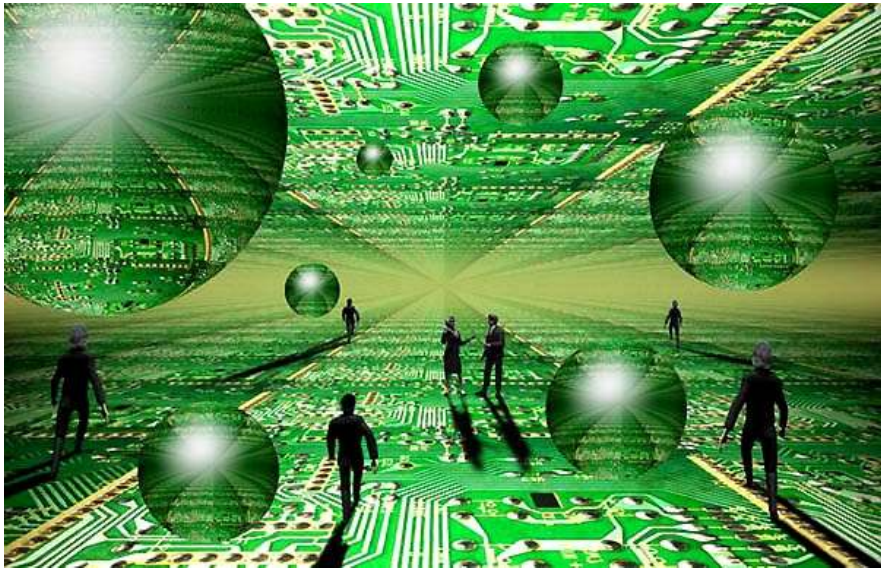
CSCI-GA.3033-015

Virtual Machines: Concepts & Applications

Lecture 3: Process VM – I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

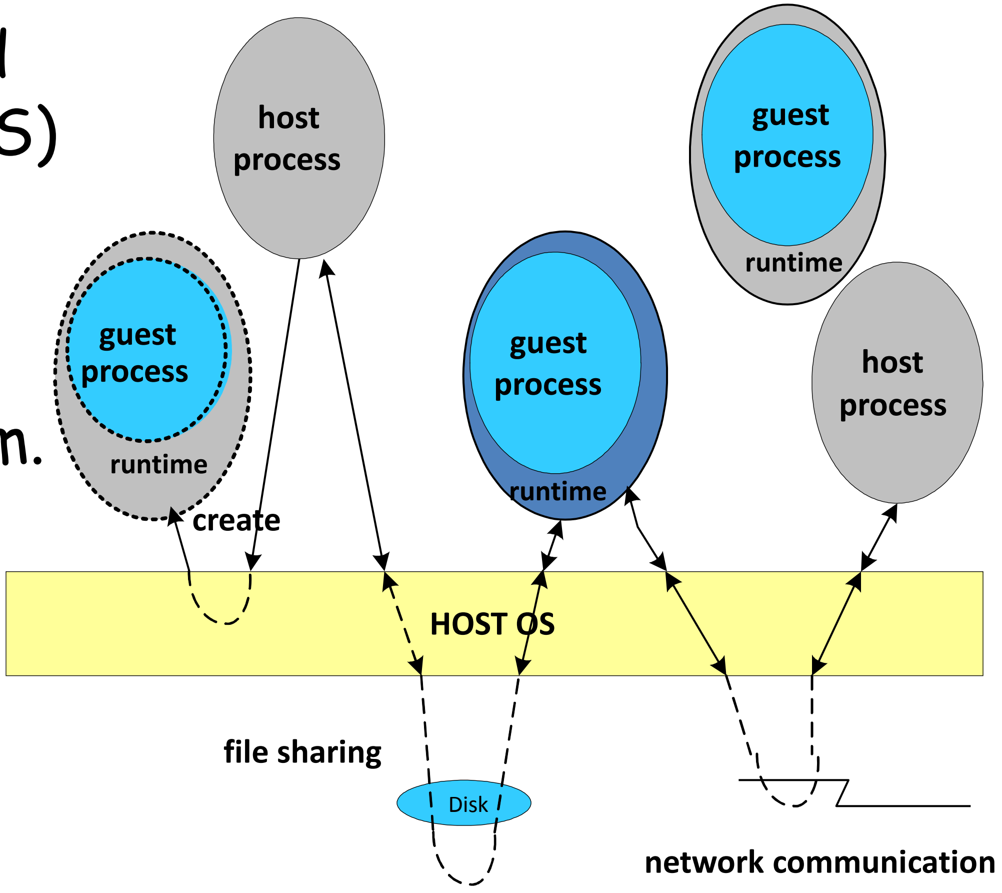


Outline

- What is it?
- Process VM Architecture
- Compatibility issue
- Mapping
  - Registers
  - Memory
- Memory issues
- Performance issues
- Exception emulation
- OS call emulation

Process Virtual Machines

By using a process VM:
A guest program developed for a computer (ISA and OS) other than the user's host system can be used in the same way as all other programs in the host system.

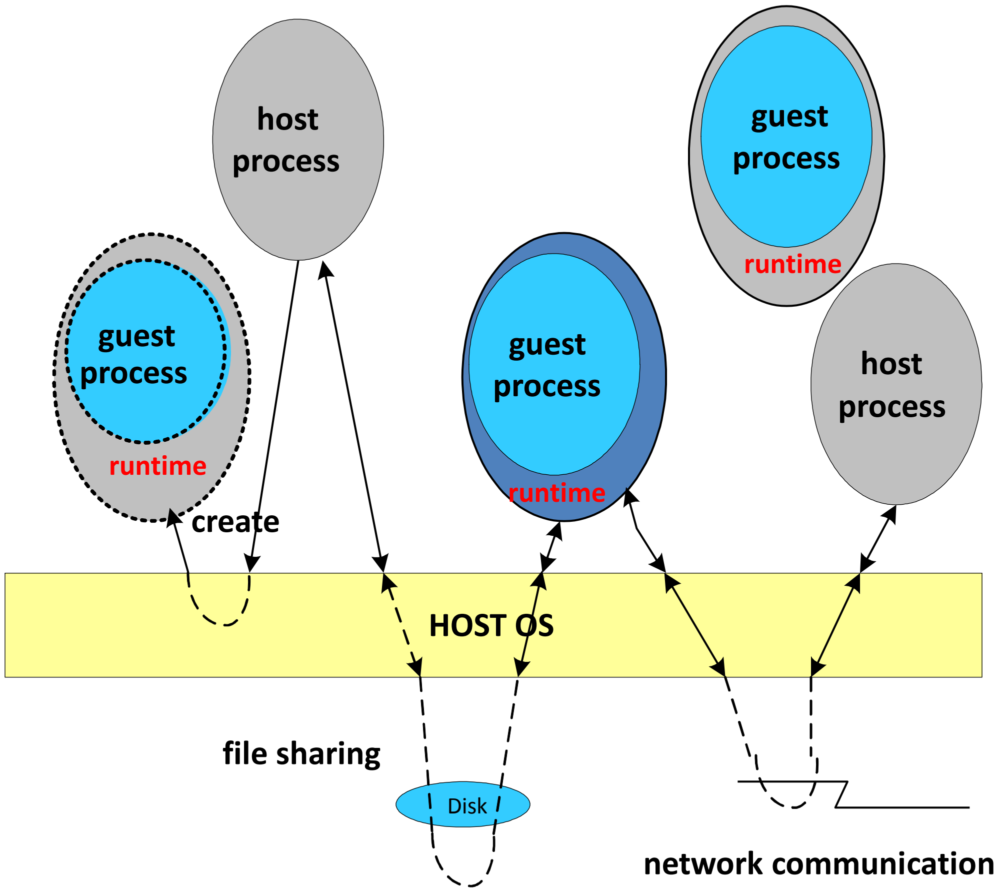


Process Virtual Machines

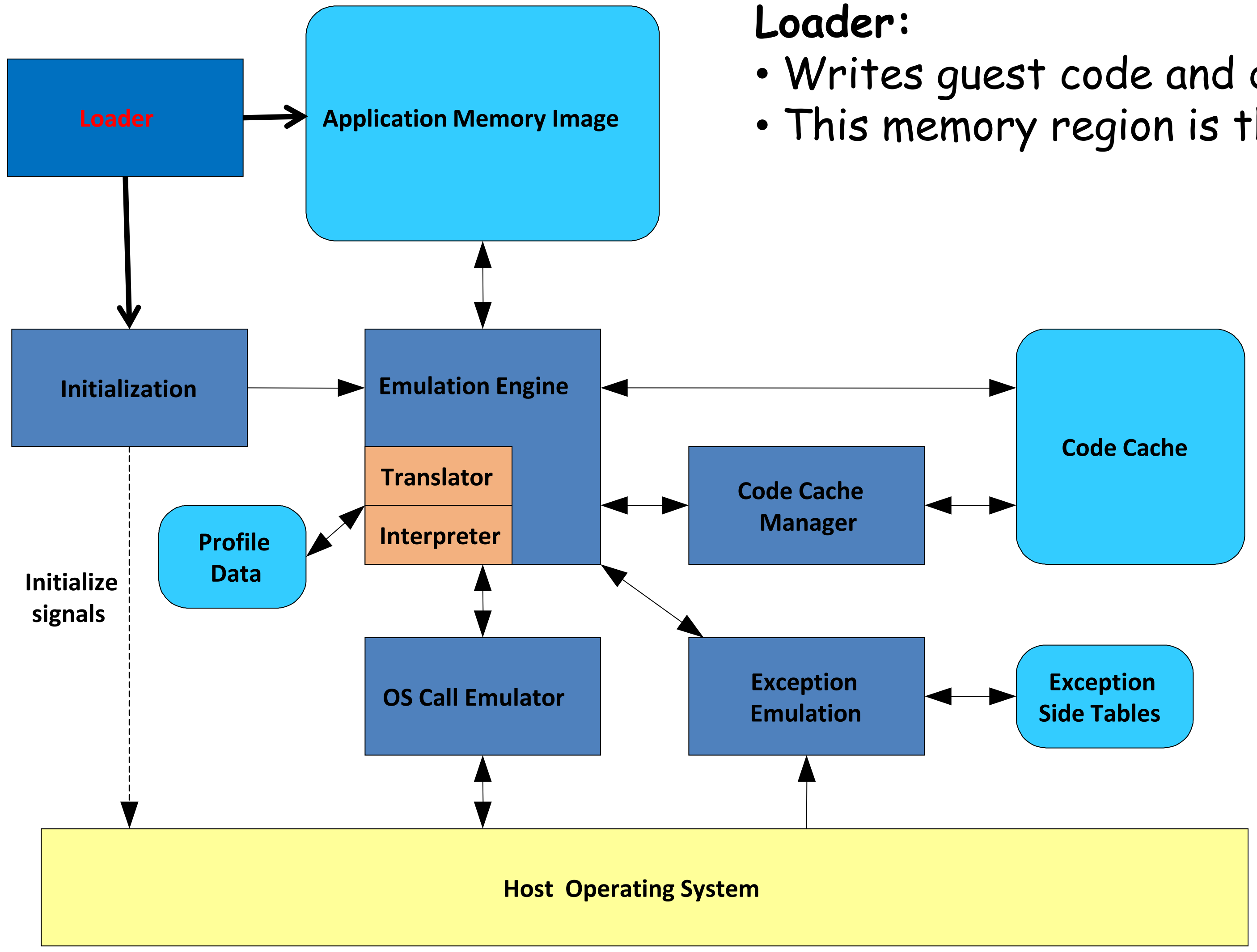
Runtime system:

- Encapsulates an individual guest process giving it the same appearance as a native host process.

