CS 736 – Minimal Abstractions

1. Minimal abstractions: does an OS need to provide abstractions, and when?
2. Questions from reviews:
    a. Why is arrakis not trivial thinking about u-net, other user-mode networking?
        i. A: preserve Posix interface for sockets
            1. Past work tended to use RDMA or other protocols, hard to program
        ii. Handle disks
            1. Past work only handled networking
        iii. Partially, it is not that new
    b. Security: who implements control?
        i. OS: currently manages control/data, just removing data so no loss in security if HW does things right
    c. Would bypass be worth it with slow devices?
        i. No – percent overhead small
    d. Native vs Linux interface
        i. Exo kernel – interface closer to HW, smaller abstraction allow stripping out more code

3. **ExoKernel** – exterminate abstractions
    a. Background:
        i. lots of effort at customizing/extensible kernels (e.g. hydra, mach, vino, spin). Mostly aimed at taking an existing kernel and adding extensions, building a new architecture that allows extensions to be downloaded, or moving functionality to user-mode where it can be replaced.
        ii. Basic challenge: all these approaches tend to be slow
    b. Context
        i. Predated modern virtual machines – they are like an exokernel
        ii. Paper presented at HotOS workshop – for new ideas, not fully-baked systems. Later published 2 papers demonstrating performance, flexibility, sharing
        iii. Trying to be inflammatory – to get attention.
    c. Technology change?
        i. QUESTION: was there a technology change?
    d. Problem:
        i. operating systems getting big and bloated. QUESTION: HOW BIG IS A KERNEL? Linux?

ii. Abstractions offered useful for low-perf apps (sometimes), but mismatch between offering and what apps need makes programming difficult, hurts performance.
iii. Example: automatic virtual memory. App may have better sense of its needs, e.g. what pages to replace and when

e. Question: how do you get the extensibility, flexibility, sharing and protection with all the overhead?

f. CLAIM: impossible to provide abstractions good for all applications and efficient
   i. Let's examine claim:
   ii. What is an OS? code that securely multiplexes + abstracts hardware.
      1. Code one can neither neither change nor avoid (baked in!)
      2. Microkernel move code around, perhaps allow some substitution but not much.
      3. Securely multiplex: why? so can run multiple programs
      4. Abstract: why?
         a. Makes multiplexing easier (at higher level, conceals some details. E.g. file systems)
         b. Easier to write program - don't need to deal with low-level details
      5. Lots of good ideas proposed but not adopted by OS - demonstrates limits of current OS design
         a. how do we know all the ideas are good?
         b. lots of things are adopted into operating systems
g. Current design leads to:
   i. **poor reliability,** because complex systems needed (e.g. vm, COW, mmap, multithreading)
   ii. **Poor adaptability**: hard to add new policy, mechanism. Change not localized because OS applies to all apps
      1. **Lots of** linux changes rejected because some apps depend on old behavior
   iii. **Poor performance**: abstractions take time to execute; applications that don't need them still pay. Example: GC or DB that interacts poorly with VM and would do better managing memory itself
   iv. **Poor flexibility**: apps can't implement their own abstractions, just emulate on top of existing OS at high cost
h. Discussion of value of abstractions

i. Benefits
    1. Higher level of programming
    2. Code reuse – don't have to rewrite low-level code
    3. Higher-level policy; can enforce policies with more knowledge
        a. E.g. more information about sharing & cooperation
    4. More security: can group information for finer-level access control
    5. Can share at a higher level – e.g. files instead of blocks, to get better consistency semantics
    6. Easier to write programs – can deal with things you think about (contiguous memory, files), not hardware
    7. Performance:
        a. Can more easily overlap operations of multiple processes
            i. E.g. disk scheduling
        b. Can do system-wide caching
            i. File system caching
        c. Have semantics to make better decisions
            i. What blocks are data vs metadata

