OS Models: Microkernels, Exokernels, and others

This week:
OS kernel models

Milestone 2 continues…

Last week's lecture: threading models

- How does the OS implement processes/threads?

This week's lecture: OS structure

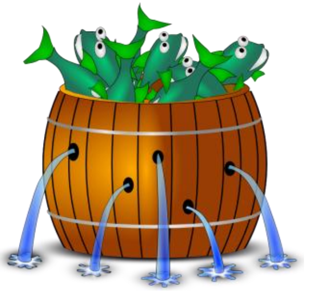
- How do processes/threads implement the OS?

OS ARCHITECTURE

What is "OS Architecture"?

- Coarse-grained structure of the OS
- How the complexity is factored
- Mapping onto:
  - Programming language features
  - Execution environment presented to applications
  - Address spaces
  - Hardware protection features (rings, levels, etc.)
  - Execution patterns (subroutines, threads, coroutines)
  - Hardware execution (interrupts, traps, call gates)

Architectural Models

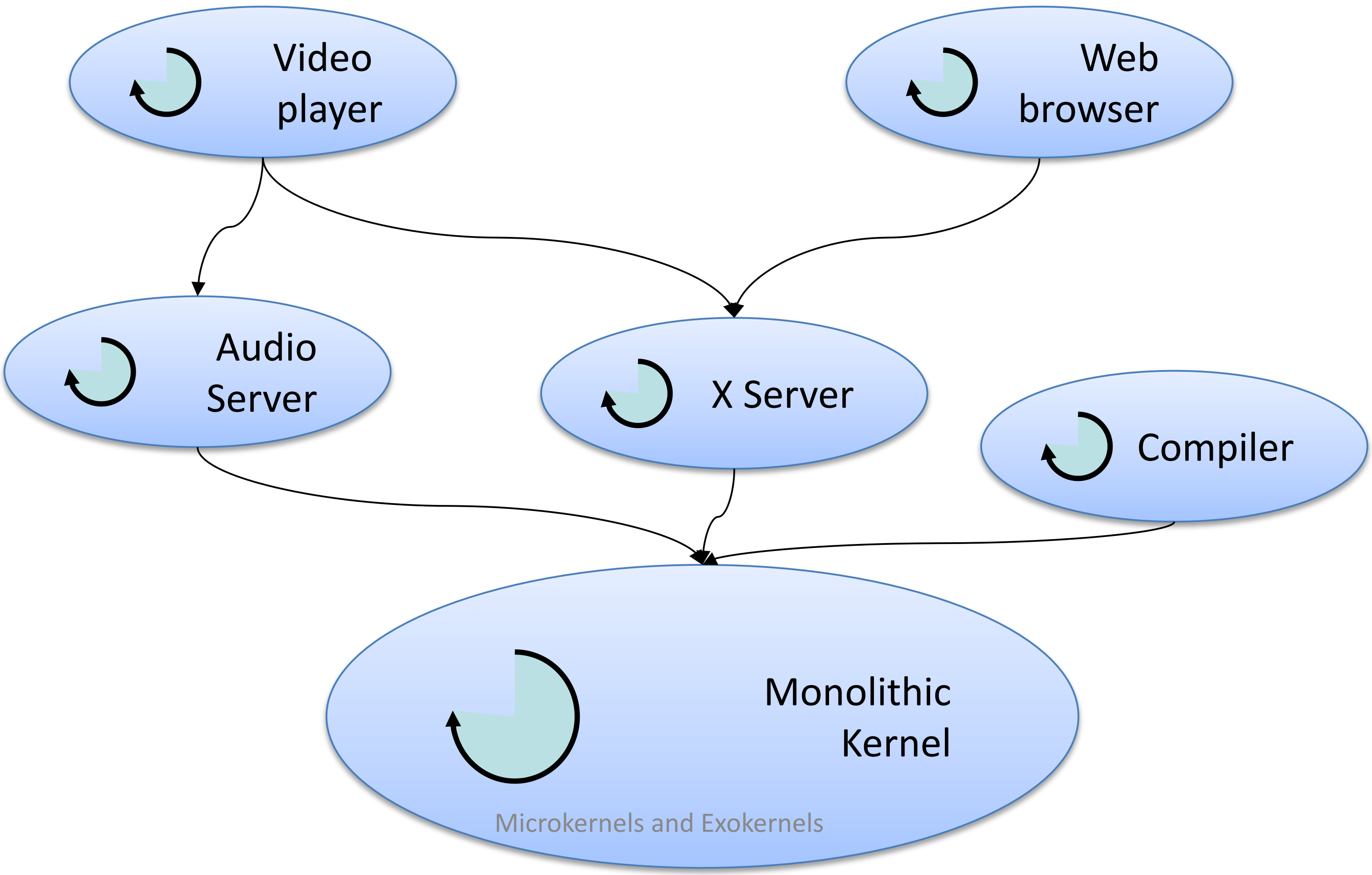


- There are many, and they are **models**!
  - Idealized view of how the system is structured
- Real systems always entail compromises
  - Hard to convey $\Rightarrow$ it's good to build a few
- Think of these as tools for thinking about OSes
  - Each model attacks a particular problem
  - Often address problems in previous systems
  - We'll use *scheduling* as a motivating example

Monolithic Kernels

- Examples: Unix [Thompson, 1974], VMS, Windows NT+
- Hardware enforces user vs. kernel mode
- Kernel enforces or provides:
  - All shared services
  - All device abstraction/multiplexing: threads, address Spaces, devices
  - Protection domains
- Service via **syscall**

Server-Based Monolithic OS



Approaches to tackling OS complexity

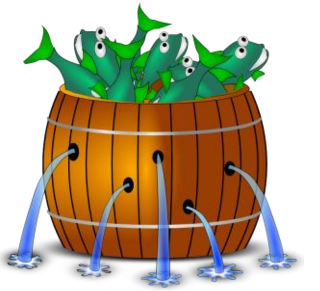
- Classical software engineering approach: modularity
  - Relatively small, self-contained components
  - Well-defined interfaces
  - Enforcement of interfaces
  - Containment of faults
- Doesn’t work with monolithic kernels
  - All kernel code executes in privileged mode
  - Faults aren’t contained
  - Interfaces cannot be enforced
  - Performance takes priority over structure

Multiplexing Systems

- Examples: Cedar [Swinehart et al., 1986], TinyOS [Hill et al., 2004], Oberon, Singularity [Hunt and Larus, 2007]
- Hardware enforces time multiplexing
  - Interrupts, Threads (Cedar)
- Language guarantees modularity
  - Module calls
  - Type safety
- Service via **call/coroutine**

