OS Structure: Exokernel End-to-End Arguments

Emmett Witchel

CS380L

Faux quiz

answer any two (5 min)

Exokernel

1. Why would we want to customize or extend a kernel?
2. What data structure does exokernel use for scheduling?
3. How should a batch task minimize its execution time on exokernel?
4. What is a “software TLB?”
5. What is an Application Specific Handler (ASH) and why is it needed?
6. What is the difference between a synchronous and asynchronous protected control transfer?
7. What is a “self-authenticating capability”? How does Exokernel use them? How well would the same techniques apply to a modern CPU micro-architecture?

End-to-End

- SSH encrypts *user* data in connections: Why is or isn’t this an example of the end-to-end argument?
- How would a proponent of the end-to-end argument likely fix the PC losering problem?

File Transfer: host A → host B

- A: read file from disk in blocks
- A: transmit in a series of packets
- Network: move packets to B
- B: receive packets, unpack
- B: write data on disk in blocks

HW fault → read incorrectly

Buggy buffering/copying

HW faults during buffering/copy

Either host can crash

Depending on protocol, packet loss/reorder/corrupt

What could possibly go wrong?

Conclusion: Only an end-to-end check would result in a file transfer program with failure probability proportional to file size

End-to-end: a religion?

Examples: illustration or no?

- TCP

- Airline reservations

TCP:
- Tries to provide reliable in-order packet delivery over IP with ACK
- Failure of higher-level protocol such as HTTP is still an app-level concern

Airline reservations:
- Lots of reliability mechanisms in use
- Still requires compensating transactions

The end-to-end argument is not an absolute rule, but rather a guideline that helps in application and protocol design analysis; one must use some care to identify the end points to which the argument should be applied."
Saltzer, Reed, & Clark, "End-to-end Arguments in System Design

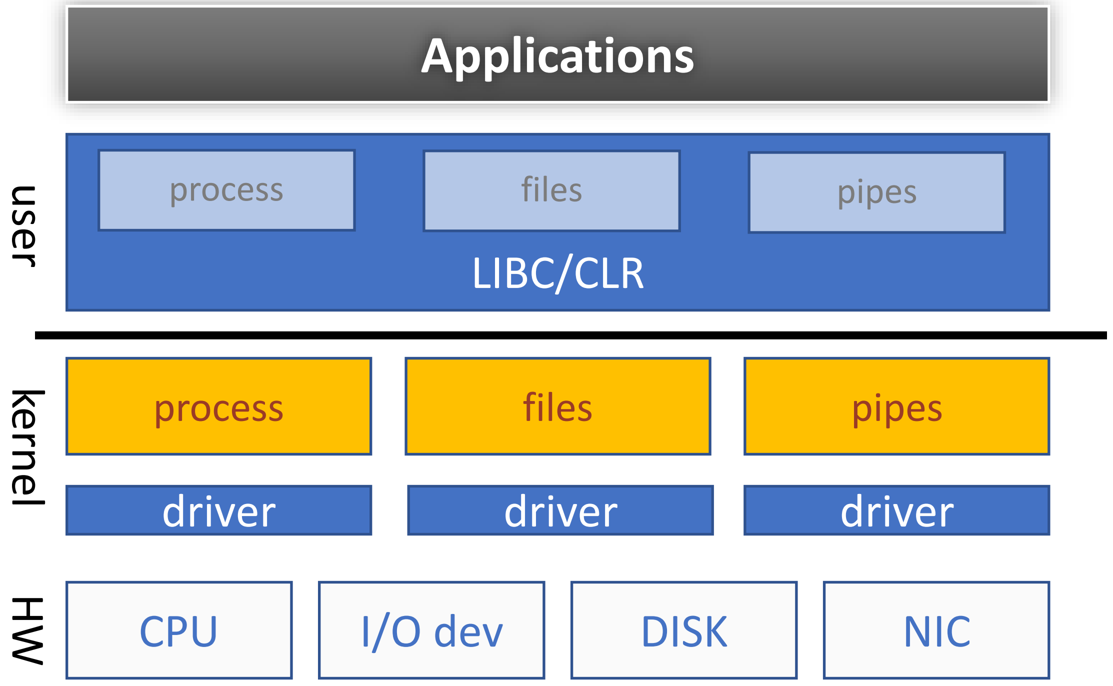
End-to-end wisdom

- Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer.
- Thus the amount of effort to put into reliability measures within the data communication system is seen to be an engineering tradeoff based on performance, rather than a requirement for correctness.
- What the application wants to know is whether or not the target host acted on the massage; all manner of disaster might have struck after message delivery but before completion of the action requested by the message.

