Exokernel – Engler, Kaashoek etc.…
"Mechanism is policy"

Outline:

Overview – 20 min
Specific abstractions – 30 min
Critique – 20 min

1. High-level goals

Goal – "Improved performance for standard applications; order of magnitude for aggressive applications"

example applications: web server, set top box, …
trend: hw cheap → if your app cares about performance, dedicate a machine to it (vs. "fine grained sharing")

Approach –

extensibility – application knows best – it should make decisions whenever possible

minimalist – kernel's job is to protect resources, not to manage them
separate protection from resource mgmt

2. Background

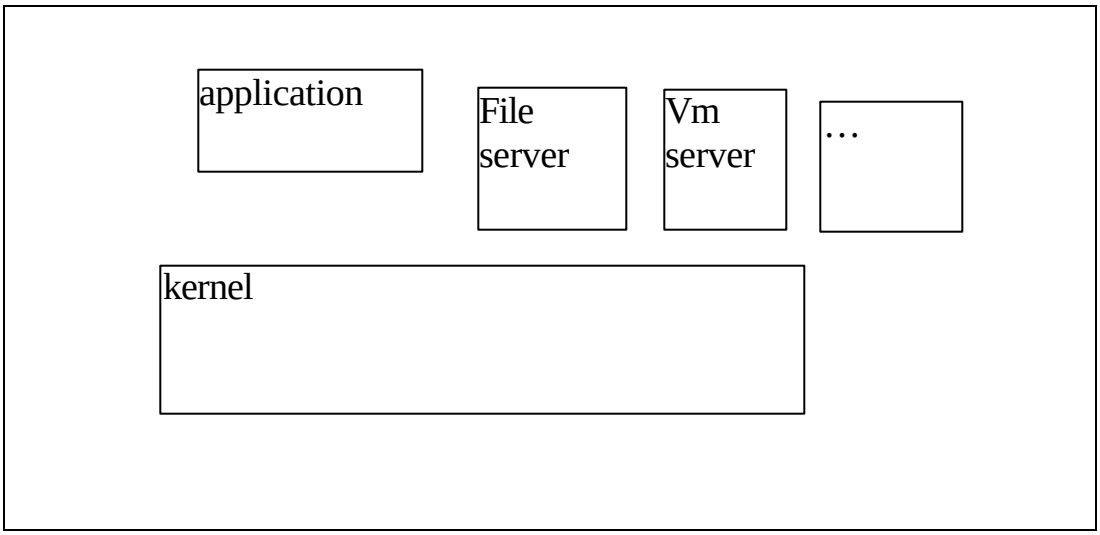
5 approaches to extensibility

1) OS per application (OSKit Fluke?)

DA: co-existance
DA: kernels are fragile and hard to modify

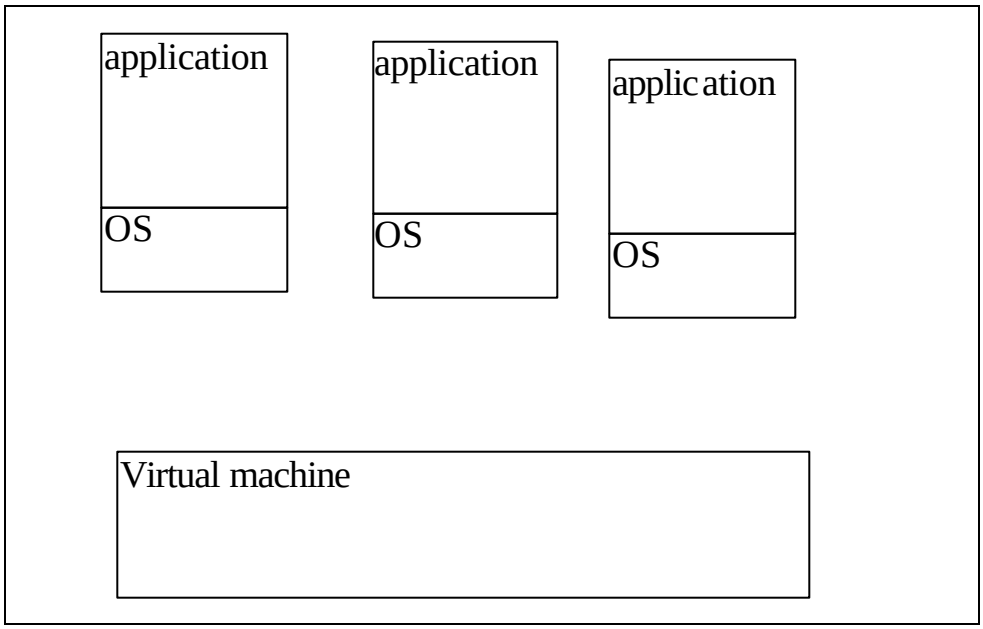
2) microkernels (hydra, mach, …)

