PERSONAL - Virtualization Techniques

- Introduction
- ISA Emulation
  - Terminology
  - Emulation
    - Interpretation vs. Binary Translation
    - Fast Emulation (Adaptive)
    - Interpretation
    - Binary Translation
- Process VMs
  - Components
  - Compatibility
  - State Mapping
- High-Level Language VMs
  - Java Virtual Machine (JVM)
    - Comparison: ISA vs. V-ISA
    - Implementation
- System VMs
  - State Management
  - Resource Control
    - CPU Virtualization
    - Memory Virtualization
    - I/O Virtualization
  - Optimizations

Introduction

- **virtual machine (VM)**: virtualization or emulation of a computer system
- **abstraction**: simplification of a system for a given purpose, with unnecessary details omitted
  - e.g. "root, branches, leaves" (complex) vs. "tree" (abstraction)
- **(fixed) interface**: decouples development of layer implementations, increases exchangability (usually fixed ISA + OS)

    - e.g. a program designed for the x86 ISA and Windows can run on any* computer with Windows and a processor implementing the x86 ISA, e.g. AMD Opteron, Intel Core i7...
    - **(!) inflexible**: given binary bound to fixed platform (e.g. you can't natively run an x86 Windows application on an ARM-based UNIX system), optimizations hardware-bound, changes to ISA / OS interfaces not possible
- **virtualization**: construction of a *virtual environment* for an *inner system* using an *outer system*
    - **virtual interface**: provided *to* the inner system *by* the outer system (action)
    - **inner system**: guest system (virtual)
    - **outer system**: host system (real)
    - **formal**: given two structures X and Y...
        - **abstraction**: *simplified* model X abstracts *complex* model Y
        - **virtualization**: *virtual* machine X is realized by *real* machine Y
        - **elements**: states in X / disjoint sets of states in Y
            - **elements in X**: states of the simplified model / virtual machine
                - **(!)** every state in X must be mappable to a set of states in Y!
                    - **abstraction**: every simplified state must be realizable in complex model
                    - **virtualization**: every virtual state must be implementable by real machine
            - **elements in Y**: disjoint sets of states in complex model / real machine
            - **monomorphism (injective mapping)**: $X \to Y$
        - **operations**: state transitions
            - **abstraction**: $S'_j = A(op(S_i)) = op'(A(S_i))$
                - A: abstraction mapping function from state S in X to disjoint sets of states S' in Y
                - op: state-transition in X
                - op': set of states-transition in Y
                    - for any state transition op in X, there exists a matching function op' mapping sets of states in Y
            - **virtualization**: $S'_j = V(exec(S_i)) = exec'(V(S_i))$
                - V: virtualization mapping function from guest state X to host set of states S' in Y
                - $exec$: state-transition in X
                - $exec'$: set of states-transition in Y
                    - for any state transition $exec$ in X, there exists a matching function $exec'$ mapping sets of states in Y
            - **(!) difference**: virtualization usually does *not* hide complex details; abstraction simplifies one thing, while virtualization maps among different things

- **resource**: computer environments consist of various resources, on different abstraction layers, accessed via interfaces
  - **categories**:
    - **I/O** (e.g. hard drive, network card, GPU, display, mouse, keyboard...), **memory** (e.g. RAM, SQL database...), **execution** (e.g. CPU, interpreter...), **others** (e.g. time, credentials, user roles, PIDs...)
    - **OS-managed** vs. **not-OS-managed**
  - **system seen by user application**: user ISA, virtual address space, OS syscalls
  - **system seen by OS**: complete ISA, physical address space, direct I/O
  - **strategies**:
    - **partitioning** (e.g. break down large hard drive into multiple smaller ones, time multiplexing...)
    - **indirection** (e.g. virtual memory implemented by page tables, register i stored in register j ...)
- [hypervisor / virtual machine monitor (VMM)]: a type of computer software, firmware or hardware that creates and runs virtual machines
  - **classification of VMs**:
    - by **abstraction level** of machine interface (e.g. System VMs vs. Process VMs) **(*)**
    - by **virtualization technique** (e.g. full virtualization vs. paravirtualization)
    - by **amount of virtualization covered by HW** (i.e. VM source line count)
  - **types of VMs**:
    - **system VMs**: virtualization of a complete system environment, including I/O devices
      - **subtypes**: by privilege level...
        - **type I**: system level, bare-metal (native) (e.g. Xen, Hyper-V...)
        - **type II**: user level, on top of OS (e.g. QEmu)
        - **splitted**: some parts on system level, some on user level (e.g. VirtualBox, VMware...)
    - **process VMs**: virtualization of a process environment, where every process gets its own environment
      - **containers**: OS-level virtualization, *same OS and ISA*, communicating processes split into groups with own environments (e.g. namespaces, cgroups, chroot...)
    - **HLL VMs (high level language VMs)**: programs using ISA of HLL-VM, OS calls done by HLL platform library (e.g. JVM, HTML5/JS...)

