Popek/Goldberg Theorem

Hardware and Software Support For Virtualization

Chapter 2

Introduction

- Virtual machines was an intense research area in late 60/early 70
- **Instruction Set Architecture** (ISA) design was a really hot-topic too in engineering
- In 1974 proposed as a way to detect if new architectures enable or not the construction of VMM (Virtual Machine Monitors)
  - PDP10 and how “seemingly” arbitrarily decision in **ISA design impacts** on VMM (preventing its implementation)
- Later in early/mid 2000s this theorem was used as guide for Intel/AMD to design his virtualization extension (**Pacifica** & **Vanderpool**)

Review: Address Translation

- Hardware transforms a **virtual address** to a **physical address**.
  - The desired information is actually stored in a physical address.
- The OS must get involved at key points to set up the hardware.
  - The OS must manage memory to judiciously intervene.

Review: Address Translation

- C - Language code

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

- **Load** a value from memory
- **Increment** it by three
- **Store** the value back into memory

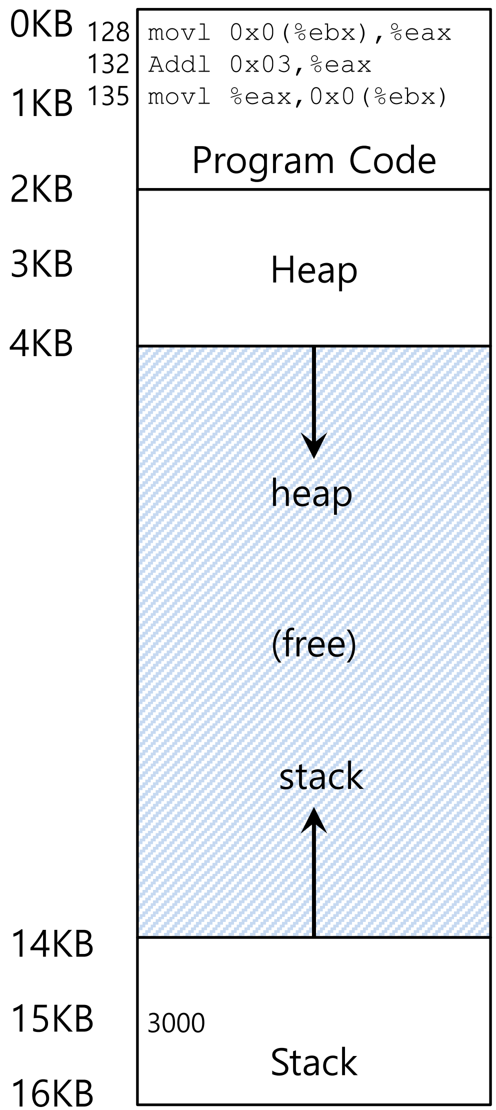
Review: : Address Translation(Cont.)

- Assembly

```
128 : movl 0x0(%ebx), %eax          ; load 0+ebx into eax
132 : addl $0x03, %eax              ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)          ; store eax back to mem
```

  - **Load** the value at that address into `eax` register.
  - **Add** 3 to `eax` register.
  - **Store** the value in `eax` back into memory.

Example: Address Translation(Cont.)



