6.033 - Operating Systems: Virtual Machines
Lecture 6
Katrina LaCurts, lacurts@mit.edu

1. Virtual Machines
  - How to run multiple OSes on one machine?
  - Constraint: compatibility. Don't want to change existing kernel code.
  - We'll run multiple virtual machines (VMs) on a single CPU. Kernel equivalent is the "virtual machine monitor" (VMM)
  - Can run VMM as user-mode app inside host OS, or run VMM on hardware in kernel mode with guest OSes in user mode. We'll talk about second, but the issues are the same.
  - Role of VMM also involves allocating resources and dispatching events, but we're focused on dealing with instructions from guest OS that require interaction with the physical hardware
  - Attempt 1: emulate every single instruction
    - Problen: Slow
  - Attempt 2: guest OSes run instructions directly on CPU
    - Problem: dealing with privileged instructions (can't run in kernel mode; then we'd be back to our original problem)
  - VMM will deal with handling privileged instructions

2. VMM Implementation
  - Trap and emulate
    - Guest OS in user mode
    - Privileged instructions cause an exception; VMM intercepts these and emulates
    - If VMM can't emulate, send exception back up to guest OS
  - Problems:
    - How to emulate (what does it mean? does it depend on the instruction?)
    - How to deal with instructions that don't trigger an interrupt but that the VMM still needs to intercept

3. Virtualizing memory
  - VMM needs to translate guest OS addresses into physical memory addresses. Three layers: guest virtual, guest physical, host physical
  - Approach 1: Shadow pages
    - Guest OS loads PTR; causes interrupt. VMM intercepts
    - VMM locates guest OS's page table. Combines guest OS's table with its own table, constructing a third table mapping guest virtual to host physical
    - VMM loads host physical addr of this new page table into the hardware PTR
    - If guest OS modifies its page table, no interrupt thrown. To force an interrupt, VMM marks guest OS's page table as read-only memory
  - Approach 2

    - Modern hardware has support for virtualization
    - Physical hardware (effectively) knows about both levels of tables: will do lookup in the guest OS's page table and then the VMM's page table

4. Virtualizing U/K bit
  - Problem with basic trap-and-emulate: U/K bit involved in some instructions that don't cause exception (e.g., reading U/K bit, writing it to U)
  - Few solutions:
    - Para-virtualization: modify guest OS. Hard to do, and goes against our compatibility goal
    - Binary translation: VMM analyzes code from guest OS and replaces problematic instructions
    - Hardware support: some architectures have virtualization support built in. Have special VMM operating mode in addition to the U/K bit
  - Hardware support is arguably the best. Makes VMM's job easier.

5. Monolithic kernels
  - VMs protect OSes from each other’s faults, protect physical machine
    from OS faults. Why so many bugs, though?
  - The Linux kernel is, effectively, one large C program. Careful software engineering, but very little modularity within the kernel itself.
  - Bugs come about because of its complexity
  - Kernel bugs = entire system failure (recall the in-class demo)
  - Even worse: adversary can exploit these bugs

6. Microkernels: alternative to monolithic kernels
  - Put subsystems -- file servers, device drivers, etc. -- in user programs. More modular.
  - There will still be bugs but:
    - Fewer, because of decreased complexity
    - A single bug is less likely to crash the entire system
  - Why isn't Linux a microkernel, then?
    - High communication cost between modules
    - Not clear that moving programs to userspace is worth it
    - Hard to balance dependencies (e.g., sharing memory across modules)
    - Redesign is tough!
    - Spend a year of developer time rewriting the kernel or adding new features?
    - Microkernels can make it more difficult to change interfaces
  - Some parts of Linux do have microkernel design aspects

7. Performance
  - One reason to prefer monolithic kernels over microkernels is performance: communication costs are high in a microkernel

- We've seen a lot of examples in lecture; let's put things together
- To measure performance, need metrics:
  - Throughput: number of requests over a unit of time
  - Latency: amount of time for a single request
  - Relationship between these changes depending on the context
  - As system becomes heavily-loaded:
    - Latency and throughput start low. Throughput increases as users enter, latency stays flat...
    - ..until system is at maximum throughput. Then throughput plateaus, latency increases
  - For heavily-loaded systems: focus on improving throughput
- Need to compare measured throughput to possible throughput: utilization
- Improving performance: Utilization sometimes makes bottleneck obvious, sometimes not. When bottleneck is not obvious, use measurements to locate candidates for bottlenecks, fix them, see what happens (iterate)

8. Improving performance ("relaxing" the bottleneck)
   - Better algorithms, etc. These are application-specific. 6.033 focuses on generally-applicable techniques
   - Batching, caching, concurrency, scheduling
   - You've seen some examples of most of these in OSes, but they apply everywhere in systems (and esp. to your design project)
   - Understanding where these types of techniques can apply requires an understanding of how your system works and how it is used (e.g., can't just put a cache anywhere and magically see improved load times)