Exokernels

(or, making the operating system just another application library)

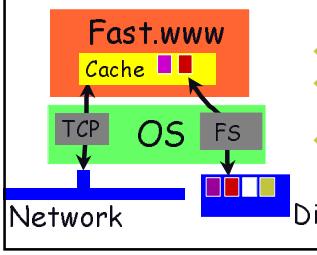
Dawson R. Engler

MIT Lab for Computer Science

F. Kaashoek, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, J. Jannotti

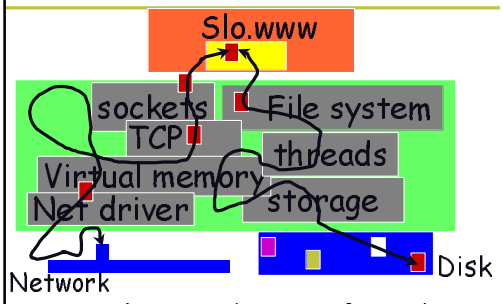
A motivating example

We want a fast web server:



- Only 1 copy of www page in main memory
- Zero touch
- Collapse protocol layers
- Related www pages clustered on disk

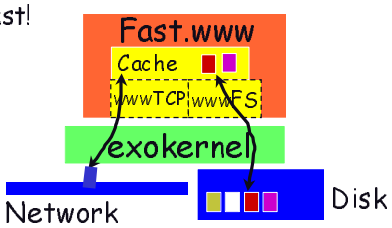
Traditional OS structure



- slow and can't fix it!

Exokernel: application control

- Application software can override OS
- Fast!



Exokernels in a nutshell

- Anyone can manage resources
- Exokernel safely exports resources
  - Separate protection from management
  - Virtual memory, file system, etc. are in application libraries (libOSes)
  - Hardware safely shared by different libOS implementations
- Ideal: LibOS as powerful as privileged OS

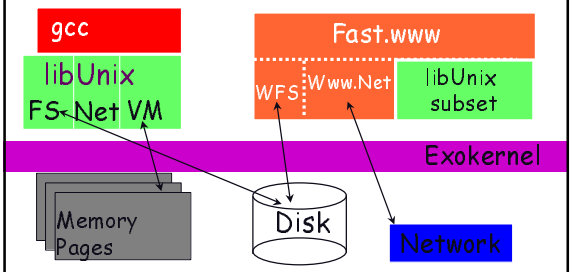
Focusing questions

- How to:
  - protect without overhead?
  - protect what you don't understand?
  - safely let applications track what they own?(!)
- Can you build a real system?
- Does an exokernel matter?
  - And, what about global performance?

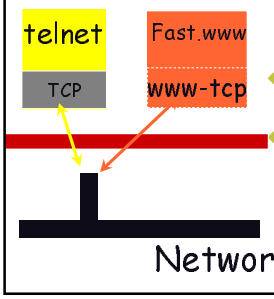
The rest of the talk

- Exokernel architecture
- DPF: application-level networking
- XN: application-level file systems
- Does an exokernel matter?
  - Application performance results (10x)
- What are the lessons?

Exokernel Architecture

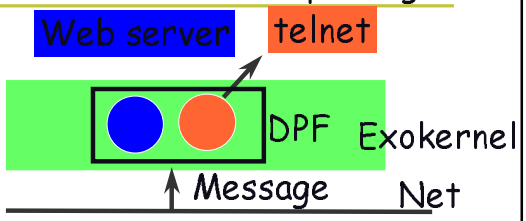


Application networking



- Goal: application libraries implement all networking
  - e.g., TCP & UDP in libNet
- libNets control: DMA, interrupt processing, ...
- Protection: who owns this network message?
  - Problem: exokernel doesn't understand msg semantics!
  - Solution: DPF!

