OS Extensibility

15-712

OS Organization (reminder)

- Many ways to structure an OS. How to decide?
- What *must* an OS do? (consider desktop/server)
  - Let apps use machine resources
    - (Provide convenient abstractions; hide pain)
  - Multiplex resources among apps
  - Prevent starvation
  - Provide isolation and protection, but still
  - Allow cooperation and interaction

Parts of these lecture notes taken from Robert Morris's 6.824 course notes

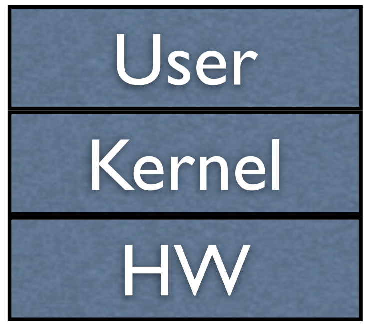
Traditional Approach

- Virtualize some resources: CPU and memory
  - Give each app virtual CPU and memory
  - Simple model for programmers! No need to worry about TLBs, limited physical mem, memory layout, etc.
- Abstract other resources
  - Storage, network, IPC
  - Layer a shareable abstraction over h/w
    - Filesystems and files
    - TCP/IP

Ex: Virt. CPU

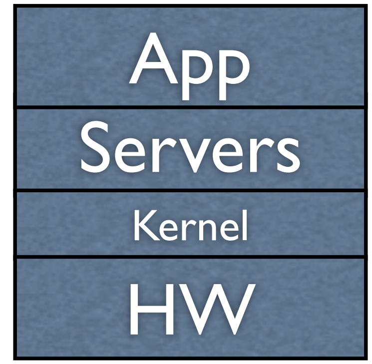
- Goal: Simulate dedicated CPU per process
  - Processes don’t need to worry about sharing
- O/S runs each process in turn via clock interrupt
  - Clock -> processes don’t have to yield; prevents hogging
- Making it transparent:
  - OS saves & restores process state in process table

Monolithic OS



- Kernel is big program: process control, VM, FS, net
- All of kernel runs with full privilege (easy...)
  - Good: subsystems can cooperate easily (e.g., paging & FS)
    - Just a function call away
    - Direct access to all phys memory & data structs if needed
  - Bad: Complex! Bugs easy, no isolation inside OS

Alternate: Microkernels



- Basic idea: user-space servers talking via IPC
- Servers:
  - VM, FS, TCP/IP, even many device drivers
  - Kernel provides *just* the basics:
    - Fast IPC, most basic mem access, interrupts, etc.
    - Gives servers semi-priv. access to some HW
- Apps talk to servers via IPC/RPC
- Good: simple/fast kernel, subsystem isolation, enforces better modularity
- Bad: cross subsystem performance opt harder; using many, many IPCs expensive despite years of tuning
- Ideas really good but whole package didn't catch on

Exokernels and SPIN

- Running "stuff" in the (real) kernel is handy
  - Obvious goal: performance
  - Less obvious goals: Making new things possible/easier
- Two very different approaches...

Exokernel Philosophy

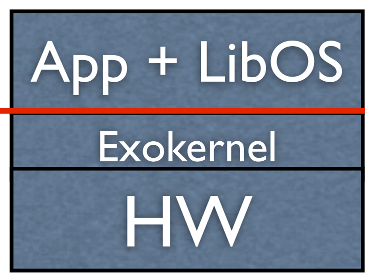
- Eliminate all abstractions!
- For any problem, expose h/w or info to app
  - Let app do what it wants

- Exokernel doesn't provide address space, virtual CPU, FS, TCP, etc.
  - Gives raw pages, page mappings, interrupts, disk i/o, net i/o directly to app
  - Let app build nice address space if it wants - or not!
- Should give aggressive apps great flexibility
- Deliberately strong position (inflammatory)...

Exo-Challenges

- How to multiplex cpu/mem/etc. if you expose them directly to app?
- How to prevent hogs of above?
- How to provide isolation / security despite giving apps low-level control?
- How to multiplex resources w/out understanding them? e.g. contents of disk, formats of pkts

Exo-Architecture



App stuff
Resource management
Filesystem layout,
network protocols, etc.

Protection
low level allocation
physical names
revocation requests
exposes h/w information

Ex: Exokernel memory

- First, kernel provides a few "guaranteed mappings" from virt -> phys
  - App virtual address space has two segments
  - First holds normal app code & data
  - virt addrs in second segment can be "pinned"
  - These addrs hold exception handling code & page tables
- On TLB miss
  - If virt addr in 2nd seg & pinned, kernel installs TLB entry
  - Otherwise, kernel dispatches to app

mem, contd.

- App checks VA in its page table and then calls into kernel to setup TLB entry & capability
- Kernel verifies that capability = the access rights requested by the application. Installs TLB entry.

