Operating System Structure

Joey Echeverria joey42+os@gmail.com
modified by: Matthew Brewer mbrewer@andrew.cmu.edu

Nov 15, 2006

Overview

- Motivations
- Kernel Structures
  - Monolithic Kernels
    - Kernel Extensions
  - Open Systems
  - Microkernels
  - Provable Kernel Extensions
  - Exokernels
  - More Microkernels
- Final Thoughts

Motivations

- Operating systems have a hard job.
- Operating systems have 3 jobs:

1. Protection boundaries
2. Abstraction layers
3. Hardware Multiplexers

Motivations

- Job 1) Protection Boundaries
  - Protect processes from each other
  - Protect crucial services (like the kernel) from process
- Implications
  - Everyone trusts the kernel
- Complicated
  - See Project 3 :)
  - Full OS is millions of lines of code
  - Very Roughly: correctness $\propto$ 1/code_size

Motivations

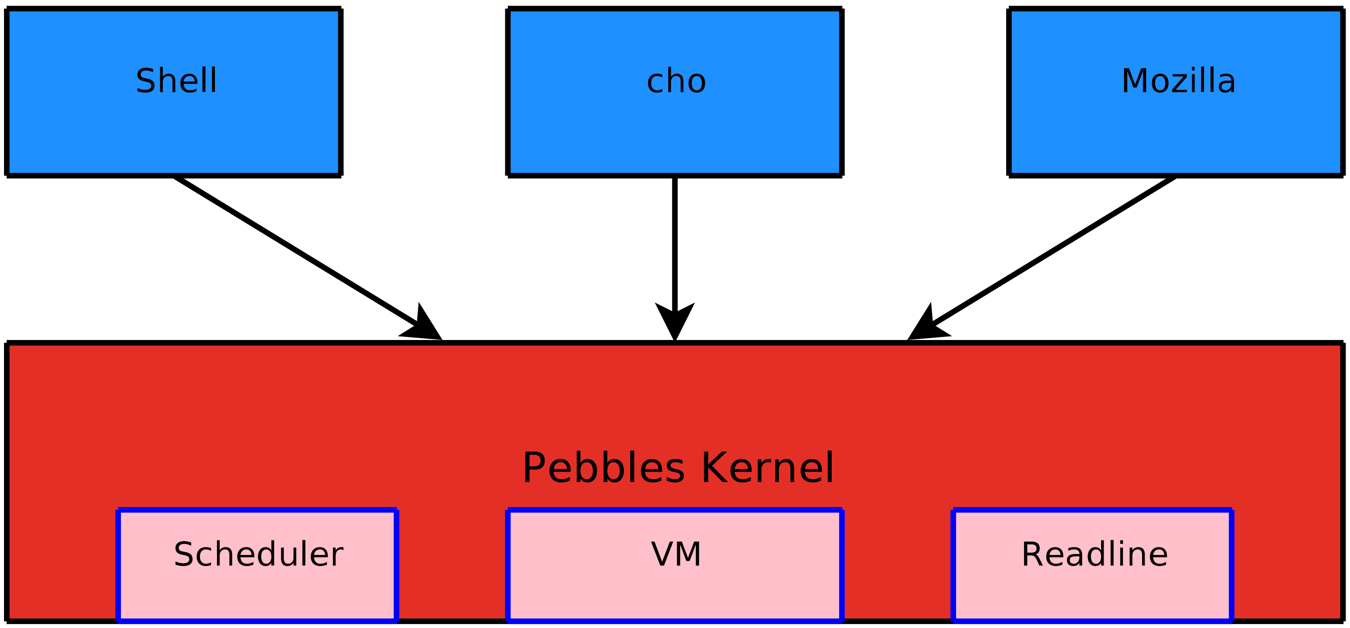
- Job 2) Abstraction Layer
  - Presents “simple”, “uniform” interface to hardware
  - Applications see a well defined interface (system calls)
    - Block Device (hard drive, flash card, network mount, USB drive)
    - CD drive (SCSI, IDE)
    - tty (teletype, serial terminal, virtual terminal)
    - filesystem (ext2-4, reiserfs, UFS, FFS, NFS, AFS, JFFS2, CRAMFS)
    - network stack (TCP/IP abstraction)

Motivations

- Job 3) Hardware Multiplexer
  - Each process sees a “computer” as if it were alone
  - Requires allocation and multiplexing of:
    * Memory
    * Disk
    * CPU
    * IO in general (network, graphics, keyboard etc.)
- If OS is multiplexing it must also allocate
  - Priorities, Classes? - HARD problems!!!

Monolithic Kernels

- Pebbles Kernel



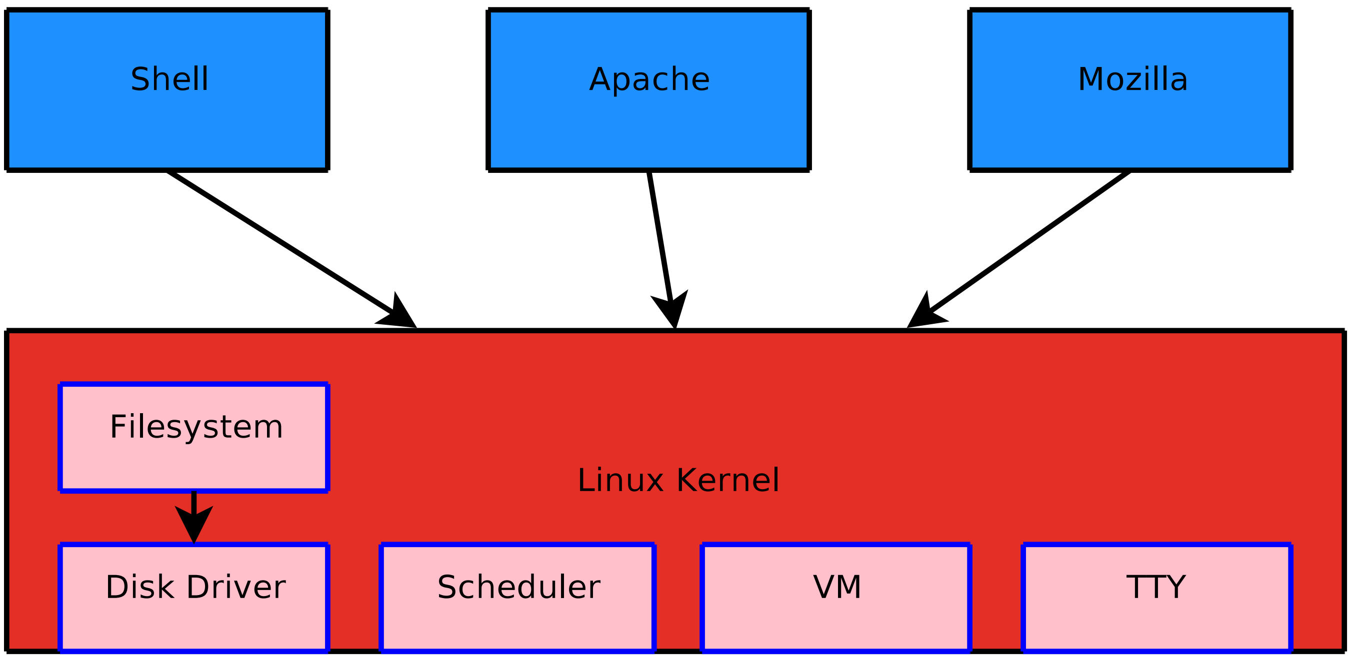
Monolithic Kernels

Pebbles Kernel

- Syscalls ≈ 20
  - fork, exec, cas_runflag, yield
- Lines of trusted code ≈ 2000 to 24000

Monolithic Kernels

- Linux Kernel... similar?

