Virtual Machines

“A virtual machine (VM) is an efficient, isolated duplicate of a real machine”
- Duplicate: VM should behave identically to the real machine
  - Programs cannot distinguish between execution on real or virtual hardware
  - Except for:
    - Fewer resources available (and potentially different between executions)
    - Some timing differences (when dealing with devices)
- Isolated: Several VMs execute without interfering with each other
- Efficient: VM should execute at a speed close to that of real hardware
  - Requires that most instruction are executed directly by real hardware

Types of Virtual Machines

- Contemporary use of the term VM is more general
- Call virtual machines even if there is no correspondence to an existing real machine
  - E.g.: Java virtual machine
  - Can be viewed as virtualizing at the ABI level
  - Also called process VM
- We only concern ourselves with virtualizing at the ISA level
  - ISA = instruction-set architecture (hardware-software interface)
  - Also called system VM
  - Will later see subclasses of this

Virtual Machine Monitor (VMM), aka Hypervisor

- Program that runs on real hardware to implement the virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
  - Mediates access to shared resources
    - e.g. console
  - Performs world switch
- Implications:
  - Hypervisor executes in privileged mode
  - Guest software executes in unprivileged mode
  - Privileged instructions in guest cause a trap into hypervisor
  - Hypervisor interprets/emulates them
  - Can have extra instructions for hypercalls

Simulator
- Provides a functionally accurate software model of a machine
  - May run on any hardware
- Is typically slow (order of 1000 slowdown)

Emulator
- Provides a behavioural model of hardware (and possibly S/W)
  - Not fully accurate
  - Reasonably fast (order of 10 slowdown)

Virtual Machine
- Models a machine exactly and efficiently
  - Minimal slowdown
  - Needs to be run on the physical machine it virtualizes (more or less)
Why Virtual Machines?

Historically used for easier sharing of expensive mainframes
- Run several (even different) OSes on same machine
- Each on a subset of physical resources
- Can run single-user single-tasked OS in time-sharing system
  - legacy support
- “world switch” between VMs

Gone out of fashion in 80’s
- Time-sharing OSes common-place
- Hardware too cheap to worry...

Renaissance in recent years for improved isolation
- Server/desktop virtual machines
  - Improved QoS and security
  - Uniform view of hardware
  - Complete encapsulation
    - replication
    - migration
    - checkpointing
    - debugging
- Different concurrent OSes
  - e.g.: Linux and Windows
- Total mediation
  - Would be mostly unnecessary
    - if OSes were doing their job...

Native vs. Hosted VMM

Native/Classic/Bare-metal/Type-I
- Hosted VMM can run besides native apps
  - Sandbox untrusted apps
  - Run second OS
  - Less efficient:
    - Guest privileged instruction traps into OS, forwarded to hypervisor
    - Return to guest requires a native OS system call
  - Convenient for running alternative OS environment on desktop

Hosted/Type-II
- e.g. VMware Player/Fusion

VMM Types

Classical: as above
Hosted: run on top of another operating system
- e.g. VMware Player/Fusion
Whole-system: virtual hardware and operating system
- Really an emulation
  - e.g. Virtual PC (for Macintosh)
Physically partitioned: allocate actual processors to each VM
Logically partitioned: time-share processors between VMs
Co-designed: hardware specifically designed for VMM
  - Co-designed: hardware specifically designed for VMM
    - e.g. Transmeta Crusoe, IBM i-Series
Pseudo: no enforcement of partitioning
- Guests at same privilege level as hypervisor
- Really abuse of term “virtualization”
  - e.g. products with “optional isolation”

Virtualization Mechanics

Traditional “trap and emulate” approach:
- guest attempts to access physical resource
- hardware raises exception (trap), invoking hypervisor’s exception handler
- hypervisor emulates result, based on access to virtual resource

Most instructions do not trap
- makes efficient virtualization possible
- requires that VM ISA is (almost) same as physical processor ISA

Address Translation

Virtualization Mechanics: Address Translation

Traditional “trap and emulate” approach:
- guest attempts to access physical resource
- hardware raises exception (trap), invoking hypervisor’s exception handler
- hypervisor emulates result, based on access to virtual resource

Most instructions do not trap
- makes efficient virtualization possible
- requires that VM ISA is (almost) same as physical processor ISA
Definitions:

