# Exokernel: An OS Architecture for Application-Level Resource Management

D. Engler, F. M. Kaashoek and J. O'Toole Jr.

SOSP 1995

*Presented by Fabián*

NORTHWESTERN UNIVERSITY
1851

# What is a traditional OS?

- Resource manager – bottom-up/system-view
  - Everybody gets a fair-share of a resource
  - A control program to prevent errors & improper use

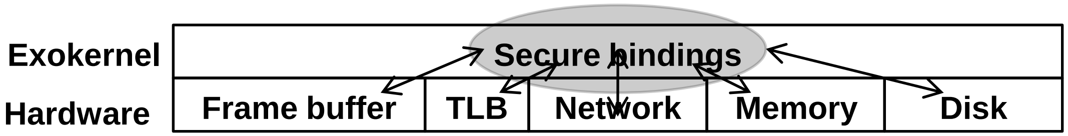
- Extended machine – top-down/user-view
  - Hides the messy details, presenting a virtual machine that's easier to program than the HW
    - Using several high-level abstractions; e.g. processes, files, address spaces, IPC
    - All applications must use these abstractions
    - Un-trusted applications cannot modify the abstractions' implementations

# Motivation for Exokernels

- Abstractions in traditional OS are overly general – all what anyone may need
  - Apps “pay” for what they don’t use, and
  - Apps cannot take advantage of domain-specific optimizations
- Fixed high-level abstractions
  - Hurt application performance – both abstractions and their implementations are compromises, i.e. somebody gets less than what they need/want
  - Hide information from application, making it hard for the app to implement their own resource mgmt abstractions
  - Limit the functionality of applications, as everybody must use them, very few changes (and new ideas) are incorporated

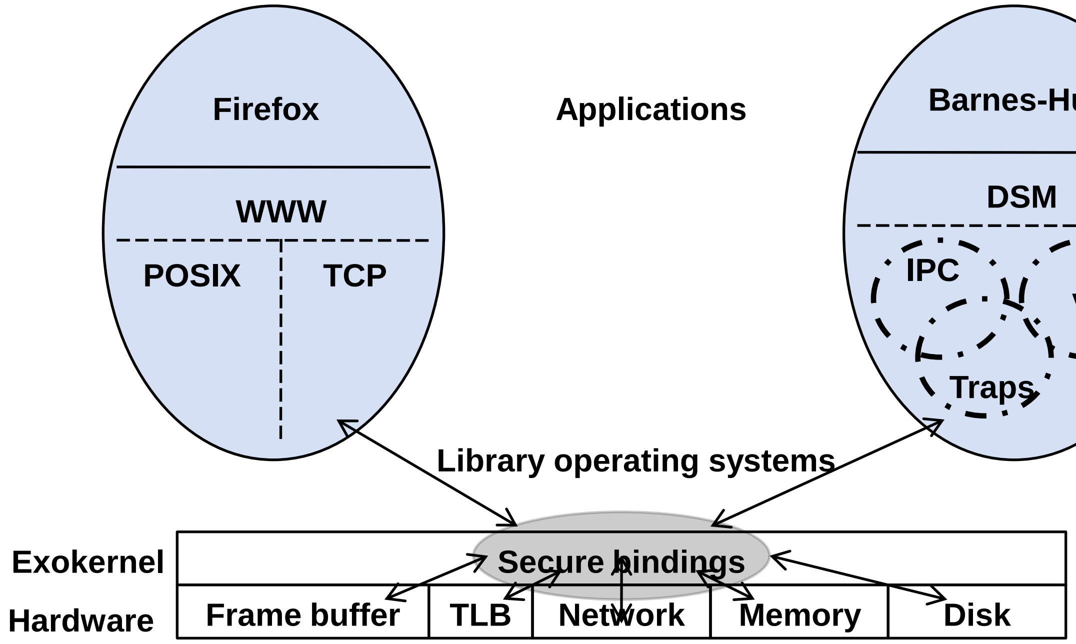
# High-level idea

- End-to-end argument
  - Applications know better than the OS what their resource management decisions should be, so
  - Implement traditional abstractions entirely at the app level
- Exokernel – a thin layer that multiplexes and control physical resources through low-level primitives
  - Allows extensions, modifications, replacement of abstractions
  - Simpler implementation that's more reliable, more efficient, easier to maintain



# High-level idea

- Library OSs implement the needed abstractions
  - Simpler and more specialized; no need to please everyone
  - Closer integrated w/ apps, since they are not trusted by kernel
  - More efficient given fewer kernel crossings
  - Portability by implementing whatever needed abstractions (e.g. LibOS that implement POSIX)



# Exokernel Design

- Main challenge – Give libOS freedom to manage resources while protecting them from each other
- To do this …
  - Track ownership of resources
  - Guard resource usage or binding points
  - Revoke access to resources
- Three techniques
  - *Secure bindings* of applications to machine resources
  - *Visible resource revocation*; applications participate in resource revocation protocol
  - *Abort protocol* to break secure bindings of uncooperative applications

