Operating System Structure

Joey Echeverria
joey42+os@gmail.com

May 1, 2006

Synchronization

- Not *that* kind of synchronization :)
- David Tarditi, Microsoft Research
  - An Overview of the Singularity Project
  - Today at 3:30pm
  - NSH 1305
- Homework 2 is available on the course web site
- Don't forget Faculty Course Evaluations

Overview

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

Motivations

- Operating systems have a hard job.
- Operating systems are:
  - Abstraction layers
  - Resource allocators
  - Protection boundaries
  - Resource Schedulers
  - Complicated

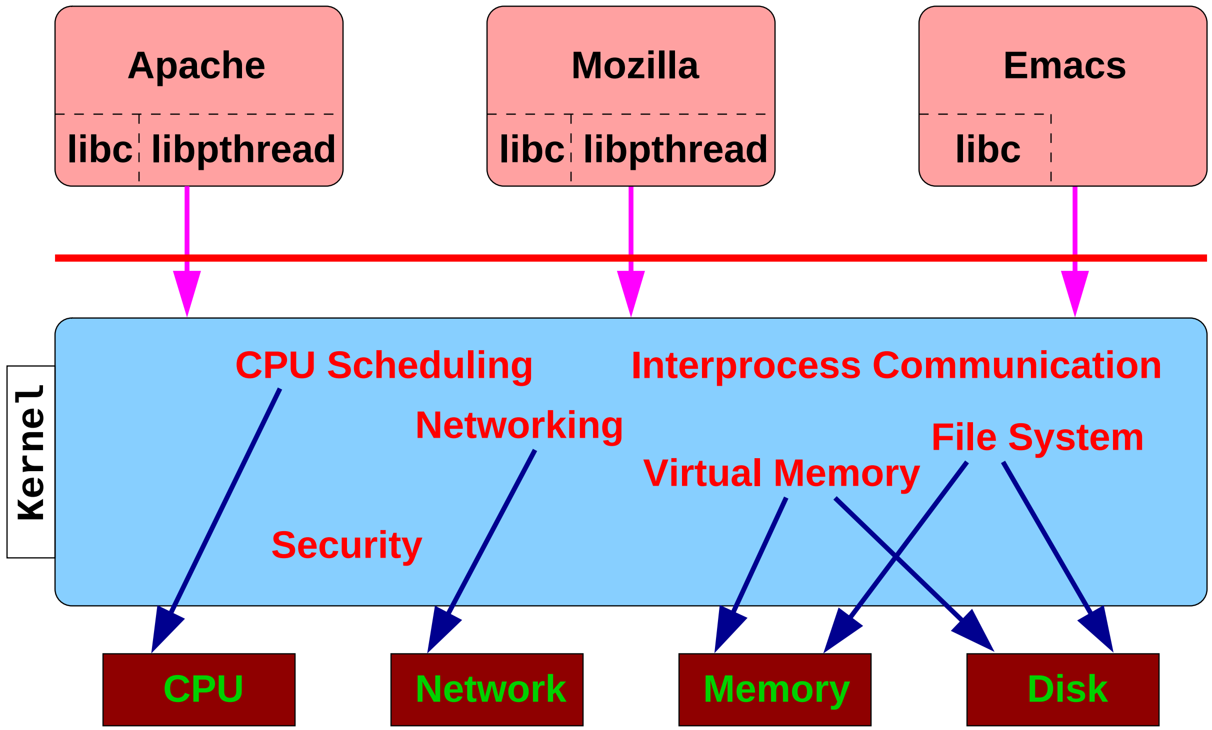
Motivations

- Abstraction Layer
  - Operating systems present a simplified view of hardware
  - Applications see a well defined interface (system calls)
- Resource Allocator
  - Operating systems allocate hardware resources to processes
    - memory
    - network
    - disk space
    - CPU time
    - I/O devices

Motivations

- Protection Boundaries
  - Operating systems protect processes from each other and itself from process
  - Note: Everyone trusts the kernel
- Resource Schedulers
  - Operating systems schedule access to resources
  - e.g., process scheduling, disk scheduling, etc.
- Complicated
  - See Project 3 :)