- Privileged instruction: executes in privileged mode, traps in user mode
  - Note: trap is required, NO-OP is insufficient
- Privileged state: determines resource allocation
  - Includes privilege mode, addressing context, exception vectors,...
- Sensitive instruction: control-sensitive or behaviour-sensitive
  - Control sensitive: changes privileged state
  - Behaviour sensitive: exposes privileged state
    - Includes instructions which are NO-OPs in user but not privileged mode
- Innocuous instruction: not sensitive

Note:

- Some instructions are inherently sensitive
  - e.g. TLB read
- Others are sensitive in some context
  - e.g. store to page table

Requirements for Virtualization

- Binary translation: modifies binary
- Para-virtualization: changes ISA

Two standard approaches:

- VMM needs to maintain virtualized privileged machine state
  - processor status
  - addressing context
  - device state...
- VMM needs to emulate privileged instructions
  - translate between virtual and real privileged state
  - e.g. guest → real page tables
- Virtualization traps are expensive on modern hardware
  - can be 100s of cycles (x86)
- Some OS operations involve frequent traps
  - STI/ICI for mutual exclusion
  - frequent page table updates during fork()...
- MIPS KSEG address used for physical addressing in kernel

Unvirtualizable Architectures

- x86: lots of unvirtualizable features
  - e.g. sensitive PUSH of PSW is not privileged
  - segment and interrupt descriptor tables in virtual memory
  - segment descriptor exposes privileged level
- Itanium: mostly virtualizable, but
  - interrupt vector table in virtual memory
- THASH instruction exposes hardware page tables address
- MIPS: mostly virtualizable, but
  - kernel registers K0-K7 (needed to save registers state) user-accessible
  - performance issue with virtualizing KSEG addresses
- ARM: mostly virtualizable, but
  - some instructions undefined in user mode (banked registers, CPSR)
  - PC is a GPR, exception return in MOV to PC, doesn't trap
- Most others have problems too
- Recent architecture extensions provide virtualization support hacks

Trap-and-Emulate Requirements

- An architecture is virtualizable if all sensitive instructions are privileged
- Can then achieve accurate, efficient guest execution
  - by simply running guest binary on hypervisor
- VMM controls resources
- Virtualized execution is indistinguishable from native, except:
  - Resources more limited (running on smaller machine)
  - Timing is different (if there is an observable time source)
- Recursively virtualizable machine:
  - VMM can be built without any timing dependence

VMM

Guest

Exception

Unvirtualizable Architectures

Virtualization Overheads

- VMM needs to maintain virtualized privileged machine state
  - processor status
  - addressing context
  - device state...
- VMM needs to emulate privileged instructions
  - translate between virtual and real privileged state
  - e.g. guest → real page tables
- Virtualization traps are expensive on modern hardware
  - can be 100s of cycles (x86)
- Some OS operations involve frequent traps
  - STI/ICI for mutual exclusion
  - frequent page table updates during fork()...
- MIPS KSEG address used for physical addressing in kernel

Impure Virtualization

- Used for two reasons:
  - unvirtualizable architectures
  - performance problems of virtualization
- Change the guest OS, replacing sensitive instructions
  - by trapping code (hypercall)
- by in-line emulation code

Two standard approaches:

- Para-virtualization: changes ISA
- Binary translation: modifies binary

Binary Translation

- Locate sensitive instructions in guest binary and replace on-the-fly by emulation code or hypercall
  - pioneered by VMware
  - can also detect combinations of sensitive instructions and replace by single emulation
  - doesn’t require source, uses unmodified native binary
  - this appears like pure virtualization!
  - very tricky to get right (especially on x80)
  - needs to make some assumptions on same behaviour of guest
Virtualization Techniques

E.g. lazy update of virtual machine state
- virtual state is kept inside hypervisor
- keep copy of virtual state inside VM
- allow temporary inconsistency between local copy and real VM state
- synchronize state on next forced hypervisor invocation
  - actual trap
  - explicit hypervisor call when physical state must be updated

Virtualization Techniques

Page table implementation options
- Strict shadowing of virtual page table
  - write protect PTs: force trap into hypervisor on each update
  - can combine multiple updates in single hypercall
    - e.g. during fork()
- Lazy shadowing of virtual page table
  - identity synchronization points
  - possible due to TLB semantics
    - new PT updates only become effective once loaded into TLB
    - explicit TLB loads and flushes are natural synchronization points
  - PTs are big: need to tell hypervisor which part to sync
- Expose real page tables (write-protected)
  - emulate updates
  - guest must deal with PT reads differing from what was written
- Complex trade-offs
  - Xen changed approach several times