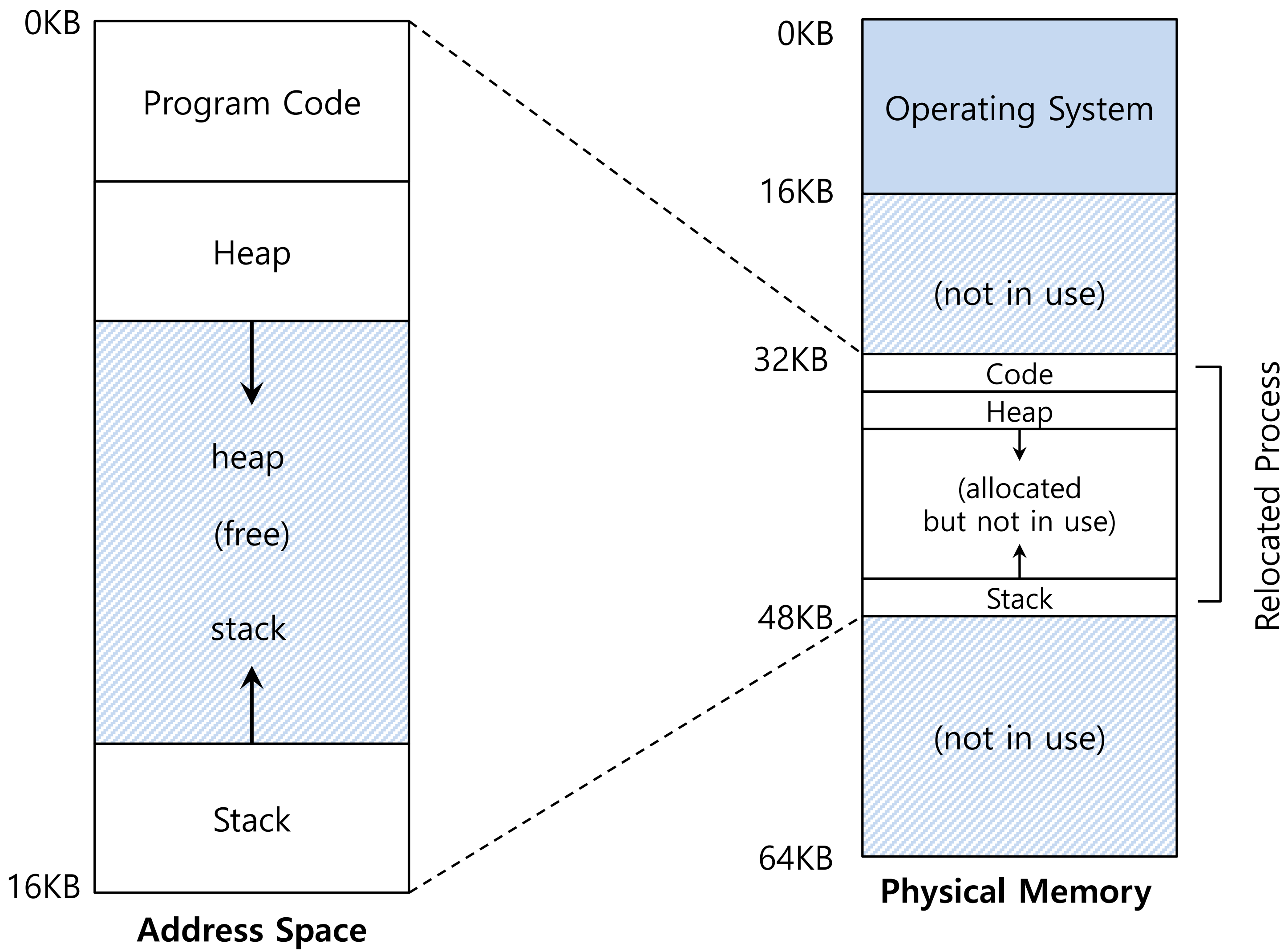
- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