Monolithic Kernels



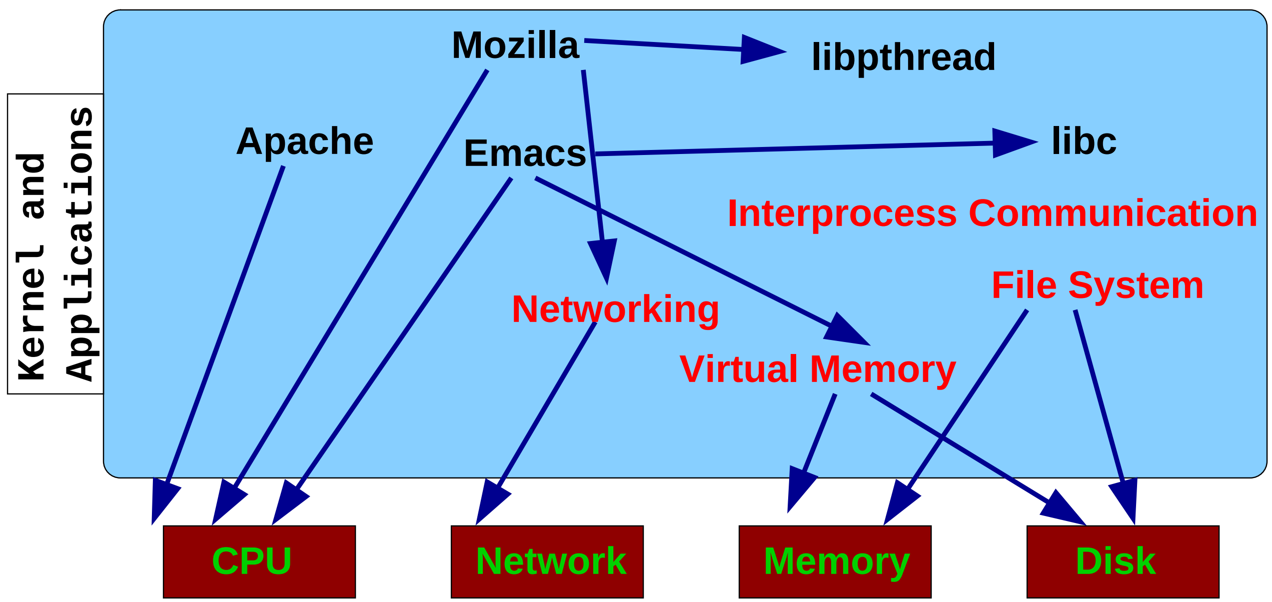
Monolithic Kernels

- You've seen this before
- The kernel is all in one place with no protection between components
  - See Project 3 :)
- Applications use a well-defined system call interface to interact with the kernel
- Examples: UNIX, Mac OS X, Windows NT/XP, Linux, BSD, i.e., common

Monolithic Kernels

- Advantages:
  - + Well understood
  - + Good performance
  - + High level of protection between applications
- Disadvantages:
  - – No protection between kernel components
  - – Not extensible
  - – Overall structure is complicated
    - ∗ Everything is intermixed
    - ∗ There aren't clear boundaries between modules

