# Operating System Structure

Joey Echeverria joey42+os@gmail.com
modified by: Matthew Brewer mbrewer@andrew.cmu.edu
rampaged through by: Dave Eckhardt staff-410@cs.cmu.edu

December 5, 2007

# Synchronization

- P4 - due tonight
- Homework 2 - out today, due Friday night
- Book report - due Friday night (late days are possible)
- Friday lecture - exam review
- Exam - room change in progress; discard any cached values

# Outline

- OS responsibility checklist
- Kernel structures
  - Monolithic kernels
    - Kernel extensions
  - Open systems
  - Microkernels
  - Provable kernel extensions
  - Exokernels
  - More microkernels
- Final thoughts

# OS Responsibility Checklist

- It's not so easy to be an OS:

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

# Protection Boundaries

- Protection is “Job 1”
  - Protect processes from each other
  - Protect crucial services (like the kernel) from processes
- Notes
  - Implied assumption: everyone trusts the kernel
  - Kernels are complicated
    - See Project 3 :)
    - Something to think about
      - Full OS is millions of lines of code
      - Very roughly: correctness $\propto$ 1/code_size

# Abstraction Layer

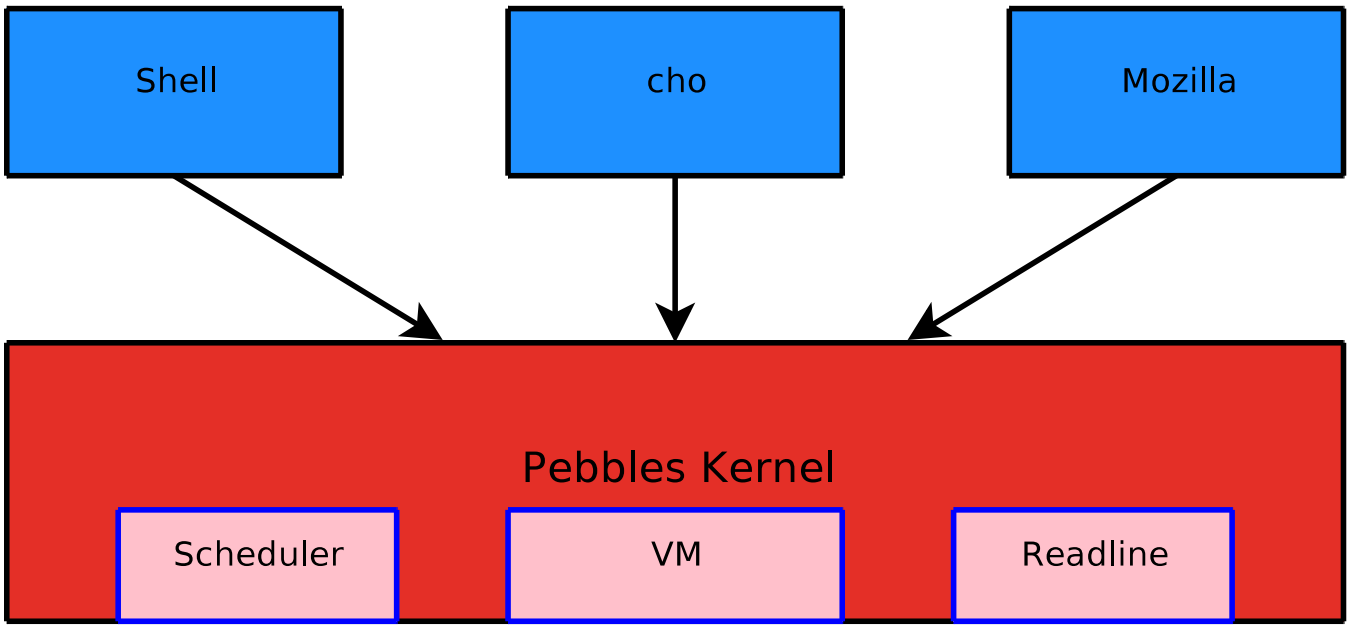
- Present “simple”, “uniform” interface to hardware
- Applications see a well defined interface (system calls)
  - Block Device (hard disk, flash card, network mount, USB drive)
  - CD drive (SCSI, IDE)
  - tty (teletype, serial terminal, virtual terminal)
  - filesystem (ext2-4, reiserfs, UFS, FFS, NFS, AFS, JFFS2, CRAMFS)
  - network stack ($\{$Unix,Internet,Appletalk$\} \times \{$stream,message$\}$)

# Hardware Multiplexer

- Each process sees a “computer” as if it were alone
- Requires division and multiplexing of:
  - Memory
  - Disk
  - CPU
  - I/O in general (network, graphics, keyboard etc.)
- If kernel is multiplexing it must also apportion
  - Fairness, priorities, classes? - HARD problems!!!

# Monolithic Kernels

- Pebbles Kernel

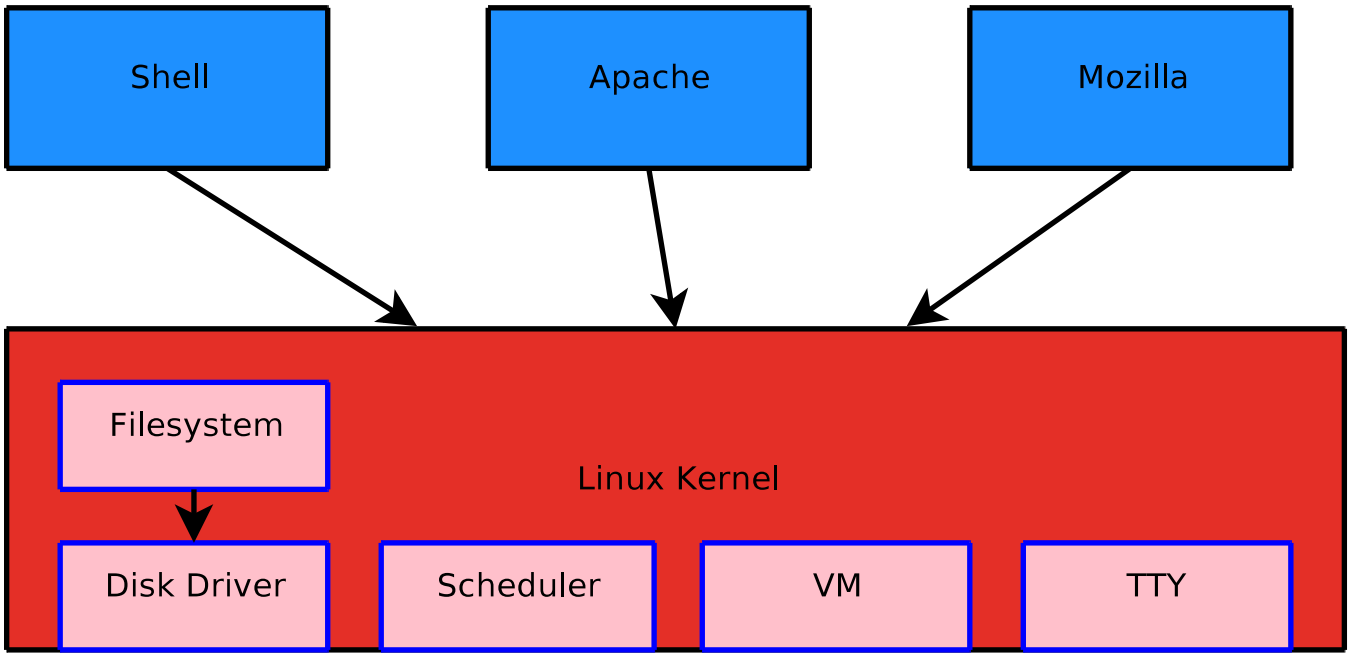


# Monolithic Kernels

- Consider the lowly Pebbles kernel
  - Syscalls $\approx 20$
    - fork(), exec(), cas2i_runflag(), yield()
  - Lines of trusted code $\approx 3000$ (to 24000)

