ADVANCED TOPICS: VIRTUAL MACHINES

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Project 5 happened? → last week

Project 6 – last project!

- Early deadline this week! → Tomorrow
- Final deadline end of next week → Friday

Shivaram office hours

- TODAY at 1pm!
  ↳ ~ 2hrs

AGENDA / LEARNING OUTCOMES

How to virtualize a machine underneath the OS?

PERSISTENCE RECAP

- Managing I/O devices significant part of OS
- Disk Drives, SSDs (pages, blocks)

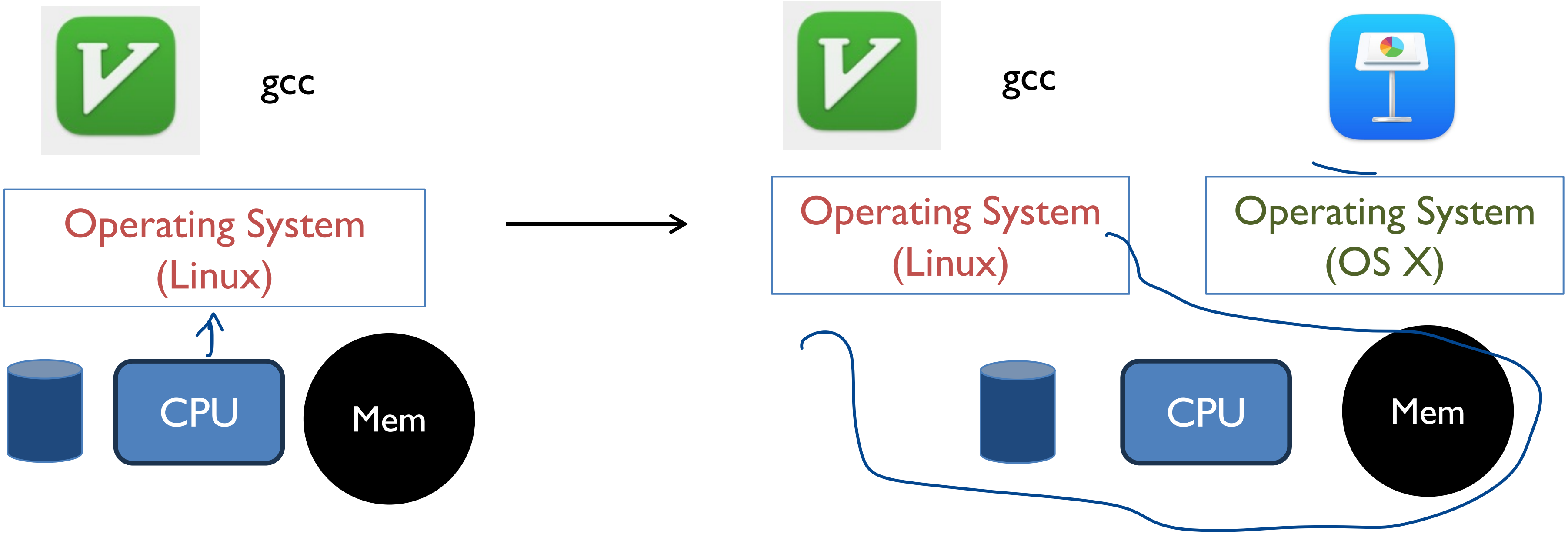
- File Systems: OS provided API to access disk
- Simple FS: FS layout with supberblock, bitmaps, inodes, datablocks

→ Project 6

- Fast File System: Key idea – put inode & data close together, namespace locality
- FSCK, Journaling – Handling/Preventing data inconsistencies

- Log Structured File System - Organize data based on writes

VIRTUAL MACHINES



VIRTUAL MACHINE USE CASES

Share mainframe systems (1970s)