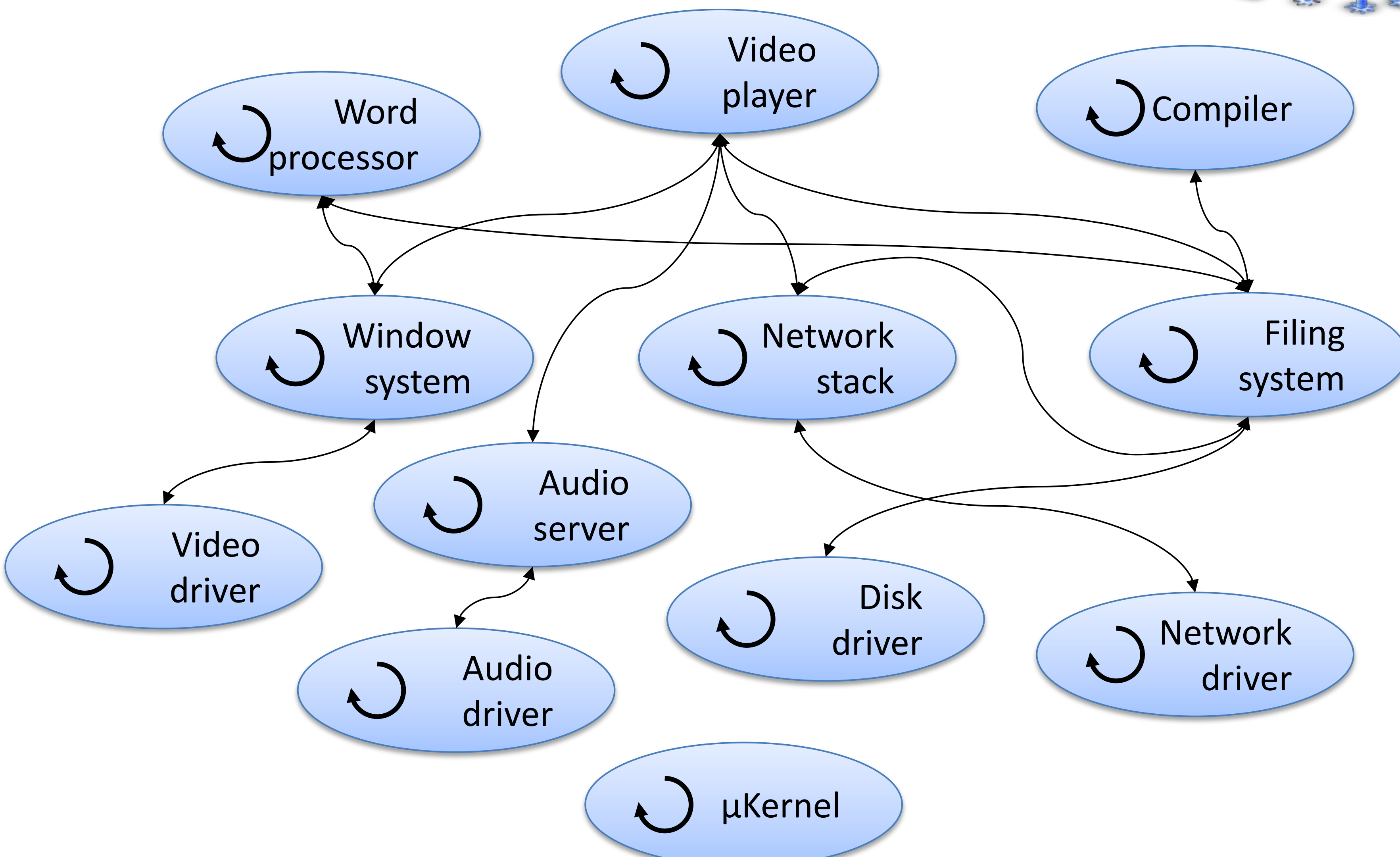
Protection-Based Systems

- Examples: KeyKOS [Bromberger et al., 1992], Pebble [Bruna et al., 1999]
- Hardware/Kernel enforces protection domains
- Scheduling etc. in user space
- Aimed at:
  - High security (very small TCB)
  - Embedded systems (highly configurable)
- Service via **capability invocation**



EXOKERNELS

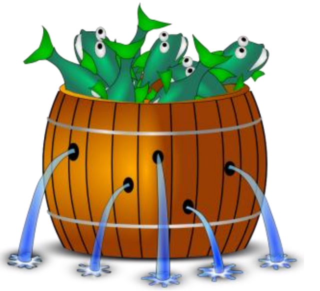
Microkernel OS



Unpredictable Performance

- Lots of (implicitly) shared resources
  - Each with contention
  - Server/Kernel has no application knowledge
  - Long dependency chains
  - No global scheduling algorithm can help
  - High IPC performance doesn’t solve this
- Relevant early ‘90s work:
  - “SVR4 scheduler unacceptable” paper
  - Processor Capacity Reserves (complexity!)
  - Resource Containers (ignore the problem!)

Exokernels



- Kernel multiplexes HW **once**
  - No abstraction!
  - All other functionality in userspace **libraries**
  - Unlike microkernels, where this in servers
  - “LibraryOS” concept
- Enables:
  - Performance isolation between applications
  - Application-specific policies

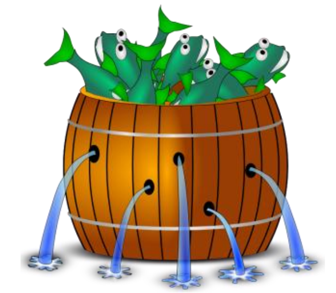
Two Exokernel Systems

Two different systems. Two different motivations:

1. Complexity, adaptability, performance
   $\Rightarrow$ Aegis [Kaashoek et al., 1997]
2. QoS crosstalk
   $\Rightarrow$ Nemesis [Leslie et al., 1996]

- Similar approaches:
  – Exterminate OS abstractions
  – Move code into the application $\Rightarrow$ library OS

Aegis motivation
Exterminate all OS abstractions!