# Monolithic Kernels

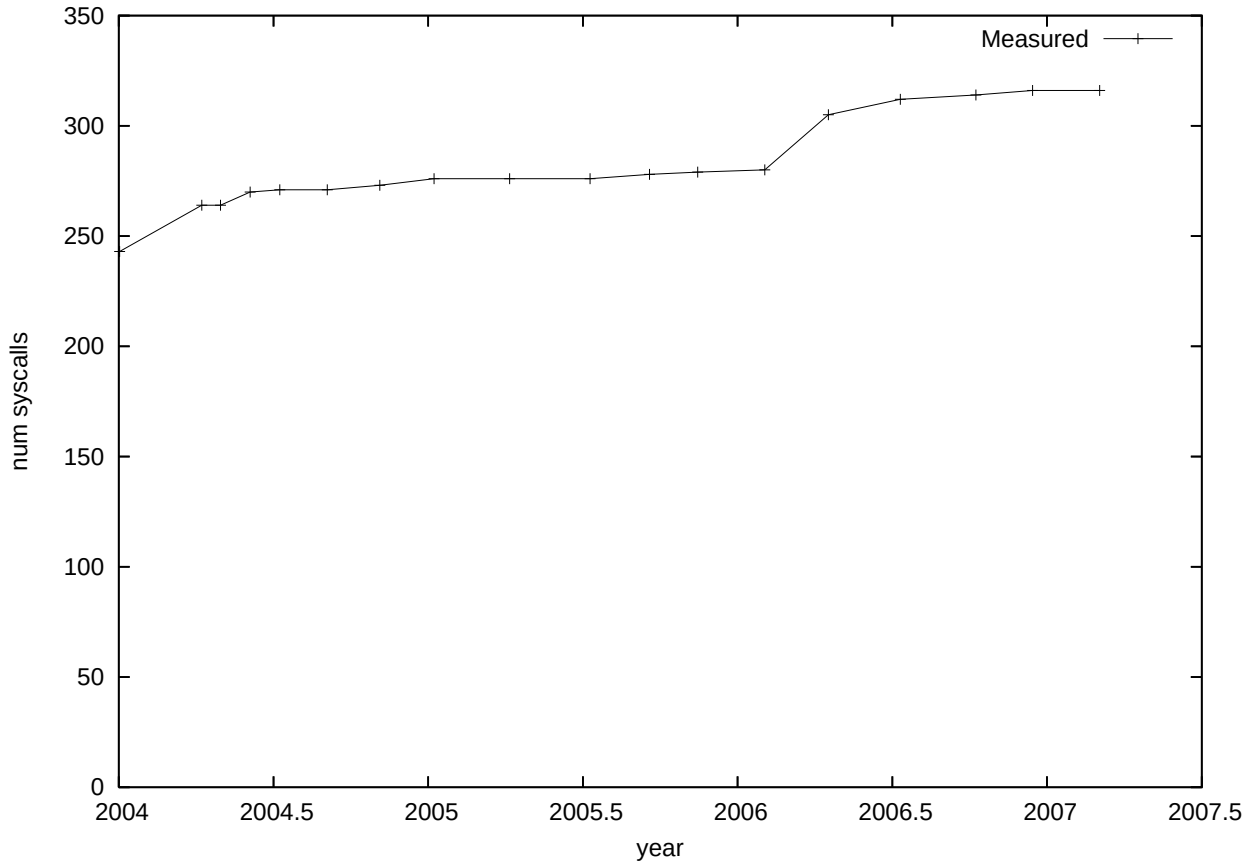
- Linux Kernel... similar?



# Monolithic Kernels

- Now consider a recent Linux kernel
  - Syscalls: $\approx 243$ in 2.4, and increasing fast
    - fork(), exec(), read(), getdents(), ioctl(), umask()
  - Lines of trusted code $\approx 7$ million as of May 2007
    - $\approx 200,000$ are just for USB drivers
    - $\approx 15,000$ for USB core alone
    - Caveats - Many archs/subarchs, every driver EVER

# Monolithic Kernels



# Monolithic Kernels

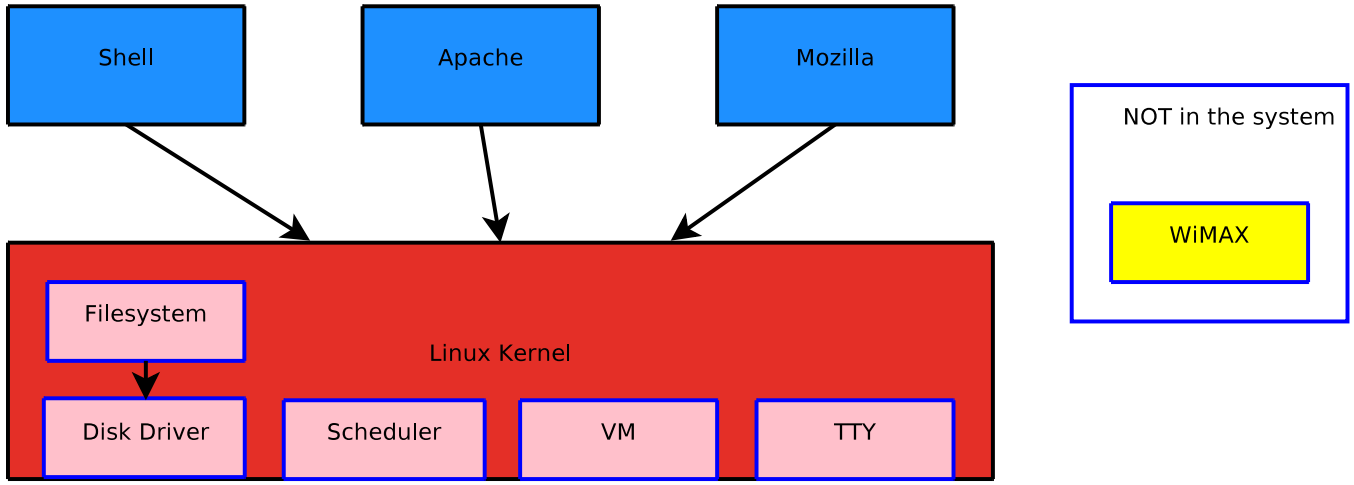
- Advantages:
  - + Well understood
  - + Good performance
  - + High level of protection between applications
- Disadvantages:
  - – No protection between kernel components
  - – LOTS of code is in kernel
  - – Not (very) extensible
- Examples: UNIX, Mac OS X, Windows NT/XP, Linux, BSD, i.e., common