Virtualization Techniques

Over-committing memory
- like classical virtual memory
- sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]

Memory reclamation using ballooning
- load pseudo device driver into guest, colludes with hypervisor
- to reclaim memory, hypervisor instructs driver to request memory
- hypervisor can re-use memory hoarded by ballooning driver
- guest controls which memory it gives up

Virtualization Techniques

Full virtualization of device
- Hypervisor contains real device driver
- Native guest driver accesses device as usual
- implies virtualizing all device registers etc
- trap on every device access

Virtualizes at device interface
- Hypervisor implements device-sharing policies

Drawbacks:
- must re-implement/port all drivers in hypervisor
  - unfeasible for contemporary hardware
- very expensive (frequent traps)
- will not work on most devices
- timing constraints violated

Device Virtualization Techniques

Over-committing memory
- like classical virtual memory
- sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]

Memory reclamation using ballooning
- load pseudo device driver into guest, colludes with hypervisor
- to reclaim memory, hypervisor instructs driver to request memory
- hypervisor can re-use memory hoarded by ballooning driver
- guest controls which memory it gives up

Virtualization Techniques

Virtual memory tricks
- Over-committing memory
  - like classical virtual memory
  - sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]

Memory reclamation using ballooning
- load pseudo device driver into guest, colludes with hypervisor
- to reclaim memory, hypervisor instructs driver to request memory
- hypervisor can re-use memory hoarded by ballooning driver
- guest controls which memory it gives up

Virtualization Techniques

Device Virtualization Techniques

Full virtualization of device
- Hypervisor contains real device driver
- Native guest driver accesses device as usual
- implies virtualizing all device registers etc
- trap on every device access

Virtualizes at device interface
- Hypervisor implements device-sharing policies

Drawbacks:
- must re-implement/port all drivers in hypervisor
  - unfeasible for contemporary hardware
- very expensive (frequent traps)
- will not work on most devices
- timing constraints violated

Device Virtualization Techniques

Over-committing memory
- like classical virtual memory
- sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]

Memory reclamation using ballooning
- load pseudo device driver into guest, colludes with hypervisor
- to reclaim memory, hypervisor instructs driver to request memory
- hypervisor can re-use memory hoarded by ballooning driver
- guest controls which memory it gives up

Virtualization Techniques

Virtual memory tricks
- Over-committing memory
  - like classical virtual memory
  - sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]

Memory reclamation using ballooning
- load pseudo device driver into guest, colludes with hypervisor
- to reclaim memory, hypervisor instructs driver to request memory
- hypervisor can re-use memory hoarded by ballooning driver
- guest controls which memory it gives up

Virtualization Techniques

Device Virtualization Techniques

Full virtualization of device
- Hypervisor contains real device driver
- Native guest driver accesses device as usual
- implies virtualizing all device registers etc
- trap on every device access

Virtualizes at device interface
- Hypervisor implements device-sharing policies

Drawbacks:
- must re-implement/port all drivers in hypervisor
  - unfeasible for contemporary hardware
- very expensive (frequent traps)
- will not work on most devices
- timing constraints violated

Device Virtualization Techniques

Over-committing memory
- like classical virtual memory
- sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]

Memory reclamation using ballooning
- load pseudo device driver into guest, colludes with hypervisor
- to reclaim memory, hypervisor instructs driver to request memory
- hypervisor can re-use memory hoarded by ballooning driver
- guest controls which memory it gives up

Virtualization Techniques

Virtual memory tricks
- Over-committing memory
  - like classical virtual memory
  - sum of guest physical RAM > physical RAM

Page sharing
- multiple VMs running the same guest have a lot of common pages
  - text segments, zoned pages
  - hypervisor detects pages with same content
  - keeps hash of every page
  - uses copy-on-write to map those to a single copy
    - up to 50% memory savings [Waldspurger 02]
Device Virtualization Techniques

#### Virtual device drivers
- Guest OS contains virtual drivers
  - forwards guest I/O requests to real driver via hypervisor
  - very simple driver
- Need only support small number of different virtual devices
  - e.g. one type of virtual NIC, one type of virtual disk
- Virtualizes at driver interface
  - must re-implement/port all drivers in hypervisor
  - unfeasible for contemporary hardware

