The Exokernel
Or, How I Learned to Stop Worrying and Hate Operating System Abstractions

Dawson Engler, M. Frans Kaashoek, et al
Motivation

- OS Level Abstractions are bad!
  - Require large, difficult to maintain kernels
    - (Not really much of a plus – code for abstractions has to be maintained anyway)
  - Applications pay unnecessary overhead when features are unwanted
    - Code might not use
  - Can’t utilize the hardware efficiently
    - e.g., Databases can’t make own paging scheme, no random access data storage
Their Solution

- Eliminate OS level abstractions
  - Necessary abstractions still exist, but placed in libraries (libOSes)
    - No HTTP servers at all!
  - Applications can use generic abstractions in shared libraries or create their own
  - Kernel provides interface to hardware, protection for shared resources
The Exokernel

- A minimal kernel allows applications secure access to hardware
  - Exokernel performs three tasks
    - Track ownership of resources
    - Ensure protection of resources
    - Revoking access to resources
- Their kernel is called Xok, their libOS is called ExOS
Design Principles

- Expose as much as possible
  - Allow hardware to be accessed as directly as possible
  - Expose names, allocation, revocation
    - User sees, e.g., physical addresses, freelists, knows when it is scheduled/descheduled
    - Kernel still controls allocation and revocation of resources
Three Design Techniques

- Secure Bindings
  - Apps bind to resources securely

- Visible Revocation
  - Apps participate in resource allocation/deallocation

- Abort Protocol
  - Kernel can forcibly break bindings
Secure Bindings

- Lightweight binding of an application to a resource
  - Allows the kernel to protect a resource without understanding it
- Performs authorization when resource is first requested (bind time)
- After that, binding can be used frequently without authorization (access time)
Two Obvious Techniques

- Hardware implementations
  - Ex: SGI hardware allows ownership of pixels in frame buffer

- Software Caches
  - Virtual -> real address mappings can be generated by libOS, stored in kernel TLB
Problems with Secure Bindings

- If ownership is granted through abstraction, how do we do that in kernel?
  - Example: files are owned, not disk blocks
- Answer: download *some* code into the kernel that explains the abstraction
Installing App Code in Kernel

- Application loads code into kernel
- Code is invoked on every resource access to determine ownership
  - Bonus: Can cut down on kernel crossings
    - They are cheap anyway in ExOS (18 instructions)
  - Bonus: Can be executed when application cannot be scheduled
    - If there is only a couple of ms of free processor time
Application-Specific Safe Handlers

- Untrusted message handlers for things like incoming packets
  - Sandboxed for protection
  - Code is examined
- Used for networking, file systems
Example: XN

- File System for Xok: exports most of everything
- How do you share disk space among apps using a library when the protection is not hardware–based, but abstraction based?
  - Must prevent user from grabbing other user’s blocks
  - Many onerous solutions
  - XN allows you to download code into kernel that verifies libraries track ownership correctly (in a minimal sense)
A template for each element of FS, block-wise is stored on disk

- UNIX FS has templates for inode blocks, data blocks, blocks for all sorts of metadata...

- Each template has function provided by libOS, *owns-udf* (untrusted deterministic fn)
  - UDF is not allowed access to other data, or loop, or perform other unsafe operations
Downloaded Functions in XN

- **owns-udf**
  - Input is metadata about disk blocks
  - Output is list of blocks belonging to metadata

- When an app wishes to update metadata \( m \) (say, allocate inode block) to include block \( b \), it calls XN, with \( m, b \), and the proposed modification

- XN verifies that \( \text{owns}(m+b) = \text{owns}(m)+b \)
Visible Revocation

- Apps participate in resource allocation/deallocation
- Example: scheduling
  - Time slices represented as vector
  - App can place self wherever it wants in vector, within protection limits
  - Can avoid things like one thread w/ spin lock causing a queue of waiting threads
Scheduling, cont.

- Processor is explicitly revoked at end of time-slice
- Application can call its own epilogue
  - End of time-slice function
  - Just save what needs to be saved, nothing else
  - Not a great example: happens too frequently to perform well
Abort Protocol

- What if an application has a secure binding and someone else needs it?
- Can explicitly revoke secure bindings when necessary
Performance of Xok/ExOS

- **Uses C−FFS**
  - FS that takes advantage of XN

- For apps that do not take advantage of hardware access, mostly no difference between OpenBSD, FreeBSD, Xok

- For disk-intensive, C−FFS+XN+Xok beats Free/OpenBSD
Other Performance Benefits

- Mostly for specialized apps (with good reason!)
- OpenBSD emulator ran marginally faster than real OpenBSD!
  - It all runs in user space – e.g., getpid() is twice as fast
- Customized HTTP server ran up to 8 times faster
  - Could implement server in kernel for same performance advantages (Linux)
  - But can you implement everything in kernel...?
Are Extensible Kernels Leading OS Research Astray?

- Argues that extensible systems are good for research because they allow you to experiment using a flexible framework
  - Can make a new FS really quickly
- But not so good for production OSes
  - Experiments can be incorporated into kernel without additional overhead of things like sandboxing, etc.
WWW Servers

- Extensions to support HTTP servers
- Looked at 8 possible extensions
  - 5 of them can be handled with kernel extensions
  - 3 can be handled with new application techniques
- Great! But...
Other Applications?

- Database vendors like to use their own FSes
  - Significant performance differences/tradeoffs
  - Are they all going to be incorporated into kernel?