Exokernel: An Operating System Architecture for Application-Level Resource Management

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A

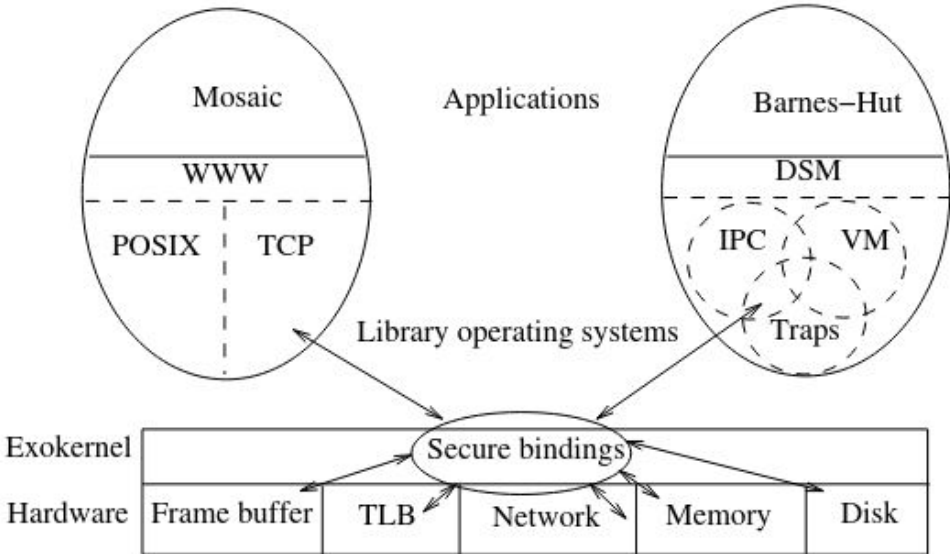
Motivation

Traditional OSes: Abstractions that cannot be specialized, extended, or replaced

- Fixed high-level abstractions:
  - hurt application performance - no single way to abstract physical resources or implement an abstraction that is best for all applications.
    - EX. garbage collectors have very predictable data access patterns - their performance suffers when a LRU page replacement strategy is imposed
  - hide information from applications.
    - EX. database implementors must struggle to emulate random-access record storage on top of file systems.

Exokernels

Applications know better what the goal of their resource management decisions should be and therefore, they should be given as much control as possible over those decisions : allow traditional abstractions to be implemented entirely at application level.



LibOSes

- Implementations of abstractions in LibOSes can be simpler and more specialized than in-kernel implementations, LibOSes need not multiplex a resource among competing applications with widely different demands.

- The number of kernel crossings in an exokernel system can be smaller, since most of the operating system runs in the address space of the application.

Exokernel Design Principles

- An exokernel should:
  - Expose hardware
    - provide secure low-level primitives that allow all hardware resources to be accessed as directly as possible.
  - Expose allocation
    - allow LibOSes to request specific physical resources.
  - Expose Names
    - export physical names. ex. freelists, disk arm positions, and cached TLB entries
  - Expose Revocation
    - utilize a visible resource revocation protocol so that well-behaved LibOSes can perform effective application-level resource management.

Policy

- An exokernel hands over resource policy decisions to LibOSes
- However, as in all systems, an exokernel must include policy to arbitrate between competing LibOSes: it must determine the absolute importance of different applications, their share of resources, etc.
- For instance, while an exokernel cedes management over to LibOSes, it controls the allocation and revocation
- By deciding which requests to grant and from which applications to revoke resources, an exokernel can enforce traditional partitioning strategies, such as quotas or reservation schemes

Secure bindings

- A secure binding is a protection mechanism that decouples authorization from the actual use of a resource.
- A secure binding performs authorization only at bind time, which allows management to be decoupled from protection.
- Simply put, a secure binding allows the kernel to protect resources without understanding them