End-to-end examples

- NetApp’s NFS appliance sometimes recommends UDP (lossy) and sometimes TCP (reliable)
- Google file system (originally) allowed duplicate data that was filtered by libraries
- Wireless networking puts more reliability into lower layers
- Application-level file checksumming was popular, now checksums being put into file systems

Background: extensibility



These high level abstractions are very nice and all, but…

- What if my app doesn't need them?
- What if they don't do what my app really needs?
- In a traditional OS, the OS feature set is fixed for apps

- Canonical example: ftp or web server serving static content

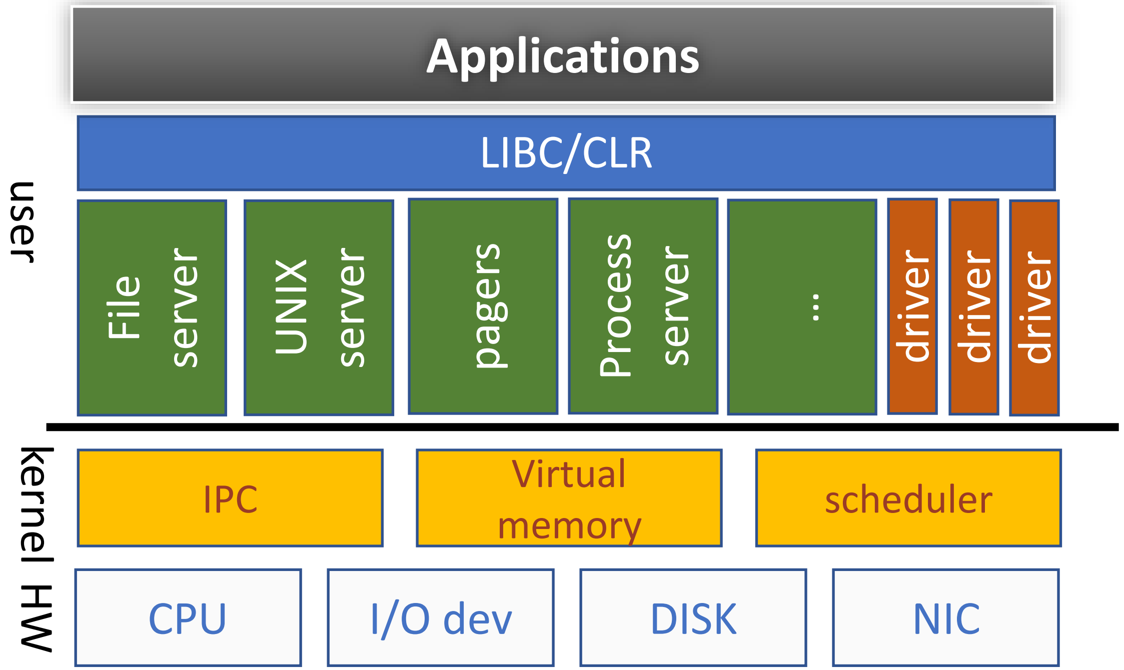
```
handle_get(URL url) {
    string local_path = get_local_path(url);
    FILE * fp = fopen(local_path);
    while(!feof(fp)) {
        read(buffer, … );
        write(buffer, …);
    }
}

Problem?
```

Data sourced from file (kernel managed object)
gets sent over the network (using kernel managed objects)
But is copied into user-space through FS API as a side effect
(sendfile() API is one solution)

Extensibility: *how can we customize an OS?*

- Microkernels (*Hydra, mach*)
- Virtual machines (*VM370, Disco, VMware, Xen*)
- OS per application (*Fluke, Unikernels*)
- Execute untrusted code in kernel (*Spin, Vino, Exokernel*)
- Exokernel/libOS (*Drawbridge, Bascule, Graphene, JITSU*)
  - (containers are a close relative)
  - (WSL 2 is *amazing*)

Microkernels



Core idea(s):

- *Minimal OS core to manage hardware*
- *Higher level abstractions in user space*
- *IPC fundamental cross-domain primitive*
- *…Many variants on this theme*

Pros/cons?