Relocation Address Space

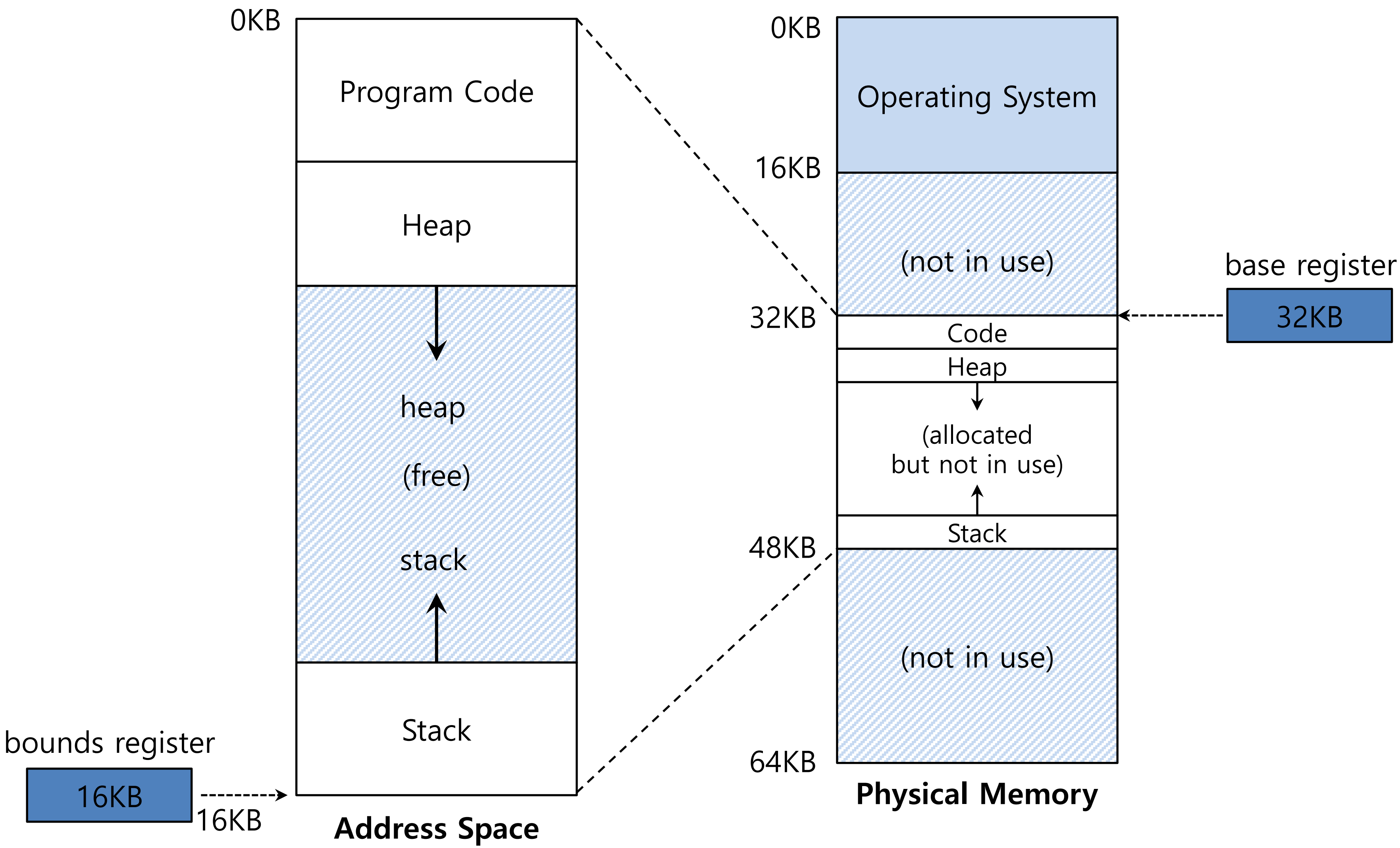
- The OS wants to place the process **somewhere else** in physical memory, not at address 0.
  - The address space start at address 0.

- But how make it **transparently**?

A Single Relocated Process



Base and Bounds Registers



Aside: Software-based Relocation

- If Hardware-support is not present?
  - Static-Relocation
- Loader should "transform" the executable
- Problems
  - No protection
  - Late relocation is hard
- **Hardware support is mandatory!**

Dynamic (Hardware based) Relocation

- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**.
  - Set the **base** register a value.

$$phycal\ address = virtual\ address + base$$

  - Every virtual address must **not be greater than bound** and **negative.**

$$0 \leq \ virtual\ address < bounds$$

  - ISA provide
    - **Privileged** ins to handle those registers
    - Specific **exception** to detect (and handle) miss behaviors

