Process VM (1)

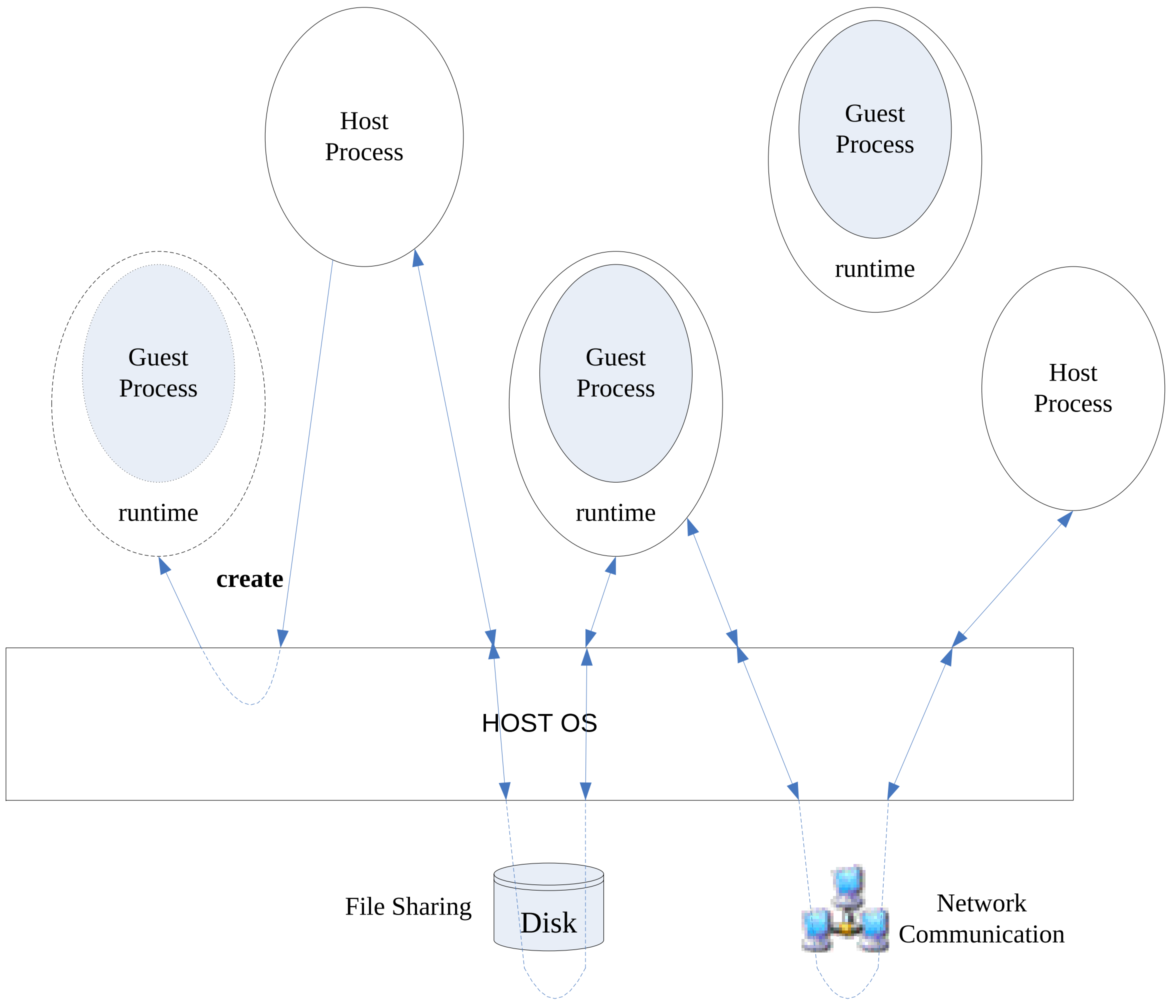
Outline

- Overview of Virtual Machine Implementation
- Compatibility
- Memory Address Space Mapping
- Memory Architecture Emulation
  - Memory protection
  - Self-modifying code

