Process VM II

Prof. Shih-Wei Li

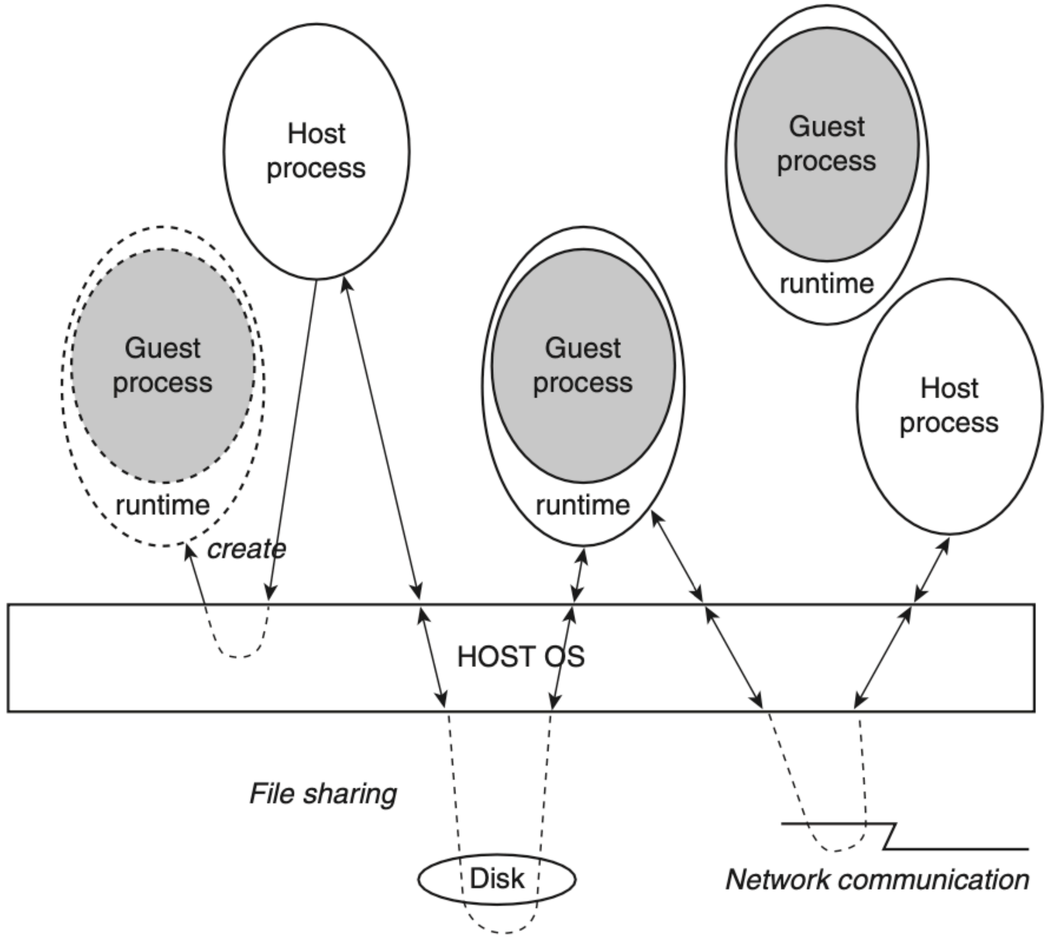
Department of Computer Science and Information Engineering
National Taiwan University

Administration

- The final presentation begins next week
  - Please follow the group order sent out by the TA
  - You will be granted ~30 mins in total, 30 mins (hard limit) for presentation (HINT: you should try to make it shorter than 30 mins) and 5 mins for Q&A
  - This means that if you do not finish in time, we will have to force-cut your talk

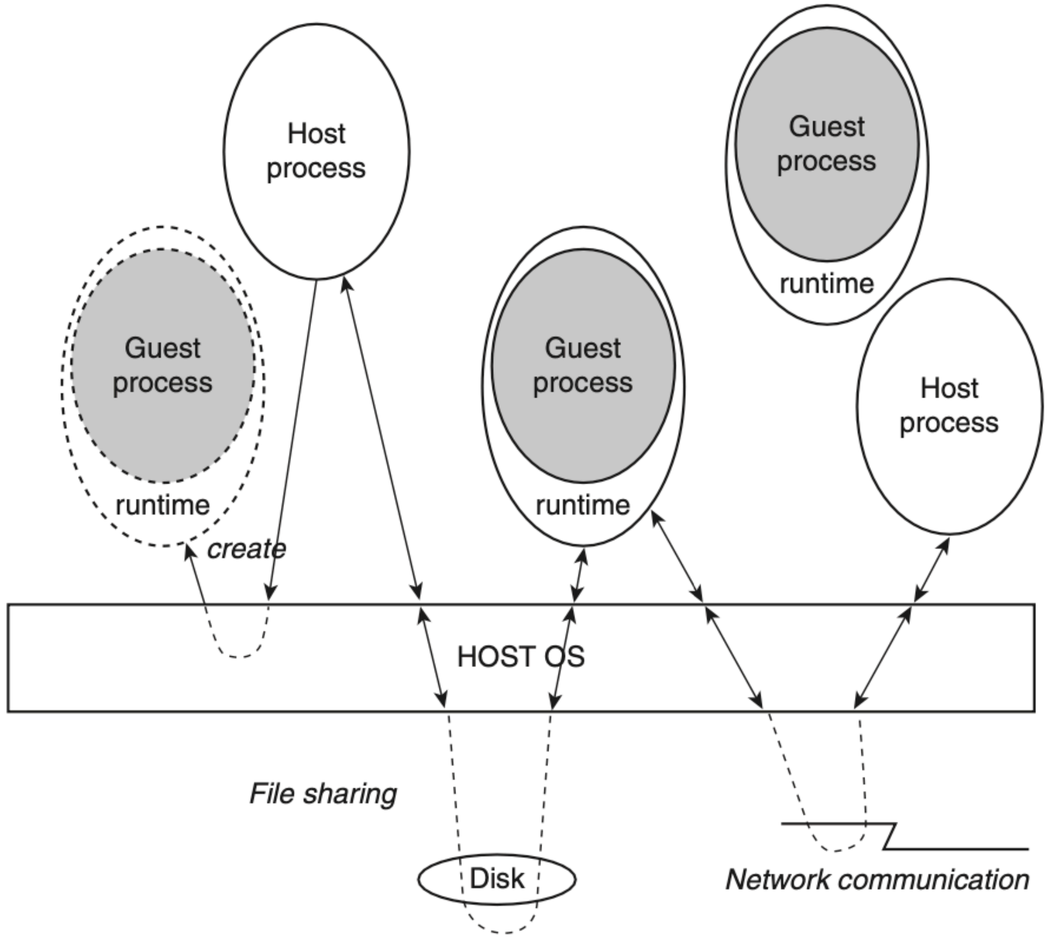
Process Virtual Machines

- Perform guest/host mapping at the ABI (user ISA + system call) level
- The ***runtime*** encapsulate a guest in a process-level, giving it the same outward appearances like other host processes
- Example: Apple Rosetta for Mac OS, Intel IA32-EL/Windows&Linux



Process Virtual Machines

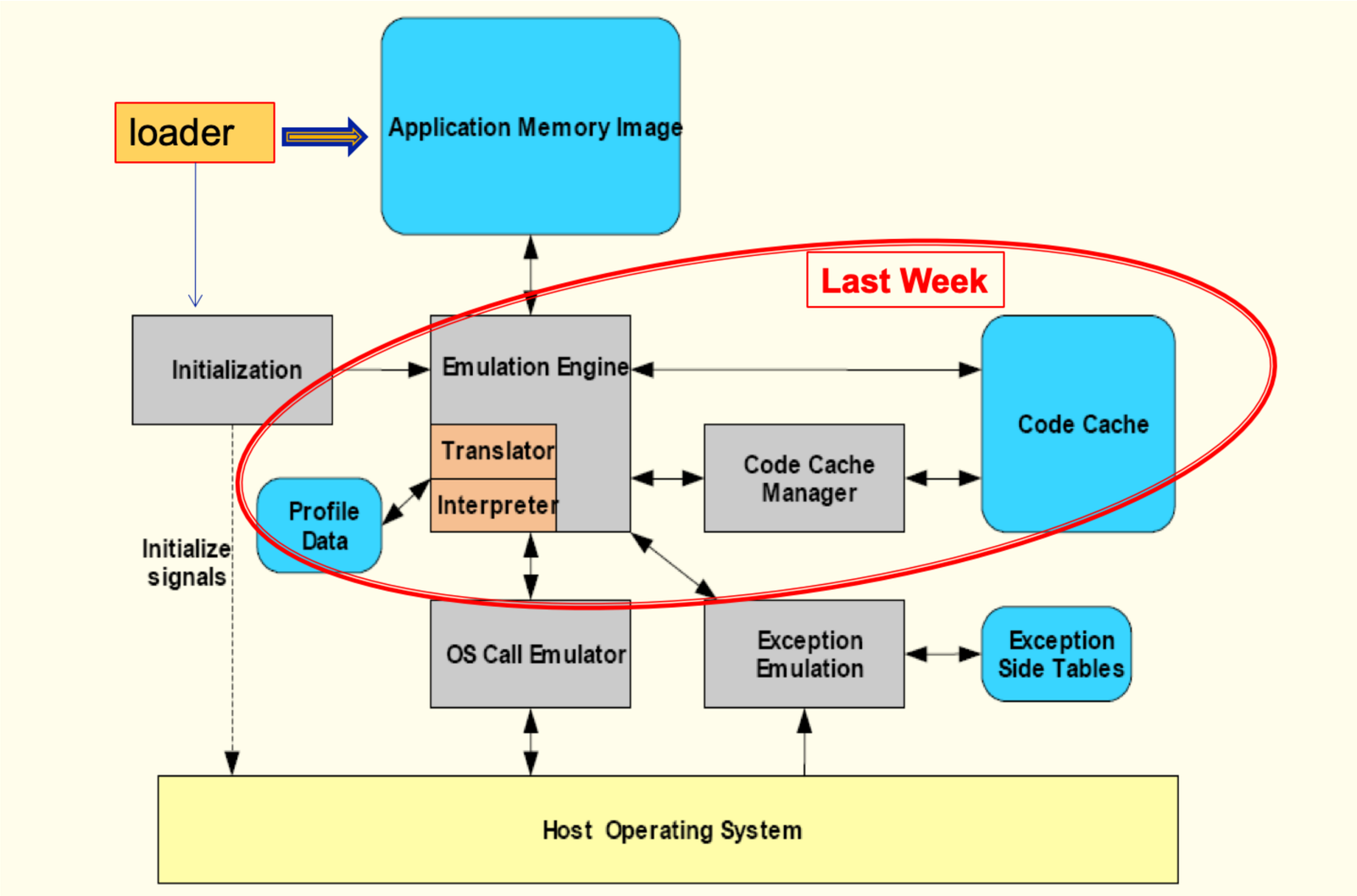
- We talked about two different methods for instruction emulation last week: interpretation and binary translation
- We will discuss this week how to use them to virtualize the ABI interface for supporting process VMs