80s / 90s personal computing

Cloud Computing

- Consolidate multiple tenants running different OS
- Strong Isolation

↳ tenants do not interfere with each

tenants → computer

Data Center

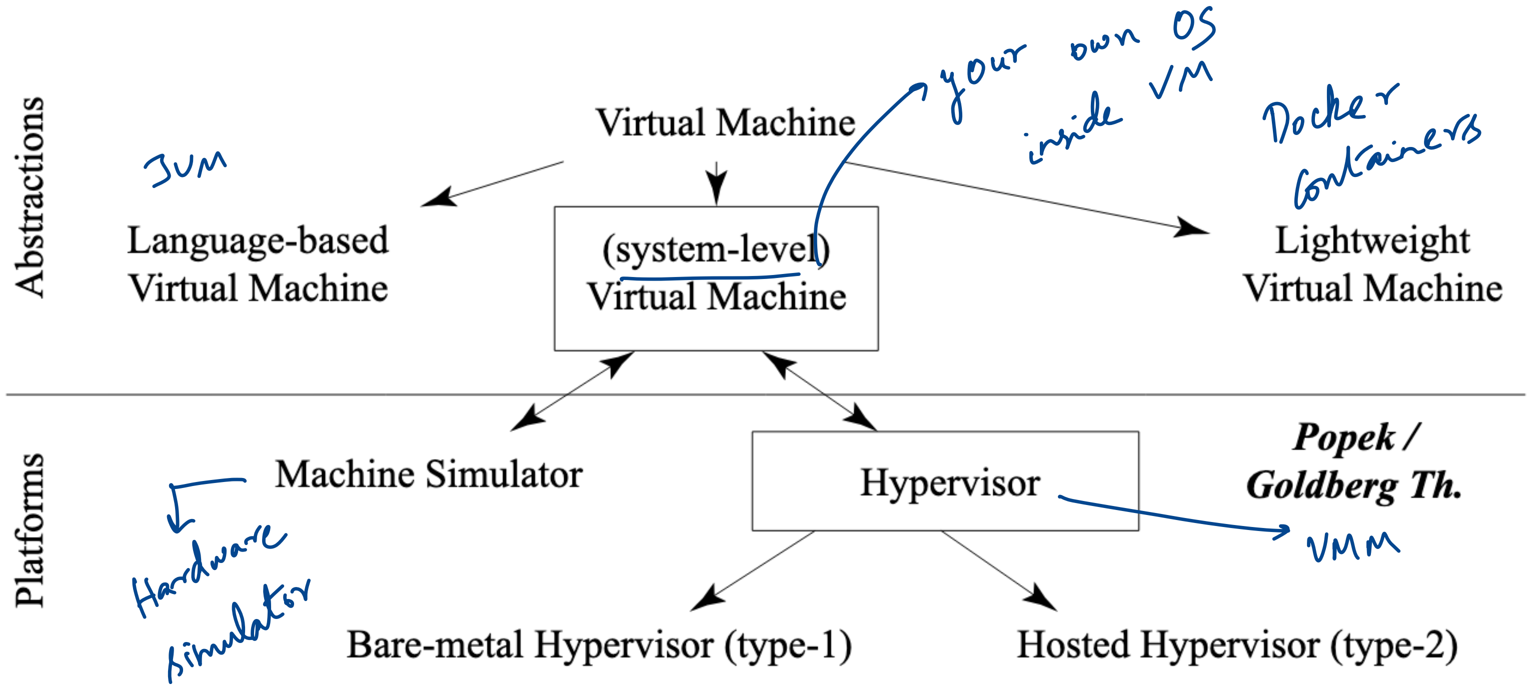
Run applications that only exist for specific OS

Testing, Debugging

↳ want to check if your code works on a diff OS.

DEFINITIONS

A virtual machine is a complete compute environment with its own isolated processing capabilities, memory, and communication channels.