ii. Drawbacks:
    1. Performance: wrong abstraction means what you want is very expensive
        a. E.g. file systems when you want block storage, or have large files, or billions of small files
        b. E.g. networking – can't get to inner access of how protocols work, hard to avoid copying
    2. Hard to change monolithic kernel – difficult, complex code
        a. E.g. optimize to deliver network packets right off disk to web server
        b. Issue isn't as much one of booting into another OS, as to modifying to do what you want
    3. End-to-end argument
        a. Application knows best
        b. application must handle things anyway
        c. might as well give power to the application to do things their way
            i. Optimize for mutual trust instead of distrust

a. Solution:
    i. Big idea:

1. Kernel provides minimum necessary to securely multiplex hardware, but no abstractions (if possible)
2. **OS runs as a library attached to application, provides abstractions**
    a. **should be easier to extend, better tuned to applications**
    b. **provides same high level abstractions to make it easy to program**
    c. **EVERYTHING that used to be in the kernel other than sharing & protecting hardware moves to a library IN YOUR PROCESS.**
    d. **THEY IMPLEMENTED POSIX interface in a library, to run standard Unix code**
3. Idea is NOT:
    a. have a trusted user-mode implementation of a service
4. Idea IS:
    a. Run the OS code yourself in your process
    b. NO trusted third party
5. **DISCUSSION: what happens to compatibility?**
    a. **Answer: you have this in the form of libraries, but you can bypass it or modify the libraries.**
    b. **E.g. can use raw packet send instead of standard TCP/IP implementation**
    c. **E.g. a database can directly write to disk and manage buffering, instead of using file system**
6. **WHAT HAPPENS TO RELIABILITY?**
    a. **You run less code.**
    b. **You can run common shared code if you want (but at lower performance).**
    c. **Kernel crashes are worse than user-level crashes, as they impact everything and force a reboot**

ii. Minimal kernel, exokernel, that provides access to hardware if at all possible
1. QUESTION: How?
    a. Provide base minimum: wired pages for exception handlers and page table, addresses of exception handlers

   b. Ensure safety at all times
      i. No use of other's resources (e.g. memory, CPU)
   c. Allow safe code downloaded to kernel
      i. For packet filter to decide which process gets a packet
2. Expose hardware names – e.g. physical address
   a. So can do HW-dependent things, such as page coloring
   b. Use SECURE BINDINGS:
      i. When first using a resource, check access and cache.
         1. E.g. a TLB: verify mapping, then let it be used
         2. E.g. packet filter: verify selects right packets, then let it use
   c. For memory:
      i. Create capabilities to each page accessible by a process, which it presents to establish mappings.
      ii. Provide ability to insert into a TLB (could be software TLB in kernel) or remove
      iii. QUESTION: How compare with normal address space?
         1. Can use physical addresses directly, can use large pages if want, or page coloring`
   d. Processes:
      i. Provide address of an exception handler
         1. Know where to start code
         2. What to do on seg fault, divide-by-zero, etc.
         3. Know what to save/restore on context switch
3. Expose hardware events – e.g. swapping memory out
   a. So can choose what do do
4. Expose revocation: ask libos to do revocation
   a. Take away a CPU: libos may want to do some scheduling, or decide what state to save
   b. Take away memory: update PTE

5. **WHAT ABSTRACTIONS DO THEY KEEP?**
    a. **Capabilities: a right to use a HW resource**
        i. Check capabilities on use
        ii. Use HW support – e.g. TLB, or tables for disk blocks. Very lightweight
6. Provide SW TLB – apps can fill in own mappings safely
7. QUESTION: how do you handle resource sharing?
    a. Don't have enough policy in the kernel to make accurate decisions

iii. Provide abstractions in libraries – programs can choose what libraries they want

