Application Performance and Flexibility on Exokernel Systems
MIT CSAIL, 1997

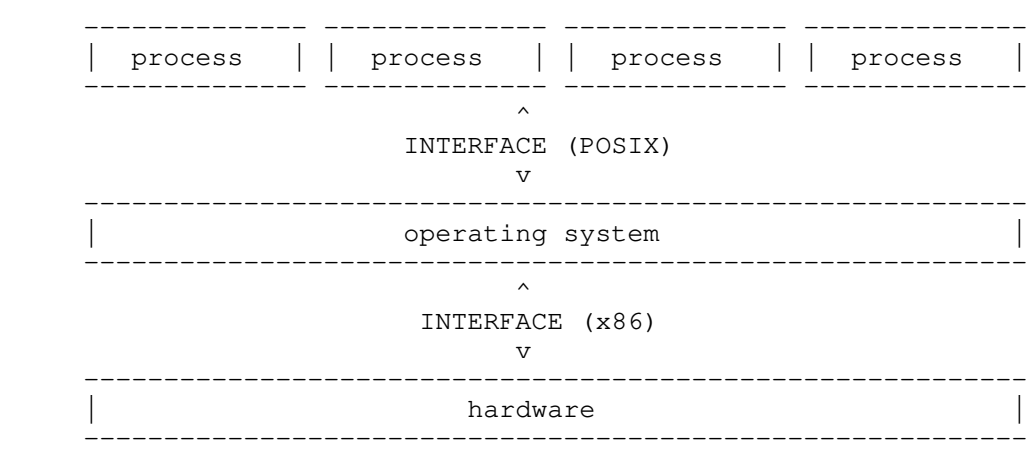
Administrivia
  - Lab 2 has been released.
    - There's more to figure out here than in lab 1.
    - Really requires you to have read the MapReduce paper.
    - Plenty of opportunity to exercise your new Go skills.
    - Can't use `corn`: need Unix domain sockets, can't with AFS.

Tiny revisit to Monday's lecture: generics in Go.

[Golang slides - Internet Post]

Exokernel Lecture
==========

Q: What's an operating system?
A: Traditionally, an operating system protects and manages (hardware) resources
   by exposing some kind of interface to multiplexed hardware.



resource	POSIX	x86
compute	processes, threads	processors
volatile memory	address space (mmap)	physical, virtual memory
persistent storage	files	hard disk (blocks)
async. notification	signals	interrupts, faults, exceptions
network I/O	sockets	rings buffers
communication	IPC (p = process)	IPI (p = processor)

Q: What's the problem with the abstractions that POSIX provides?
A: They're fixed: that's all you get. This might result in:
  1) lower performance
      - There is no single _best_ way to abstract resources
      - OS makes tradeoffs and tries to appeal to many app workloads
  2) hide important information
      - all trapped behind heavy-handed abstractions
      - don't get interrupts, exceptions, can't do raw I/O, etc.
      - example: can't implement scheduler activations since:
        - we don't get notified about being descheduled
        - no information about the resources we have to compute on (CPUs)
  3) limit functionality
      - too hard to change, so have to live with existing abstraction
      - maybe why research abstractions (scheduler activations) didn't make it

The key here is to note that POSIX (etc.) _couple_ resource protection with
resource management:

    Protection = can it happen? who can use it?
    Management = when and how does it happen?

  * compute protection/management: processes isolated and are scheduled by OS
  * memory protection/management: always virtual addresses, given out by OS
  * persistent storage: file system protects and manages blocks
  * async. notification: OS determines who gets signals when
  * network I/O: OS determines if read/write can happen, when, and how
  * communication: OS determines whether IPS can happen, when, and how

Q: What is this paper trying to accomplish? Why do we need a new OS arch.?
A: Fixed heavy abstractions are bad for several reasons:
  - denies domain-specific optimization (GC wants VM, DB wants Disk, etc.)
  - discourages changes to implementations of abstractions
  - restricts the flexibility of application builders
    - new abstractions must (if even possible) use old abstractions