Agenda

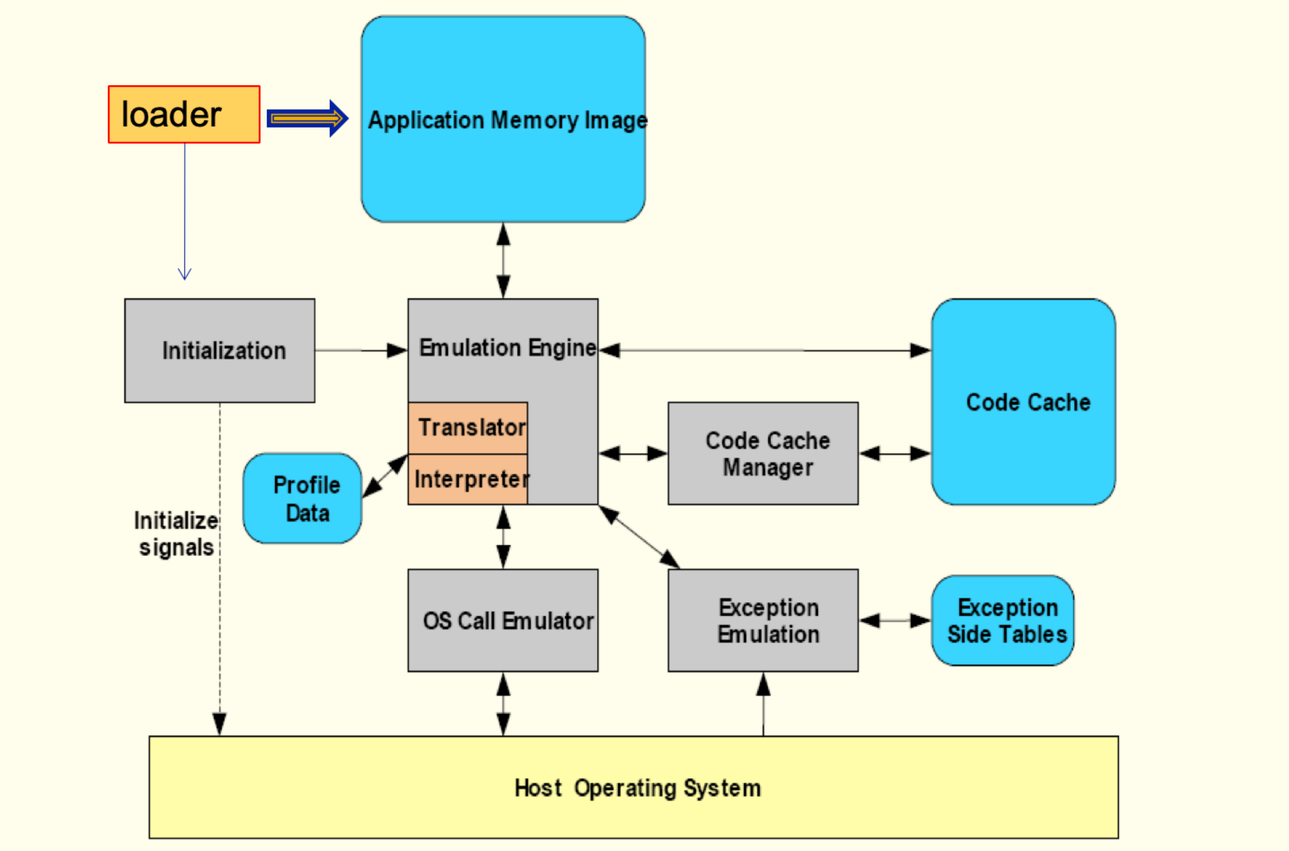
- **Process VM Implementation**
- Compatibility & State Mapping
- Emulation Challenges

Process VM Implementation



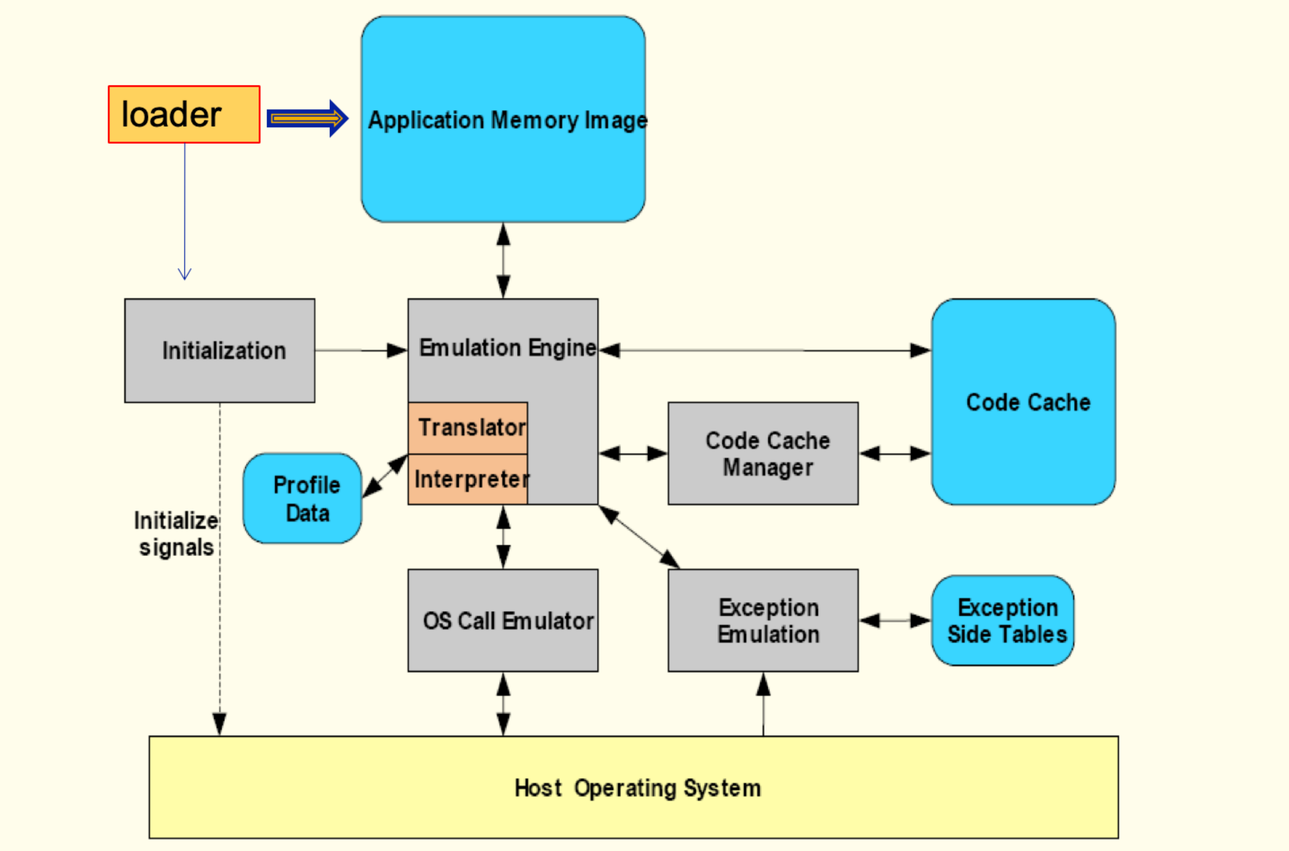
Process VM Implementation

- ***Loader:***
  - Writes the guest code and data into a region of memory and loads the runtime code
- ***Initialization:***
  - Allocates memory for the code cache and other tables
  - Initialize runtime data structures and invoke the host OS to establish signal handlers
- ***Emulation Engine:***
  - Emulate guest instructions with interpreter and/or binary translator



Process VM Implementation

- ***Code Cache Manager:***
  - Decide which code cache (limited size) to flush out in case run out of space
- ***OS Call Emulator:***
  - Translate guest system calls to appropriate call to the host OS
  - Handle host OS system call return
- ***Exception Emulator:***
  - Handle exceptions cause by guest's execution or during interpretation
  - For precise guest state when an exception condition occurs

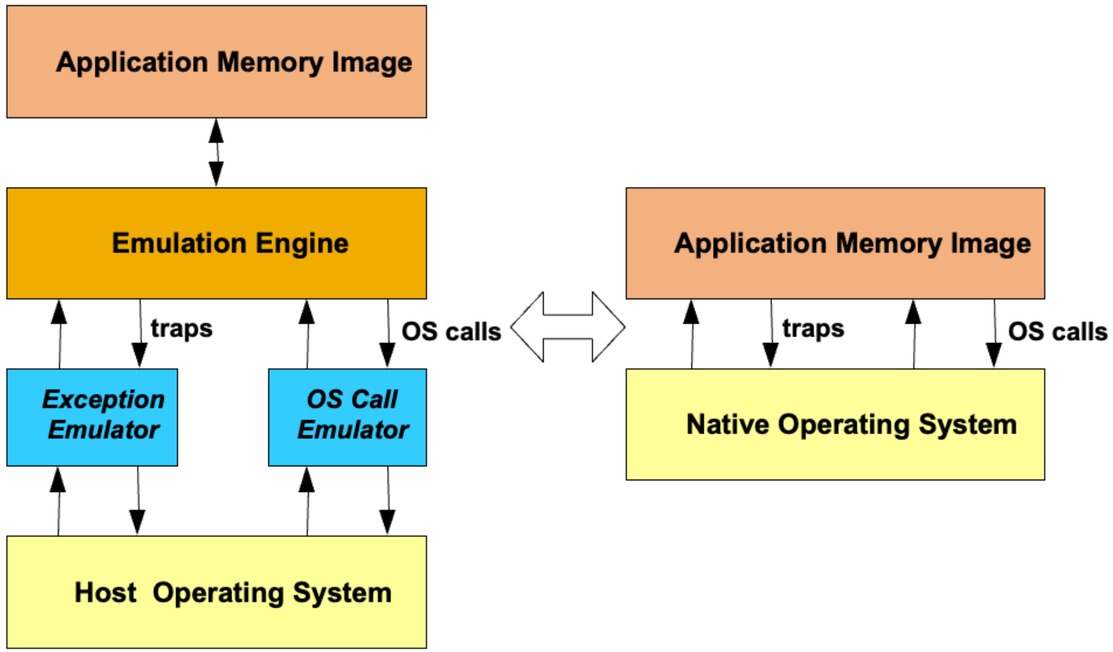


Agenda

- Process VM Implementation
- **Compatibility & State Mapping**
- Emulation Challenges

Process VM Compatibility Framework

- For process VMs: identify correspondence points between the VM and native system at process/instruction control transfer points
- Control transfer points for binary translation systems:
  - Code: branches and jumps
  - Process: exceptions, interrupts, and system calls



State Mapping

- Mapping guest states (registers and memory) to host states: Register and Memory Mapping

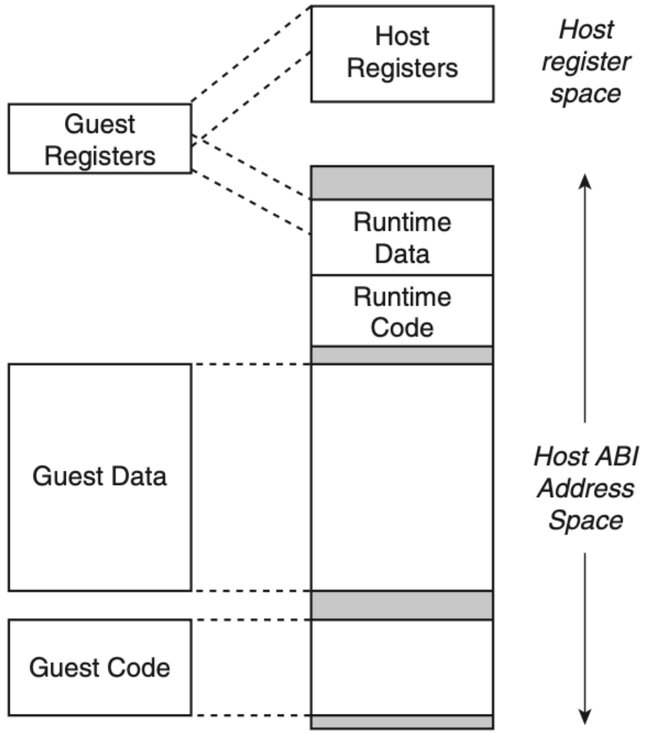


Figure 3.5 Mapping of Guest State to Host State.