- advantage: fault isolation
- slow (kernel crossings)



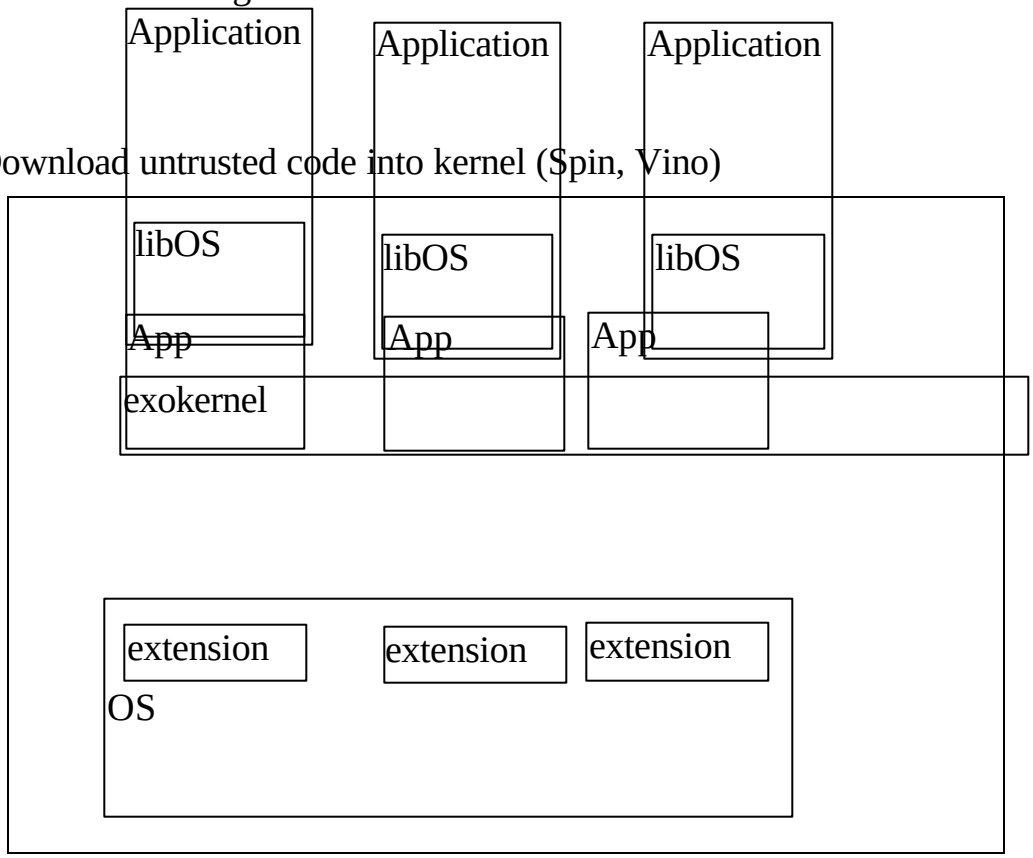
- limited extensibility (may make it easier for OS developer to extend, but not user)

3) virtual machines (VM370, Disco)



+ low-level interface (“ideal” according to Engler)
DA: “emulate” machine v. “export” resources
e.g. need to emulate “privileged” instructions