Open Systems



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

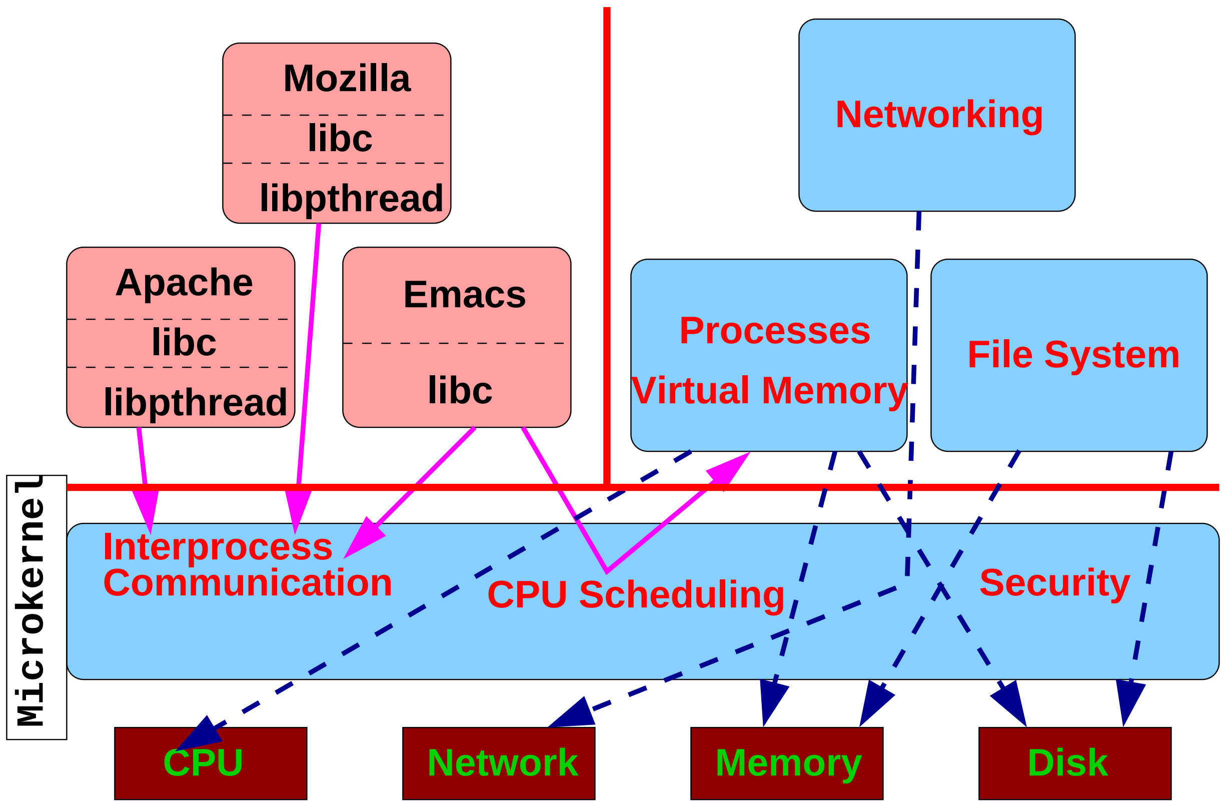
- Used to be *very* common

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

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

Microkernels



Microkernels

- Replace the monolithic kernel with a “small, clean, logical” set of abstractions
  - Tasks and Threads
  - Virtual Memory
  - Interprocess Communication
- Move the rest of the OS into *server* processes
- Examples: Mach, Chorus, QNX, GNU Hurd
- Mixed results: QNX commercially successful in the embedded space, microkernels are mostly nonexistent elsewhere

Microkernels

- Advantages:
  + Extensible: just add a new server to extend the OS
  + "Operating system" agnostic:
    - ⋆ Support of operating system *personalities*
    - ⋆ Have a server for each system (Mac, Windows, UNIX)
    - ⋆ All applications can run on the same kernel
    - ⋆ IBM Workplace OS
      - ∗ one kernel for OS/2, OS/400, and AIX
      - ∗ based on Mach 3.0
      - ∗ failure