Register Mapping

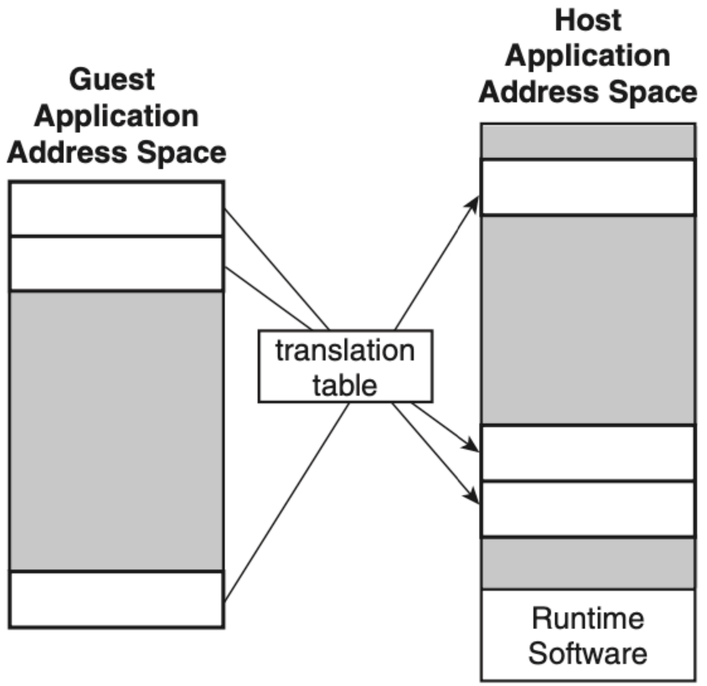
- Essentially the same as discussed last week
- The actual implementation depends on the number of host/guest registers
  - One-to-one guest-host register map is possible if host > guest
  - Use register context block in host's memory if guest > host

Memory Mapping

- The runtime maps the guest's address space to the host's address space and maintain protection requirements
  - An guest address A maps to a host address B
  - A permission P for a guest address maps to P' on the host

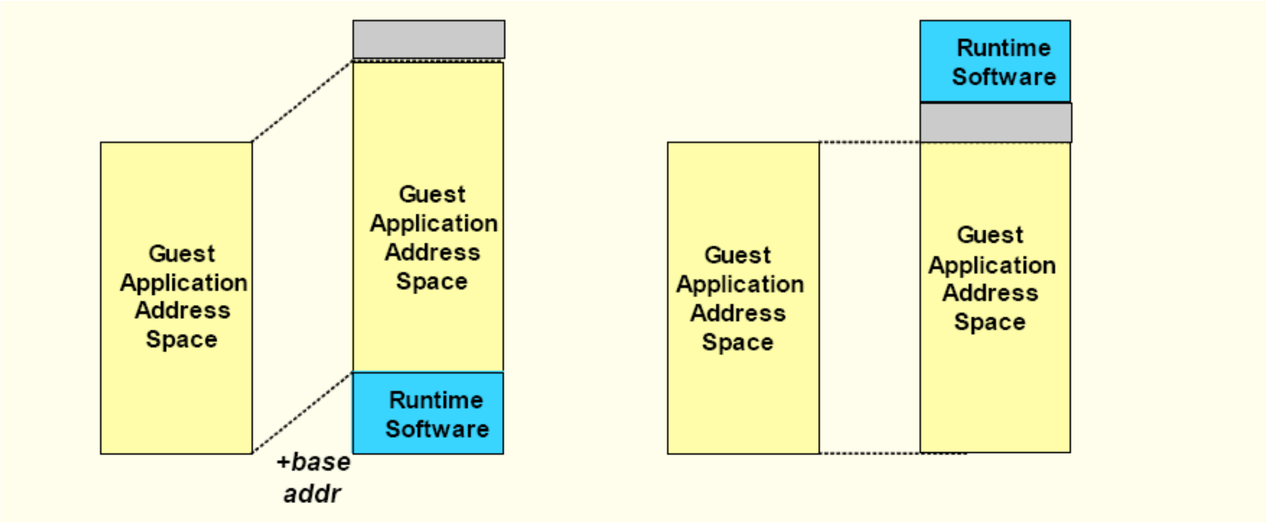
Memory Mapping

- First approach: Runtime software to maintain a software translation table
  - Similar to hardware page table/TLB
- Addresses used by source memory load/ store instructions must be translated in software using the table— relatively slow!
- Works for instruction interpretation and binary translation-based framework



Memory Mapping

- Second approach: Direct Translation Methods
- Addresses used by source memory load/ store instructions can be translated to a target address easily (no table lookups)
- Fit guess application and runtime within the host address space

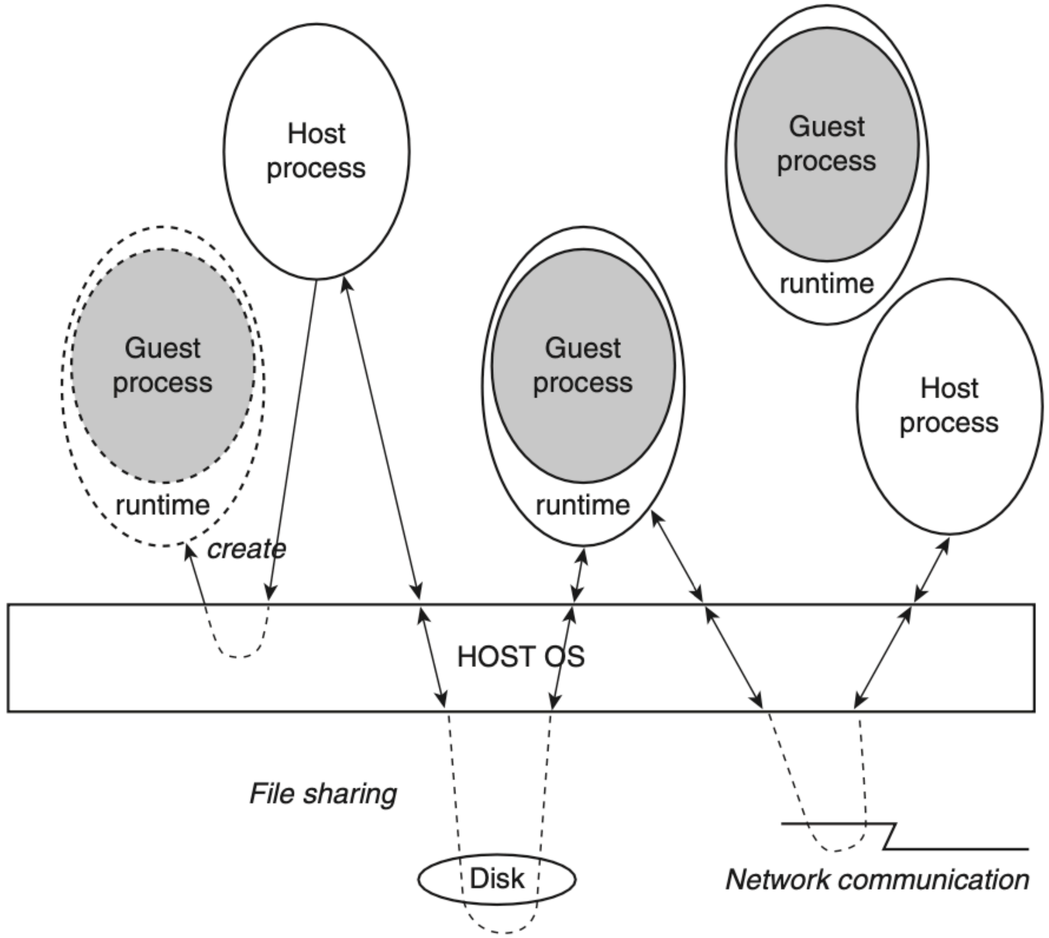


Agenda

- Process VM Implementation
- Compatibility & State Mapping
- **Emulation Challenges**

Emulation Issues for Process VMs

- ***Memory architecture***
- Instruction
- Exception architecture
- OS call emulation



Memory Architecture Emulation

- The guests and host instruction sets have their specific memory architectures
- For a given memory mapping method (direct or table), 3 important aspects of memory architectures must be considered for supporting process VMs
  - **The overall structure of the address space**
    - Segmented or flat memory
  - **The access privilege types that are supported**
    - Read/Write/Executable
  - **The protection/allocation granularity**
    - Size of the smallest block of memory for allocation and the granularity for protection

Memory Architecture Emulation: Host-Supported Memory Protection

- Linux's system call "*memprotect*" and the signal "SIGSEGV" can be used to fulfill the requirements

