Lecture 4: Extensibility
(and finishing virtual machines)

CSC 469H1F
Fall 2006
Angela Demke Brown
Announcements

• First assignment out tomorrow
  • Today’s tutorial looks at some of the tools you will need to use
  • Due 2 weeks after it goes out

• IBM Technical & Business Info Session
  • TODAY
  • 5:30 – 7:30 p.m.
  • BA 1160
What is a virtual machine?

- An efficient, isolated duplicate of the real machine
  - Popek & Goldberg, 1974 “Formal Requirements for Virtualizable Third Generation Architectures”
  - Provide by “virtual machine monitor” with three essential characteristics:
    - Essentially identical execution environment (as real machine)
    - Minor performance penalty for programs in VM
    - VMM has complete control over system resources
- Software added to the execution platform to give the appearance of a different platform or multiple platforms
  - Smith & Nair, 2004 “Virtual Machines”
Why virtual machines?

- Originally motivation in 1960’s
  - Large, expensive computers shared by many users
  - Different groups wanted or needed different operating systems
  - Convenient timesharing mechanism (each user gets own virtual machine)

- Today’s motivation
  - Large scale servers have similar issues as original motivation
  - Portability/compatibility
    - Avoid dealing with multiprocessor issues in OS
  - Security
  - Reliability/fault tolerance
  - Migration
  - Performance
  - Innovation
Types of virtual machines

- Many uses of the term “virtual machine”
- Conventional software is developed/compiled for a specific OS and instruction set architecture (ISA)
  - Together, these are the application binary interface (ABI)
  - Can distinguish virtual machines depending on whether they virtualize the ABI or the ISA.
- Process virtual machines provide virtual ABI
  - Created and destroyed along with the process they run
- System virtual machines provide a complete system environment
  - Multiple user processes, file system, I/O, GUI, etc.
Process Virtual Machines

- **Multiprogramming**
  - Each conventional process has illusion of own machine
    - Address space, CPU, file table, etc

- **Emulation / dynamic binary translators**
  - Code compiled for one ISA translated on-the-fly to host ISA
    - E.g. Digital FX!32 runs x86 Windows binaries on Alpha

- **Dynamic optimizers**
  - Same guest/host ISA, only purpose is optimization

- **High-level language VMs**
  - Designed together with language
  - Mainly for portability & to support language features
    - E.g. Pascal P-code, Java bytecode
System VMs

- "classic" VMM
  - VMM runs on bare hardware, everything else runs on top
  - VMM is most privileged software, everything else less

- "hosted" VM
  - Virtualizing software installed on top of existing OS
    - E.g. VMWare Workstation
Requirements for Virtualizability

• **Architecture requirements**
  - Dual mode operation
  - A way to call privileged operations from non-privileged mode
  - Memory relocation / protection hardware
  - Asynchronous interrupts for I/O to communicate with CPU
    • Goldberg, 1972

• **Generic VM operation / implementation**
  - Dispatcher component
  - Allocator
  - Interpreter
Instruction Requirements

- Privileged instructions: required to trap if not executed in supervisor mode
- Sensitive instructions: affect the operation of the system in some way

- **THEOREM:** An efficient VMM may be constructed if the set of sensitive instructions is a subset of the set of privileged instructions
- Intel Pentium: 17 instructions are sensitive but not privileged (Robin & Irvine, USENIX Security 2000)
  - VMware does binary rewriting to deal with this
  - Xen requires changes to the OS
OS Extensions

- Adding new function to OS “on the fly”
- Why?
  - Fixing mistakes
  - Supporting new features or hardware
  - Efficiency / Custom implementations
- How?
  - Give everyone their own machine (VMs)
  - Allow users to modify the OS (modules)
  - Allow some OS function to run outside (ukernel)
Loadable Kernel Modules

- Giving everyone a virtual machine doesn’t entirely solve the extension problem
  - You can run what you want on your VM, but do you really want to write a custom OS?
- Often just want to modify/replace small part
- Solution: Allow parts of the kernel to be dynamically loaded / unloaded
  - Requires dynamic relocation and linking
- Common strategy in monolithic kernels for device drivers (FreeBSD, Windows NT/2K/XP, Linux)
Linux Loadable Kernel Modules

- **Module writer must define (at least) two functions**
  - `init_module` - code executed when module loads
  - `cleanup_module` - code executed when module unloads
  - Module functions can refer to any exported kernel symbols

- **Module is compiled into relocatable .o file**

- **`insmod` command loads module into running kernel**
  - Resolves references to kernel symbols

- **`rmmod` command removes module from kernel**

- **`lsmod` command lists currently-installed modules**
insmod

- User-level command (program) restricted to superuser
- Gets help from some special system calls
  - `sys_create_module` - allocate kernel memory to hold module
  - `get_kernel_syms` - get kernel symbol table to link module (patch symbolic references in .o file to actual kernel addresses)
  - `sys_init_module` - copy relocatable .o file into kernel space
- Then calls `init_module` function
rmmod

- Unlinks module from kernel
- Needs to ensure no one is using module first!
  - Reference count incremented whenever module is used, or a module that depends on this one is loaded
- Removes module symbols from symbol table
- Frees memory
Problems with module approach

- Requires stable interfaces
  - Linux uses version numbers to check if module is compiled for correct version of kernel, but it is easy to get this wrong
- Unsafe
  - Module code can do anything because it runs privileged
    - E.g. recall VMWare Workstation driver?
      - “hijacks” machine by changing interrupt descriptor table (IDT) base register and then jumps to code in the VM application!
Alternate kernel-level schemes

• Trusted compiler (or certification authority) + digital signatures
  • Allows verification of source of code added to kernel
  • You still have to decide if you trust that source
  • Code can still do anything

• Proof-carrying code
  • Consumer (OS) supplies a specification for what extensions are allowed to do
  • Extension must supply a proof that it is safe to execute according to specification
  • OS validates proof
  • Proof should be easy to check, but may be hard to generate (e.g. maze example)
Alternates (2)

- Sandboxing (software fault isolation)
  - Limit memory references to per-module segments
  - Check for certain unsafe instructions
- Examples:
  - SPIN (U. of Washington)
    - Modula-3 + trusted compiler
    - Safety properties provided by language
    - Problems with dynamic behavior (e.g. “while(1)”)
  - Vino (Harvard)
    - Sandboxed C/C++ code called “grafts”
    - Timeouts to guard against misbehaved grafts
    - Resource limits + transactional “undo”