+ fault isolation

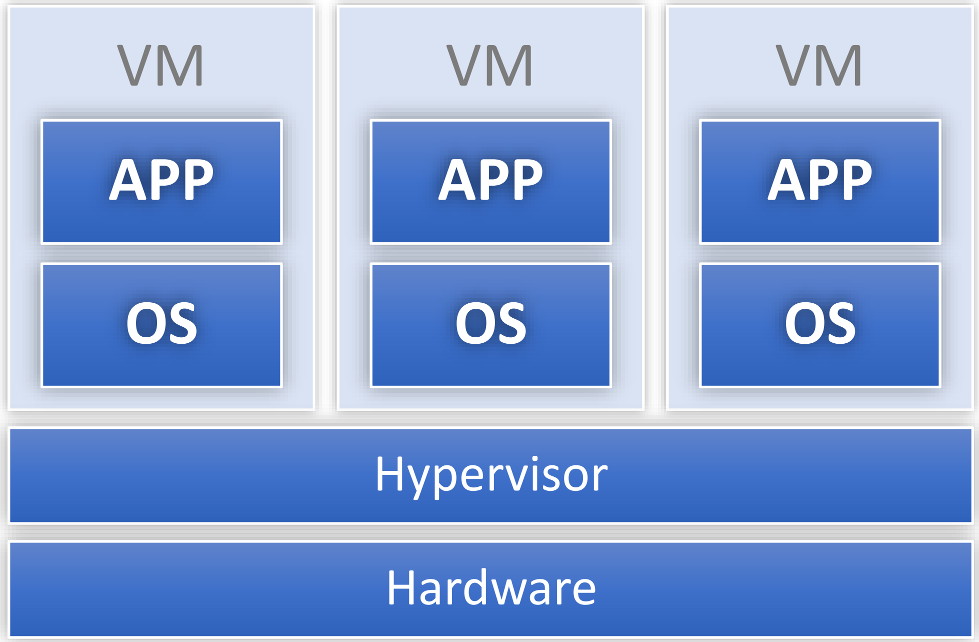
+ better extensibility

- slow (kernel crossings)

- limited extensibility

(see the contradiction?)

Extensibility: VMs



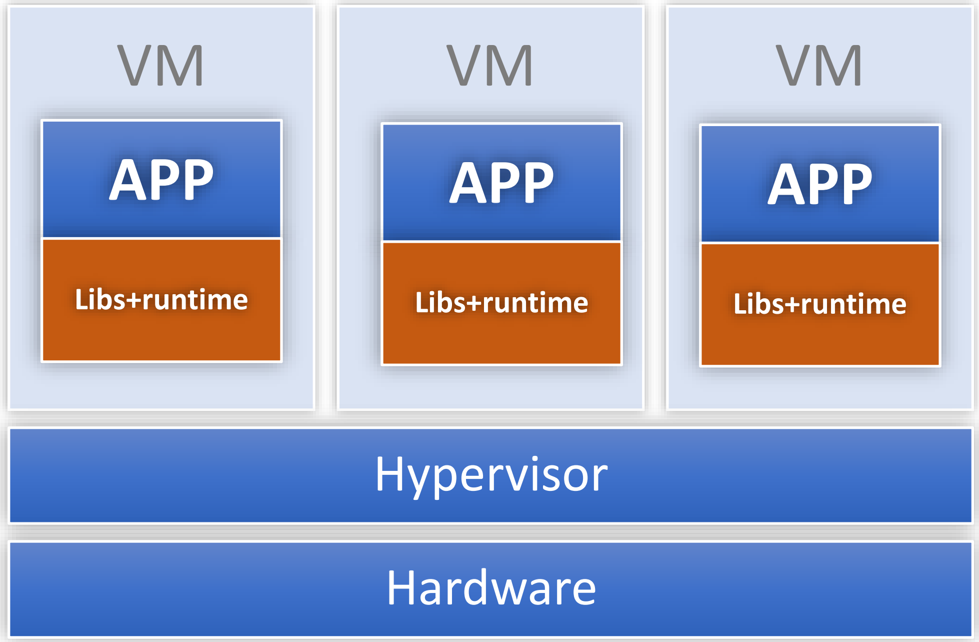
Core idea:
Different apps need different OSes, so…
figure out how to run more than one OS at a time

Pros/cons?