mprotect(2) — Linux manual page

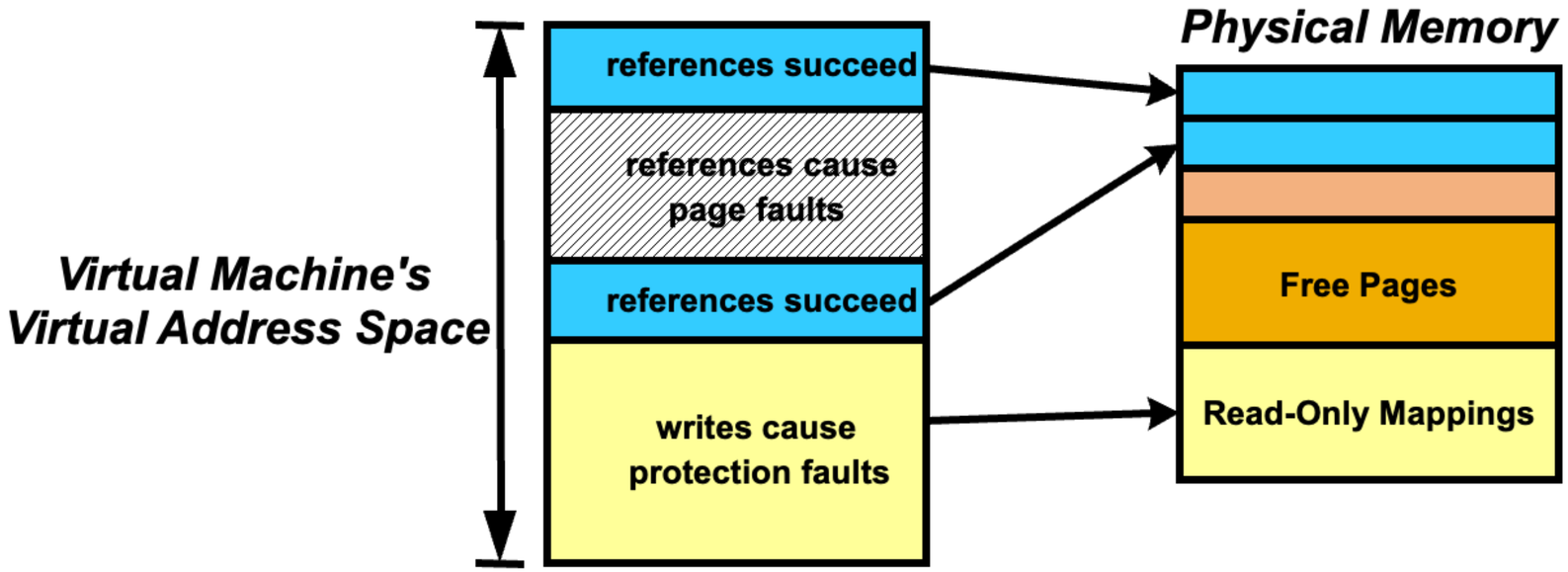
```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

DESCRIPTION top

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [*addr*, *addr*+*len*−1]. *addr* must be aligned to a page boundary.

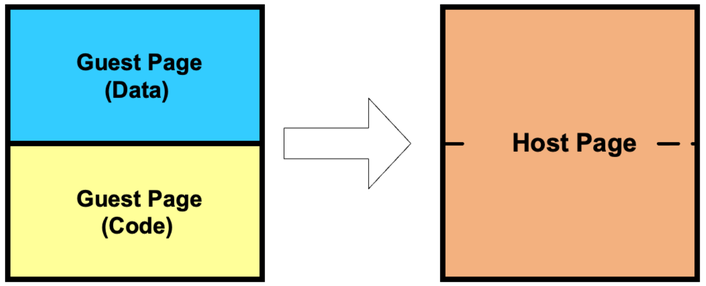
If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a **SIGSEGV** signal for the process.

Memory Architecture Emulation: Host-Supported Memory Protection



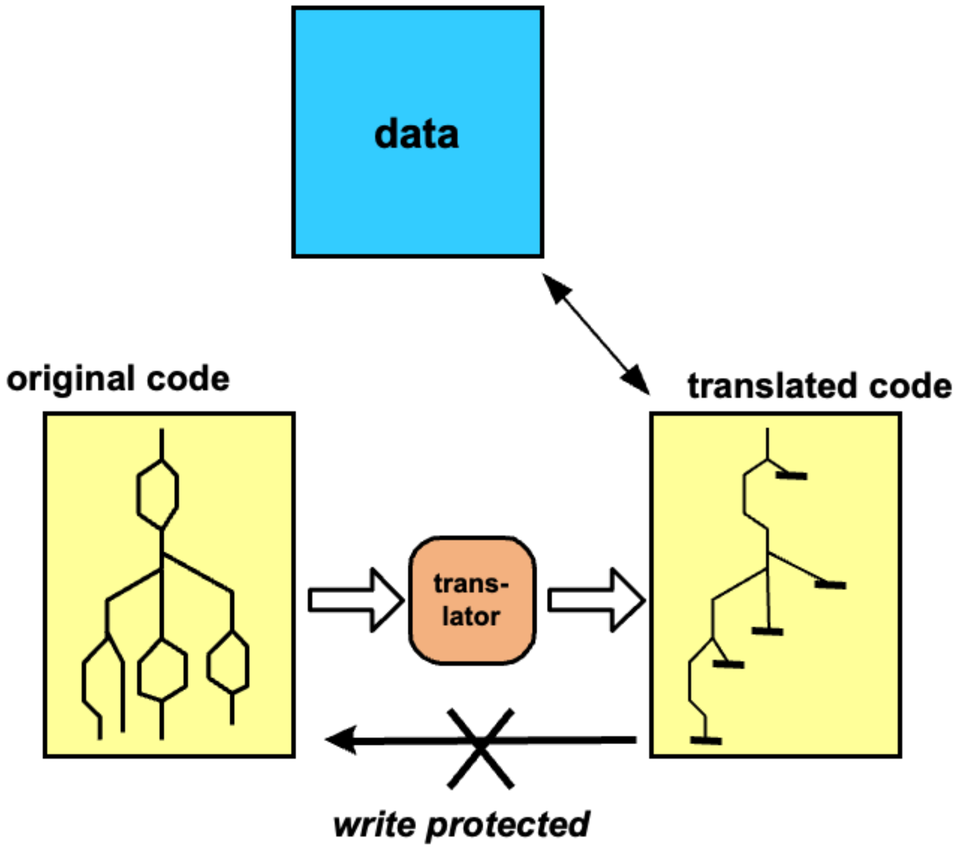
Memory Architecture Emulation: Page Size Issues

- Host and guest pages could be of different sizes
  - Cause issues when allocating and protecting memory
- Consider two cases:
  - Guest page size is larger than the host: Assign the same protection to the multiple host pages allocated to back up a certain guest page
  - Guest page size is smaller than the host: Many guest pages are included in a single host page; any issues?



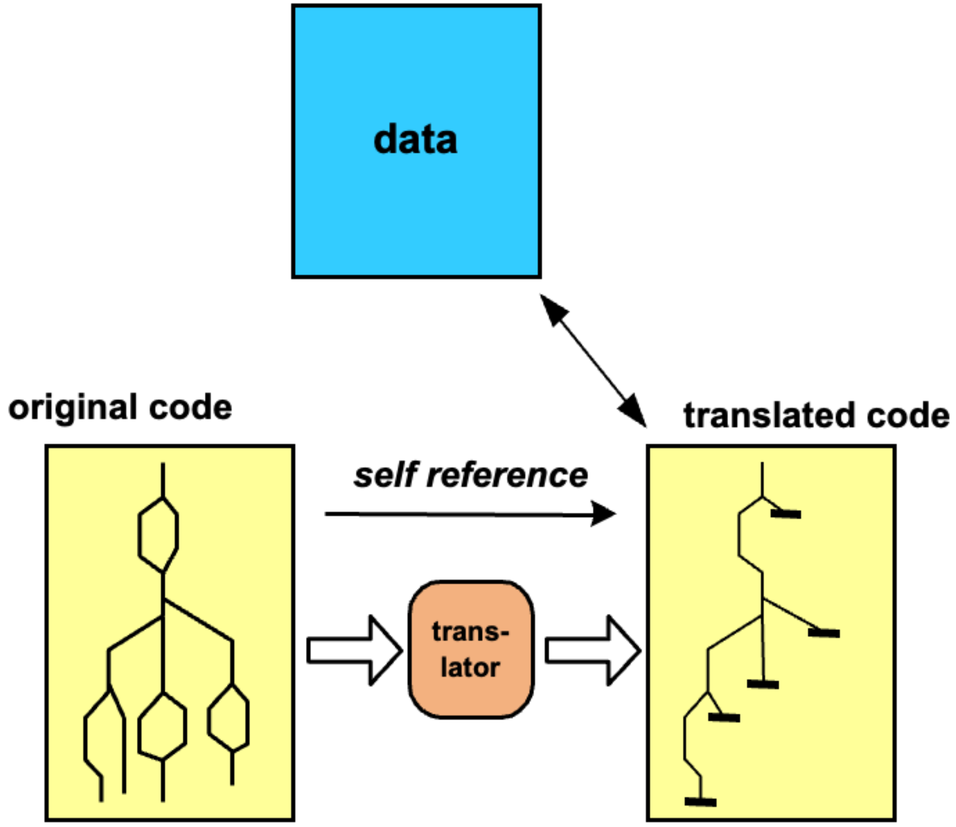
Memory Architecture Emulation: Self-Modifying Code

- Write-Protect the original source code to catch code self-modification
  - Previously translated code is invalidated
  - New code is translated and re-generated