ISA Emulation

Terminology

- **virtualization**: focus on providing virtual environment, not on how to get there
- **emulation**: *implementation technique*; mapping of one (operation of an) interface to another
  - **(!)** produces the same effect (functional behavior) (e.g. `call` on x86 $\longrightarrow$ `jal` on RISC-V, functionally)
- **simulation**: method used to *derive behavior* for a system model (not only functionality, but also estimation of non-functional behavior, such as power consumption or performance...)

Emulation

- **emulation** of a *source ISA* (guest) via a *target ISA* (host)
- **components**: instruction set, registers, memory model, interrupt handling

Interpretation vs. Binary Translation

- **interpretation**: step-by-step emulation of *every instruction*
  - high asymptotic runtime, low setup time
- **binary translation**: previous compilation of *code blocks* (i.e. sequences of instructions are translated from a source instruction set to the target instruction set); may include optimization (which increase setup time...)
  - low asymptotic runtime, high setup time
- **(!) rule of thumb**:
  - if an instruction is ran only once, use **interpretation**
  - if an instruction is ran multiple times (very often), use **binary translation**

Fast Emulation (Adaptive)

- **emulation loop**: fetch, decode, choose to interpret or use binary translation based on profiling data

Interpretation

- **emulation loop (decode-and-dispatch)**: central loop until halt or interrupt; fetch and *decode* the next instruction and *dispatch* it to an interpretation routine
- **(*) branch optimization**: reduce number of branches, use static prediction (e.g. tell compiler which branch is more likely), make branches predictable (e.g. avoid `push / ret` or `call / pop`)
- **optimizations**: various optimizations done to reduce branching as much as possible...
  - **indirect threaded interpretation**: eliminates interpreter loop and uses dispatch LUT...
    - **etymology**: the `switch` statement is replaced by an indirect jump using the address of the emulation routine found in a new `dispatch` lookup table
    - **problem**: jumps are time consuming (branch mispredictions, pipeline conflicts)

        - **solution**: decode next instruction at the end of the current instruction's emulation routine; eliminate main loop `switch` statement using dispatch lookup table (**key**: opcode [+ extended opcode], **value**: address of interpretation routine)
    - **predecoding**: parse instruction and put it in a form that's easier to interpret (e.g. instead of using opcodes, extended opcodes and arguments separately, convert them to a single value)
        - **problem**: single source instruction interpreted multiple times (instruction consists of multiple parts instead of just one; operands are coded in bits, which ruin alignment)
        - **solution**: convert to *intermediate form* (usually by combining separate information of opcodes and extended opcodes into one single, "new" opcode), store in easily accessible, *aligned* fields (array of structs containing instruction information)
            - **target program counter (TPC)**: used to index into predecoding table
            - **source program counter (SPC)**: used for jumps or when explicitly using PC
            - need space for predecode table; noticeable improvement for CISC interpretation
    - **direct threaded interpretation**: remove indirection caused by dispatch table...
        - **problem**: overhead created by centralized dispatch table (memory access + register indirect branch)
        - **solution**: replace opcodes in intermediate code with actual addresses of interpreter routines
            - may cause portability issues, but there exist workaround (e.g. using relative addresses, using gcc's `&&` operator...)
- **interpreting CISC**: problematic due to variability in opcode length, structure...
    - **problem 1**: complex operations
    - **solution**: hybrid decode-and-dispatch and threaded methods
        - **simple instructions**: direct threaded
        - **complex instructions**: centralized decode-and-dispatch function
    - **problem 2**: predecoding before interpretation difficult due to variable space needed for each instruction (you'd have to use the maximum space needed, even for simple instructions, to maintain consistency) and variable opcode lengths (detecting boundaries becomes difficult)
    - **solution**: two-step process
        - **first interpretation**: do predecoding on the fly and fill predecode table
        - **further interpretations**: use predecoded data

Binary Translation

- source registers usually mapped to target registers
- **static binary translation**: translation of entire program before emulation
    - **problem**: code discovery (e.g. indirect jump with register contents unknown until runtime, CISC ISAs...), code location (e.g. jump addresses)