# Loadable Kernel Modules

- Problem - Roger has a WiMAX card, and he wants a driver
- Dave doesn't want a (large, unstable) WiMAX driver muddying his kernel
  - Probing for the nonexistent hardware at boot time may crash his machine!
- Solution - kernel modules
  - Special binaries compiled "along 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

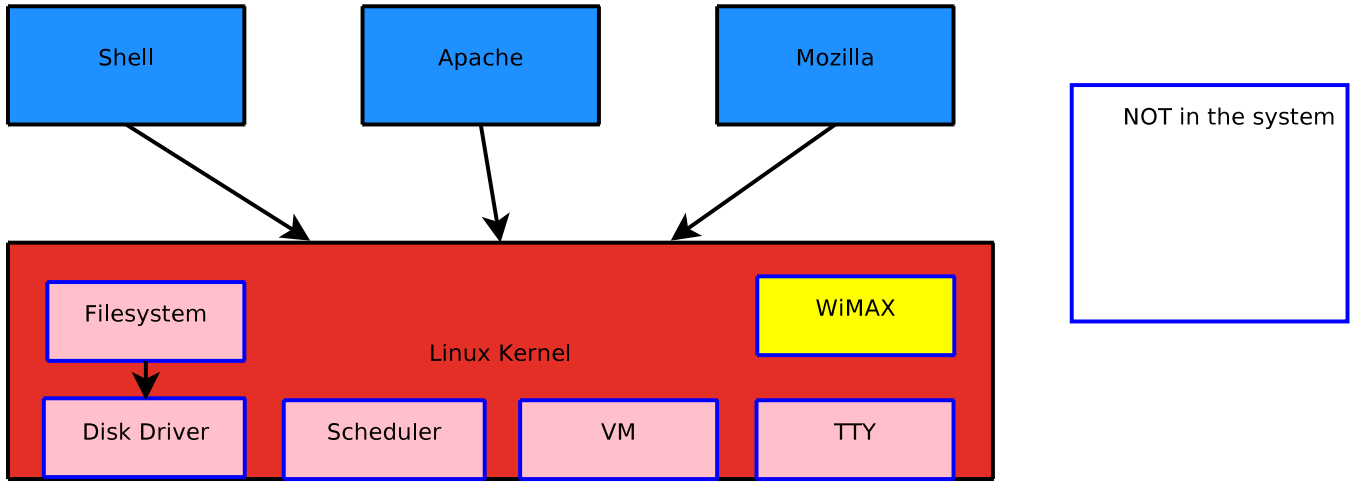
# (Loadable) Kernel Modules

Linux Kernel



# (Loadable) Kernel Modules

Linux Kernel with WiMAX module



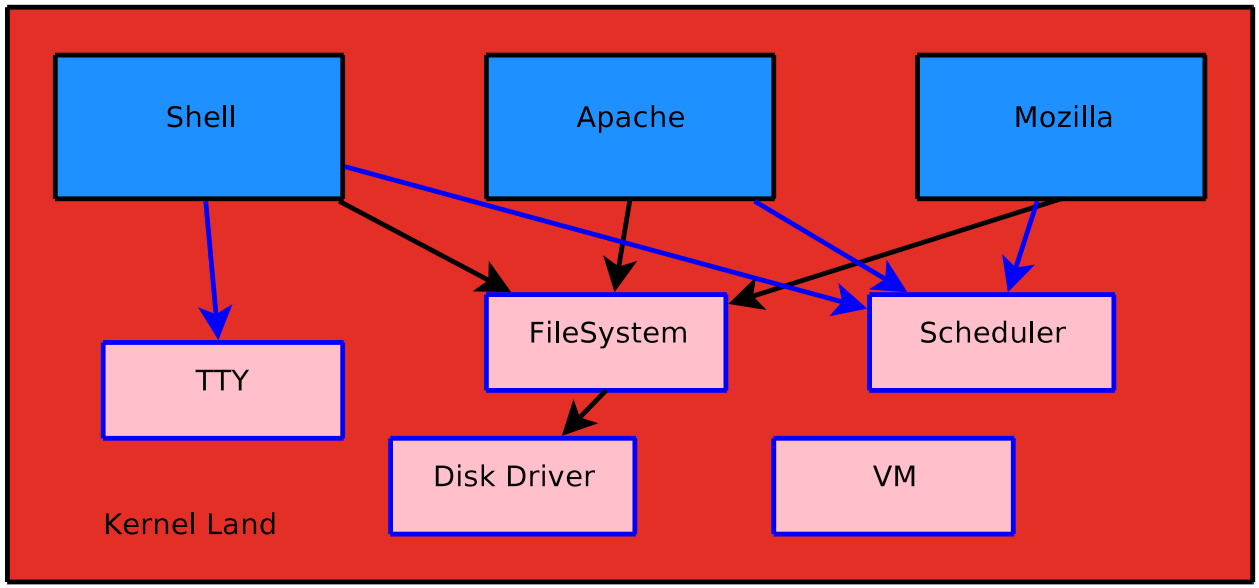
# Kernel Extensions

- Advantages
  - + Can extend kernel
  - + Extensions run at “full speed” once loaded into kernel
- Disadvantages
  - – Adding things to kernel can break it
  - – Must petition system administrator to get modules added
- Any alternatives?

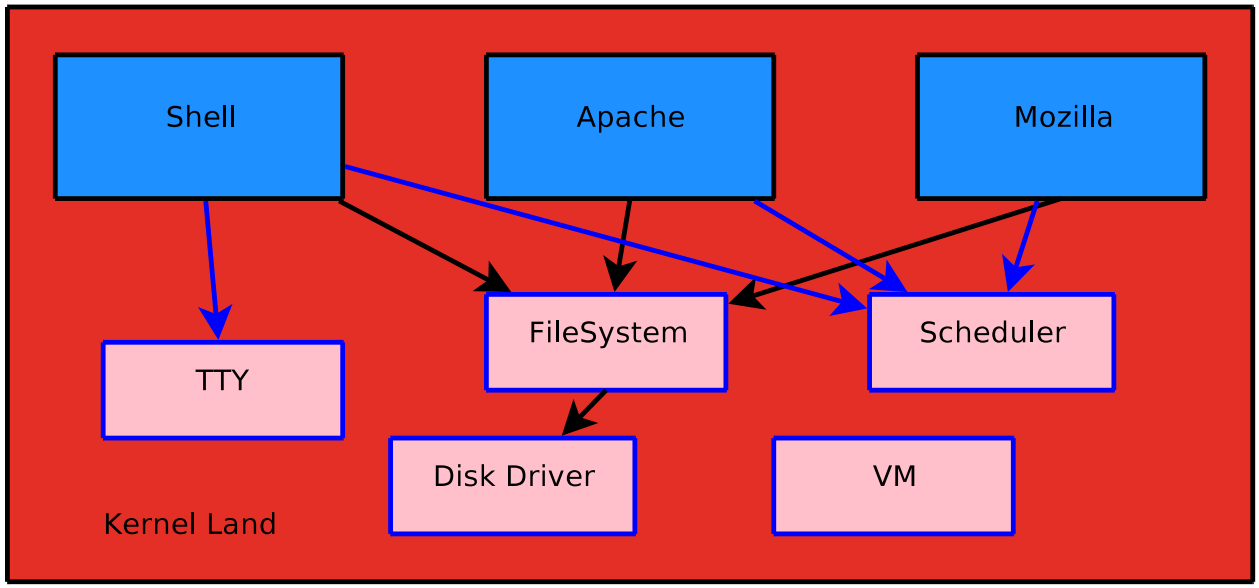
# Musings

- Monolithic kernels run reasonably fast, and can be extended (at least by root)
- Some pesky overheads, though...
  - System call: $\approx$ 90 cycles invoke PL0 code on x86
  - Address space: Context switch dumps TLB - more painful over time
- Protection looks expensive...do we need it?