Microkernels

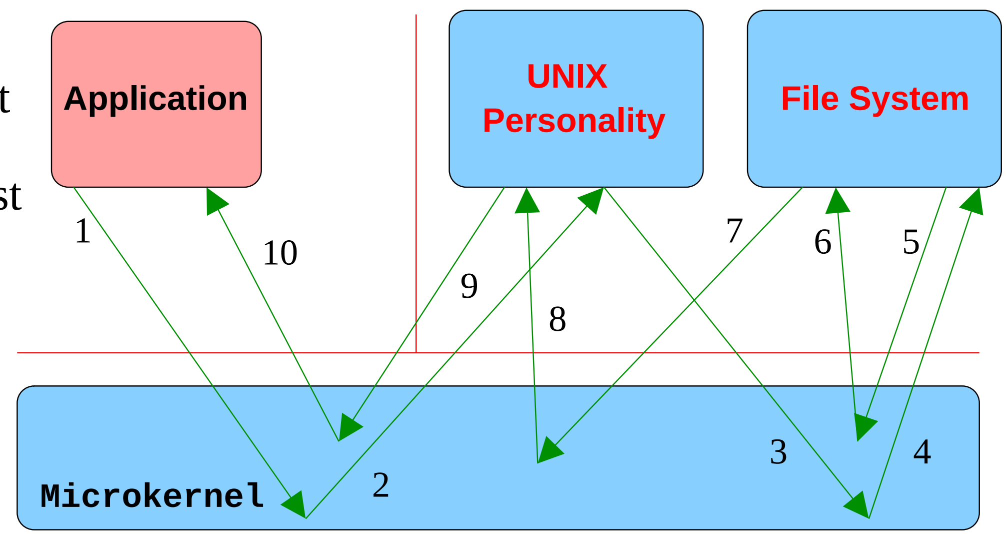
- Advantages:
  - + Mostly hardware agnostic
    - ⋆ Threads and IPC don't care about the details of the underlying hardware
  - + Strong protection, the operating system is protected even from itself
  - + Naturally extended to distributed systems

Microkernels

- Disadvantages:
  - Performance
    - System calls can require a large number of protection mode changes (see next slide)
    - Mach frequently criticized for its performance
    - Is this really an issue?
  - Expensive to re-implement everything using a new model

Microkernels

1. Application calls read(), traps to kernel
2. UNIX personality recieves read() request
3. UNIX personality requests the data
4. File System server recieves read() request
5. File System Server asks for disk blocks
6. Microkernel sends disk blocks
7. File System server sends results
8. UNIX Personality recieves data
9. UNIX Personality sends data
10. Application recieves data



Mach

- Started as a project at CMU (based on RIG project from Rochester)
- Plan
  1. Proof of concept
     – Take BSD 4.1, fix parts like VM, user visible kernel threads, IPC
  2. Microkernel and a “single-server” Unix
     – “Take Unix kernel, saw in half”
  3. Microkernel and multiple servers (FS, paging, network, etc.)
     – Servers glued together by OS *personality modules* which catch syscalls

Mach

- What actually happened:

  1. Proof of concept
     - Completed in 1989
     - Unix: SMP, kernel threads, 5 architectures
     - Commercial deployment: Encore Multimax, Convex Exemplar (SPP-UX), OSF/1
     - Avie Tevanian took this to NeXT: NeXTStep → OS X
  2. Microkernel plus single-server
     - Completed, deployed to 10's of machines (everybody graduated)
  3. Microkernel plus multi-server OS
     - Never really completed (everybody graduated)