    - **solution**: dynamic binary translation, *code caches*
- **unit**: dynamic basic block (BB)
    - **static basic block**: single entry point, single exit point (begin and end at all *branch instructions* and *branch targets*)
    - **dynamic basic block**: determined by flow of program during execution (begin at *instruction immediately executed after branch*, end at *next branch or jump*) $\rightarrow$ always leads to new translation if branch into middle of block is encountered (no entry found in translation table)
- **execution**: emulation manager (EM), starting with SPC...
    - is there a corresponding SPC-TPC entry in the translation table?
        - **no**: interpret and translate code and place into code cache; update translation table
        - **yes**: branch to TPC and execute translated block
    - get SPC for next block...
- **code cache**: similar to processor cache, but with some differences...
    - **~**: limited size $\rightarrow$ replacement strategy needed (e.g. LRU, LFU, FIFO...)
    - *****: variable sizes of translated blocks; may have interdependencies among entries (chaining)
    - **problems**: self-modifying code (cache must be invalidated), self-referencing code (must be according to source code, not translated code), trap handling (correct state must be produced)
        - keep copy of original code...
        - **solution 1**: hardware assisted; guest code is write protected - any attempt to write a page containing translated code results in a trap, after which the runtime can invalidate all the code on said page (can be very slow)
        - **solution 2**: check for modification before every BB execution (also very slow)
        - **solution 3**: for cases in which the HW allows separate instruction and data caches, the translation system can detect instruction cache invalidation instructions in guest code
    - **eviction**: different strategies are used when the code cache is full...
        - **LRU (least recently used)**: evict least recently used BB
            - **advantage**: makes the most sense
            - **problems**: must maintain order of uses, which adds additional overhead; must update all predecessor blocks; cache fragmentation when removing blocks throughout the cache
        - **complete flush**: completely invalidate entire cache
            - **advantage**: eliminate stale control paths that have changed and no longer reflect the hot path
            - **problem**: frequently used BBs have to be retranslated from scratch
        - **coarse-grained FIFO**: partition code cache into blocks; flush block when full

        - **advantage**: no backpointers needed for code blocks within a FIFO block
- **code management**: an emulator can examine the executed source code in any detail, which is useful for collecting program data (e.g. memory accesses, thread tracing...), hence, same-ISA emulation also makes sense
- **optimizations**: similar to interpretation...
    - **chaining**: the counterpart to threading in interpreters
        - **problem**: branching back to EM every time after exiting a translated block
        - **solution**: link blocks into chains as they are constructed
            - replace jump and link back to EM with direct branch to successor block at translation time
                - **unknown successor**: when possible, replace JAL with successor address; for indirect jumps, do if-then-else-chains, with the most common targets first, then table lookup as a last resort; for function call returns, use shadow stack to avoid map lookup
                - **unchaining**: when translated blocks are removed; translation table should contain references to predecessors of a block
    - **superblock**: a block of code with a single entry point but several possible exit points
        - **problem**: jumps are still expensive...
        - **solution**: combine (ideally frequently used) sequence of BBs
    - for individual BBs...
        - common subexpression elimination: searches for instances of identical expressions, and replaces them with a single variable holding the computed value
        - constant propagation, constant folding, **strength reduction**: evaluate constant expressions at compile time, substitute values of known constants in expressions at compile time, recognize and use more efficient operations (e.g. `<< 1` instead of `* 2`)
        - copy propagation: replace occurrences of targets of direct assignments with their values (i.e. eliminate `y = x`)
        - **multi-versioning**: translate same BB multiple times, specialized according to input data
- **multi-stage emulation**: multiple optimization levels, modification of translations (concatenate instead of chain, duplicate same translation a la loop unrolling, generate superblocks)
- **profiling**: measurement of runtime behavior (either via injection of measurement code or by using HW performance counters)
    - **node profile (BB profile)**: how often was a block executed? (required for multi-stage emulation)
    - **edge profile**: how often was a jump taken? (required for optimal superblock creation)
- **problems²**: register architectures (target might have less registers than source), condition codes (e.g. x86 sets these automatically, SPARC sets them explicitly, MIPS doesn't have any $\rightarrow$ lazy

evaluation, store operands and operations and evaluate codes only when needed), data formats (e.g. some ISAs may not support floating point arithmetic), memory address resolution (e.g. byte / halfword / word addressing), memory data alignment (e.g. word access lower bits 00, halfword access lowest bit 0), byte order (big vs. little endian)

