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

0. Intro
  - We've enforced modularity on a single machine. Saw virtual memory, system calls, bounded buffers, threads, etc.
  - New question: Can we rely on the kernel itself to work properly?

1. Monolithic kernels
  - 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

2. 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

3. Virtual Machines
  - New problem: how to deal with kernel bugs without redesigning kernel from scratch?
  - Solution: run multiple instances of Linux on a single machine
    - Gives us fault isolation, customized OS for each program
  - 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:
    - Allocate resources
    - Dispatch events
    - Deal 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

4. 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 do the emulate
    - How to deal with instructions that don't trigger an interrupt but that the VMM still needs to intercept

5. 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 mark's guest OS's page table as read-only memory
  - Approach 2
    - Modern harware 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

6. 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.

7. Virtualizing disk:
  - Guest OS accesses disk by issuing special instructions. These are only accessible in K/VMM mode and raise an exception. VMM, again, traps and emulate.

8. Summary
  - Other cool things we do with VMs: run different OSes on a single machine, move VMs from one physical machine to another
  - Microkernels and VMs solve orthogonal problems
    - Microkernels: split up monolithic designs
    - VMs: let us run many instances of an existing OS. They are, in some sense, a partial solution to monolithic kernels (at least we can run these kernels safely). But their goal is to run multiple OSes on a single piece of hardware, not to target monolithic OSes specifically.
  - VMs most commonly implemented with hardware support (a special VMM mode in addition to U/K bit)