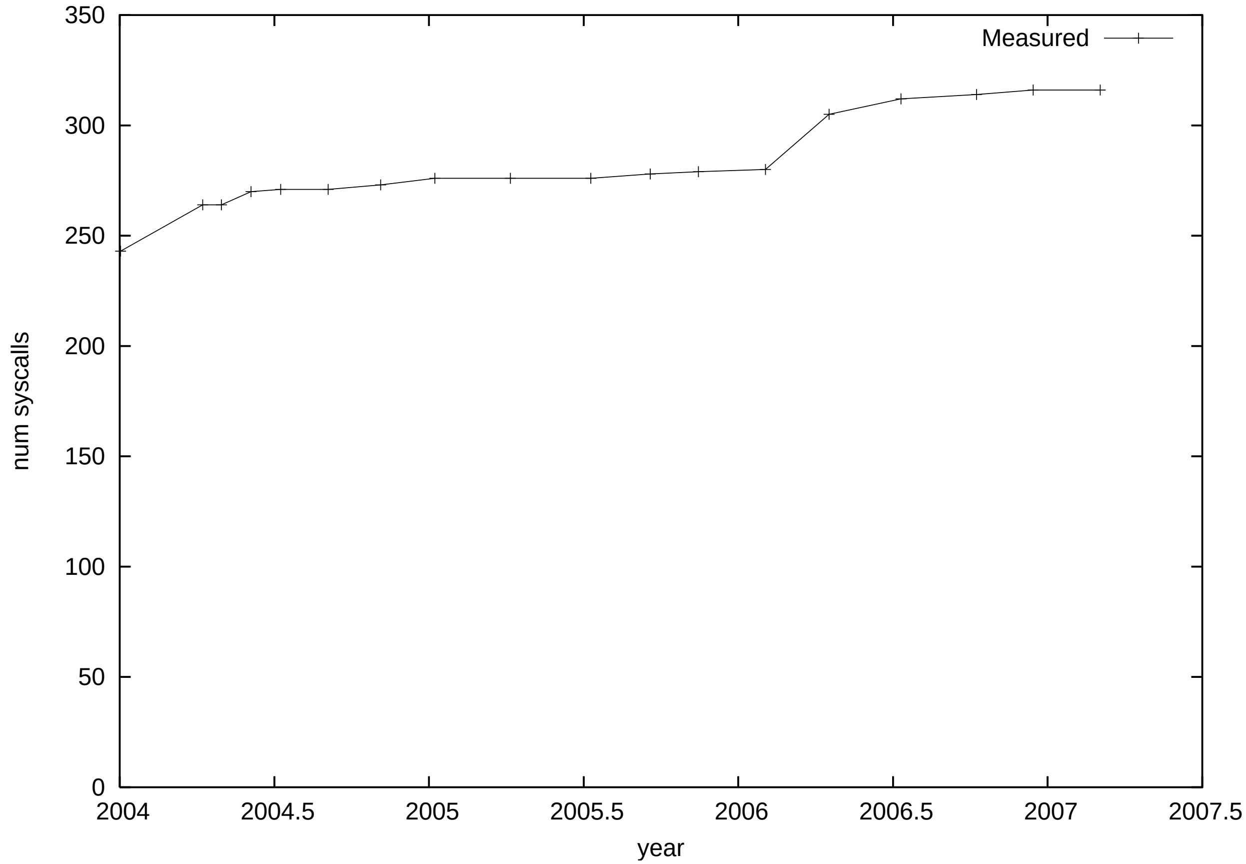


Monolithic Kernels

2.6 Linux kernel

- Syscalls: ≈ 243 in 2.4, and increasing fast
  - fork(), exec(), read(), readdir(), ioctl()
- Lines of trusted code ≈ 7 million as of 2 weeks ago
  - $\approx 200,000$ are just for USB drivers
  - $\approx 15,000$ for USB core alone
- Caveats - Many archs/subarchs, every driver EVER

Monolithic Kernels



Monolithic Kernels

- Advantages:
  - + Well understood
  - + Good performance
  - + High level of protection between applications

- Disadvantages:
  - – No protection between kernel components
  - – LOTS of code is in kernel
  - – Not (very) extensible

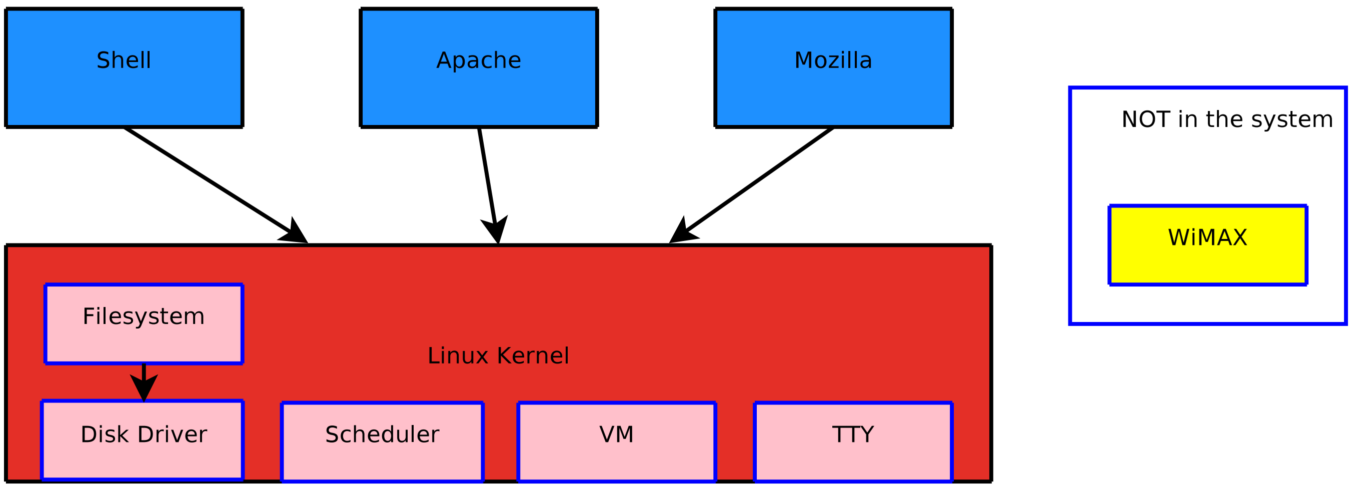
- Examples: UNIX, Mac OS X, Windows NT/XP, Linux, BSD, i.e., common

(Loadable) Kernel Modules

- Problem - Bob has a WiMAX card, and he wants a driver
- I don't want a (large, unstable) WiMAX driver muddying my kernel
- Solution - kernel modules!
  - Special binaries compiled with kernel
  - Can be loaded at run-time - so we can have LOTS of them
  - Can break kernel, so loadable only by root
- done in: VMS, Windows NT, Linux, BSD, OS X