# 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 3.1, 95, 95, ME, 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
    * Mac OS and Palm OS each had associated extensions *industry*
  + Can work well in practice
  + Lack of abstractions can make real-time systems easier

- Disadvantages:
  – No protection between kernel and/or applications
  – Not particularly stable
  – Composing extensions can result in unpredictable behavior

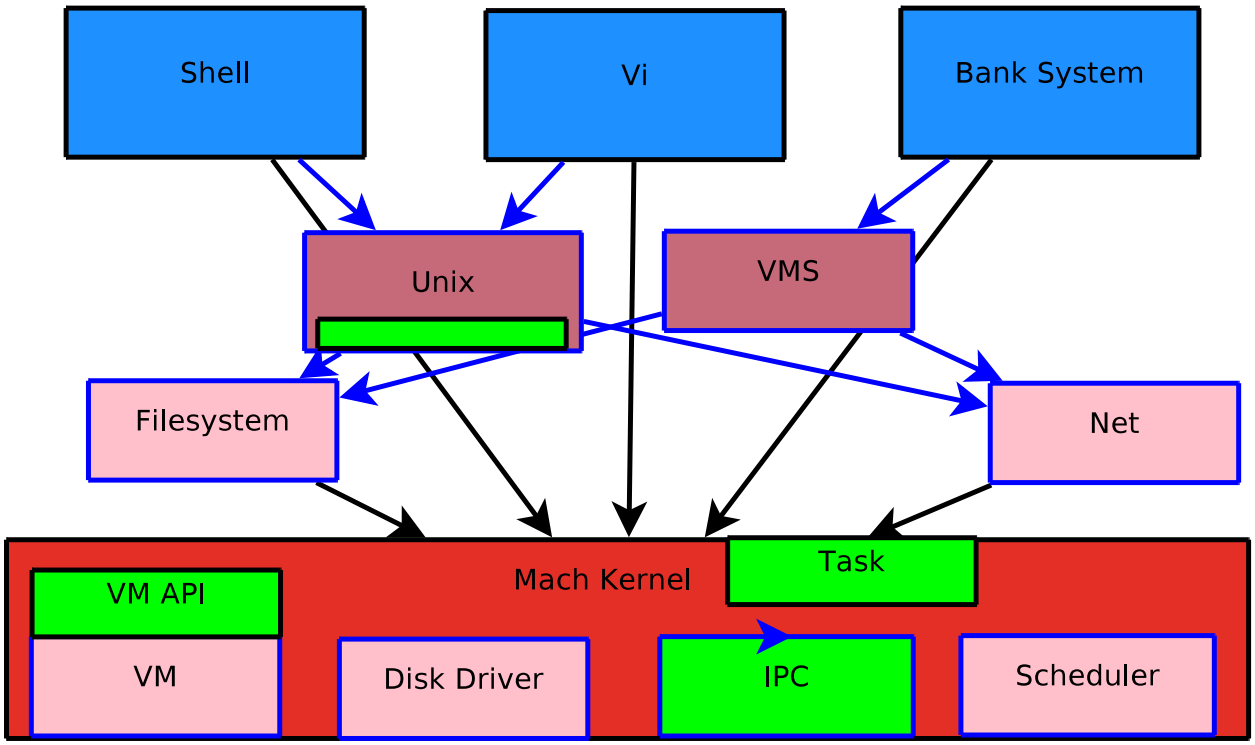
# Musings

- Monolithic Kernels
  - Extensible (by system administrator)
  - User programs mutually protected
  - No internal protection - makes debugging hard, bugs CRASH
- Open Systems
  - Extensible (by everyone)
  - Fast, flexible
  - No protection at all - unstable, plus can't be multi-user
- Is there a way to get user extensibility *and* inter-module 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*

# Mach “Multi-Server” Vision



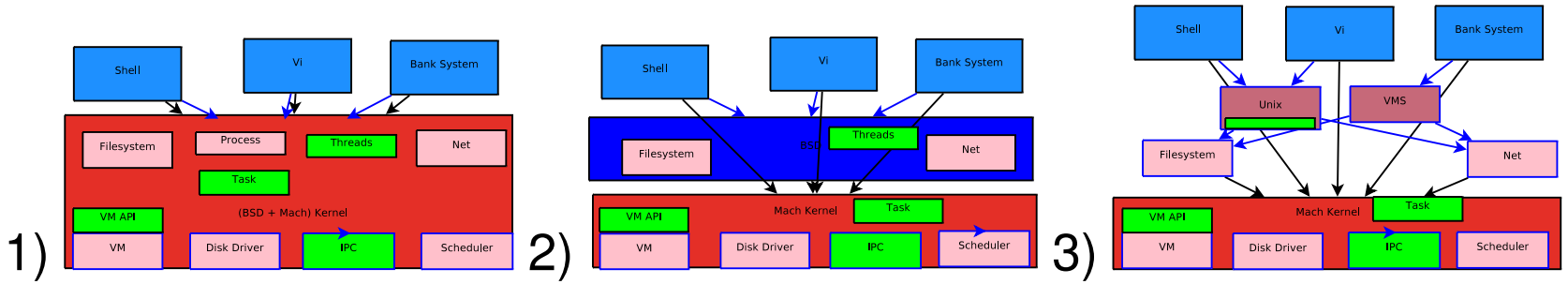
# Microkernels (Mach)

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: BSD 4.1 Unix with new VM plus IPC, threads, SMP
  2. Mach 3: Saw kernel in half and run Unix as “single server”
  3. Mach 3 continued: decompose single server into smaller servers



# Microkernels (Mach)

- Results

1. Mach 2: completed in 1989
   - "Unix" with SMP, kernel threads, 5 architectures
   - Used for Encore, Convex, NeXT, and subsequently OS X
   - Success!
2. Mach 3: Finished(ish)
   - Unix successfully removed from kernel (!!)
   - Ran some servers & desktops at CMU, a few outside
3. Mach 3 continued: ...?
   - Multi-server systems: "Mach-US," Open Software Foundation
   - Not really deployed to users

# Microkernels (Mach 3)

- Advantages:
  + Strong protection (most of “Unix” outside of kernel)
  + Flexibility (special non-kernel VM for databases)

- Disadvantages:
  – Performance
    * It looks like extra context switches and copying would be expensive
    * Mach 3 ran slow in experiments
    * Some performance tuning, but not as much as commercial Unix distributions
  – Kernel still surprisingly large -
    “It’s not micro in size, it’s micro in functionality”

# Mach Microkernel as a hypervisor

- IBM's rationale
  - Our mainframes have done virtualization since the 1970's...
  - Can Mach microkernel be a multi-OS platform for tiny little machines?