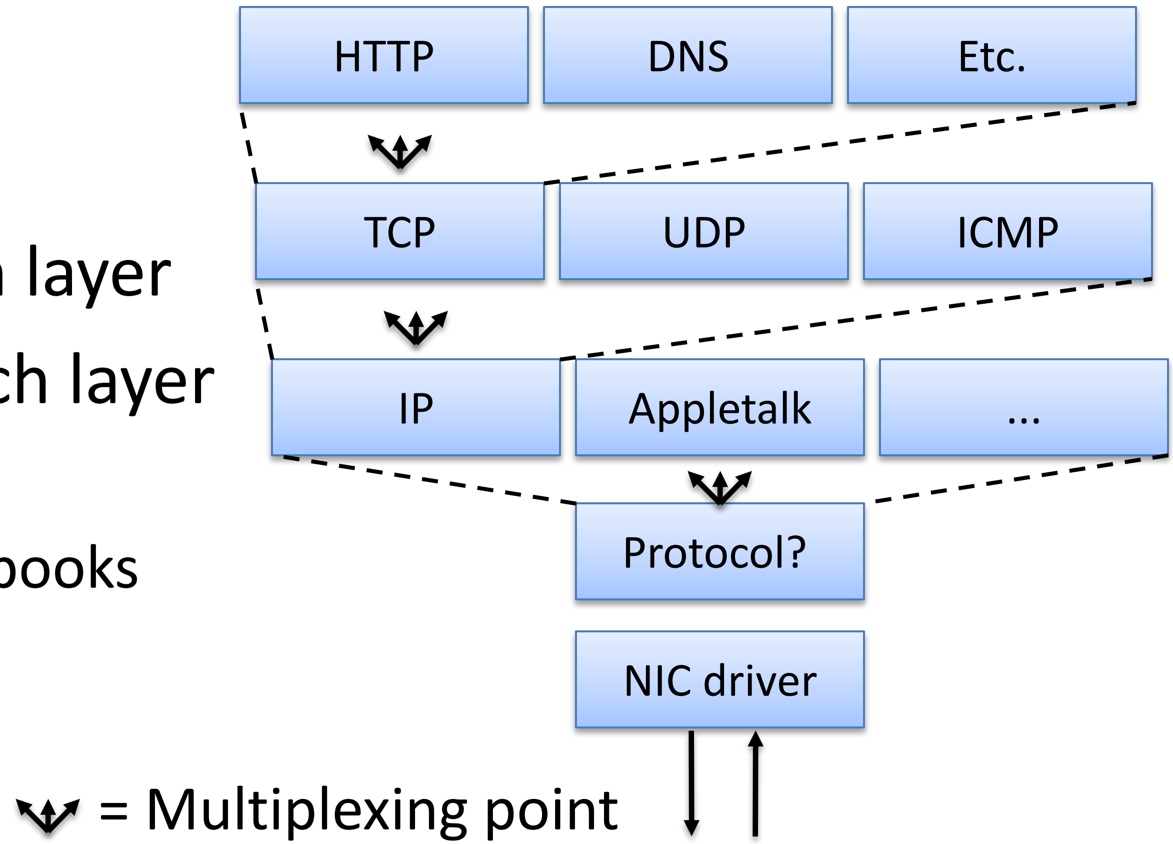
- A traditional OS or a microkernel such as L4:
  - Multiplexes physical resources
  - Shared, secure access to CPU, memory, network, etc.
  - Abstracts the same physical resources
    - Processes/threads, address spaces, virtual file system, network stack
- Multiplexing is required for security

. . . but why should an OS abstract what it multiplexes?

Nemesis Motivation: Eliminate QoS Crosstalk

Consider a network stack:

- Layered protocol implementation
- Multiplex at each layer
- Conceptually, each layer is a process
  - c.f. early Comer books

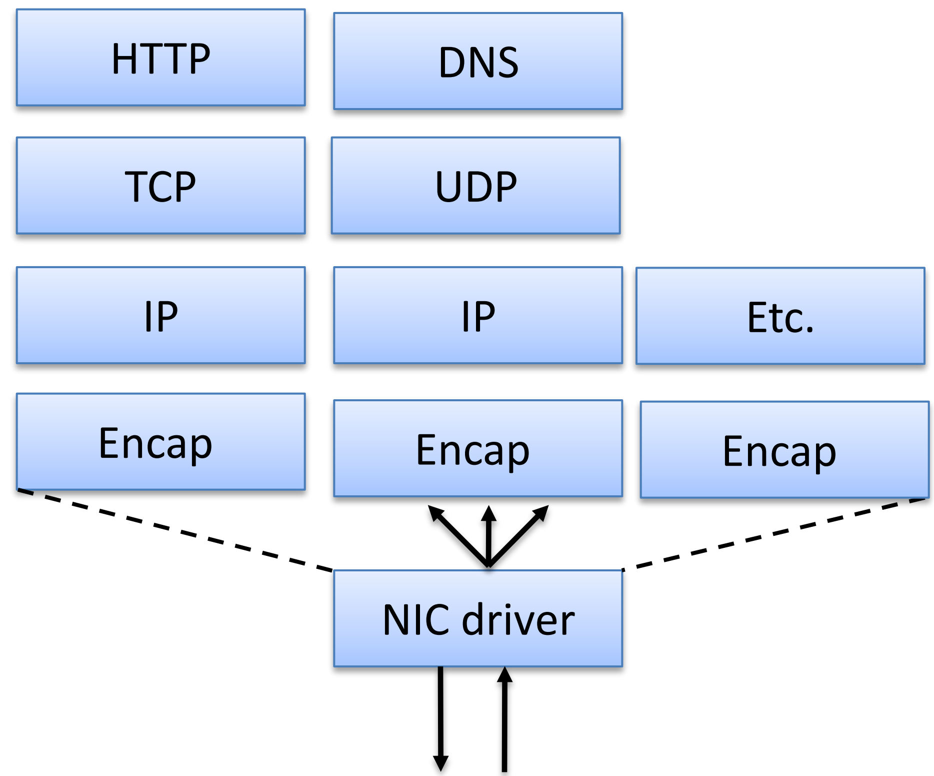


Consequences

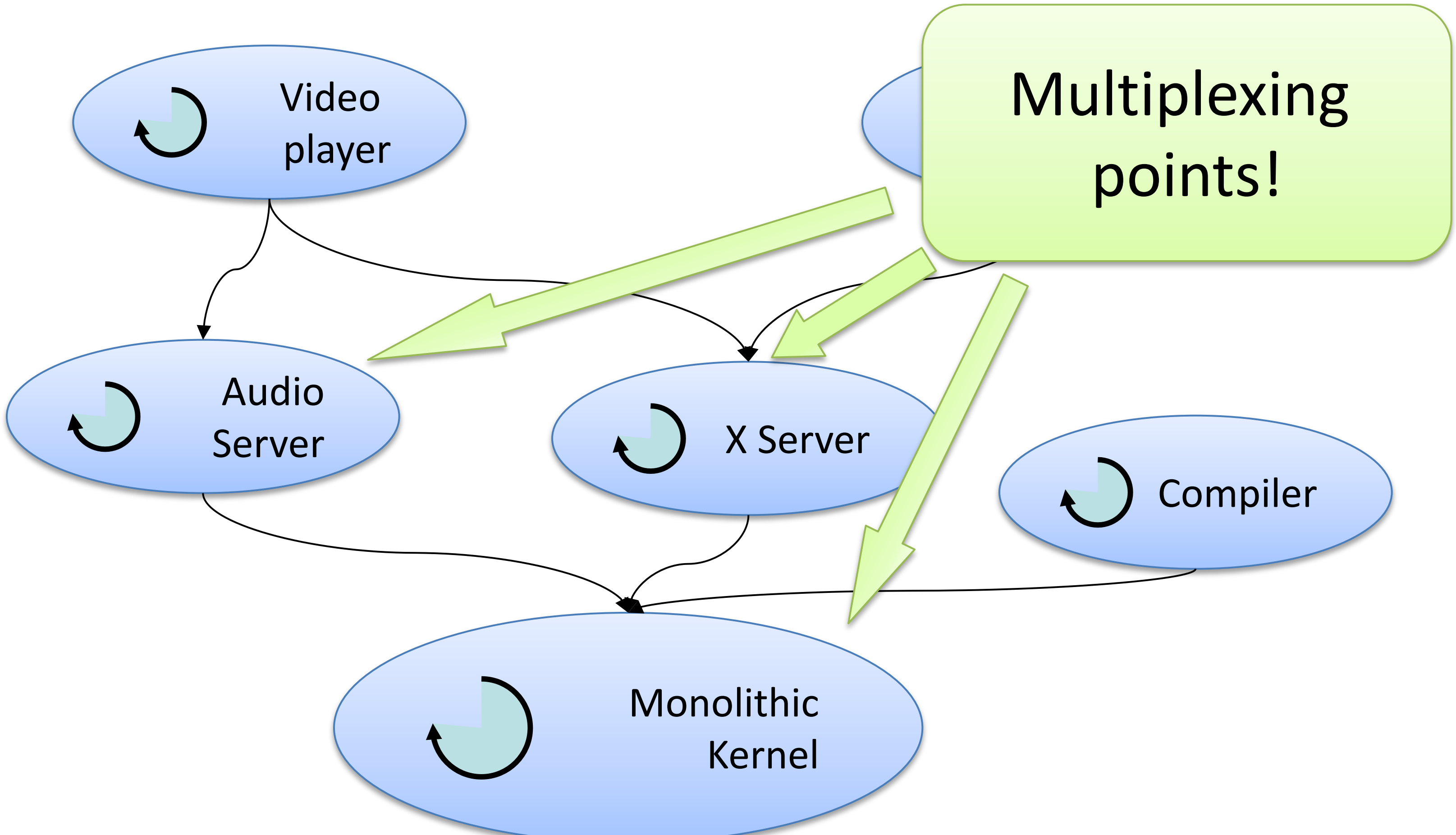
- Pluses:
  - Conceptually simple, easy to code
  - Efficient resource usage
- Minuses:
  - Application of packet only known at top of stack
  - QoS - every multiplexing point must schedule
  - Disaster for multimedia / realtime mix
  - “QoS Crosstalk”