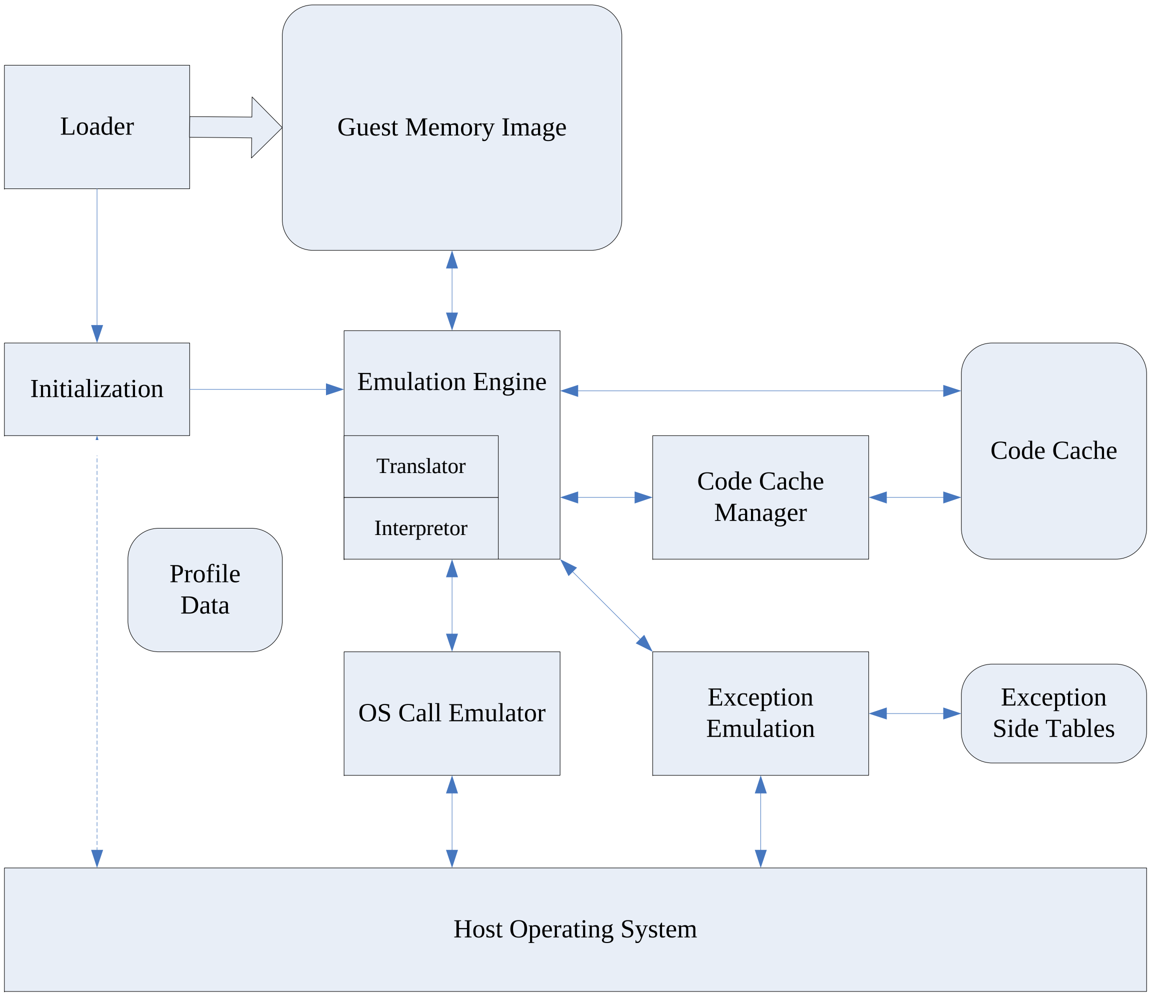
Process VM

- Allows running programs compiled for other systems
  - With different OS and/or ISA
  - Provide a virtual environment at the process level
- Examples
  - IA-32 EL
    - Running IA-32/Windows programs on Itanium/Windows
  - FX!32
    - Running IA-32/Windows programs on Alpha/Windows
  - ARIES
    - Running PA-RISC/HP-UX programs on Itanium/HP-UX
- Runtime software encapsulates a guest process
  - Give it the same appearance as a native host process