#### Core idea:
Combines advantages of pure and para-virtualization [LeVasseur et al, 08]

#### Hypervisor passes requests through to host
- Forwards guest I/O requests to real driver via hypervisor
- guest

#### Drawbacks:
- Hypervisor implements device-sharing policies
  - need only support small number of different virtual devices
- X: Hypervisor unfeasible for contemporary hardware
  - e.g. one type of virtual NIC, one type of virtual disk
  - Forwards guest I/O requests to real driver via hypervisor

#### Device-driver OS
- Special guest OS contains real drivers
  - Xen: Dom, guest
  - Hypervisor passes requests from virtual driver through to driver OS
  - Can re-use driver guest's native drivers unchanged

#### Disadvantages:
- needs source (at least assembler output of compiler)

#### Soft Layering aka Pre-Virtualization
- Combines advantages of pure and para-virtualization [LeVasseur et al, 08]
- post-process assembly code (compiler output)
  - prepares "pre-virtualizing" code
  - more flexible than binary re-writing
  - use semantic info from compiler
  - replace instruction sequences by hypervisor
  - hook onto macros etc.
  - no need to keep addresses invariant
  - jump to virtualization code may need more space than virtualized instruction
  - inter-well fix up addresses
  - can expand code for virtualization
    - can do much virtualization in-line
  - avoid branches to virtualization code

#### 2nd idea: do actual fix-up at load time
- leaves original (unvirtualized) instructions in binary
- patches the original instructions (and no-ops) during load
- generates a "hypervisor-neutral" patch
- can patch at load time for any supported hypervisor
- can run on bare hardware without any patches
- has most of the properties of pure virtualization
  - need only support small number of different virtual devices
  - can expand code for virtualization
- can do much virtualization in-line
- Design criteria: needs source (at least assembler output of compiler)
Soft Layering aka Pre-Virtualization

- 3rd idea: feedback loop for optimisation
  - initially only substitute most important subset of instructions
  - non-trapping sensitive instructions
  - obviously performance-critical
  - profile virtualization traps at run-time
  - hypervisor records location and frequency
  - use this to reduce virtualization overheads
  - identify hot spots from profiling data
  - annotate hot spots in source code
  - add replacement rules to pre-virtualizer
  - re-run pre-virtualization and link

- Advantage: guided optimization
  - similar to “optimized para-virtualization” (Magenheimer & Christian 04)
  - but less ad-hoc

Uses of Virtual Machines

- Multiple (identical) OSes on same platform
  - the original raison d’être
  - these days driven by server consolidation
  - interesting variants of this:
    - different OSes (Linux + Windows)
    - old version of same OS (Windux for stuff broken under Vista)
    - OS debugging (most likely uses Type II hypervisor)
  - Checkpoint-restart
    - minimise lost work in case of crash
    - useful for debugging, incl. going backwards in time
    - re-run from last checkpoint to crash, collect traces, invert trace from crash
    - life system migration
      - load balancing, environment take-home
  - Ship application with complete OS
  - reduce dependency on environment
  - “Java done right”
  - How about embedded systems?

Virtualization Performance Enhancements (VT-x)

- Extended page tables (EPT) provide two-stage address translation
- guest virtual -> guest physical by guest’s PT
- Hypervisor-configurable register makes some VM exits optional
- allows delegating handling of some events to guest
  - e.g. interrupt, floating-point enable, I/O bitmaps
- selected exceptions, eg syscall exception
- reduce hypervisor traps
- Exception injection allows forcing certain exceptions on VM entry
- TLB refill walks both PTs in sequence

I/O Virtualization Enhancements (VT-d)

- Introduce separate I/O address space
- Mapped to physical address space by I/O MMU
- Makes DMA safely virtualizable
- device can only read/write RAM that is mapped into its I/O space
- Useful not only for virtualization
  - safely encapsulated user-level drivers for DMA-capable devices
  - ideal for microcontrollers
  - AMD IOMMU is essentially same