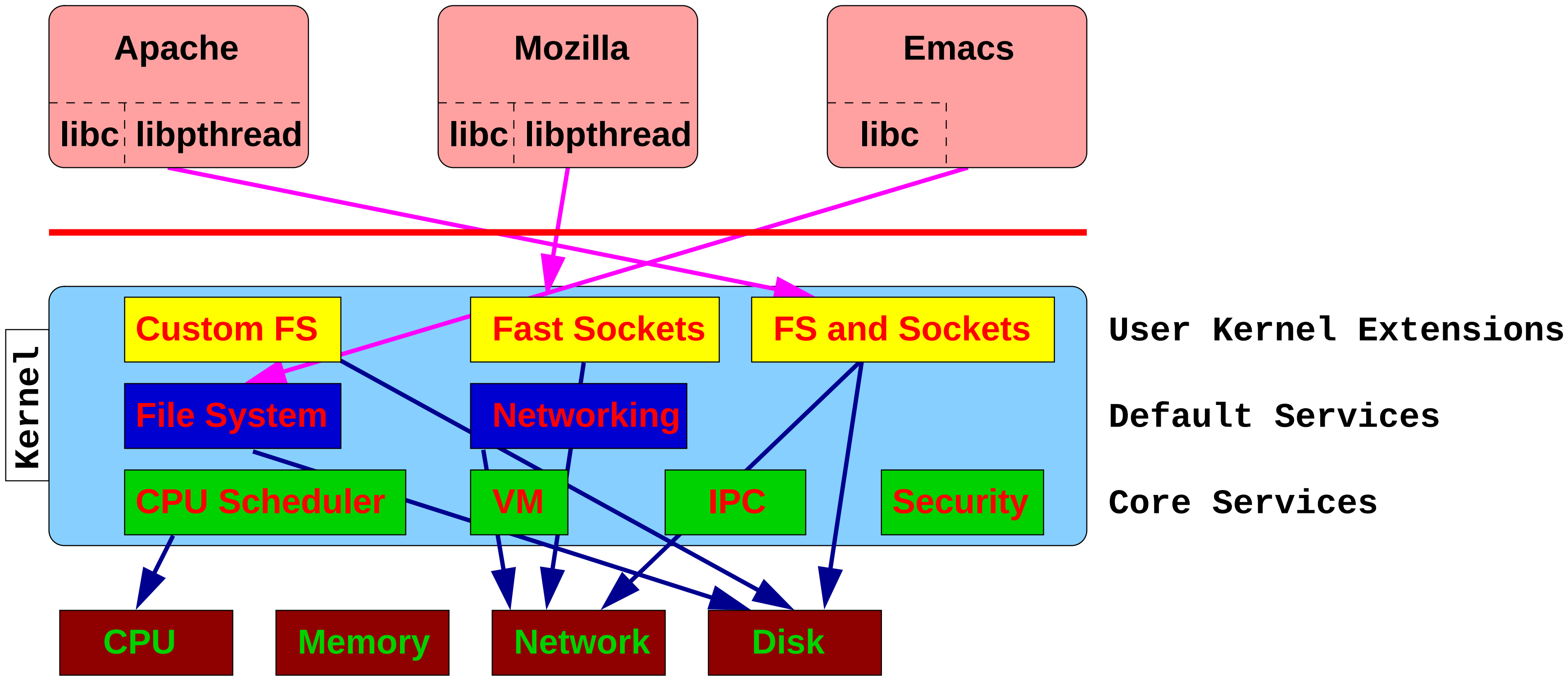
Microkernel Performance

- Mach was never aggressively tuned in the desktop/server context
  - Is it fair to compare Mach to monolithic kernels?
- QNX is at least strong enough to be competitive with other real-time operating systems, such as VxWorks
- The literature has between 5 and 50 percent performance overhead for microkernels
- *The Performance of μ-Kernel-Based Systems*
  - http://os.inf.tu-dresden.de/pubs/sosp97/

GNU Hurd

- Hurd stands for 'Hird of Unix-Replacing Daemons' and Hird stands for 'Hurd of Interfaces Representing Depth'
- GNU Hurd is the FSF's kernel
- Work began in 1990 on the kernel, has run on 10's of machines
- Hurd/Mach vaguely runs now, so they have sort of abandoned work on it in favor of Hurd/L4
- Ready for mass deployment Real Soon Now™

Kernel Extensions



Kernel Extensions

- Two related ideas: old way and new way
- Old way:
  - System administrator adds a new module to an existing kernel
  - This can be hot or may require a reboot: no compiling
  - VMS, Windows NT, Linux, BSD, Mac OS X
  - Safe? “of course” - if sysadmin chooses wisely

Kernel Extensions

- New way:
  - Allow *users* to download enhancements into the kernel
    - Compiler-checked source safety (UW: Spin: Modula-3)
    - Kernel-verified binary safety (CMU: Proof-carrying code)
  - Safe? Guaranteed