- All host processes appear to conform to the guest's worldview.



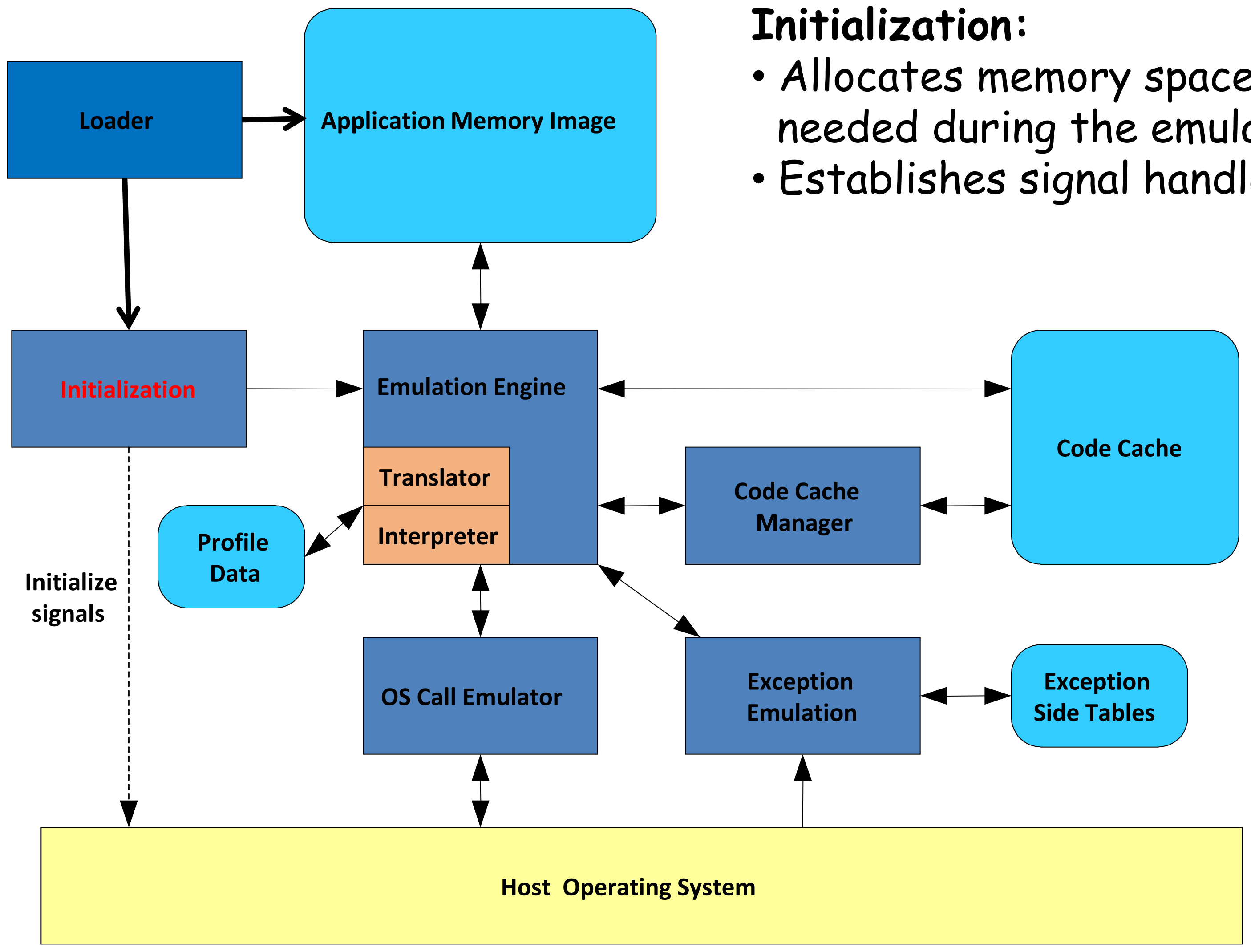
Process VM Architecture



Loader:

- Writes guest code and data into memory.
- This memory region is the runtime data.

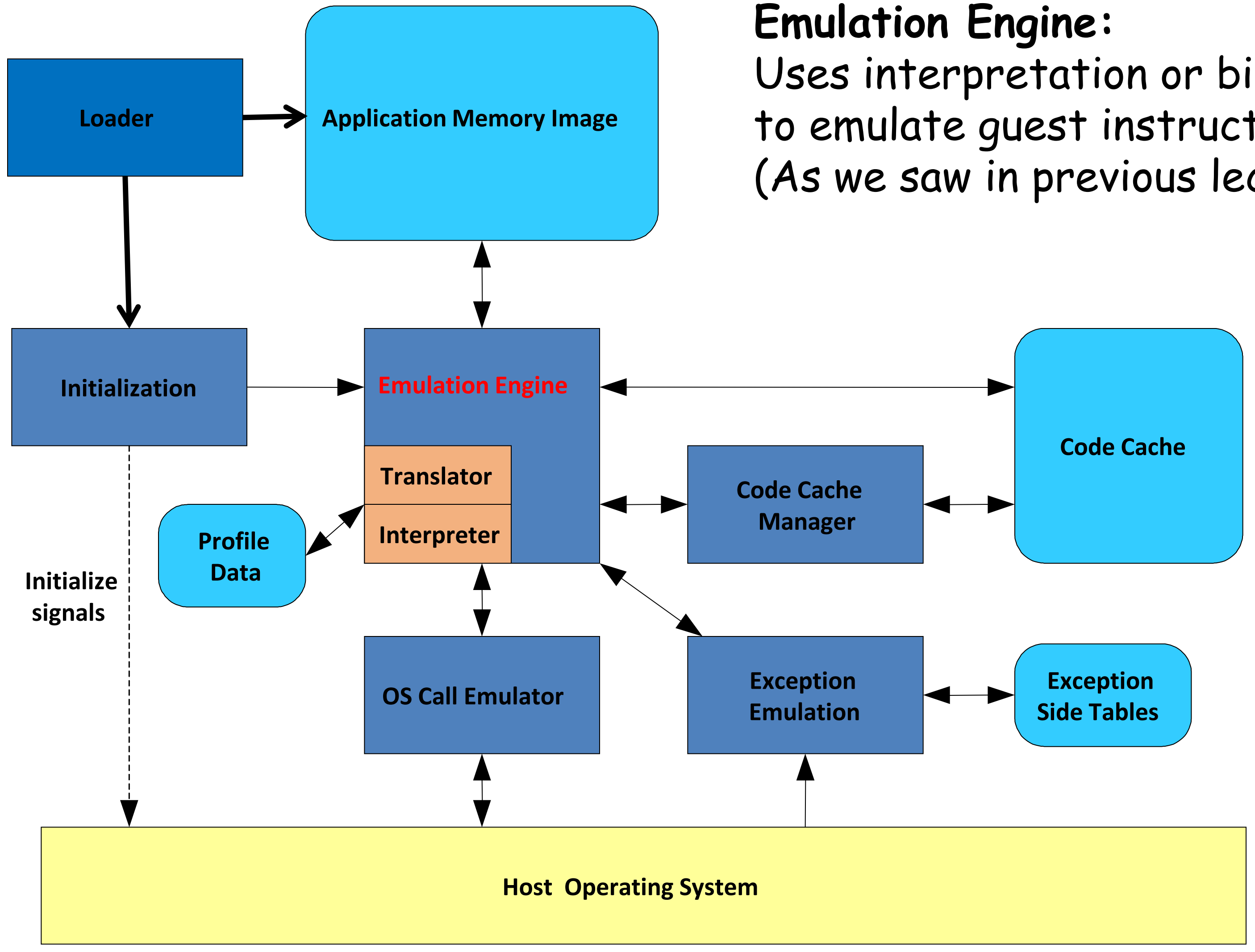
Process VM Architecture



Initialization:

- Allocates memory space for data structures needed during the emulation process.
- Establishes signal handlers with OS.

Process VM Architecture

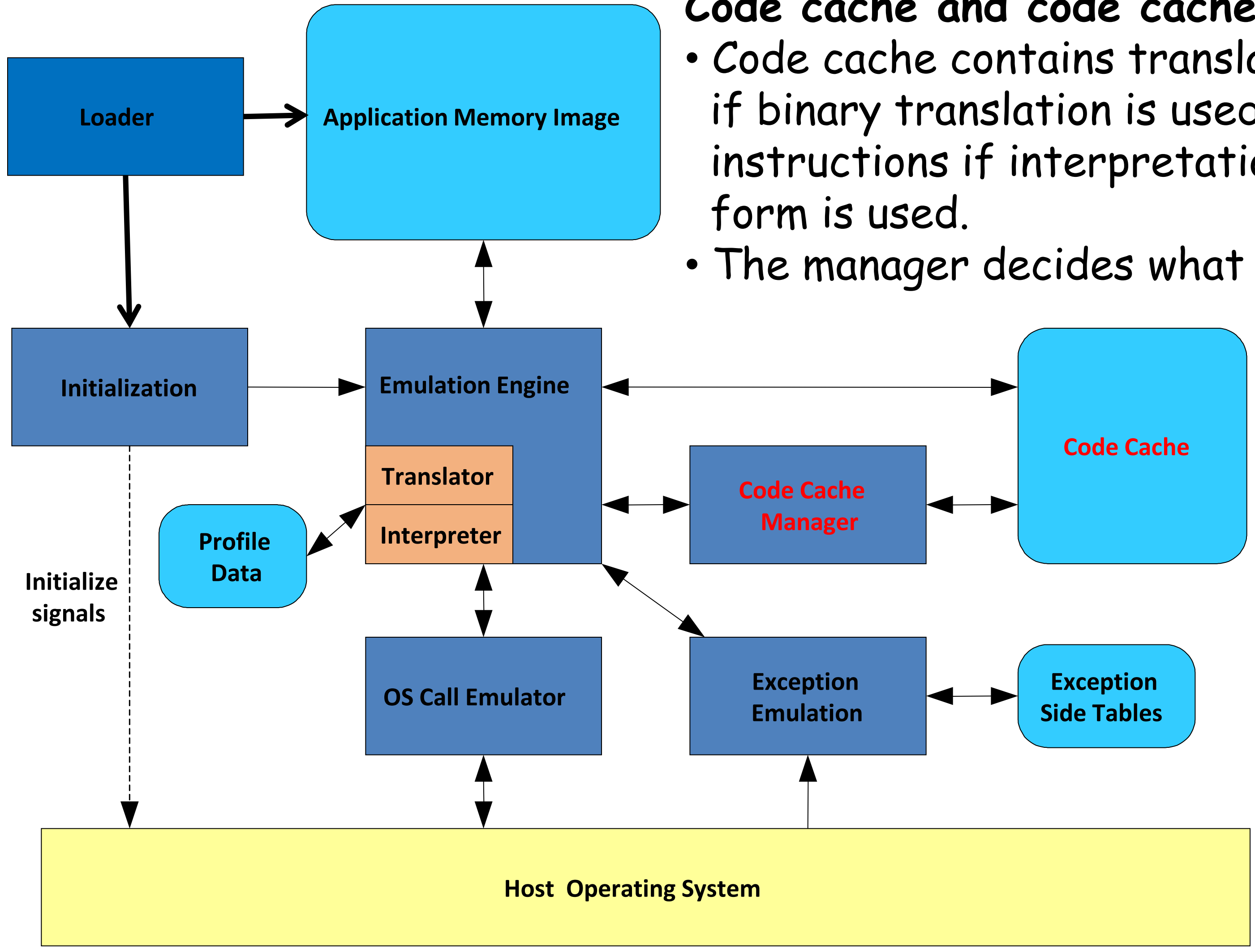


Emulation Engine:
Uses interpretation or binary translation to emulate guest instructions.
(As we saw in previous lecture.)