Relocation and Address Translation

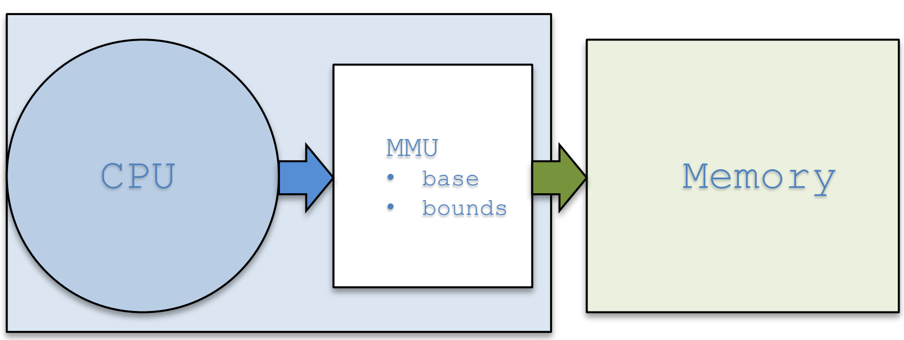
```
128 : movl 0x0(%ebx), %eax
```

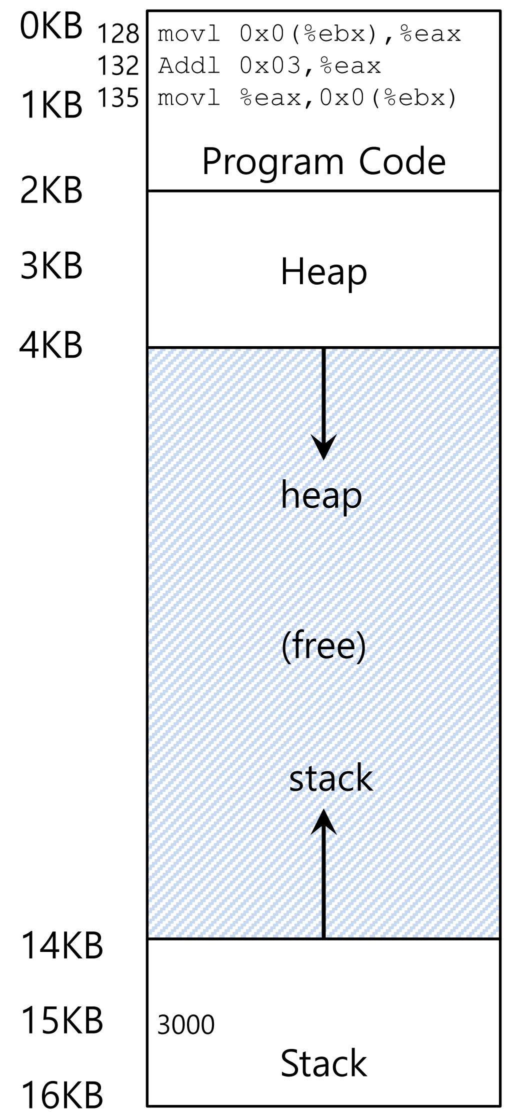
- **Fetch** instruction at address 128

$$32896 = 128 + 32KB(base)$$

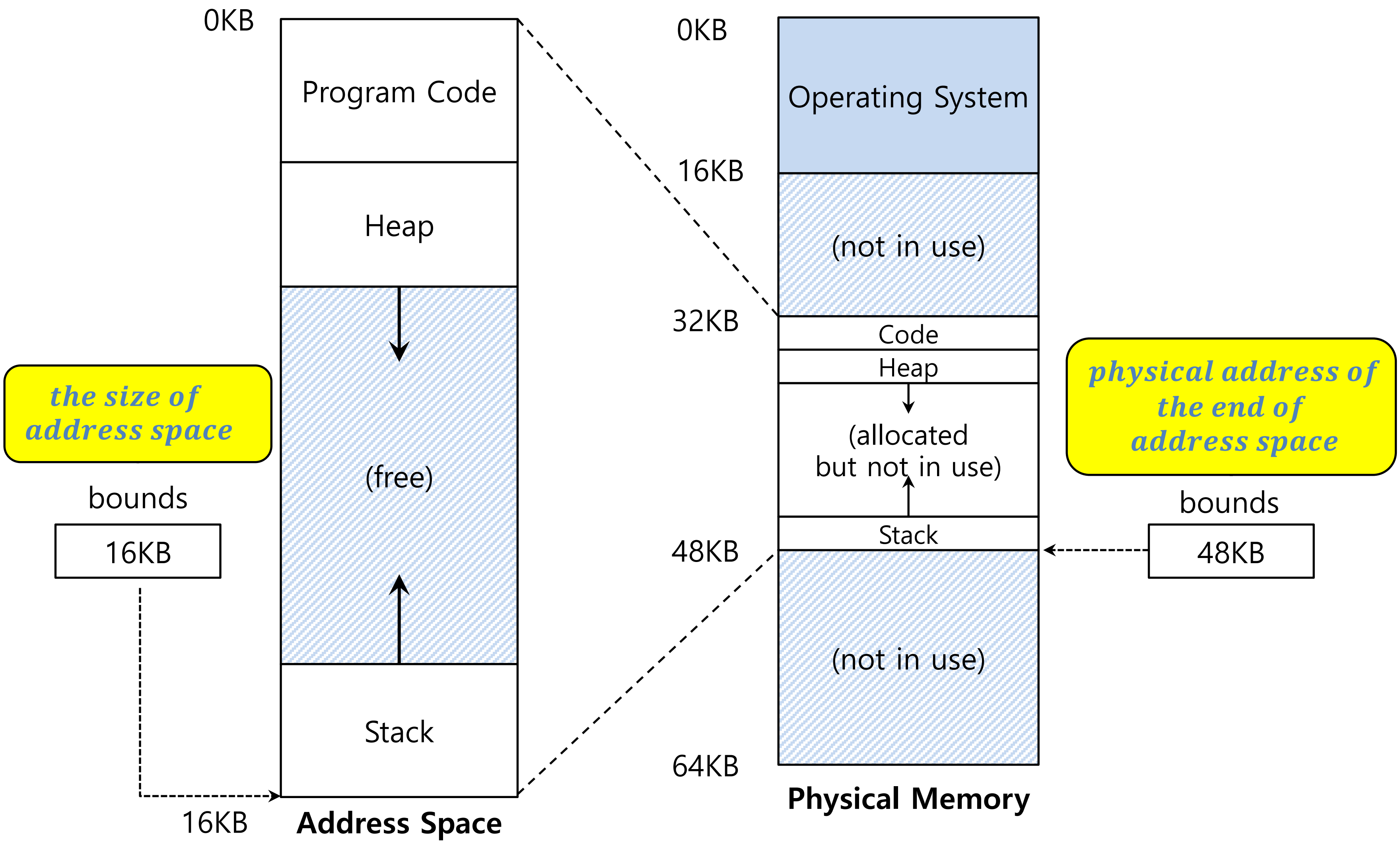
- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$





Two ways of Bounds Register



OS Issues for Memory Virtualizing

- The OS must **take action** to implement **base-and-bounds** approach.
- Three critical junctures:
  - When a process **starts running**:
    - Finding space for address space in physical memory
  - When a process is **terminated**:
    - Reclaiming the memory for use
  - When context **switch occurs**:
    - Saving and storing the base-and-bounds pair

Review: Limited Direction Execution Protocol

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from **proc kernel stack** move to user mode jump to main	
		Run main() … Call system **trap** into OS

Review: Limited Direction Execution Protocol (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to **proc kernel stack** move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from **proc kernel stack** move to user mode jump to PC after trap	
		… return from main trap (via `exit()`)
Free memory of process Remove from process list		

Back to P/G(VM)

The Model