- IBM Workplace OS (1991-1996)
  * One kernel for MS-DOS, OS/2, MS Windows, OS/400, and AIX
  * One kernel for x86 and PowerPC
  * "One ring to rule them all..."
    * *Much* time consumed to run MS-DOS, OS/2, and Unix on x86 kernel
    * But people wanted x86 hardware to run MS Windows
    * But Apple wanted PowerPC hardware to run MacOS
    * But IBM decided not to really sell desktop PowerPC hardware

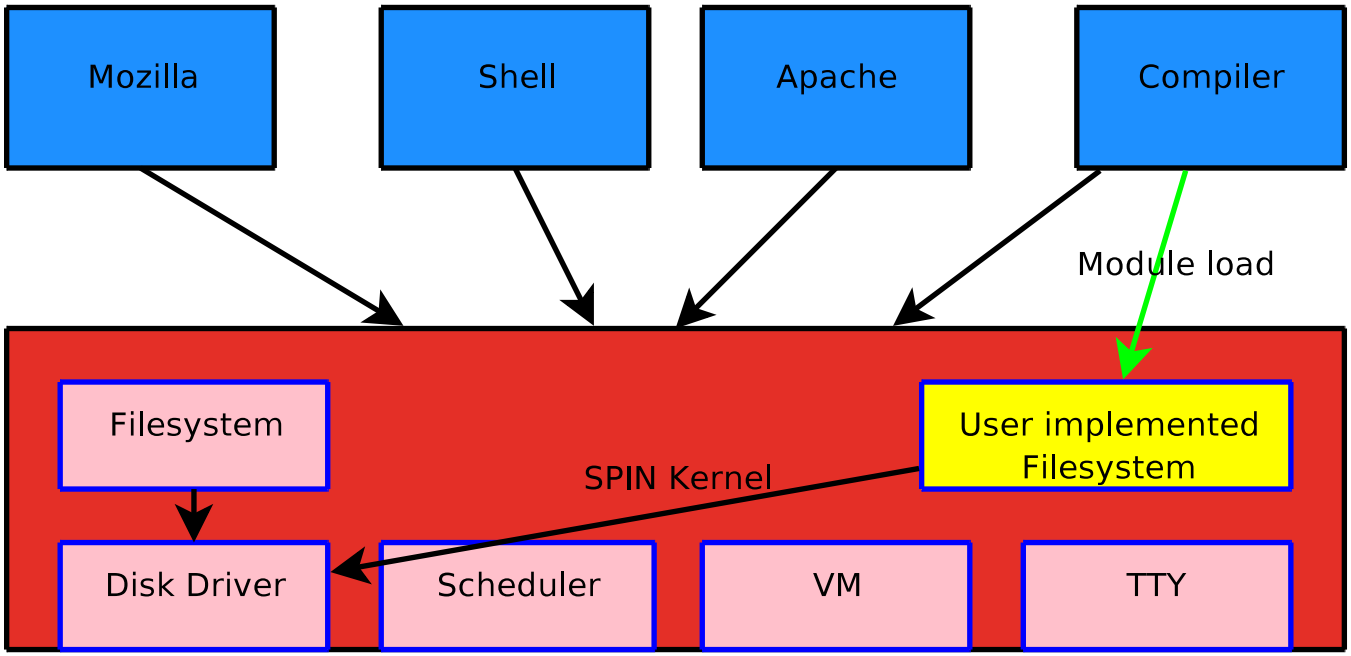
# Microkernels (Mach)

- Things to remember about Mach history
  - Mach 3 == microkernel, Mach 2 == monolithic
  - Code ran slow at first, then everyone graduated
  - Demonstration of microkernel feasibility
  - Performance cost of stability/flexibility unclear
  - (Mac OS X is Mach 2, not Mach 3)
- Other interesting points
  - Other microkernels from Mach period: ChorusOS, QNX
  - ChorusOS, realtime kernel out of Europe, now open sourced by Sun
  - QNX competes with VxWorks as a commercial real-time OS

# Musings

- We want an extensible OS
- Micro-kernel protection and scheduling seem slow
- We don't want unsafe extensibility
- Can we *safely* add code to a monolithic kernel?

# Provable Kernel Extensions

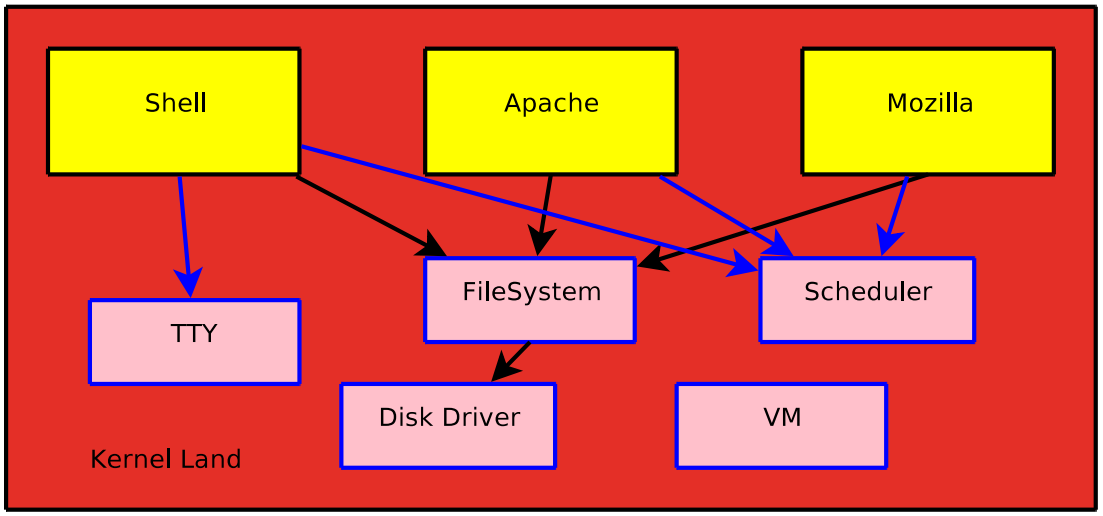


# Provable Kernel Extensions

- *Prove* the code is safe to add to kernel
- Various (very conservative) approaches to “gatekeeper”
  - Interpreter (CMU: Packet filters)
    - Slow but clearly safe - can even bound time
  - Compiler-checked source safety (UW: Spin: Modula-3)
    - Faster code, must trust compiler
  - Kernel-verified binary safety (CMU: Proof-carrying code)
    - Language agnostic - in theory any compiler can generate proofs
- Safe? If you trust base kernel and gatekeeper.

# Provable Everything

What if *everything* were a proven kernel extension?