Layered Multiplexing Considered Harmful

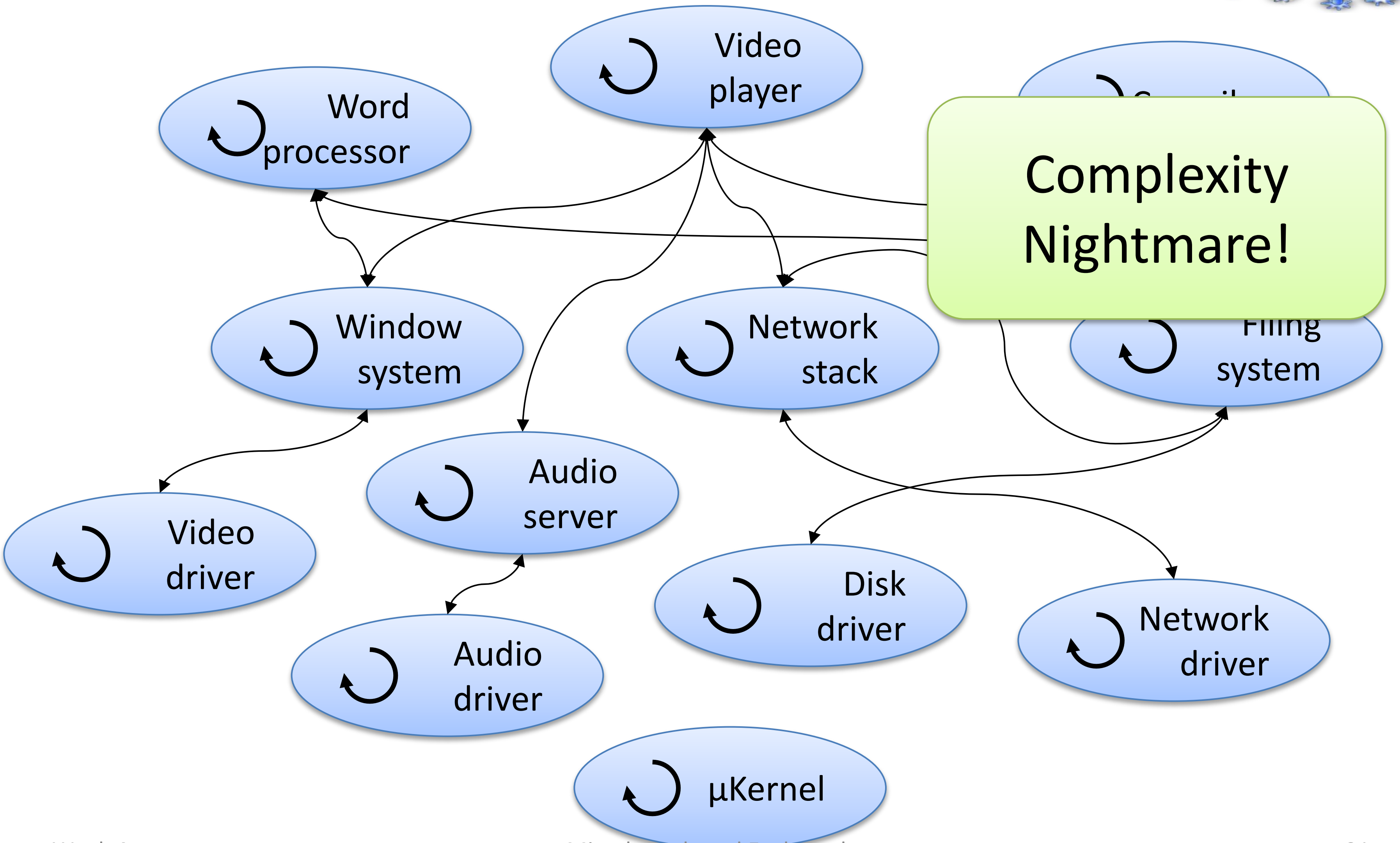
- Mux once, down low
- Rx:
  - Find target app first
  - Then execute protocol
- Tx:
  - Construct the wire format
  - Check it and mux last
- All packets scheduled with the application
- Works great with circuits!



Server-based monolithic OS



Microkernels?



Multiplexing in OS kernels

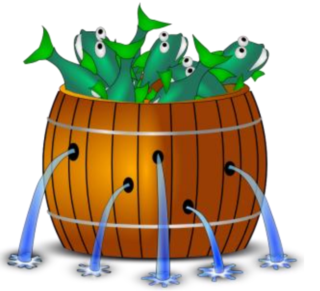
- Every server multiplexes some resource:
  - Needs to schedule it.
  - Needs to implement system-wide policy
  - Must be trusted to apply it
  - Has to cope with contention and locks
  - Is operating outside the control of any application

Mix and Match

- Real systems don’t fit neatly into a box:
  - seL4:
    - Kernel threads like L4
    - Capabilities like KeyKOS
  - Barrelfish:
    - Scheduler activations like Nemesis
    - Capabilities like KeyKOS and seL4
    - LibraryOS like and Exokernel
- Pick and choose to solve your problem