- Proposed in 74 as "Formal Requirements for Virtualizable Third-Generation Architectures"
- **Two execution** modes with **one** processor (**user** and **supervisor**)
- Hardware support for virtual memory, using segments (base register *B* limits *L*)
- Physical memory contiguous and start in `0` and ends on *SZ-1*
- Processor state determined by Processor Status Word (`PSW`), contains
  - Execution level `M={`*u* or *s* `}`
  - Segment register `(B, L)`
  - Program counter virtual address `(PC)`
- Trap architecture has the mechanisms to save in `MEM[1]` the **old** `PSW` and load from `MEM[0]` the **new** `PSW`
- ISA includes instructions to manipulate `PSW`
- I/O and interrupts are ignored (to simplify the discussion)

Assume only OS (in absence of VMM)

- The kernel would run in `M = s` and applications in `M = u`
- During initialization, the kernel sets the trap entry point as
  `MEM[0] ← (M:s, B:0, L:SZ, PC:trap_en)`
- The kernel will allocate a contiguous range of physical memory for each application
- To launch or resume an application (already stored in physical memory `[L,C]`), the operation system would simply
  `PSW←(M:u,L,C,PC)`
- At the trap entry point (`PC == trap_en`), the kernel would first decode de instruction stored in `MEM[1].PC` to determine the cause of the trap and the appropriate actions

