Virtual Machines

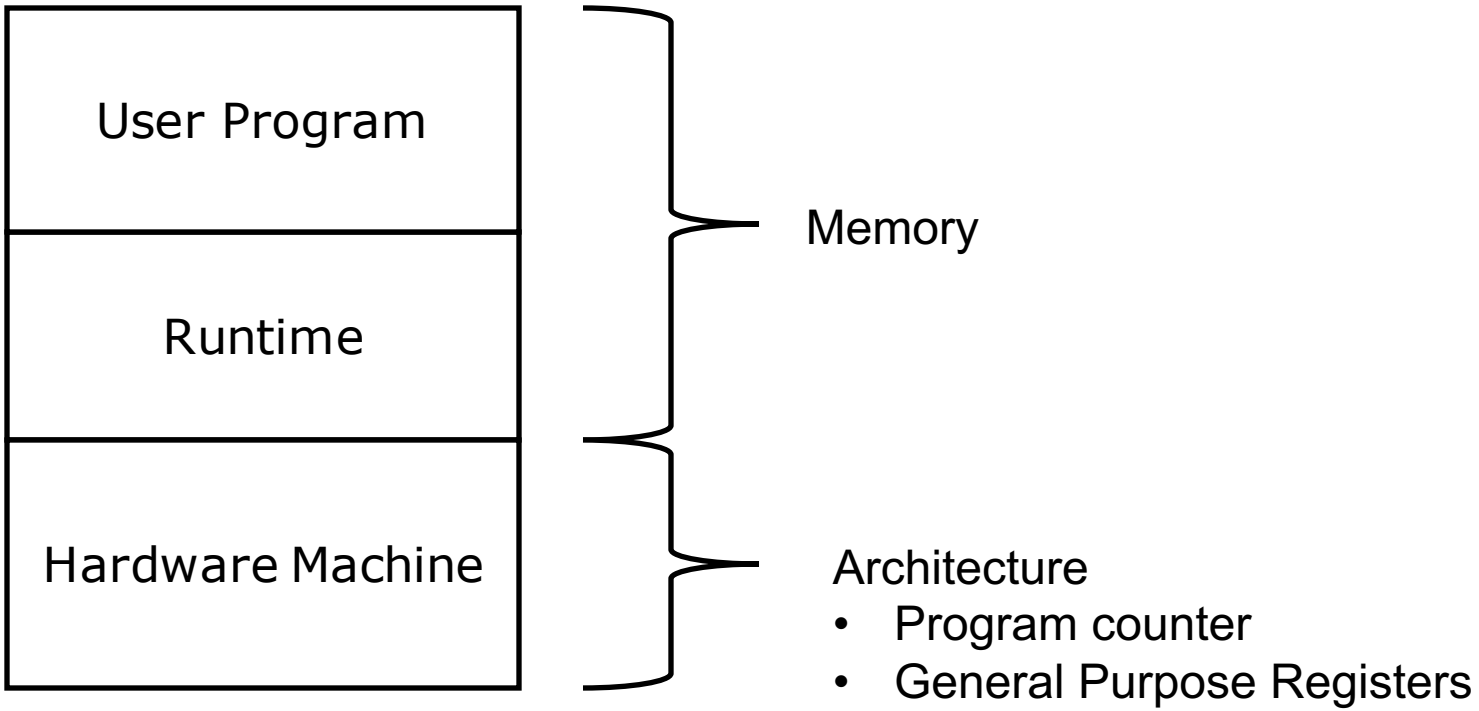
Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Evolution in Number of Users

IBM 1620 1959	IBM 360 1960s	IBM PC 1980s	Cloud Servers 1990s
			
<u>Single User</u>	<u>Multiple Users</u>	<u>Single User</u>	<u>Multiple Users</u>
Runtime loaded with program	OS for sharing resources	OS for sharing resources	Multiple OSs

Single User Machine

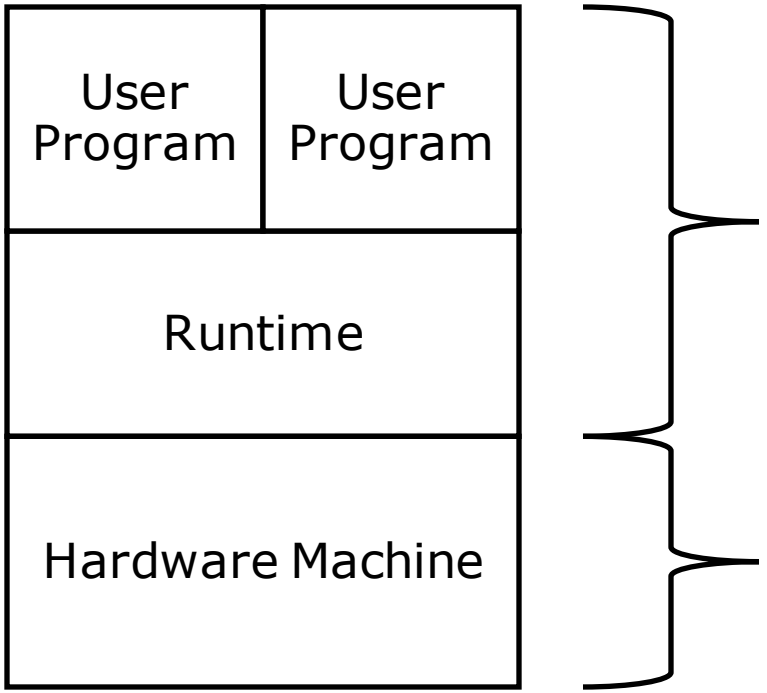
Single user process runs with access to all memory



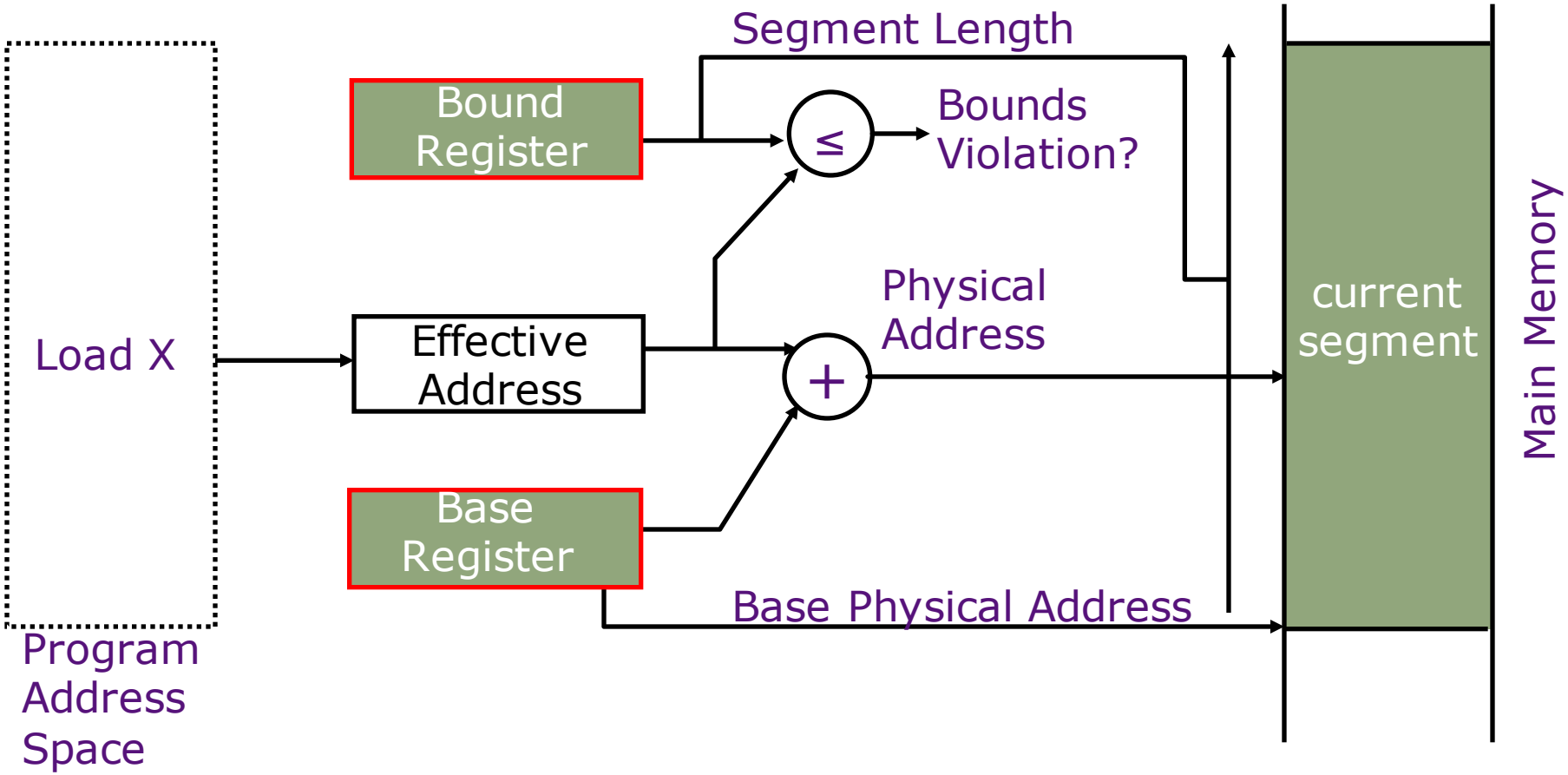
Multi-user systems

Challenge:

Don't want user programs to write over each other.