1. QUESTION: Who writes these?
    a. Could be OS developer, but leaves open modules for extension.
    b. Could be a port of an existing OS for compatibility
    c. Can run multiple simultaneously
2. QUESTION: What are implications? Do app developers have to write their own OS? Do app developers have to write own window routines in X because in user space, compared to windows? But can X be changed more easily?
    a. Can think of OS code being more modular – can replace parts of it, just for your app. Can leave interface alone, or subclass & add new features.
3. QUESTION: portability- how handled?
    a. A: library has hardware abstraction layer. But need not, if want extra performance
    b. E.g. routines to manipulate page tables, handle specific exception formats
4. QUESTION: What are assumptions
    a. Not for timesharing; relatively trusted libOS because computer user chooses it,
    b. Probably not for malicious applications
5. Each application links to a libOS, which provides OS-level APIs
    a. Applications trusting each other can have a libOS that shares state
6. LibOS chooses what state to save/restore on context switch

7. LibOS can be customized to the application – different mechanisms and policices
8. QUESTION: Impact on complexity?
    a. Previously OS / device drivers hide complexity – only do once
    b. Now: all libOS have to do it, or share code
    c. Now: can have libOS that only has the features you need – leave out everything else
        i. makes libOS simpler
    d. When you need new functionality:
        i. need to change libOS
        ii. maybe copy code, maybe reimplement
        iii. e.g. static web server -> dynamic web server
        iv. Kernel handles dependencies between services, but in libOS, when add services, need to manage this separately

iv. Provide some functionality via safe downloaded code into kernel
    1. Code has guaranteed completion
    2. Limited access to memory
    3. Used for packet filters, event notification (wake up), file system block translation

v. Big issues are sharing
    1. If kernel doesn't handle it, how does it happen?
        a. Between libOS – but can optimize for trust relationship
            i. Mutual trust (e.g. unix processes of same user)
                1. Allow sharing w/o OS intervention
            ii. Unidirectional trust (e.g. process/kernel, microkernel server)
                1. Trusted sides retains ownership of shared resources, e.g. page tables
            iii. Mutual suspicion
                1. Provide kernel support via. Downloaded code
                2. LibOS must treat all data suspiciously

iv. QUESTION: should you trust? How does that impact reliability / security?

b. Comparison to virtual machines (coming soon)
   i. Expose communication primitives (IPC)
      1. Better support for sharing (e.g. memory)
   ii. Some changes from HW interface
      1. Exception tables vs interupt vectors
      2. Software TLBs vs page tables

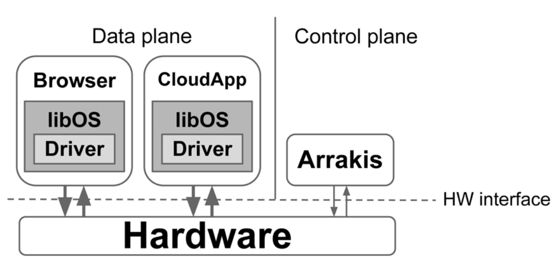
Arrakis:

1. What hardware enables this work?
   a. Self-virtualizing devices – enables user-mode I/O
   b. RDMA devices – user mode IO
2. What is the problem?
   a. OS overhead high on data plane to devices relative to modern devices
      i. File systems
         1. Can to read/write in 25us – compared to 1-10ms for disk
      ii. Network
         1. 10gbps network sends a packet in 1us, round trip with RDMA is 3 us
   b. Past solutions
      i. New APIs/data structures for zero-zopy
      ii. Hardware offload – TCP etc
      iii. System-call batching
3. Basic research approach:
   a. Benchmark a system see what is slow
      i. Profile to understand why slow
      ii. Network:
         1. Network stack – protocols
            a. Must demultiplex packets between processes
         2. Scheduler – context switches if process not running
         3. Kernel crossing – trap/return
         4. Copy – copy data into kernel buffers
      iii. Storage: similar
         1. Trap to kernel
         2. Scheduling waiting for I/O to complete
      iv. Application:
         1. For simple service (memory cache), most of time is in OS/IO path
            a. E.g. socket handling
      v. Why a problem?