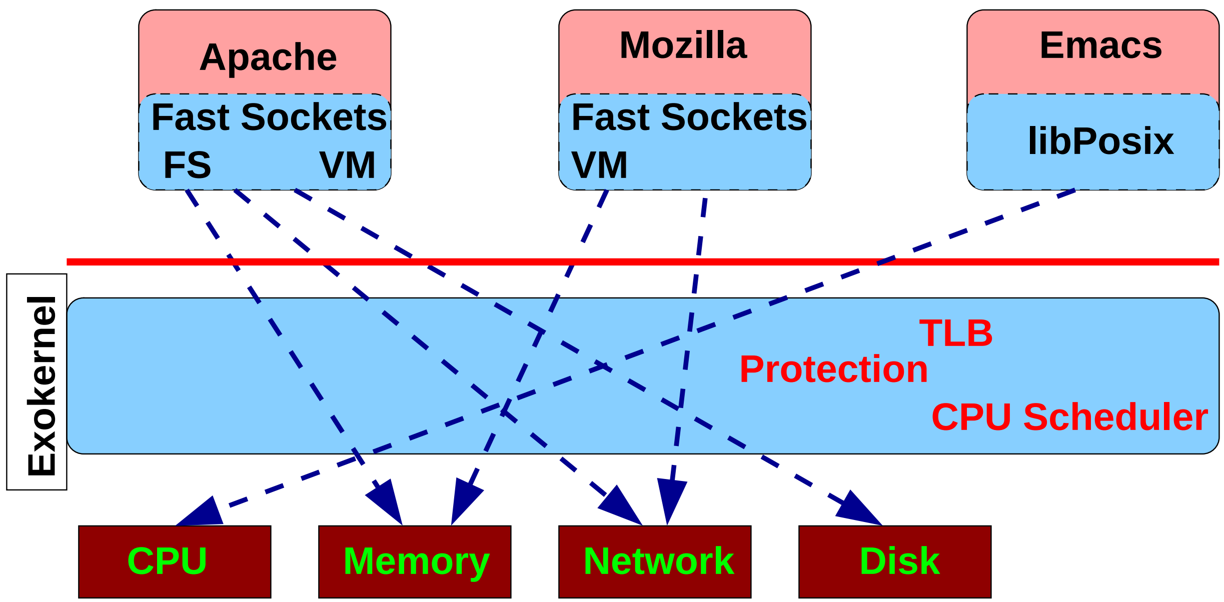
Kernel Extensions

- Advantages:
  - + Extensible, just add a new extension
  - + Safe (“Of course!” / “Guaranteed!”)
  - + Good performance because everything is in the kernel
- Disadvantages:
  - – Rely on compilers, PCC proof checker, head of project, etc., for safety.
  - – Constrained implementation language on systems like Spin
  - – The old way doesn’t give safety, but does give extensibility

Pause

- So far we've really just moved things around
- There is still a VM system, file system, IPC, etc.
- Why should I trust the kernel to give me a filesystem that is good for me?
  - Best performance for small, big, mutable, and static files
  - The right ACL model
- Let's try something different

Exokernels


Apache
Fast Sockets
FS
VM
Mozilla
Fast Sockets
VM
Emacs
libPosix
Exokernel
TLB
Protection
CPU Scheduler
CPU
Memory
Network
Disk

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
- Is this safe? Sure, the Exokernel's job is to provide *safe*, *multiplexed* access to the hardware
- This separates the security and protection from the management of resources

Exokernels: VM Example

- There is no fork()
- There is no exec()
- There is no automatic stack growth
- Exokernel keeps track of physical memory pages and assigns them to an application on request
- Application makes a call into the Exokernel and asks for a physical memory page
- Exokernel manages hardware level of virtual memory

Exokernels: simple fork()

- fork():
  - Acquire a new, blank address space
  - Allocate some physical frames
  - Map physical pages into blank address space
  - Copy bits 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 this service

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
- Basically, the fork() implementation can choose how to handle these details

Exokernels: VM

- To revoke a virtual to physical mapping, the Exokernel asks for a physical page victim
- If an application does not cooperate, the Exokernel can take a physical page by force, writing it out to disk
- The application is free to manage its virtual to physical mappings using any data structure it wants

Exokernels: Web Server Example

- In a typical web server the data must go from:

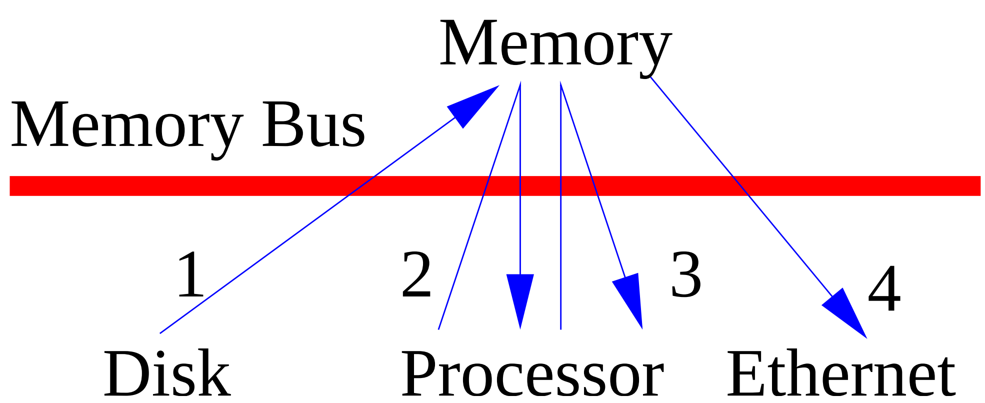
  1. the disk to kernel memory
  2. kernel memory to user memory
  3. user memory back to kernel memory
  4. kernel memory to the network device

- In an exokernel, the application can have the data go straight from disk to the network interface

Exokernels: Web Server Example

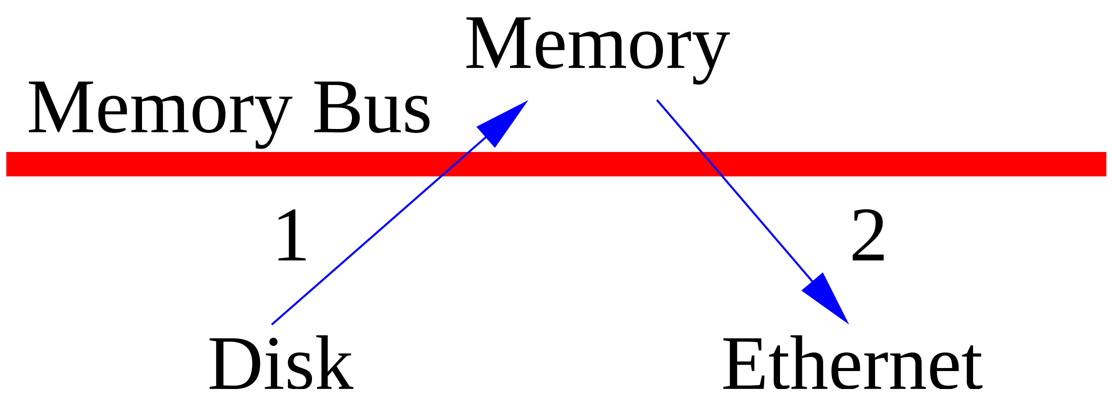
- Traditional kernel and web server:

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



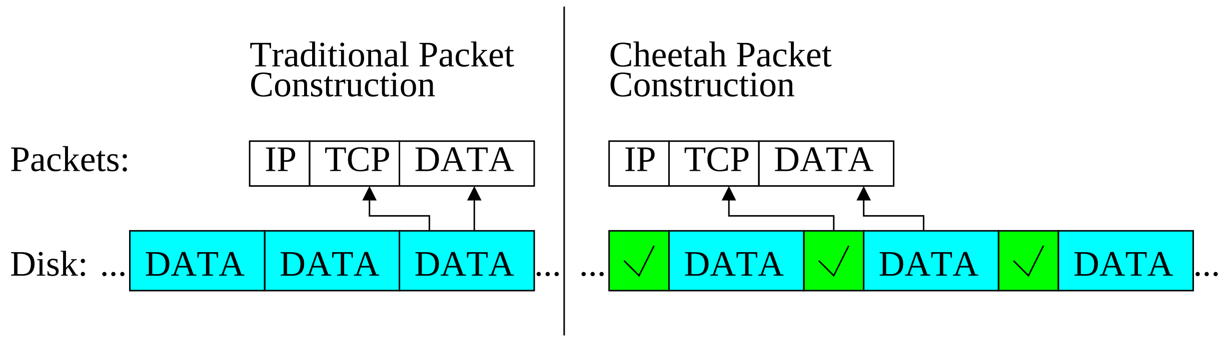
Exokernels: Web Server Example

- Exokernel and Cheetah:
  1. Copy from disk to memory
  2. Copy from memory to network device