Simple Base and Bound Translation

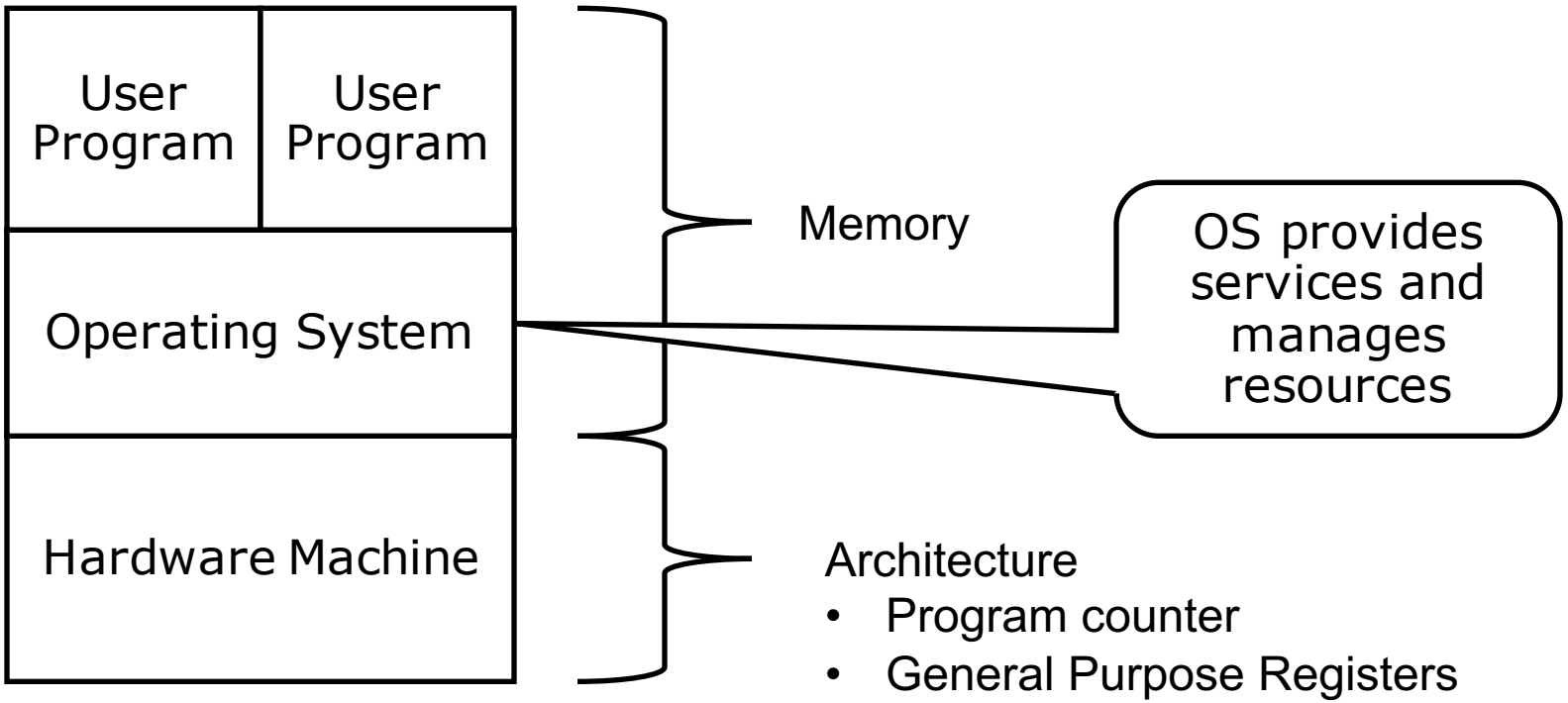


Base and bounds registers are visible/accessible only when processor is running in the *supervisor mode*

Operating System-Based Systems

Challenge:

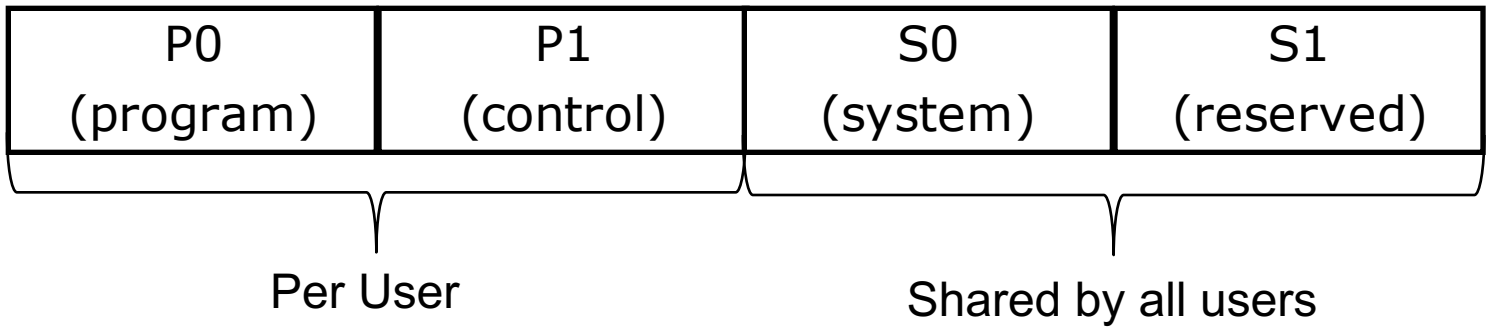
Don't want user programs to write over runtime.



Execution Modes

- Every process runs in a execution mode, e.g.,
  - User (3)
  - Executive (2)
  - Supervisor (1)
  - Kernel (0)
- Each page can allow read/write access based on current mode