# Design principles

- Exokernel defines the I/F that libOS use to claim/release/use resources
- What guides the I/F design? Basic principles
  - Expose hardware (securely) – central tenet of exokernel arch (Resources exported – CPU, physical mem, TLB, …)
  - Expose allocation – allow the app to request specific resource, no implicit allocation
  - Expose names – avoid indirection overhead and expose useful resource attributes; also export bookeeping data structures (e.g. freelists, cached TLB entries)
  - Expose revocation – so that well behaved libOS can do manage resources more effectively
- Some policy is part of exokernel
  - While exokernel cedes management of resources to libOSs,
  - It still controls allocation and revocation of resources

# Design – secure binding

- Multiplex resources *securely* among Library OSes
- Secure binding
  - Decouples authorization from use
  - Allows kernel to protect resource without understanding their semantics
- Better performance
  - Authorization to use resource only done at bind time
  - Simple, fast, protection check done when resource is accessed
- Example: TLB entry
  - Virtual to physical mapping performed in the library (above exokernel)
  - Binding loaded into the kernel; used multiple times

# Implementing secure bindings

- Hardware mechanisms
  - Capability for physical pages of a file
  - Frame buffer regions (SGI) – HW checks the ownership tag when I/O takes place
- Software caching
  - Exokernel large software TLB overlaying the hardware TLB
- Downloading code into kernel
  - E.g. Packet filter for demultiplexing network packets, application specific handlers (ASH)
  - Avoid expensive boundary crossings
  - Similar to the SPIN idea
  - Other use of downloaded code
    - Execute code on behalf of an app that is not currently scheduled
    - E.g. application handler for garbage collection could be installed in the kernel

# Design – visible revocation

- Traditional revocation is invisible, application is not involved (think page frames)
  - Lower latency, no need to talk to the application
  - Little information to guide it, since the application/libOS cannot guide it or knows there's a problem
- Visible revocation for most things
  - Including processor revocation, allowing the application to decide what part of its state to keep

# Design – abort protocol

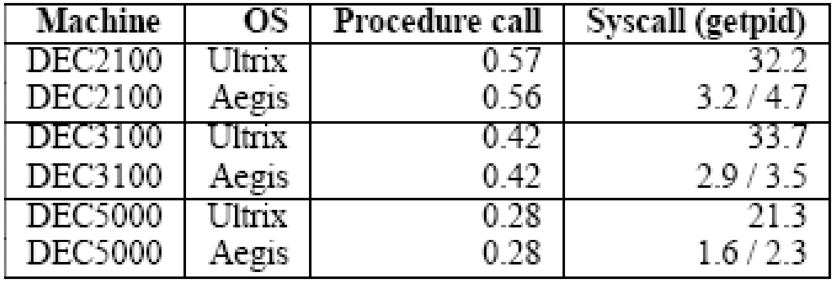
- For uncooperative libOSs, eventually use force
- Simply terminating the libOS and associated app makes it hard to work with, instead
- Break all existing secure bindings and inform the libOS
  - To inform repossession – repossession vector and repossession exeption
  - If resource has state, exokernel dumps this into another memory or disk resource (potentially preconfigured by libOS)
- Guarantee a minimum set of resources that will not be repossess (expect under emergency and with previous warning)

# Experiment: Aegis & ExOS

- Aegis: an exokernel on MIPS-based DECstation
  - Glaze – another exokernel for SPARC-based shared-memory multiprocessors
  - Xok – … for Intel x86 computers
- ExOS: the corresponding library OS
  - Virtual memory, IPC are managed at application level
  - Can be extended
- Performance compared with Ultrix 4.2, a monolithic UNIX
  - But ExOS do not offer the same level of functionality as Ultrix

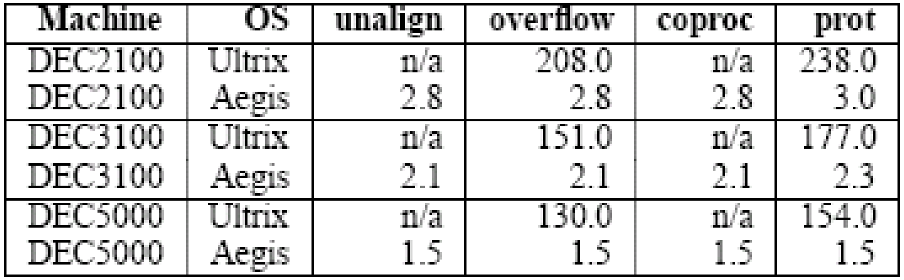
# Aegis performance

- Time (microsec) to perform a null procedure and system calls (for Aegis', first entry is for syscalls that do not use the stack) – an order of magnitude difference



| Machine | OS | Procedure call | Syscall (getpid) |
|---|---|---|---|
| DEC2100 | Ultrix | 0.57 | 32.2 |
| DEC2100 | Aegis | 0.56 | 3.2 / 4.7 |
| DEC3100 | Ultrix | 0.42 | 33.7 |
| DEC3100 | Aegis | 0.42 | 2.9 / 3.5 |
| DEC5000 | Ultrix | 0.28 | 21.3 |
| DEC5000 | Aegis | 0.28 | 1.6 / 2.3 |

