CS 134
Operating Systems

April 8, 2019

Operating System Organization

What should a kernel do?

- What kind of system calls should it support?
- What abstractions should it provide?
- Depends on the application and on programmer taste
  - No single best answer
  - There exists lots of ideas, opinions and debates
    - We'll see some in later papers this course
  - This lecture is more about ideas and less about specific mechanisms

Traditional approach

- Big abstractions, and
- Monolithic kernel implementation

Unix, Linux, xv6, VMS

Traditional treatment of CPU

- Kernel gives each process its own virtual CPU—not shared
- Implications
  - Interrupts must save/restore *all* registers for transparency
  - Timer interrupts force transparent context switches
- Maybe good:
  - Simple model. Many irritating details abstracted away
- Maybe bad:
  - Much is hidden (for example, scheduling). May be slow

Clever VM tricks played by traditional kernels

- Lazy page table fill—fast startup for big allocations
- Copy-on-write fork (like Lab 4 but hidden in the kernel)
- Demand paging:
  - Process bigger than available memory?
  - *Page-out* (writes) pages to disk, marks PTEs invalid
  - If process tries to use one of those pages, MMU causes page fault
    - kernel finds phys mem, *pages-in* from disk, marks PTE valid
    - Then returns to process—transparent
- Shared physical memory for executables and libraries

Philosophy of traditional kernels: **abstraction**

- Portable interfaces
  - Files, not disk controller registers
  - Address spaces, not MMU access
- Simple interfaces, hidden complexity
  - All I/O via FDs and read/write, not specialized for each device
  - Address spaces with transparent disk paging
- Abstractions help the kernel manage resources
  - Process abstraction lets kernel be in charge of scheduling
  - File/directory abstraction lets kernel be in charge of disk layout

Philosophy of traditional kernels: **abstraction**

- Abstractions help the kernel enforce security
  - File permissions
  - Processes with private address spaces
- Lots of indirection (Fundamental Theorem of Software Engineering!)
  - E.g., FDs, virtual addresses, filenames, PIDs
  - Helps kernel virtualize, hide, revoke, schedule, etc.

Traditional kernels are *monolithic*

- Kernel is one big program, like xv6
- Easy for subsystems to cooperate: no irritating boundaries
  - For example, integrated paging and file system cache
- All code runs with high privileges—no internal security restrictions

What’s wrong with traditional kernels?

- Big→complex, buggy, and unreliable (in principle—not so much in practice)
- Abstractions may be over-general (and thus slow)
  - Maybe I don’t need all my registers saved on every context switch
- Abstractions are sometimes not quite right
  - Maybe I want to wait for a process that’s not my child
- Abstractions can hinder app-level optimizations
  - Database may be better at laying out B-tree files on disk than kernel FS

Microkernels–an alternate approach

- Big idea: move most OS functionality to user-space service processes
- Kernel can be small: mostly IPC
- The hope:
  - Kernel can be fast and reliable
  - Services are easier to replace and customize
- Examples: Mach 3.0, L4
- JOS is a mix of microkernel and exokernel

Microkernel wins

- You really can make IPC fast
- Separate services force kernel developers to think about modularity
- Good IPC is great for new user-level services (e.g., X server)

Microkernel losses

- Kernel can’t be tiny—needs to know about memory and processes
- You may need lots of IPC—slow in aggregate
- It’s hard to split the kernel into lots of service processes
  - And, it makes cross-service optimization harder
  - So, server processes tend to be huge, not a big win

Microkernels have seen some success

- IPC/service idea widely used—e.g., OSX
  - But not much for traditional kernel services
  - Most for (lots of) new services, designed to be client/server
- Some embedded OSes have strong microkernel flavor

Exokernel paper (1995)