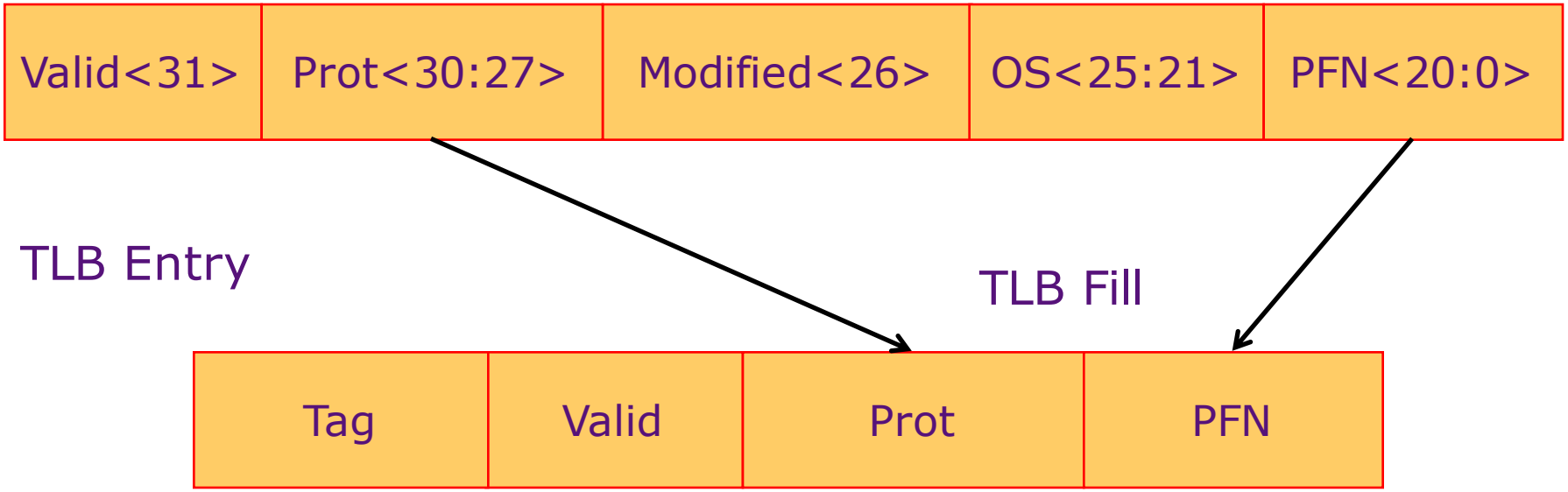
E.g., VAX Memory Partitions



Each partition can have pages with different protections

Protecting Memory

Page Table Entry



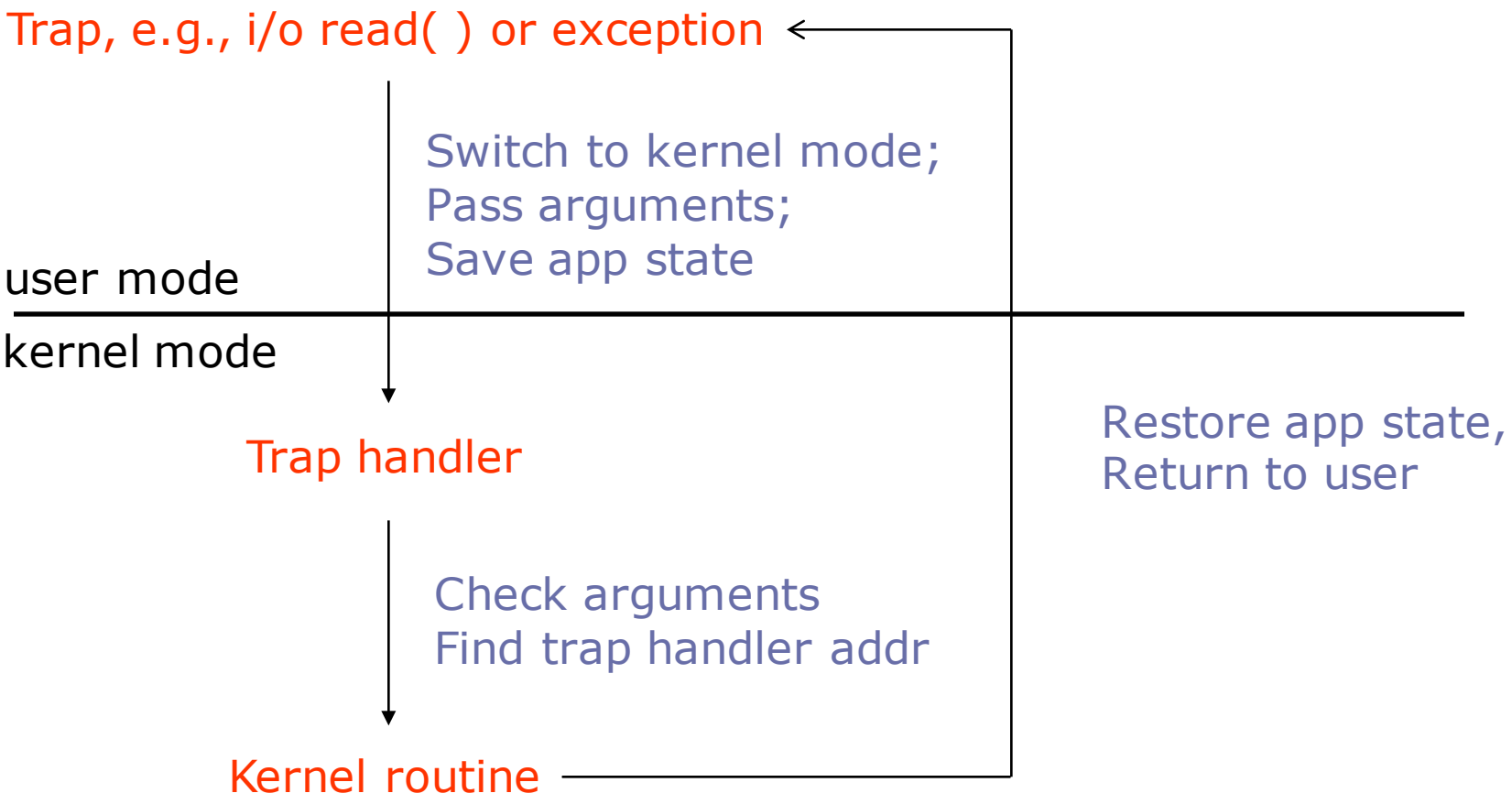
- TLB fill is a privileged operation.
- TLB access checks if protection allows access for current mode

Protection – VAX Modes

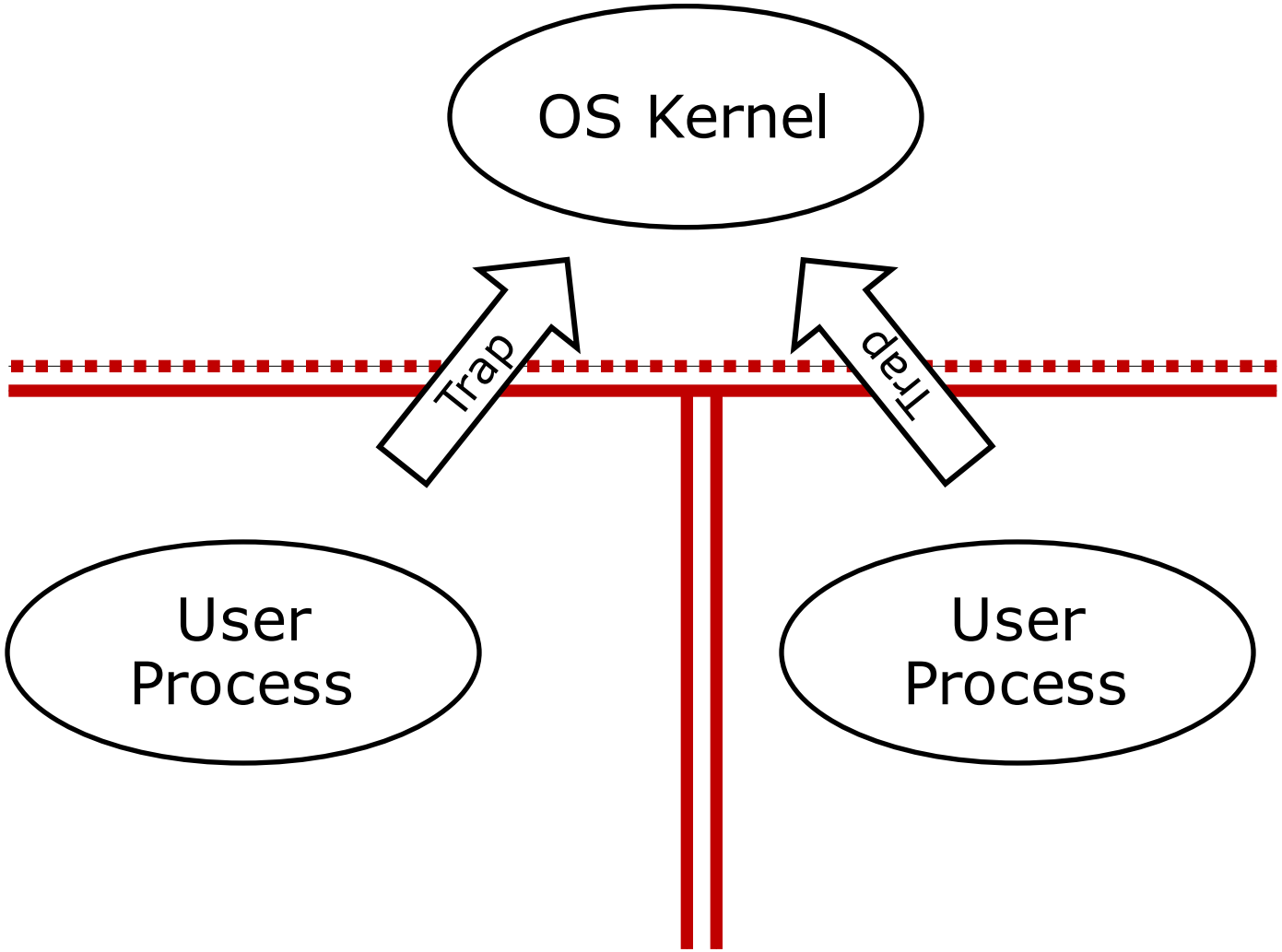
Name	Mnemonic	Decimal	Binary	Kernel	Exec	Super	User
No access	NA	0	000	none	none	none	none
Reserved		1	0001	Unpredictable			
Kernel write	KW	2	0010	write	none	none	none
Kernel read	KR	3	0011	read	none	none	none
User write	UW	4	0100	write	write	write	write
Exec write	EW	5	0101	write	write	none	none
Exec read, kernel write	ERKW	6	0110	write	read	none	none
Exec read	ER	7	0111	read	read	none	none
Super write	SW	8	1000	write	write	write	none
Super read, exec write	SREW	9	1001	write	write	read	none
Super read, kernel write	SRKW	10	1010	write	read	read	none
Super read	SR	11	1011	read	read	read	none
User read, super write	URSW	12	1100	write	write	write	read
User read, exec write	UREW	13	1101	write	write	read	read
User read, kernel write	URKW	14	1110	write	read	read	read
User read	UR	15	1111	read	read	read	read