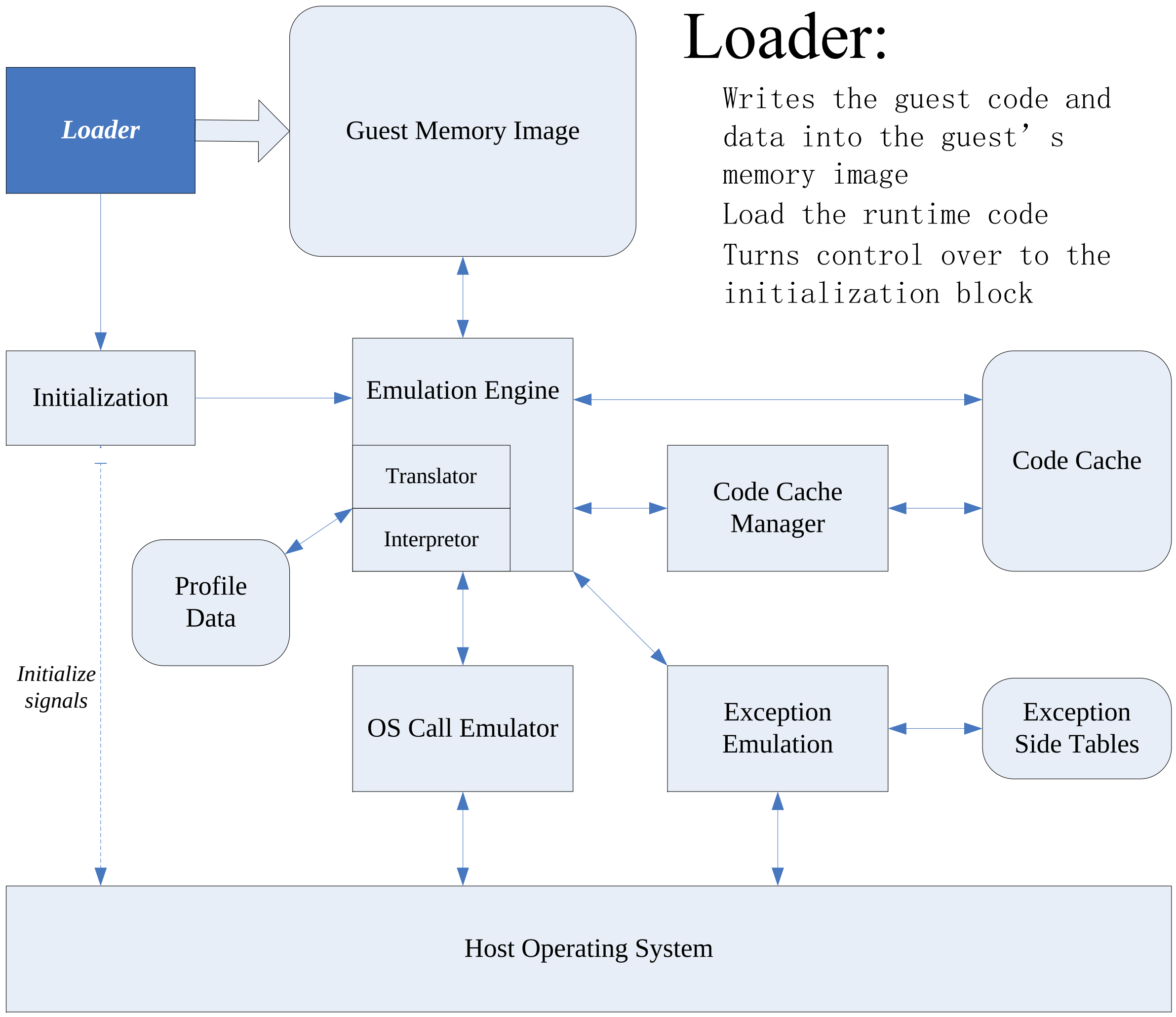
Process Virtual Machine



Virtual Machine Implementation



Virtual Machine Implementation



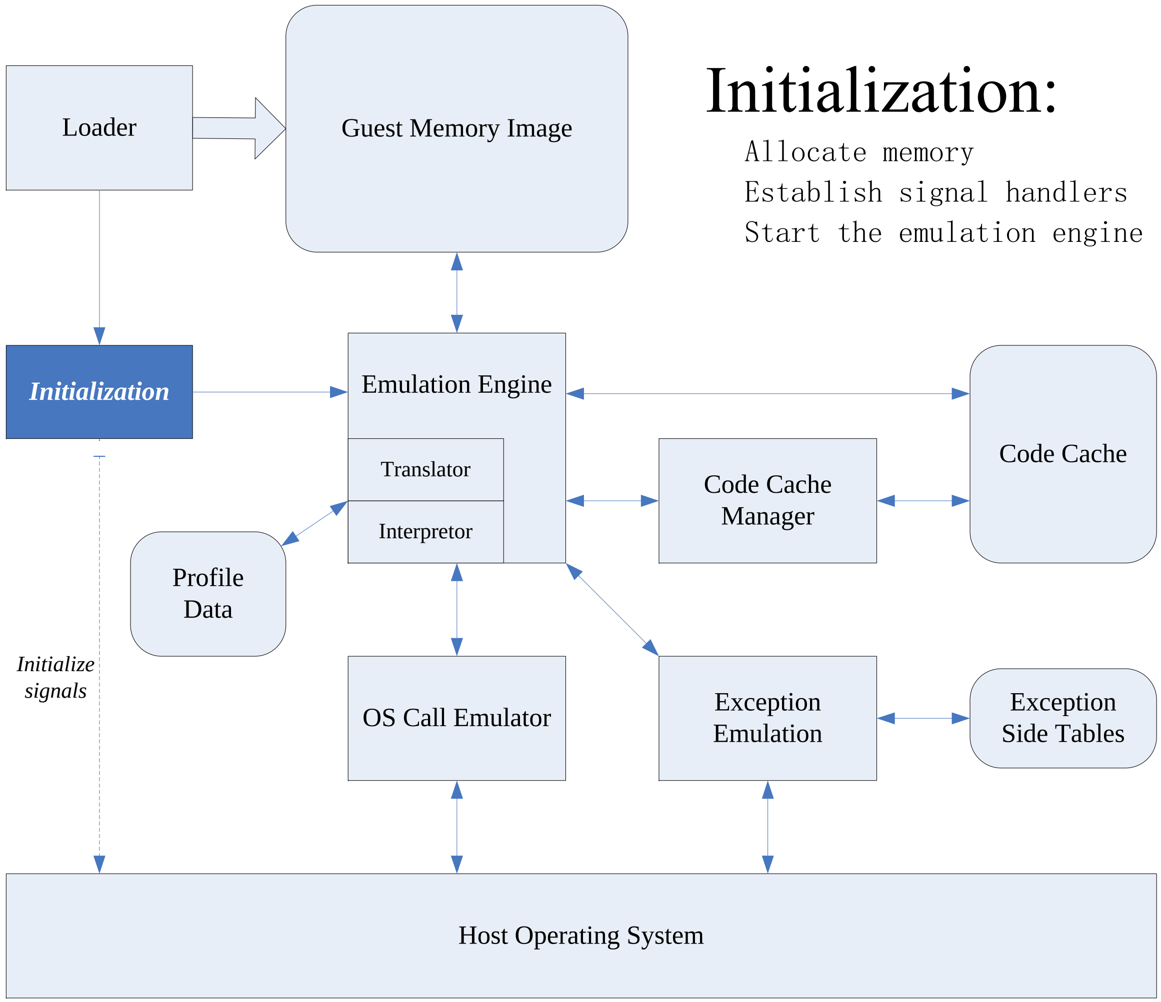
Loader:

