DISCO and Virtualization

1. Announcements:
    a. Project now due Friday at 9 pm
    b. Class moving to CS 1325 starting Thursday.
2. Questions from reviews:
    a. NFS scalability bottleneck?
        i. Yes, other things exist but it was easy
    b. Save TLB entries on context switch?
        i. TLB is not readable… Expensive!
    c. Is memory overhead worth CPU reduction on in scalabilty tests?
    d. Why is kseg not useful?
    e. Is any data shared besides monitor kernel and host oS?
        i. Yes: CoW from DMA
    f. Vs Exokernel: implement a software driver for resources, or software copy
        i. "virtualize" a resource – may not use physical resource (e.g. interrupt disabling)
        ii. Exokernel: strive instead to not virtualize, but present exactly hardware but safely share it
        iii. Difference is in sematics, but actual result very similar in most cases
            1. One difference: CPU, memory. Not try to do guest context switching or guest page reclamation
    g. Scheduling?
        i. One-to-one VCPU/PCPU
    h. If do page replacement at VMM, how happen?
        i. Change pmap which use machine address (HPA) for each GPA, invalidate TLB & L2TLB
    i. Irix changes:
        i. Remove things that were hard to or expensive to do in Disco but were isolated in Irix
            1. Kseg0
            2. Privileged instructions replaced with access to memory
            3. Device drivers
        ii. Pass hints to VMM for better performance
    j. Applicable to other architectures?
        i. Uses SW TLB, supervisor mode, etc
        ii. Most architectures have something similar

k. Why Flash?
    i. Why write for unavailable HW?
    ii. Goal was write a new OS for an experimental HW – was ccNUMA
        1. Commodity hardware was not ccNUMA
l. CoW Disks
m. Why flush TLB on every swap?

3. History of virtualization:
    a. Invented around 1967:
        i. IBM users wanted a timesharing OS, it only provided batch OS
        ii. Created CP (control prog fam), which was a VMM, and CMS, a timesharing system to run alongside normal OS
        iii. Big benefits in
            1. OS development (didn't have extra machines, could debug)
            2. Upgrade: move some apps to new OS in a separate VM
    b. 1973 – Popek and Goldberg really investigate, lay out definition and needed HW support
        i. Requirements:
            1. efficiency: normal instructions execute natively at no slowdown.
            2. Resource control: code in a VM canot affect system resources, e.g access memory it doesn't own
            3. Equivalency property: executes instructions indistguishably from native HW
        ii. Basic idea:
            1. Run OS kernel outside privileged mode
            2. All privileged instructions trap the VMM
            3. VMM emulates privileged instructions against a software copy of HW state
        iii. HW support:
            1. Sensitive instructions whose behavior differs based on mode are not allowed
                a. X86 popf
            2. More about this later
            3. Largely ignored by Intel
4. Big problem 1: NUMA
    a. Memory is different distances from different CPUs
        i. show picture
    b. What OS support needed?

        i. Allocate physical pages on same node where code is running that uses the pages
        ii. Replicate physical pages that are accessed by all nodes AND miss in the cache
        iii. Reduce lock contention
        iv.
5. Big problem: OS extensibility hard. Particularly for cross-cutting concerns like scalability
    a. QUESTION: Is this still true?
        i. Harder when HW and SW are different: Dell, Microsoft
    b. Approach: solve problem in a layer below, expose virtual standard HW to OS
        i. Contrast to Exokernel: expose HW, solve in the OS
        ii. Different assumptions: what can be changed and what cannot
6. Question: what kinds of things can you do from below?
    a. Hard to do anything app specific
        i. page replacement
        ii. scheduling policy
    b. Easier to do HW-specific
        i. NUMA memory management
        ii. Optimize communication and I/O
7. Question: why bother buying a big machine and running multiple OS?
    a. High-speed communication can be faster than a cluster
    b. Simpler HW administration
    c. Simpler SW administration – all run same disk image
8. Overall approach: virtual machine monitor
    a. Goal: Emulate complete HW interface in software
        i. OS runs on SW copy, manipulates SW copy of HW structures
            1. privileged registers
            2. TLB
            3. I/O devices
        ii. QUESTION: Why?
            1. Minimizes changes to OS
    b. Regaining scalabilty benfits of single OS
        i. QUESTION: Where come from?
            1. Less memory used due to single copy of OS data
            2. Faster communication through memory instead of network
            3. Bigger FS cache since single copy of data