More powerful modes never loss privileges

Process Mode Switching



Protection – Single OS



Motivation for Multiple OSs

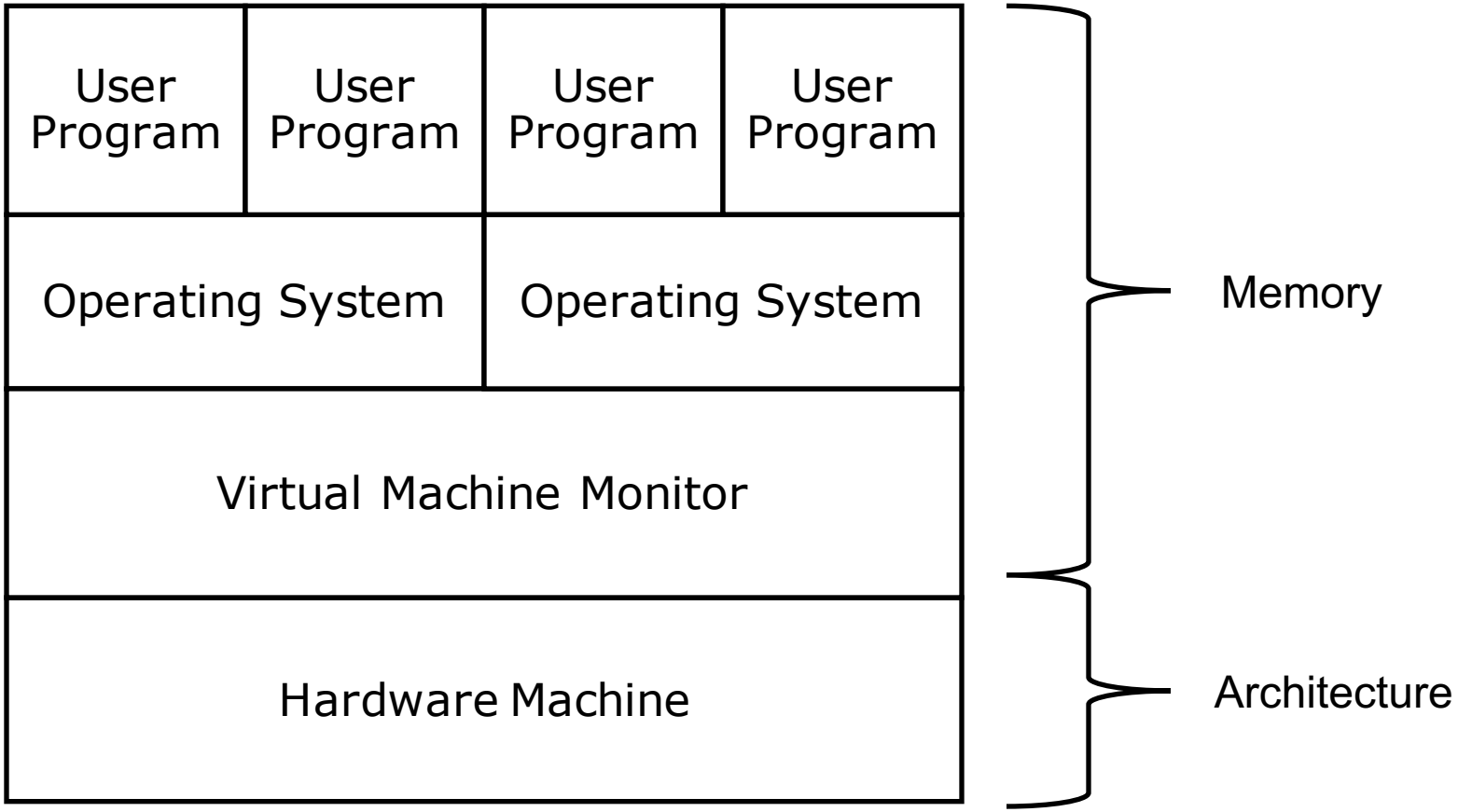
Some motivations for using multiple operating systems on a single computer:

- Allows for graceful operating system upgrades.
- Allows operating system development without making entire machine unstable or unusable.
- Allows use of capabilities of multiple distinct operating systems.
- Allows for load balancing and migration across multiple machines.

Multiple Operating Systems

Challenge:

Want multiple OSs running independently



Virtualization Nomenclature

From (Machine we are attempting to execute)

- Guest
- Client
- Foreign ISA

To (Machine that is doing the real execution)

- Host
- Target
- Native ISA

Virtual Machine Requirements
Popek and Goldberg

- Equivalence/Fidelity: A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
- Resource control/Safety: The VMM must be in complete control of the virtualized resources.
- Efficiency/Performance: A statistically dominant fraction of machine instructions must be executed without VMM intervention.

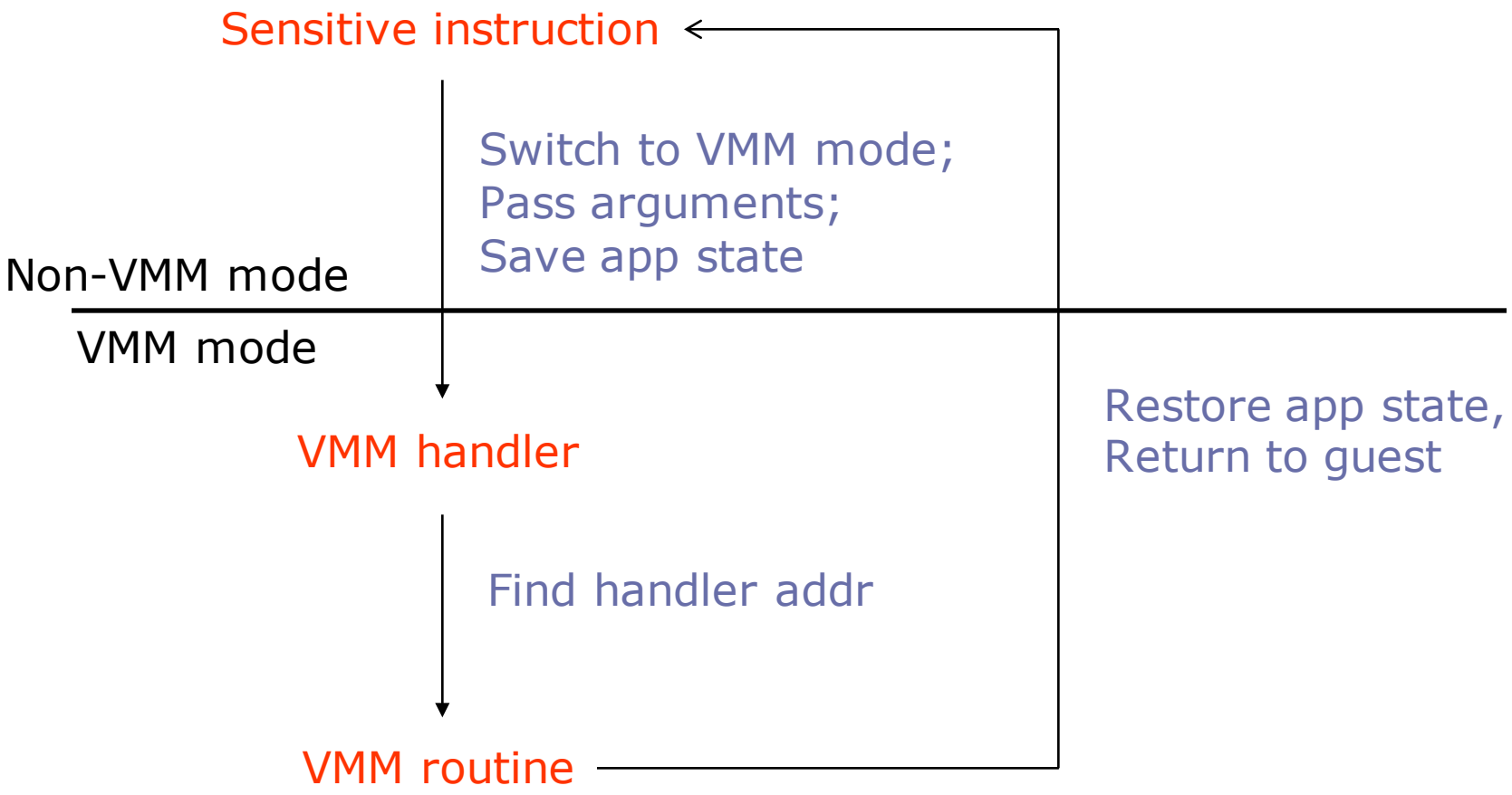
Virtual Machine Requirements
Popek and Goldberg