Writes the guest code and data into the guest’s memory image

Load the runtime code

Turns control over to the initialization block

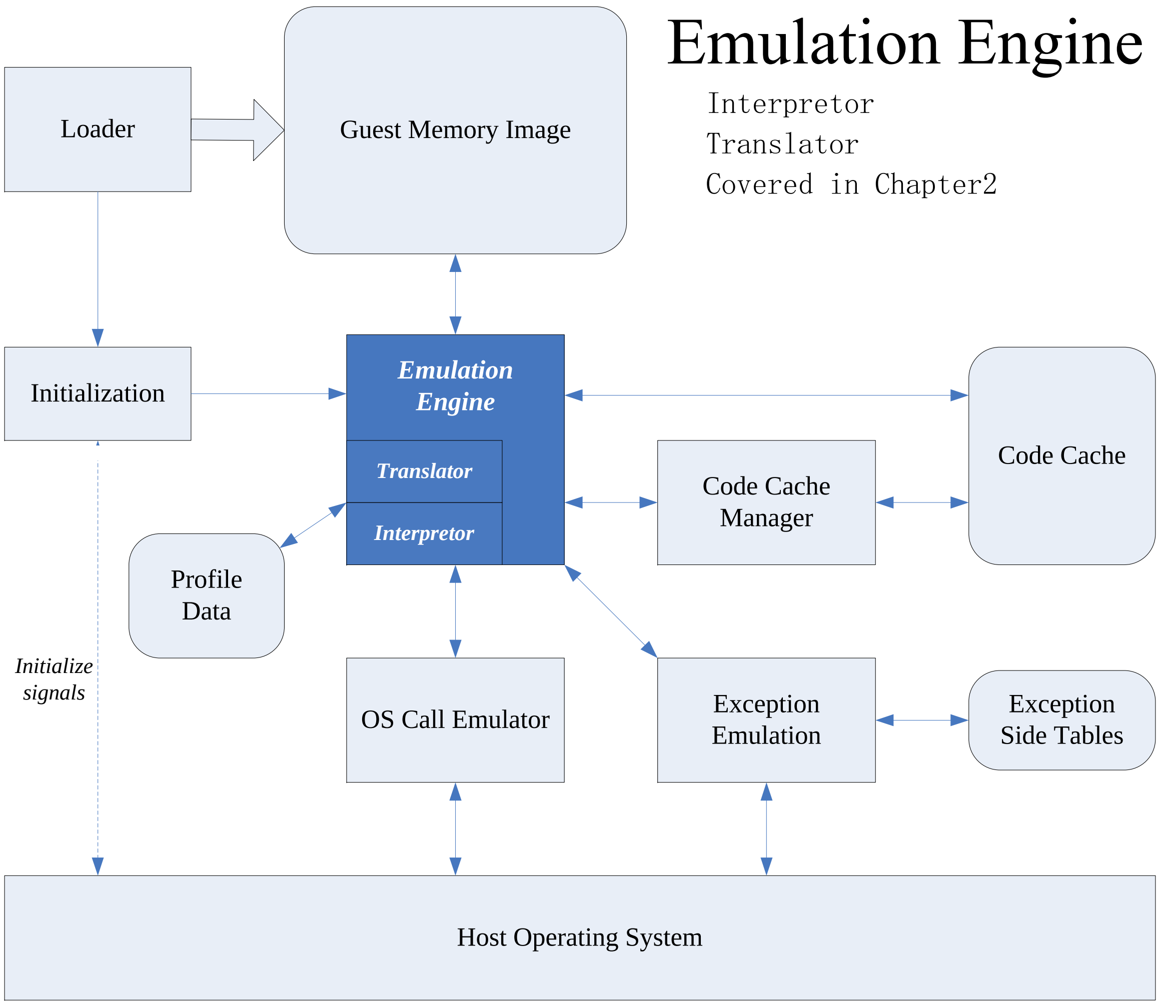
Virtual Machine Implementation



Initialization:

Allocate memory
Establish signal handlers
Start the emulation engine

Virtual Machine Implementation



Emulation Engine

Interpretor
Translator
Covered in Chapter2

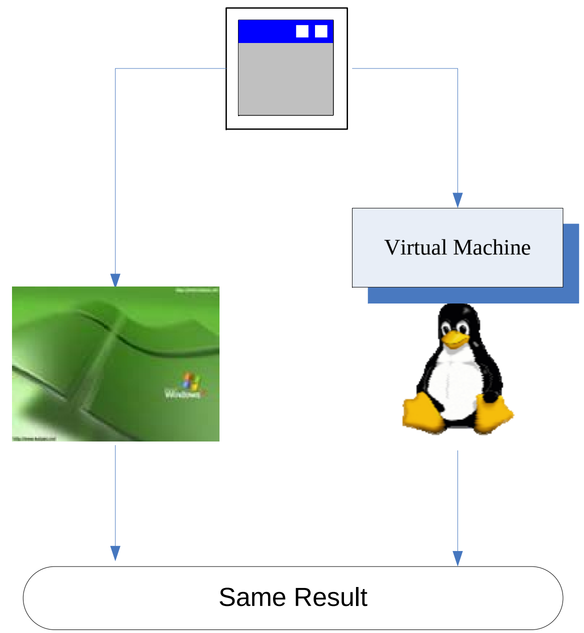
Process VM Components

- OS call emulator
  - Translates a system call issued by the guest program into appropriate system call(s) to the host OS and handles results
- Exception emulator
  - Handles traps occurring as a result of executing interpreted or translated instructions or external interrupts
    - Runtime should secure precise guest state and do appropriately
    - Resorts to signal handlers established at the initialization
      - ✓ All signals are registered
- Profile Database
  - Dynamically collected info which will be used for optimization

Compatibility

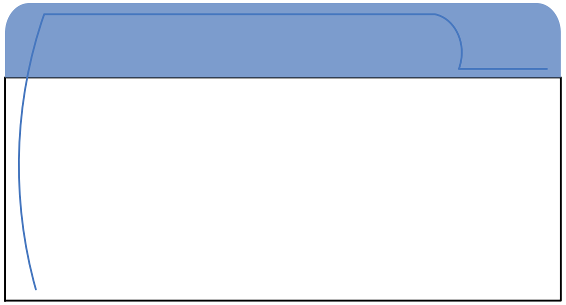
- Definition
  - Accuracy with which a guest's behavior is emulated on the host platform, as compared with its behavior on its native platform
  - Simply, result on VM = result on native platform ?