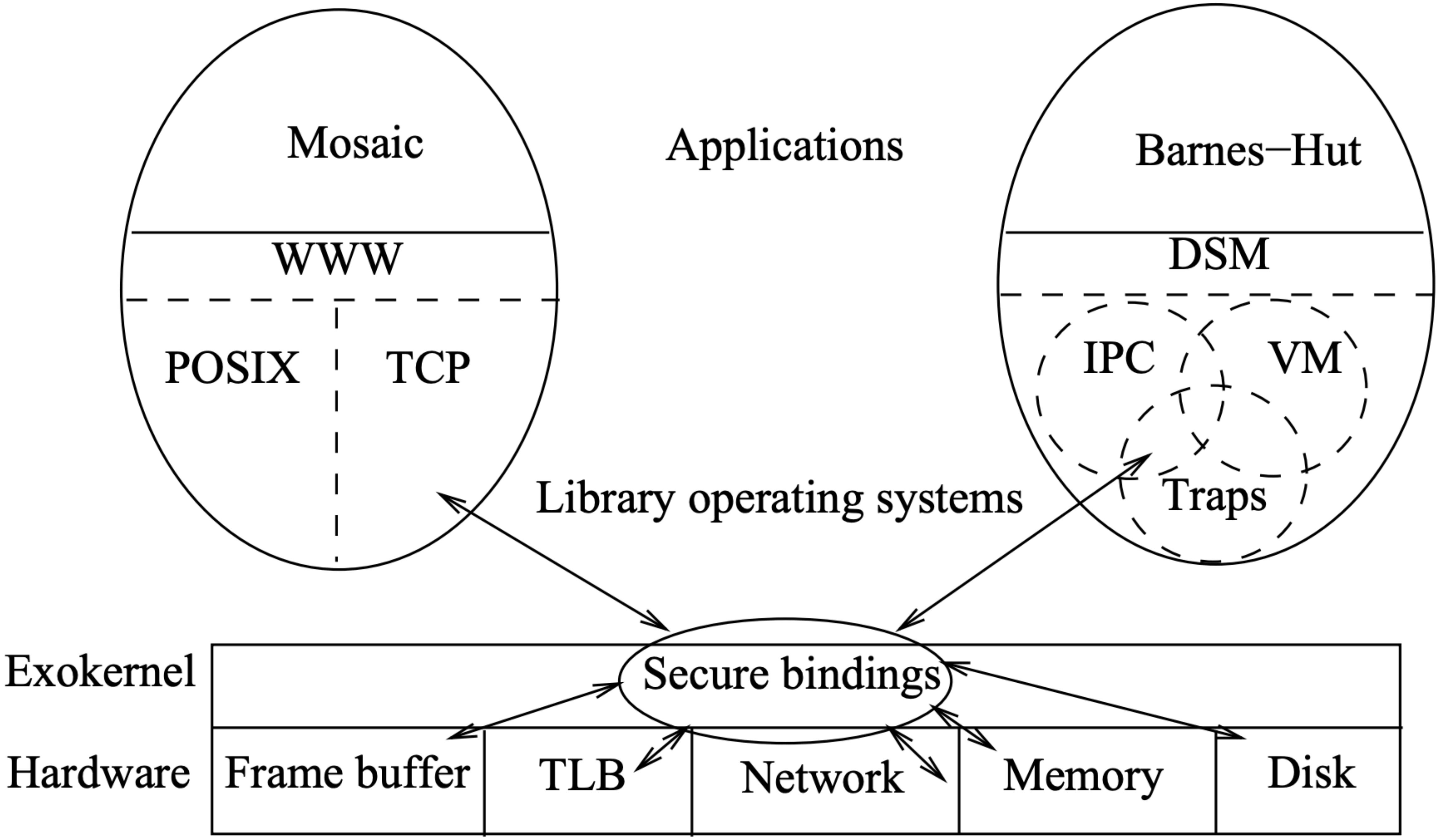
- OS community paid lots of attention
- Full of interesting ideas
- Describes an early research prototype
- Later SOSP (Symposium on Operating System Principles conference) 1997 paper realizes more of the vision

Principal goal of an exokernel: give applications control

Exokernel overview

- Philosophy: eliminate all abstractions
  - Expose HW—let application do with it what it wants
- An exokernel would not provide address space, pipes, file system, TCP
  - Instead, let apps use MMU, phys mem, NIC (Network Interface Controller), timer interrupts
  - Not portable, but lots of application control
- Per-app libOS implements abstractions
  - Perhaps POSIX address spaces, fork, file system, TCP, etc.
  - Each app can have its own custom libOS and its own abstractions
- Why?
  - Kernel may be faster due to streamlining, simplicity
  - Apps may be faster—can customize libOS

Exokernel diagram



Exokernel challenges

- What resources to expose to libOSes
  - What kernel API needed to implement copy-on-write fork at user level?
- Can libOSes share? securely?
  - E.g., compiler reading editor's files
  - Can we have sharing+security without big kernel abstractions?
- Will enough apps benefit from custom libOSes

Exokernel memory interface

- What are the resources?
  - Kernel exposes physical pages and VA→PA MMU mappings
- What's the app→kernel API?
  - pa = AllocPage()
  - TLBwr(va, pa, perms)
  - Grant(env, pa, perms)
  - DeallocPage(pa)
- and, these kernel→app upcalls:
  - PageFault(va, info)
  - PleaseReleaseMemory(amount)

Exokernel memory interface

- What does exokernel need to do?
  - Track what env owns what phys pages
  - Ensure only creates mappings to phys pages it owns
  - Decide which app to ask to give up a phys page when memory runs out
    - That app gets to decide which phys page(s) get given up