- Result:
  - App gets total control over its virt->phys mappings
  - But doesn't need to deal with _real_ pain of TLB mgmt
  - Safe, b/c kernel only lets app point to its own phys memory addrs (separate mgmt and protection)

mem interface

- App gets to ask of kernel:
  - pa = AllocPage()
  - DeallocPage(pa)
  - TLBwr(va, pa)
  - TLBvadelete(va)

- Kernel asks of app:
  - PageFault(va)
  - PleaseReleaseAPage()

- Point: App interface to kernel looks like (but not exactly) kernel -> hw. App gets lots of control.

Example

- Why useful? Consider database page caching
- On traditional OS:
  - If OS needs phys page, may transparently write that page to disk.
  - But that's a waste! The DB knows that page is just a cache - better to release than to unnecessarily write. Data is *already* present on disk...
- Exokernel:
  - Kernel says "Please free something up!"
  - App can examine its cache to toss those out
  - If that fails, can write data to disk on its own

Other protection

- LibOS must be able to protect its own "apps" from each other
  - e.g., a UNIX LibOS.
  - Memory controlled by hieararchically-named capabilities
    - Allows delegation of control to children
  - Wakeup predicates
    - Download tiny code into kernel to specify when it should wake up app
  - Network sharing
    - Download tiny code to specify packet dispatching
    - Unlike SPIN, "tiny language" - domain specific and small
  - Critical sections by turning off interrupts

Cheetah on XOK

- Merged file cache and retransmission pool
  - Zero-copy. Similar benefits could arise from sendfile()
  - IO-Lite @ Rice (Vivek Pai) did something similar - found similar benefits in speed and reduced memory pressure (but did it in a normal kernel w/some app changes)
- Batches I/O ops based on knowledge of app
  - e.g., doesn't ACK the HTTP req. packets immediately
  - Delays and sends ACK w/response instead
- App-specific file layout on disk
  - Groups objects in adjacent disk blocks if those objects appear in same web page (bigger sequential reads)

Cheetah overall

- Vastly faster than NCSA and Harvest
- But so are other web servers!
  - Apache faster than NCSA
  - "Flash" - Vivek pai - user-level web server - 50% faster than Apache...
- The usual question: does this level of perf matter for serving static web loads?
  - Pai argues otherwise in recent NSDI paper ("Connection Conditioning")
  - A $200 computer can saturate a $1,000/month 100Mbit/sec Internet connection.
  - But disk seek avoiding could be critical for some loads
- Exokernel folk made startup, ExoTech. Tried to make uber-fast video-on-demand server appliances. Didn't really take off.

Opinions about Exo?

SPIN

- Alternate approach: download "safe" code into kernel
- Same goal: Adapt OS behavior to app
- Note uses of downloaded code
  - Modern unix: BPF (Berkeley Packet Filter)
    - Download small code to select packets @ low-level network code
  - Exokernel: DPF (Dynamic Packet Filter)
    - Same idea, but code actually compiled dynamically = faster
  - These are "tiny languages" (no loops, etc.)
  - SPIN instead d/l's general modula-3 code

SPIN

- Goals:
  - Ensure trustworthy system w/untrusted code
  - High performance
  - Maximize flexibility (let user override as many kernel funcs as possible)
- Approach:
  - Download code into kernel
  - Split kernel into many small components
  - Allow apps to register handlers for those components to override behavior

SPIN challenges

- Safety - code can’t crash, loop forever, etc.
- Isolation - code must apply only to the user or process that downloaded it
- Information leaks - code running in kernel must not be able to access or leak private information
- Granularity: What events to expose?
- Multiplexing: What if multiple apps want to handle an event?
- Performance

Design

- Kernel & extensions in modula 3
  - Certifying compiler digitally signs binaries
  - Language + runtime is typesafe, provides security
- Pointers to kernel objects are indirect capabilities
  - Can't be forged or re-pointed by untrusted app code
- Name-based protection domains
  - Can't extend if you can't name
  - Register proc that authorizes (or denies) linking
- Network - packet filter too...

Design 2

- System designers specify the lowest level set of events
  - e.g., "Console.print"; page fault handler
  - Compare to XOK approach - by default, everything provided in app vs. by design, things can be overridden by app
- Choosing events is hard!
  - Not too fine-grained (overhead, clunky)
  - Not too coarse-grained (insufficient control, forces overriding func to re-implement)

Interfaces

- Raised interface
  - Requests a service
    - e.g.,“allocate a page”
- Handled interface
  - Obj makes demands of clients
  - e.g.,“reclaimPage”
- (Note similarity to XOK memory interface)

Evaluation

- For both of these systems -
- What do you evaluate?
  - What is a metric for “flexibility”?
  - Easy to focus on performance...
  - Is there new functionality these approaches enable?
  - Sometimes speed = “new functionality” by making new things practical, not just possible
- What do you compare against?
- Micro or macro benchmarks?