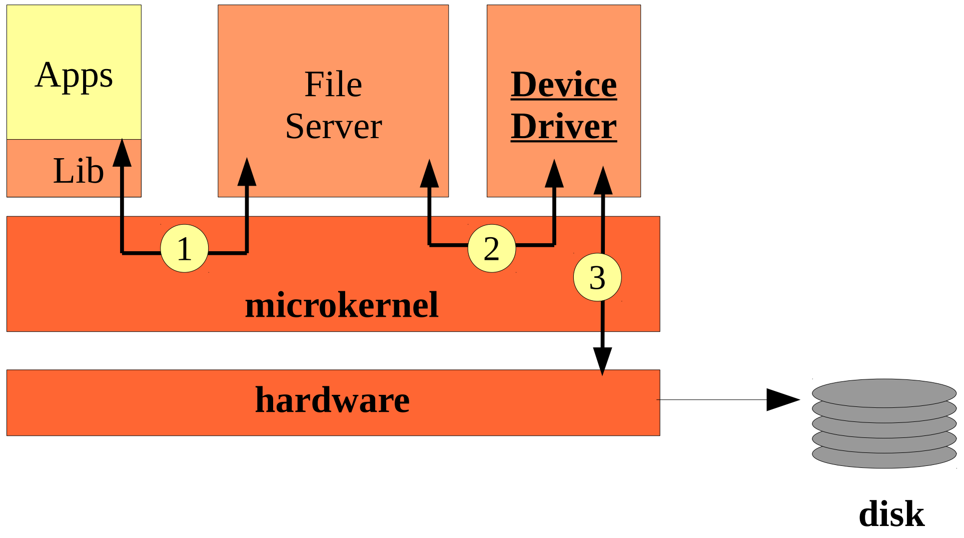
Design the internals of the libC for file operations

- Operations are the following: open, read, write, seek, close.
- Assume a file cache in your server.
- Discuss in details the IPC dialog between the libC in a client process and your file server
  - How many ports?
  - Which IPC messages?
  - Details both their semantics and contents.
- Pay particular attention to buffer management
  - On both sides, libC buffers and file server buffers
  - Discuss ownership
  - Discuss thread safety
  - Discuss L1/L2 cache management
- Do not forget that you can use IPC messages but also external pagers
- Do not forget that you must be able to clean up when a process is killed

Microkernels – Device Driver

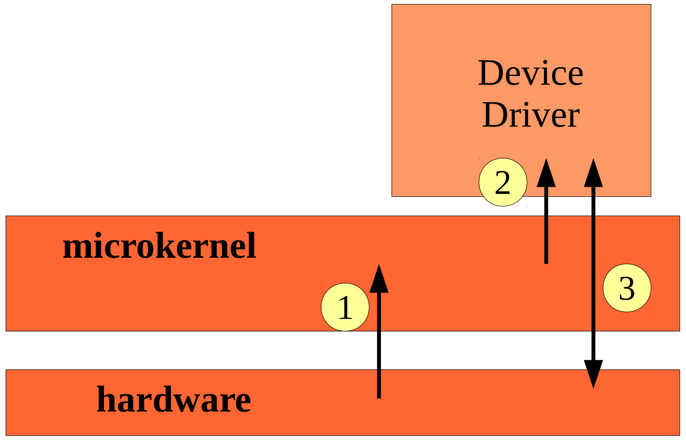
- Device driver process
  - Host one or more regular drivers
  - Require to have access to memory-mapped I/O registers
  - Require to have access to interrupts

3 How do we implement these operations with IPC messages?



Microkernels – Device Driver

- MMIO registers
  - Simply map the physical pages into the virtual space
- Interrupt handling
  - The microkernel captures the interrupts 1
  - Forwards it to the concerned driver, generating an IPC 2
- The device driver handles the interrup 3
  - Potentially dialoging with the device through mmio registers
  - Interrupt may indicate the end of the DMA transfer



Questions – Your mind at work...

Design the device driver framework

- Remember that we have character and block devices
- Discuss the IPC messages for both types of device drivers
- Discuss the IPC messages to forward interrupts
- Discuss how mmio registers are manipulated by device drivers
- Discuss in/out rings of buffers with respect to virtual vs physical memory

What about safety?

- Is safety really improved by user-level device drivers?

Questions – Your mind at work…

What about overall performance?

Compare our micro-kernel design with regular linux

Key points to discuss:

- Discuss syscalls
- Discuss memory-to-memory copies
- Discuss latencies
- Discuss context switches
- Discuss cache efficiency, for both data and instruction caches
- Discuss TLB misses and MMU page faults

Microkernels – Success or Failure

- Past spectacular “failures”
  - Most notorious: IBM Workspace OS
  - Also GNU Hurd
- Today's spectacular “successes“
  - Apple Mac-OS based on Mach CMU
  - GreenHills (world leader in RTOS)
  - Nicta (1.5 billions smart phones use L4, and still counting...)

Not clear if microkernels are a success or a failure...
Not clear what it means, are we considering this with respect to
- Goals?
- Commercial?
- Technical?