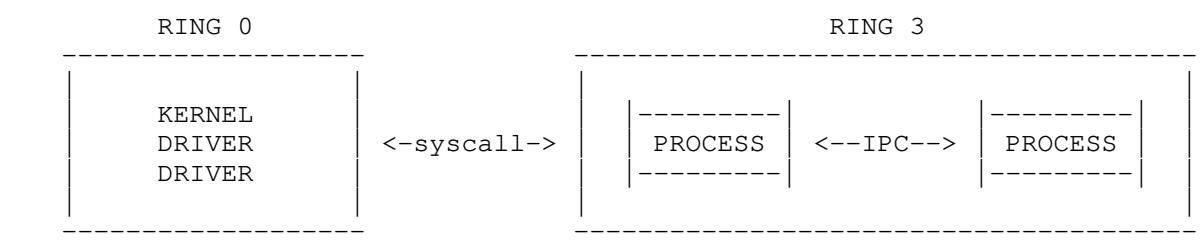
Idea is to _export_ low-level resources in some secure manner.
  - Can then implement resource management in user space.

Q: People seem to be okay with Linux. Why?
A: Kernel modules, mostly. I can change the kernel. Also ioctl.

Q: What does Dawson say about kernel modules?
A: Brittle and must be trusted! They can mess up the entire system.

Popular OS Architecture: Monolithic and Micro

First, the monolithic design:
  * Protection and isolation provided by hardware.
    * Protection: Rings
    * Isolation: MMU
  * Two "types" of software, essentially:
    * Kernel code: kernel, drivers, IPC services, etc.
    * User code: processes, threads
  * Kernel <--> User communication happens via syscalls
  * Kernel <--> Driver communication happens via function calls
  * Driver <--> User communication happens via syscalls
  * User <--> User communication happens via IPC (pipes, sockets)
    * Can also share memory, but less commonly used.

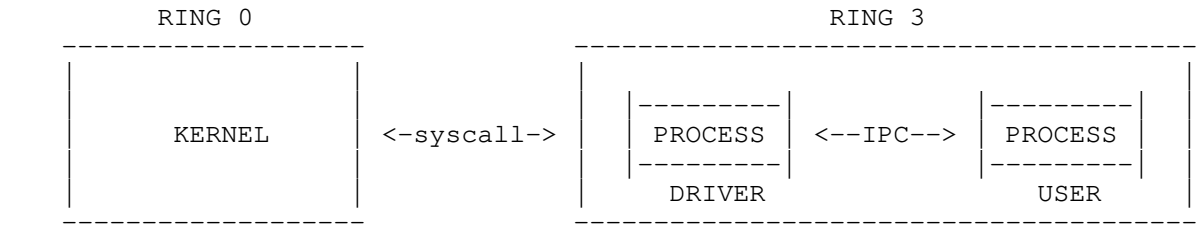


Advantages:
  * Tends to be fast!

Disadvantages:
  * Kernel implements a lot of driver interfaces.
  * Driver crashes = machine crashes.
  * A lot of trusted code => exploits easier to happen.

Now, the microkernel design:
  * Protection and isolation provided by hardware.

    * Protection: Rings
    * Isolation: MMU
  * Two "types" of software, essentially:
    * Kernel code: kernel, IPC services
    * User code: drivers, processes, threads
  * Kernel <--> User communication happens via syscalls
  * Kernel <--> Driver communication happens via syscalls
  * Driver <--> User communication happens via IPC
  * User <--> User communication happens via IPC



Advantages:
  * Kernel is much, much smaller: no driver code, minimal driver interface.
  * Drivers are now untrusted! Less exploits, hopefully.
  * Driver crash != kernel crash.

Disadvantages:
  * IPC tends to be slow.
  * Drivers now need to syscall to get into kernel.

Q: What kind of architecture do Linux, BSD, OS X, Windows, iOS, and Android use?
A: Monolothic.