# Provable Everything

- Advantages:
  - + Freely extensible by users (every application is a kernel extension)
  - + Good performance because everything is in the kernel
  - + "Provably" safe
- Disadvantages:
  - – Effectiveness strongly dependent on quality of proofs
  - – Some proofs are hard, some proofs are impossible!
  - – Proof *checking* can be slow
  - – Code simple enough to prove correct might cost more than protection boundaries
  - – Current research: MSR's "Singularity" OS

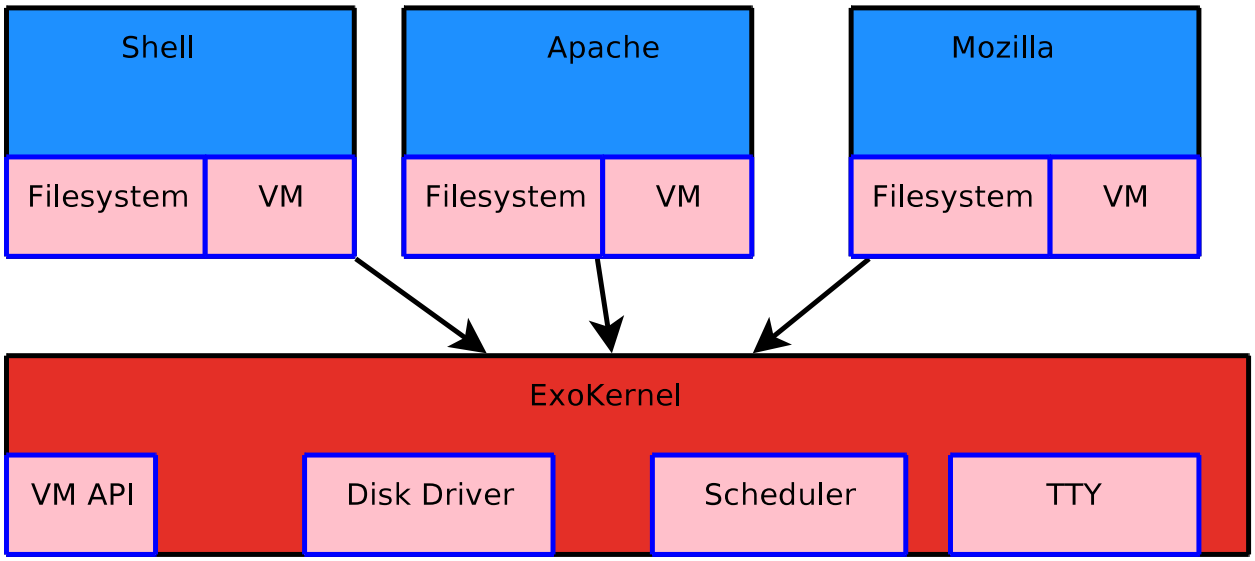
# Musings

- Monolithic kernel
  - Extensibility limited (kernel modules are privileged)
  - Large “base system” is mandatory for all users
- Open systems: unstable
- “Abstraction” microkernels (Mach)
  - Performance concerns; Were the best kernel abstractions chosen?
- Proof systems: feasible for complex applications?
- If applications control system, can optimize for their usage cases

# Exokernels

- Allow application writers full control over hardware resources
- Kernel's job is to safely share hardware *without abstractions*
  - Application knows page-table format
  - Application flushes TLB when necessary
- Remove *all* of "operating system" from kernel, leaving threads and mini-VM
- Separates security and protection from the management of resources

# Exokernels (Xok/ExOS)



# Exokernel (Xok)

Xok

- Syscalls $\approx 120$
  - 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 $\rightarrow$ 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 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()

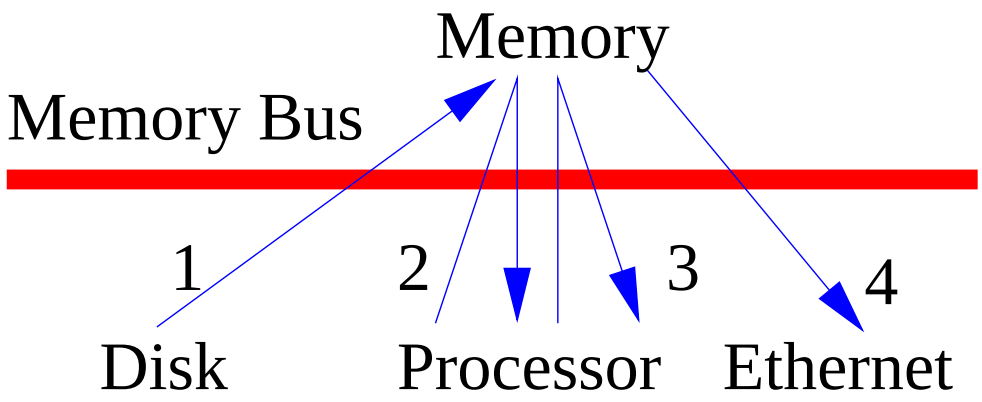
# Exokernels: COW fork()

- fork(), advanced:
  - Acquire a new, blank address space
  - Create copy-on-write table in each address space
  - Add R/O PTE's for my frames into the blank address space
  - Replace each of my PTE's with a R/O PTE
  - Flush TLB
  - *Application's* page-fault handler (like a signal handler) copies/re-maps
- Each process can have its own fork() optimized for it
  - If I know certain pages will fault, I can "pre-copy" exactly those pages

# Exokernels: Web Server Example

- Traditional kernel and web server:

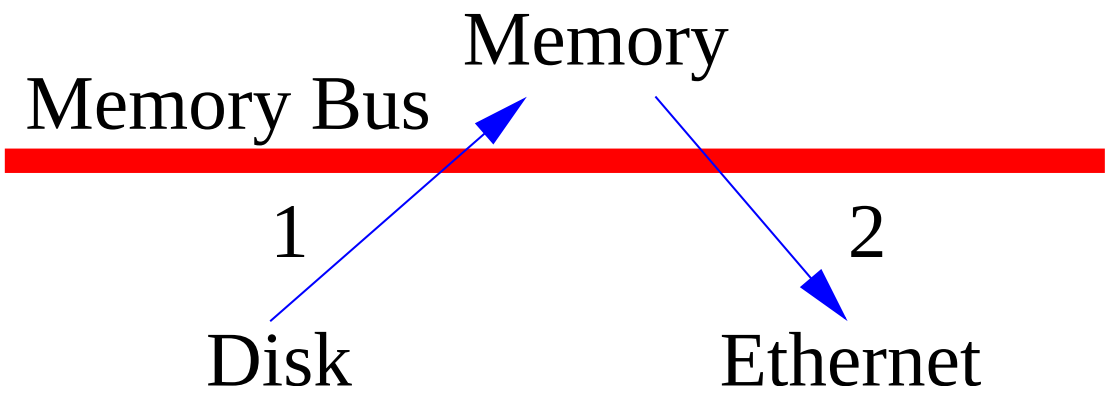
1. read() − copy from disk to kernel buffer
2. read() − copy from kernel to user buffer
3. send() − user buffer to kernel buffer
   −− data is check−summed
4. send() − kernel buffer to device memory