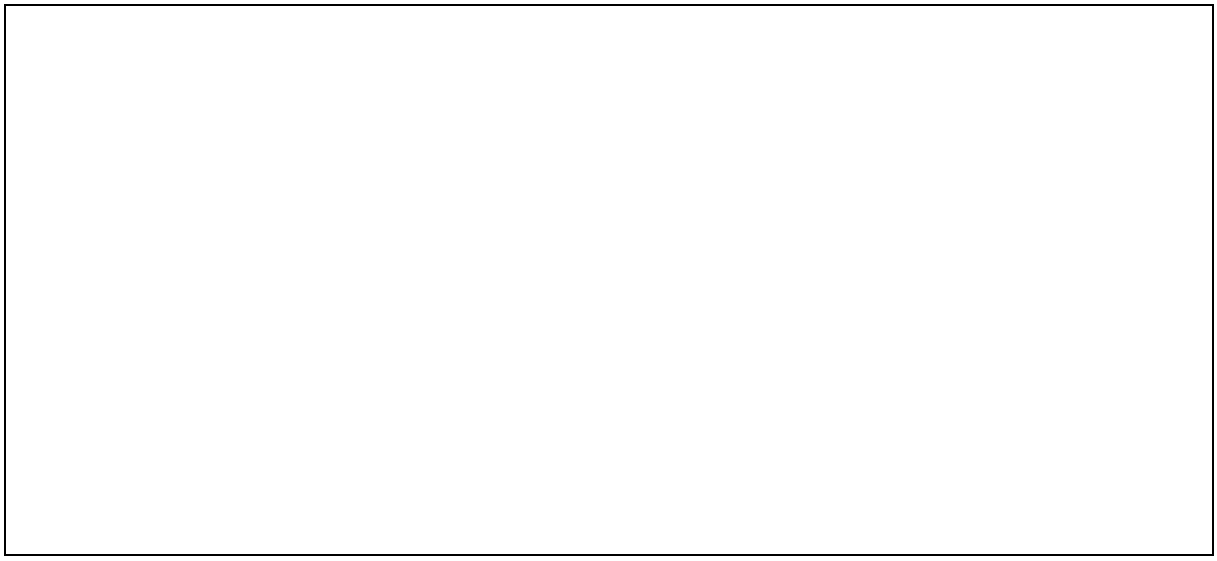
DA: poor IPC (traditionally) – machines isolated
DA: hide resource mgmt



4) Download untrusted code into kernel (Spin, Vino)

+ extension
DA: still working with basic OS
“complimentary to exokernel”

5) Exokernel/libOS



Top-level structure

1) an extendible microkernel
    low-level, fixed interface
    few and simple abstractions
    extension types
        resource state data – tlb table entries
        specialized resource mgmt modules
2) libraries of untrusted resource mgmt routines (may be different for different apps)
    VM replacement
    file system
    IPC
    …

Note: libraries *are* part of OS

- historically: OS was set of libraries for math, etc
- today – if it ain't got stdio, it ain't Unix!

Key difference – trust
Application can write over library, jump to bad addr in library, etc
→ kernel can not trust library

Exokernel borrows liberally from other approaches:

- Like Fluke: make it easy for each app to have custom OS
- Like virtual machine: exokernel exports virtual machine (difference: transparency – traditional VM wants to run unmodified OS's; exokernel VM wants to support custom OS's)
    - "Export, rather than emulate, resource" – LibOS is *aware* of multiplexing
- Like Vino, Spin: one mechanism for extensibility is to download untrusted code into kernel

2.1 Philosophy

- Traditional OS = protection + abstraction
- Exokernel:
    - Protection = kernel – minimal mechanism
      + library – resource sharing policy

- Abstraction = library

"To provide applications control over machine resources, an exokernel defines a low-level interface. The exokernel architecture is founded on and motivated by a single, simple, and old observation: the lower the level of the primitive, the more efficiently it can be implemented, and the more latitude it grants to implementers of higher-level abstractions.

- Minimalist approach
  - Key challenge – understand **core** of the abstractions for different resources
  - Next several weeks – papers trying to find core abstractions for concurrency, scheduling, networking, file systems, …
  - As we read these other papers, consider them from point of view of Exokernel (not necessarily the best point of view, but an interesting one…)

3. 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
  - ♦ "mechanism is policy"
- expose allocation – applications allocate resources explicitly
- expose names – use physical names whenever possible
- 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, …)

4. Key mechanisms

Initially, only talked about first 3 or 4, but as system matures, have to add more stuff in.