Classification of instructions into 3 groups:

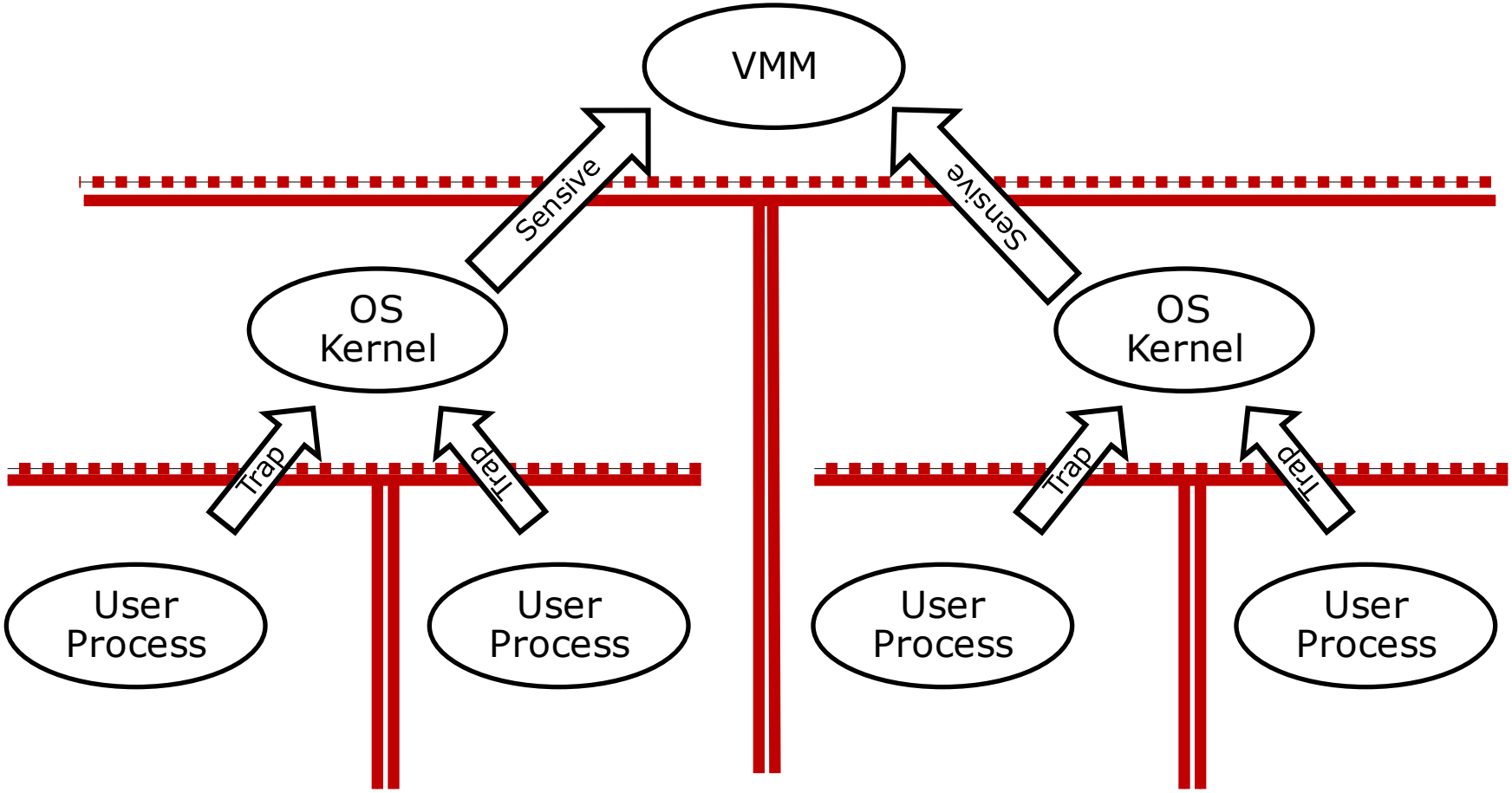
- Privileged instructions: Instructions that trap if the processor is in user mode and do not trap if it is in a more privileged mode.
- Control-sensitive instructions: Instructions that attempt to change the configuration of resources in the system.
- Behavior-sensitive instructions: Those whose behavior depends on the configuration of resources, e.g., mode

Building an *effective* VMM for an architecture is possible if the set of sensitive instructions is a subset of the set of privileged instructions.

Sensitive instruction handling



Protection – Multiple OS



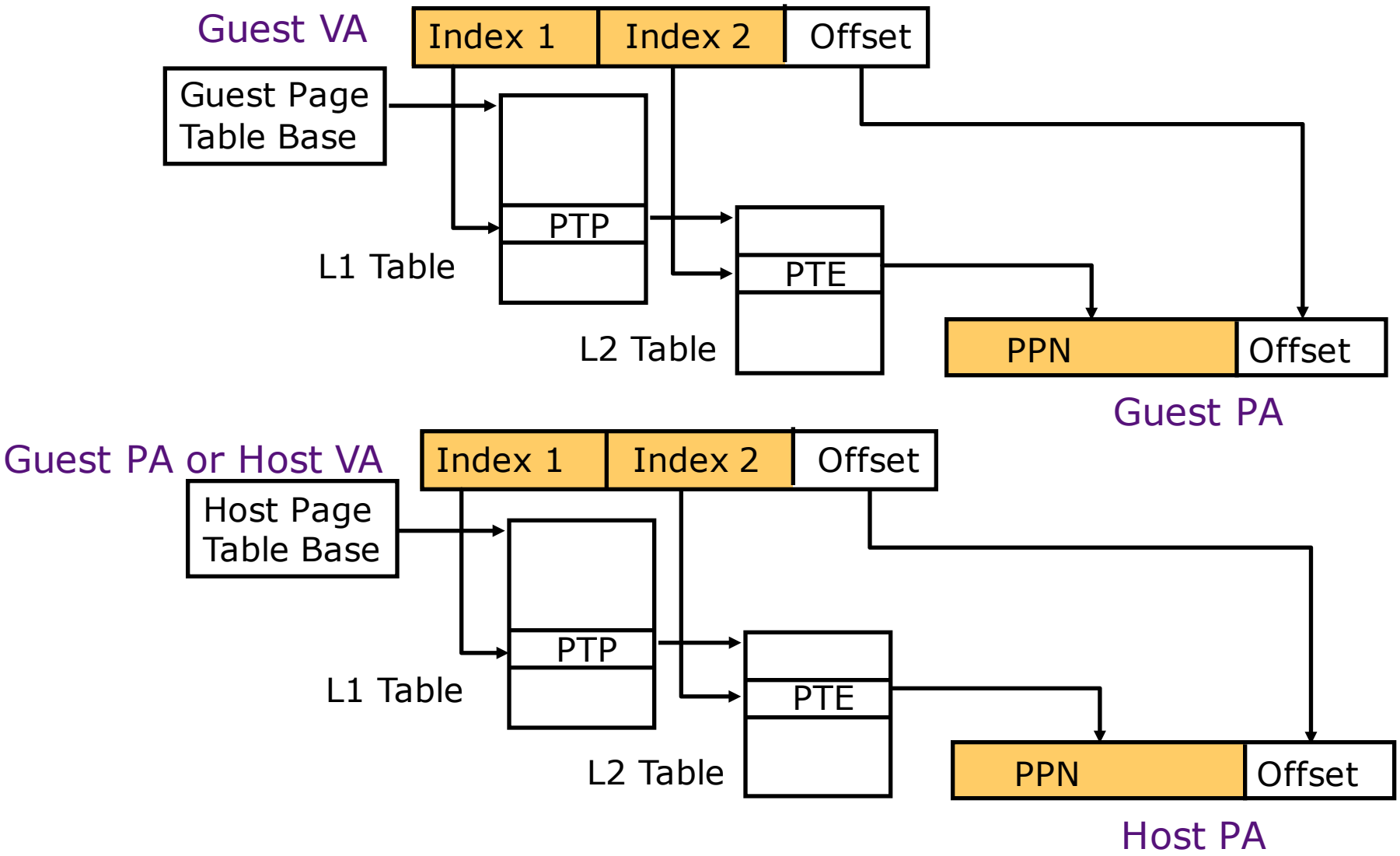
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but…