That is: six bus crossings

# Exokernels: Web Server Example

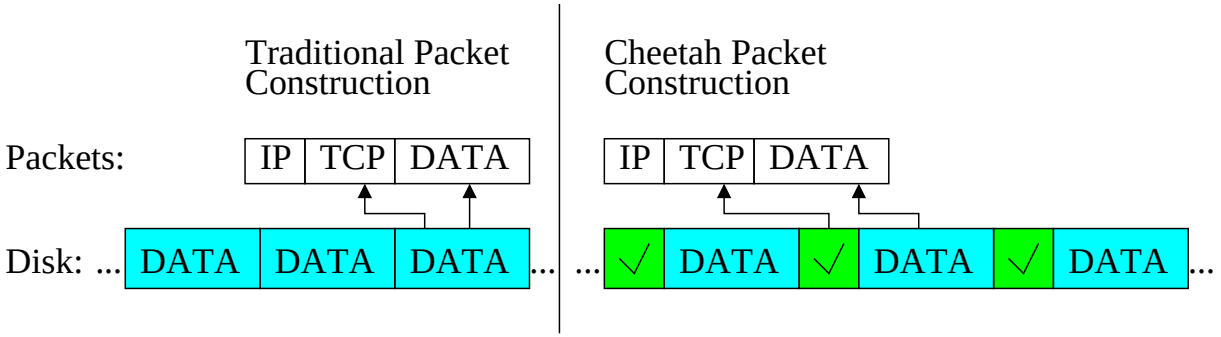
- What fundamentally needs to happen:
  1. Copy from disk to memory
  2. Copy from memory to network device



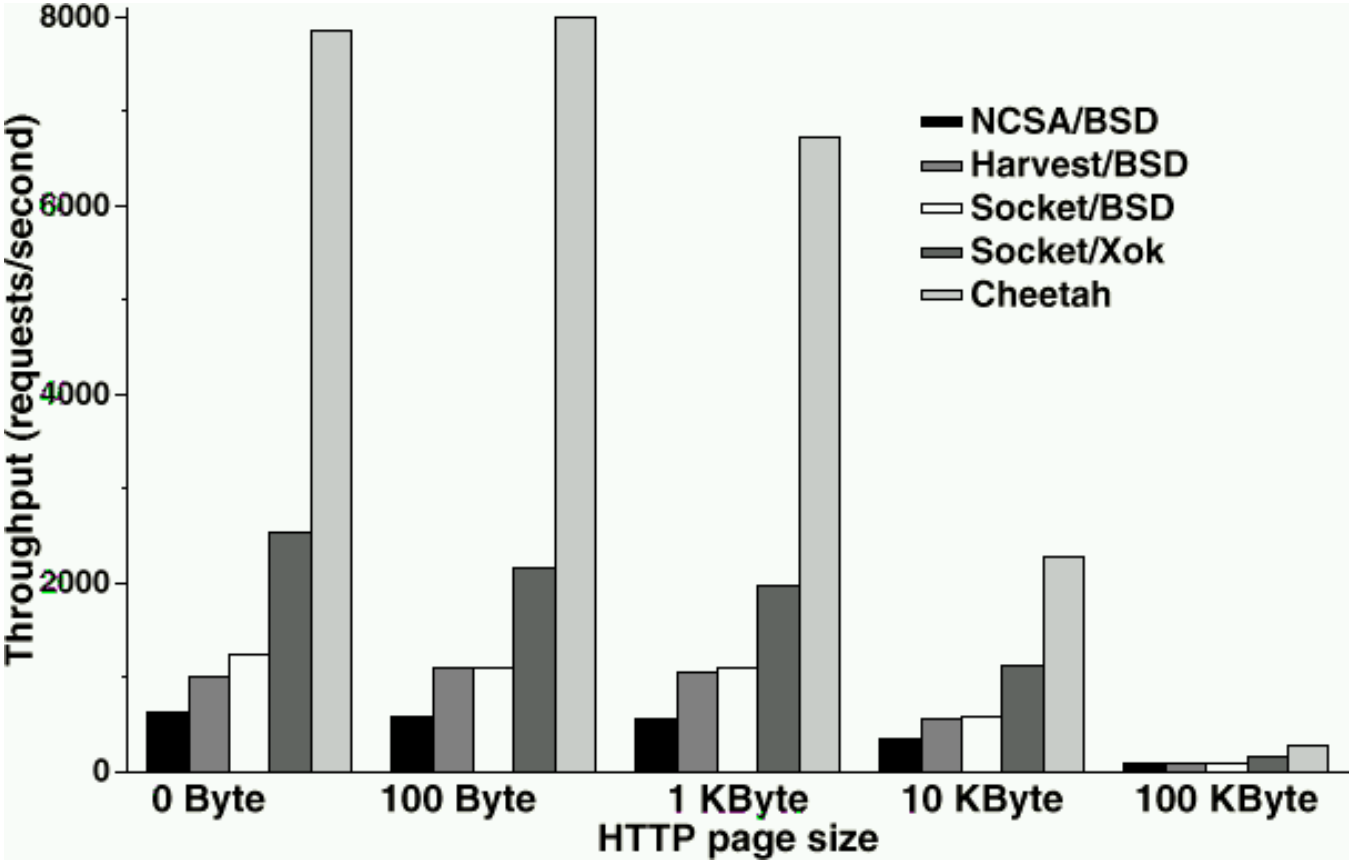

That is: two bus crossings

# Exokernels: Web Server Example

- Exokernel and Cheetah:
  - "File system" doesn't store files, stores packet-body streams
    - Data blocks are co-located with pre-computed data checksums
  - Header is tweaked before transmission (TCP checksums can be "patched")
  - No need to re-chunk file data into packets, checksum all data bytes