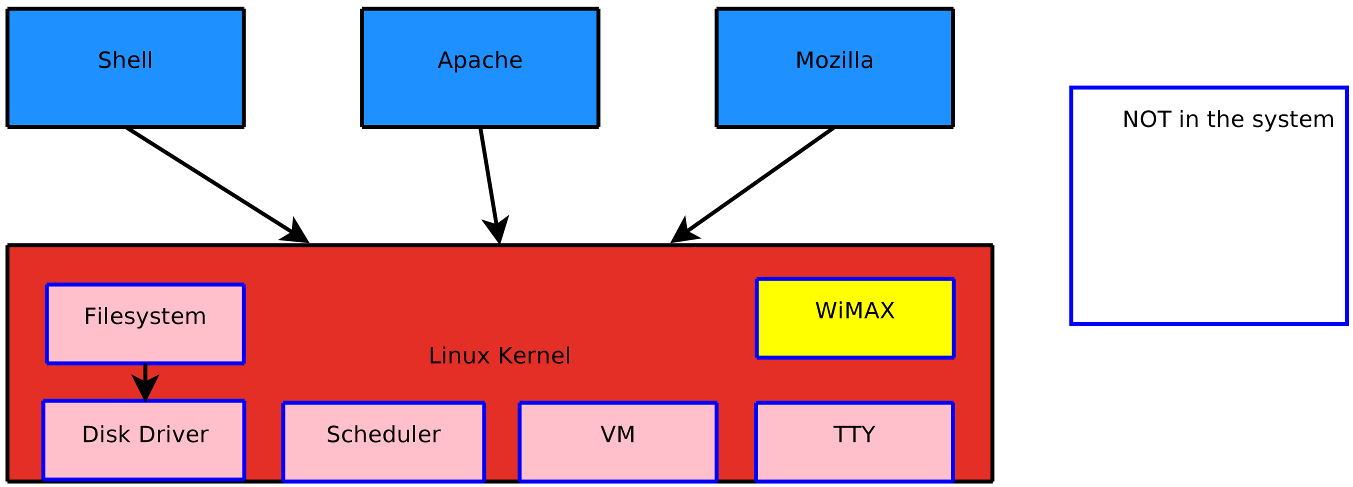
(Loadable) Kernel Modules

Linux Kernel



(Loadable) Kernel Modules

Linux Kernel with WiMAX module



Kernel Extensions

- Advantages
  - Can extend kernel
  - Extensions run FAST
- Disadvantages
  - Adding things to kernel can break it
  - Have to ask sysadmin nicely

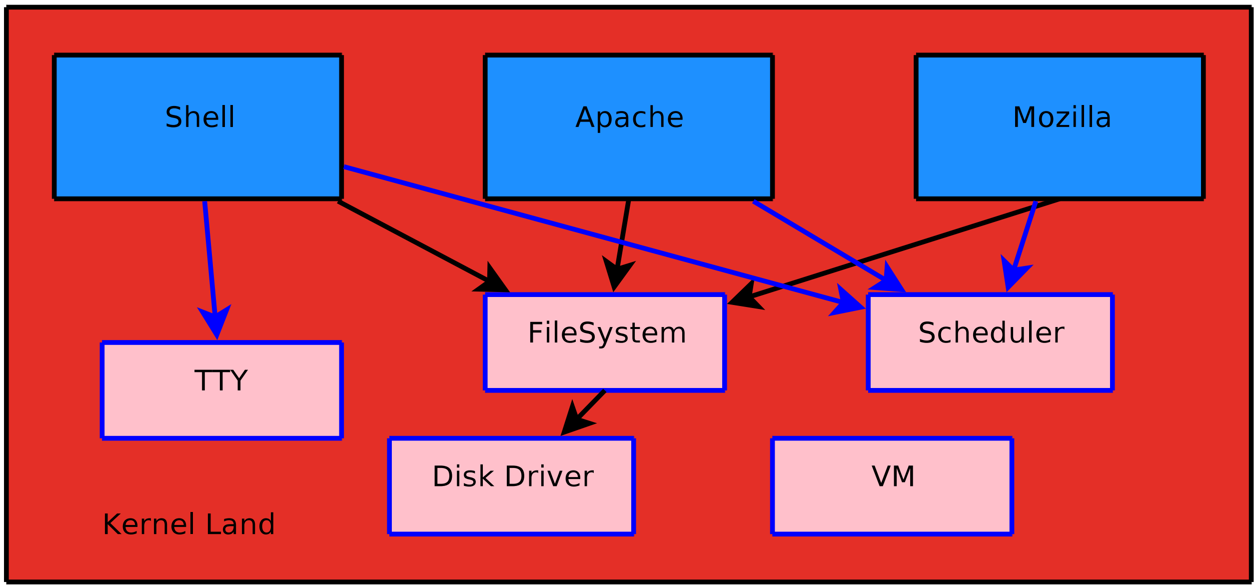
Open Systems

- Monolithic kernels run reasonably fast, and can be extended (at least by root)

- Overhead
  - System calls
    X86 processor - minimum of 90 cycles to trap to higher PL
  - Address space
    Context switch must dump TLB, this costs more every day (see x86-64)

- So, do we really need protection?

Open Systems



Open Systems

- Syscalls - none!
- Lines of trusted code - all of it!

Open Systems

- Applications, libraries, and kernel all sit in the *same address space*
- Does anyone actually do this craziness?
  - MS-DOS
  - Mac OS 9 and prior
  - Windows ME, 98, 95, 3.1, etc.
  - Palm OS
  - Some embedded systems
- Used to be *very* common

Open Systems

- Advantages:
  + *Very* good performance
  + Very extensible
    * Undocumented Windows, Schulman et al. 1992
    * In the case of Mac OS and Palm OS there's an extensions *industry*
  + Can work well in practice
  + Lack of abstractions makes realtime systems easier
- Disadvantages:
  – No protection between kernel and/or applications
  – Not particularly stable
  – Composing extensions can result in unpredictable behavior