        ii. QUESTION: How do in VMM?
            1. Convert DMA to explicit page sharing
                a. All data comes from disk into memory, then shared by whoever reads from disk
            2. Use COW extensively
                a. share all unmodified copies of data
            3. Build DMA-based network device
                a. Effectively send pointers to data instead of data
                b. Allows sharing across VMs from NFS server
            4. NOTE: Not running more VCPUs that PCPUs
        iii. QUESTION: What are the alternatives?
            1. Implement a layer within the OS

9. Virtualization Types
    a. Type 1: bare metal.
        i. Hypervisor provides all functionality – I/O, scheduling, virtual memory
            1. Xen
            2. VMware Server
    b. Type 2: hosted
        i. Host OS treats it as process, runs with native processes
        ii. Used for providing special OS for some apps but not all
        iii. Used to re-use host OS functionality and avoid re-developing
            1. KVM
            2. Microsoft Hyper-V
            3. VMware workstation, VirtualBox
    c. Management interfaces
        i. Type 2: in host OS
        ii. Type 1:
            1. directly to hypervisor with hypervisor processes
            2. Via management domain with special privileges
10. Virtualizing the CPU
    a. Complete compatibility approach
        i. “Trap and emulate”
            1. Run OS not in privileged mode
            2. All priv inst cause trap to VMM
            3. Use memory protection to separate user from kernel

4. NOTE: REQUIRES ARCHITECTURE TO BE PURE
    a. all operations that behave differently in user/kernel mode must trap in kernel mode
    b. Not true for X86

ii. Emulate complete HW – Virtual PC
  1. device registers for I/O
  2. all privileged**-mode** ops

iii. x86/MIPS approach: ring compression
  1. Run kernel in ring 1 (supervisor mode), hypervisor in ring 0
    a. ring 1 is not privileged but has separate memory access than ring 0
      i. full access to higher levels, but not to lower levels
    b. Ring 0 is privileged
    c. user code still runs in ring 3
    d. Memory setup allows ring 1 to access ring 1, ring 3 addresses, and ring 0 to access everything

iv. Alpha approach:
  1. Run kernel in user mode, apps in user mode
  2. Change page-table permissions when enter privileged mode to allow translations of kernel-mode addresses
    a. shoot down TLB when leaving kernel

v. Interrupts / traps
  1. Decide if caused by guest OS – e.g. divide by zero
    a. If so, emulate virtual interrupt in guest
    b. Else handle in hypervisor

b. Problems

i. Some ops cannot be done this way
  1. Direct-mapped KSEG0: not accessible outside privileged mode, at a fixed virtual address
    a. Not virtualized by hardware – bypasses TLB (good for TLB handlers!) so these virtual addresses can **never** be translated.
    b. Problem: too expensive to trap and emulate every memory access
  2. x86: ops that don't trap but have different behavior

a. popf pops flags, can disable interrupts in kernel mode but not in usermode

3. SOLUTION:
    a. **Paravirtualization**: modify code to be smarter
        i. done by Disco – device drivers,
        ii. Calls VMM directly for ops instead of trap
        iii. OR,Expose special memory region as priv hardware
            1. e.g. interrupt disable
            2. VMM looks at value when emulating hardware
    b. **Instruction rewriting:** replace code sequence with one that does the right thing
        i. VMware: modify popf to trap to VMM
    c. **HW support**: Intel VT extensions
        i. Make instructions trap
        ii. Add virtual hardware to track two copies (VMM and guest OS copy)