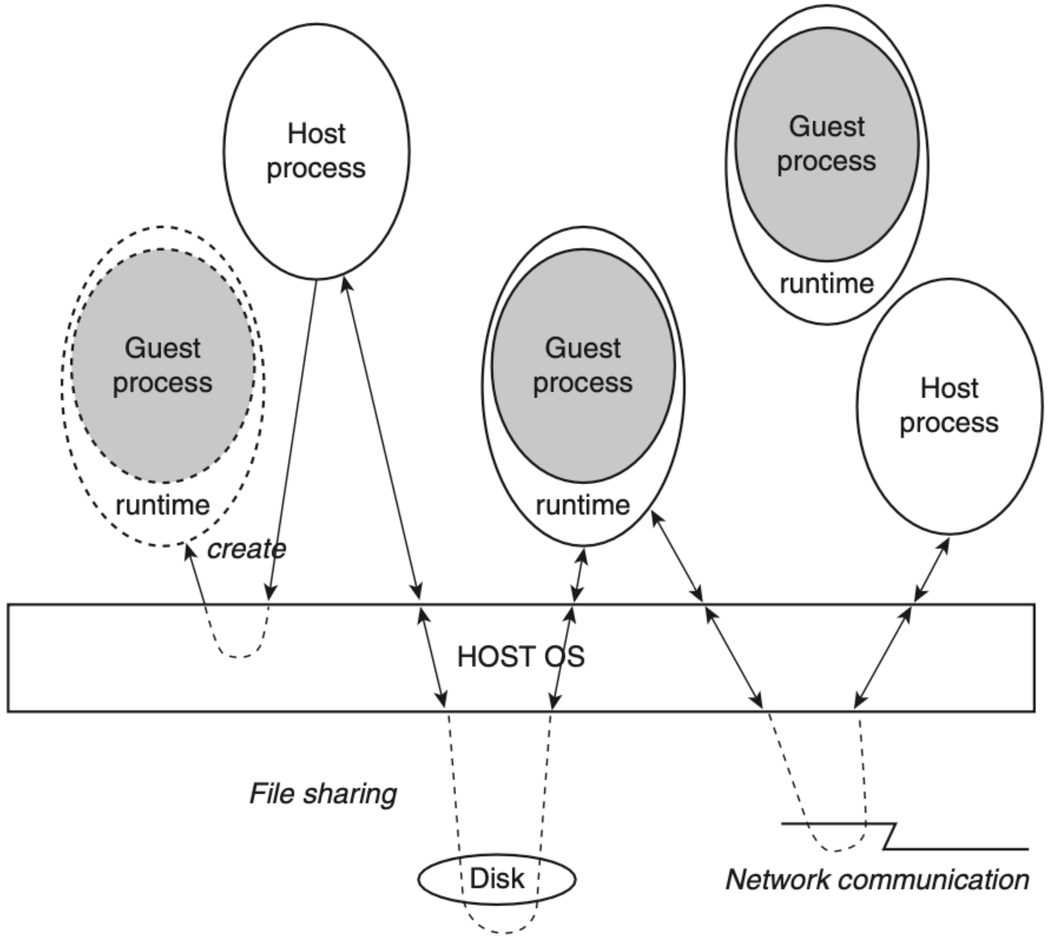
Memory Architecture Emulation: Self-Referencing Code

- Problem: the original code will reference itself — can do wrong if reference to the translated code
  - Direct the reference to the original code
- Translator maintains the original code
  - For self-references, map the load and store “addresses” in the translated version to the source memory region



Emulation Issues for Process VMs

- Memory architecture
- ***Instruction***
- Exception architecture
- OS call emulation

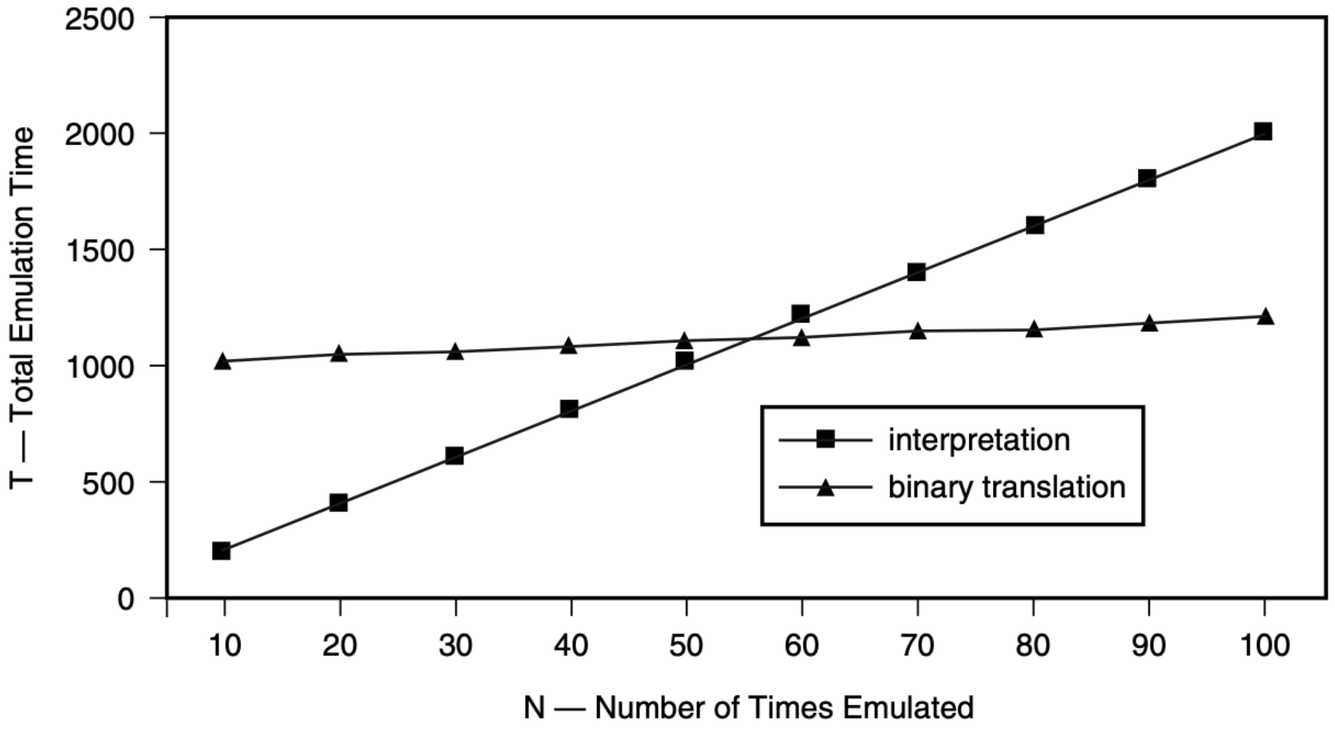


Instruction Emulation

- Important performance tradeoffs
  - Start-up time: Cost of converting code in one form to code in a different form prior to emulation
  - Steady-state performance: Cost of emulation (the time required per emulated instruction)
- Different performance tradeoffs in different way of instruction emulation
  - Interpretation: Low start-up cost, high steady-state cost
  - Binary Translation: High start-up cost, low steady-state cost

Instruction Emulation

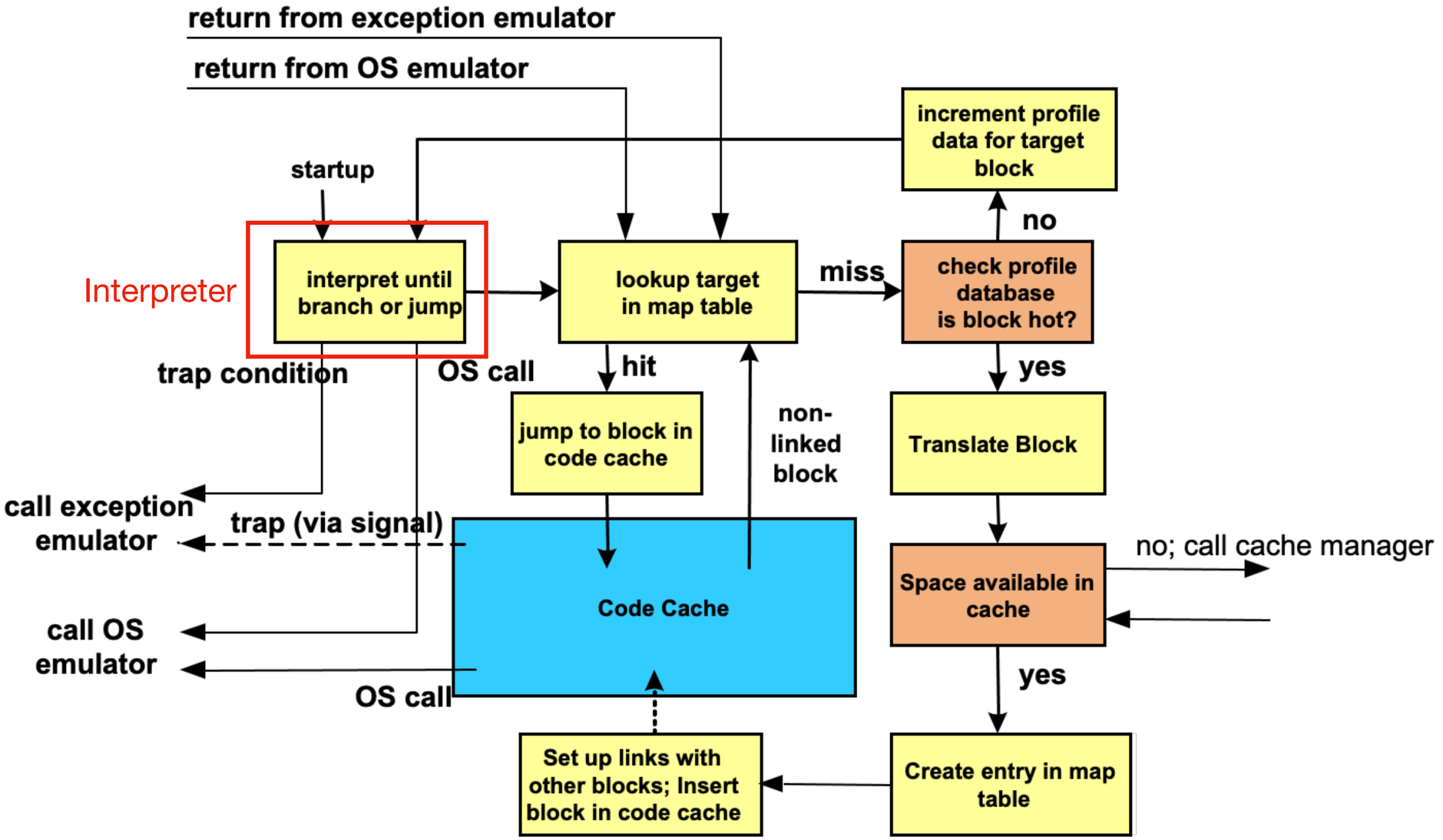
The overall time for emulating an instruction N times is expressed as $S + NT$, where S is the one-time start-up overhead for the given instruction and T is the time required per emulation in steady state. The critical perfor-