Definitions

- Popek and Goldberg Definition

> A virtual machine is taken to be an **efficient**, **isolated duplicate** of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). As a piece of software, a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs running in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

- Equivalence
  - Duplicating real resources
- Safety
  - Isolation between VM and hypervisor
- Performance
  - Separates Hypervisors from Simulators

Question to Address by the Theorem

> Given a computer that meets this basic architectural model, under which precise conditions can a VMM be constructed, so that the VMM:
>
> - can execute one or more virtual machines;
> - is in complete control of the machine at all times;
> - supports arbitrary, unmodified, and potentially malicious operating systems designed for that same architecture; and
> - be efficient to show at worst a small decrease in speed?

- The theorem must confirm compliance with the following criteria:
  - Equivalence
    - Identical behavior in bare metal and VM for any app/OS
    - Except availability of certain resources (e.g., memory)
  - Safety
    - **VMM in full control** of the HW, as if were running in a separate computer (from VM perspective)
    - Different VM (if needed) should be isolated
  - Performance
    - At worst, **minor performance degradation** over bare metal

Theorems

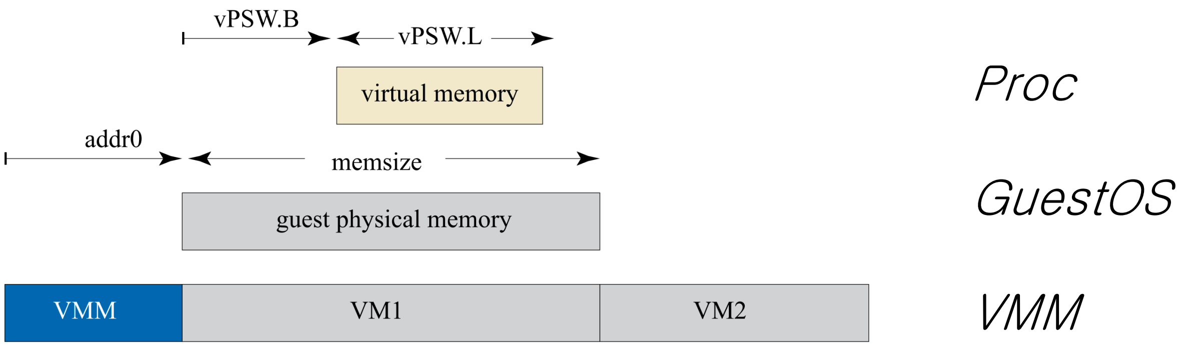
Theorem 1 [143]: For any conventional third-generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

- **Control-sensitive** Instructions
  - It can update the system state
- **Behavioral-sensitive** Instructions
  - Its semantics depend on the system status (**user** or **supervisor**)
- **Innocuous Instructions**
  - Others
- **Privileged Instruction**
  - Can be executed only in supervisor mode

$$\{\boldsymbol{control\_sensitive}] \cup \{\boldsymbol{behavioral\_sensitve}\} \subseteq \{\boldsymbol{privileged}\}$$

Proof by Construction

1. **Only** VMM runs in **supervisor mode**. Reserves a portion of the VMM physical memory for himself (never shared by the VM)
2. VMM allocates a contiguous portion of physical for each VM
3. The VMM keeps in memory a copy of each VM (`vPSW`)
4. Before resume VM, VMM loads in PSW the corresponding state, i.e. `{M',B',L',PC'}`
    - `M'` ← `u`: always in user mode
    - `B'`← `addr0+vPSW.B`: the guest-physical offset is added to the base register VM
    - `L'`← `min{vPSW.L, vPSW.memsize-vPSW.B}`
    - `PC` ← `vPSW.PC`: resumes execution