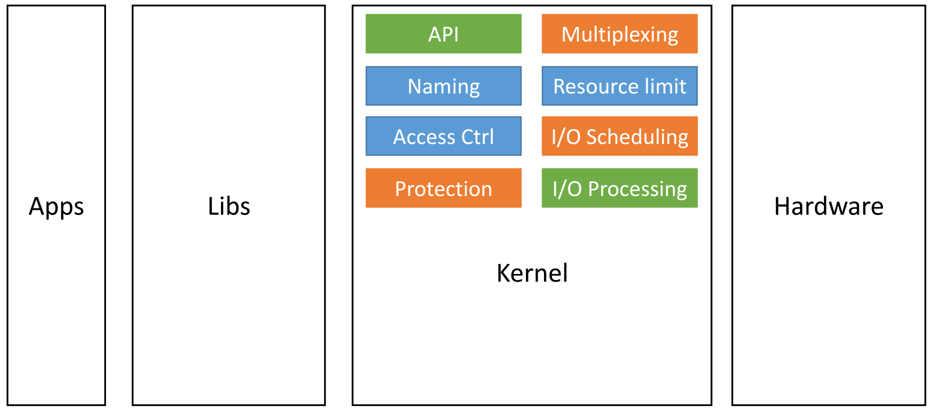
1. New hardware is fast: 40Gbit ethernet, 25us SSDs
2. SW is now comparable to HW; for hard disk or slow network SW is cheap
    a. 10ms seek time vs 100us FS time

b. What is new opportunity: self-virtualizing hardware
    i. Idea: HW knows about processes, can take independent request from different processes
        1. Sidebar: why not work normally?
    ii. How?
        1. Connection per process – queue of requests
        2. Rules to distinguish data per process
            a. Disk: req/resp queue
            b. Net: network address (port, IP address)
        3. Scheduling: mechanisms to share HW between processes
        4. OS retains control – creates connection to processes (limited number)
        5. IOMMU: allow using virtual addresses from user-mode
            a. Page table in PCIe bus does translation, knows about processes, so device gets correct physical data, maintains security

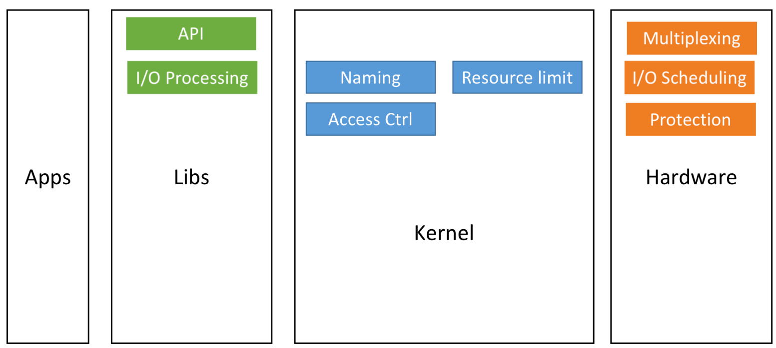
4. Arrakis
    a. Problem:
        i. I/O is fast, but kernel adds overhead
            1. Abstraction overhead (e.g. sockets, file systems)
            2. Time overhead: trapping/returning
            3.
    b. Big idea: data plane out of kernel
        i. **Control plane** == connection set up, control over who gets to do what, deciding policy on resource use
        ii. **Data plan** = actual requests to read/write data, send/transmit packets
        iii. **Origin:** comes from networking; connection establishment, forwarding policy (routing) vs packet movement (just forwarding)
    c.
        i. Once connection established, data movement an be done w/o kernel involvement, securely