11. Virtualizing Memory
    a. 2 level translation
        i. guest VA -> Guest PA (or VA -> PA)
        ii. Guest PA -> Host PA (or PA -> MA)
    b. How?
        i. Implement large SW TLB – a "Shadow Page Table" that translates GVA -> HPA Directly
            1. only contains subset of translations
            2. Must be switched on guest context switch
        ii. Efficient TLB management
            1. MIPS has sw-filled TLB with instructions to write to the TLB
                a. On fill:
                    i. Guest OS:
                        1. priv writeTLB (GVA,GPA)
                        2. traps
                    ii. VMM: lookup GPA -> GPA locally
                        1. install GVA -> HPA into TLB
                        2. Install GVA -> HPA into SW TLB
                    iii. On miss:
                        1. VMM: check SW TLB

            a. cache of recent SW TLB fills
            b. On miss: invoke guest OS TLB miss
        2. Guest OS:
            a. see above to fill TLB
    b. Effect: move most misses from guest OS to VMM due to SW TLB

iii. QUESTION: What about a HW page table?
    1. HW reads page table structure directly
    2. Answer:
        a. Treat modification to page table like SW TLB fill
            i. Need to write-protect guest page table to detect changes
            ii. need to trap on guest context switches
        b. Store SW TLB as HW page table
        c. Store GVA->HPA mapping as HW page table

iv. HW SUPPORT: Intel VT
    1. Provide 2 page tables in HW, do the whole thing.
    2. Each access to entry in guest page table leads to full translation in nested page table
    3. Example?

v. Performance?
    1. Shadow page table cost: tracking guest page table, trapping on context switch
    2. Nested page table cost: 2-d lookup

vi. **DATA STRUCTURES**
    1. What do you need?
        a. Find who is using a physical page for CoW, reclamation, migration
        b. Find where a page is physically (what node)
    2. Mem_map: map of all machine (HPA) pages, what VM is using them. Knows physical node of memory. Maps HPA -> VM/pmap entry
    3. Pmap:
        a. Mostly maps GPA to HPA (as part of TLB entry), but also GPA backwards to GPA

            b. Has TLB entry pre-created to insert, virtual address backmap (for invalidation, etc.)
        4. L2TLB: a hash table of GVA->HPA translations to make TLB misses fast
    vii. **TLBS**:
        1. MIPS supports ASID to avoid TLB flush on OS context switch
        2. Disco does a full TLB flush on VM context swith –**WHY**?
            a. Otherwise has to virtualize ASIDs and remap
            b. Back to: virtualize things that are shared, protect things that are not; in this case, just protect via flush rather than virtualize.

c. NUMA memory management **CAN SKIP**

- i. QUESTION: What is NUMA?
    1. Processors attached to local memories that are faster (2-3x) than memory attached to other processors
- ii. QUESTION: What do you want?
    1. Code on a CPU should try to only access data locally
- iii. QUESTION: What is hard about this?
    1. Shared structures: where do you put them? Every place is remote to someone
    2. Sharing patterns: may have pipeline that moves data between nodes
    3. Thread migration: code may move between nodes
- iv. QUESTION: What is a good overall strategy?
    1. Replication: make copies of widely accessed read-only data
    2. Migration: relocate pages to the CPU that accesses it the most
    3. QUESTION: Why hard to do in OS?
        a. OS data structures not in virtual memory (really), so hard to apply to OS itself without lots of coding
        b. E.g. process list

d.