DPF: Network Multiplexing



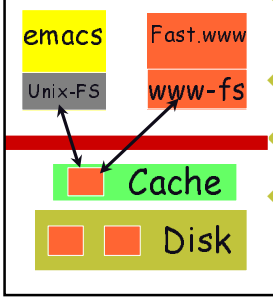
DPF packet filters:

- small functions downloaded by libNets into kernel
- determines who owns a message
- protection: dpf language makes overlap checking easy

DPF: A compiler hacker's dream

- Tiny, declarative language = easy opt.
  - filter merging ("inter-filter CSE")
  - dynamic compilation
  - trivial analysis that removes all static checks & aggregates dynamic ones
- Some weird ones too:
  - hash table compilation: emit binary search code for small tables, elide collision checks if none, create entry-driven hash functions, ...

Library file systems



- Goal: libFS as powerful as privileged FS
  - e.g., LFS, FFS, in libFS
- libFSes control: caching, layout, write orders, ...
- Cache coherence using buffer cache registry
- Key: guarantee different libFSes can safely share the *same* disk blocks

THE problem: access control

- Who can use a disk block?
- Access control = building a file system
  - insight: use libFS to do access control!
  - How? Reuse its metadata
- But, how to understand metadata?
  - Making libFSes build it from fixed set of known components not viable...
- What to do??

General solution

- Flexibility: application code ("owns") interprets metadata
  - owns(metadata) = {set of owned blocks}
- Correctness: use inductive testing to check incremental changes
- Technology: UDFs
  - deterministic: once owns(meta) satisfies our tests, it always will
  - all metadata modifications guarded by UDF checks

Using UDFs: ad hoc induction

- When meta allocated, check that:
  - owns(meta) = {}
- To give meta control of block `b'

```
old_set = owns(meta)
<let libFS scribble on meta>
if owns(meta) != old_set U {b} then
  error "bogus modification!"
```

- Result: kernel can trust metadata without understanding it or owns!

How things work "for real"

- To create a new FS:
  - Download FS types into kernel
    - type = owns + methods to modify meta
    - kernel checks determinism + safety
  - Allocate block for FS root and insert it into exokernel's root table
- To use:
  - load root of FS and walk down tree via owns(meta) to blocks you want

The Story So Far

- What is an exokernel?
  - Key idea: Separate management from protection
  - Ideal: libOS as powerful as privileged OS
- Protecting what we don't understand:
  - example: DPF
- Safely let applications track what they own:
  - example: UDFs
- Next: but do exokernels matter?

Can you build a real system?

Yes. We've built three.

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, …
- Caveats: no VM paging, no SFI on methods

Experimental questions

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

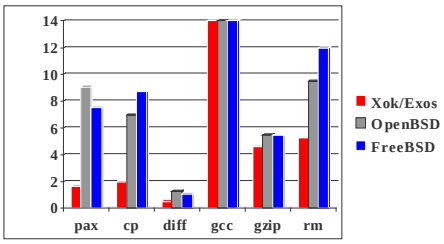
Experimental Methodology

- Xok vs. OpenBSD and FreeBSD:
  - Xok uses OpenBSD device drivers
  - Shares large code base (libc, most apps)
- Main experimental caveat:
  - Some structures aren't fully protected
  - Estimate cost of full protection by performing all checks and adding 3 extra system calls per reference

Do normal applications need to manage resources to benefit?

No. Their libOS does the work.

Normal Applications Benefit

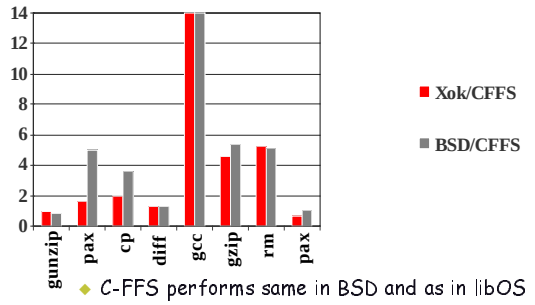


- Unaltered Unix apps + aggressive libFS (C-FFS)
- Untrusted resource mgm't = up to 4x faster

Is exokernel flexibility costly?

No. Protection is off critical path: we conservatively duplicate checks, overhead lost in noise.