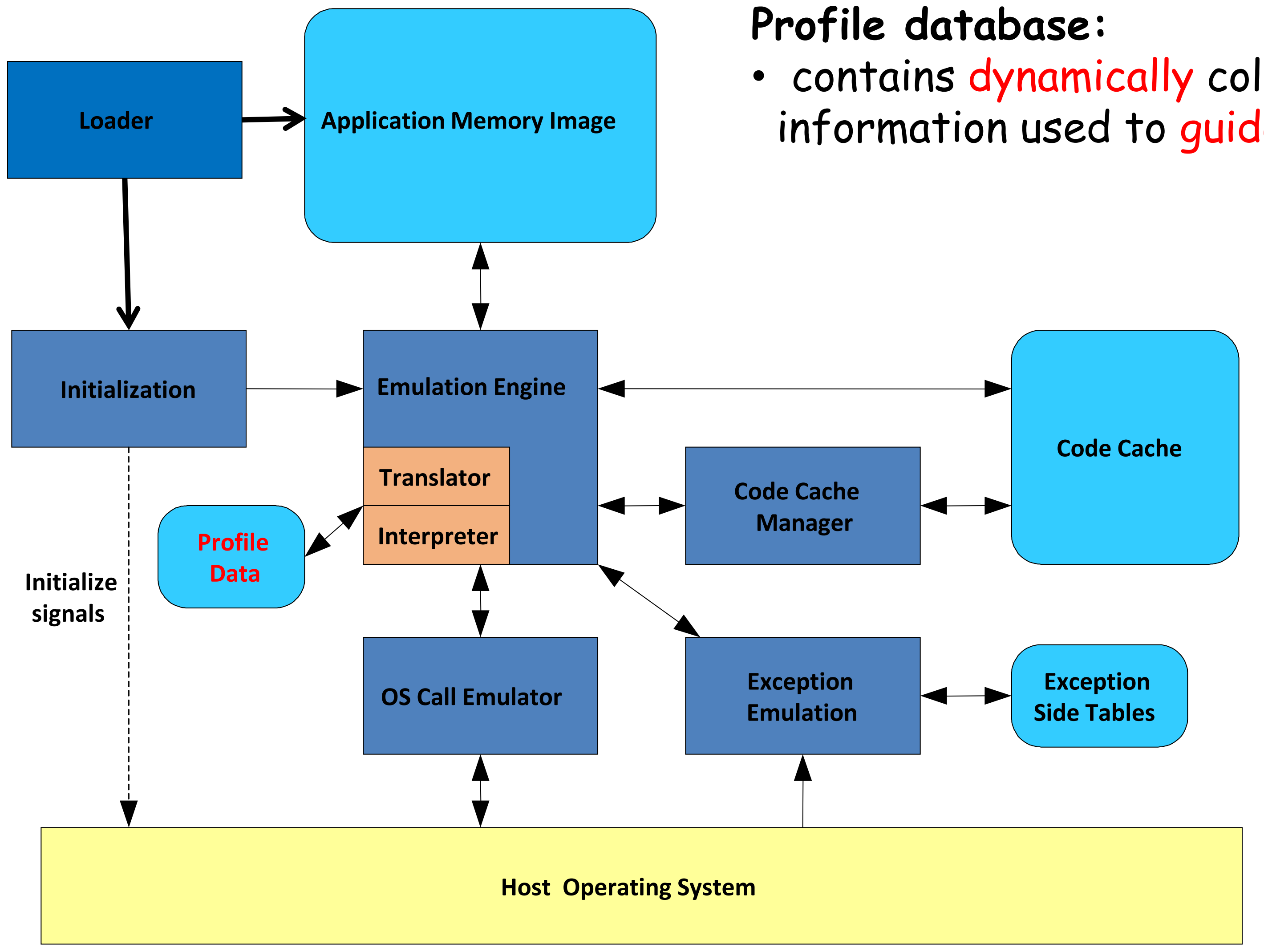
Process VM Architecture

Code cache and code cache manager:

- Code cache contains translated target code if binary translation is used. Or pre-decoded instructions if interpretation with intermediate form is used.
- The manager decides what goes into the cache.



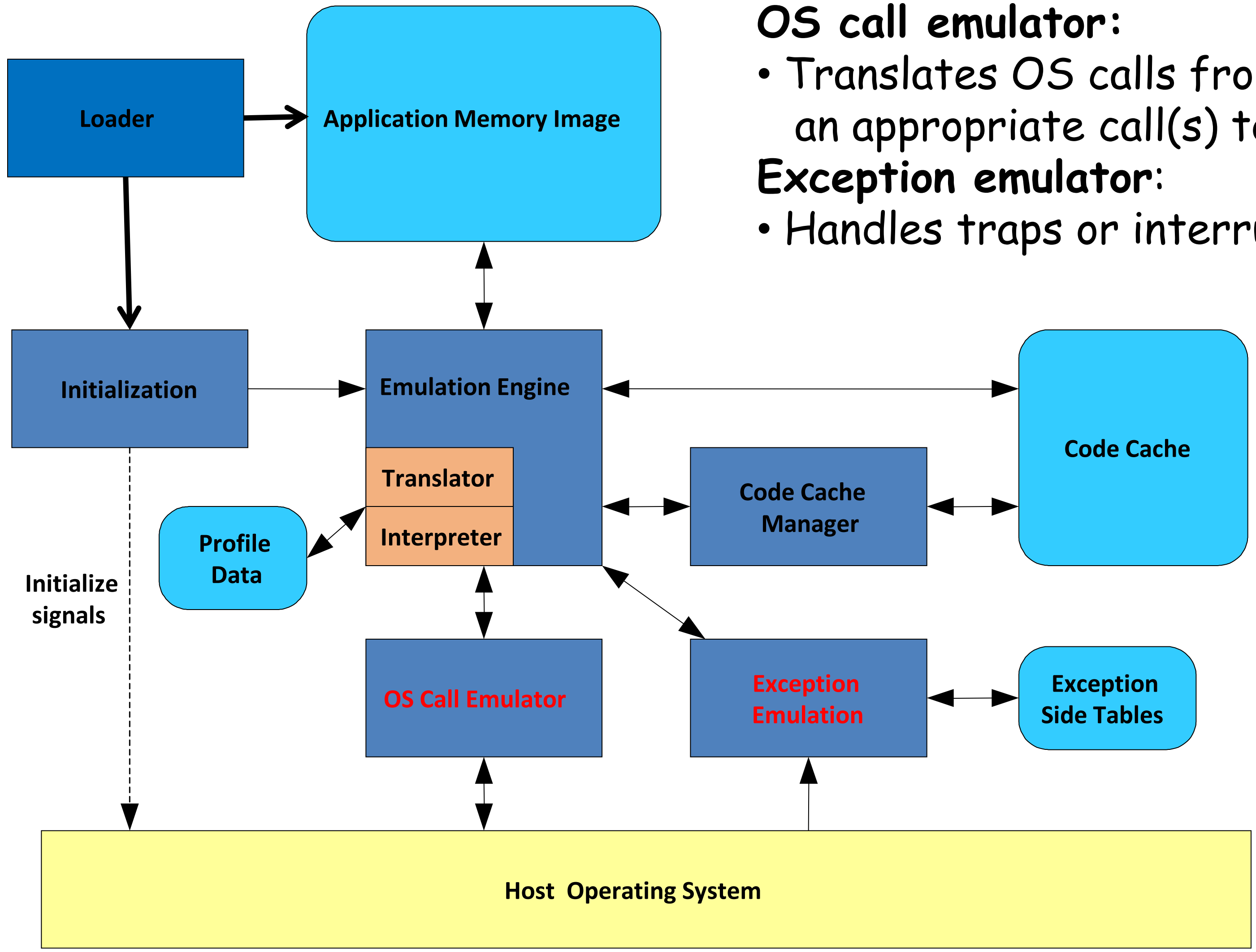
Process VM Architecture



Profile database:

- contains dynamically collected program information used to guide optimizations.

Process VM Architecture



OS call emulator:

- Translates OS calls from the guest into an appropriate call(s) to host OS.

Exception emulator:

- Handles traps or interrupts that may occur.

Compatibility

- A key consideration in VMs
- Tricky to define
  - Two systems are compatible if, in response to the same sequence of input values, they give the same sequence of output values.
  - In VMs, we are usually concerned with functional compatibility rather than performance compatibility.
- Two types of compatibility
  - Intrinsic compatibility
  - Extrinsic compatibility