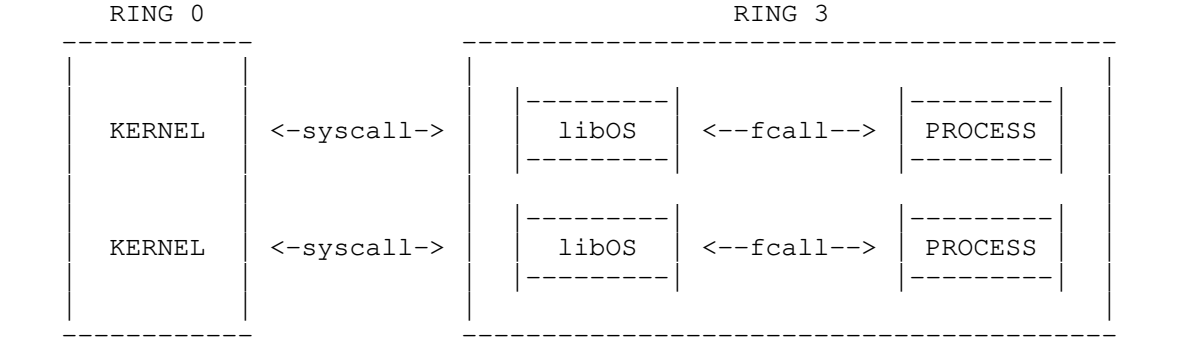
Q: Why? And does anyone use the microkernel architecture?
A: Speed, speed, speed.

Exokernel vs. LibOS

How do we write an exokernel?
  - Export low-level resources in some secure manner!
  - Still want kernel to protect resources. So still have syscalls.
    - But syscalls should be super fast since abstraction is minimal.
    - But syscalls should expose low-level hardware details (in nice way).

An exokernel exports hardware resources via:
  1) secure bindings: application can bind to resources and handle events
  2) visible resource revocation: cooperative resource revocation
  3) abort protocol: forced resource revocation by kernel

Exokernel proposes that we modify previous lecture's picture to look like:



Now, the 'syscall's are stricly to bind to resources. libOSs make all of these

syscalls, and user-level applications usually don't.

Q: What does a LibOS do?
A: It implements abstractions: manages the resources.

Q: So...now we have to replicate Linux a bunch of times?
A: Library Operating Systems are simpler than a regular OS
  - The don't multiplex: just implement abstractions
  - They're not trusted by the kernel, so can trust the app.
    - Only the app will be affected if the LibOS messes up because of it
  - The LibOS will need to cross rings less since it implements most of logic
    NOTE: Not sure I buy this argument.

Q: What about portability?
A: Applications aren't necessarily less portable because they target a LiBOS
    - LibOS could just implement POSIX for example
    - Expected that mostly a handful of LibOSs will exist

Q: What's the point of only a handful of LibOSs are expected to exist?
A: Apps that need the flexibility still have it. So, while many apps are okay
   with the inflexibility of POSIX, a database might want to handle the disk
   itself.

Exokernel

So, the previous was the philosophy here. That's the idea. The kernel of the
idea, at least. Now we want to instantiate it. How do we make an Exokernel?

Again, three main ideas. An exokernel exports hardware resources via:

  1) secure bindings: application can bind to resources and handle events
  2) visible resource revocation: cooperative resource revocation
  3) abort protocol: resource revocation by kernel

Exokernel should only manage resources so that protection can be enforced
  - no more than this! Leave everything else to the LibOS

A few principles lead to the realization of these goals:
  1) expose allocation: allow a LibOS to rquest specific physical resources
    - along with this, don't allocate resources for apps implicitly
  2) expose names: remove a layer of indirection by exposing real names
    - i.e: led LibOSs know physical page numbers to check for cache conflicts
  3) expose revocation: allow well-behaved LibOSs to be effective by notifying
     them when you revoke their resources

Secure Bindings: securely multiplex resources amongst LibOSs
  - Such efficient and isolating bindings are called secure bindings
  - IE bind a subset of the resource (like some of the network or some of the
    physical memory space) in a secure way, so that two LibOSs can't mess with
    each other's allocated resources, and you want to do this efficiently

Example #1: Compute

  LibOS wants to run some code! How does it get compute resources?