Exo-flexibility is not costly

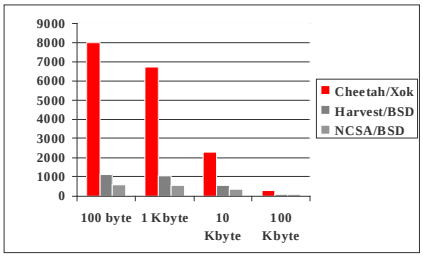


- C-FFS performs same in BSD and as in libOS

Nano, pico, exo, endo, whatever. Does OS structure matter!?

Yes! One reason: Exokernel enables aggressive optimization without sacrificing protection.

The Cheetah Web Server



- Aggression yields 8x speedup

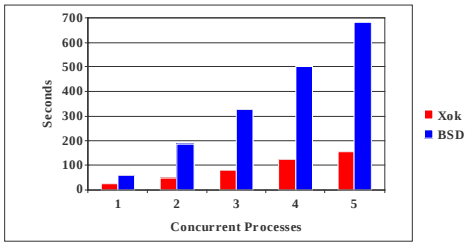
What about global performance?

(Tentative: it is good!)

Issues in Global Performance

- What about badly written libOSes?
  - No worse than bad apps now
- What about conflicting policies?
  - Exokernel can enforce any global policy required for "performance protection"
  - Open challenge: recovering lost information
- Insight: Most optimizations result in fewer resources used!

Optimization = More Resources



- Randomized mix of non-cooperative apps

Experiential debris

- Building OS much harder than libOS
  - edit, compile, reboot, printf tedious
- Fast applications are indifferent to fast microbenchmarks
  - xok poorly tuned, many extra syscalls
- Downloaded code's main benefit?
  - Power, not reduced kernel crossings
- Giving control is hard
  - All exo-interfaces 2nd/3rd generation

Conclusion

- Exokernel Architecture:
  - Goal: safe app. control of all resources
  - How: separate management & protection
- Results are promising!
  - Unaltered apps run up to 4x better
  - Global performance up to 4x better
  - Custom applications up to 8x better

Analogy: Compilers

- LibOS similar to compiler
  - potentially large, complex
  - not something to hack everyday
- But: since anyone can write a compiler
  - 1000's of languages and implementations
  - Rapid innovation and evolution
  - Can actually deploy results
- Imagine if compilers were in OS...

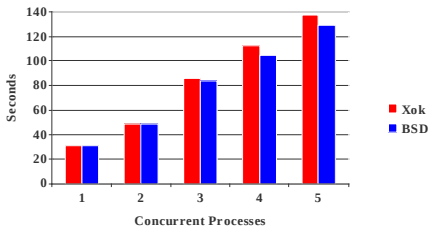
Exokernel: easy innovation

- LibOSes are:
  - Unprivileged = anyone can innovate
  - fault-isolated = cheap to use innovations
  - co-existant = innovation composition
  - easily deployed: use ftp or web.
- Key: Untrusted software evolves much faster than trusted software

Challenges

- Portability, preventing system chaos
  - Interfaces, good programming
- Sharing state with malicious peers
  - Layer protection on exokernel
- Greed and global performance
  - greed = faster apps = more resources

Same workload = same perf.

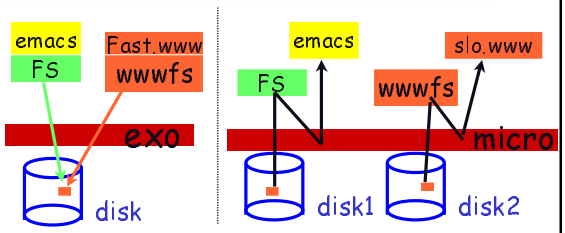


- Randomized mix of non-cooperative apps

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

Exokernel vs. microkernel



- micro: privileged, cannot safely share
- exo: unprivileged, can safely share

What about Linux/FreeBSD?

- Exokernel/libOS advantages:
  - Fault-isolation
  - Library development (much!) easier
  - Co-existence = can compose innovations
  - Unices: slow rate of delivered innovation
- Cons:
  - Linux & co. available NOW
  - Large scale exokernel deployment may expose problems