Intrinsic Compatiblity

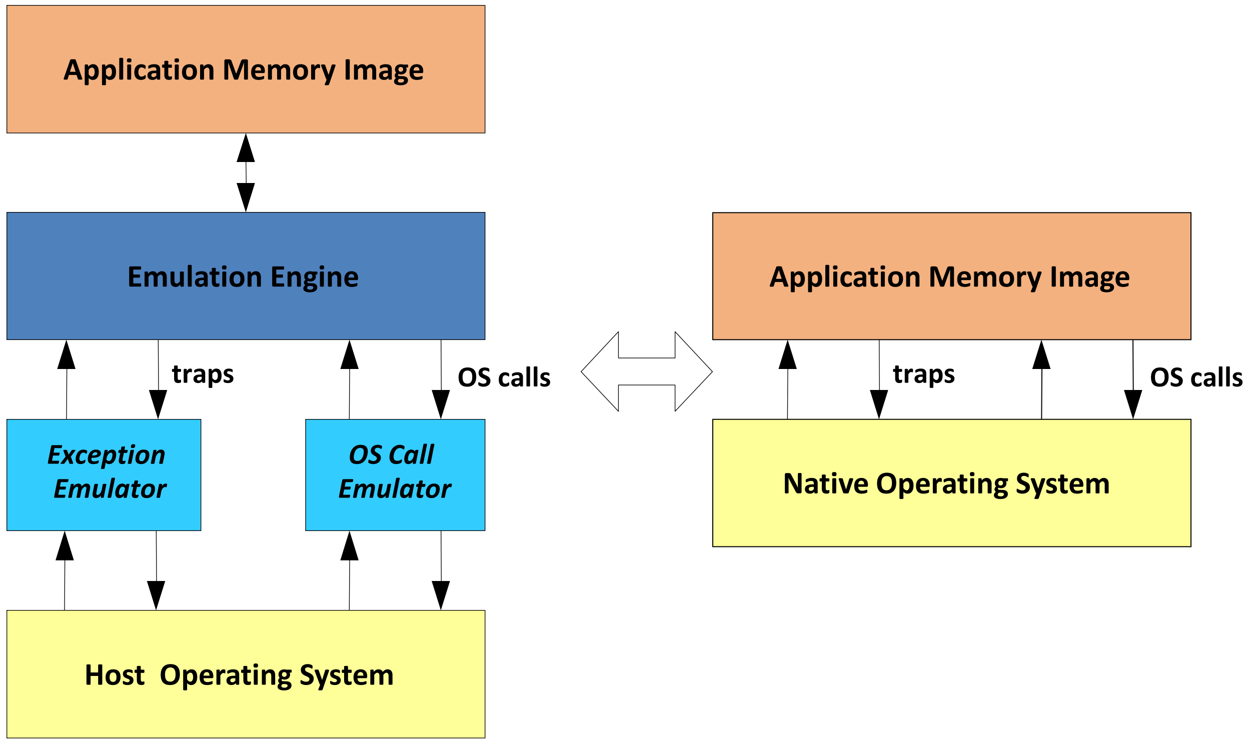
- Two systems are *intrinsically compatible* if they are compatible for all possible input sequences
  - Example: Intel strives for intrinsic compatibility when it produces a new x86 microprocessor

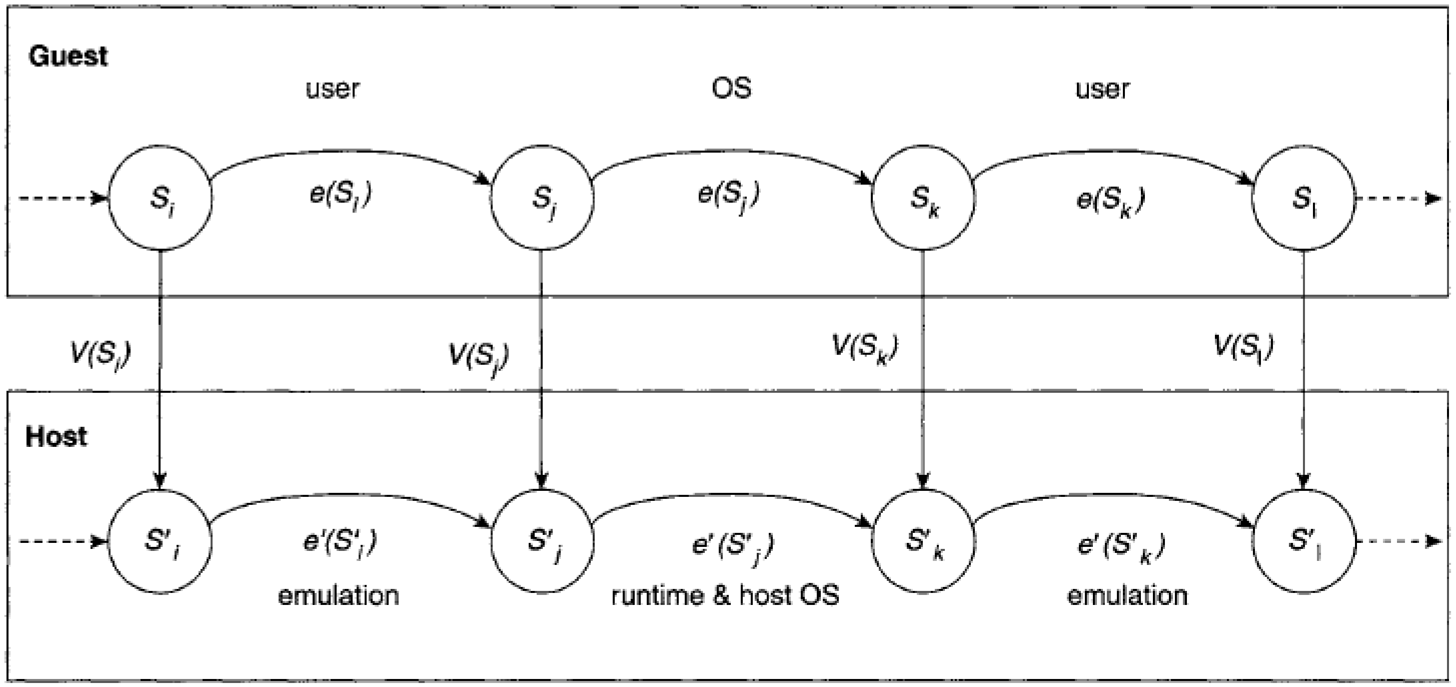
Extrinsic Compatibility

- Many useful VM applications do not achieve (or need) intrinsic compatibility
  - Limited application set: run Microsoft productivity tools
  - Compiled for a certain API
  - Certified by the programmer to be free of certain types of bugs
- Two systems are *extrinsically compatible* if they are compatible for a well-defined subset of input sequences
  - Based on VM implementation, architecture/OS specifications, *and* certain external guarantees/certifications
  - Some burden on the user to make sure that the guarantees are met
  - Example: FX!32 is certified for certain Windows apps.

Process VM Compatibility Framework

- Form correspondence between VM and Native System at points where control is transferred to OS
- At these points: equivalent vales for both register and memory state





Two type of states:

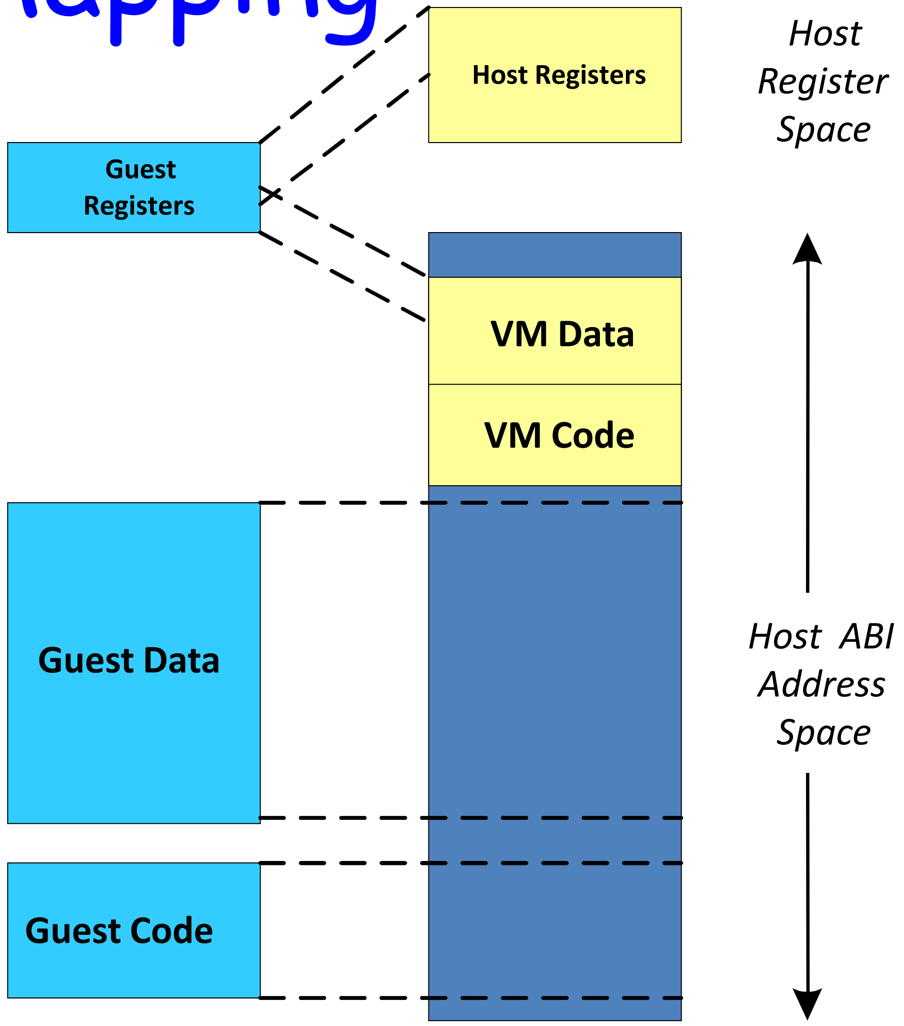
- User managed states: modified by user instructions
- OS managed states: modified by OS

Constructing a Process VM

- Mapping of user-managed state: State Mapping
  - held in registers
  - held in memory
- Perform emulation (operations to transform state)
  - memory architecture emulation
  - instruction emulation
  - exception emulation
  - OS emulation

State Mapping

- State is maintained in resources → memory + registers
- For best performance
  - Guest register space fits "inside" host register space
  - Guest memory space fits "inside" host memory space
- Best case does not always happen
- Resource does not always map to the same resource type (e.g. registers mapped to memory)

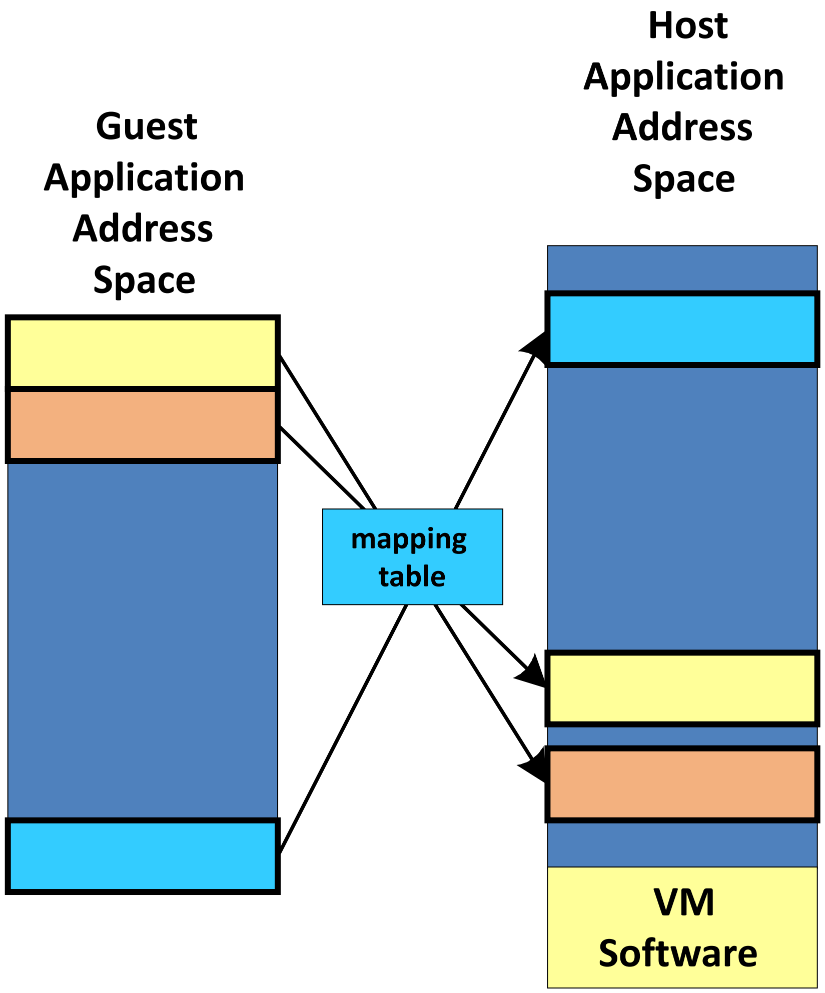


Register Mapping

- host registers > guest registers
  - can map all guest registers to host registers
- host registers < guest registers
  - must use some registers from host + memory from host
- host registers = guest registers
  - must use some registers from host + memory from host
- At host memory: register context block