Microkernels

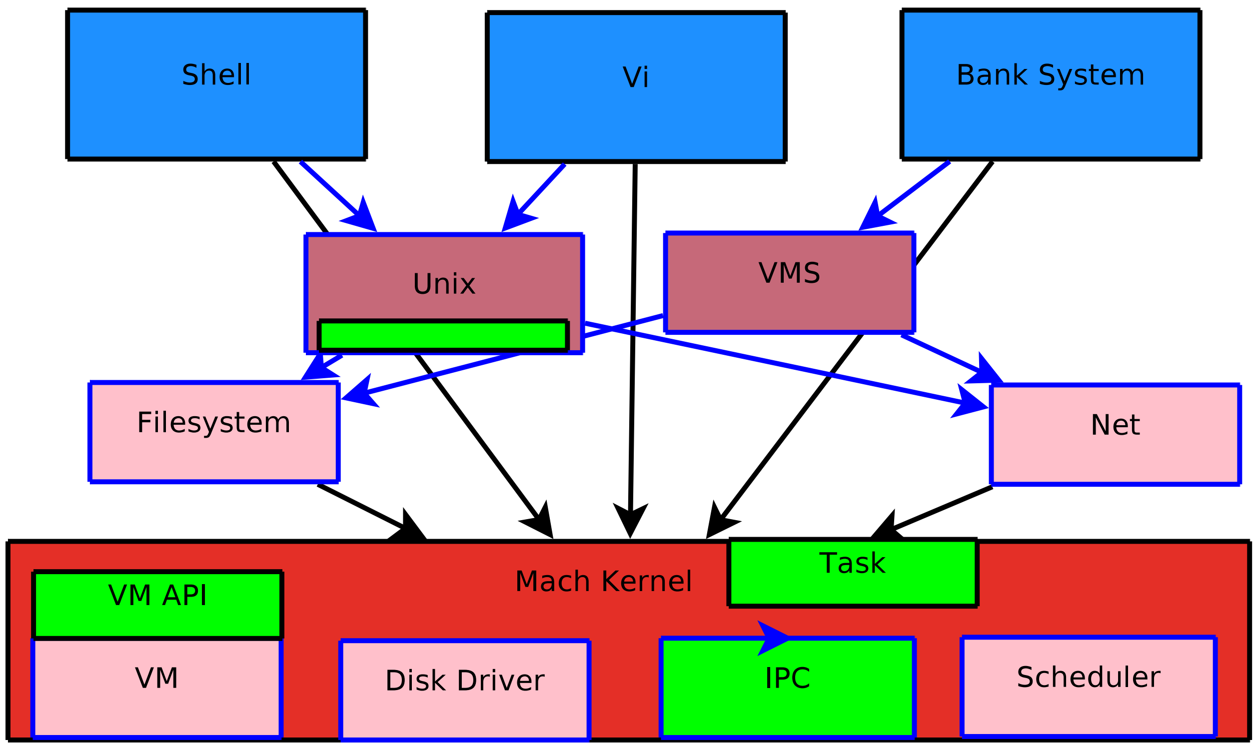
- Monolithic Kernels
  - Extensible (by root)
  - User protection
  - No internal protection - makes debugging hard, bugs CRASH
- Open Systems
  - Extensible by everyone
  - No protection at all - same_deal++ AND can't be multi-user
- ... Can we have user extensibility, and internal protection?

Microkernels

- Replace the monolithic kernel with a “small, clean, logical” set of abstractions
  - Tasks
  - Threads
  - Virtual Memory
  - Interprocess Communication
- Move the rest of the OS into *server processes*

Microkernels (Mach Vision)

multi-server



Microkernels (Mach)

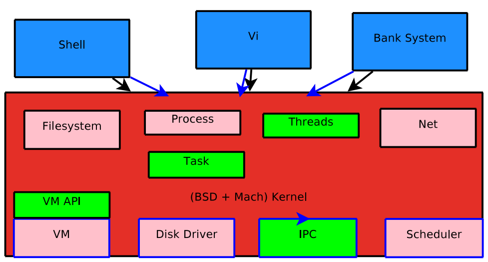
Mach

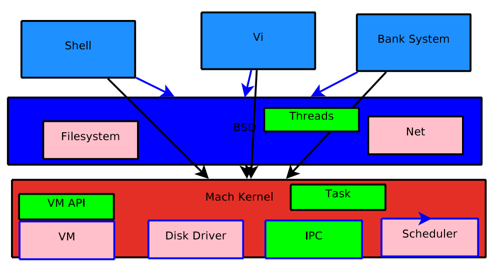
- Syscalls: initially 92, increased slightly later
  - msg_send, port_status, task_resume, vm_allocate
- Lines of trusted code $\approx 484,000$ (Hurd version)
- Caveats - several archs/subarchs, some drivers

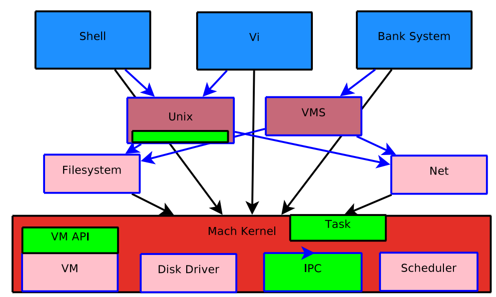
Microkernels (Mach)

- Started as a project at CMU (based on RIG project from Rochester)
- Plan

1. Mach 2: Take BSD 4.1 add VM API, IPC, SMP, and threading support
2. Mach 3: saw kernel in half and run as "single-server"
3. Mach 3 continued: decompose single server into smaller servers

1) 

2) 

3) 

Microkernels (Mach)

- Results
  1. Mach 2 completed in 1989
     - Unix: SMP, kernel threads, 5 architectures
     - Used for Encore, Convex, NeXT, and subsequently OS X
     - success!
  2. Mach 3 Finished(ish)
     - Mach 2 split in 2
     - Ran on a few systems at CMU, and a few outside
  3. Mach 3 continued
     - Multi-server systems: Mach-US, OSF
     - Never deployed

Microkernels (Mach 3)

- Advantages (Mach 3):
  - + Strong protection (even from itself)
  - + Untrusted system services (user-space filesystem... see Hurd)
- Disadvantages:
  - – Performance
    - ∗ It looks like extra context switches and copying would be expensive
    - ∗ Mach 3 ran slow in experiments
    - ∗ Kernel still surprisingly large -
      "It's not micro in size, it's micro in functionality"
    - ∗ Still hasn't REALLY been tried

Microkernel as hypervisor (Mach 3)

- A few other uses of the microkernel, look what we can do!
- IBM Workplace OS (Mach 3.0)
  - * one kernel for OS/2, OS/400, and AIX
  - * failure
- Call it a “hypervisor” - idea is rather popular again
  - * Xen, L4

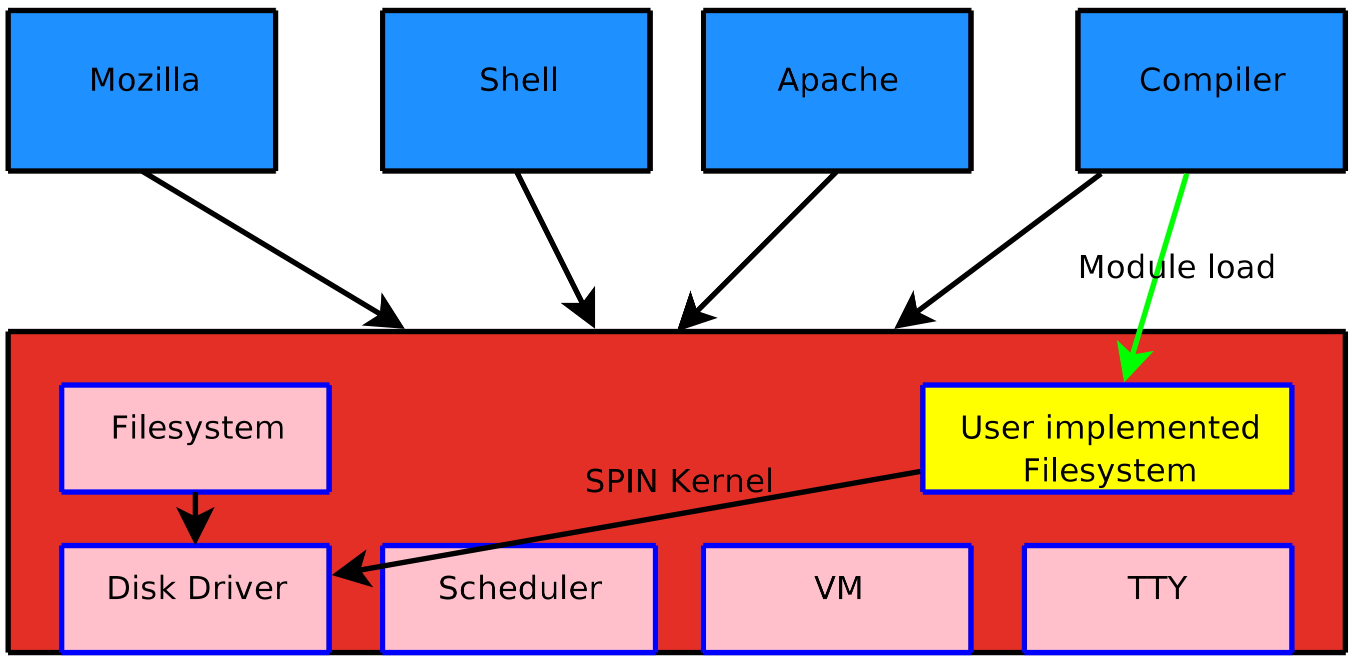
Microkernels (Mach)