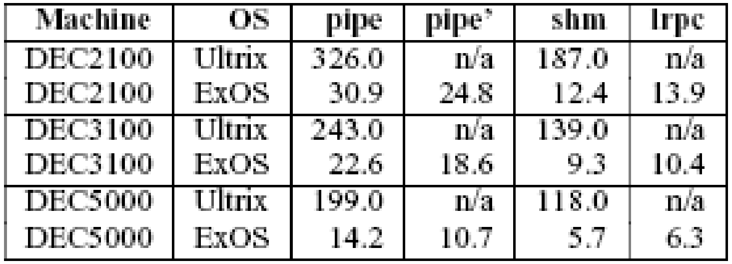
- Time (microsec) to dispatch an exception in Aegis and Ultrix – two order of magnitude faster



| Machine | OS | unalign | overflow | coproc | prot |
|---|---|---|---|---|---|
| DEC2100 | Ultrix | n/a | 208.0 | n/a | 238.0 |
| DEC2100 | Aegis | 2.8 | 2.8 | 2.8 | 3.0 |
| DEC3100 | Ultrix | n/a | 151.0 | n/a | 177.0 |
| DEC3100 | Aegis | 2.1 | 2.1 | 2.1 | 2.3 |
| DEC5000 | Ultrix | n/a | 130.0 | n/a | 154.0 |
| DEC5000 | Aegis | 1.5 | 1.5 | 1.5 | 1.5 |

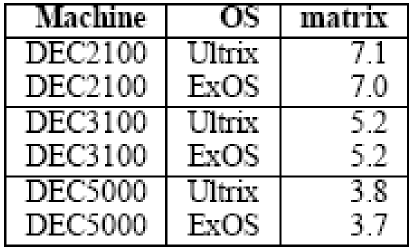
# ExOS – library OS

- ExOS manages fundamental OS abstractions at application level
- Evaluation shows efficiency for
  - IPC abstraction



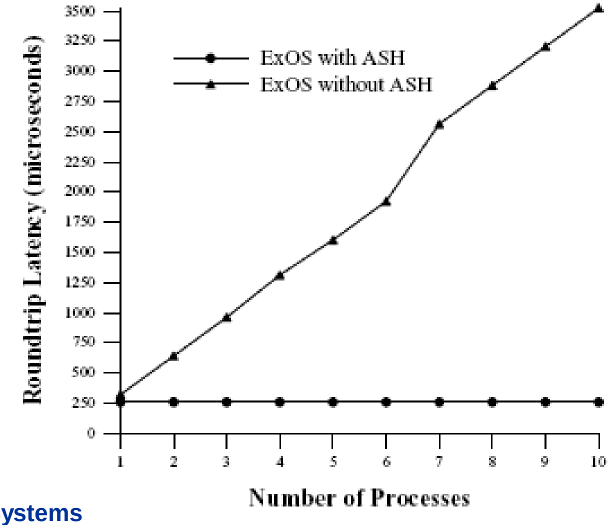
| Machine | OS | pipe | pipe' | shm | lrpc |
|---|---|---|---|---|---|
| DEC2100 | Ultrix | 326.0 | n/a | 187.0 | n/a |
| DEC2100 | ExOS | 30.9 | 24.8 | 12.4 | 13.9 |
| DEC3100 | Ultrix | 243.0 | n/a | 139.0 | n/a |
| DEC3100 | ExOS | 22.6 | 18.6 | 9.3 | 10.4 |
| DEC5000 | Ultrix | 199.0 | n/a | 118.0 | n/a |
| DEC5000 | ExOS | 14.2 | 10.7 | 5.7 | 6.3 |

  - VM (a 150xc150 integer matrix multiplication)



| Machine | OS | matrix |
|---|---|---|
| DEC2100 | Ultrix | 7.1 |
| DEC2100 | ExOS | 7.0 |
| DEC3100 | Ultrix | 5.2 |
| DEC3100 | ExOS | 5.2 |
| DEC5000 | Ultrix | 3.8 |
| DEC5000 | ExOS | 3.7 |

  - Remote communication using ASH (application specific safe handlers)



# Extensibility with ExOS

- Easy to redefine OS abstractions
- Examples
  - Extensible RPC – a trusted LRPC that's 40% faster than the untrusted one
  - Extensible page-table structures – linear or inverted, your choice (inverted for sparse address space)
  - Extensible schedulers – a proportional-share scheduling mechanism (stride scheduler)

# Summary

- Argue OS abstractions can be bad for applications
- Traditional OS abstractions implemented in Library OS, at application level
- Key idea – securely export hardware resources without abstraction
- Measurements indicate significant performance benefits – primitive kernel operations 10-100x faster than Ultrix
- Issues to think about
  - Potential for many different Library OSes
  - Portability?
  - Security?