1) secure bindings

bind at large granularity; access at small granularity

· Applicable in many systems – not just exokernel

Do access check at bind time, not access time
e.g. when loading TLB entry for a page, not when accessing page

mechanisms/examples

a) hardware – TLB
b) SW – SW tlb cache
c) downloaded code – type safe language, interpreters, etc
d) traditional file system: open file/read and write file

Challenge: secure bindings v. Saltzer "complete mediation"

2) visible revocation

Continuum of resource multiplexing:

Transparent Revocation	Notify-on-revocation	Cooperative Revocation
Traditional OS n OS decides how many resources to give to apps n OS chooses what to revoke and takes it	Exokernel – abort protocol; repossession vector Scheduler activations n OS decides how many resources to give to apps n OS chooses what to revoke, takes it, and tells application (or libOS)	Exokernel – callbacks n OS decides how many resources to give to apps. n 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

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

4) capabilities – encryption-based tokens to prove right to access
   idea is to make kernel access-rights decision
   a) simple
   b) generic across resources

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

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

7) etc. (block state to order writes, UDF, …)

5. PART 2: Specific abstractions

1) **exception handler**
2) **page protection/sharing**
3) processor scheduling
4) fork/exec
5) VM replacement
6) **network protocol**
7) **file system**

6. Exceptions

1. Save 3 scratch registers to agreed-upon "save area"
   (use physical addresses in user space WHY?)
   phys → avoid TLB faults
   user space → can resume after exception w/o going back to kernel

2. load exception PC, badVA, exception type into those registers
3. jump to user handler

→ 18 instructions/1.5us!!!

v. 150 us in Ultrix → 2 orders of magnitude

QUESTION: Why?

- more demultiplexing *note: libOS may have to do some of this work*
  - *E.g., ExOS – 4 contexts: exception context, interrupt context, protected entry context, addressing context*
- save/restore registers note: libOS may have to do some of this work…*won't exokernel have to save/restore at least some of its registers in handler?*
- exokernel kernel runs in physical memory → no kernel TLB misses (simplifies exception handler even if no misses while handling this particular exception)

7. Memory Page protection/sharing

QUESTION: what are goals of abstraction?

1) protection -- user program can issue any VA it wants; can't touch anyone else
2) relocation – user can move data structures around in virtual address space
3) sharing – different address spaces can share same phys page
   crucial for exokernel
   - otherwise libOS's are huge waste of mem
   - "export information" from kernel by allowing users to map read-only
4) fast – use TLB, etc.

key idea – kernel decides what VA→PA translations application is allowed to install in TLB

approach

- kernel gets exception and calls application

exceptionHandler(exceptionPC, badVA, PAGE_FAULT)

- application does VA→ PA lookup
  page table, inverted page table, random…

  if no valid mapping, library signals application "Seg Fault"
  otherwise…

- application does system call to install TLB entry into kernel

  Alloc(TLB_ENTRY, VA, PA, state bits, capability)

  QUESTION: any security issues here?

- kernel checks that your capability has the rights to access page PA with permissions indicated by state_bits

  They don't tell you how this lookup takes place. Any ideas?

- return to application
- application performs cleanup and resumes execution

Details

- TLB refill must be fast

→ maintain 4096-entry cache in kernel
fast path swtlb hit → 18 cycles

QUESTION: what if user's TLB miss handler has a TLB miss?
Answer: keep a few pages in a special segment for "bootstrapping" before jumping to user on exception, check to see if page is one of these "special" pages. If so, kernel does TLB refill itself.

8. Processor scheduling

(Not a complete discussion here, but some questions/comments)

They give basic vector-of-time-slice framework

+ Simple model
+ Can build real-time scheduler (grab one slot every 16ms), gang scheduler (grab one slot on each CPU for same slice), compute-bound scheduler (grab lots of slots in a row to minimize context switch overhead)

How does cross-process coordination policy work?

- Above is fine if only one job running.
- Need **policy** to decide among competing claims
  - How would you build multi-level feedback?
  - What if one jobs wants response time (lots of slots scattered evenly) and another wants throughput (cluster slots into long shots to minimize context switch) – who wins?
  - How would you build stride scheduling?

  They give stride scheduling example

  How does this work?

  - Scheduler process grabs all of the time slices in vector
  - Other processes register with scheduler process
  - Exo-scheduler gives CPU to scheduler process at each slice
  - Scheduler processor yields to other process

  (How is this different than microkernel with a special user-level scheduling process? Perhaps they can make this "opt in" – processes that want stride scheduler to schedule them donate slots to server; others take charge of own slots…)

- Is this the right "minimal mechanism"?
- Compare with Vin's hierarchical scheduler (any advantages? Any disadvantages? How does this compare to uploading scheduling algorithm code?)