- Things to remember about Mach 3
  - Mach 3 == microkernel, Mach 2 == monolithic
  - Code ran slow at first, then everyone graduated
  - Proved microkernel is feasible
  - Stability/Speed of seperation both unproven
- Other interesting points
  - Other microkernels from Mach period: ChorusOS, QNX
  - QNX, competes with VxWorks as a realtime OS
  - ChorusOS, realtime kernel out of europe, now open sourced by Sun
  - More later

Provable Kernel Extensions

- We want an extensible OS
- We want extensions to run fast, but be safe for addition by users
- Assume we don't like microkernels (slow, more code, whatever)
- So... other ideas?

Provable Kernel Extensions

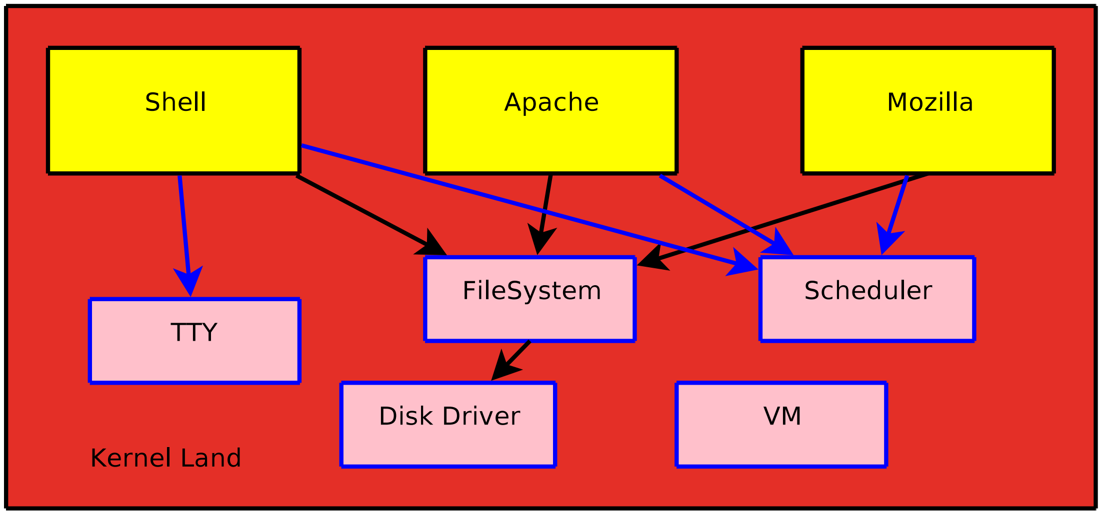


Provable Kernel Extensions

- PROVE the code does what we want
- Checker can be EXTREMELY conservative and careful about what it lets in
  - Interepreter safety (CMU: Acceta)
  - Compiler-checked source safety (UW: Spin: Modula-3)
  - Kernel-verified binary safety (CMU: Proof-carrying code)
    - More language agnostic - *just* need a compiler that compiles to PCC
- Safe? Guaranteed (if compiler is correct... same deal as a kernel)

Provable Everything

What if were a proven kernel extension?



Provable Everything

- What if ALL code was loaded into the “kernel” and just proved to do the “right” thing?... Is this silly, or a good idea?
  - Looks a lot like Open Systems
  - Except compiler can enforce more stability
- Effectiveness strongly dependent on quality of proofs
- Some proofs are HARD, some proofs are IMPOSSIBLE!
- Actual Work: groundwork being done here, MSR’s ”Singularity” - take this as you will

Provable Everything

- Advantages:
  - \+ Extensible even by users, just add a new extenssion/application
  - \+ Safe, provably so
  - \+ Good performance because everything is in the kernel
- Disadvantages:
  - – Proofs are hard - and checking can be slow
  - – We can't actually DO this for interesting code (yet?)
  - – Constrained implementation language
  - – Constraints may cause things to run slower than protection boundaries
  - – Still very limited in scope, not used widely

Exokernels

- Monolithic kernel
  - Too many abstractions can get in the way
  - Not easily extensible for every application (special kernel mods)
- Microkernel
  - Maybe Mach is still too much kernel?
  - Too heavy an abstraction, too portable, just too much
- Proof systems
  - Useful proof checkers are large & still can't do everything
- If applications control system, can optimize for their usage cases

Exokernels

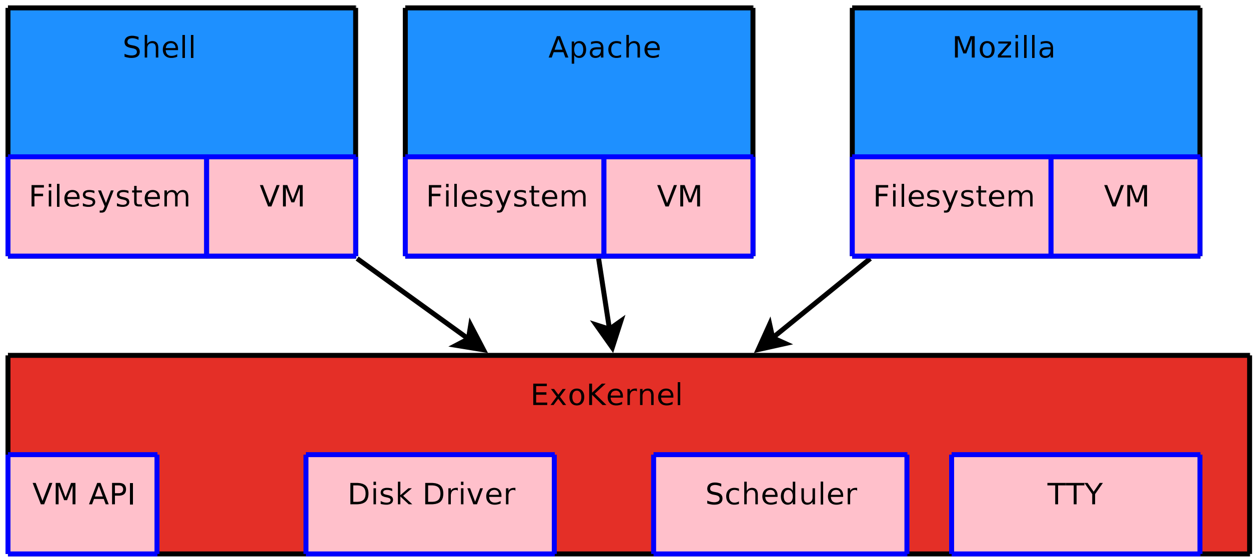
- Basic idea: Take the operating system out of the kernel and put it into libraries

- Why? Applications know better how to manage active hardware resources than kernel writers do

- Safe? Exokernel is simply a hardware multiplexer, and thus a permissions boundary.