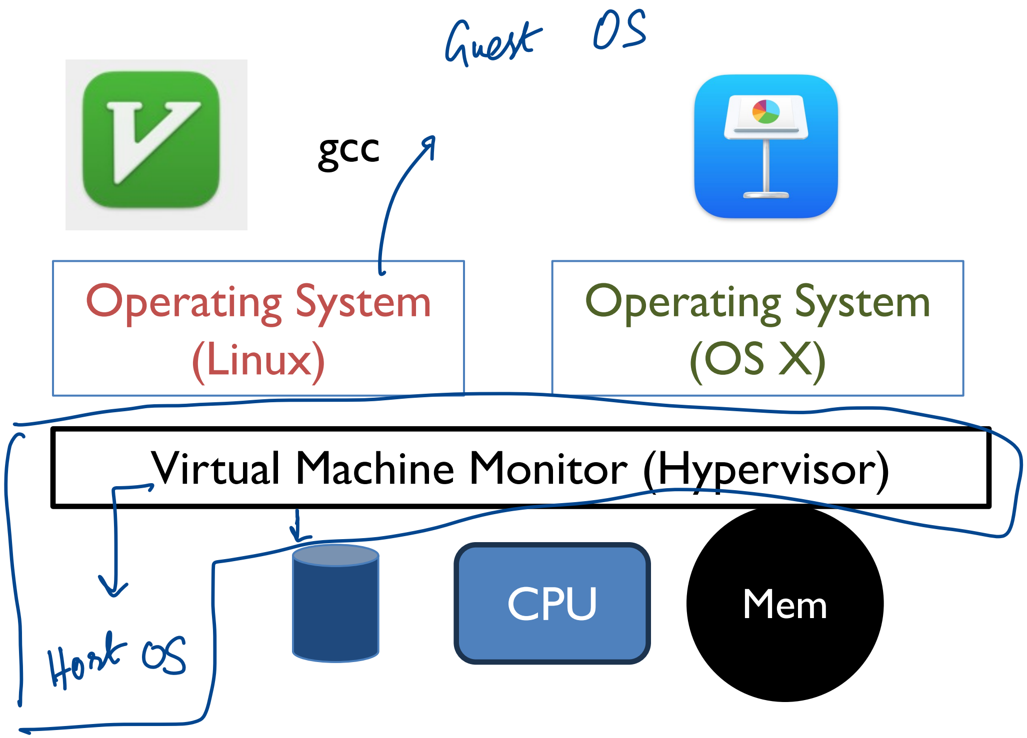
VIRTUAL MACHINE MONITORS

Bare-metal Hypervisor (type-1)
direct control of all resources

Xen

Hosted Hypervisor (type-2)
operates as part of or on top of an existing host OS

KVM → part of Linux



GOALS

- Equivalence – The exposed resource is equivalent with the underlying computer.

  → Diff guest OS

- Safety – Isolation requires that the virtual machines are isolated from each other as well as from the hypervisor.

- Performance – The virtual system must show at worst a minor decrease in speed.

CAN WE VIRTUALIZE? (POPEK GOLDBERG 1974)

The processor's system state, called the processor status word (PSW) consists of the tuple (M, B, L, PC):

kernel or user mode

- the execution level M = {s, u} (superuser or user mode)
- the segment register (B,L); (Segmented Memory Model) and
- the current program counter (PC), a virtual address

B → base
L → limit

A virtual machine monitor may be constructed if the set of sensitive instructions for a computer is a subset of the set of privileged instructions.

$$\{control\text{-}sensitive\} \cup \{behavior\text{-}sensitive\} \subseteq \{privileged\}.$$