How does interactive/IO bound scheduling work?

- First paper: An interactive job grabs one slot every few slots to make sure it is scheduled frequently
- Any problems with this?
  - [big waste – most of time my processes are waiting for keyboard input]

    - [how to make self-tuning – as more processes arrive, each process needs to "claim" fewer slots]
    - [responsiveness gets worse as # jobs increases – see ASH figure below]
- Solutions
    - Paper 1: ASH – for special case of network IO, inject code into kernel that is run when interrupt occurs
        - Could probably generalize to other IO events?
        - Is a common ASH "schedule my job in next free slot?"
    - Paper 2: Wakeup predicates – before running a job in a slot it has claimed, check this predicate and don't schedule it if false. [allows job to, e.g., wait for page fault to be serviced]
        - Still not quite what you want – want to get scheduled when you are not scheduled not avoid being scheduled when you already have a slot?
    - Traditional abstraction – process the IO, demux it, identify who was waiting for it, put that job on ready queue to be scheduled "soon"
    - K42 work…

Wakeup predicates

9. Networks

Example of downloading code into kernel.

1) Multiplexing the network – packet filter
idea: load a small piece of code that examines packet and decides 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 to make sure that a new filter never overlaps with an old one (does that solve problem?) (today: I can listen on any port (numbered larger than 1024) that is

not currently being used – whoever claims a port first, gets it. More or less the same in Exokernel…)

2) application-specific safe handlers (ASH)
Load handlers for application-specific messages into kernel
→ can reply to packet w/o context switch

example – auspex file server responds to NFS *getattr* requests in hardware in network interface

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
- *danger of deadlock?*
- control initiation – "active messages"

Figure 2 compares ASH to no-ASH for Exokernel

- W/o ASH – exokernel just drops message in application buffer and later, when application is scheduled, application handles it → round robin scheduler → linear increase in ping latency

- What would happen in, say, Unix?

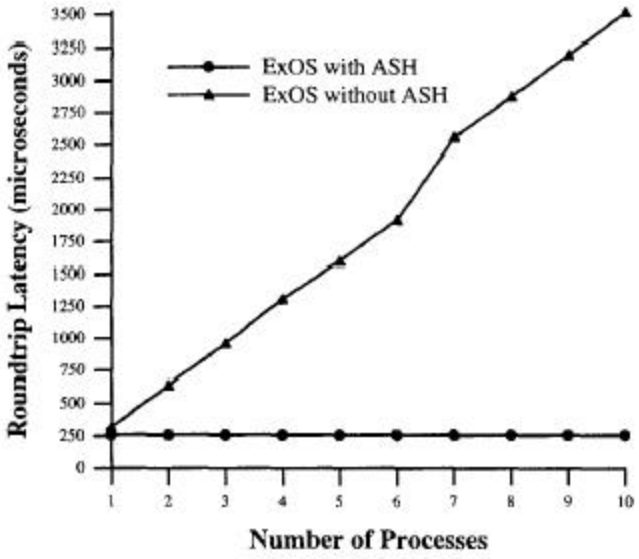


Figure 2: Average roundtrip latency with increasing number of active processes on receiver.

10. XN: Disk abstraction (follow-on paper)

How would you build a file system in Exokernel?

Strawman: Minimalist approach

- Allow application to DMA between memory page and disk sector
- Access control: Check capability to memory page and disk sector
  - Kernel keep a table sector number → capability
  - User-level file system: start with well known root directory and capability; embed capabilities in directories, inodes, etc.

Arm wave: I could build a file system with this…

Problems:

- Scale
  - Capabilities array is essentially another piece of file system metadata (~FAT table?) → add one disk access per read/write
  - Caching can help, but cost still is high
  - "Brute force" capability system OK for memory, but for large file system, may need to be more careful…
- Shared Caching
  - Above would allow per-application cache