- A matter of correct functioning
  - Not a matter of Performance

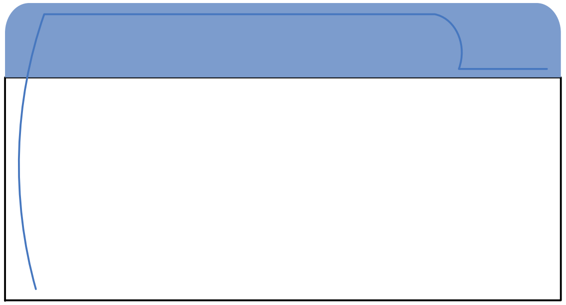


Level of Compatibility

- Intrinsic compatibility
  - Strict form of compatibility
  - Required by some system VMs
  - Too strict for process VMs
  - Another term: ***complete transparency***



- Extrinsic compatibility
  - Relies on externally-provided assurance of guest program as well as on the VM
    - E.g., “a program compiled with gcc using C standard libraries are compatible”
    - Resource requirement, verification, etc.



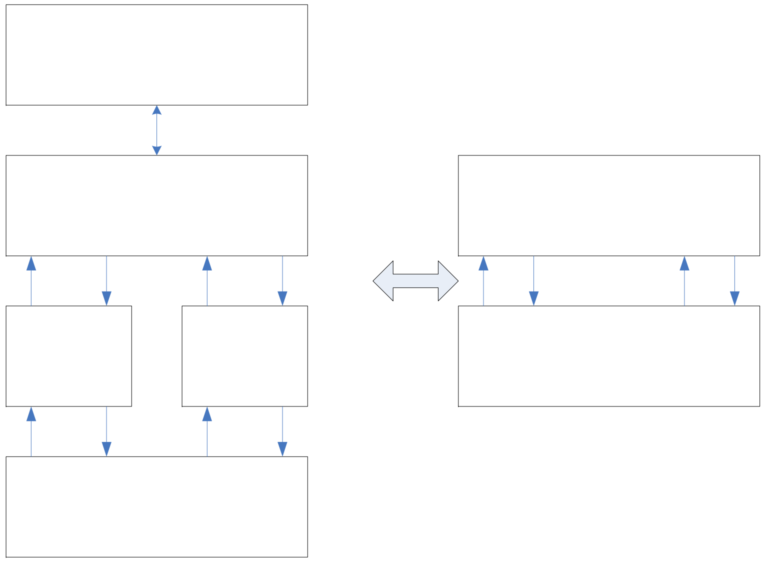
A Compatibility Framework

Proving compatibility is too hard, so we define a framework

- By decomposing the system (Guest program, runtime, OS, H/W)

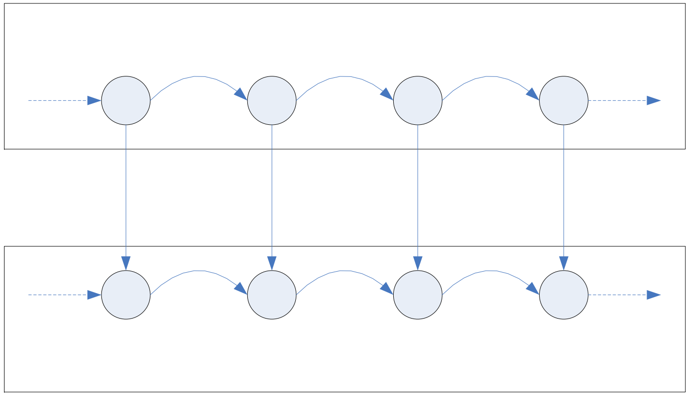
- Dividing states and mapping
  - User-managed state
  - OS-managed state
- Dividing operations
  - User-level instructions
  - Operating system operations

For each control transfer between user code and OS on a native platform, there is a corresponding control-transfer point in the VM



Sufficient Compatibility Conditions

- At the point of control transfer from emulating user instructions to the OS, the guest state (both user & OS managed) is equivalent to the host state, under the given state mapping
  - Equivalence is maintained at OS control transfer, not at instruction granularity, providing more flexibility in emulation
- At the point of control transfer back to user instructions, the guest state is equivalent to the host state, under the given state mapping



Why Sufficient?

- User-OS control transfers are the only points where the state may be made visible to the "outside world", yet
- Same compatible results could be achieved in other ways
  - E.g., when a system call reads/writes a small portion of guest memory, our condition requires all memory state should be equivalent