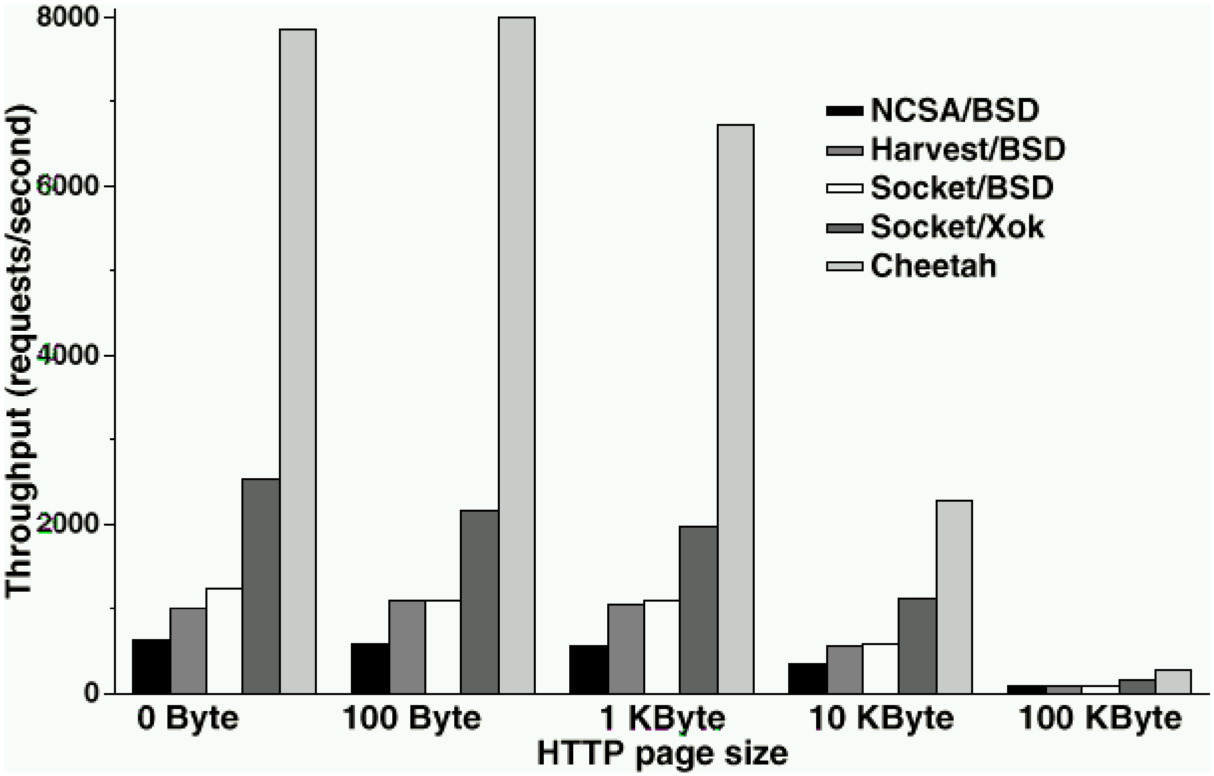


Exokernels: Web Server Example

- 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 checksums to be "patched"
  - This saves the system from recomputing a checksum, saves processing power



Exokernels: Cheetah Performance



Exokernels

- Xok development is mostly over
- Torch has been passed to L4
  - For fun and profit: `http://os.inf.tu-dresden.de/L4/`

Exokernels

- Advantages:
  - + Extensible: just add a new libOS
  - + Fast: Applications get direct access to hardware
  - + Safe: Exokernel allows safe sharing of resources

- Disadvantages:
  - Still complicated, just moving it up into user space libraries
  - Extensible *in theory*, in practice must change libPosix which is a lot like changing a monolithic kernel
  - Expensive to rewrite existing kernels
  - send_file(2) - Why change when you can steal?
  - Requires policy, despite assertions to the contrary

Final Thoughts

- Operating systems are complicated
- Structure *does* matter
- Many alternatives, but monolithic with a little bit of kernel extensions thrown in are the most common
- Why did none of the other structures win?
- Why should I re-implement my kernel when I can just add the functionality that gave you better performance numbers? (see send_file(2))

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