    - Not really what we want
        - Short lived processes → want cache to survive process death
        - Widely shared files (binaries, libraries) → want cache to be shared across processes
        - Read/write shared files
        - Write buffer: What if process exits before flushing dirty file to disk?
    - Need kernel-managed file system cache
        - No longer so minimalist – common naming/location convention across processes, common replacement policy (how does user-control of replacement work for shared pages?)
- Sharing with mutual distrust
    - 3 cases for sharing (follow on paper)
        - mutual trust → easy
        - one-way trust → not too bad
        - mutual distrust → hard
    - Problem: file systems fundamentally about sharing
        - → need to share file among multiple users
        - → need to share disk among multiple file systems
    - Global invariants across FS
        - E.g., directory structure, free list, order of writes to allow recovery
        - Even if your process is allowed to write directory /foo/bar, it must obey invariants (no loops, no repeated inumbers, free list corresponds to free sectors, …) [in FFS, imap and free list are global data structures – who can write them? Directory updates are "raw pointer writes" to these global data structures – who can write them?
        - How to enforce ordering constraints (e.g., don't update pointer to inode until inode has been initialized) when you have a bunch of processes all issuing concurrent async writes?

HARD PROBLEMS – Exokernel went through 4 complete redesigns of the file system

(1) Disk-block-level multiplexing (see above)

[next two: essentially try to create a safe type system for disk; this would allow OS to prevent pointer forging, etc.]

(2) Self-descriptive metadata – metadata blocks (inodes, directories, etc.) start with headers describing the structure of the block (e.g., "the next 10 words are block pointers")

- "We discovered that this approach both caused unacceptable amounts of space overhead and required excessive effort to modify existing file system code, because it was difficult to shoe-horn existing file system data structures into a universal format."

(3) Template-based description – self-descriptive metadata + type system → only need to describe each type of block once per system instead of within each block

1) "This system was simple and better than self-descriptive metadata, but still exhibited what we have come to appreciate as an indication that the applications do not have enough control: the system made too many trade-offs. We had to make myriad of decisions about which base types were available and how they were represented (how large disk block pointers could be, how the type layout could change, how extents were specified.) Given the variety of on-disk data structures described in the file system literature, it seems unlikely that any fixed set of components will ever be enough to describe all useful metadata."

(4) XN

"Mechanism is policy" – keeping track of metadata -- which blocks belong to which files, who can access what, etc --

Want to give applications (libOS's) complete control over file system.
Problem: file systems fundamentally about sharing
→ need to share file among multiple users
→ need to share disk among multiple file systems

→ Exokernel forced to develop several fairly complex in-kernel systems to support libOS file systems

Tricky to get this right – complete control v. controlled sharing. This is their 4th design (did they get it right?)

3 new mechanisms

1) **Security** -- UDF – untrusted deterministic function – determine who "owns" disk page w/o specifying common metadata format
2) **ordered disk writes** -- "tainted" page state
   Why is this "fundamental" to sharing?
3) **share cached pages** -- Buffer cache registry

10.1 UDF

protection – who owns what block?

Approach – metadata on disk has a type T
each type of metadata has 3 UDF's defined for it

owns() → returns list of disk blocks owned by this metadata block
acl() → template-specific access control and semantic invariants; run before any metadata modification
size() → returns size of metadata object

Key idea: rather than define an interface "change structure X to add info Y" (declarative description language), allow libFS to define data structures and change them any way they want. Interface now checks to make sure the right thing happened

- Owns – verifies that change to list of owned blocks is what is purported
- ACL – XN doesn't know what the constraints are, but it can use per-FS code to validate constraints (e.g., "no two directory entries have the same name")

  NB: acl and size are actually not deterministic – they are in kernel but can be non-deterministic. Why?

UDFs are deterministic – only depend on input (simple language can be statically checked)
→ kernel can't be "spoofed" by UDF

Example: To allocate a block *b* by making *m* point to it

1) tell XN *m, b, diff* (diff is how to change m)

2) XN does O = owns(m)
3) XN does m' = m + diff
4) XN does O' = owns(m')
5) XN checks O' = O + b

Bottom line: kernel can make sure your metadata doesn't claim to own something it doesn't really own.
Notice – you can use any metadata format you want, but only kernel is allowed to modify it.
Kernel doesn't know what is in metadata, but you tell it what to do
UDF makes sure that you don't mislead it

QUESTION: Why is owns a true UDF (in kernel, deterministic) but acl() and size() are in kernel but may be nondeterministic?

Owns needed for cross-libFS security/integrity while ACL and size are only relevant within a given file system. Still need to be in kernel to **allow** a libFS to enforce security, but if libFS gets it wrong, it only hurts libFS.