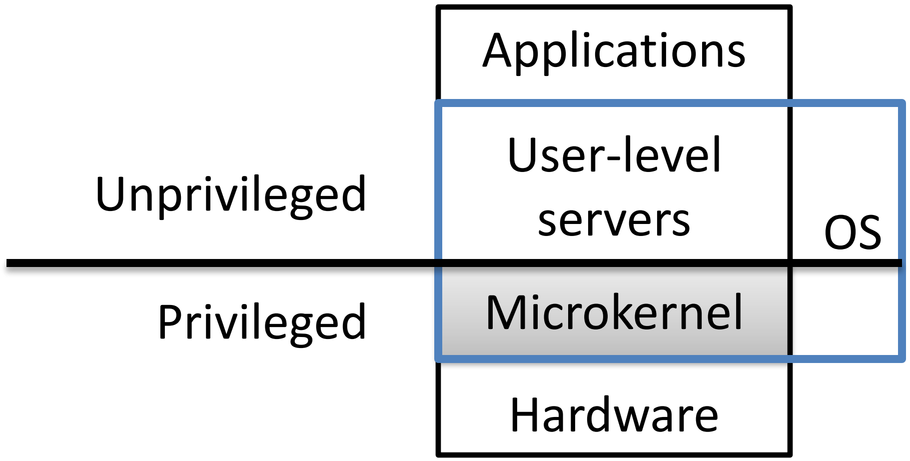
MICROKERNELS

Microkernels



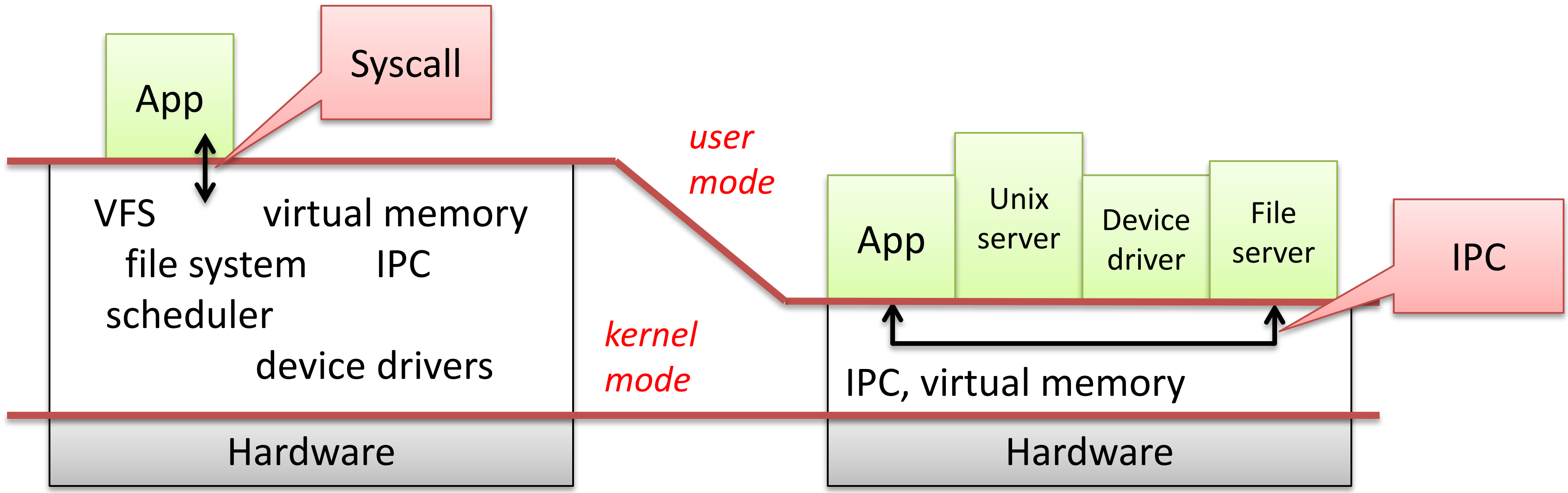
- Examples: L4, Mach, Amoeba, Chorus
- Kernel provides:
  - Time multiplexing: threads
  - Abstraction/Protection domains: address space
  - IPC
- All other functionality in server processes
  - Device drivers
  - File systems
- Services via **IPC** to user-level servers

Microkernel



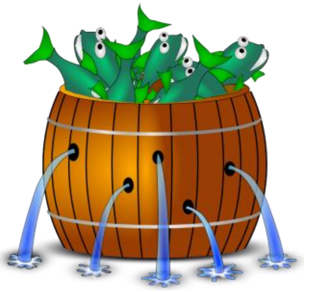
Based on ideas of the "Nucleus" [Brinch Hansen, 1970].

Monolithic vs. microkernel OS structure



- Monolithic OS
  - lots of privileged code
  - services invoked by syscall
- Microkernel OS:
  - little privileged code
  - services invoked by IPC
  - “horizontal” structure

Microkernel OS



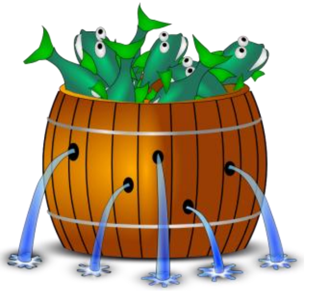
Kernel:

- Contains code which must run in privileged mode
- Isolates hardware dependence from higher levels
- Small and fast extensible system
- Provides **mechanisms**

User-level servers:

- Are hardware independent/portable
- Provide the “OS environment/personality”
- May be invoked:
  - From application (via message-passing IPC)
  - From kernel (via upcalls)
- Implement **policies**

Popular example: Mach



- Developed at CMU by Rashid and others from 1984 [Rashid et al., 1988]

Goals:

- **Tailorability**: support different OS interfaces
- **Portability**: almost all code H/Windependent
- **Real-time** capability
- **Multiprocessor** and **distribution** support
- **Security**
- Coined term **microkernel**

Basic features of Mach kernel

- Task and thread management
- Inter-process communication
  - asynchronous message-passing
- Memory object management
- System call redirection
- Device support
- Multiprocessor support

Mach = µkernel?

- Most OS services implemented at user level
  - Using memory objects and external pagers
  - Provides mechanisms, not policies
- Mostly hardware independent
- Big!
  - 140 system calls (300 in later versions), >100 kLOC
    - Unix 6th edition had 48 system calls, 10kLOC without drivers
  - 200 KiB text size (350 KiB in later versions)
- Poor performance
  - Tendency to move features into kernel

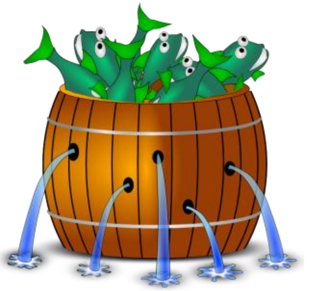
THE GREAT MICROKERNEL DEBATE

Critique of microkernel architectures

*"Personally, I'm **not** interested in making device drivers look like user-level. They aren't, they shouldn't be, and microkernels are just stupid."*

-- Linus Torvalds

Microkernel performance



- First generation microkernel systems ('80s, early '90s)
  - Exhibited poor performance when compared to monolithic UNIX implementations
  - Particularly Mach, the best-known example
- Typical results:
  - Move OS services back into the kernel for performance
  - Move complete OS personalities into kernel
    - Chorus Unix
    - Mac OS X (Darwin): complete BSD kernel linked to Mach
    - OSF/1
- Some spectacular failures
  - IBM Workplace OS
  - GNU Hurd

