Why Exokernels Matter

The Two Questions of the Talk

- Do exokernels give control to applications?
  - Can interesting, shared resources be exported?
  - Can a real OS be built on an exokernel?

- Do exokernels matter?
  - Do normal applications benefit significantly?
  - Do aggressive applications improve by 10x?
  - Is global performance bad?
Outline

- What is an exokernel?
- Most interesting problem: disk multiplexing
- Xok/ExOS: a real exokernel system
- Application performance
- Summary
Exokernels in a Nutshell

- The problem with traditional OS structures:
  - Most interesting resource management decisions already made and cannot be altered

- The exokernel belief:
  - Allowing anyone to manage resources safely will hugely improve innovation/performance

- Why?
  - Anyone can innovate, using result has low risk
Exokernel Architecture

Key: Separate protection from management

Memory Pages

Disk

Network

gcc

libOS

FS, Net, VM

Specialized Web Server

FS, Net, libOS subset
How do you build a file system?

Main idea: untrusted deterministic functions (UDFs) let libFS metadata safely specify which disk blocks it owns.
How To Multiplex the Disk

- **Goal:** libFS as powerful as privileged FS
- **Hardest problem:** who can use a disk block?

- **Issue 1:** access control ≈ file system
  - Sol’n: reuse libFS’ own data structures
- **Issue 2:** must understand metadata
  - Fixed set of components would be infeasible
Novel Solution: UDFs

Result: libFS metadata tracks what it owns without kernel understanding how
C-FFS: A Fast LibFS

- Faster than in-kernel file systems (e.g. FFS)
- Uses exokernel control to:
  - Embed inodes in directories
  - Co-locate related files together on disk
  - Fetch large chunks of disk on every read
- To guarantee metadata integrity:
  - Use “protected methods” (specified along with UDFs) to guard modifications
What is an exokernel?
- Key idea: Separate management from protection
- Ideal: libOS as powerful as privileged OS

Hardest problem: disk multiplexing
- Reuse libFS metadata for access control using UDFs
- Built C-FFS (performance results coming up)

Next: A real exokernel system + app. performance
Xok/ExOS: A Real OS

◆ Xok:
  – Runs on x86
  – Multiplexes disk, memory, network, …

◆ Default libOS: ExOS
  – “Unix as a library”
  – Runs many unmodified Unix applications
    » csh, perl, gcc, telnet, ftp, …
  – Compiles itself

◆ Caveats: no VM paging, no SFI on methods
Experimental Methodology

- Xok vs. OpenBSD 2.1 and FreeBSD 2.2:
  - Xok uses OpenBSD-derived device drivers
  - Shares large code base (libc, most apps)
- Main experimental caveat:
  - Some ExOS data structures are not fully protected
  - Estimate cost of full protection by performing all necessary checks and adding 3 extra system calls per reference
Experimental Questions

- Do normal applications benefit?
- Is exokernel flexibility costly?
- Do aggressive applications get 10x?
- What happens to global performance?
Do normal applications need to manage resources to benefit?

No. Their libOS does the work.
Normal Applications Benefit

- Unaltered Unix apps + aggressive libFS
- Untrusted resource management = up to 4x faster
Does adding another layer of protection make everything slower?

No. Protection is off critical path: we conservatively duplicate checks, overhead lost in noise.
Is Exokernel Flexibility Costly?

Tentative answer: No

- gunzip
- pax
- cp
- diff
- gcc
- gzip
- rm
- pax

- Xok/CFFS
- BSD/CFFS
Nano, pico, exo, endo, whatever.
Why does OS structure matter?

One reason: Exokernel enables aggressive optimization without sacrificing protection.
The Cheetah Web Server

- Customization = 8 x perf. improvement
What about global performance?

(Tentative: it is good!)
Issues in Global Performance

- Wasteful applications?
  - No different than current systems

- Conflicting policies?
  - Exokernel architecture can enforce any global policy required for “performance protection”
  - Open challenge: recovering lost information

- Most optimizations result in less resources used
Optimization = More Resources

- Randomized mix of non-cooperative apps
Conclusions

- Exokernel Architecture:
  - Goal: safe application control of all resources
  - Ideal: libOS can do anything OS can
  - How: separate management from protection

- Results are promising:
  - Unaltered applications run same or 4x better
  - Custom applications up to 8x better
  - Global performance as good or better than Unix
Protecting high-level shared state

- Problem: enforcing high level invariants
  - General soln: layer protection on exokernel
  - How: “privileged”/unprivileged libOS code
    » protection code ~ 10% of code base

- Problem: inflexibility! Solutions:
  - Privileged sw only implements protection
  - Localizing state
  - Declarative guards
  - Note: most sharing is merely fault isolation
Exokernel Advantages

- Multiple libOSes co-exist
  - Tight coupling to applications and domain
- Fast, easy innovation:
  - Unprivileged = anyone can innovate
    » # of system hackers >> # trusted kernel hackers
  - Fault-isolated = cheap to use innovations
  - Possible to deploy innovations to other systems
  - Strong analogy to compilers
Challenges

- **Portability, preventing system chaos**
  - standard soln: interfaces, good programming

- **Sharing state with buggy/malicious peers**
  - General soln: layer protection on exokernel
  - How: “privileged”/unprivileged libOS code
    » protection code ~ 10% of code base

- **Reconciling greed and global performance**
  - greed = faster apps = more resources
What about Linux/FreeBSD?

- Exokernel/libOS advantages:
  - Fault-isolation
  - Library development easier
  - Unices: slow rate of delivered innovation

- Cons:
  - Linux & co. available NOW
  - Large scale deployment may expose problems
Global Performance is Good
But you don’t handle ‘x’

- What to protect is somewhat orthogonal
- Exokernel mostly comes in after you decide what to protect: get everything else out
- Note, however, typically not much that is protection (fault-isolation, etc)