- Separates the security and protection from the management of resources

Exokernels (Xok/ExOS)



Exokernel (Xok)

Xok

- Syscalls ≈ 120
  - insert_pte, pt_free, quantum_set, disk_request
- Lines of trusted code $\approx 100,000$
- Caveats - One arch, few/small drivers

Exokernels: VM Example

- There is no fork()
- There is no exec()
- There is no automatic stack growth
- Exokernel keeps track of physical memory pages
  Assigns them to an application on request
  - Application (via syscall):
    1. Requests frame
    2. Requests map of Virtual → Physical

Exokernels: simple fork()

- fork():
  - Acquire a new, blank address space
  - Allocate some physical frames
  - Map physical pages into blank address space
  - Copy bits (from us) to the target, blank address space
  - Allocate a new thread and bind it to the address space
  - Fill in new thread's registers and start it running
- The point is that the kernel doesn't provide fork()

Exokernels: COW fork()

- fork(), advanced:
  - Acquire a new, blank address space
  - Ask kernel to set current space's mappings to R/O
  - Map current space's physical pages R/O into blank space
  - Update copy-on-write table in each address space
  - Application's page-fault handler (like a signal handler) copies/re-maps
- Each process can have its own fork() optimized for it – or none at all

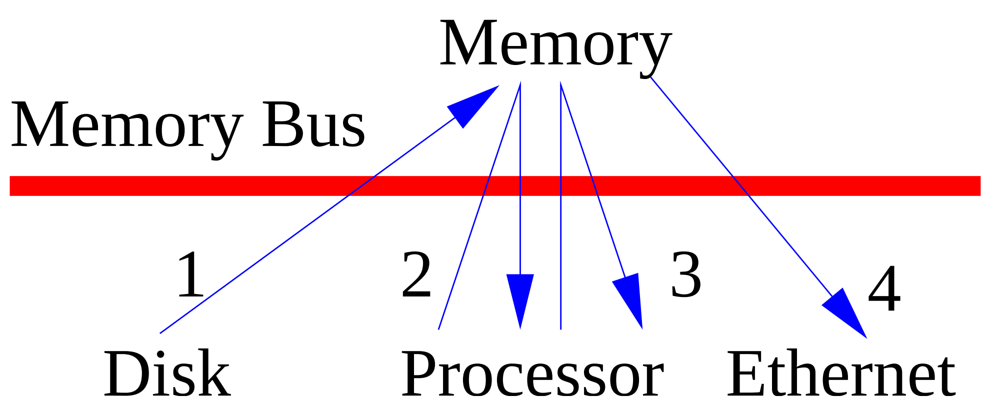
Exokernels: Web Server Example

- In a typical web server the data must go from:
  1. the disk to kernel memory, read()
  2. kernel memory to user memory, memcpy()
  3. user memory back to kernel memory memcpy()
  4. kernel memory to the network device write()
- In an exokernel, the application can have the data go straight from disk to the network interface

Exokernels: Web Server Example

- Traditional kernel and web server:

1. read() − copy from disk to kernel buffer
2. read() − copy from kernel to user buffer
3. send() − user buffer to kernel buffer
   −− data is check−summed
4. send() − kernel buffer to device memory

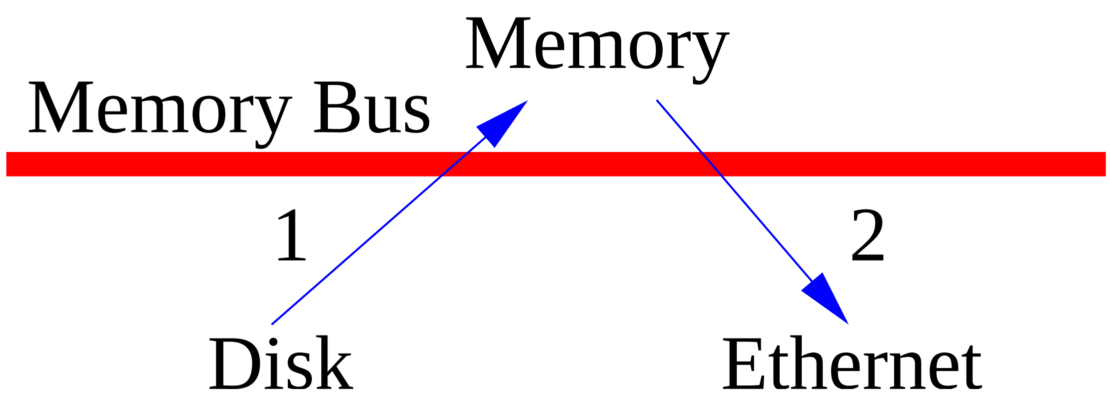


That is: six bus crossovers

Exokernels: Web Server Example

- Exokernel and Cheetah:

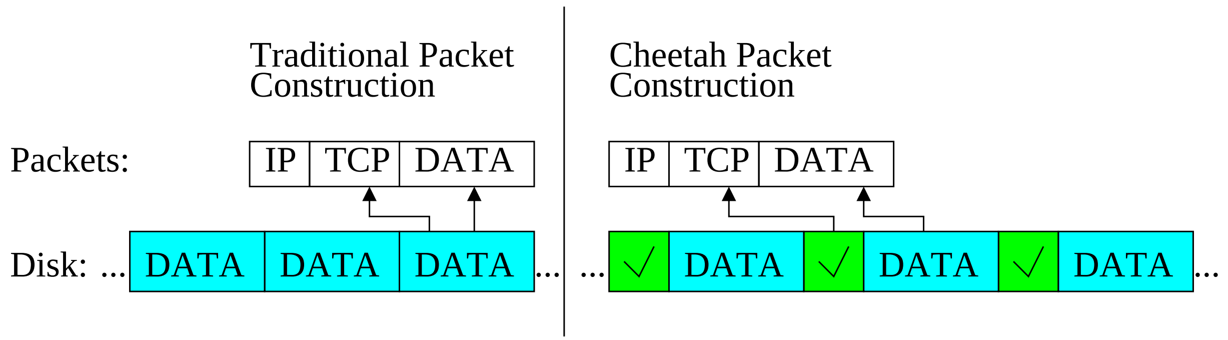
1. Copy from disk to memory
2. Copy from memory to network device