+ low-level interface ("ideal" according to Engler)

- "emulate" machine v. "export" resources (e.g. need to emulate "privileged" instructions)
- poor IPC (traditionally) – machines isolated
- hide resource management

Extensibility: OS per application



Core idea:

- *Hypervisor provides resource management and isolation*
- *Additional guest-OS layers redundant and unnecessary*
- *Collapse guest OS and application into same domain*
    - *Typically compiles OS and app into the same binary*

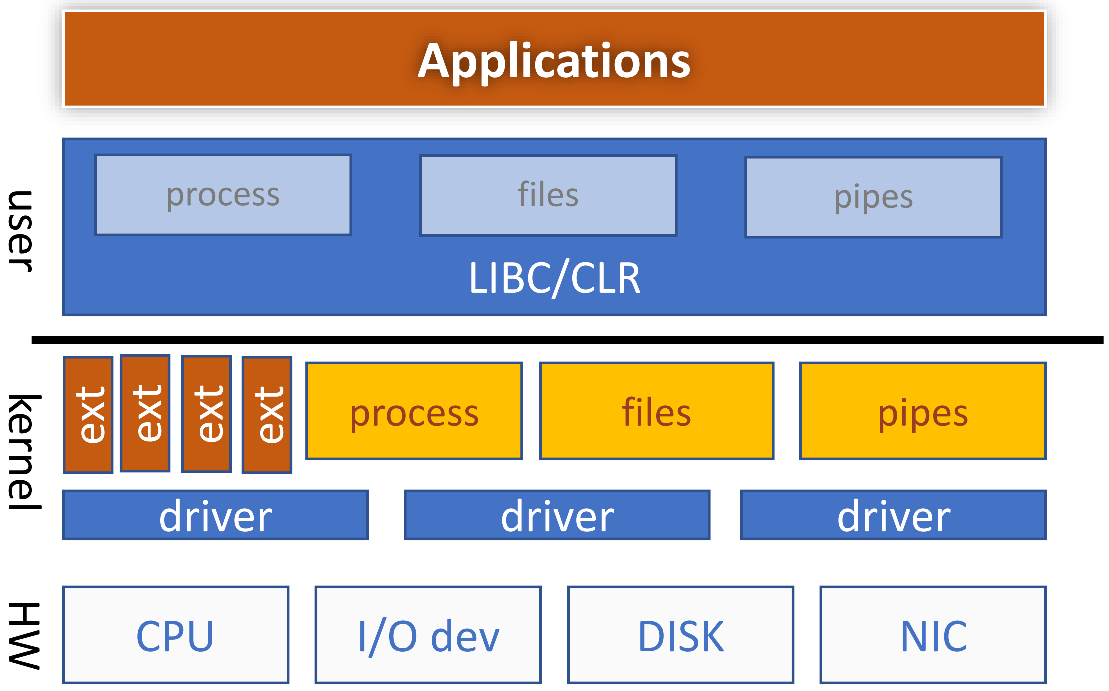
What are the pros/cons?

\+ Fast! (recent work in this area after long dormancy)

\- co-existing apps?