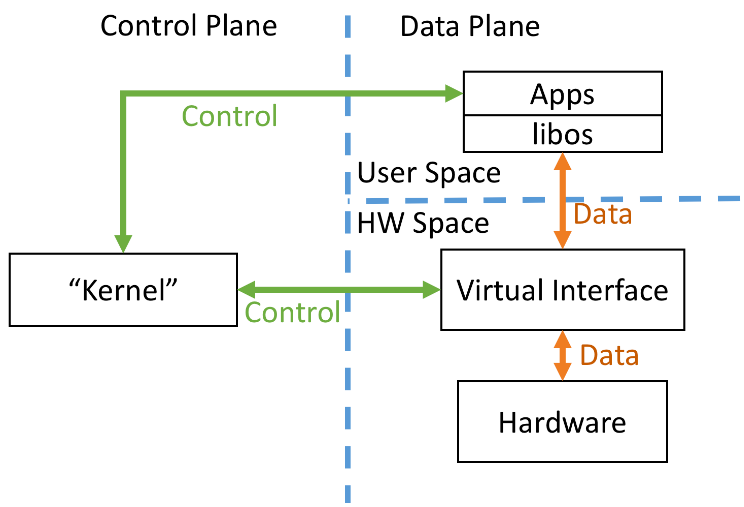


        ii.
        iii. How?

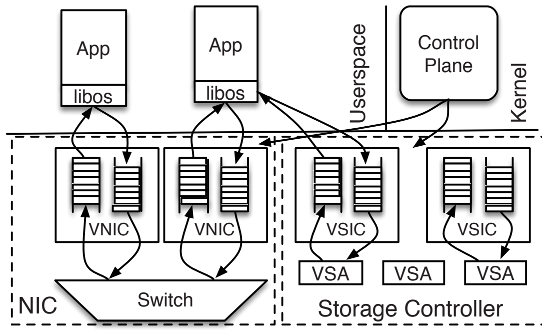
iv. 

v. becomes

vi. 

vii. 

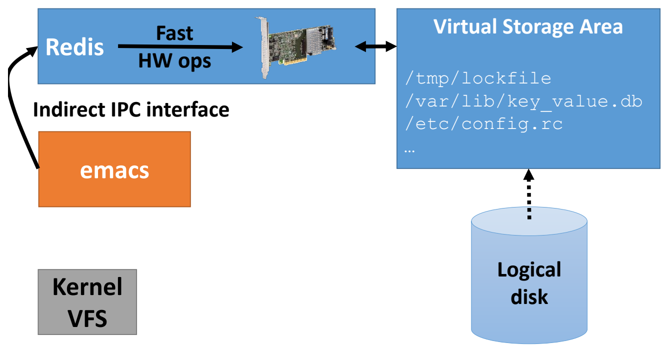
d. Retain kernel API but handle data plane in a library
    i. Implement POSIX apis
e. Abstract hardware enough to have generic apps, but not too much

i. 

5. Hardware model: how get hardware to be safely sharable?
    a. What is needed:
        i. Virtual interfaces to devices
            1. Per-app queues
                a. Pass virtual address of application buffers to send/receive
                b. VNIC for networking, VSIC for storage
                    i. Ensures req/resp go to correct process
            2. Per-app rate limiters (to apply policy)
            3. Per-app filters – what packets/data go to this app?
                a. Transmit: prohibit sending from someone else's port
                b. Receive: only get data for this app
                c. Read/write for disk: what range of blocks?
                d. **QUESTION: What is OS protection role for a filter?**
                    i. Must ensure non-overlapping filters
                    ii. Must ensure not take packets for rest of OS
                    iii. Must not claim too much port space
                e. **QUESTION: Why a capability to a filter?**
                    i. Allows passing between processes, like a socket or file descriptor
                    ii. Can create a filter, fork a process, give to child to take
            4. Challenges:
                a. HW may not support enough for every process
                    i. Solution: go back to SW for some processes
                b. HW may not filter on right fields
                    i. Limit to what can be used
                c. HW may not do protection for disks
                    i. Wait for it? Doable –no technical challenge, just business
            5. NOTE: research not stopped by what is currently shipping, but look at what is possible in HW even if not in current products
                a. Does not need new inventions in HW!