control-sensitive → instructions which can change state

behavior-sensitive → can be modified by state

privileged → can only be run in kernel mode

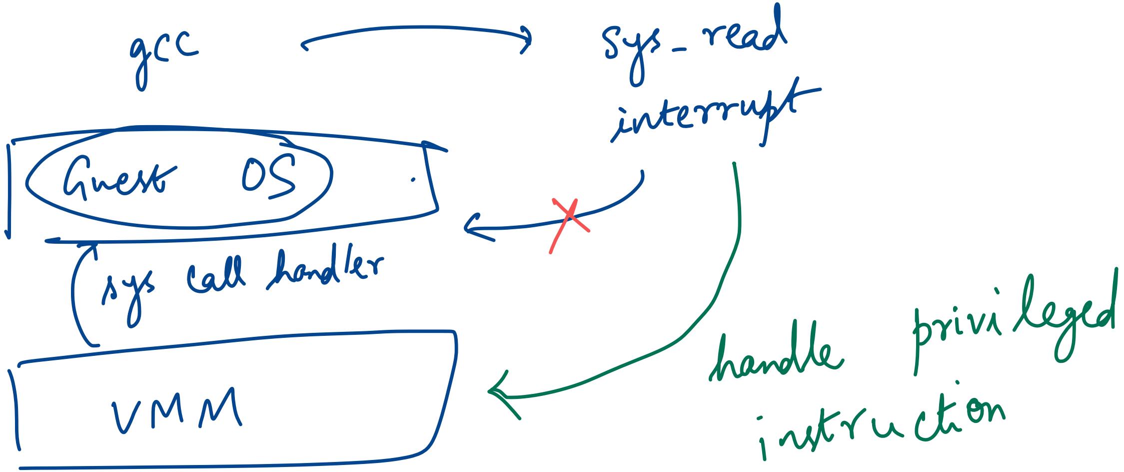
VIRTUALIZING THE CPU

Limited Direct Execution

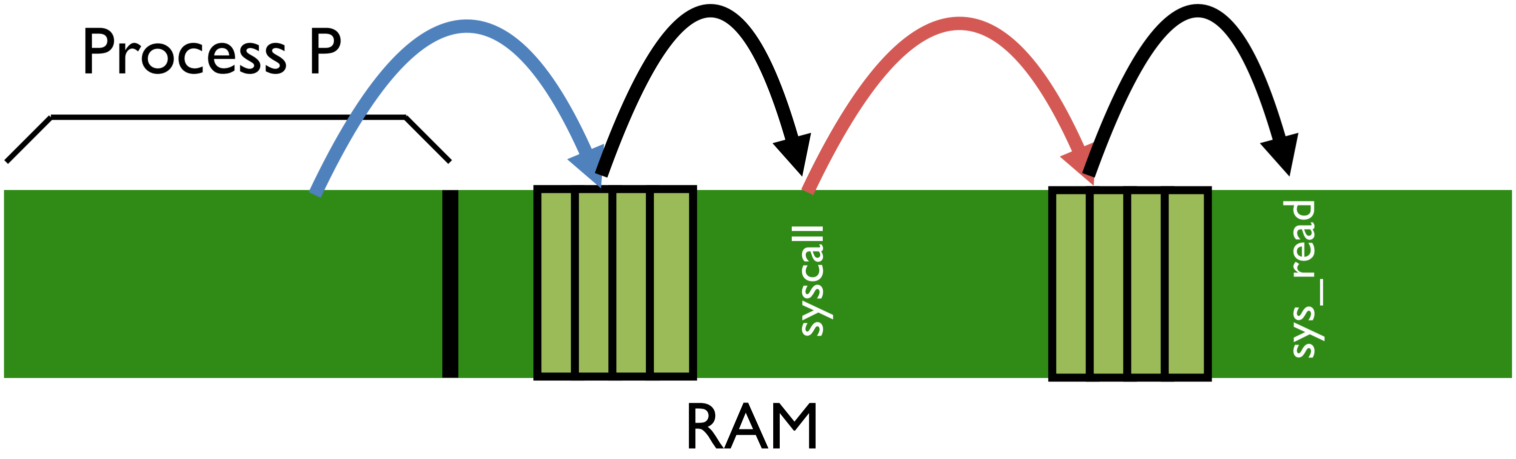
How to handle privileged instructions (e.g., traps for system calls) ?

Trap and Emulate!