Memory Address Space Mapping

- VM Software (i.e runtime) maintains mapping table
  - Similar to hardware page tables/TLBs
  - Slow, but can always be made to work
  - Usually at a granularity of x*(host page size)


Guest Application Address Space
Host Application Address Space
mapping table
VM Software

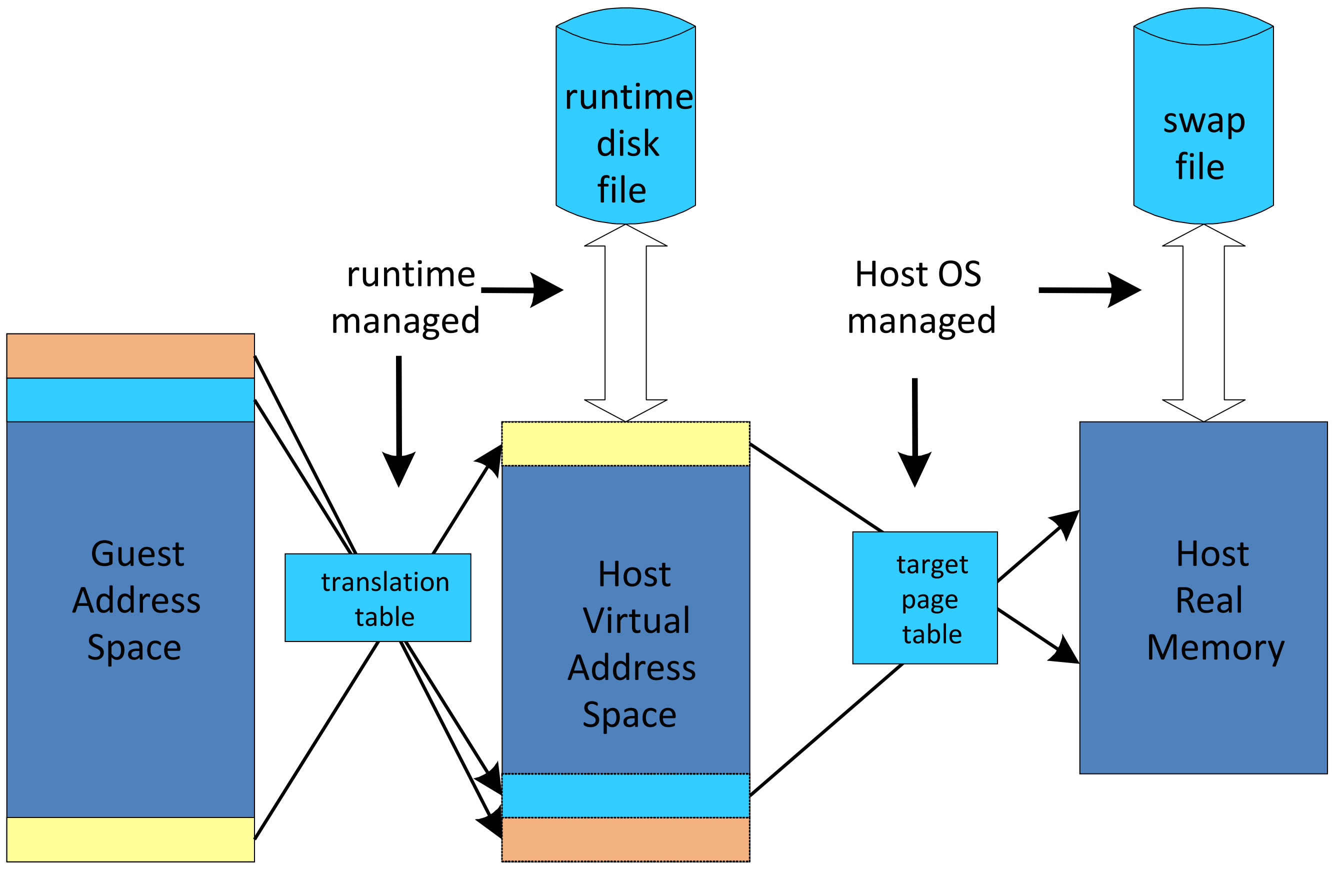
Example

```
Initially,  R1 holds source address
            R30 holds base address of mapping table
```

```
srwi  r29,r1,16    ;shift r1 right by 16
slwi  r29,r29,2    ;convert to a byte address
lwzx  r29,r29,r30 ;load block location in host memory
slwi  r28,r1,16    ;shift left/right to zero out
srwi  r28,r28,16  ;source block number
slwi  r29,r29,16  ;shift up target block number
or    r29,r28,r29 ;form address
lwz   r2,0(r29)    ;do load
```

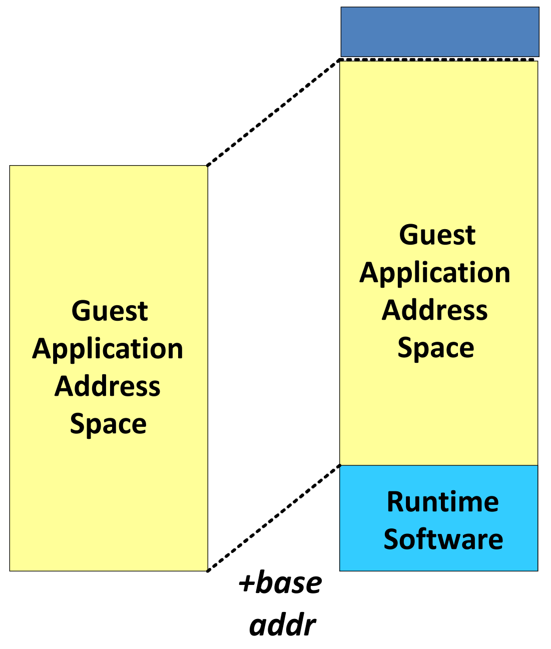
Memory Space Mapping: General Case

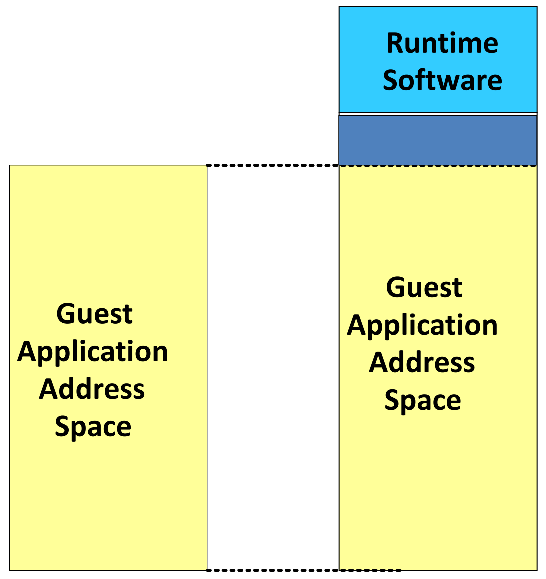
- Guest address space > Host address space + Runtime



Direct (Hardware) Mapping

- VM software mapping is slow
  - Several instructions per load/store
- Use underlying hardware
  - If guest address space + runtime fit within host space


Guest Application Address Space
Guest Application Address Space
Runtime Software
+base addr


Runtime Software
Guest Application Address Space
Guest Application Address Space

Memory Space Mapping

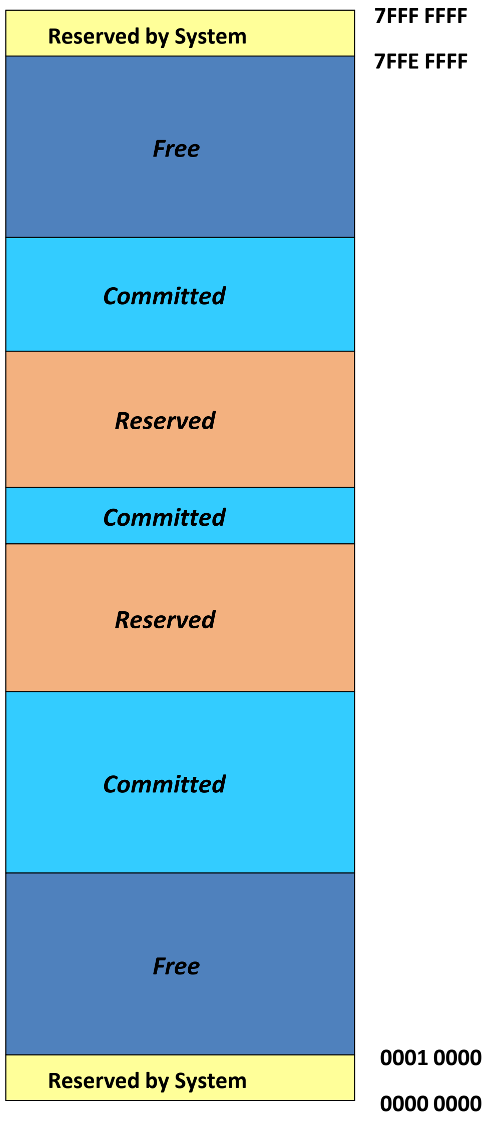
- Runtime + guest space ≤ host space
  - May be able to use underlying hardware to efficiently map addresses
- Guest space > host space
  - Use software translation
    - Software page table
    - Several instructions overhead per load/store
- Guest space == Host space
  - Important case
    - Occurs often
    - Especially in same-ISA dynamic optimization
  - No room left over for runtime
    - Use software translation
    - Settle for extrinsic compatibility

Now that we saw state mapping,
how to make state transition (emulation)?

Memory Architecture Issues

- Overall address architecture – paged, segmented, etc.
  - We will focus on paged architectures
- Privilege types R, W, E
- Protection/allocation granularity – page sizes
- Most ABIs have a very specific architected memory
  - Example: Win32

How will we emulate this?