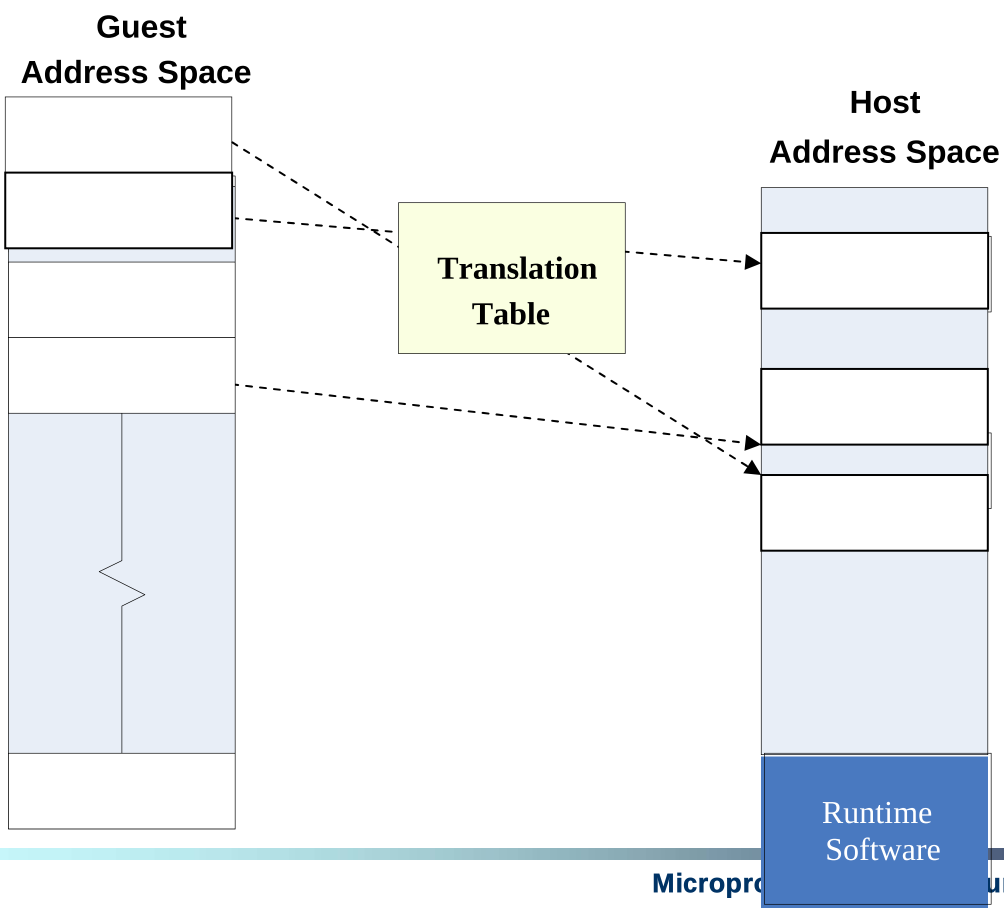
State Mapping

- Mapping user-managed state in registers and memory
  - Guest data and code mapped to host's user address space
  - Guest registers mapped to host registers and/or runtime data region of memory

- Register mapping is straightforward
- Memory space mapping
  - Map guest's address space to host's address space
    - Runtime emulator should map addresses for guest's load/store/fetch
  - Maintain protection requirements
  - Address mapping can be done by S/W or by H/W or by both

Runtime Software-Supported Translation

- Guest addresses are not contiguously mapped
  - Translation table is used for guest-to-host translation
  - Most flexible, but most software-intensive method



An Address Translation Example

- Code sequence for a load instruction

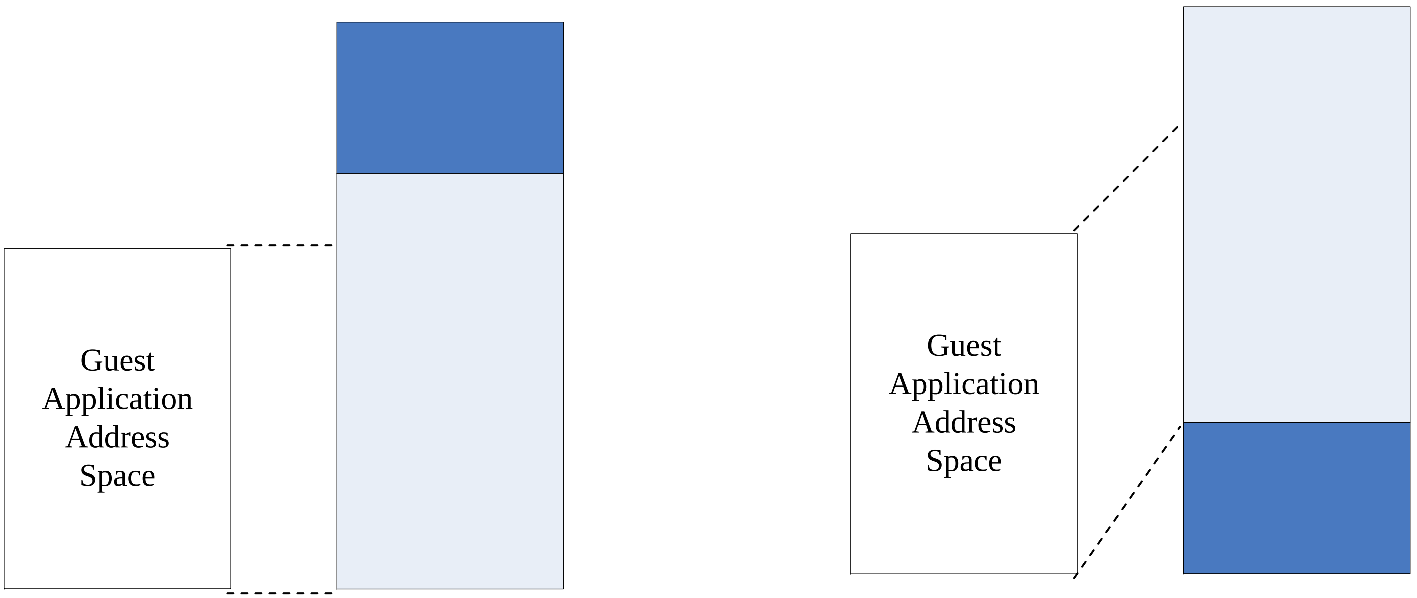
r1 ⇔ source address

r30 ⇔ base address of translation table

srwi r29,r1,16 ; shift r1 right by 16 bits

Direct Translation Method

- Guest address space is mapped contiguously
  - Source load/store can be translated 1-to-1 to target load/store (with a fixed offset added in the right case)
  - More efficient than the table method