Process VMs

- **goal**: emulation a user level process for a possibly different ISA and / or OS (emulation of user ISA and ABI, with each guest getting its own environment; process(es) running inside process VM look exactly the same as host processes and can interact with eachother); embed it into host OS as seamlessly as possible

Components

- **loader**: writes guest code and data into memory and loads runtime code
- **initialization**: allocates memory for the code cache and other tables used during emulation; invokes host OS to establish signal handlers, primarily for exceptions
- **emulation manager**: uses interpretation and / or binary translation using a predecoding / code cache
- **code cache manager**: decides which translations should be flushed out
- **profiling data**: dynamically collected; used to guide optimization during translation process
- **OS call emulator**: translates guest syscall into appropriate host OS call(s); handles result
- **interrupt / exception emulator**: handles traps and interrupts directed at guest process; precision (i.e emulating complete state) is important!
- **helper tables**: tables that help with precise exception handling

Compatibility

- **compatibility**: the accuracy with which a guest's behavior is emulated on the host platrform
  - **intrinsic / strict compatibility**: 100% accuracy, including processor bugs, implementation-defined behavior etc.; guest process cannot detect any difference in virtual vs. real environment
    - **formally**: given $S'_j = V(exec(s_i)) = exec'(V(s_i))$ for a guest system X running on a host system Y, $exec$ is **completely emulated** via $exec'$ (i.e. $exec$ has to be **known completely**)
  - **extrinsic / relaxed compatibility**: compatibility only holds for a subset of program binaries; relies on externally provided assurances / certifications of compatibility (e.g. all programs compiled with a specific compiler, program formally verified, limited resource requirements, certain bugs not present...)
    - **formally**: $exec''$ constructed such that $exec'$ is **sufficiently approximated**
    - **conditions**: conditions are imposed on a subset of guest software w.r.t...

        - **state mapping**: extrinsic compatibility only achieved for processes that do not exceed certain memory space
        - **mapping of control transfers**: extrinsic compatibility only achieved for programs where trap conditions are known not to occur
        - **user-level instructions (floating point instructions)**: extrinsic compatibility only achieved for programs where accuracy is sufficient to satisfy user's needs
        - **OS operations**: guest OS may avoid certain OS features to achieve extrinsic compatibility
- **verification**: complete verification extremely difficult, but can be simplified by splitting up operations and states of a process
    - **operations**: *execution of user ISA* vs. *interaction with OS* (syscalls, exceptions / interrupts)
    - **state**: *user-managed state* (memory, registers...) vs. *OS-managed state* (storage devices, networks...)
    - consistency of state mapping only checked at borders of operation sequences (user-OS); these are the only points where the state may be made visible to the outside

State Mapping

- **split on types**: registers, memory, storage... (guest-host types don't have to match, e.g. guest registers can be mapped in host memory)
- **register mapping**: based on number of registers
    - **less guest ISA registers than host ISA registers**: possible to map all guest registers to host registers
    - **same / similar number of registers**: might be theoretically possible, but problematic (e.g. EM needs registers for its own use, like interpretation) $\rightarrow$ use register context block
    - **more guest ISA registers than host ISA registers**: must use register context block, must be managed at runtime
- **memory address space mapping**: map guest address space to host address space and maintain protection requirements (i.e. map guest address A to host address A') (assuming flat, linear memory structure; not segmented...)
    - **translation table**: akin to a page table; fragmented, slow, but can be used as *general solution* (e.g. emulating 64-bit guest on 32-bit host)
        - **formally**: `A' = (A & (PageLen - 1)) + AddrTab[ A / PageLen ]` (offset inside page + index in translation table)
    - **direct / offset translation**: mapping via some fixed offset (can be 0 if the guest OS allocates address ranges); contiguous, *efficient* (best case 1-to-1 translation for offset 0), but typically for this to work, the host memory (including memory needed for VMM) must be *larger* than guest memory
        - **formally**: `A' = offset + A`