Hardware Virtualization Support

- Intel VT-x/VT-c: virtualization support for x86itanium
- Introduces new processor mode: VMX root mode for hypervisor
- In root mode, processor behaves like pre-VT x86
- In non-root mode, all sensitive instructions trap to root mode (“VM exit”)
  - orthogonal to privilege rings, i.e. each has 4 ring levels
  - very expensive traps (700+ cycles on Core processor)
  - not used by VMware for that reason [Adams & Agesen 06]
  - Supported by Xen for pure virtualization (as alternative to para-virtualization)
  - Used exclusively by KVM
  - KVM uses whole Linux system as hypervisor!
  - Implemented by loadable driver that turns on root mode
  - VT-T (Itanium) also reduces virtual address-space size for non-root
  - Similar AMD (Pacifica), PowerPC
  - Other processor vendors working on similar feature
  - ARM TrustZone is partial solution
  - Aim is virtualization of unmodified legacy OSes

Halfway There: ARM TrustZone

- ARM TrustZone extensions introduce:
  - new processor mode: monitor
  - similar to VT-x root mode
  - banked registers (PC, LR)
  - can run unmodified guest OS binary in non-monitor kernel mode
  - new privileged instruction: SM
  - enters monitor mode
  - new processor status: secure
  - partitioning of resources
    - memory and devices marked secure or insecure
    - in secure mode, processor has access to all resources
    - in insecure mode, processor has access to insecure resources only
  - monitor switches world (secure -> insecure)
  - really only supports one virtual machine (guest in insecure mode)
  - need another hypervisor and para-virtualization for multiple guests
Why Virtualization for Embedded Systems?

Use case 1: Mobile phone processor consolidation
- High-end phones run high-level OS (Linux) on app processor
  - supports complex UI software
- Base-band processing supported by real-time OS (RTOS)
  - Medium-range phone needs less grunt
    - can share processor
    - two VMs on one physical processor
    - hardware cost reduction

Use case 1a: License separation
- Linux desired for various reasons
  - familiar, high-level API
  - large developer community
  - free
- Other parts of system contain proprietary code
- Manufacturer doesn’t want to open-source
- User VM to contain Linux + GPL

Use case 1b: Software-architecture abstraction
- Support for product series
  - range of related products of varying capabilities
- Same low-level software for high- and medium-end devices
- Benefits:
  - time-to-market
  - engineering cost

Use case 1c: Dynamic processor allocation
- Allocate share of base-band processor to application OS
  - Provide extra CPU power during high-load periods (media play)
  - Better processor utilisation
  - Higher performance with lower-end hardware
  - HW cost reduction

Use case 2: Certification re-use
- Phones need to be certified to comply with communication standards
- Any change that potentially affects comms needs re-certification
- UI part of system changes frequently
- Encapsulation of UI
  - provided by VM
  - avoids need for costly re-certification

Use case 2a: Open phone with user-configured OS
- Give users control over the application environment
  - perfect match for Linux
- Requires strong encapsulation of application environment
  - without undermining performance!
**Why Virtualization for Embedded Systems?**

**Use case 2b: Phone with private and enterprise environment**
- Work phone environment integrated with enterprise IT system
- Private phone environment contains sensitive personal data
- Mutual distrust between the environments
  - Strong isolation needed

**Why Virtualization for Embedded Systems?**

**Use case 2c: Security**
- Protect against exploits
  - Modern software attacked by UI exploits
    - Compromised application OS could compromise RT side
    - Could have serious consequences
      - E.g., jamming cellular network
- Virtualization protects
  - Separate apps and system code into different VMs

**Why Virtualization for Embedded Systems?**

**Use case 3: Mobile internet device (MID) with enterprise app**
- MID is open device, controlled by owner
- Enterprise app is closed and controlled by enterprise IT department
- Hypervisor provides isolation

**Why Virtualization for Embedded Systems?**

**Use case 3a: Environment with minimal trusted computing base (TCB)**
- Minimise exposure of highly security-critical service to other code
  - Need a minimal programming environment
  - Goes beyond capabilities of normal hypervisor
  - Requires basic OS functionality

**Why Virtualization for Embedded Systems?**

**Use case 3b: Point-of-sale (POS) device**
- May be stand-alone or integrated with other device (e.g., phone)
- Financial services providers require strong isolation
  - Dedicated processor for PIN/key entry
  - HW cost reduction