  Aegis, in particular, seperates time into slices and allows a LibOS to ask for
  certain slices. They can ask for them in any configuration.
    - want join slices for things like scientific computations
    - want separate but equidistant slices for responsiveness

  1) LibOS requests some number of time slices in some configuration
    - partition time into a linear vector (logically)
      - partitioned at the clock granularity: can't partition less than clock

    - 15.625ms
  - Exokernel checks to ensure that this is possible, or something.

2) Scheduler issues timer interrupt to application on wake and exit
  - wake = start of contiguous time slice
  - exit = end of contiguous time slice
  - similiar to scheduler activations
  - LibOS must save its own state for next time slice
  - If LibOS takes too long to context switch out, exokernel records this
    - recorded in 'excess time counter'
    - if too many time slices are in excess, LibOS is destroyed

Example #2: Multiplexing Network w/Packet Filters

LibOS wants to receive some network traffic. On some port or IP or something.

1) LibOS sends a packet filter to the kernel.
  Q: What's a packet filter, actually?
  A: Usually a description of which packets to accept. Maybe be code.
  - This is BIND time.
  - Kernel checks that the filter doesn't step on other LibOSs.
    - IE, it won't steal traffic other LibOSs want
    - How can it actually do this?...
      - Restrict language a lot.

2) Packets come in. Kernel checks against filters.
  - Don't need to context switch into user space to do this.
  - This is ACCESS time.

Example #3: Multiplexing the disk. (XN)

LibOS wants to create its own file system. Needs disk blocks. Not in this paper: Dawson will explain this later.

Answer: Use need UDF trick.

Idea: Have pure (and thus deterministic) functions. User gives you a pure user-defined function (UDF) that, given some state and a block number, determines if the block is owned by that LibOS. The structure of the state is chosen by the user (example: free block table, inode map, etc.).

When the user requests a blcok, you run the function with the current state and ensure the function says it does NOT own the block. Then you give the user the block and allow them to update their state. You rerun the function with the new state and ensure that they now say they own the block. Excellent.

Now, given a block numer, can easily tell who owns it without duplicating ownership information: just call the functions.

XN Usage (!)
  - We're going to walk through how a libFS begins its life and reads and writes it disk blocks. Then how another libOS can read/write its disk blocks.
  SETUP
    1) libFS installs templates for its metadata types
      - installed into a "type catalogue"
      - each template can be referenced by a unique string:
        - "ext3 inode", "ext3 superblock"
    2) the "root" block is registered in the "root catalogue"
      - IE: the "superblock" in ext
      - identified via a unique string "ext3" and its template type
  STARTUP
    1) libFS starts up, loads its root block from "root catalogue"
      - ext3 would ask for "ext3" root block: superblock
      Q: How is the root block protected?
      A: No idea. Probably just given to the "first" to ask for it. IPC after.

ALLOCATE (copied from UDF section)
  1) libFS chooses a block ‘b‘ from XN’s free list
    - tells XN that it’s allocating a block by calling some syscall
    - syscall takes as input the block ‘b‘, metadata ‘m‘ for the parent block that will point to the new block, and modifications to ‘m‘ (as set of bytes to write to ‘m‘) that make ‘m‘ point to ‘b‘
  2) XN verifies the allocation as follows:
    a) XN runs ‘owns-udf‘ on ‘m‘, ensures ‘b‘ isn’t there
    b) XN writes the modifications to ‘m‘ -> ‘m’‘
    c) XN runs ‘owns-udf‘ on ‘m’‘, ensures ‘m’‘ has one more block: ‘b‘
      - if ‘m’‘ isn’t correct, modification is reverted, allocation denied
WRITE
  - requires the block to already be in the registry, so first do...
READ
  1) ask for a page in the buffer cache registry for block
    - need to supply the block that’s going to be read
    - need to supply the parent of the block
    - XN checks that the parent is already in the registry
      - and that the parent owns the block (via owns-udf)
    - XN checks access control through ‘acl-uf‘
      - unclear what capabilities are passed here. "hierarchical"..
WRITE
  - just check ‘acl-uf‘, presumbaly, though the paper doesn’t actually say anything about this.