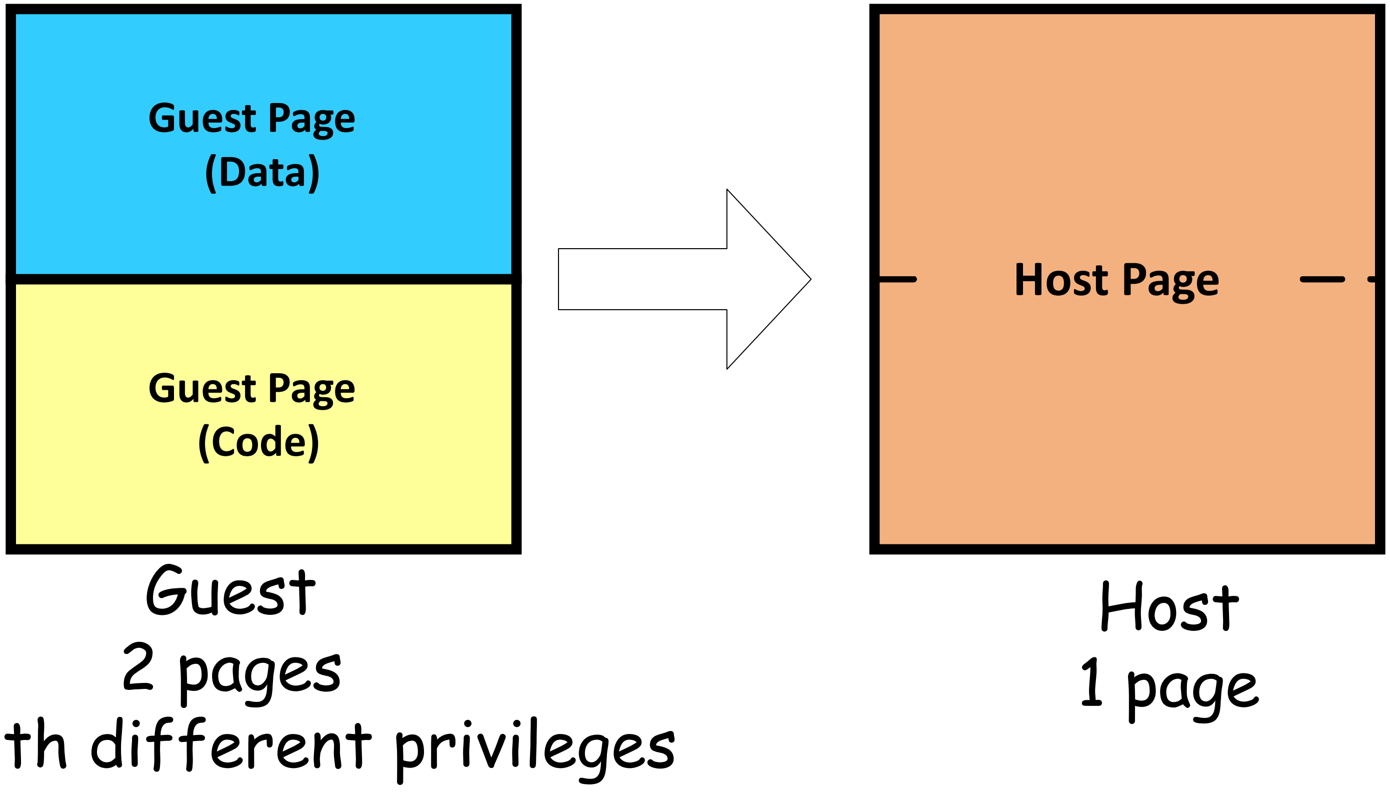


Guest Memory Protection

- VM software mapping can be easily used
  - Place (and check) protection info in mapping table
- Use underlying hardware
  - Runtime must be able to set privileges
  - Protection faults should be reported to Runtime
  - Requires some support from Host OS
    - mprotect() and SIGSEGV in Linux

Page Granularity Issue

- Problems arise if guest page size < host page size



Do you see the problem? What do we do?

Suggested Solutions

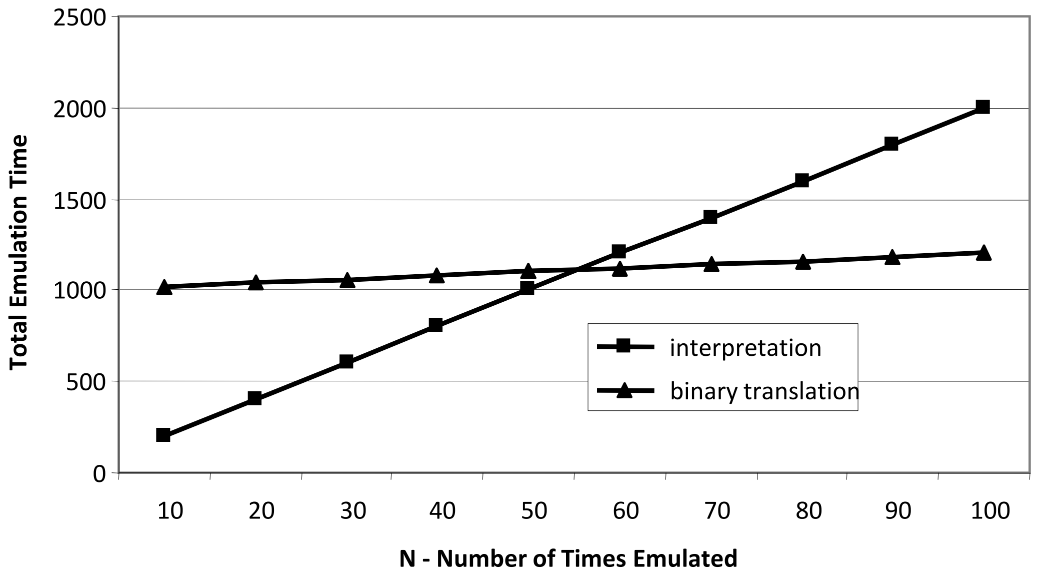
- Software-based mapping
- Aligning the code and data regions on host page boundaries
- Giving the host page the lesser of privileges and let runtime handle the extra traps

What if host and guest do not support the exact type of protections?

High Performance Emulation

- Important tradeoff
  - Startup time -- Cost of converting code for emulation
  - Steady state -- Cost of emulating
- Interpretation:
  - Low startup, high steady state cost
- Binary translation
  - High startup, low steady state cost