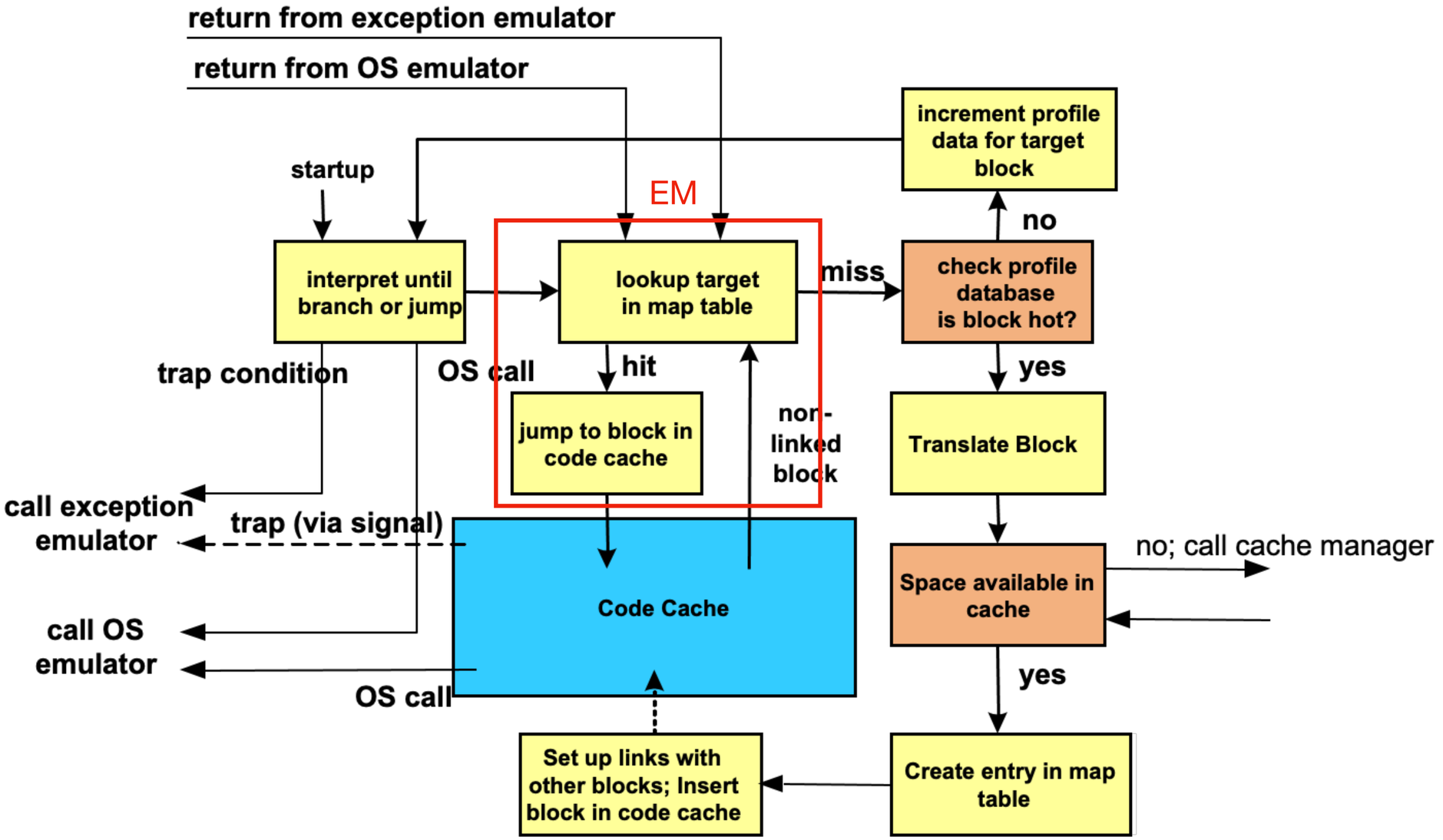
Staged Emulation

- Hard to predict in advance how many times instructions in the program will be emulated — cannot quickly tell what the optimal method is
  - Adjust optimization level to execution frequency
- General Strategy:
  - 1. Begin interpreting
  - 2. Collect profile data for the number of times a code block is executed
  - 3. Invokes binary translation to translate code if the time the code block is executed is above a threshold
  - 4. Optimize more for translated code executed above a higher threshold

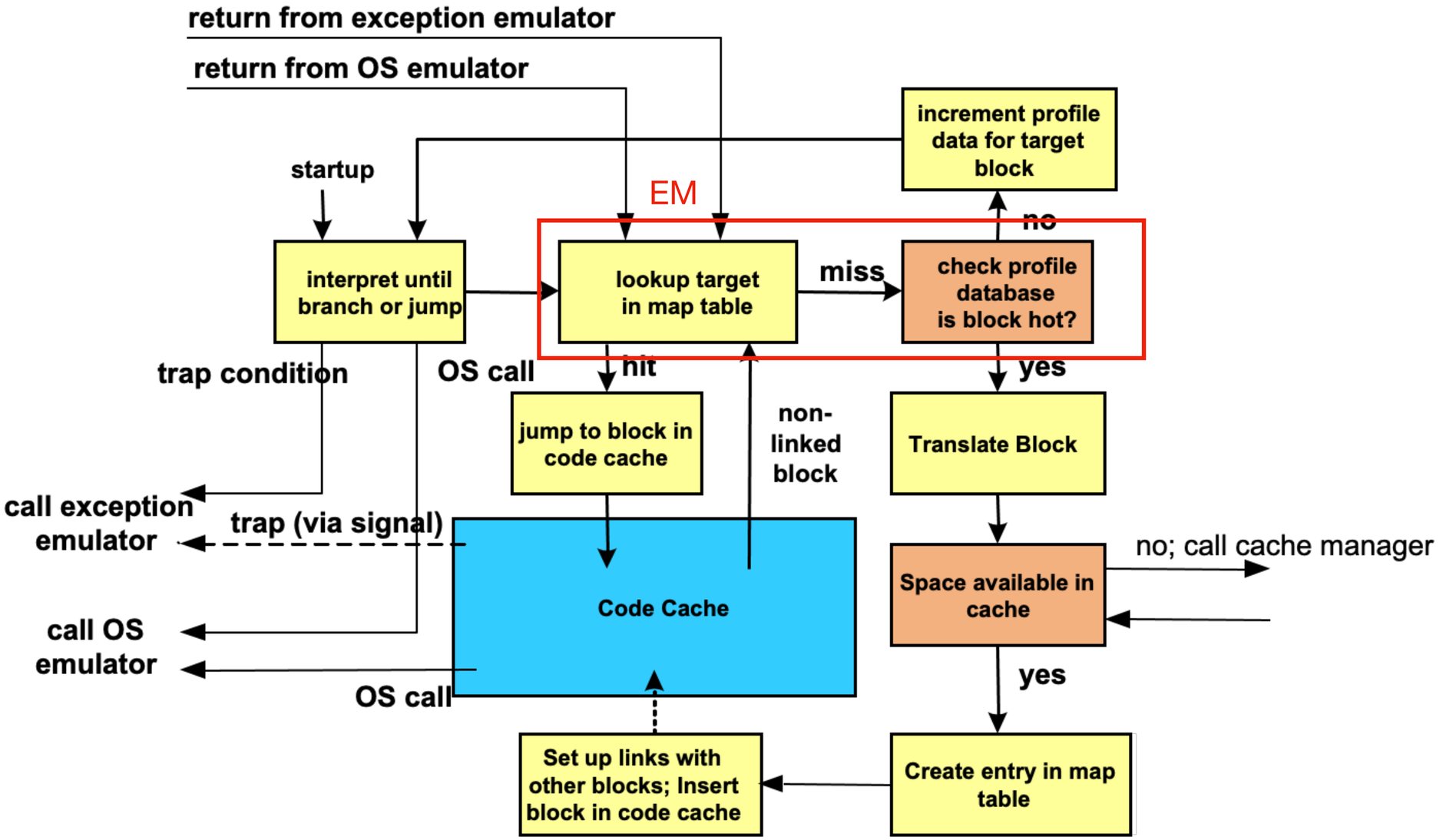
Staged Emulation



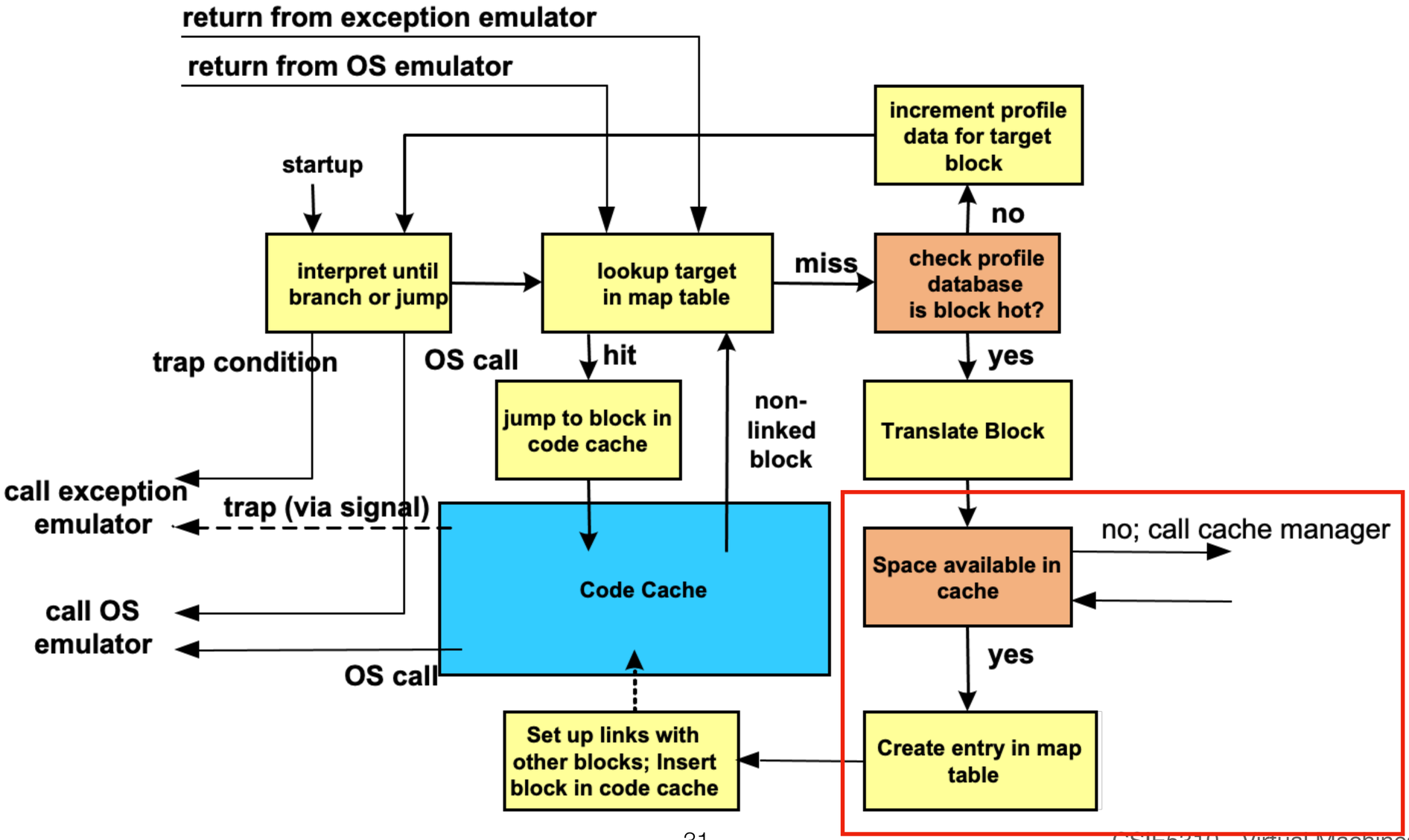
Staged Emulation



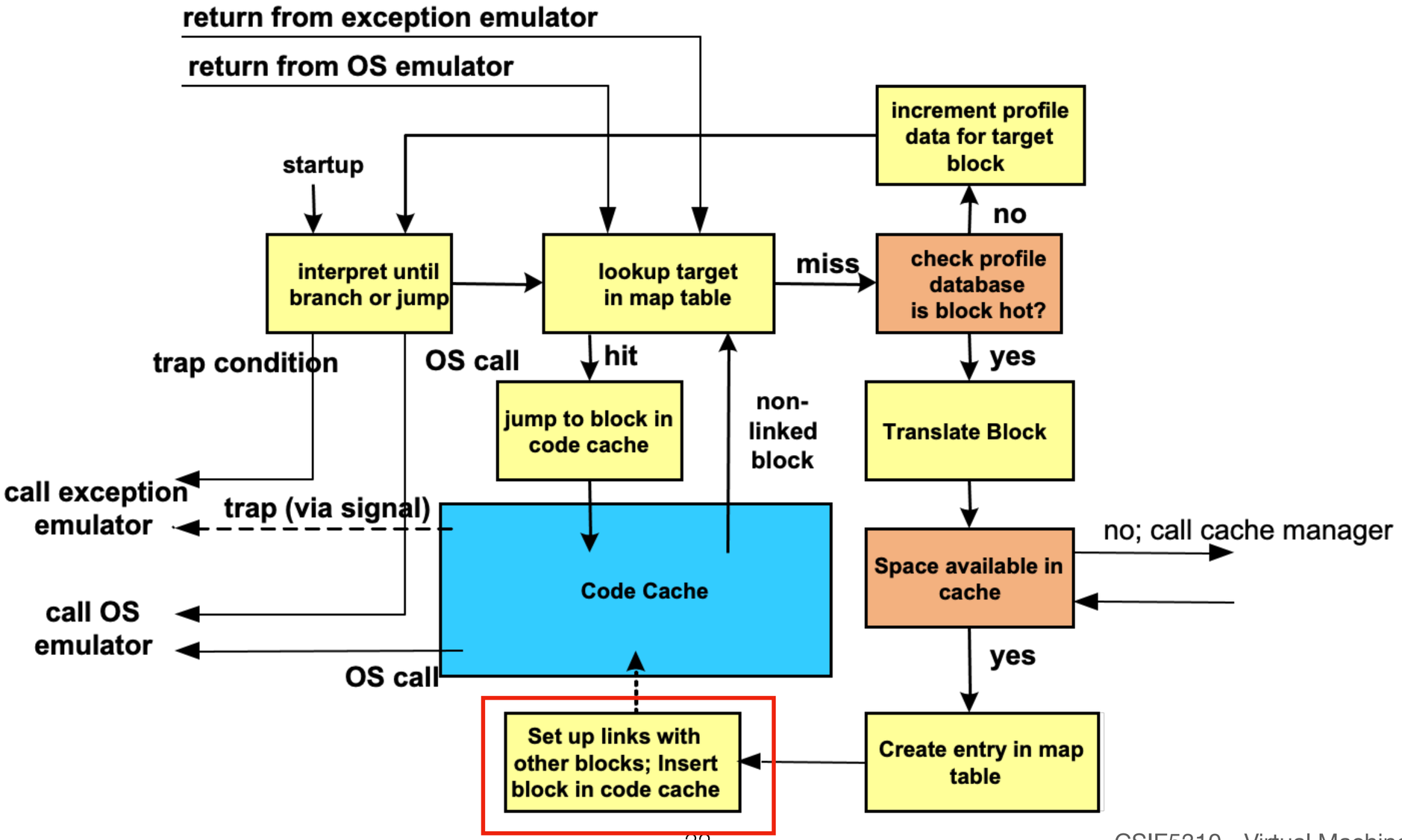
Staged Emulation



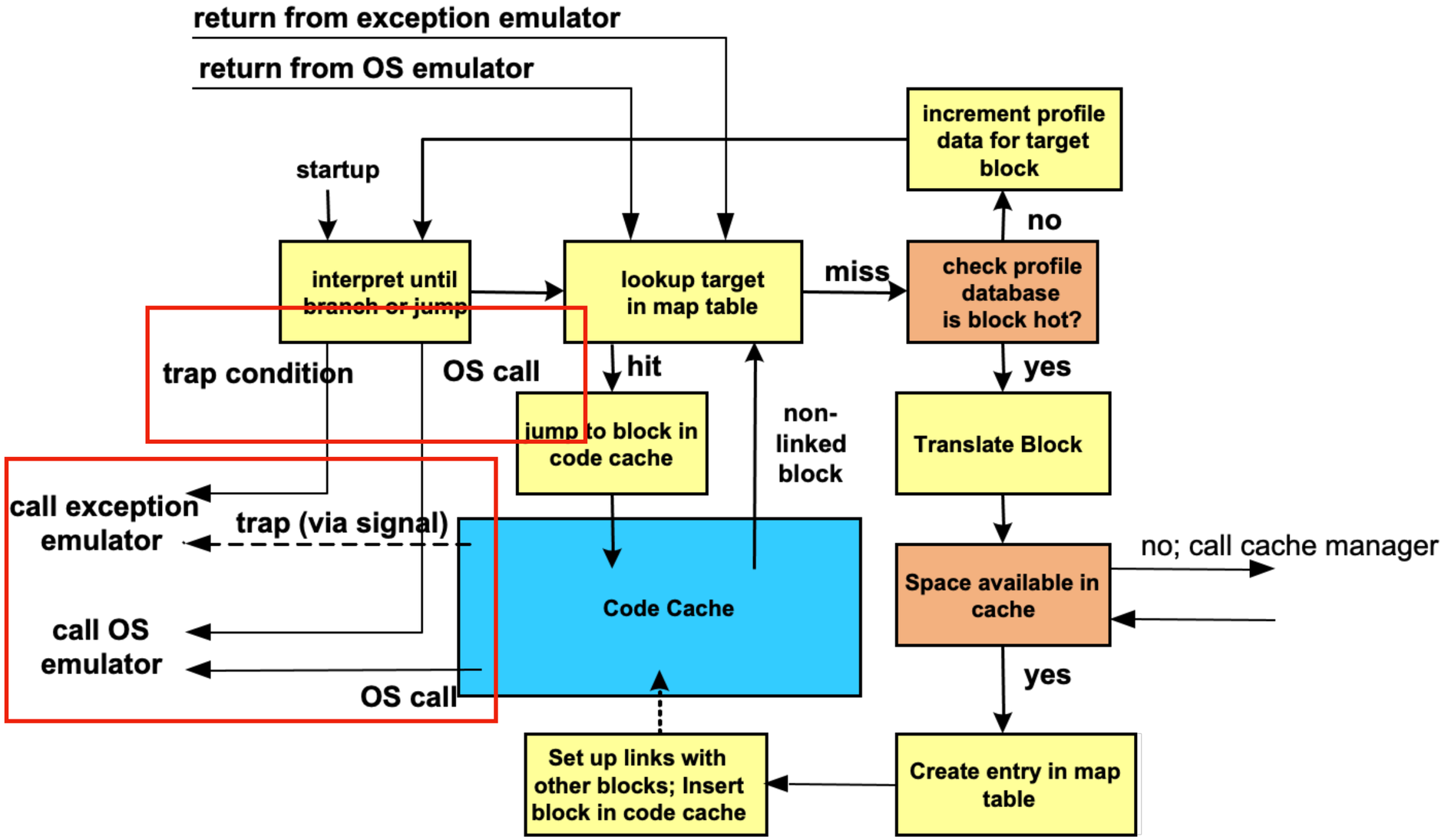
Staged Emulation



Staged Emulation

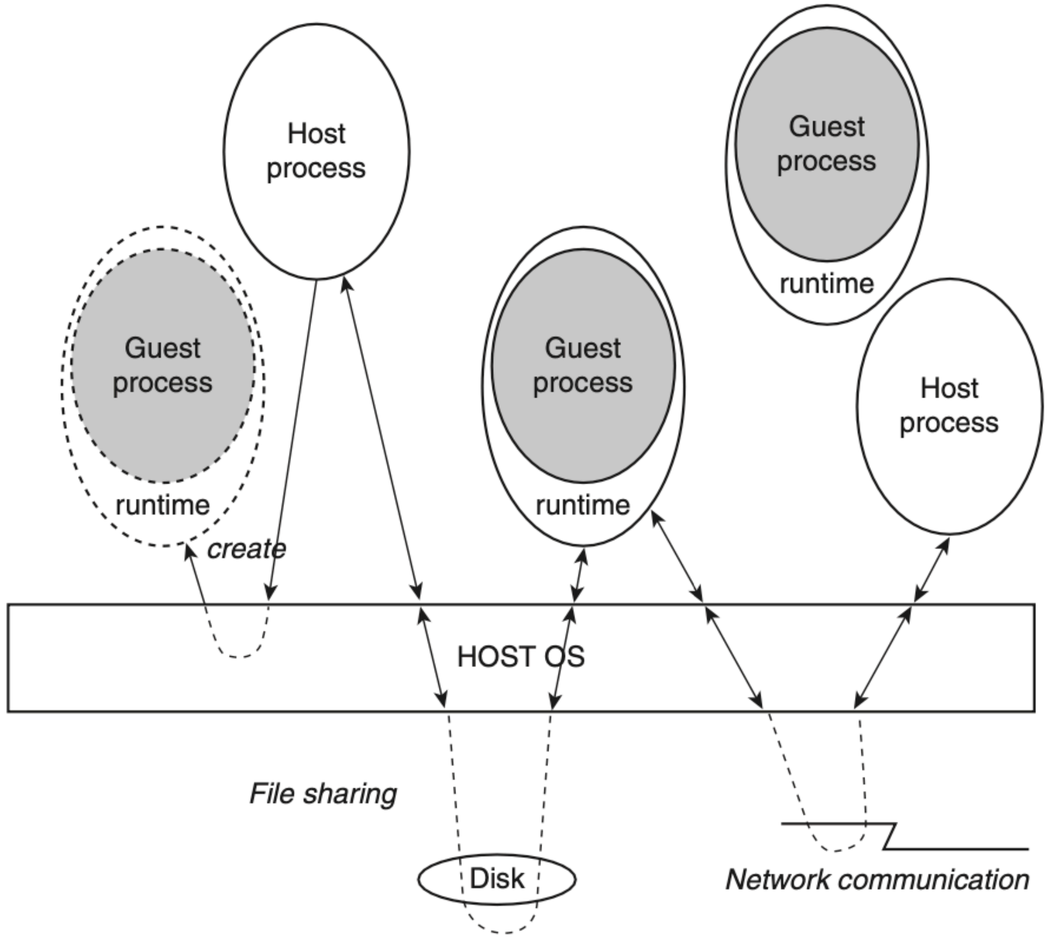


Staged Emulation



Emulation Issues for Process VMs

- Memory architecture
- Instruction
- ***Exception architecture***
- OS call emulation



Exceptions Emulation

- The runtime has to correctly emulate exceptions for the process VM
- Exceptions to consider for a process VM implementation in 2 categories:
  - ABI visible: include all exceptions that are returned to the application via OS signal (e.g. a signal for memory protection fault, exceptions that cause the application to terminate)
  - ABI invisible: include all exceptions that the application is unaware of (e.g. timer interrupts)

Exceptions Emulation: Exception Detection

- Traps caused by instruction execution are difficult to deal with
  - Have to find the exception PC (in SPC) and restore exception states
- Exceptions in process VMs can be detected in two ways:
  - *Detect during instruction interpretation* — slow but easier to implement
  - *Detect by host platform hardware when an instruction is executed* — efficient, but depend on the semantic match between the source and target ISAs

Exceptions Emulation: Determining Precise Guest State

- After an exception condition has been discovered, the runtime must be able to provide the proper state for the guest
  - Interpretation: easy and straightforward
  - Binary translation: complex, has to map target states to source states

Exceptions Emulation: Determining Precise Guest State

- Interpretation for Add instruction