    - **compatibility**: based on size of host OS memory
        - **host OS memory larger than guest OS memory + VMM memory**: high performance, intrinsic VM implementation possible
        - **host OS memory not larger than guest OS memory + VMM memory**: either performance or intrinsic compatibility must be sacrificed
            - **sacrificing performance**: use offsets or, worse, translation tables
            - **sacrificing intrinsic compatibility**: the guest process doesn't have to use the maximum possible amount of memory, only a smaller chunk
    - **protection**: support restrictions (r,w,x) placed on different regions of memory space
        - **translation table**: directly supported in table; just add an information bit for the corresponding entry (slow, but correct)
        - **direct / offset translation**: host-supported memory protection (i.e. VMM directs host OS to implement page protection, via either *syscall* (`mprotect()`) or *signal* (`SIGSEGV`))
    - **page size differences**: *easy* if the guest OS page size is bigger than host OS page size, *difficult* otherwise (e.g. two pages with different protections in one host OS page)
- **exception emulation**: assuming *precise* (!) exceptions (where all instructions before X have been executed, no instructions following X were executed, and w.l.o.g. X *hasn't* been executed)
    - **exception (trap)**: direct result of program execution, produced by a specific instruction (e.g. division by 0, memory violation)
    - **interrupt**: async, external event (e.g. async I/O, async IPC)
    - **ABI visible**: exceptions that are visible at ABI level, including all exceptions returned to the application via an OS signal (which itself also includes all exceptions that cause the program to terminate)
    - **ABI invisible**: the ABI is unaware of the exceptions existence (no signal, no termination) (e.g. timer interrupt for scheduling)
    - **exception detection**: various ways
        - **interpretive trap detection**: trap explicitly checked as part of interpretation routine (e.g. overflow recognized by explicitly comparing input operands and final sum); can always be done, albeit (very) inefficient
        - **hardware-triggered exception**: registration of notification for all possible exception types at host OS by VMM (initializer), semantic check (i.e. check if the semantics of the exception are slightly different from guest to host OS), translation of host OS notifications into corresponding guest OS notifications
        - **issues**: if there are differences in ABI visibility or functionality between a guest and host OS exception, this could lead to some problems
            - **guest ABI visible, host ABI invisible**: use interpretive trap detection

        - **guest ABI invisible, host ABI visible**: needs check whether exception should be ignored
        - **one exception type in host ABI for multiple exception types in guest ABI**: needs differentiation at runtime (e.g. host has same overflow exception for integers and FP)
    - **interrupt handling**: depends on method of emulation
        - **interpretation**: delay fine; let routine finish, then create exception state and call interrupt handler
        - **binary translation**: more difficult due to possiblity of chaining causing huge delays $\rightarrow$ once interrupt is received, transfer control from translated block back to runtime, unlink translated block from subsequent blocks, return control to translated block, runtime handles interrupt after translated block finishes
            - **assumptions**: no loops in translation block, target code is at a precise interruptible state
- **OS emulation**: since a process VM is only required to maintain compatibility at ABI level, only the function or semantics of OS calls are emulated, not the actual individual instructions of the guest OS code
    - **same OS**: while the syscalls themselves may be the same, arguments and return values may be formatted differently (e.g. arguments passed via the stack instead of registers) $\rightarrow$ conversion wrapper needed (can be inlined for optimization)
    - **different OS**: no general solution here, since OS calls are generally much more complicated than ISA stuff (e.g. file systems, device I/O, process management...) and some are downright impossible (e.g. if a guest OS has a time-of-day feature, but the host OS doesn't, it becomes impossible to emulate); done on a case-by-case basis, similar to porting code from one OS to another, but more general

High-Level Language VMs

- **high-level language VM (HLL VM)**: special kind of process VM, where the (virtual) ISA only consists of user-level instructions and is not designed for a real hardware processor (so, it should* only be executed on a virtual processor); rather, it's designed to be supported on a number of operating systems and hardware platforms (also, the virtual ISA contains *a lot* of metadata)
- **advantages**: depending on the use case...
    - the **ABI** (V-ISA, metadata, library functions) is **freely constructible**
        - **abstractions**: more specific abstractions as a result of decoupling from hardware ISAs (e.g. complex types, synchronization abstractions like monitors)
        - **robustness**: strong type-checking and garbage collection, among others
        - **security and protection**: guest programs operate within their own *sandboxes*
        - **compact code**: easier to share via network
    - **platform-independent programs**: enables platform-independent distribution of application software through use of platform-independent code and metadata

        - **end user**: can run on every new processor, can use new hardware extensions (e.g. SSE), availability not dependent on developer porting for target system
        - **software developer**: larger market, correctness checked only against documented ABI (so no processor bugs, undocumented features or other quirks) no porting required (and, by extension, no maintenance of multiple ports)
        - **hardware manufacturer**: larger market, free from ISA compatibility requirement
- **disadvantages**: the traditional disadvantages come with being a VM...
    - **additional VM dependency**: more memory, potentially slow with bad implementations
    - target hardware performance optimizations by programmers not possible