\- Disadvantage: kernels are complex, hard to modify and specialize

Download untrusted code into kernel



Core idea:

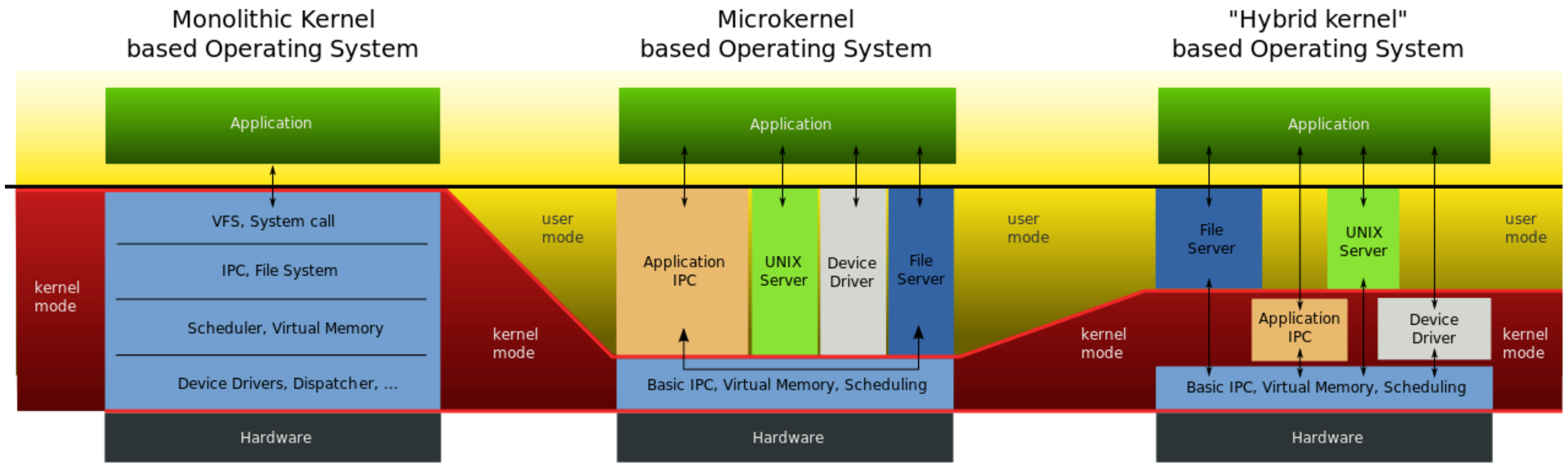
- *OS provides extensibility interfaces*
- *Apps provide extensions that execute in kernel mode*

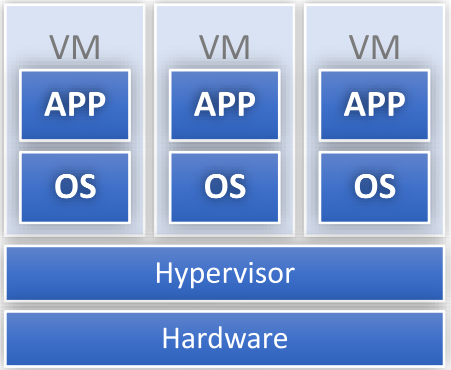
Pros/cons?

+ extensible

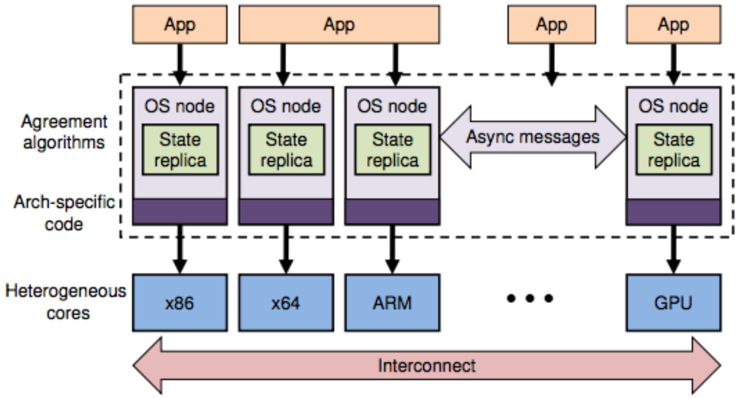
- still working with same OS structure
- Only extensible within limits of extensibility API
- New thicket of isolation and trust issues (eBPF is state of art)