Overall time for emulating an instruction N times = S + NT

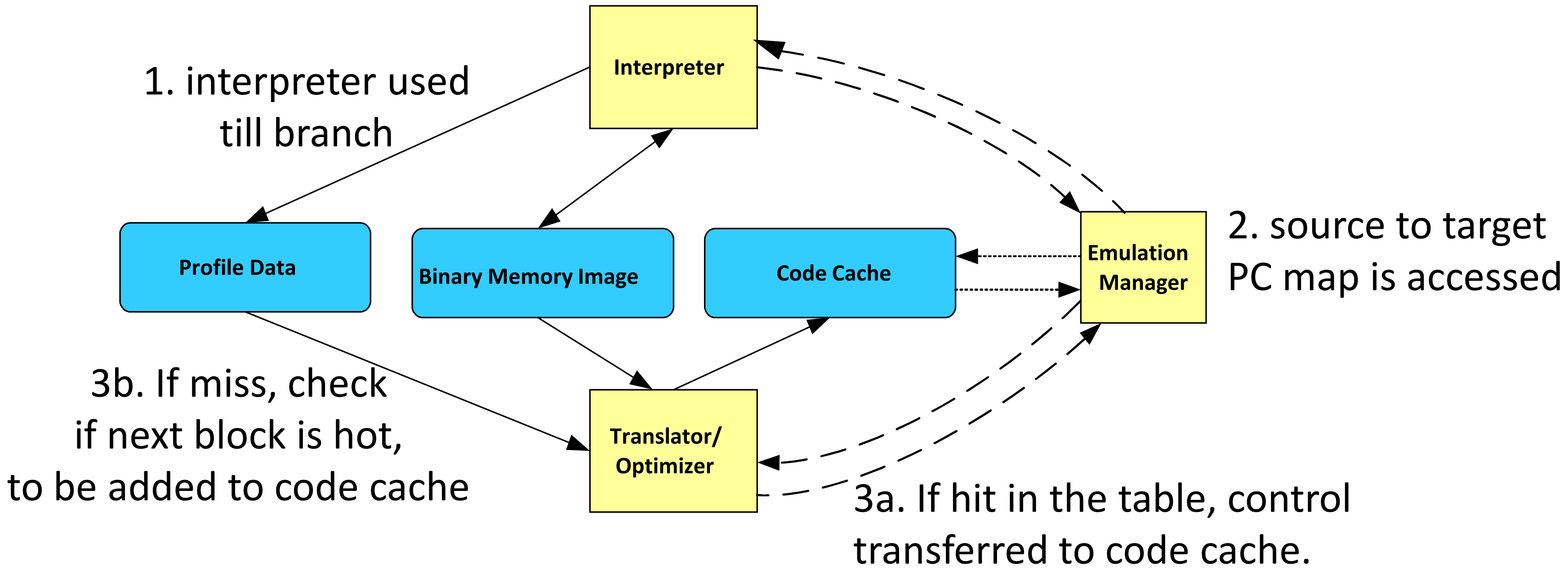


Binary translation:
T = 2 cycles
S = 1000 cycles

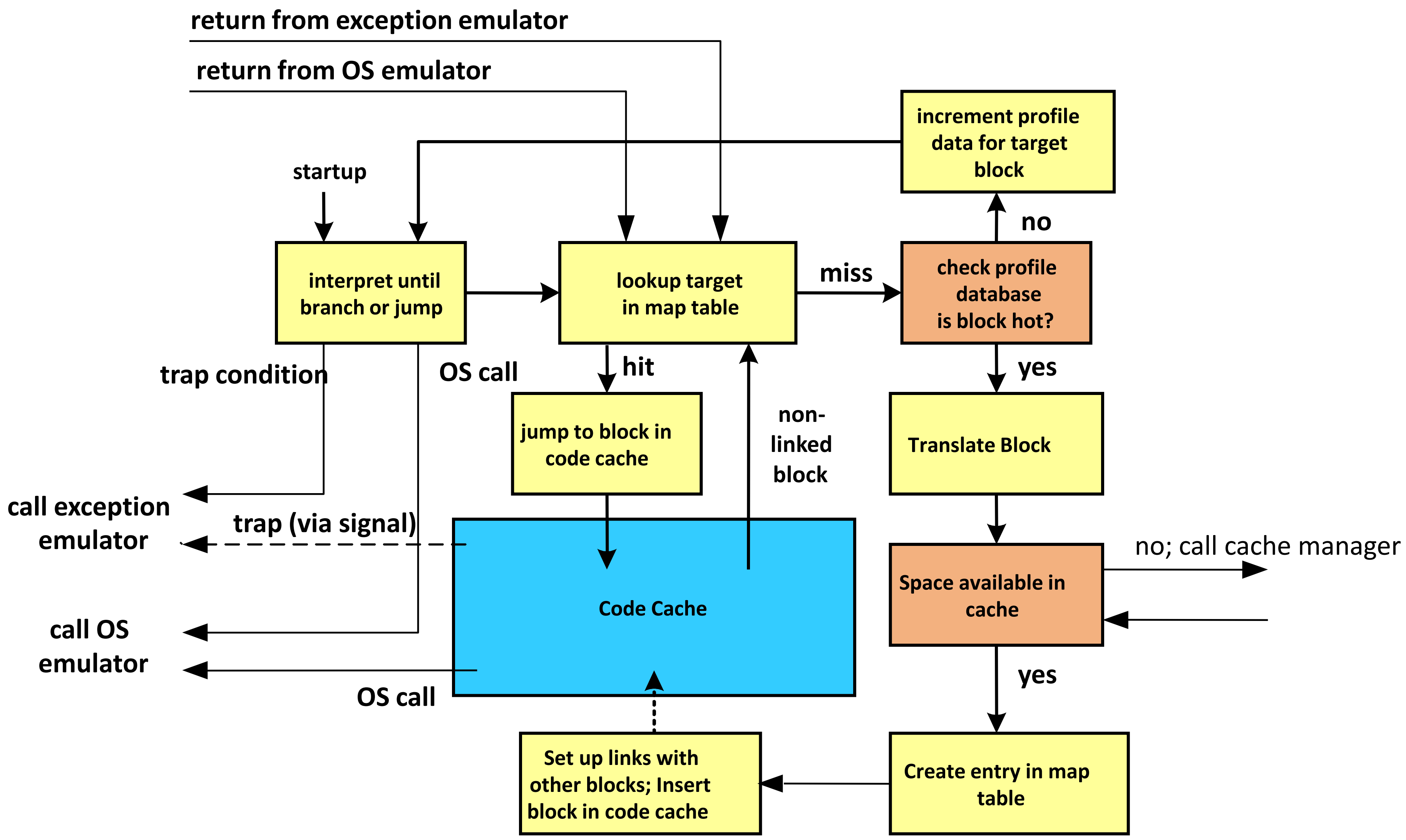
Interpretation:
T = 20 cycles
S = 0 cycles

Staged Emulation

- Adjust optimization level to execution frequency
- Tradeoff
  - Total runtime = program runtime + translation overhead
  - Higher optimization ⇒ shorter program runtime
  - Lower optimization ⇒ lower overhead


Interpreter
1. interpreter used till branch
Profile Data
Binary Memory Image
Code Cache
Emulation Manager
2. source to target PC map is accessed
3b. If miss, check if next block is hot, to be added to code cache
Translator/ Optimizer
3a. If hit in the table, control transferred to code cache.

Emulation Engine Execution Flow


return from exception emulator
return from OS emulator
startup
increment profile data for target block
interpret until branch or jump
lookup target in map table
miss
check profile database is block hot?
no
yes
trap condition
OS call
hit
jump to block in code cache
non-linked block
Translate Block
call exception emulator
trap (via signal)
Code Cache
Space available in cache
no; call cache manager
call OS emulator
OS call
yes
Set up links with other blocks; Insert block in code cache
Create entry in map table

Staged Emulation

- General Strategy
  1. Begin interpreting
  2. For code executed above a threshold
     - Use *simple translation/optimization*
  3. For translated code executed above a threshold
     - *Optimize more*
  - etc.
- Specific Strategies may skip some of the steps
  - Shade uses 1 and 2
  - Wabi uses 2 and 3
  - FX!32 uses 1 and 3
  - IA32-EL, UQDBT use 2 and 3

How to Emulate Exceptions?

- Exceptions: interrupts and traps
  - Interrupts: an external event
  - Trap: direct result of program execution, and produced by a specific instruction
- Exceptions:
  - ABI visible
  - ABI invisible
- Exception handling must be precise

Exceptions: Traps

- Can be detected directly via interpretation
  - interpretive trap detection
  - interpreter jumps to a runtime trap handler
- Can be detected indirectly via target ISA trap and signal
  - signal by host OS delivered to runtime trap handler
- Semantic "matching"
  - If trap is architecturally similar in target and source then trap/signal may be used
  - Otherwise interpretive method must be used

Exceptions: Interrupts

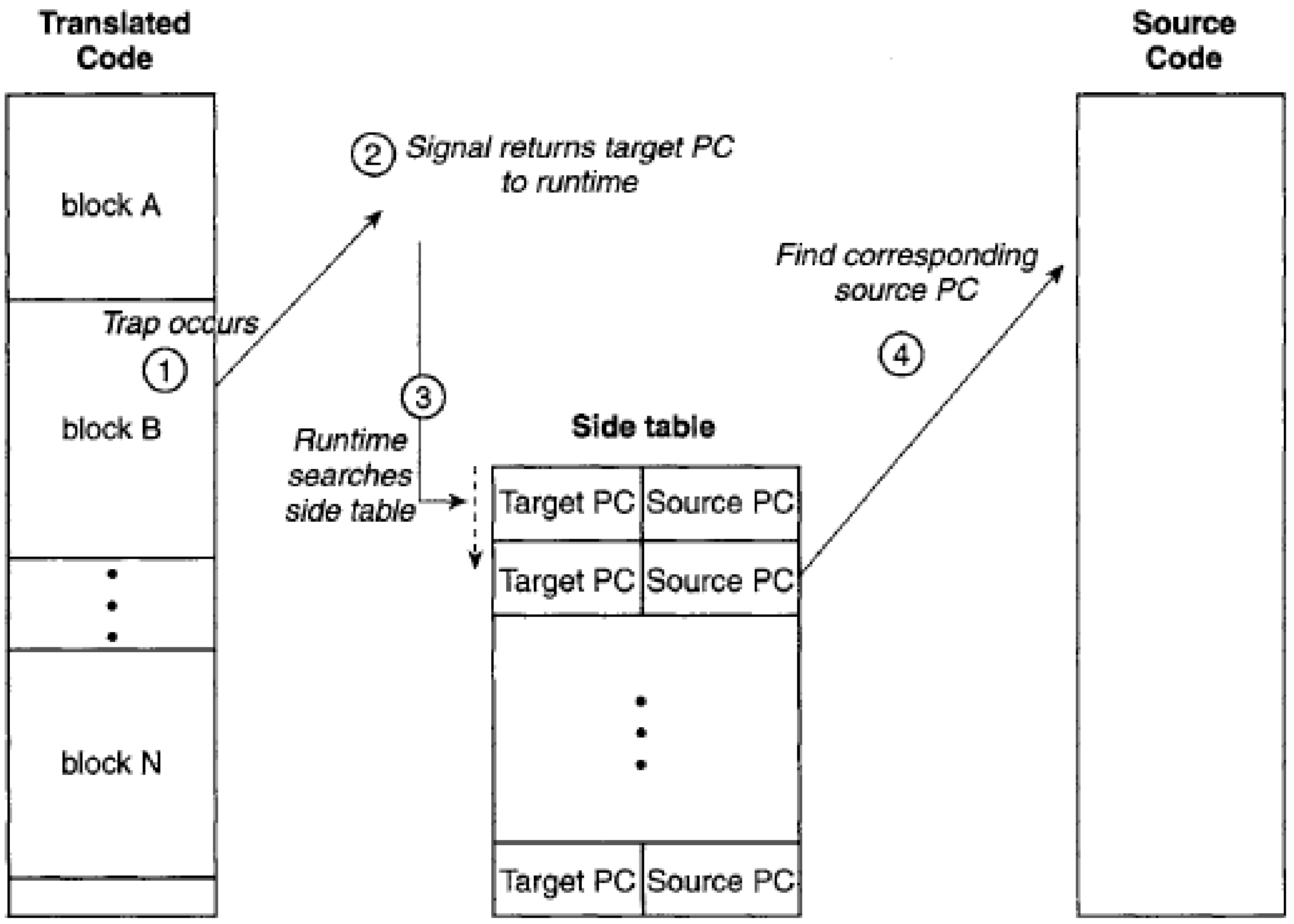
- Application may register some interrupts
  - Precise state easier than traps
    - (because there is more flexibility wrt location)
- Problem: Translated blocks may be executed for an unbounded time period
- Some Solutions:
  - Interrupt signal goes to runtime
  - Runtime unchains translation block currently executing
  - Runtime returns control to current translation
  - Translation soon reaches end (and precise state is available)
- If interrupts are common, runtime may inhibit all chaining

After exception, precise state of the guest program must be maintained.

Precise State: Program Counter

- Interpretation: Easy – source PC is maintained
- Binary translation: more difficult – source PC only available at translation block boundaries
  - Trap PC is in terms of target code
  - Target PC must be mapped back to correct source PC
- Simple solution
  - Use side table and reverse translate
  - Requires search of table to find trapping block
  - Requires full map of target to source PCs
  - Simple, but very expensive – probably larger than all of translated code

Reverse Translation Side Table



Recovering Memory State

- Simple if target code updates memory state in same order as source code
  - Restricts optimizations (more difficult to back-up than register state)

Recovering Register State

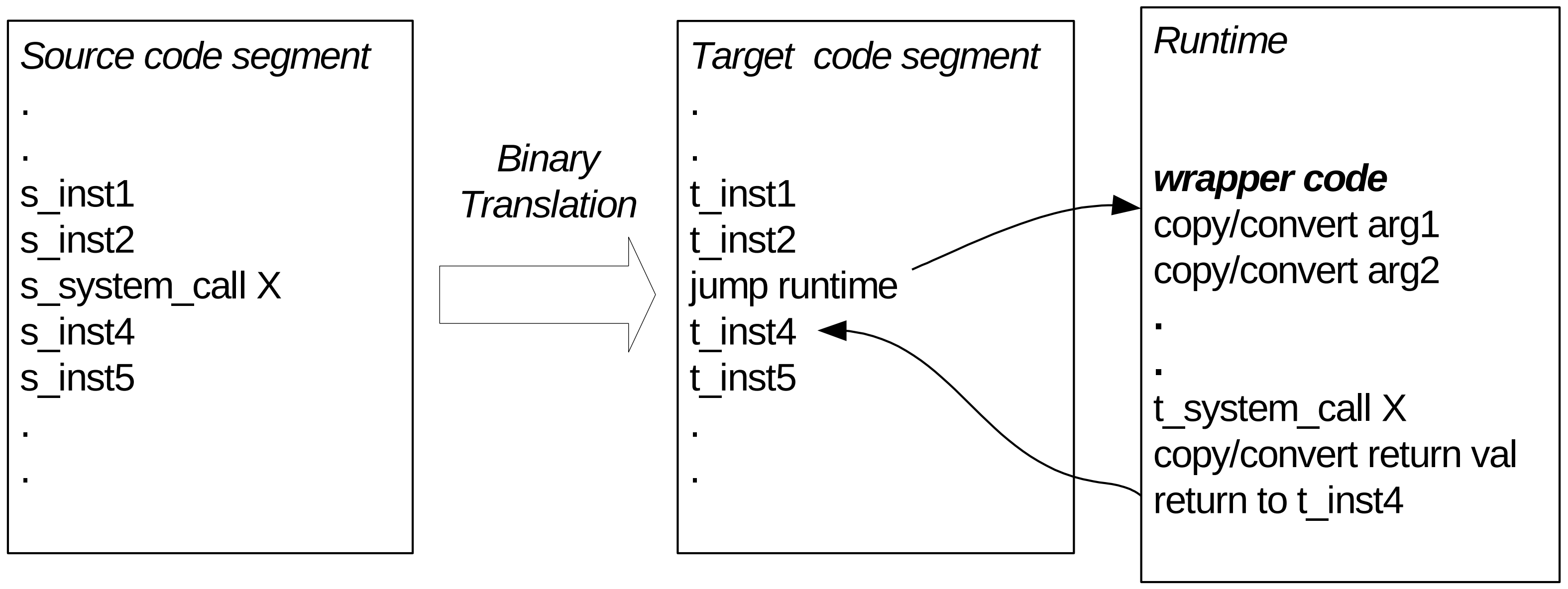
- Simple if:
  - emulator uses consistent source-to-target register mapping
  - register is updated in the same order in both source and translated code sequences

OS Call Emulation

- Process VM is required only to maintain compatibility at the ABI level.
- Need only to emulate the function/semantics of the guest OS calls.
- We have two situations
  - same host and guest OSes
  - Different guest and host OSes

OS Call Emulation: Same OSes

- "Wrapper" or "Jacket" code converts source call to target OS call(s)



Same source and target OSes (different ISAs)

- Syntactic translation only
- E.g. pass arguments in stack rather than registers

OS Call Emulation: Different OSes

- Depending on the emulator implementation, some calls may be handled by the runtime.
  - e.g. memory management system calls by guest
- Semantic translation/matching required
- May be difficult (or impossible)
- OS deals with real world
  - What if source OS supports a type of device that the target does not?