Microkernel performance

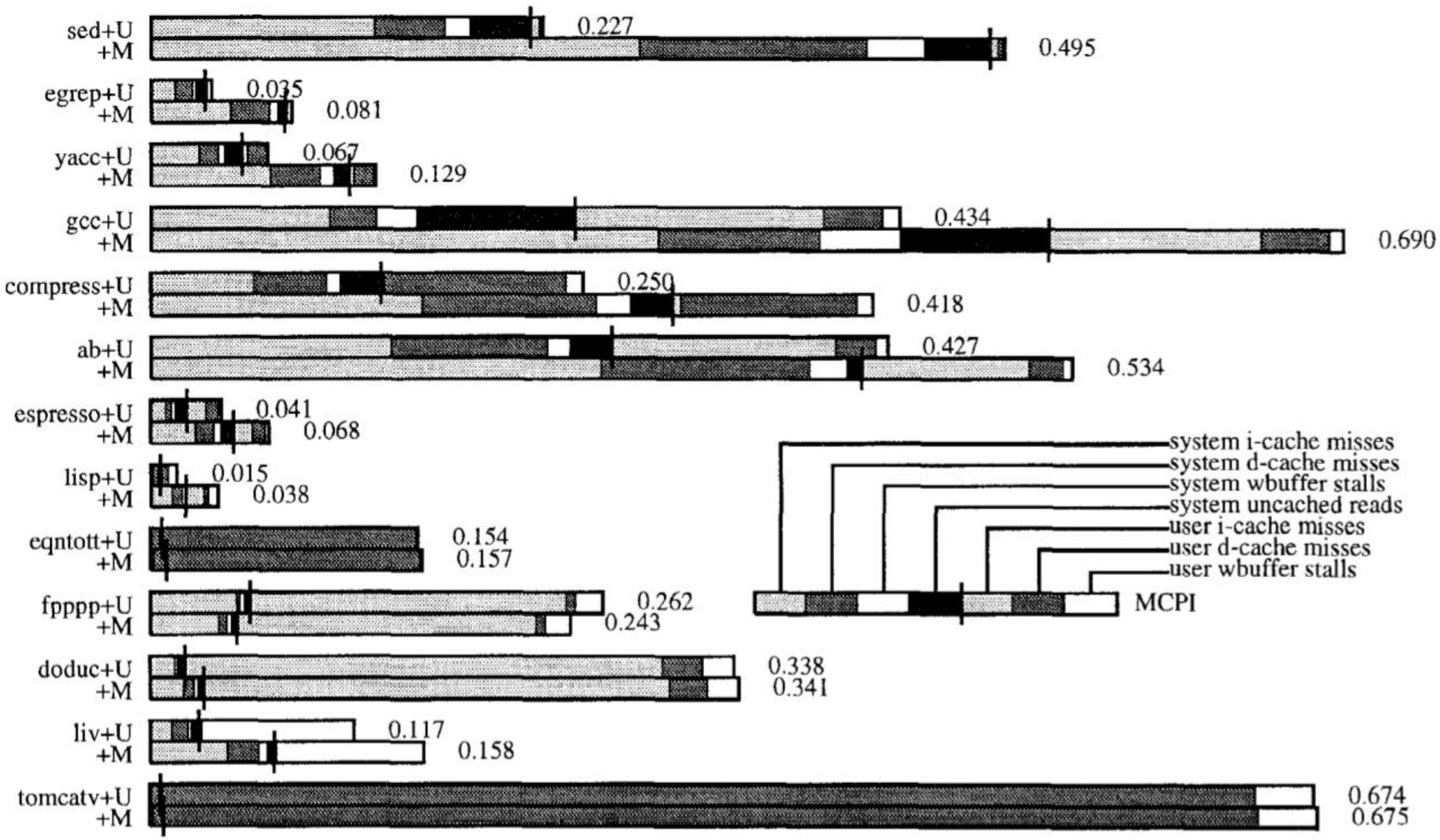
Reasons investigated [Chen and Bershad, 1993]:

- Instrumented user & system code to collect execution traces
- Run on DECstation 5000/200 (25MHz MIPS R3000)
- Run under Ultrix and Mach with Unix server
- Traces fed to memory system simulator
- Analysed memory cycles per instruction:

$$\mathit{MCPI} = \frac{\textit{stall cycles due to memory system}}{\textit{instructions retired}}$$

- Baseline MCPI (i.e. excluding idle loops)

Ultrix vs. Mach+Unix MCPI



Interpretation

Observations:

- Mach memory penalty higher
  - i.e. cache misses or write stalls
- Mach VMsystem executes more instructions than Ultrix
  - but is portable and has more functionality

Claim:

- Degraded performance is result of OS structure
- IPC cost is not a major factor:

“...the overhead of Mach’s IPC, in terms of instructions executed, is responsible for a small portion of overall system overhead. This suggests that microkernel optimizations focusing exclusively on IPC, without considering other sources of system overhead such as MCPI, will have a limited impact on overall system performance.”

Conclusions

- System instruction and data locality is measurably worse than user code
  - Higher cache and TLB miss rates
  - Mach worse than Ultrix
- System execution is more dependent than user on instruction cache behaviour
  - MCPI dominated by system Icache misses
- Competition between user and system code not a problem
  - Few conflicts between user and system cache

"The impact of Mach's microkernel structure on competition is not significant."

Conclusions

- Self-interference, especially on instructions, is a problem for system code
  - Ultrix would benefit more from higher cache associativity (direct-mapped cache was used)
- Block memory operations are responsible for a large component of overall MCPI
  - IO and copying
- Write buffers less effective for system
- Page mapping strategy has significant effect on cache

"The locality of system code and data is inherently poor"