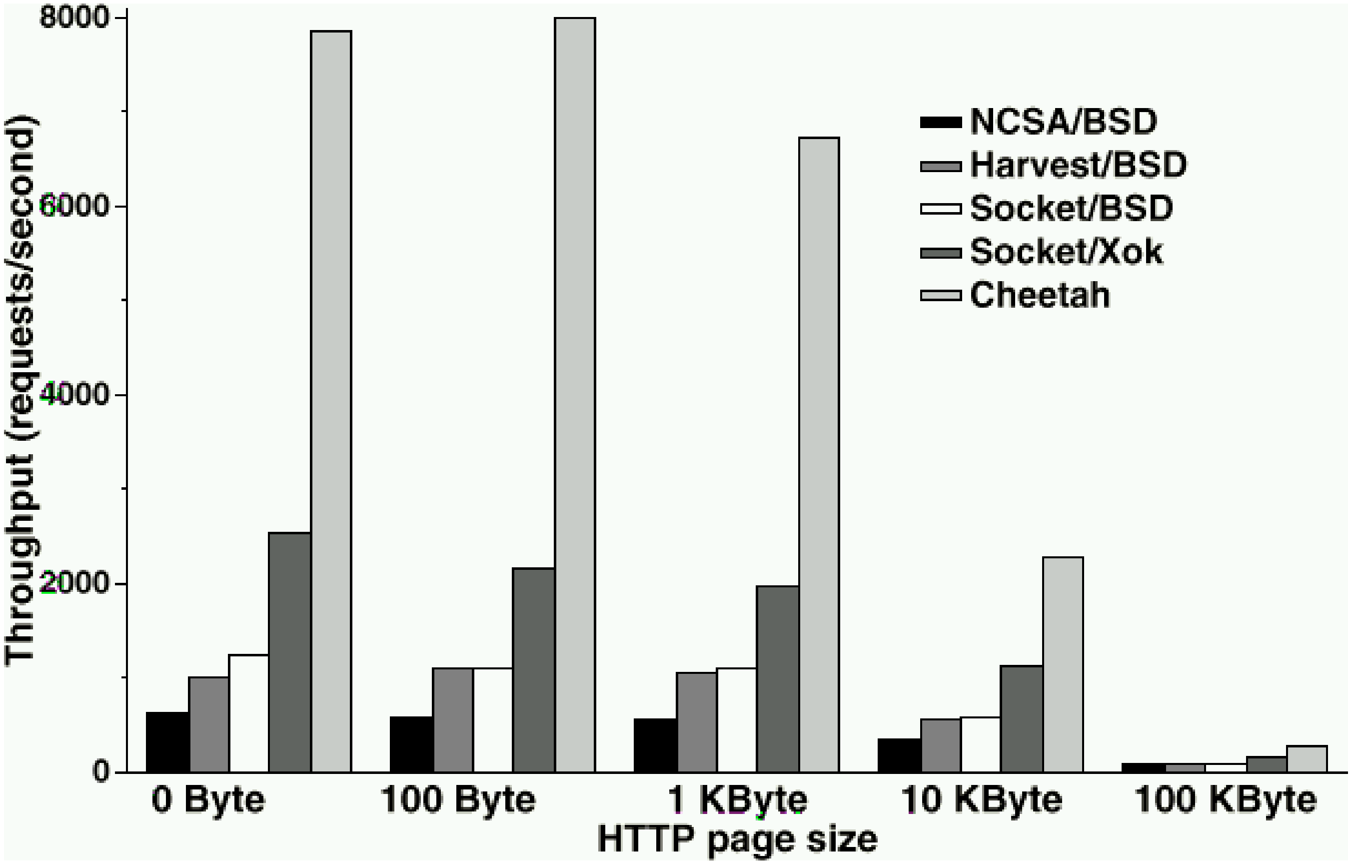
That is: two bus crossovers

Exokernels: Web Server Example

- Exokernel and Cheetah:
  - "File system" doesn't store files, stores packet-body streams
    - Data blocks are collocated with pre-computed data checksums
  - Header is finished when the data is sent out, taking advantage of the ability of TCP check-sums to be "patched"
  - This saves the system from recomputing a check-sum, saves processing power



Exokernels: Cheetah Performance



Exokernels

- Advantages:
  - + Extensible: just add a new ”operating system library”
  - + Fast?: Applications intimatly manage hardware, no obstruction layers
  - + Safe: Exokernel allows safe sharing of resources
- Disadvantages:
  - – To take advantage of Exo, basically writing an OS for each app
  - – Nothing about moving an OS into libraries makes it easier to write
  - – Slow?: Many many small syscalls instead of one big syscall
  - – send_file(2) - Why change when you can steal?
  - – Requires policy: despite assertions to the contrary

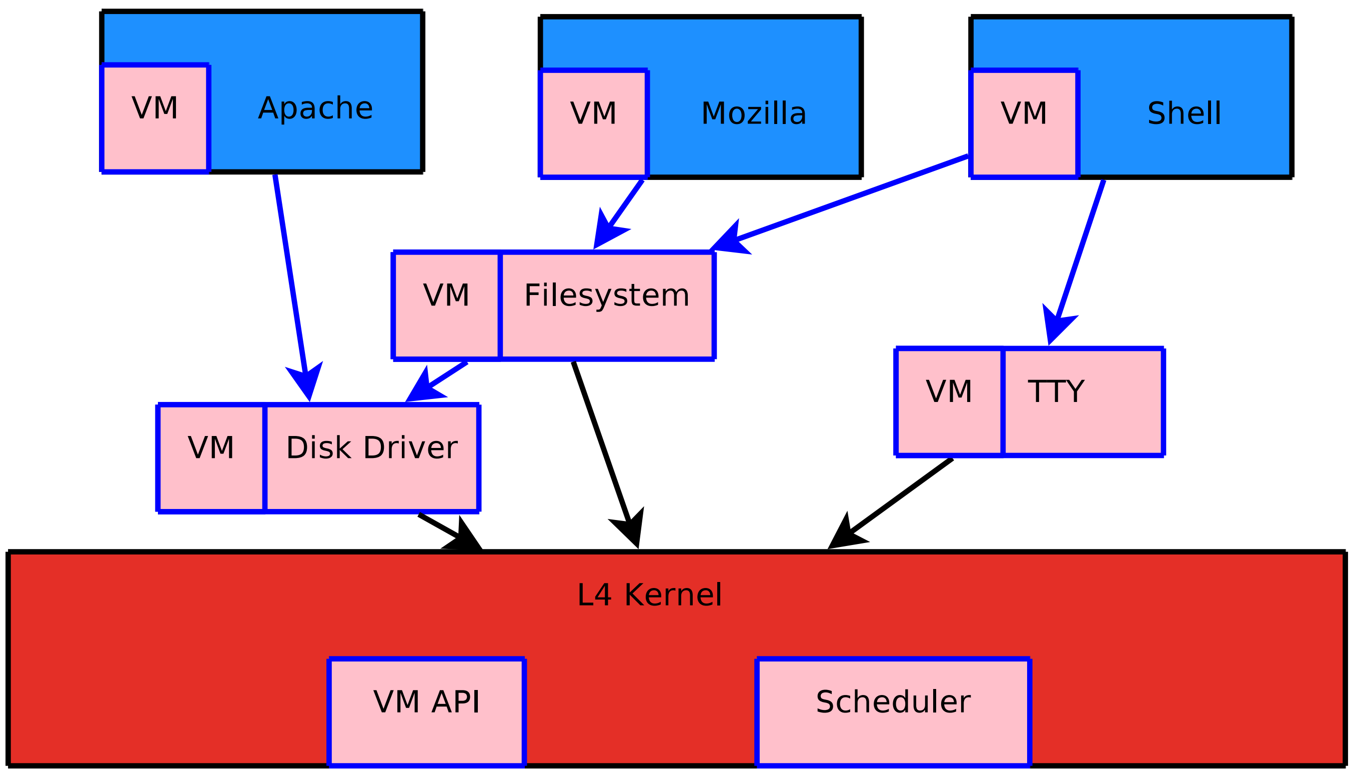
Exokernels

- Xok development is mostly over
- Torch has been passed to L4

More Microkernels (L4)

- In practice Exokernels still has some abstractions
- Exokernel still missing some abstractions that seem necessary
- But we like small: correctness $\propto$ 1/code_size
- Then what do we need?
- The RIGHT set of minimal abstractions (IPC, and VM API)