Kernel Comparisons

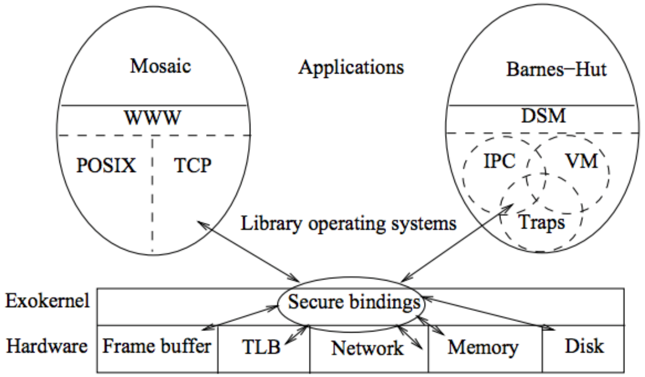




Multikernel



Exokernel



Exokernel: Key Ideas

A great exercise: identify one instance of each in exokernel and articulate why it's there and how it can be made to work.

Monolithic OS Bad:

- Centralized resource management
- All applications must use the same abstractions
- High-level abstractions
  - Overly general
  - Provide all features possible
  - Implementation cannot be modified
  - Limited functionality
- Information is hidden

Hypotheses:

- Exokernels can be very efficient
- Low-level, secure multiplexing of HW implementable efficiently
- Traditional OS abstractions can be implemented efficiently at application level
- Applications can create special-purpose implementations of these abstractions

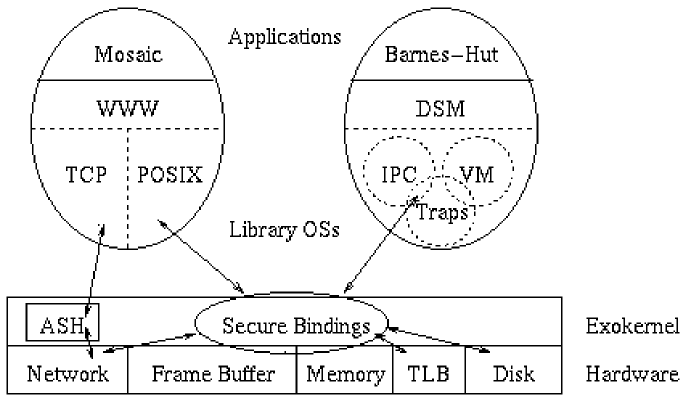
LibOS Good:

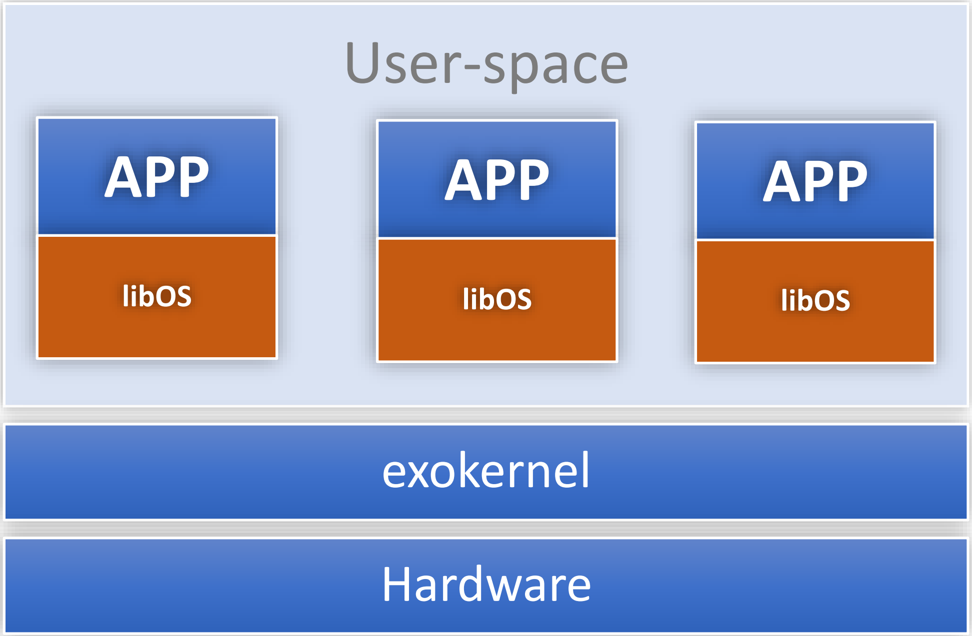
- Avoid resource management
- Allow request of specific resources
- Visible resource revocation
- Secure bindings
- Downloading code
- Abort protocol
- Extendable