Multiplexing Physical Memory

- When a LibOS allocates a physical page, the exokernel creates a secure binding by recording the owner and the permissions specified. The owner can change permissions and deallocate it.
- To ensure protection, the exokernel guards every access to a physical memory page by requiring that the capability be presented.
- Typically, the processor contains a TLB, and the exokernel must check memory capabilities when a LibOS attempts to enter a new virtual-to-physical mapping.
- Using capabilities to protect resources enables applications to grant access rights to other applications without kernel intervention.
- To break a secure binding, an exokernel must change the associated capabilities and mark the resource as free. In the case of physical memory, an exokernel would flush all TLB mappings and any queued DMA requests.

[De]multiplexing the Network

- Demultiplexing the network efficiently is challenging: protocol-specific knowledge is required to interpret the contents of incoming messages and identify the intended recipient.
- Message demultiplexing can be provided by packet filters. Packet filters can be viewed as an implementation of secure bindings in which application code—the filters—are downloaded into the kernel
- Fault isolation is ensured by:
  - careful language design
  - runtime checks
- A filter can “lie” and accept packets destined to another process. Only allowing a trusted server to install filters can address this problem
- Outgoing messages are simply copied from application space into a transmit buffer.

Visible Resource Allocation

- An exokernel uses visible revocation for most resources. Even the processor is explicitly revoked at the end of a time slice; a LibOS can react by saving only the required processor state
    - For example, a library operating system could avoid saving the floating point state or other registers that are not live
- However, since visible revocation requires interaction with a LibOS, invisible revocation can perform better when revocations occur very frequently.

Abort Protocol

- An exokernel must also be able to take resources from LibOSes that fail to respond satisfactorily to revocation requests
- Two stages:
  - please return a memory page
  - return a page within 50 microseconds
- If a LibOS fails to respond , the secure bindings need to be broken “by force.”
- The abort protocol defines the actions taken when a LibOS is recalcitrant

Abort Protocol

- If a LibOS fails to comply, an exokernel simply breaks all existing secure bindings to the resource and informs the LibOS
- When an exokernel takes a resource → register in repossession vector →“repossession” exception to LibOS → LibOS updates mappings.
  - For resources with state: write the state into another memory or disk resource.
  - The LibOS can load the repossession vector with resources that can be used for this purpose.
- An exokernel should not arbitrarily choose the resource to repossess.
  - A libos may use some physical memory to store vital bootstrap information
  - Guarantee each libos a small number of resources that will not be repossessed

Aegis : Processor Time Slices

- Aegis represents the CPU as a linear vector, and uses round robin scheduling
- Timer interrupts denote the beginning and end of time slices, and are delivered in a manner similar to exceptions
- A register is saved in the “interrupt save area,” → exception program counter is loaded, → jump to user-specified interrupt handling code
  - LibOS is responsible for context switching
- Fairness is achieved by bounding the time an application takes to save its context:
  - Applications pay for each excess time slice by forfeiting a subsequent time slice.
  - If the excess time counter exceeds a threshold, the process is killed

Aegis: Exceptions

- Saves 3 registers in “save area”
- Loads exception PC , last address that failed to have a valid translation, cause of exception
- Indirect jump to application-specified exception handler

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

Table 5: Time to dispatch an exception in Aegis and Ultrix; times are in microseconds.

Aegis: Address Translations

- An exokernel must provide support for bootstrapping the virtual naming system
- Aegis uses guaranteed mappings
- A miss on a guaranteed mapping will be handled automatically by Aegis
- Split address space to two segments
  - first segment contains normal code/data
  - addresses in second can be pinned using guaranteed mappings
- Use a STLB to cache mappings in the kernel, no need to go to application level if cache hit

Aegis: Protected Control Transfers

- Change PC to callee →Donate time slice to callee → Install callee processor context
- Asynchronous:
  - Donate the remainder of time slice
- Synchronous:
  - Done the remainder and every subsequent time slice
- Can use registers to pass data

Aegis: Protected Control Transfers

OS	Machine	MHz	Transfer cost
Aegis	DEC2100	12.5MHz	2.9
Aegis	DEC3100	16.67MHz	2.2
Aegis	DEC5000	25MHz	1.4
L3	486	50MHz	9.3 (normalized)

Table 6: Time to perform a (unidirectional) protected control transfer; times are in microseconds.