**Why Virtualization for Embedded Systems?**

**Use case 4: DRM on open device**
- Device runs Linux as app OS, uses Linux-based media player
  - DRM must not rely on Linux
  - Need trustworthy code that:
    - Loads media content into on-chip RAM
    - Decrypts and decodes content
    - Allows Linux-based player to display
  - Need to protect data from guest OS

©2008 Gernot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License
**Why Virtualization for Embedded Systems?**

**Use case 4a: IP protection in set-top box**
- STB runs Linux for UI, but also contains highly valuable IP
  - highly-efficient, proprietary compression algorithm
  - operates in hostile environment
- Needs highly-trustworthy code that
  - loads code from Flash into on-chip RAM
  - decrypts code
  - runs code protected from interference

**Use case 5: Automotive control and infotainment**
- Trend to processor consolidation in automotive industry
  - top-end cars have > 100 CPUs!
  - cost, complexity and space pressures to reduce by an order of magnitude
- AUTOSAR OS standard addressing this for control/convenience function
- Increasing importance of infotainment
  - driver information and entertainment function
  - not addressed by AUTOSAR

**Enterprise vs Embedded Systems VMs**
- **Homogenous vs heterogenous guests**
  - Enterprise: many similar guests
    - hypervisor size irrelevant
    - VMs scheduled round-robin
  - Embedded: 1 HLOS + 1 RTOS
    - hypervisor resource-constrained
    - interrupt latencies matter

**Core Difference: Isolation vs Cooperation**
- Enterprise
  - Independent services
  - Emphasis on isolation
  - Inter-VM communication is secondary
    - performance secondary
    - VMs connected to Internet (and thus to each other)
  - VMs are subsystems accessing shared (but restricted) resources

**Devices in enterprise-style virtual machines**
- Hypervisor owns all devices
- Drivers in hypervisor
- Need to port all drivers
- Huge TCB

**Devices in embedded virtual machines**
- Some devices owned by particular VM
- Some devices shared
- Some devices too sensitive to trust any guest
- Driver OS too resource hungry
- Use isolated drivers
  - protected from other drivers
  - protected from guest OSes
Isolation vs Cooperation: Scheduling

→ Round robin scheduling of VMs
→ Guest OS schedules its apps

• Similar for energy management
  → energy is a global resource
  → optimal per-VM energy policies are not globally optimal

Inter-VM Communication Control

Modern embedded systems are multi-user devices!
→ Eg a phone has three classes of "users":
  • the network operator(s)
    → assets: cellular network
  • content providers
    → media content
  • the owner of the physical device
    → assets: private data, access keys

Modern embedded systems are multi-user devices!
→ Eg a phone has three classes of "users":
  • the network operator(s)
    → assets: cellular network
  • the owner of the physical device
    → assets: private data, access keys

Modern embedded systems are multi-user devices!
→ Different "users" are mutually distrusting
→ Need strong protection / information-flow control between them
→ Isolation boundaries ≠ VM boundaries
  • some are much smaller than VMs
  • individual buffers, programs
  • some contain VMs
  • some overlap VMs
→ Need to define information flow between isolation domains
High Safety/Reliability Requirements
- Software complexity is mushrooming in embedded systems too
  - millions of lines of code
- Some have very high safety or reliability requirements
- Need divide-and-conquer approach to software reliability
  - Highly componentised systems to enable fault tolerance

Componentisation for IP Blocks
- Match HW IP blocks with SW IP blocks
- HW IP owner provides matching SW blocks
  - encapsulate SW to ensure correct operation
  - Stable interfaces despite changing HW/SW boundary

Componentization for Security — MILS
- MILS architecture: multiple independent levels of security
- Approach to making security verification of complex systems tractable
- Separation kernel provides strong security isolation between subsystems
- High-grade verification requires small components

Embedded Systems Requirements
- Sliding scale of isolation from individual program to VM running full-blown OS
  - isolation domains, information-flow control
- Global scheduling and power management
  - no strict VM-hypervisor hierarchy
  - increased hypervisor-guest interaction
- High degree of sharing is essential and performance-critical
  - high bandwidth, low latency communication, subject to security policies
- Real-time response
  - fast and predictable switches to device driver / RT stack
- High safety/security requirements
  - need to maintain minimal TCB
  - need to support componentized software architecture / MILS

Virtualization in embedded systems is good, but different from enterprise
  - requires more than just a hypervisor, also needs general OS functionality
  - perfect match for good microkernel, such as OKL4...