Typical use of VM calls

- Application wants memory for a 100MB sparse array, lazily allocated
  - Similar to mmap homework
- `PageFault(va)`

```
if va in range:
  if va in table:
    TLBWr(va, table(va), RW)
  else:
     pa = AllocPage()
     table[va] = pa
     TLBWr(va, pa, RW)
 jump to faulting PC
```

Nice use of exokernel-style memory

- Databases like to keep a cache of disk pages in memory
- Problem on traditional OS:
  - Assume an OS with demand paging to/from disk
  - If DB caches some data and OS needs a phys page, it may page-out a DB page holding cached disk block
  - Waste of time: if DB knew, it'd not write the page (could always read it back from DB file later)
- Exokernel needs a page for another app
  - Sends DB PleaseReleaseMemory() upcall
  - DB picks a clean page, p, calls DeallocPage()
  - Or, DB picks dirty page, saves to DB file, and then calls DeallocPage()

Exokernel CPU interface

- **Not** transparent process switching. Instead:
  - Kernel upcall to app when it gives CPU to app
  - Kernel upcall to app when it wants the CPU back
    - Upcalls to fixed app locations: not transparent)
- If app is running and kernel timer interrupts at end of slice:
  - CPU interrupts from app into kernel (timer)
  - Kernel jumps back into app at “please yield” upcall
  - App saves registers
  - App calls Yield()
- When kernel resumes the app:
  - Kernel jumps into app at “resume” upcall
  - App restores saved registers
- Exorkernel doesn’t save/restore user registers (except PC)—fast syscall/trap/contextswitch

Nice use of exokernel-style CPU

- Suppose timeslice occurs in the middle of:
  - acquire(lock);
    …
    release(lock);
  - You don’t want the app to hold the lock despite not running
    - Then, maybe other apps can’t make forward progress
  - So, the “please yield” upcall can complete the critical section before yielding

Fast IPC

- IPC on traditional kernel:
  - Pipes (or sockets)
  - Message/communication abstraction
  - Slow:
    - write+read + read+write—8 crossings
    - Two blocking calls (reads)
- IPC on Aegis kernel:
  - Yield() can take a process argument
    - Kernel up-calls into target
    - Almost a direct jump to an instruction in target
    - Only at approved locations
  - Kernel leaves registers alone (args + return value)
  - Target uses Yield to return
  - Fast: only 4 crossings

Summary of low-level performance ideas

- Mostly about fast system calls, traps, and upcalls
  - System call speed can be very important
  - Slowness encourages complex system calls, discourages frequent calls
- Trap path doesn’t save most registers
- Fast upcalls to user space (no need for kernel to restore registers)
- Protected call for IPC (just jump to known address; no pipe or send/recv)
- Map some kernel structures into user space (e.g., page table)

Bigger ideas—mostly about abstractions

- Custom abstractions are a win for performance
  - apps need low-level operations for this to work
- Much of kernel can be implemented at user-level
  - While preserving sharing and security
  - Very surprising
- Protection does not require kernel to implement big abstractions
  - E.g., can protect process pages without kernel managing address spaces
- Address space abstraction can be decomposed
  - Into phys page allocation and va→pa mappings

Lasting influence from exokernels

- Unix gives much more low-level control than it did in 1995
  - Very important for some applications
- People think a lot about kernel extensibility now
  - Kernel modules
- Library operating systems are often used
  - For example: unikernels

Questions

- Any work on making portable exokernel interfaces?
- If any application can schedule processes or mess with VM, how does exokernel ensure isolation and security?
- Unclear how multiplexing and packet filters work
- By allowing apps to manage VM, etc, can't that cause potentially high workloads on the kernel, slowing down OS performance?
- Unclear about dynamic packet filters? Need to be from trusted source?

Questions (*cont.*)

- Exokernel gives applications more authority and responsibility. Are there disadvantages and loopholes where malicious apps can do harm to the kernel?
- What is the difference between bind-time and access-time authorization?
- What is a microkernel?
- What is an end-to-end argument?
- What is an example of a high-cost general purpose memory primitives that are expensive compared to a GC implemented in an exokernel-like fashion?
- What about resource revocation and abort?

Questions (*cont.*)

- Are there cases where an exokernel would not be preferred?
- What things are there a user program can do on top of a monolithic kernel that isn’t possible on top of an exokernel+libOS?
- “One possible abort protocol is to simply kill any libOS+app that fails to respond quickly to revocation requests”. However, they decided not do to that because “programmers have a great difficulty reasoning about hard real-time bounds”. Why is this different from other misbehaviors where killing the process seems the right thing?