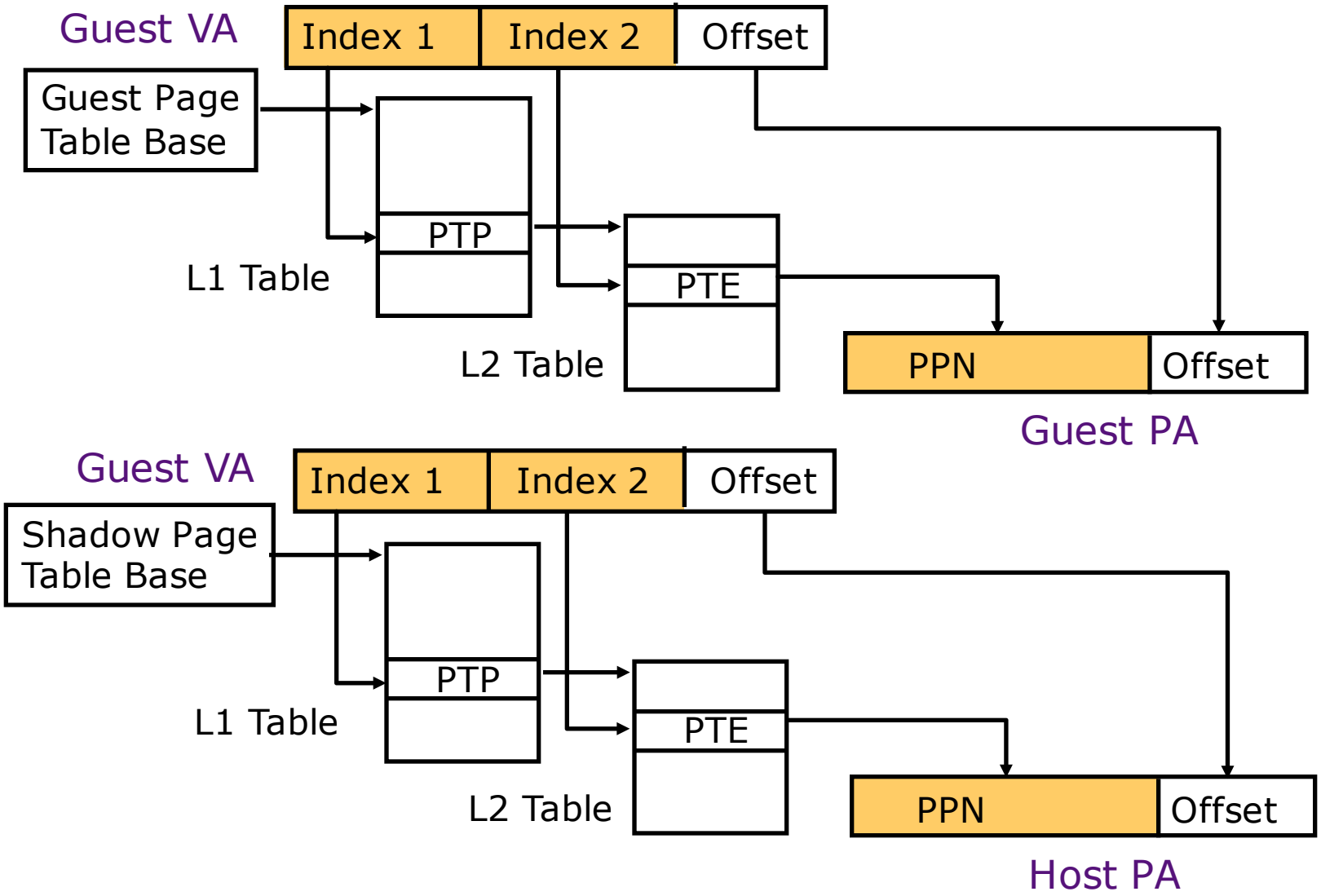
- TLB misses are a 'sensitive' operation
- TLB misses happen very very frequently

- So how expensive are TLB fills…

Nested Page Tables



Shadow Page Tables



Nested vs Shadow Paging

	Native	Nested Paging	Shadow Paging
TLB Hit	VA->PA	gVA->hPA	gVA->hPA
TLB Miss (max)	4	24	4
PTE Updates	Fast	Fast	Uses VMM

On x86

Application-level virtualization

- Programs are usually distributed in a binary format that encodes the program's instructions and initial values of some data segments. These requirements are called the application binary interface (ABI), which can be virtualized

- ABI specifications include
  - Which instructions are available (the ISA)
  - What system calls are possible (I/O, or the *environment*)
  - What state is available at process creation

- Operating system implements the virtual environment
  - At process startup, OS reads the binary program, creates an environment for it, then begins to execute the code, handling traps for I/O calls, emulation, etc.

Partial ISA-level virtualization

Often good idea to implement part of ISA in software:

- Expensive but rarely used instructions can cause trap to OS emulation routine:
  - e.g., decimal arithmetic in μVax implementation of VAX ISA
- Infrequent but difficult operand values can cause trap
  - e.g., IEEE floating-point denormals cause traps in almost all floating-point unit implementations
- Old machine can trap unused opcodes, allows binaries for *new* ISA to run on *old* hardware
  - e.g., Sun SPARC v8 added integer multiply instructions, older v7 CPUs trap and emulate

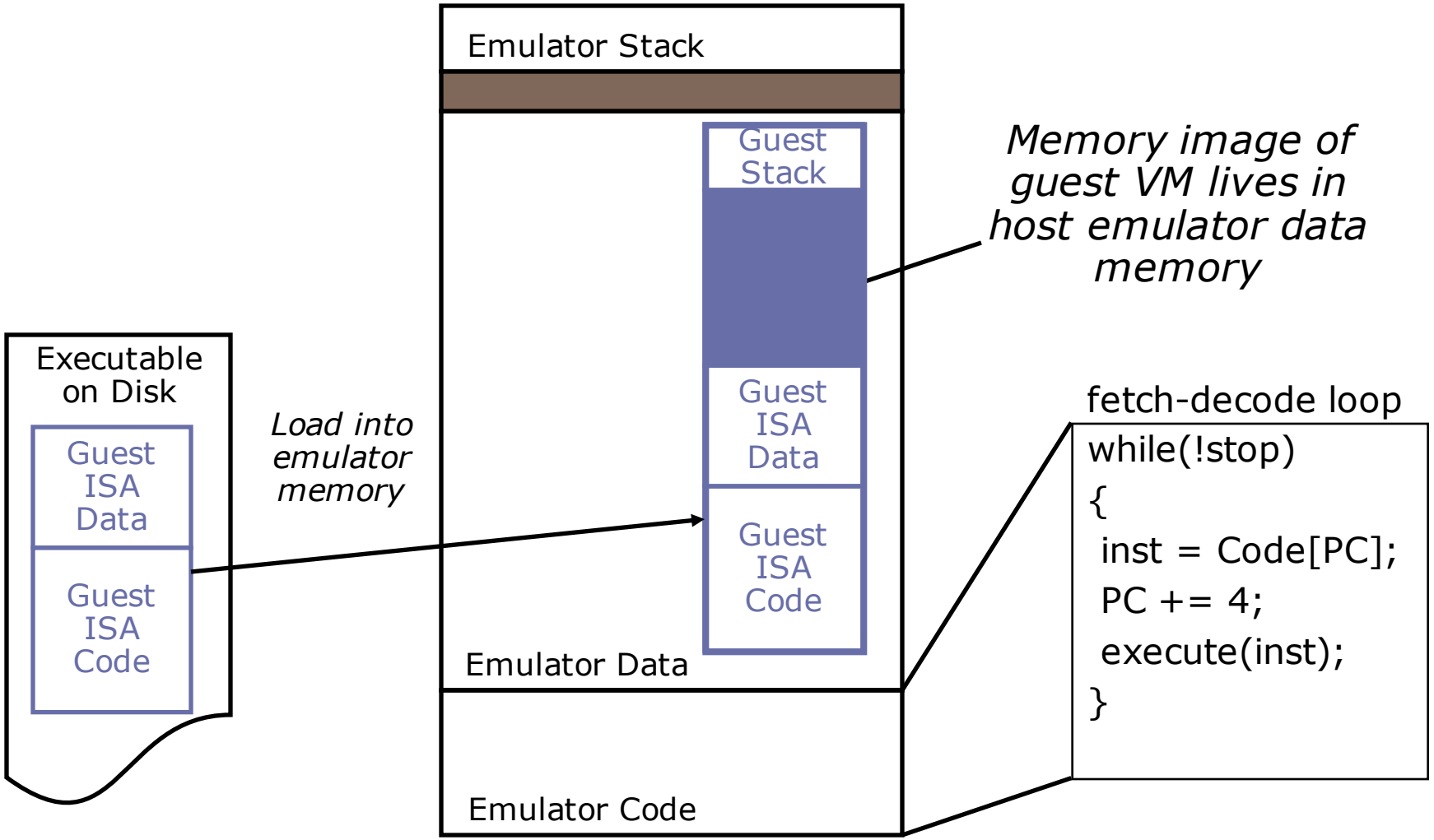
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA