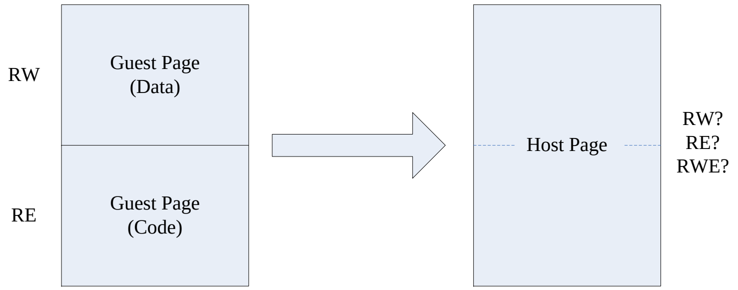


Memory Protection

- Protect guest address space in the host memory
  - e.g., protect a store to the code area of guest program
  - Protect according to guest ISA's read/write/execute protection
- If translation table is used, use protection information added in the translation table (correct but slow)
- If direct translation method is used, use the host OS and the host hardware for page protection
  - A system call specifying a page and its access privileges (e.g., `mprotect()` system call in Linux)
  - A signal for a memory access fault (e.g., `SIGSEGV`) which will be delivered to the runtime if there is a disallowed access

Page Size & Protection Type Issues

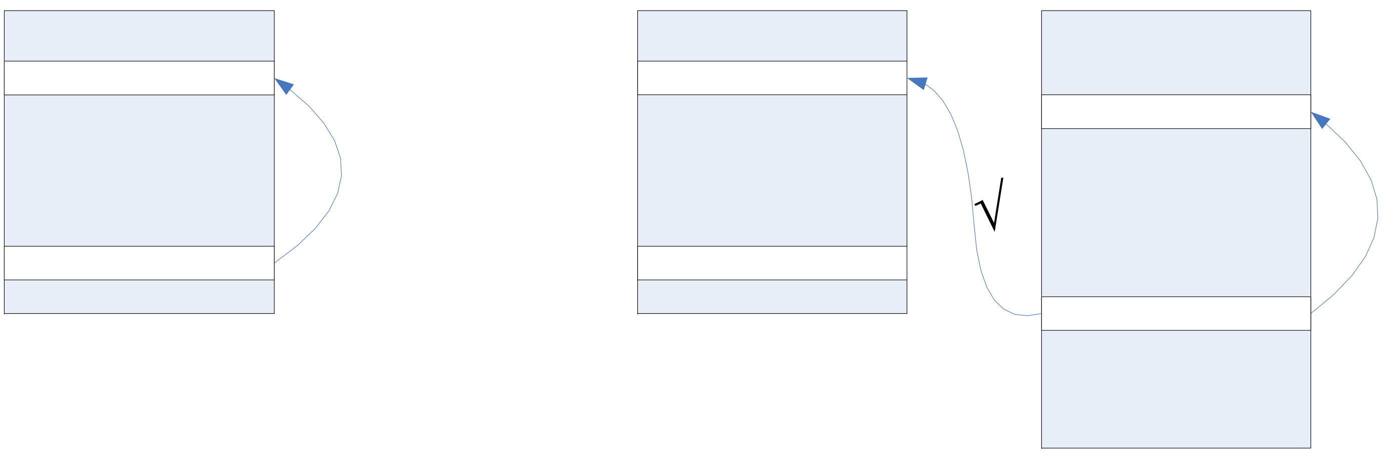
- Guest page size smaller than the host page size
  - If two differently-protected guest pages allocated to a host page
  - One solution is aligning to the host page boundaries
    - ✓ Reduce efficiency & portability
  - Another is giving lesser privilege
    - ✓ Handle the “extra” traps



- Protection types mismatch
  - Host supports a subset of guest protections

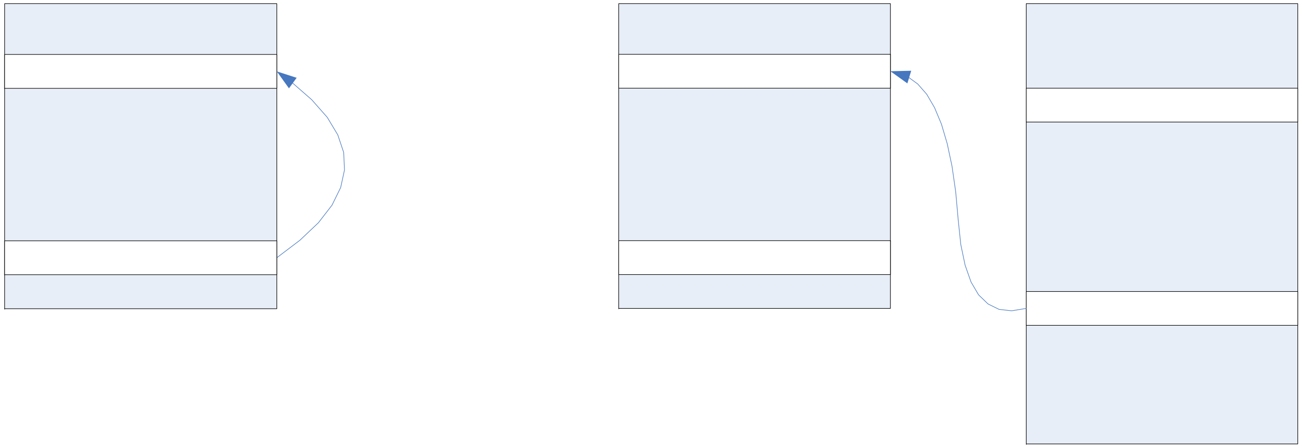
Self-Referencing & Self-Modifying Code

- Self-Referencing Code
  - An application program refers to itself
  - No problem since all load and store addresses mapped to source memory region, not translated region