More Microkernels (L4)



More Microkernels (L4)

L4

- Syscalls < 20
  - memory_control, start_thread, IPC (send/recv on stringItem, Fpage)
- Lines of trusted code $\approx 37,000$
- Caveats - one arch, nearly no drivers (though none necissary)

More Microkernels (L4)

- Idea - a truly minimal kernel:
  - Kernel provides minimal VM abstraction (protection domains)
  - Kernel provides processor multiplexing (avoiding DDOS)
  - Kernel provides synchronous IPC (not Mach IPC™)
  - Kernel Doesn't provide device drivers, so we can have untrusted ones
  - like Exo: implement OS in libraries for mere abstractions
    * Fork, Exec, Filesystem Interface, VM interface
  - new: Implement OS in processes for required protection
    * Filesystem, Global Namespace, Device Drivers
  - For fun and profit: `http://os.inf.tu-dresden.de/L4/`

Microkernel OS'n (GNU Hurd Project)

- GNU Hurd Project:
  - Hurd stands for 'Hird of Unix-Replacing Daemons' and Hird stands for 'Hurd of Interfaces Representing Depth'
  - GNU Hurd is the FSF's kernel (Richard M Stallman)
  - Work began in 1990 on the kernel, has run on 10's of machines
  - Hurd/Mach vaguely runs, so abandoned in favor of Hurd/L4
  - Hurd/L4 suspended after a particular OS TA (and a former OS TA) tried to write their IPC layer.
  - Ready for mass deployment Real Soon Now™

Microkernel OS'n (L4Linux, DROPS)

- L4Linux - run Linux on L4
  - You get Linux, but a bit slower
  - You get multiple Linux's at a time
  - You get a realtime microkernel too
- DROPS - a realtime OS for L4
  - Realtime, and minimal
  - No security
- Combine the two for a realtime OS and linux... (mostly dead)

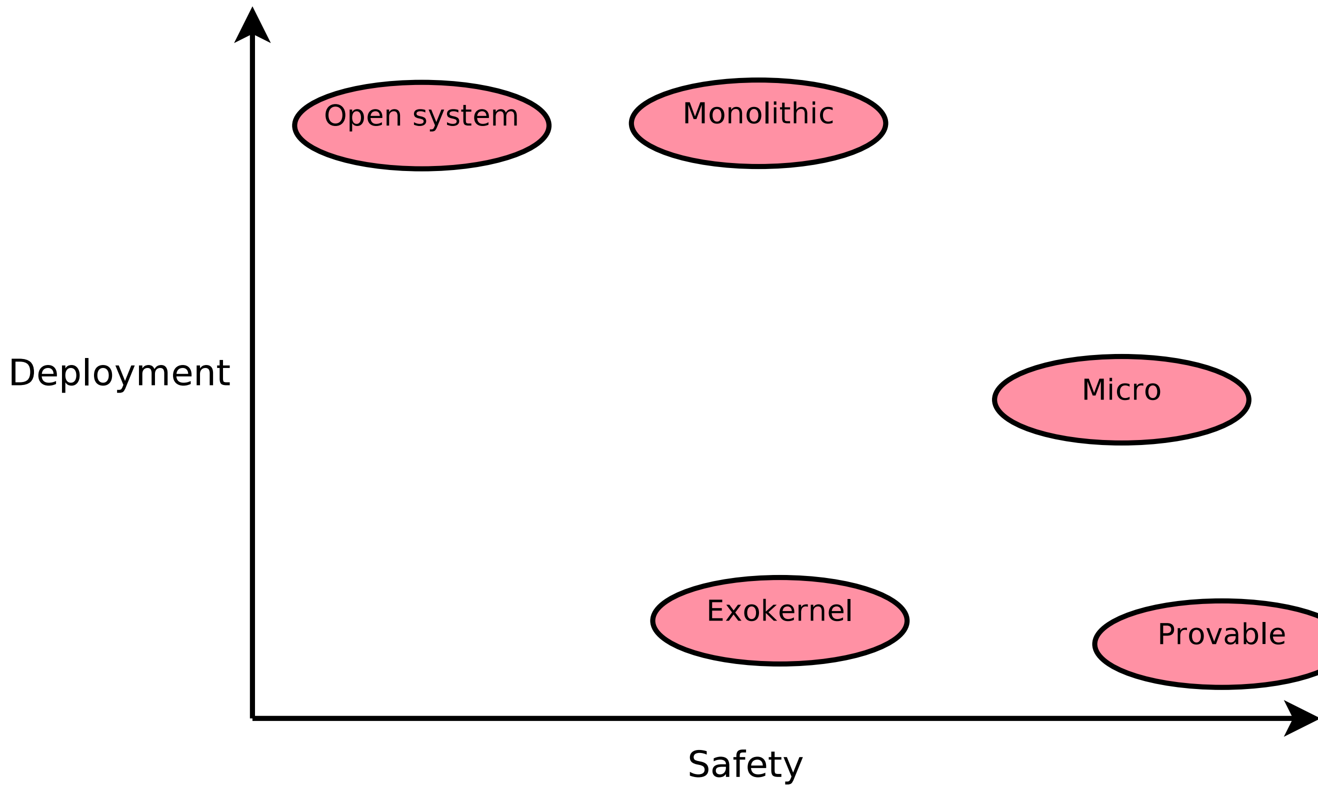
More Microkernels (L4)

- Advantages:
  - + Fast as hypervisor, similar to Mach (L4Linux 4% slower than Linux)
  - + VERY good separation (if we want it)
  - + Supports multiple OS personalities
  - + Soft realtime
- Disadvantages:
  - – Recreated much of Mach, but smaller, entails same problems
  - – Still notable missing abstraction: capabilities (more on this shortly)
  - – No Micro-OS written for it with protection boundaries
  - – Still untested with a multiserver topology

Microkernel OS'n

- The literature has between 5 and 50 percent overhead for microkernels
  - See *The Performance of μ-Kernel-Based Systems*
    - http://os.inf.tu-dresden.de/pubs/sosp97/

Summing Up



Deployment
Open system
Monolithic
Micro
Exokernel
Provable
Safety

Summing Up

- So why don't we use microkernels or something similar?
- Say we have a micro-(or exo)-kernel, and make it run fast
  - We describe things we can do in userspace faster (like Cheetah)
  - Monolithic developer listens intently
  - Monolithic developer adds functionality to his/her kernel (send_file(2))
  - Monolithic kernel again runs as fast or faster than our microkernel
- So, if monolithic kernel runs as fast, why bother porting to new OS?
  - Stability - new device drivers break Linux often, we use them anyway
  - The story above can get painful, hard to write, hard to debug

Summing Up

What's the moral?

- There are many ways to do things
- Many of them even work

Further Reading

- Jochen Liedtke, On Micro-Kernel Construction
- Willy Zwaenepoel, Extensible Systems are Leading OS Research Astray
- Michael Swift, Improving the Reliability of Commodity Operating Systems
- An Overview of the Singularity Project, Microsoft Research MSR-TR-2005-135
- Harmen Hartig, *The Performance of* μ*-Kernel-Based Systems*

Further Reading

CODE: (in no particular order)

- Minix (micro)
- Plan 9 (midsized)
- NewOS/Haiku (micro'ish)
- L4 pistachio (micro)
- Solaris (monolithic)
- (net/dragonfly)BSD (monolithic)