- Run-time Hardware Emulation
  - IBM System 360 had IBM 1401 emulator in microcode
  - Intel Itanium converts x86 to native VLIW (two software-visible ISAs)
  - ARM cores support 32-bit ARM, 16-bit Thumb, and JVM (three software-visible ISAs!)
- Emulation *(OS software interprets instructions at run-time)*
  - E.g., OS for PowerPC Macs had emulator for 68000 code
- Static Binary Translation *(convert at install time, load time, or offline)*
  - IBM AS/400 to modified PowerPC cores
  - DEC tools for VAX->Alpha and MIPS->Alpha
- Dynamic Binary Translation *(non-native ISA to native ISA at run time)*
  - Sun's HotSpot Java JIT (just-in-time) compiler
  - Transmeta Crusoe, x86->VLIW code morphing

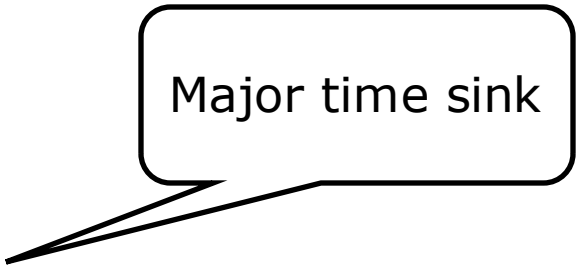
Emulation

- Software instruction set interpreter fetches and decodes one instruction at a time in emulated VM



Emulation

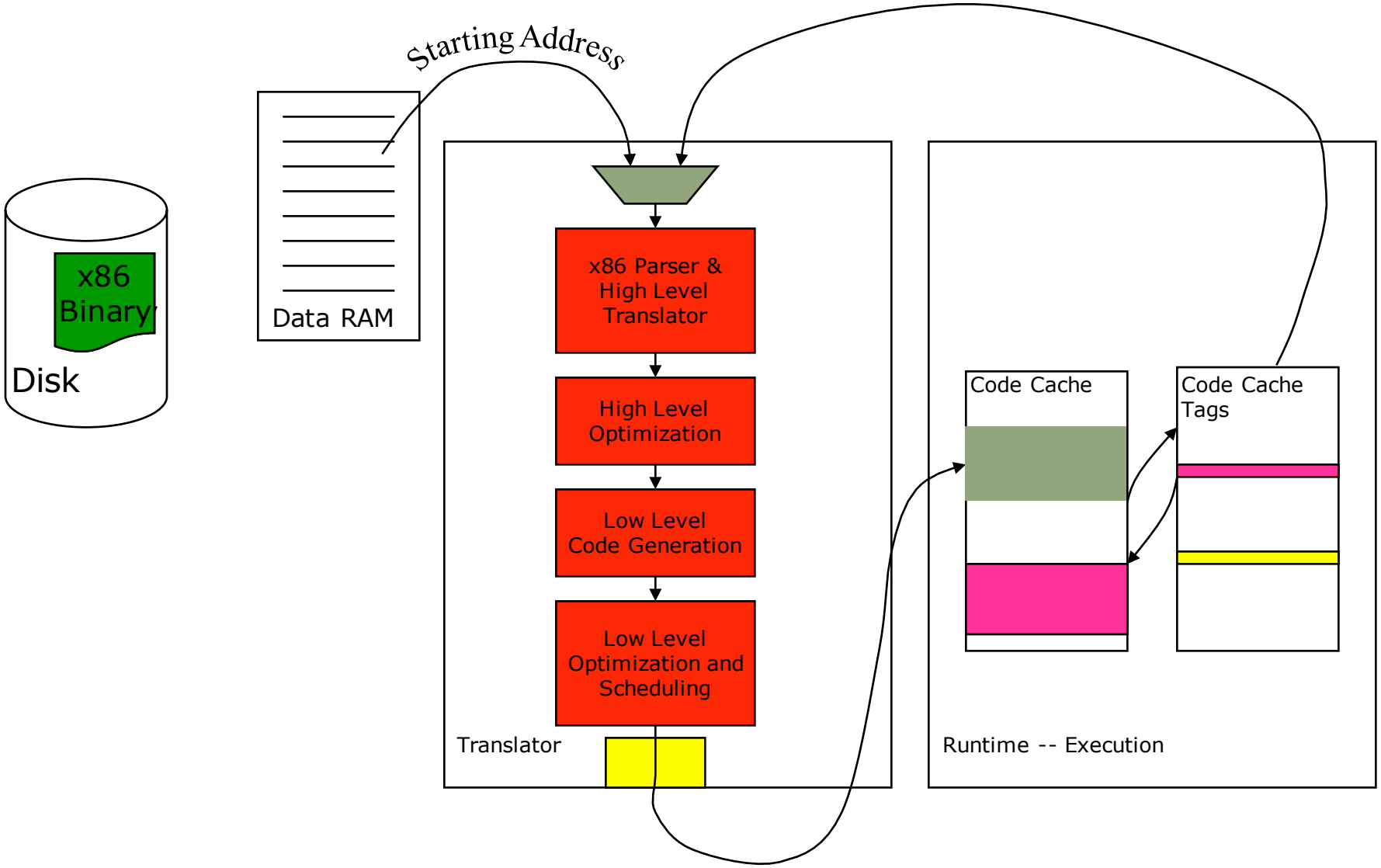
- Easy to code, small code footprint, but *slow*, approximately 100x slower than native execution for RISC ISA hosted on RISC ISA
  - Fetch
    - Instruction accessed from memory
  - Decode
    - Opcode decode with switch tables
    - Extract register specifiers using bit shifts
  - Register access
    - Access register file data structure
  - Execute operation
  - Next PC calculation
    - return to main fetch loop



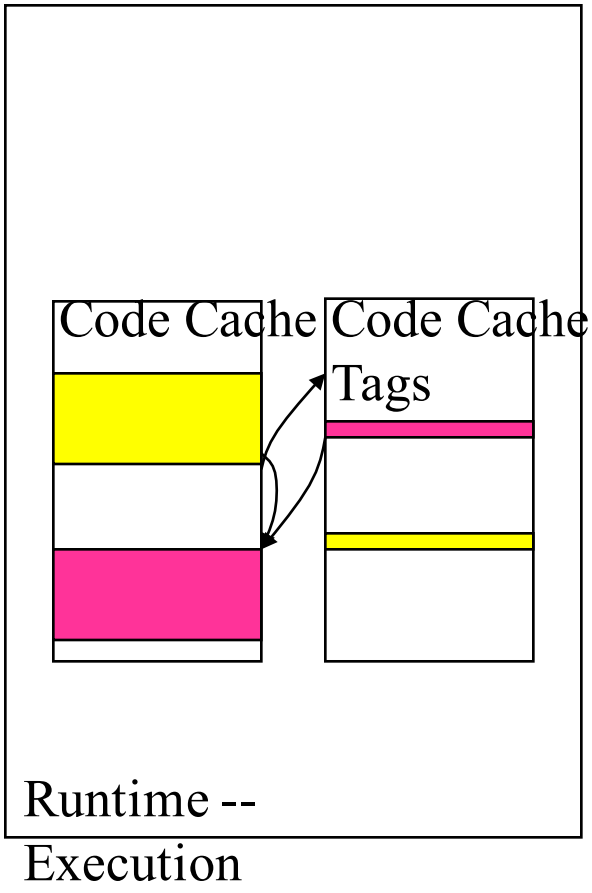
Dynamic Binary Translation

- Translate code sequences as needed at run-time, but cache results
- Can optimize code sequences based on dynamic information (e.g., branch targets encountered)
- Tradeoff between optimizer run-time and time saved by optimizations in translated code
- Technique used in Java JIT (Just-In-Time) compilers and PIN dynamic instrumentation system
- Also, Transmeta Crusoe for x86 emulation