```
Add:
        RT = extract(inst,25,5);
        RA = extract(inst,20,5);
        RB = extract(inst,15,5);
        source1 = regs[RA];
        source2 = regs[RB];
        sum = source1 + source2;
        regs[RT] = sum;
        PC = PC + 4;
        If (halt || interrupt) goto exit;
        inst = code[PC];
        opcode = extract(inst,31,6);
        extended_opcode = extract(inst,10,10);
        routine = dispatch[opcode, extended_opcode];
        goto *routine;
```

Add potentially causes overflows, jumps to the signal handler of the runtime

Exits to the runtime to switch execution to the source PC of the guest signal handler

Exceptions Emulation: Determining Precise Guest State

- For binary translation systems, source PC is only available at translation block boundaries
  - Trap PC is in terms of target machine code
  - Trap PC must be mapped back to correct source PC
- Solution: keep PC related information in a *reverse translation side table*

Exceptions Emulation: Determining Precise Guest State

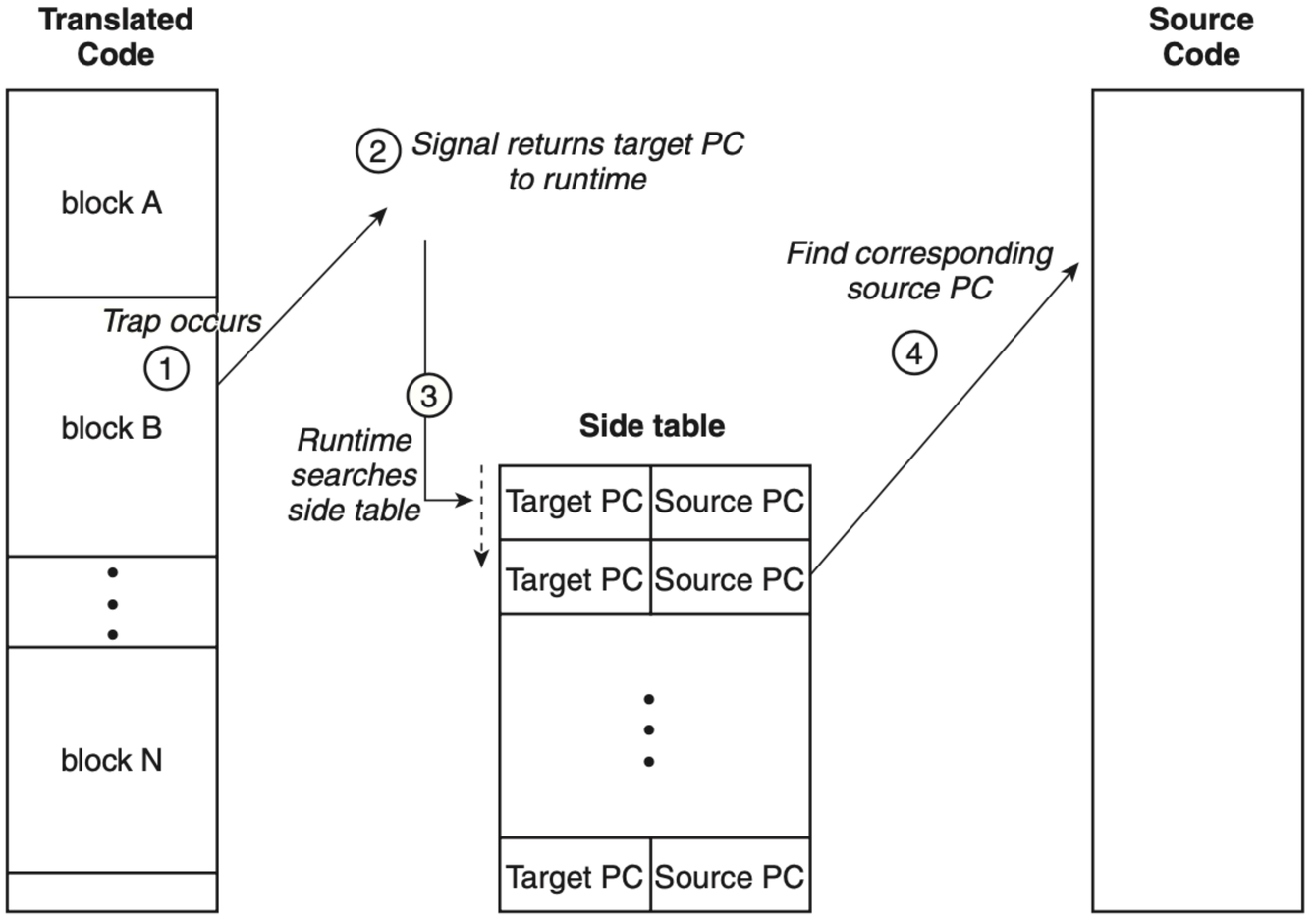
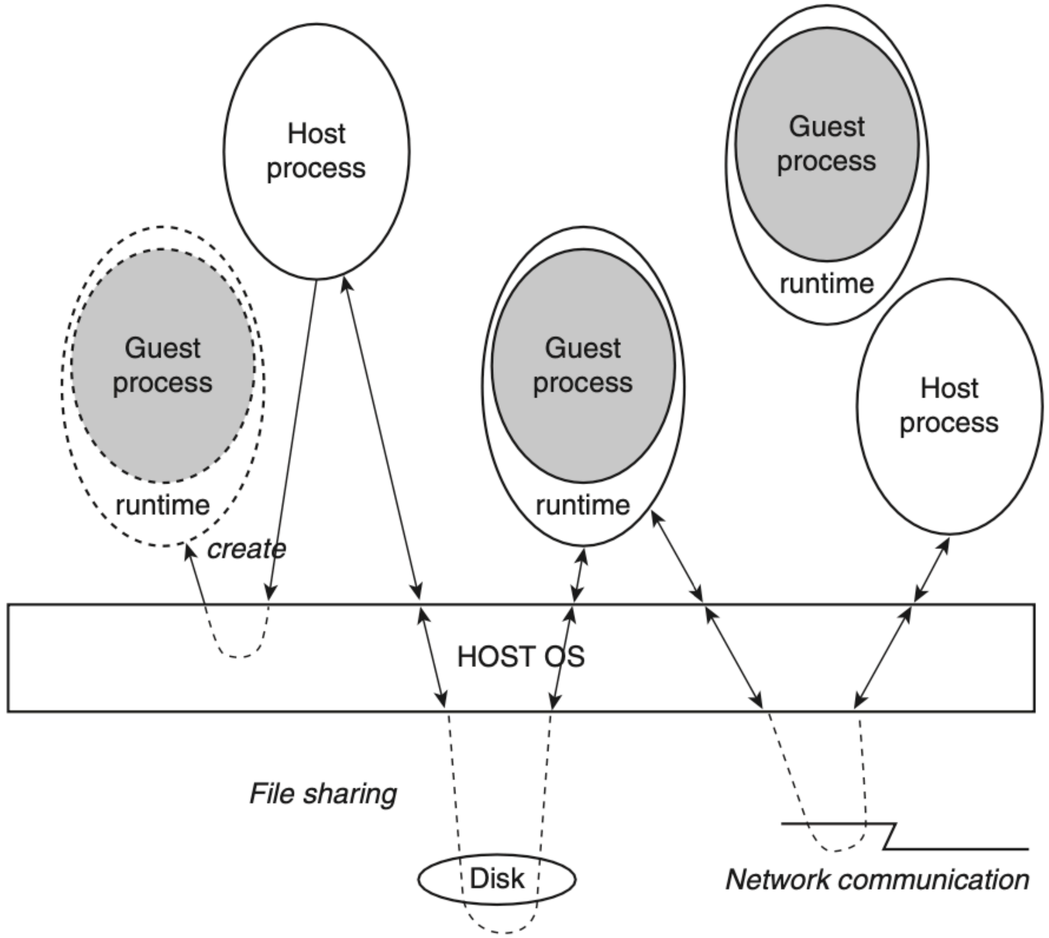


Figure 3.21 Finding the Trapping Source PC, Given the Target PC. *(1) The trap occurs and (2) the signal handler returns the target PC to the runtime software. The runtime (3) does a search of the side table (4) to find the corresponding source PC that caused the trap.*

Emulation Issues for Process VMs

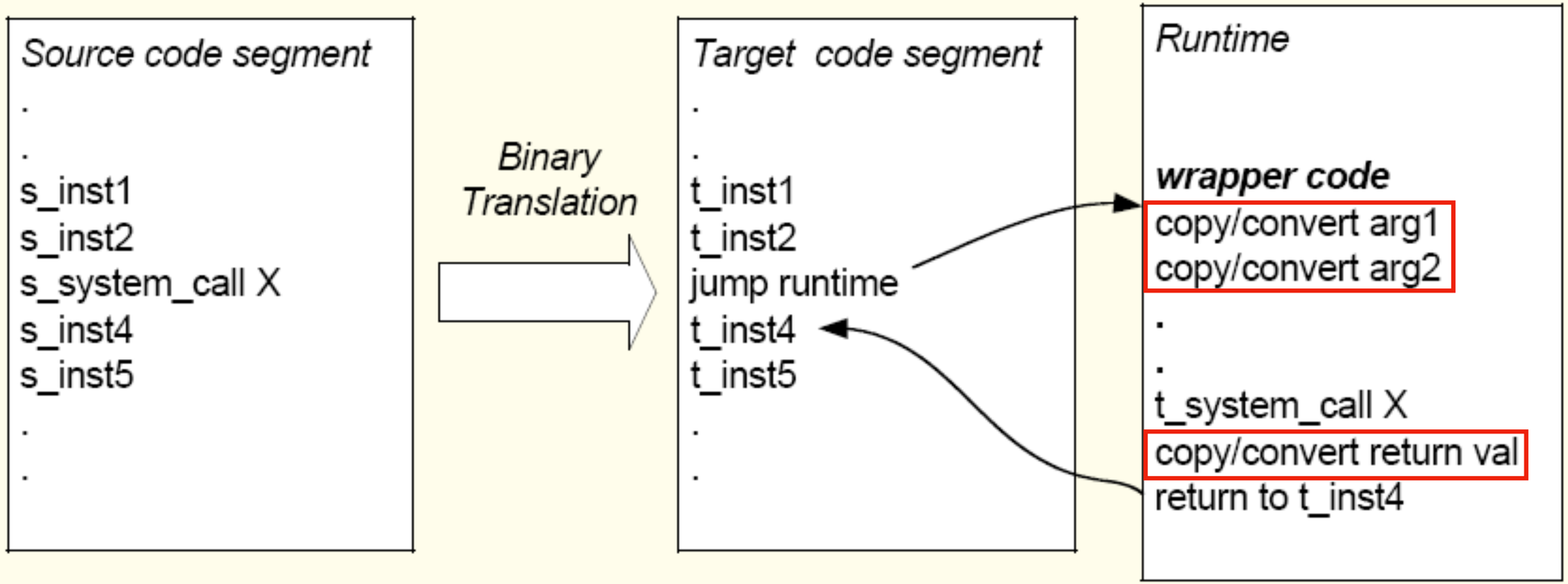
- Memory architecture
- Instruction
- Exception architecture
- ***OS call emulation***



OS Emulation

- The operating system interface is a key part of the ABI specification
- Process VM is required to maintain the compatibility at the ABI level
  - It does not emulate the individual instructions in the guest's OS code but it emulates the functionality of guest OS (system) calls
  - Typically done by converting guest OS calls to the host OS
- Two cases to be considered:
  - Same OS emulation
  - Different OS emulation

Same OS Emulation



Different OS Emulation

- Different OS emulation is very much an *ad hoc* process
  - A single guest OS call may require multiple host OS calls for emulation
  - Some of the guest OS calls are emulated by the runtime

Different OS Emulation

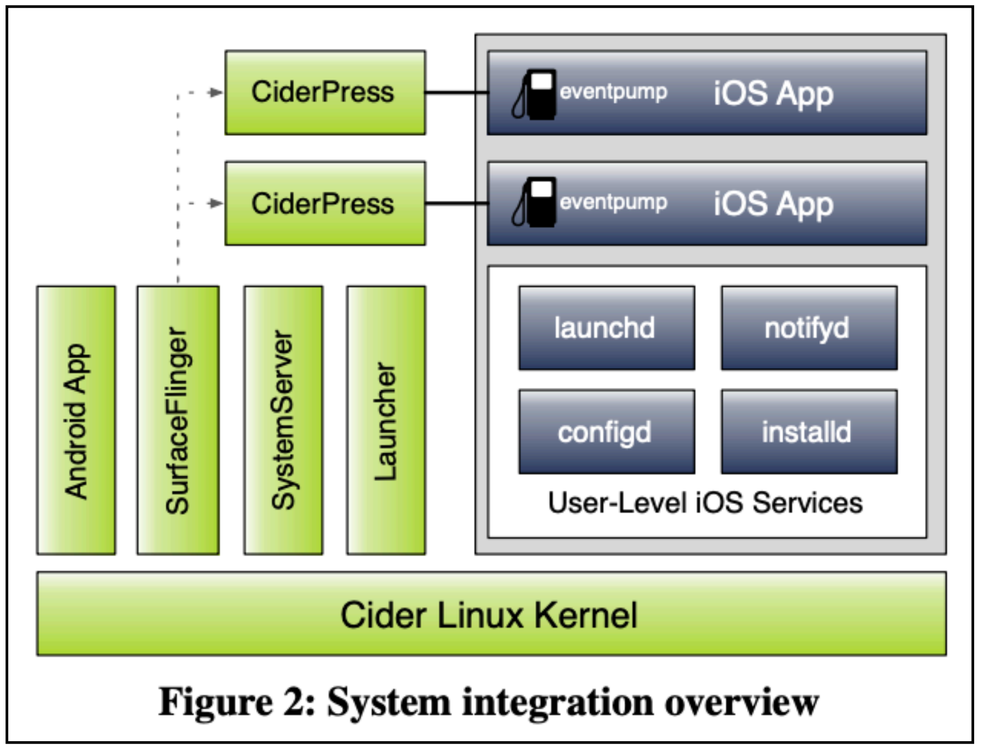


Figure 2: System integration overview

https://dl.acm.org/doi/10.1145/2644865.2541972