Other experience with μkernel performance

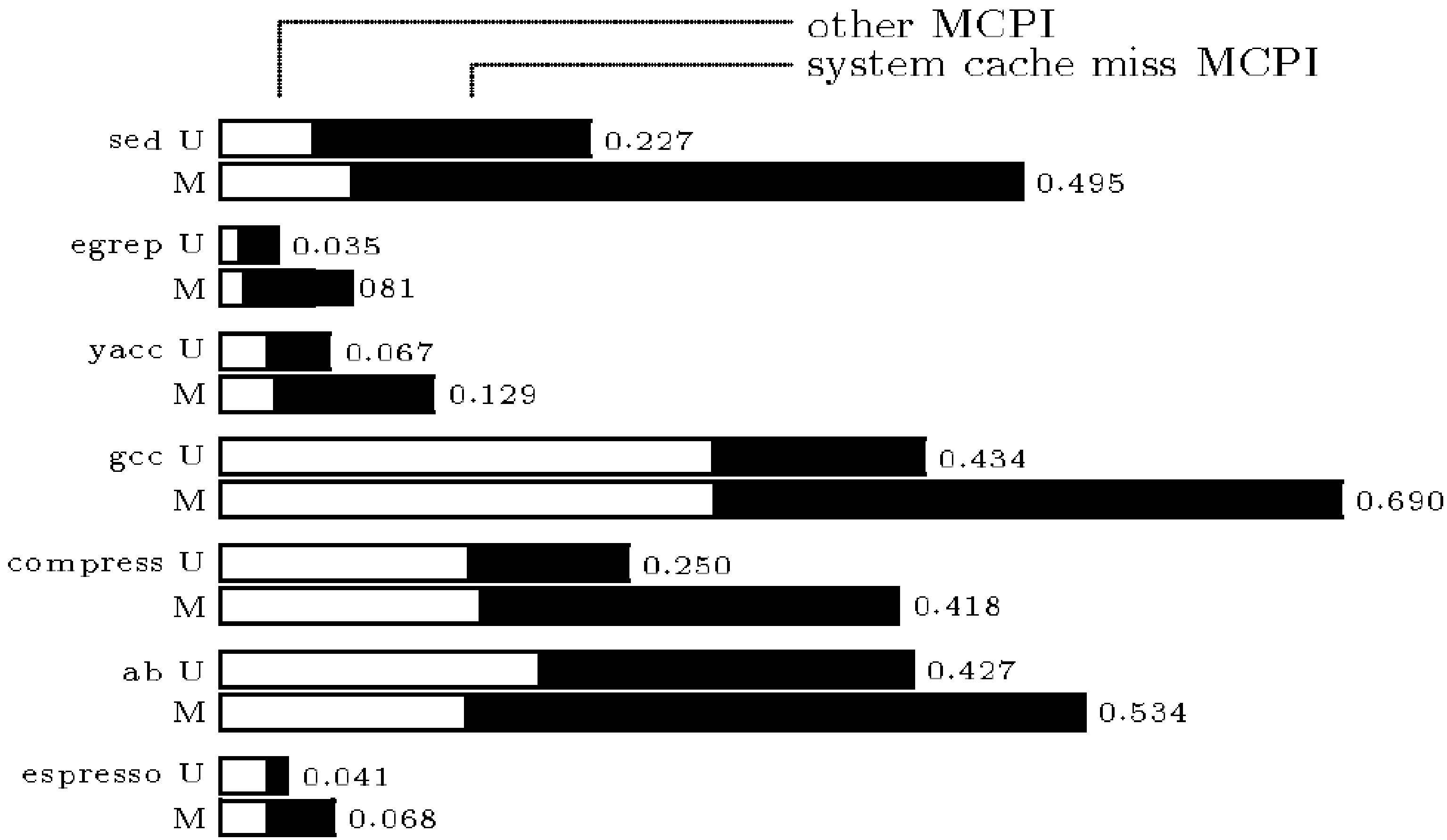
- System call costs are high
- Context switching costs are high
  - Getting worse with increasing CPU/memory speed ratios and lengthening pipelines

$\Rightarrow$ IPC (system call + context switch) expensive

- Microkernels depend heavily on IPC
  - Is the microkernel idea inherently flawed?

A Critique of the critique

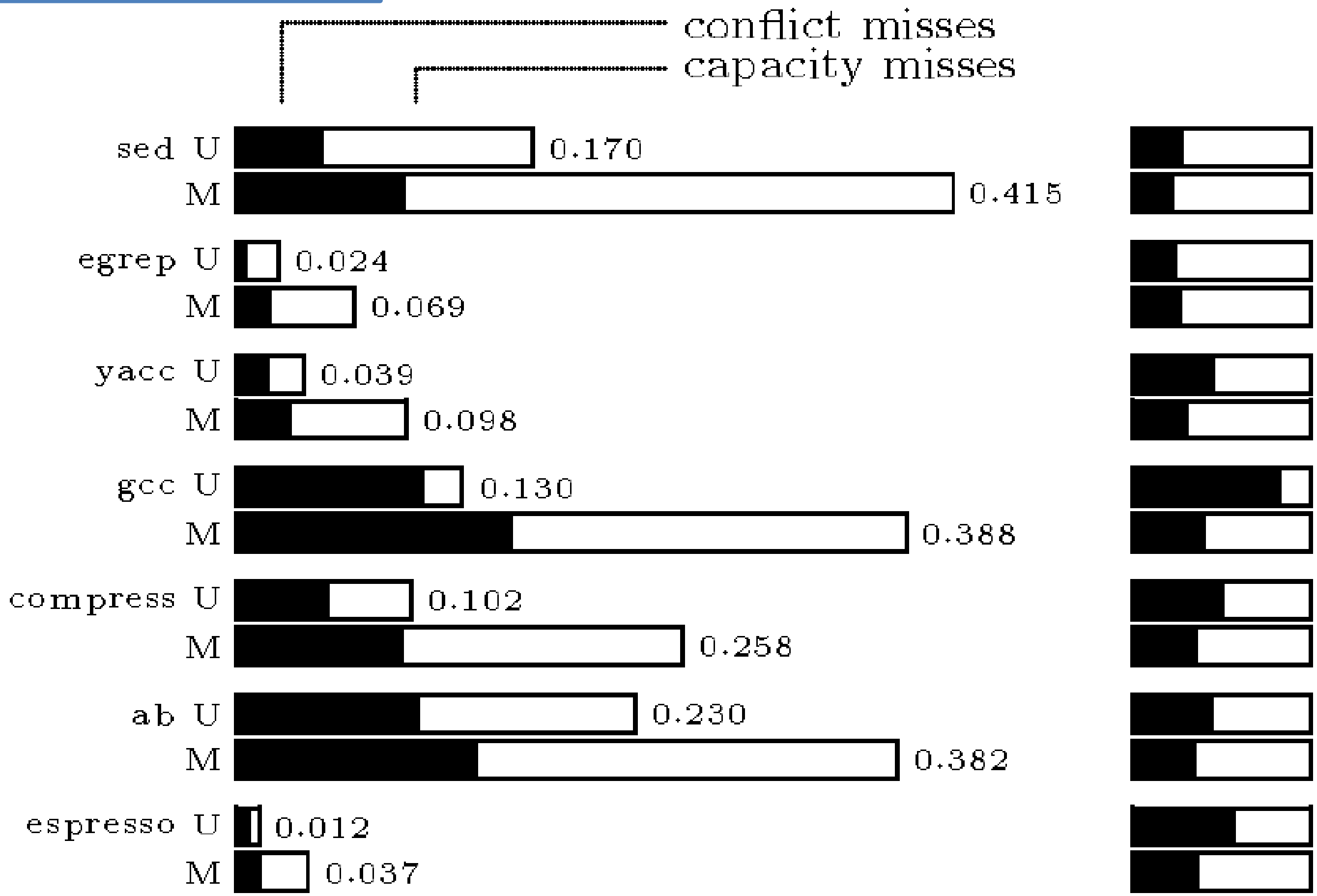
MCPI for Ultrix and Mach:



[Liedtke, 1995]

A Critique of the critique

MCPI caused by cache misses:



[Liedtke, 1995]

Conclusion

- Mach system is too big
  - Kernel + Unix server + emulation library
- Unix server is essentially the same as Unix
- Emulation library irrelevant [Chen and Bershad, 1993]
- Conclusion:

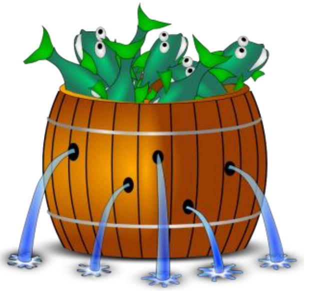
Mach kernel working set is too big

- Can we build microkernels which avoid these problems?