Dynamic Translation Example



Chaining



Pre Chained

```
add %r5, %r6, %r7
li %next_addr_reg, next_addr #load address
                             #of next block
j dispatch loop
```

Chained

```
add %r5, %r6, %r7
j physical location of translated
        code for next_block
```

Dynamic Translation Example: Transmeta Crusoe (2000)

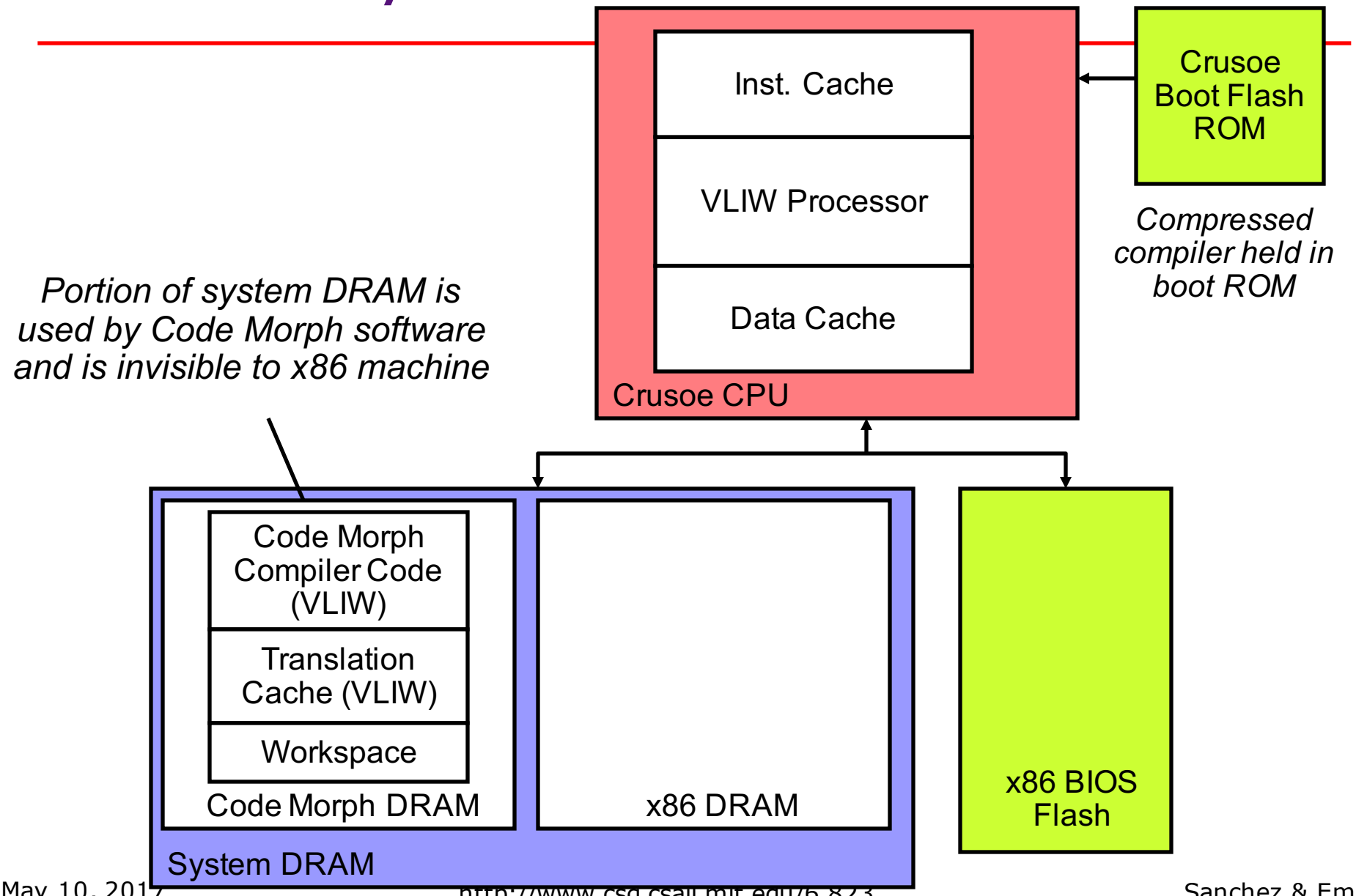
- Converts x86 ISA into internal native VLIW format using software at run-time ➔ "Code Morphing"
- Optimizes across x86 instruction boundaries to improve performance
- Translations cached to avoid translator overhead on repeated execution
- Completely invisible to operating system – looks like x86 hardware processor

[Following slides contain examples taken from "The Technology Behind Crusoe Processors", Transmeta Corporation, 2000]

Transmeta VLIW Engine

- VLIW engine optimized for x86 code emulation
  - evaluates condition codes the same way as x86
  - has 80-bit floating-point unit
  - partial register writes (update 8 bits in 32 bit register)
- Support for fast instruction writes
  - run-time code generation important
- Native ISA difference invisible to user, hidden by translation system
  - Low-end 2-wide VLIW (TM3120)
  - Higher-end 4-wide VLIW (TM5400)
  - Top-end 8-wide VLIW (Efficeon/TM8000 series)

Crusoe System



Transmeta Translation

x86 code:

```
addl %eax, (%esp) # load data from stack, add to eax
addl %ebx, (%esp) # load data from stack, add to ebx
movl %esi, (%ebp) # load from ebp* into esi
subl %ecx, 5      # sub 5 from ecx
```

first step, translate into RISC ops:

```
ld %r30, [%esp]        # load from stack into temp1
add.c %eax, %eax, %r30 # add to eax, set cond.codes
ld %r31, [%esp]        # load from stack into temp2
add.c %ebx, %ebx, %r31 # add to ebx, set cond.codes
ld %esi, [%ebp]        # load from epb* into esi
sub.c %ecx, %ecx, 5    # subtract 5 from ecx
```

Compiler Optimizations

RISC ops:

```
ld %r30, [%esp]        # load from stack into temp1
add.c %eax, %eax, %r30 # add to eax, set cond.codes
ld %r31, [%esp]        # load from stack into temp2
add.c %ebx, %ebx, %r31 # add to ebx, set cond.codes
ld %esi, [%ebp]        # load from epb* into esi
sub.c %ecx, %ecx, 5    # subtract 5 from ecx
```

Optimize:

```
ld %r30, [%esp]        # load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30   # reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5    # only this cond. code needed
```

Scheduling

Optimized RISC ops:

```
ld %r30, [%esp]       # load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30  # reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5   # only this cond. code needed
```

Schedule into VLIW code:

```
ld %r30, [%esp]; sub.c %ecx, %ecx, 5
ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```

Translation Overhead

- Highly optimizing compiler takes considerable time to run, adds run-time overhead
- Only worth doing for frequently executed code
- Translation adds instrumentation into translations that counts how often code executed, and which way branches usually go
- As count for a block increases, higher optimization levels are invoked on that code

Exceptions

Original x86 code:

```
addl %eax, (%esp) # load data from stack, add to eax
addl %ebx, (%esp) # load data from stack, add to ebx
movl %esi, (%ebp) # load esi from memory
subl %ecx, 5      # sub 5 from ecx
```

Scheduled VLIW code:

```
ld %r30, [%esp]; sub.c %ecx, %ecx, 5
ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```

- x86 instructions executed out-of-order with respect to original program flow
- Need to restore state for precise traps

Shadow Registers and Store Buffer

- All registers have working copy and shadow copy
- Stores held in software controlled store buffer, loads can snoop
- At end of translation block, commit changes by copying values from working regs to shadow regs, and by releasing stores in store buffer
- On exception, re-execute x86 code using interpreter

Handling Self-Modifying Code

- When a translation is made, mark the associated x86 code page as being translated in page table
- Store to translated code page causes trap, and associated translations are invalidated

Thank you !