Example: Unix FFS

- What are types?
- What are owns(), acl(), size() functions for each type?
-

Inode

Owns(): up to 15 tuples
Up to 12 tuples {type = raw block, start block, nblocks}
Up to 1 tuple {type = indirect block, start block, nblocks = 1}
Up to 1 tuple {type = double indirect block, start block, nblocks = 1}
Up to 1 tuple {type = triple indirect block, start block, nblocks = 1}

ACL: run before any metadata modification
e.g., if a user asks to modify inode, return false if (a) user doesn't have permission or (b) write fails to update size to be consistent with new set of allocated blocks or (c) write fails to change the last modified time to current time +/- delta, or (d) write munges data structure to illegal state or …

single/double/triple indirect node

Owns() a bunch of tuples with type/range
ACL(): similar to (but simpler than) Inode

Directory – unix says "directory is just a file" – should XN treat directory as a file or as metadata?

Owns(): a bunch of tuples of type {type = inode, start block, nblocks}

ACL: enforce no repeated file names, enforce proper formatting of directory (e.g., list of variable sized entries or list of fixed sized entries or tree of entries sorted by name or…), enforce permission to write, …

10.2 Ordered disk writes

Problem – some metadata depends on other metadata; if kernel (or application) writes to disk in wrong order, crash could wipe out data

Constraints

1) never reuse an on-disk resource before nullifying all previous pointers to it
2) never create persistent pointer to block before block is initialized → *tricky*
3) when moving a resource, never reset the old pointer before new one is set → only matters within file system, not exokernel's problem

Conceptually simple, but tricky to do efficiently. Naïve implementation will do many unneeded synchronous writes.

1 and 2 required for global integrity (enforced by XN) but 3 only affects a given libFS. Why?

What if 1 is violated?
What if 2 is violated?
What if 3 is violated?

QUESTION: How to do 1 and 3?
(Paper just says "the first rule is implemented by deferring a block's deallocation until all on-disk pointers to the block have been deleted; a reference count performed at crash recovery time helps the libFSes implement the third rule")

Perhaps: keep reference count in each entry of free list;

- since each modification of metadata explicitly tells XN the change to owns() list, XN can maintain reference count, right?
- use lazy writes (+ recovery after crash) to avoid extra synchronous writes

DA: requires scan of disk on recovery

DA: increases size of free list and/or puts upper bound on reference count

Other options?

How to do 2)?
System tracks *tainted* blocks

A block is tainted if it points to an uninitialized metadata block or it points to a tainted block

When allocating a block, I update a metadata node
→ mark it tainted

But wait, parent metadata node is tainted too!
When I bring in any page (e.g. the metadata node), the buffer cache registry keeps track of its parent → I can find parent and mark it tainted (and on up the tree)

QUESTION: create a new empty file in FFS

- get block B from free list
- add block B to directory D
  - XN.add(old D, new D, B, TYPE_INODE)
  - XN marks D as tainted, B as tainted, increments B's reference count
  - Does XN need to mark D's inode as tainted? (No, since FFS doesn't change the location of D; in LFS (next week) yes)
- Initialize B
  - XN.initialize(new B, B, {})
  - XN marks B as untainted
- Write B to disk
  - XN marks D as untainted
- Write D to disk

QUESTION: When should we update the free list on disk? (If we do it first, what happens if we crash? If we do it last, what happens if we crash?)

10.3 Buffer cache registry

Allow protected sharing of blocks among libFSes
Track mapping of cached disk blocks and their metadata to phys pages

Pages, themselves, stored in application memory

Buffer cache registry tracks mapping and state

Anyone can order cache to write a block (subject to *tainted*)

10.4 FS questions and lessons

Lessons

- "Plan to build 2, you will anyhow"
  - XN is 4th iteration of FS design, Xok is 3rd exokernel
  - Engler could have called it good after first Exokernel
  - Learned a lot of deep lessons by iterating
  - Patterson rule: build it to find out what the real questions are
- Cross-disciplinary work/skills → creative new approaches
  - Engler comes from a languages and compilers background and many tricks in Exokernel build on this
  - Opportunities: AI/OS, theory/OS, architecture/OS
- Writing trick
  - Often a temptation to "tell the story" of how you came up with a solution. But tedious for reader to hear the story before knowing where you are going with it
  - Comes up over and over in writing
  - Almost always the right thing: put short discussion of essence of idea first
  - Example: Section 4 – 4.1 is "overview" and 4.2 is "Problem and history"