Exokernel/libOS

Top-level structure

- 1) small monolithic kernel
  - low-level, fixed interface.
  - Ideally HW interface
  - few and simple abstractions
  - extension types
    - resource state data – page table entries
    - specialized resource mgmt modules
- 2) libraries of untrusted resource mgmt. routines
  - VM replacement
  - file system
  - IPC
  - …
- Note: libraries are part of OS
  - historically: OS was set of libraries for math, etc
- Key difference – trust
  - App can write over library, jump to bad addr, etc. kernel can not trust library





What does exokernel share with other approaches?

Exokernel Principles

- Separate protection and management
  - export resources at lowest level possible with protection
  - e.g. disk blocks, TLB entries, etc
  - resource mgmt only at level needed for protection – allocation, revocation, sharing, tracking of ownership
  - “abstraction (mechanism) is policy”
  - *The implementation of abstractions in library operating systems can be simpler and more specialized than in-kernel implementations, because library operating systems need not multiplex a resource among competing applications with widely different demands.*
- expose allocation – applications allocate resources explicitly
- expose names – use physical names (physical memory (cache coloring), disk arm position?)
- expose revocation – let apps choose which instances of a resource to give up
- expose information – let application map in (read only) internal kernel data structures (e.g. swTLB, CPU schedule, …)
- ***Exterminate all operating system abstractions (end-to-end)***

Mechanism: secure bindings

Bind at large granularity; access at small granularity

- Applicable in many systems, not just exokernel
  - E.g. malloc vs sbrk & mmap
- Allow kernel to protect resources without understanding them

Core idea: access check at bind time, not access time

Enables decoupling access check from abstraction being checked

Examples:

- Check at TLB entry load time for a page, not at address translation time
- Downloading code: type safe language, sandbox interpreter, validate at install time
- Others?

Mechanism: visible revocation

Continuum of resource multiplexing:

Transparent Revocation	Notify-on-revocation	Cooperative Revocation
Traditional OS • OS decides how many resources to give to apps • OS chooses what to revoke and takes it • Needed for performant frequent revocation (e.g., ASIDs)	Exokernel – abort protocol; repossession vector Scheduler activations • OS decides how many resources to give to apps • OS chooses what to revoke, takes it, and tells application (or libOS) • Reposes dirty disk block? Store it where? (3.4)	Exokernel – callbacks • OS decides how many resources to give to apps. • OS asks application or libOS to give up a resource; libOS/app decides which instance to give up

call application handler when taking away page, CPU, etc
→ application can react

- update data structures (e.g. reduce # threads when CPU goes away; *scheduler activations*
- decide what page to give up

ASIDs (processor addressing-context identifiers) are identified as a resource best revoked transparently, because of frequent revocation.

Using capabilities to protect resources enables applications to grant access rights to other applications without kernel intervention. Applications can also use “well-known” capabilities to share resources easily

More exokernel key mechanisms

abort protocol
    when voluntary revocation fails – kernel *tells* application what it took away
    reason – library can maintain valid state specification

capabilities – encryption-based tokens to prove right to access
    idea is to make kernel access-rights decision
    a) simple
    b) generic across resources
    c) hierarchical – child has a subset

wakeup predicates (from later paper)
    wakeup process when arbitrary condition becomes true (checked when scheduler looking for something to run)

buffer cache registry – bind disk blocks to memory pages
    → applications can share cached pages

Downloading code into kernel

- Multiplexing the network – packet filter
- idea: load code to examine packet and decide if it is for me.
- Implement by downloading code into kernel
  - written in simple, safe language – no loops, check all mem references, etc.
- Problem – what if I lie and say “yes it is for me” when it isn’t?
- Solution – “assume they don’t lie”
- claim – could use a trusted server to load these things or could check