L3 AND L4

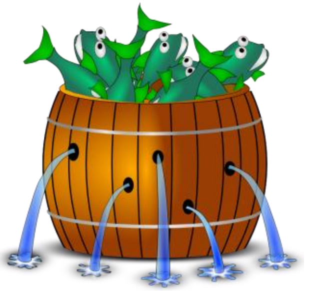
Improving IPC by kernel design

[Liedtke, 1993]



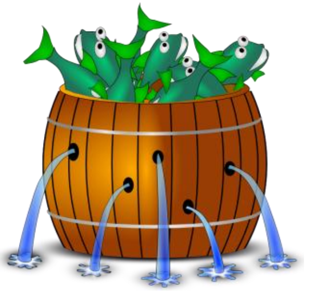
- IPC is the most important operation in a microkernel
- The way to make IPC fast is to design the whole system around it
- Design principle: aim at a concrete performance goal
  - Hardware-dictated costs are 172 cycles (3.5µs) for a 486
  - Aimed at 350 cycles for the implementation
- Applied to the L3 kernel

L3/L4 implementation techniques



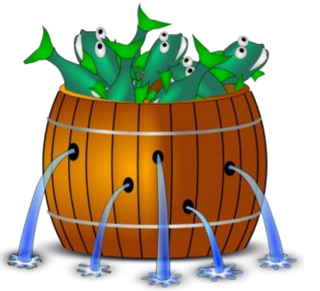
- Minimise number of system calls
  - Combined operations: `Call`, `ReplyWait`
  - Complex messages
    - Combines multiple messages into one operation
    - As many arguments as possible in registers
- Copy messages only once
  - via direct mapping, not user → kernel → user
- Fast access to thread control blocks (TCBs)
  - TCBs accessed via VMaddress determined from thread ID
  - Invalid threads caught via a page fault
  - Separate kernel stack for each thread in TCB
  - Avoids extra TLB misses on fast path

L3/L4 implementation techniques



- Lazy scheduling
  - Don’t update scheduling queues until you need to schedule
- Direct process switch to receiver
- Short messages in registers
- Reducing cache and TLB misses
  - Frequently-used TCB data near the beginning (single-byte displacement)
  - Frequently-used TCB data co-located (for cache locality)
  - IPC code and kernel tables in a single page (to reduce TLB pressure and refills)
- Use x86 alias registers (ax = al,ah) to pack arguments
- Avoid jumps and checks on fast path
- and more...

Results (L3)



- A short cross address space IPC (user to user) takes 5.2µs
  – compared to 115µs for Mach

- Code and data together use 592 bytes (7%) of on-chip cache
  – kernel must be small to be fast

On µ-Kernel Construction

[Liedtke, 1995]

What primitives should a microkernel implement?

"...a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality."

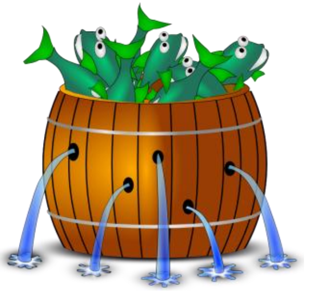
- Recursively-constructed address spaces
  - Required for protection
- Threads
  - As execution abstraction
- IPC
  - For communication between threads
- Unique identifiers
  - For addressing threads in IPC

What should a microkernel not provide?

- Memory management
- Page-fault handler
- File system
- Device drivers
- …

Rationale:

few features $\Rightarrow$ small size $\Rightarrow$ low cache use $\Rightarrow$ fast



Non-portability

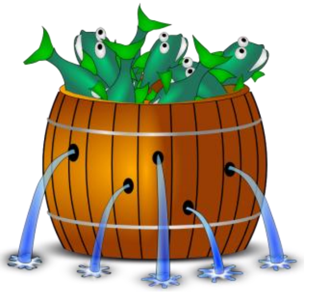
- Liedtke argues that microkernels must be constructed per-processor and are inherently unportable
- Eg. major changes made between 486 and Pentium:
  - Use of segment registers for small address spaces
  - Different TCB layout due to different cache associativity
    - Changes user-visible bit structure of thread identifiers!

FURTHER READING

Further reading

- Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. Commun. ACM 13, 4 (April 1970), 238-241. http://dx.doi.org/10.1145/362258.362278
- Dennis M. Ritchie and Ken Thompson. 1974. The UNIX time-sharing system. Commun. ACM 17, 7 (July 1974), 365-375. http://dx.doi.org/10.1145/361011.361061
- Lawrence Kenah and Ruth Goldenberg. 1987. VAX/VMS Internals and Data Structures: Version 5.2. Digital Press.
- Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. 1986. A structural view of the Cedar programming environment. ACM Trans. Program. Lang. Syst. 8, 4 (August 1986), 419-490. http://dx.doi.org/10.1145/6465.6466
- Niklaus Wirth. 1992. Project Oberon: The Design of an Operating System and Compiler. Acm Press Books. https://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html
- Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. 1992. The KeyKOS Nanokernel Architecture. In Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures. USENIX Association, Berkeley, CA, USA, 95-112. http://www.cis.upenn.edu/~KeyKOS/
- Accetta, M., Baron, R., Bolosky, W., Golub,D., Rashid, R., Tevanian, A., and Young, M. 1986. Mach: A New Kernel Foundation for UNIX Development. Proceedings of the Summer USENIX Conference, 93-112.
- J. Bradley Chen and Brian N. Bershad. 1993. The impact of operating system structure on memory system performance. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93). ACM, New York, NY, USA, 120-133. http://dx.doi.org/10.1145/168619.168629

Further reading



- Jochen Liedtke. 1993. Improving IPC by kernel design. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93). ACM, New York, NY, USA, 175-188. http://dx.doi.org/10.1145/168619.168633
- D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr.. 1995. Exokernel: an operating system architecture for application-level resource management. In Proceedings of the fifteenth ACM Symposium on Operating systems principles (SOSP '95), Michael B. Jones (Ed.). ACM, New York, NY, USA, 251-266. http://dx.doi.org/10.1145/224056.224076
- J. Liedtke. 1995. On micro-kernel construction. In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95), Michael B. Jones (Ed.). ACM, New York, NY, USA, 237-250. http://dx.doi.org/10.1145/224056.224075
- I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. 1996. The design and implementation of an operating system to support distributed Multimedia applications. IEEE J.Sel. A. Commun. 14, 7 (September 1996), 1280-1297. http://dx.doi.org/10.1109/49.536480
- Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. 1999. The pebble component-based operating system. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99). USENIX Association, Berkeley, CA, USA, 20-20. https://www.usenix.org/legacy/event/usenix99/full_papers/gabber/gabber.pdf
- David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. 2001. A Network-Centric Approach to Embedded Software for Tiny Devices. In Proceedings of the First International Workshop on Embedded Software (EMSOFT '01), Thomas A. Henzinger and Christoph Meyer Kirsch (Eds.). Springer-Verlag, London, UK, UK, 114-130. http://www.cs.berkeley.edu/~culler/papers/embedded.pdf
- Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev. 41, 2 (April 2007), 37-49. http://dx.doi.org/10.1145/1243418.1243424