The road was long, twisted, and bumpy, and good new,
it is streching far ahead...

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
- Goal – running multiple OS personalities concurrently
  - Sharing personality neutral services (PSNs)
  - Applications could use services from multiple OSes

IBM Workplace OS

- Very ambitious project (budget and manpower)
  - Plagued by internal problems
  - Too much focus on portability, not enough on personalities
  - **Personalities require changing/porting existing operating systems**
- Performance analysis
  - **Microkernel performance not mentioned as a problem**

But what was the research community saying back then?

Microkernels – Research Analysis

- 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)
- Overall results
  - Some applications performed as well or slightly better on microkernels
  - Some showed peak degradation of 50% to 66%

But it is hard to discuss and appreciate micro-benchmark numbers.

Clearly depends on how much applications use system calls.

→ **Read** Bershad et al, "*Using Microbenchmarks to Evaluate System Performance*"

Microkernels – Research Analysis

- 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 success

Chorus-OS (INRIA/Chorus SA) disappeared.
XNU (Next Inc.) adopted Mach 2.5 (Carnegie Mellon University).
Mach 3.0 is now part of the Mac-OS and iOS.

Discussing Performance

- Benchmarks used
  - Regular applications as sed, egrep, yacc, gcc or compress
  - Micro-benchmarks on the network server and file servers
- Were they really representative?
  - Mostly I/O bound benchmarks, stressing the use of the IPC
  - Notice a total lack of GUI-based benchmarks
- Would they be really representative today?
  - Most applications now run out of main memory
  - Often even from L1/L2 caches
  - What about the evolution of the memory wall?
  - What about the evolution towards heaving multi-threading?

Discussing Performance

- What about more recent applications?
  - Such as open/libre office?
  - Or a web brower with complex pages?
- These are more CPU intensive and much less I/O bound
  - A lot of processing goes in pretty printing, spelling, etc.
  - They are GUI intensive also

Wasn't graphics based on IPC-like communication?

Most graphic applications are quite IPC intensive

X11 worked flawlessly for more than 30 years...

Was it the asynchronous versus synchronous nature of IPCs?

Discussing Performance

- Was 50-60% degradation a real showstopper?
  - It was considered as one, by researchers themselves
  - The industry was not to accept something researchers found a failure
- But what about Moore's law
  - Hardware was twice faster every year back then
  - What about the advent of multi-core?
  - The evolution of the memory wall was known and ignored
- What about design/implementation choises?
  - Servers and libraries are large piece of codes
  - How good/suited is their implementation?

What should we think about all this?

Discussing Adoption

- The overhead can't justify failure
  - Did you notice that Java failed in its early years because it was slow?
  - What about newer versions of Microsoft Windows?
  - What about Ubuntu Unity versus faster ligther window managers?
- Missing a killer usecase does it
  - Flexibility and modularity are not obvious market growth opportunity
  - Safety improvements were not real enough
- Natural resistance to change
  - Strong resistance to any change to existing complex software stacks
  - Required massive changes to Linux code base
  - Not invented here syndrom

Discussing – Adoption by Apple

- XNU was originally developed at NeXT (Steve Jobs)
  - Mach kernel (Carnegie Mellon University) version 2.5
  - Components from 4.3BSD and an Objective-C API for device drivers
  - Open-sourced as Darwin
- Why did NeXT adopt Mach?
  - Modularity is often quoted
  - Isolation from device driver faults
  - Obviously because it was faster than starting from scratch...

Discussing – Adoption by Apple

- Apple bought XNU in 1996
  - Integration in OS-X, moving to Mach 3.0
  - Device drivers written in C++
  - Later use in iOS (not all sources agree on this)
- XNU is a hybrid kernel
  - Core components are back in the kernel (network, storage, USB, etc.)
  - Others components as userland servers
  - Some device drivers are userland as well

So? What can we conclude?

Microkernels have not died away, quite the contrary...

But it seems that performance is still an issue for the Mach family

Note: Windows NT is also a hybrid kernel

Mach Readings

You have a package of papers that **you must read**.

Golub et al, *"Unix as an Application Program"*,
→ A great overview of how to implement Unix as a userland server.

Julin et al, *"Generalized Emulation Services for Mach 3.0, Overview, Experiences, and Current Status"*,
→ Discusses the emulation of operating systems as userland services
It is great read to better understand the challenges of breaking up an operating systems into services.

Tevanian et al, "*Mach Threads and the Unix kernel, The Battle for Control*",
→ Discusses co-routines versus kernel threads.
It is a great read to better understand threads.
It provides a historical perspective as well.

Bershad et al, "*The Increasing Irrelevance of IPC Performance for Microkernel-based Operating Systems*"
→ Discusses the IPC overheads

L4 Readings

You have a package of papers that **<u>you must read</u>**.

Jochen Liedtke, On μ-kernel Construction
15th Symposium on Operating Principles (SOSP),1994

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

Jochen Liedtke, *Improving IPC by Kernel Design*,
14th Symposium on Operating Principles (SOSP),1993

Operating System Research Group,
L4Env – An Environment for L4 Applications
Technische Universität Desden, 2003