# Exokernels: Cheetah Performance



# Exokernels

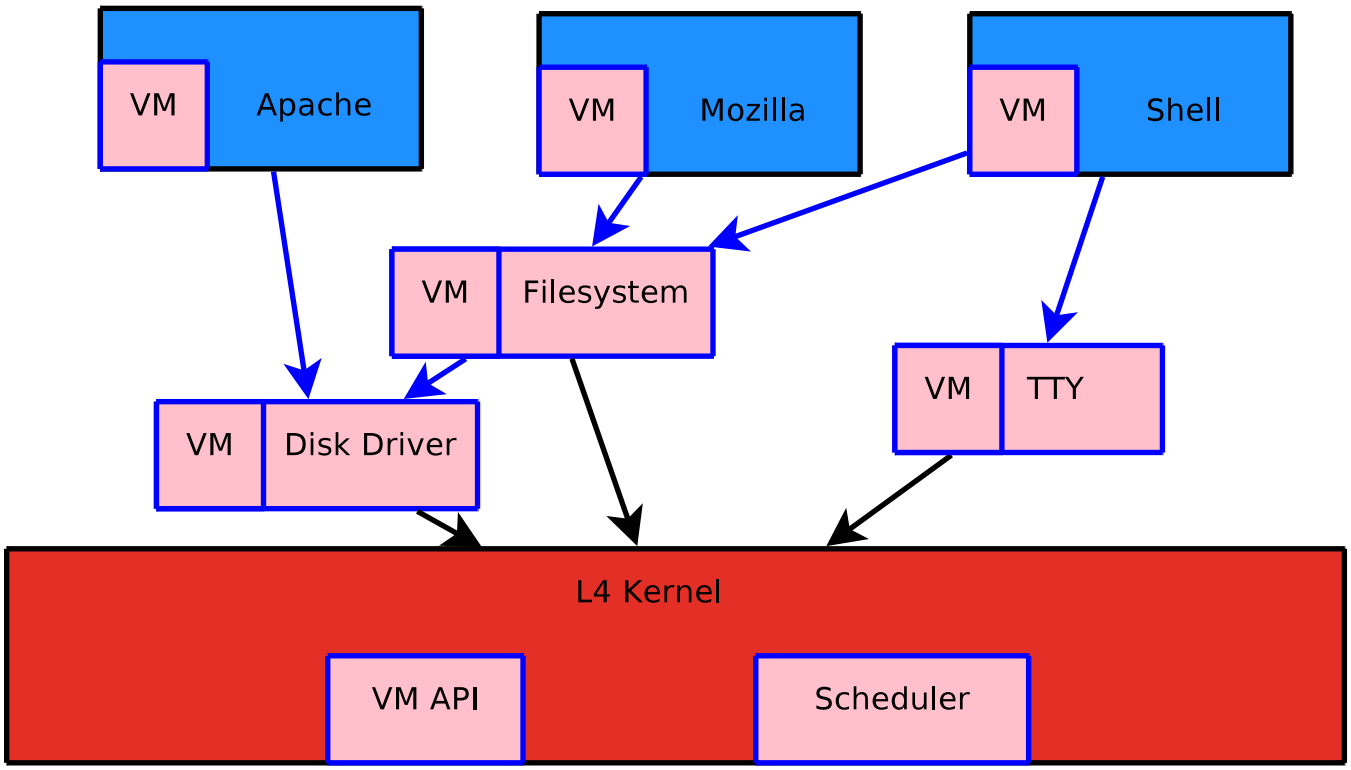
- Advantages:
  + Extensible: just add a new “operating system library”
  + Fast: Applications intimately manage hardware, no obstruction layers
  + Safe: Exokernel allows safe sharing of resources

- Disadvantages:
  – Taking advantage of Exo may mean 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
  – sendfile(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)



# More Microkernels (L4)

L4 - http://os.inf.tu-dresden.de/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 (add just what you need)

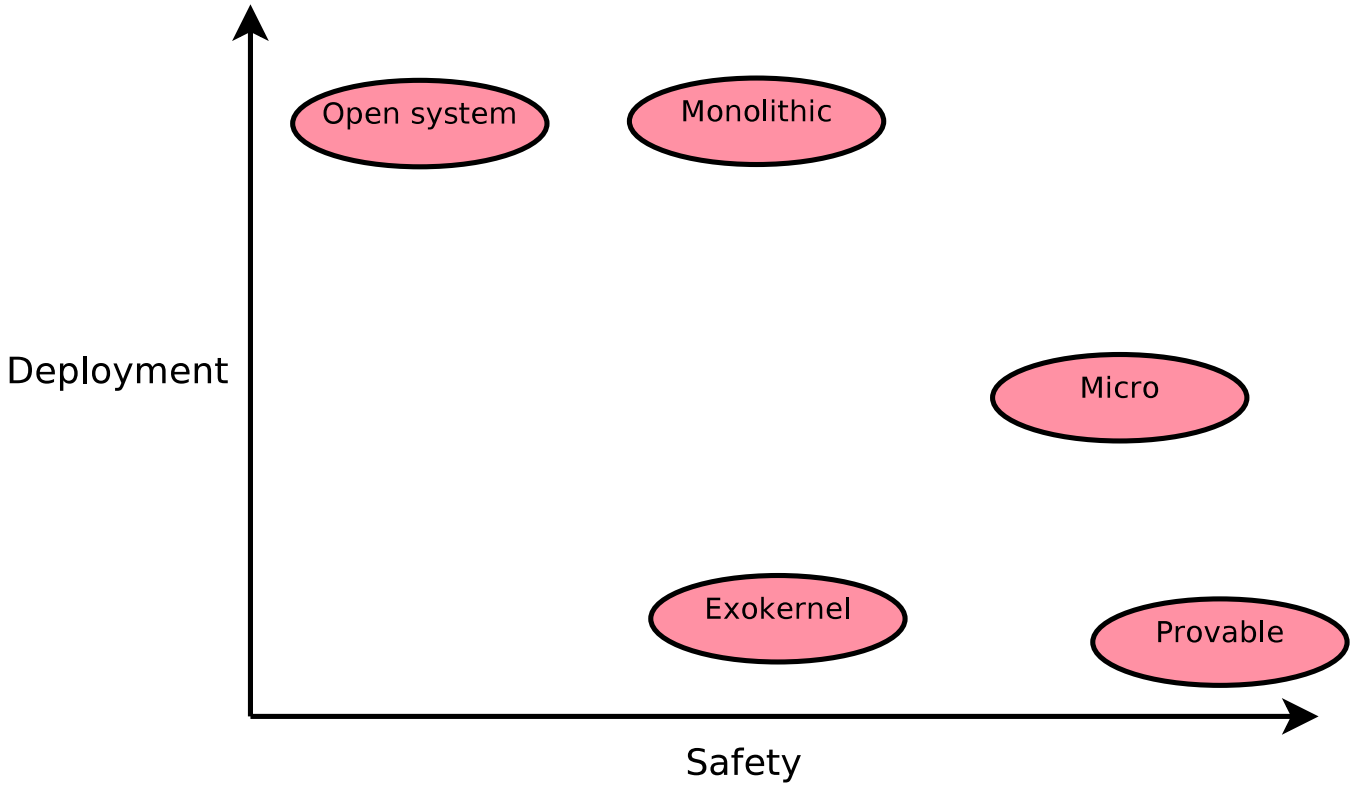
# Microkernel OS'n (L4Linux, DROPS)

- 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 real-time OS for L4
  - Realtime, and minimal (no inter-application security)
- Combine the two for a real-time 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 real-time
- Disadvantages:
  – Smaller than Mach at present
  – Still growing (capabilities, ...)
  – No experience with multi-server OS (how will it perform?)

# Summing Up


Open system
Monolithic
Micro
Exokernel
Provable
Deployment
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 (sendfile(2))
  - Monolithic kernel again runs as fast or faster than our microkernel
- If monolithic kernels run fast, why consider other organizations?
  - Stability - new device drivers break Linux often, we use them anyway
  - Complexity - when everything interacts, debugging a large kernel gets hard

# Summing Up

What's the moral?

- There are many ways to do things
- Many of them even work
- Surprisingly, we still haven't settled on a single notion of "kernel"

# 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 $\mu$-Kernel-Based Systems*

# Further Reading

CODE: (in no particular order)

- Minix (micro)
- Plan 9 (“right-sized”)
- NewOS/Haiku (micro’ish)
- L4 Pistachio (micro)
- Solaris (monolithic)
- NetBSD, DragonflyBSD (monolithic)