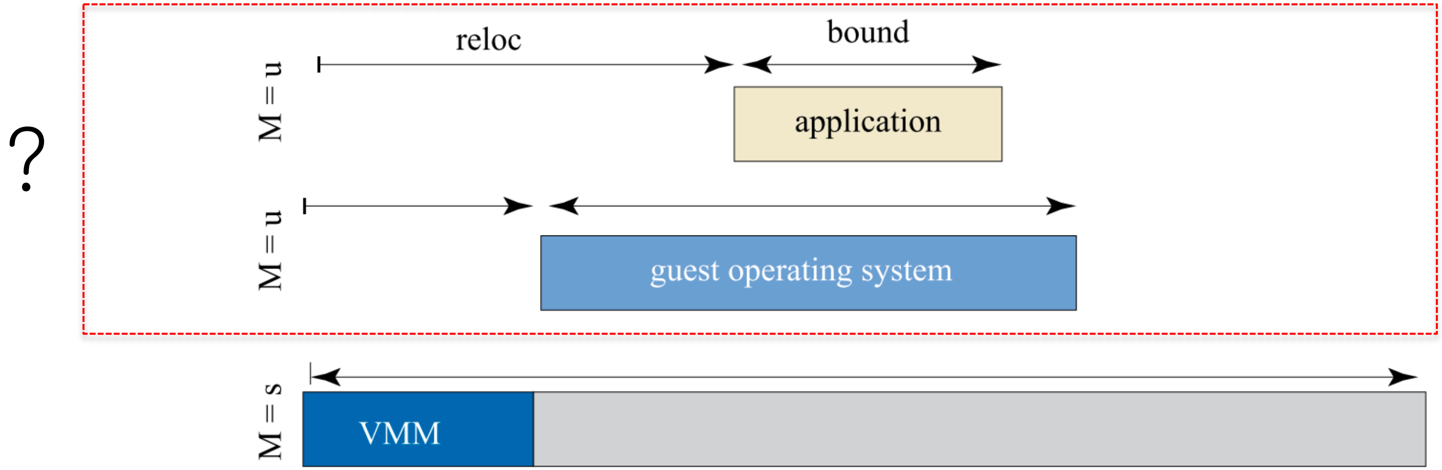


Proof by Construction (...cont.)

5. The VMM perform only `vPSW.PC` $\leftarrow$ `PSW.PC` on **every trap**. Any guest attempt to modify his own `PSW` will result in a trap (control-sensitive are privileged and `PSW.M` $\equiv$ `u`)
6. VMM emulates the behavior of the instruction that caused the trap. If `vPSW.M` $\equiv$ `s` VMM **emulates** the semantics of the instruction according the ISA, including `vPSW` modification, and resumes VM execution when possible (later)
7. Guess traps (e.g., Non-legal instructions, syscalls (i.e., privileged ins when `vPSW.M` $\equiv$ `u`)) should be redirected by the guest
8. Changes in base or bound registers should result in a trap. VMM emulates the desired change

Proof by Construction (...cont.)

9. Any behavioral-sensitive instruction should be privileged, hence when executed by the guest OS will trap, and subsequently emulated by VMM
   - e.g., The outcome of instructions reading `PSW.B` differ from `PSW.M=u` to `PSW.M=s`



Early counter-examples

- A single unprivileged control-sensitive instruction breaks the hypothesis
  - PDP-10 `JRST 1` , return to user mode (motivated the paper)
  - Apparently innocuous if NOP in user mode (from the ISA design standpoint)

- Instructions that reads system state, are behavioral-sensitive, and it use violates the equivalence criteria
  - A user level instruction can read `PSW.M` in a general purpose register → the **guest OS** can conclude that it is in **user mode**! (x86-32!)

- Instructions that bypasses virtual-memory system are behavior sensitive, since their outcome depends upon `PSW.L` and `PSW.B`
  - IBM VM/360 has such instruction. If privileged, not a problem (not the case)

What is a VMM?