ASHes

Load handlers for application-specific messages into kernel

→ *can reply to packet w/o context switch*

Advantages of ASH

- direct message vectoring – ASH knows where message should land in user memory → avoid copies
- dynamic integrated layer processing – e.g. do checksum as data is copied into NI
- message initiation – fast replies

ASHes

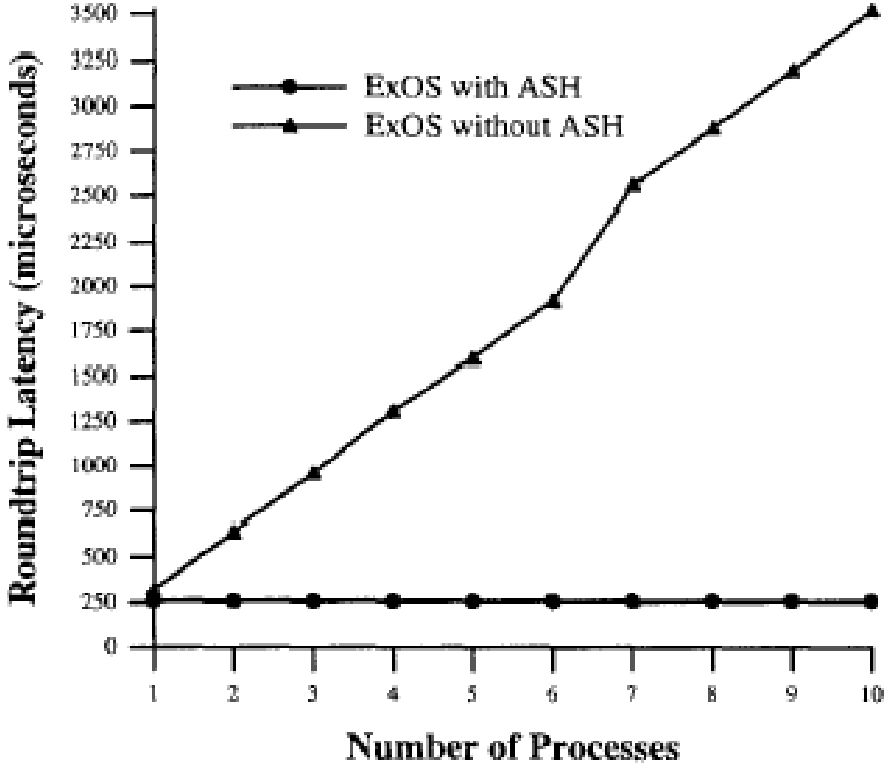


Figure 2: Average roundtrip latency with increasing number of active processes on receiver.

What is going on here?
Does this show that ASHes are just super awesome?

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

Evaluation

1) Run benchmarks several times, to warm up cache/TLB
2) Take best run for Ultrix. Exokernel is median of 3 runs
3) Instruction cache conflicts 3x problem for exokernel

- Lots of micro-benchmarks. They never show the full performance picture.
- prototype system offering one-tenth the functionality at ten times the performance?
  - a. Ping-ponging a counter
  - b. lrpc uses a single function (e.g., it does not use the RPC number to index into a table), it does not check permissions, it is single-threaded.
- ***What do you think?***

Aegis

- Scheduling
- Processor events
  - Exceptions
- Protected Control Transfers

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 to perform null procedure and system call (µs)

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

Exception dispatch time (µs)

ExOS: Interprocess Communication (IPC)

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

IPC time

ExOS: Virtual Memory

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

Virtual memory operations (μs)

Exokernel concluding observations

- This idea is important, but imperfect
  - Thin kernels, fat libraries
- More than one SOSP paper about this system
- Lessons (from second paper)
  - Provide space for application data in kernel data structures
  - Fast applications do not require good microbenchmark performance
    - "The main benefit of an exokernel is not that it makes primitive operations efficient, but that it gives applications control over expensive operations such as I/O"
  - Inexpensive critical sections are useful for LibOS's
  - User-level page tables are complex
  - Downloading interrupt handlers are of questionable utility
  - Downloaded code is powerful
    - "Advantage is *not* execution speed but rather trust and consequently power"
- Writable shared state was always a problem
  - E.g., a group writable file system directory