Code Cache Management

- Code Cache is different from typical hardware cache
  - Variable sized blocks
    - block of a code cache depends on the size of the translated target block
  - Dependences among blocks due to chaining
  - No "backing store"; re-generating is expensive
- These factors affect replacement algorithm

LRU Replacement

- Good in theory; problematic in practice
- Fragmentation problems
  - Variable block sizes
  - Need to unlink when replacing

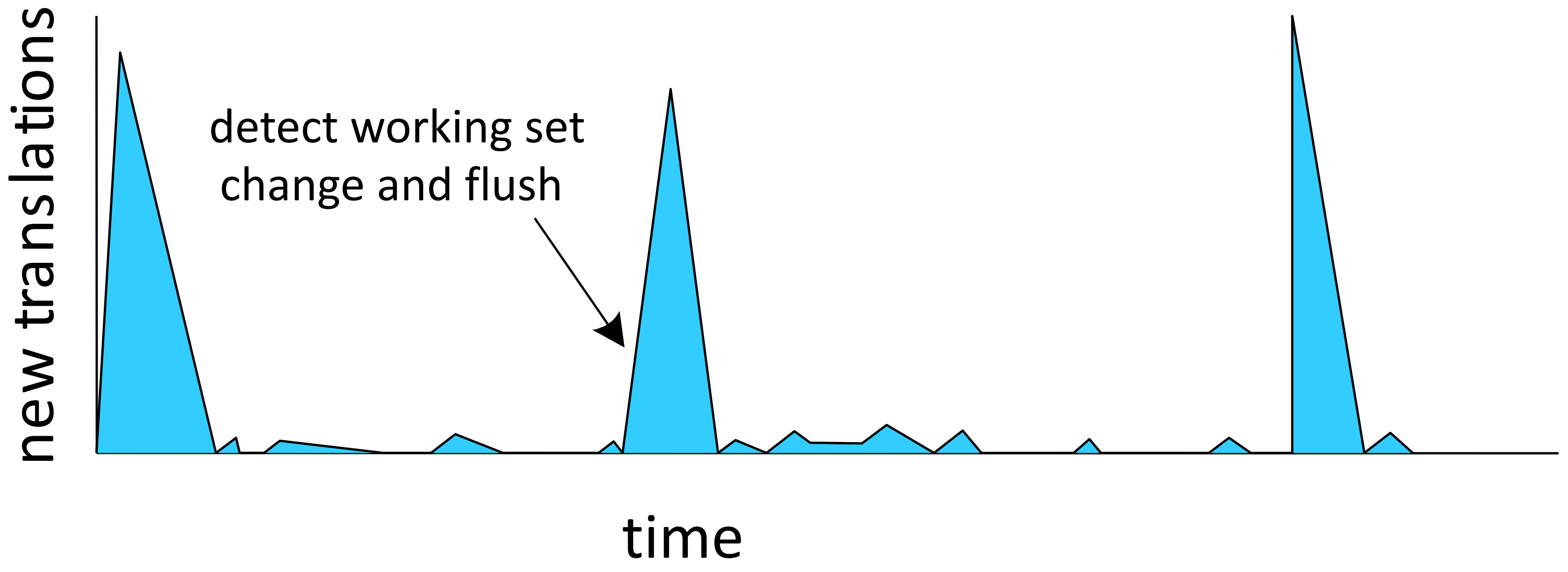
    (could be done with backpointers in code cache data structure)

Flush When Full

- Simple, basic algorithm
- Gets rid of “stale” links if control flow changes
- High overhead for re-translating after flush

Pre-emptive Flush

- Flush when program phase change is detected
  - Many new translations will be needed, anyway
- Detect when there is a burst of new translations

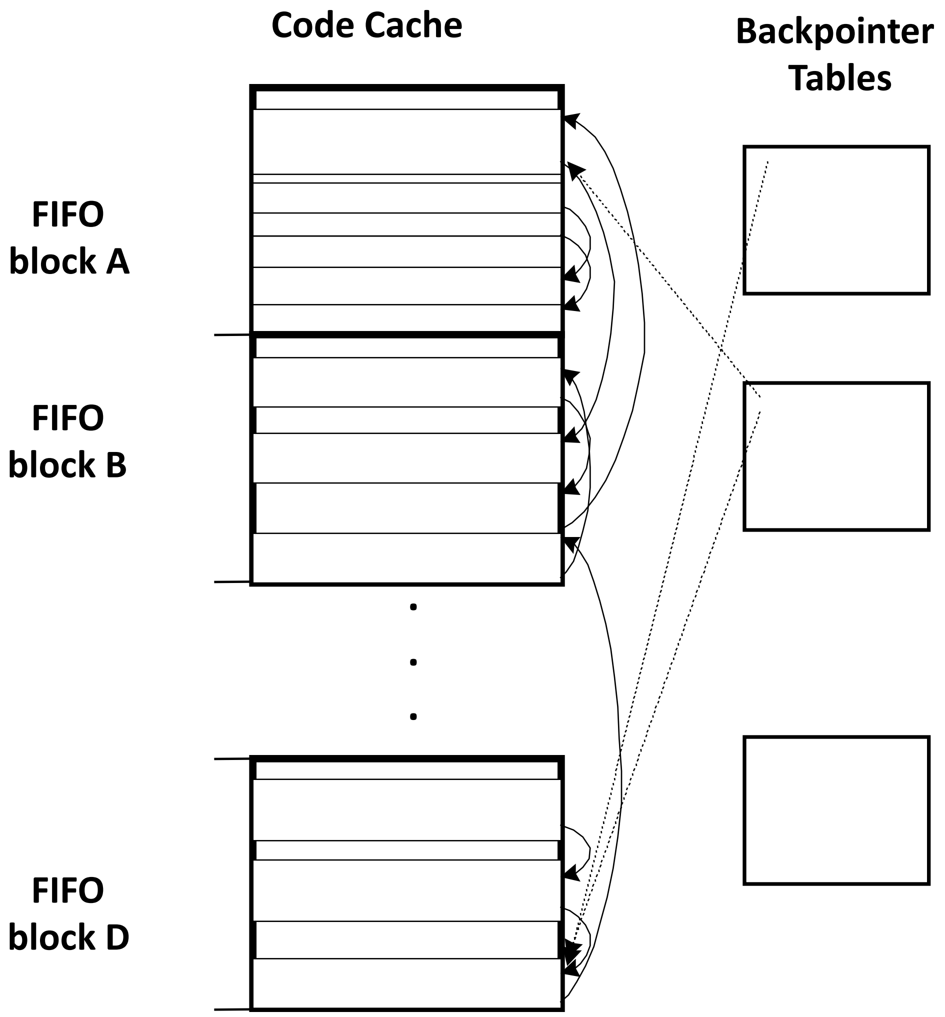


Fine-Grain FIFO

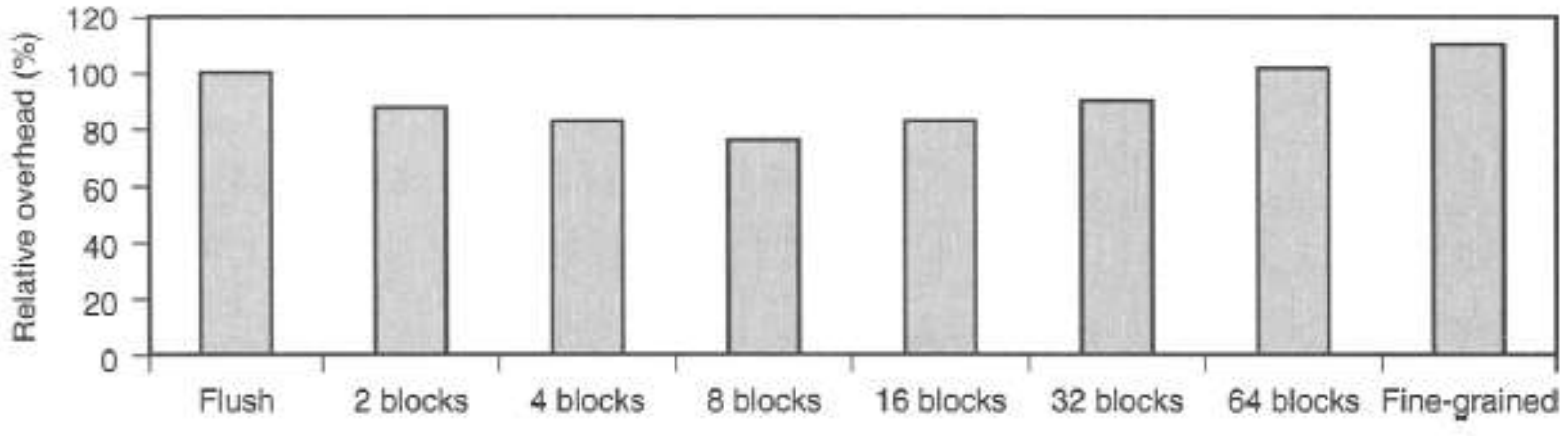
- Takes some advantage of temporal locality
- Code cache can be managed as circular buffer
- What if the new block is of different size than the victim?
  - Replace several victims
- Still requires backpointers to unlink replaced blocks

Coarse-Grain FIFO

- Replace many blocks at once
  - Large fixed-size blocks?
  - Only backpointers among replacement blocks need to be maintained
  - OR linking between large blocks can be prohibited.



Relative Performance



Conclusions

- In this lecture we studied the basics of process VM
  - The main architecture
  - The different issues and how to deal wit them