Questions

- This still seems complex
  - How doe the constraints drive complexity

- "Creating new file formats should be simple and lightweight…should not require any special privilege"
- "Protection substrate should allow multiple libFSes to safely share files at the raw disk block…level"
- "Efficient..as close to raw HW performance as possible"
- "Facilitate cache sharing among libFS's"

Compare
v. partitions

v. 1 MB volumes (e.g., allocate blocks to different file systems in 1MB units) how would you change design then. What would you gain? What would you lose?

- Section 4.4 garbage collection sounds expensive – need to scan entire disk to reconstruct free map

  FSCK is considered really expensive today
  - How to modify Exokernel to avoid this need (they say "if rebuilding the free map after a crash needs to be fast, this step can be eliminated by ordering writes to the free map" How would that work?)
  - For each libFS, it seems like we would want it to be the case that the libFS can either enforce invarients to eliminate the need for FSCK within the libFS or it can do lazy garbage collection later. How does this work?

2 project ideas
I think XN is cool "extreme" research – get at essence of file system abstraction (even if not quite what you would really ever build…)

1) They promise as future work to implement a bunch of existing file systems in XN to test whether their abstractions really do let you express "real" file systems. → Build LFS or XFS in XN and see if it really captures the essence
2) Modify XN to eliminate need for fsck on recovery

11. PART3: Critique, Questions

1) Lessons (section 9.3 of 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"

2) Will improved performance matter or will increased processor speed make the effort moot?

3) What happens when many users/competing libOS's on a system?

Exokernel must decide who gets CPU, who gets memory, etc.

e.g. CPU scheduling between batch, interactive jobs?

e.g. Page-hungry processes (react to page revocation by asking for a new page)

→ need some policy

microbenchmarks in paper are best-case for exokernel

things like multi-level feedback scheduling are complex, but they got invented for a reason. How would exokernel do with such a workload? Would it "discover" something like MLF?

Does anyone buy their scheduler discussion?

1) Will implementors of applications be willing to invest effort to build new libOS?

4) What is cost of migrating to model?

5) Do you buy their conclusions
   a) exokernel can be made efficient b/c small, low-level primitives
   b) low-level multiplexing of HW can be efficient
   c) traditional abstractions (e.g. VM, RPC) can be provided w/ low overhead
   d) application can create special-purpose implementation of abstractions

4) why are non-exokernel systems (e.g. Ultrix getpid()) so slow?

QUESTION: (written critique) what is the "fatal flaw" in the paper?

Project ideas

- Both exokernel and Disco define virtual machines. Compared to Disco, Exokernel is much less portable – it is harder to build/port an exokernel than a virtual machine monitor for a given piece of hardware (is it?) and if you want to run legacy applications, it is harder to build a libOS over an exokernel than it is to run an existing OS over a VMM (is it?) On the other hand, compared to Exokernel, Disco may give up a lot of performance opportunities because it hides virtualization. Also, as more and more performance optimizations get put into Disco, it may become as complex as a traditional OS. Build the simplest VMM possible so that existing applications/OS's can run without modification and so that the VMM is easy to maintain, and then add a few Exokernel-like extensions to the system. These extensions should not complicate the core abstraction…
- Add Exokernel's virtual memory interface to linux as an option. That is, allow a process to request that allocation return a physical page. Then, the process would manage virtual->physical mappings, page replacement, etc. Compared to Exokernel, this approach will probably be slower (how much?) and may give up some flexibility (does it?) But, the advantage is that the approach can be retrofitted to an existing OS. Are there sufficient gains to be worth it?

- Add Exokernel's network handlers (with improved security), UDFs for disk (perhaps at the Linux volume manager level?), or interrupt handlers and explore similar issues to the virtual memory issues discussed above.
- New programmable router architectures promise fast, flexible networking. We have an Intel network testbed of such machines here at UT. The question is: how to program them. Build on the Synthesis "pipeline of servers" model, the Scout path model, and/or the Exokernel ASH model (or traditional packet filters) to construct a flexible, high performance programming environment for such architectures.
-