Java Virtual Machine (JVM)

- **metadata**: format and semantics of code and data, specified in `.class` file
    - **class metadata**: module, base class, interfaces, visibility
    - **method metadata**: signature, sync properties
    - **variable metadata**: type
- **data types**: primitives (`int`, `char`, `float`...), references (`null`, objects, arrays)
- **data storage**: global (global variables), local (method-local variables), operand (temporary storage for variables while they are being operated on)
    - **stack**: holds *local and operand storage*, aswell as function arguments; no objects or arrays, just *references*
    - **global memory**: *heap* of unspecified size; can hold *static* and *dynamic* objects; objects can only be accessed via references of appropriate type
    - **constant pool**: holds *constant data*; defined in `.class` *files*
- **instructions**: 1 byte opcode + 0 or more optional operands (index: indices in constant pool or local storage; data: immediate data or offsets for branching)
    - **types**: data transfer, conversions, ALU functions, jumps (restricted to current method)
- **binary classes**: loaded on demand, verified at load time
    - **contains**: magic number, version info, constant pool (constant values and references), access flags (public/private/protected, interface, abstract...), this class, super class, interfaces, fields, methods, attributes...
- **not a Von-Neumann architecture**: separates code from data (they are treated as distinct kinds of things which are accessed via separate means), no registers (stack-based)

Comparison: ISA vs. V-ISA

- **metadata**: included in V-ISA
- **memory**: typed, unlimited, no pointers or pointer arithmetic in V-ISA (only references, which are always valid)

- **exception handling**: local / verifiable in V-ISA; precise barely needed
- **instructions**: flags avoided in V-ISA; stack-based
- **code vs. data detection**: specified in V-ISA metadata
- **self modifying code**: prohibited in V-ISA
- **OS dependence**: avoided in V-ISA through platform library

Implementation

- **major components**: class loader subsystem, memory system (incl. garbage collected heap), emulation manager
  - **memory**: program code, global memory, stacks (JVM, native library code)
    - program counter, stack pointer implicit (cannot be inspected by program)
  - **garbage collector**: responsible for finding objects no longer needed to make room for new objects
    - object becomes *garbage* when it becomes *unreachable*
    - **mark-and-sweep**: start with root references, *mark* all reachable objects, *sweep* (free) unmarked objects
    - **optimizations**: compactification on high fragmentation, generational GC (separate pools for long and short lived objects; GC called more frequently on short pool), concurrent GC (run in background; *synchronisation* required!)
  - **emulaton engine**: as seen before...
  - **native method interface**: interface for accessing OS-managed functions, e.g. I/O or graphics operations
  - **class loader**: dynamically finds and loads binary classes; verifies correctness and consistency of binary classes; network integration (fetching `.class` files over network)
    - multiple loader instances for different security domains (sandboxes)
- **optimizations**: done below the plaform interface (so, not in `javac`, but in the JVM implementation)
  - **function inlining**: classic optimization technique for OOP (+: no overhead for call, -: larger binary, code explosion)
  - **virtual function inlining**: guarded, since type needs to be checked at runtime
  - **multiversioning**: generate multiple versions of a method on demand, choose at runtime
  - **object layout**: object nesting, replace local objects with local variables

System VMs

- **system VM**: virtualization of a complete system environment, *including hardware*, using a virtual machine monitor (VMM) / hypervisor

- **use cases**: multiprogramming, multiple secure environments, mixed-OS environments, legacy applications, multiplatform app development, transitioning to a new OS, system software development, OS training, help desk support, OS instrumentation / profiling, event monitoring, checkpointing...
- **outward appearance**: *illusion* of multiple machines, acheived either through software or through replicating of a subset of hardware resources
- **assumptions (for this chapter)**: *same ISA, single-processor systems*

State Management

- **indirection**: state of guest is held in fixed location in host memory hierarchy with a VMM-managed pointer indicating currently active guest state (similar to how page tables work with page table pointers)
  - **problem**: inefficient if the memory resource that holds the guest state has characteristics that are different from those on the native platforms (e.g. mapping registers to values in memory, such that guest register copy operations become host memory operations, which are much slower)
- **copying**: copy guest state to natural level in memory hierarchy (e.g. copy guest registers to host registers), then copy back on switch to different guest OS (one-time overhead; preferable for frequently used state information, like general purpose registers)