        b. Emulate in SW: hW interface, but use a CPU core for protection
            i. Poll queues for requests, copy to single HW queue, translate virtual addresses
            ii. Use same format, just apply protection rules
ii. Control plane inteface: how connect to devices/how manage them
    1. Basic I/O inteface
        a. Requests queues
        b. Doorbells: notifications (IPC) that data is available
    2. Network control
        a. Filter:
            i. type (transmit/receive)
            ii. predicate:subset of pkt headers
                1. IP addresses, ports, protocol types
                2. Example: port 80 with any sender; allows accepting connections for HTP
                3. Example: map/reduce: allows sending/receving over a port to a whole group of machines
                4. Details:
                    a. Flags – direction, protocol
                    b. Peerlist –other machines involved
                    c. Service list: allowed ports
            iii. Interface:
                1. CreateFilter – returns a capability to send/receive packets matching the filter
                2. Can attach a filter to a queue (or more than one)
    3. Storage control:
        a. Problem:
            i. HW only grants access to blocks
            ii. Want files, want sharing/protection of these files
        b. Virtual storage area: region of disk limited to an application
            i. Not used by other processes, so can safely read/write blocks there
            ii. Effectively like a partition
            iii. Easier abstraction than a file:
                1. Cannot grow (perhaps)
                2. Is contiguous (no indexing structures)
        c. Solution:
            i. process can export sub-tree (volume in Unix) to other processes

1. Total local control over data in sub-tree
2. Othres can mount remotely
    a. RPC endpoint for others to call in
3. Local operations handled within process
4. Access control at unit of volume (VSA) not file
5. Each process is a network file server to other processes
    a. Can run separate FS server when app not running
    b. Multiple apps can access VSA if all read-only

d. 

e. QUESTION: is this reasonable?
    i. What if it doesn't respond?
        1. Time out, or use async calls to avoid blocking
    ii. Is it o.k. that it can read/write all the data?
        1. Permissions enforced at volume level
        2.

iii. Data plan interface
1. Network:
    a. User-level network stack (e.g., tcp, udp) as a library
    b. Send/receive packets using DMA descriptors
        i. Send packet over queue (scatter/gather)
    c. Notification: signal file descriptor (can be polled, selected)
2. Disk:
    a. Read/write blocks at logical addresses to VSAs – virtual storage areas (one or more)
    b. Persistent data structures to leverage low-latency storage (CALADAN)
        i. Log, queue
            1. Log: APIs to atomically create entries, append, read log

            2. Queue: push head, pop tail
        ii. Manage all metadata, high performance
        iii. Simpler than complete files, all user-mode
        iv. Handles correctness: flushing data at appropriate time
        v. No serialization – no pointers in data structures

b. Compared to ExoKernel:
    i. Not remove abstractions completely, but remove dataplane
        1. What called most often
        2. Dominant impact on performance
    ii. Leverage HW support for virtualization
        1. To much of what exokernel wants in HW
    iii. Only address I/O
        1. Not touch memory management, scheduling
    iv. Still have abstractions, but all in library
        1. Low-level matches HW: request queues
        2. Can use native interface (arrakis/p) to bypass (like ExoKernel vision)
c. Other uses of virtualization hardware
    i. Dune: allow page table manipulation in SW
d. Evaluation:
    i. Microbenchmark: show low-level operations relative to linux
    ii. Applications:
        1. show low-level perf benefits applications
        2. Show compatibilty with applications
    iii. Memcached + UDP
        1. Note: not use TCP (complicated protocol)
        2. Faster packet/send receive
    iv. Load balancer:
        1. Lots of connection setup/ teardown
        2. Avoid system calls because all use same filter
    v. Perf isolation:
        1. Job of kernel is to fairly share resources
        2. Show that HW can do the same thing itself
            a. Only show VNIC – easy – not VSA – hard!