12. Virtualizing I/O

a. Complex/expensive to do I/O

        i. Implement complete device interface – each I/O write/read to a device register
        ii. Benefit: runs existing drivers, no need to port OS
    b. Paravirtualization approach
        i. Put in **hypercalls** (monitor calls, system calls to hypervisor) to virtual devices with optimized interface
            1. just send a packet, read a disk block
            2. No device registers reads/writes
            3. Single VM exit per I/O operation
    c. Example: Network
        i. Use any size packet (no need to break up for reliability)
        ii. Map packet contents directly into other VM
            1. no need to copy data
    d. General I/O approach:
        i. Write a driver that makes hypercalls into VMM
        ii. VMM takes those calls and makes function calls into standard device driver
            1. VMM enforces protection:
                a. translate disk addresses
                b. Filter network packets by IP address / MAC address
                c. Allow access by only one VM at a time
                    i. E.g. mouse/keyboard for foreground VM only
        iii. For non-shared devices
            1. E.g. give a dedicated network card per machine
            2. Only do protection, not virtualize by handling sharing
    e. Example: Disk
        i. Map disk pages CoW into VM
        ii. Global buffer cache for widely shared data
        iii. Allows sharing (Dedup) of blocks read
            1. e.g. multiple VM boot from same disk
        iv. Works over NFS due to CoW network
    f. Shared disk/CoW disk
        i. Can boot all guest OS from same disk image for management purposes
            1. Use CoW to store copy of modified blocks in memory or elsewhere on disk
            2. gives illusion of private disks when really shared

    g. Read-only disk
        i. Can discard CoW copy on reboot to get back to clean state
        ii.
13. Paravirtualization
    a. Some kernel things hard to detect
        i. Idle loop: put in hypercall
        ii. Free page: put in hypercall
    b. BIG HINTING IDEA:
        i. Disco takes a layer approach, suffers from lack of communication across the layer (always the problem)
        ii. Hinting: a way to pass information without violating abstractions
            1. Generally not guaranteed (not change correctness)
    c. Duplication of effort between OS and VMM
        i. Need a zero page: put in hypercall
            1. VMM already zeroes pages
    d. Resource use
        i. Free pages reported to VMM to be reclaimed/shared
        ii. Idle loop hinted to VMM using power management instructions
14. Benefits of Virtualization:
    a. LibOS
        i. Can run LibOS if don't need services – just NFS for data access
            1. Like ExoKernel but with different level of abstraction
15. BIG PICTURE:
    a. VMM is another approach to OS flexibility
        i. Can run multiple OS on a machine
        ii. Can add features with new virtual hardware
        iii. Is a "layering" approach
            1. Need trick to use OS knowledge in the VMM layer, such as zero pages & free pages & idle loop
    b. QUESTION: Would we use VMMs if operating systems were better written?
16. Evaluation
    a. Look at overheads – what does Disco make more expensive
        i. All privileged operations (extra traps)
        ii. TLB misses (more expensive)

iii. Uses more memory – multiple kernel copies, etc.

b. Look at benefit of optimzations
    i. How beneficial is sharing, other optimizations?
        1. Important to know if they actually help
c. Look at actual performance gains (stated goal of NUMA)
    i. Find workloads that depend on scalability, try out
d. Use on Irix:
    i. Boot Irix, then switch to Disco – a world switch
        1. What does this mean?
            a. Exclude Irix memory from the machine memory (HPA) Disco allocates
            b. Change interrupt vectors to point to Disco
            c. Cause a trap to jump into Disco code
        2. Can switch back
            a. Change interrupt vectors back to Irix's
            b. Cause a trap
        3. Used by VMware workstation

17. Comparison to Exokernel
    a. Does not abstract hardware, only does protection/scheduling
    b. Provides scheduler upcalls (e.g. virtual interrupts) to let guest handle threading
    c. Lets guest decide which pages to evict (see VMware paper later)
    d. WHAT IS THE KEY DIFFERENCE:
        i. Optimized for isolation, not sharing
        ii. IPC mechanism designed for I/O
            1. focus on throughput using ring buffers, not low-latency for accessing generic serives
            2.

Notes:

```
void emulate_tlbwrite_instruction (VA, PA, otherdata) {
tlb_insert (thiscpu->l2tlb, VA, PA, otherdata); // cache
if (!defined (thiscpu->pmap[PA])) { // fill in pmap dynamically
  MA = allocate_machine_page ();
  thiscpu->pmap[PA] = MA; // See 4.2.2
  thiscpu->pmapbackmap[MA] = PA;
  thiscpu->memmap[MA] = VA; // See 4.2.3 (for TLB shootdowns)
}
```