- Proof-by-construct, allows to conclude that VMM is basically an OS, since both:
  - Let's the untrusted component run directly in hardware
  - Judiciously intervene to **retain control**

- OS requires HW support to run efficiently (e.g., u/s, timers, MMU, …), VMM too?
  - VMM runs OS that runs application. Seems to be the case…

Recursive and Hybrid Virtual machines

- Complements Theorem 1 postulates

- **Theorem 2**
  - If an ISA meets Theorem 1 hypotheses, it allows to create VM recursively (i.e., the guestOS can be another VMM)

- **Theorem 3**
  - If an ISA don't meet 1 hypotheses, still is possible to build a hybrid virtual machine monitor if the set of **user-sensitive** instructions are a subset of the the set of **privileged instructions** (*user-sensitive if his behavior is **CS** or **BS** in supervisor mode but **not** in user mode*)

- A Hybrid-VMM acts as a:
  - Normal VMM is the VM is running **user level code** (applications)
  - Interprets **100%** of the **system-level code** of the guest (OS itself). Performance criteria is not violated if OS code is not relevant for the workload and architecture

What about paging?

- Memory virtualization is compulsory for any kind of virtualization support
  - Original theorems use relocation (single segment)

- Paging introduces some subtleties
  - Instructions that access to memory can be **memory-sensitive** (VMM is also present in memory)
  - Guest OS should remain privileged w.r.t. the applications
  - Issues addressed with **address space compression** and **ring compression** (x86)

- VMM must compose the virtual address space of the processes
  - Requires to combine the page tables of the guest (virtual to) and the page tables of the VMM (**guest-physical** to **host-physical**)
  - Much harder than with address relocation. **Shadow Page Table**

Well known Violations

- MIPS
  - Three execution modes: **kernel** mode, **supervisor** mode, **user** mode
  - **Only kernel can execute privileged instructions**
  - **Supervisor** mode is a “user” mode with access to KSSEG
  - Great! Run guest-OS as supervisor! (guest-OS→guest-user don’t require TLB flushes, or changes in virtual memory, such as page-table pointer changes)
  - Split address space in regions, that are **location-sensitive** (form of behavior-sensitive)

Region	Base	Length	Access K,S,U	MMU	Cache
USEG	0x0000 0000	2 GB	✓,✓ ✓	mapped	cached
KSEG0	0x8000 0000	512 MB	✓,x,x	unmapped	cached
KSEG1	0xA000 0000	512 MB	✓,x,x	unmapped	uncached
KSSEG	0xC000 0000	512 MB	✓,✓,x	mapped	cached
KSEG3	0xE000 0000	512 MB	✓,x,x	mapped	cached

  - **Its not virtualizable** (every guest-OS load store might require a trap, won’t meet efficiency criteria) since guest-OS expect to run within KSEG0/KSEG1

X86-32

- Complex architecture (with many compatibility intricacies, segmented paging). Baroque privilege management (protection rings, call gates, etc...)

- Many instructions are sensitive and unprivileged (critical). Not virtualizable. Identified 17 instructions [2001]

Group	Instructions
Access to interrupt flag	`pushf, popf, iret`
Visibility into segment descriptors	`lar, verr, verw, lsl`
Segment manipulation instructions	`pop <seg>, push <seg>, mov <seg>`
Read-only access to privileged state	`sgdt, sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump, retfar, str, int <n>`

- Example `popf`, `pushf`
  - Gets/puts from the stack the EFLAGS register
  - EFLAGS includes Z, N, ... , but also DPL (mode). In user mode only writes a portion of the register!

ARM

- One user level and many supervisors (e.g., 7 modes in ARMv6, **1 in ARMv8**)
  - Each one (exception level) with independent registers bank. 24 critical instructions [2010]
- Even ARMv8 has critical instructions. Instructions to deal with user mode regs, status regs, memory access depends on CPU mode

Description	Instructions
User mode	LDM (2), STM (2)
Status registers	CPS, MRS, MSR, RFE, SRS, LDM (3)
Data processing	ADCS, ADDS, ANDS, BICS, EORS, MOVS, MVNS, ORRS, RSBS, RSCS, SBCS, SUBS
Memory access	LDRBT, LDRT, STRBT, STRT

- **Unpredictable** instructions in user mode
- Example Status Registers:
  - Current Program Status Register (CPSR) and Saved Program Status Register (SPSR)
  - CPSR→SPSR in mode escalation, SPSR→CPSR in mode de-escalation
  - `MRS` can be used in any mode to read CPSR (no longer in v8) [Control-S]
  - `CPS` can be used to write CPSR in any mode (ignored in user mode) [Behavior-S]