Virtual Memory and Virtualization
Virtual Memory

- A layer of abstraction between applications and hardware
  - Programs use “virtual addresses” to access memory locations for code & data
  - OS & architecture translate to “physical addresses” that correspond to locations in physical memory
  - OS (possibly with hardware support) manage which parts of the program’s virtual address space to allocate to physical memory
  - Address translation typically done at page granularity
Why is Virtual Memory Needed? (1/2)

- Memory Capacity
  - A program may need more memory than the main memory capacity on a system
  - Without virtual memory, this would be managed with software “overlays” where programmers or compilers swap out parts of code/data for others
  - With virtual memory, this is managed by hardware & OS

- Program Portability
  - Program compiles using virtual addresses that could be ported to run in any part of physical memory (independent of other running processes)
Why is Virtual Memory Needed? (2/2)

Protection

- Different parts of memory space need different permissions (e.g., read-only, read/write, execute)
- Virtual memory could be used to implement page-level access permissions

Security & Isolation

- A program cannot access another’s memory
- A bug (e.g., bad pointer) in one process may not corrupt memory for another process
Address Translation

- Biggest overhead needed to support virtual memory
- Need to translate virtual address in every load/store instruction to physical address before accessing cache/memory
  - Unless cache is virtually addressed (discussed before)
- Translation done at page level (page offset is the same for virtual and physical addresses)

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Offset</th>
<th>Physical Page Number</th>
<th>Offset</th>
</tr>
</thead>
</table>

- Virtual address size determined by machine (64-bit vs. 32-bit)
- Caches could be virtually indexed if index bits are all included in the page offset bits (discuss pros & cons)
Page Tables

- OS manages address translation through page tables
  - Each Page Table Entry (PTE) includes one VPN → PPN mapping
  - Separate structure per process (each process needs its own mapping)
  - May need a lot of pages to map the whole virtual address space

- Page Table Design Parameters
  - Page Size
  - Number of page sizes supported
  - Number of page table levels
  - Alternative designs, e.g., Inverted Page Tables

- Tradeoff between page size & PT levels
Hardware Support for Virtual Memory

- Without hardware support, huge performance hit
  - Example, in a four-level page table design, each load/store could need five memory accesses to be completed
  - Overhead could be decreased (though not completely avoided) if some PTEs are cached

- Translation Lookaside Buffers (TLBs) needed to avoid performance loss
  - Caches recent/frequent address translations to avoid memory accesses
  - On a TLB miss, a page walk is initiated to fetch translation
  - Tradeoff between TLB size/associativity and latency
  - Larger page sizes increase TLB reach
Virtualization

- Virtual memory is an example of virtualization

- Definition: Separation of resource/service from underlying physical delivery of that service
  (VMWare White Paper: http://www.vmware.com/pdf/virtualization.pdf)

- Why is it needed?
  - Server consolidation
  - Debugging and testing: fault containment
  - Availability
  - Security
Classical Virtualization

A virtual machine monitor (VMM) needs to satisfy three conditions:

- **Fidelity**: Software on VMM executes identically to native execution (except for timing differences)
- **Performance**: Most guest instructions need to be performed directly by the hardware with no VMM intervention
- **Safety**: The VMM manages all hardware resources

Typically implemented using “Trap and Emulate” for privileged instructions, and maintaining shadow structures for guest OS (e.g., page tables)

x86 doesn’t support “trap and emulate”
Software x86 Virtualization

- One solution: Guest executes on interpreter instead of directly on physical CPU
  - High overhead, violates performance requirement

- Binary translation
  - Input is binary x86 code (not source code)
  - Translation happens dynamically at runtime, interleaved with executing generated code
  - Produces safe code (mostly user-mode)
  - On-demand and adaptive

- Example translated code in paper
Software x86 Virtualization (Cont.)

Most instructions can be translated identically, except for:

- PC-relative addressing: translator output could be at a different address than the input
- Direct control flow (branches, calls) need mapping from guest address to translated-code address
- Indirect control flow: computed dynamically, introducing some overhead (typically < 10%)
- Privileged instructions

BT VMM could outperform classical VMM by avoiding expensive privileged instruction traps
Hardware x86 Virtualization

- Virtual Memory Control Block (VMCB) is an in-memory data structure combines control state with a subset of the state of a guest virtual CPU
- New execution mode “guest mode” supports running guest code in a less privileged mode
- `vmrun` instruction transfers from host→guest mode
- `exit` instruction returns to host mode (saving state in VMCB) based on certain conditions
  - Guest page faults, TLB flushes, context switching, I/O instructions, accessing page tables and memory mapped devices trigger `exit`
- Hardware vs. Software: Paper Figure 2, 3, 4, 5
Next Week’s Announcements

- Final exam on Tuesday
  - Open book, notes, calculator
  - No phones, computers, tablets
- Project presentations on Thursday
  - Review Oral Presentation Advice
  - Final project report due June 10
  - Submission instructions posted on webpage