Resource Control

- resources assigned to VM on creation; it is important for the VMM to be able to *get these resources back* to be able to assign them to a different VM, so the VMM maintains overall control over the hardware resources
- **privileged resource**: host resource that cannot be assigned to a guest, either because it is needed by the VM(M?) itself, or because the assignment is not revocable at any time
  - **(!)** every privileged resource *must be emulated*
- **time sharing**: using an interval timer (privileged), perform guest OS switch (in interrupt handler); privileged resources not used directly by VM; rather, they are emulated by the VMM
  - **problem**: frequent switching $\rightarrow$ high overhead; infrequent switching $\rightarrow$ suboptimal resource usage
- **native VM**: +: full access to physical resources, -: needs drivers for hardware to make use of
- **hosted VM**: +: access to abstract host OS resources, -: slow
- **dual-mode hosted VM**: +: can use host HW via kernel module (for performance critical resources, e.g. MMU, exceptions...)

CPU Virtualization

- **methods**: *emulation* (always possible; only option if guest and host ISAs differ) vs. *direct native execution* (possible only if host ISA and guest ISA are the same, under certain conditions*; has to

be revocable!)

    - **goal**: maximize performance, i.e. use direct native execution whenever possible
- **privileged instruction**: *traps* if the machine is in *user mode*, *doesn't trap* if the machine is in *system mode* (in VMM: separate interpreter for each privileged instruction)
- **control-sensitive (w) instruction**: instruction that attempts to change the configuration of resources in the system (e.g. mode of system, phyiscal memory assigned to a program) (e.g. load PSW `LPSW` traps in user mode, set CPU timer `SPT` traps in user mode)
- **behavior-sensitive (r) instruction**: instruction whose behavior / result depends on the configuration of resource (e.g. load real address `LRA` depends on mapping of real memory resource, pop stack into flags register `POPF` acts as no-op for interrupt-enable flag)
- **innocuous instruction**: instruction that is neither control-sensitive, nor behavior-sensitive
- **Popek & Goldberg sufficient condition for processor ISA virtualization**: every *sensitive instruction* (i.e. every instruction accessing a privileged resource) is *privileged* (i.e. traps in user mode)
    - all nonprivileged instructions can be executed natively on the host platform without emulation
    - **critical instruction**: sensitive, but not privileged (i.e. doesn't generate a trap in user mode, e.g. `POPF`) $\longrightarrow$ ISA violates P&G
        - **solution 1**: patching (scan guest code before execution, replace critical instructions with trap to VMM)
        - **solution 2**: hardware-assisted (e.g. Intel VTX's VMX mode)
    - **hybrid VM**: some nonprivileged instructions must be emulated

Memory Virtualization

- **physical memory**: actual hardware memory, on host
- **real memory**: the guest VM's *illusion* of physical memory; VMM maps guest's *real* memory to *physical* memory
    - VMM maintains **real map table** mapping guest real pages to physical pages (used for I/O operations, for instance, which operate on real addresses)
    - some pages of guest real memory may not be mapped to physical memory
    - real memory of VM can be larger than physical memory
- memory management unit (MMU): translates virtual memory addresses into physical addresses in main memory (privileged resource)
    - translation lookaside buffer (TLB): memory cache that stores the recent translations of virtual memory to physical memory, part of the unit's MMU
- **page table architected (page table specified in ISA)**: TLB not visible to operating system (maintained and used only by hardware); page table lookups done completely in hardware (TLB

miss → hardware finds entry in page table and will only cause a page fault for the OS to handle if the entry is not mapped to physical memory) (e.g. x86, ARM)