ExOS: IPC Abstractions

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

Table 8: Time for IPC using pipes, shared memory, and LRPC on ExOS and Ultrix; times are in microseconds. Pipe and shared memory are unidirectional, while LRPC is bidirectional.

ExOS: Application-level Virtual Memory

- appel1 : prot1 + trap + unprot
- appel2 : prot100 + trap + unprot

Machine	OS	dirty	prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Ultrix	n/a	51.6	175.0	175.0	240.0	383.0	335.0
DEC2100	ExOS	17.5	32.5	213.0	275.0	13.9	74.4	45.9
DEC3100	Ultrix	n/a	39.0	133.0	133.0	185.0	302.0	267.0
DEC3100	ExOS	13.1	24.4	156.0	206.0	10.1	55.0	34.0
DEC5000	Ultrix	n/a	32.0	102.0	102.0	161.0	262.0	232.0
DEC5000	ExOS	9.8	16.9	109.0	143.0	4.8	34.0	22.0

Table 10: Time to perform virtual memory operations on ExOS and Ultrix; times are in microseconds. The times for **appel1** and **appel2** are per page.

ExOS: ASHs

- ASHs are untrusted application-level message-handlers that are downloaded into the kernel
  - Controls where messages are copied, eliminates all intermediate copies
  - Can integrate data manipulations such as checksumming
  - Can initiate message sends
  - Can implement active messages

ExOS: ASHs

Machine	OS	Roundtrip latency
DEC5000/125	ExOS/ASH	259
DEC5000/125	ExOS	320
DEC5000/125	Ultrix	3400
DEC5000/200	Ultrix/FRPC	340

Table 11: Roundtrip latency of a 60-byte packet over Ethernet using ExOS with ASHs, ExOS without ASHs, Ultrix, and FRPC; times are in microseconds.

ExOS: ASHs

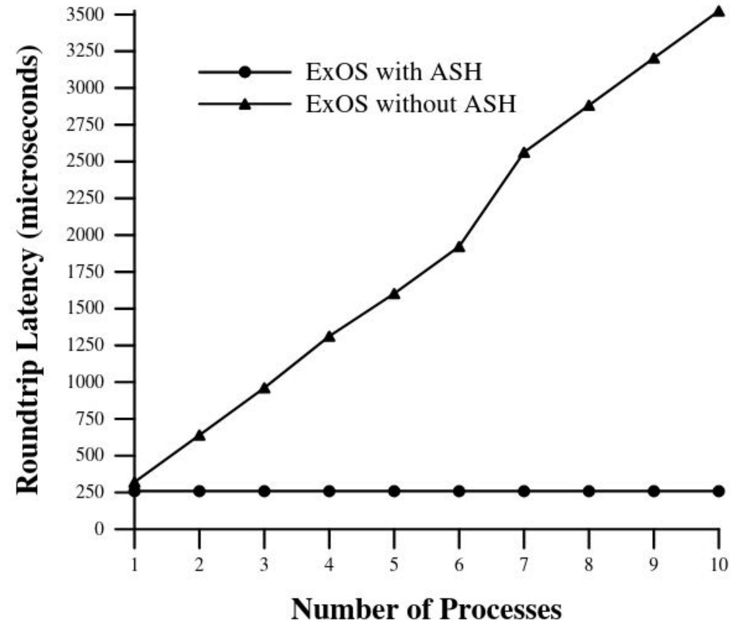


Figure 2: Average roundtrip latency with increasing number of active processes on receiver.

Thank You!

Any questions?