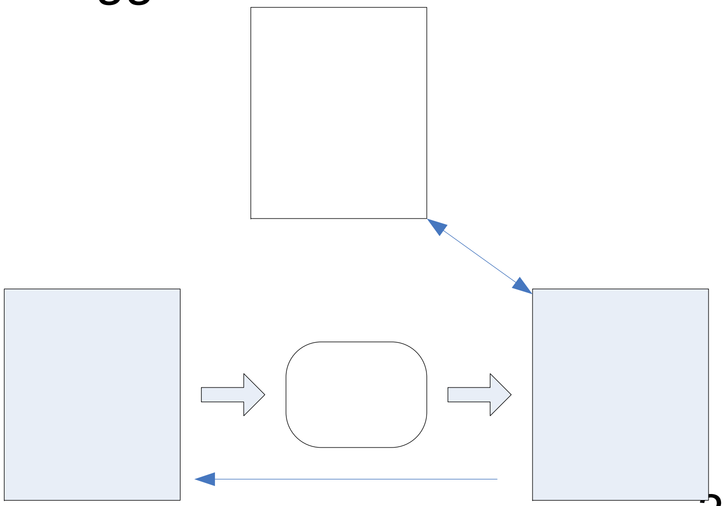
Self-Referencing & Self-Modifying Code

- Self-Modifying
  - An application program attempts to modify itself
  - Causes potential problems when binary translation is used



Self-Modifying Code

- Basic method for handling SMC
  - Original source code region: write-protected (by mprotect())
    - In binary translation, write-protect the region when translating it
  - Write to this region: SIGSEGV trap and a signal is delivered
  - Runtime throws away translated code blocks using a side table
  - Disable the write-protection temporarily
  - *Interpret through the code block that triggered the fault*
    - Really modifies itself now
  - Re-enable the write-protection
  - If the modified block is used again,
    it will be re-translated



SMC Handling Overhead

- This is costly if we discard many, un-related translations
- Fortunately, SMC is rather uncommon
- Some programs include much SMC, though
- Worse if data and code are intermixed
  - Pseudo-SMC: write into code page does not modify code but will trigger write-protection fault
  - How to deal with frequently occurring pseudo-SMC?

Fine-Grain Protection

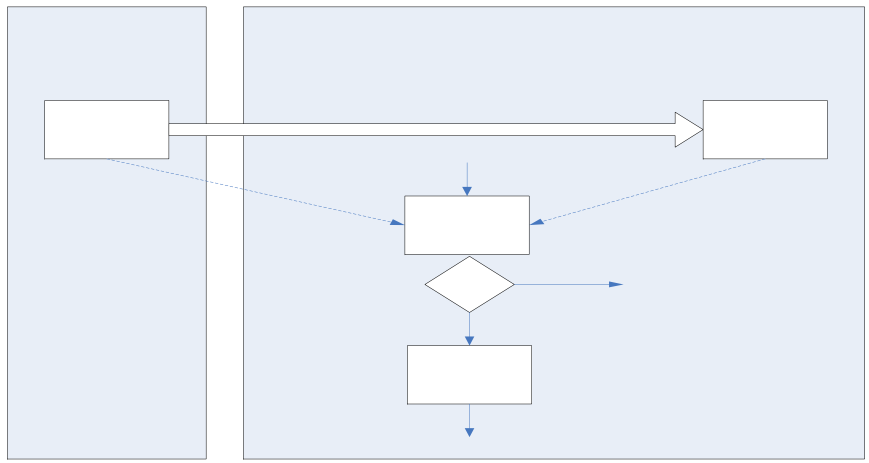
- Provide hardware support for write-protecting memory at granularity finer than full pages
  - EM maintains a finer-granularity protection table with a write-protect bit mask, where a bit corresponds to a small region
  - As code is translated, the bit for the translated region is set
  - If a write is to a data-only region (the bit is not set), no need to flush the translated code
  - Reduce flushing of translations

Table 1: Slowdown Without Fine-Grain Protection

	Faults	Slowdown
Win95 boot	52.8x	2.2x
Win98 boot	59.4x	3.8x
MultimediaMark	46.8x	1.6x
WinStone Corel	54.2x	2.1x
Quake Demo2	7.7x	1.02x

Self-Checking Translations

- Leave the page unprotected, and before the translated code is executed, check if its source code has not been changed



- When all translations forced to be self-checking
  - Code size and molecules executed are increased
  - Optimization: dynamically link/unlink prolog code
    - When there is a write-protect fault, runtime links prolog code and turn off the write protection such that prolog code is executed

Guest Memory

Copy

Source Code

Handling True Self-Modifying Code

- Many PC applications typically rely on SMC
  - E.g., modify the immediate or offset fields in instructions inside an inner loop instead of allocating a register for it

- Perform a specialized translation for common cases

```
label: add  %eax, 0x123456
```

This can be translated into Crusoe code

```
ld   %temp,[label+1]
add  %eax, temp
```

  - At least, no need for retranslation for this code