  - **shadow page table**: maps *virtual guest pages* to *physical host pages*; maintained by VMM, actually used by hardware to translate virtual addresses and update TLB; done by updating page table pointer to point to shadow page table
  - **page fault while VMM active**: different possibilities...
    - **page mapped in virtual page table of guest OS**: page fault entirely handled by VMM, invisible to guest OS (because a page fault would not have happened if the guest OS were running natively); possibly a result of the page being swapped out by the VMM and needs to be brought back
    - **page not mapped in virtual page table of guest OS**: VMM transfers control to trap handler of guest (indicating PF), guest OS modifies PT, caught by VMM, which then updates PT and SPT mapping
    - **faulty access**: guest OS passes SEGFAULT signal
- **TLB architected (TLB specified in ISA)**: TLB visible to operating system, page table format defined by OS (i.e. hardware unaware of page table structure; TLB miss → trap to OS) (e.g. MIPS, SPARC)
  - ~~VMM maintains copies of each guest's TLB (virtual TLB, VTLB) and manages real TLB~~
  - ~~**simple method**: rewrite TLB whenever a guest VM is activated using guest virtual and (translated) host physical addresses from guest TLB~~
    - ~~**problem**: high overhead each time a switch happens (from one guest VM to another *or* from a guest VM back to the VMM, and vice-versa), especially for large TLBs~~
  - ~~**address space identifier (ASID)**: a tag used in TLB entries to distinguish between different virtual address spaces; allows multiple processes to have address space mappings in a TLB simultaneously instead of having to flush the TLB each time (so, for instance, ASID 10 belongs to process ID 1, ASID 20 belongs to process ID 2 and so on; this also allows processes to have the "same" virtual page address in the TLB simultaneously, i.e. PID 1 has page 100 in TLB with ASID 10, PID 2 has different page 100 in TLB with ASID 20 and so on...)~~
    - ~~**ASID register**: stores ASID of current process; checked against ASID in TLB entry as part of address translation~~
    - ~~each guest has a virtual ASID register; VMM maintains real ASID register~~
- **memory ballooning**: a technique that is used to eliminate the need to overcommit host memory used by virtual machines (VMs) by letting each VM effectively "give back" unused pages of (virtual) memory
  - **balloon driver**: inflate (allocate memory in the guest) or deflate (release memory)

I/O Virtualization

- VMM intercepts guest request to I/O device and carries it out using the physical device

    - anything above hardware level (syscall, drivers) is *OS-specific*
- **dedicated devices (pass-through)**: by nature dedicated to a particular guest (or at the very least given a very large amount of time; e.g. display, keyboard, mouse, audio); device doesn't have to be virtualized; requests bypass VMM and go directly to / from guest OS (in practice still routed through VMM due to guest OS running in user mode + interrupt interception done by VMM which determines that it came from a passthrough device, which then forwards it to guest OS)
    - **advanced programmable interrupt controller (APIC)**: allows flexible interrupt routing and catching on host
    - **input-output memory management unit (IOMMU)**: MMU connecting DMA-capable I/O bus to main memory; maps *device addresses* (memory mapped I/O addresses) to *physical addresses*; guest OSs can use hardware not designed for virtualization, with the IOMMU handling remapping
    - **multiplexing**: revoke device (save and restore state)
    - **single-root I/O virtualization (SR-IOV)**: PCIe extension; device can be configured to look like multiple devices on a PCIe bus, so that each virtual PCIe card is passed-through to a guest (e.g. Infiniband)
- **partitioned devices**: partition available resources among VMs (e.g. large disks partitioned into several smaller virtual disks, made available to VMs as dedicated devices)
    - VMM maps and translates virtual parameters into physical parameters (e.g. track and sector locations)
- **shared devices**: shared across multiple guest VMs at a fine time granularity (e.g. network adapter, with each guest VM having its own virtual network address, maintained by the VMM for each guest VM)
- **spooled devices**: spooling, i.e. shared but at a much higher granularity (e.g. printer)
- **nonexistent physical devices**: VMM "behaves" like emulated physical device (e.g. guest VMs connected to eachother via a shared network without an actual physical network card; the VMM intercepts the I/O requests and emulates the transmission of network packets)

Optimizations

- **bottlenecks**: startup, instruction emulation (switching between user and system mode), interrupt handling (via VMM), guest switching (saving and restoring states), resource emulation
- **hardware extensions** (e.g. ISA: Intel VTX)
    - **nested page tables**: additional page table layer, where inner page table layers are passed through to guests, so no MMU emulation (SPTs) needed (may, however, be expensive)
    - **tagged TLB**: simultaneous TLB entries of multiple guests (ASIDs)
    - **multiple timers**: directly supported in hardware
- **paravirtualization**: bypassing virtualization needs by modification of guest to use VM-specific extensions to the machine interface; using "hypercalls" instead of instructions / hardware interfaces

    - critical instructions are removed in favor of hypercalls
    - **techniques**: replacing I/O drivers (e.g. VirtualBox guest additions), direct detection via OS (e.g. Linux's ParaVirt Interface), on-the-fly patching
- **reduced guests**: application with only required OS functionality provided by libraries (e.g. MirageOS w/ OCaml); one privilege level only for guest
- **guest migration**: cold migration (stop guest on host 1, transfer state to host 2, resume guest on host 2) vs. hot / live migration (transfer only "snapshots", i.e. "diffs")

Summary by Flavius Schmidt, ge83pux, 2025.
https://home.cit.tum.de/~scfl/