- enter the VMM
- emulate how this happens on real hardware



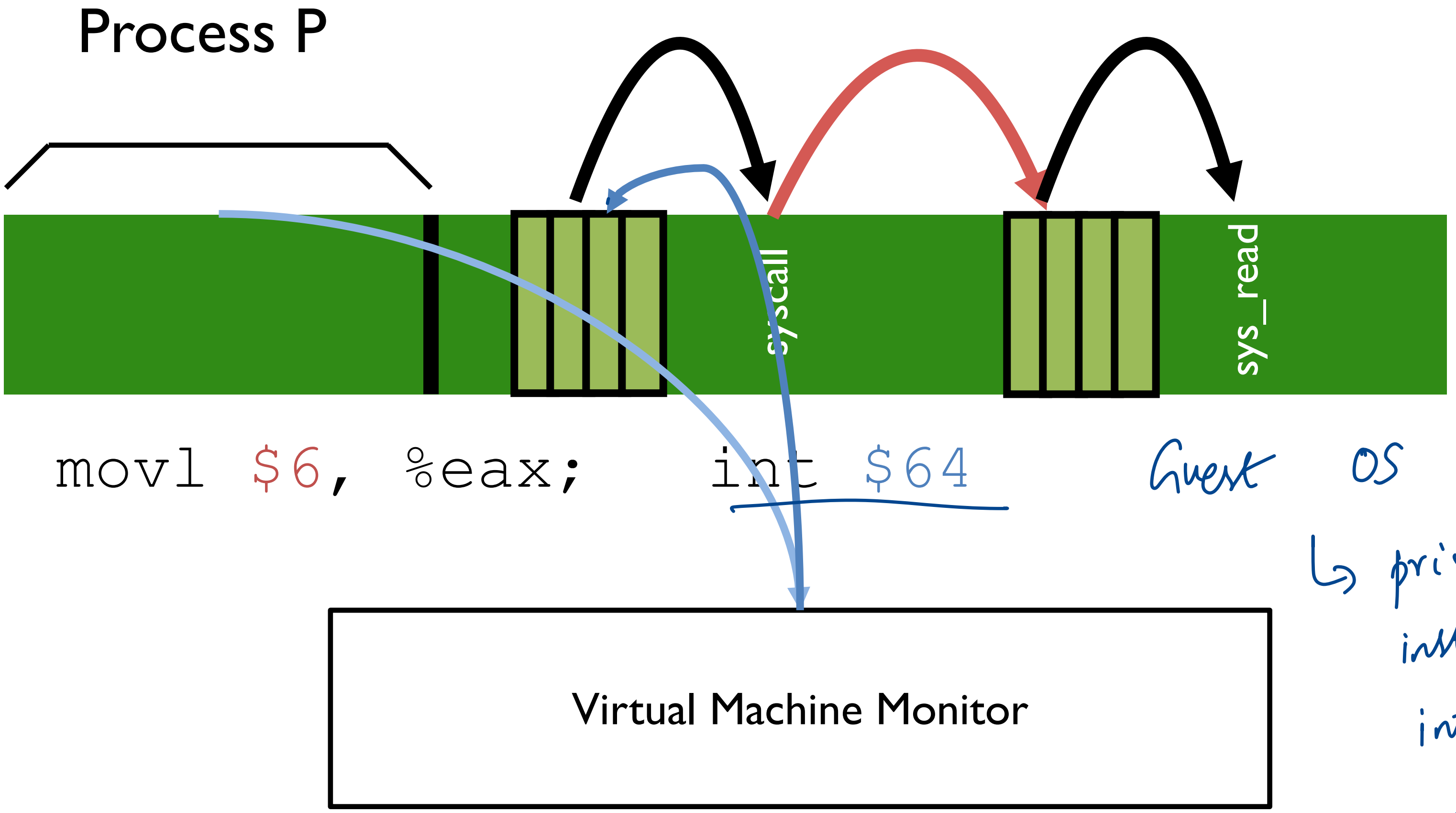
BEFORE: SYSTEM CALL FLOW




```
movl $6, %eax;    int $64
```

Transfer control to trap handler. Execute appropriate syscall routine

NEW: SYSTEM CALL



Guest OS boot up
↳ privileged inst to install interrupt handler

USER MODE, KERNEL MODE?

→ user
→ supervisor
→ kernel

MIPS architecture:

- Guest OS runs in "supervisor" mode
- No privileged instructions, some extra memory

↳ guest OS data structure

Run Guest OS in user mode

How to protect Guest OS data structures?

↳ from the other processes inside Guest OS

QUIZ 20

Log structured SSD consisting of 3 blocks and 10 pages per block. Each page holds a single character.

The state of each page (i, v, or E), the data stored at each page, and an indicator if a page is currently live (i.e. has a mapping in the FTL).

- read(page#) -- if page is live returns the character at the page, otherwise error
- write(page#,char) -- writes character to logical page #
- erase(page#) -- removes logical page # from the FTL mapping

```
FTL      0: 15    2: 18    3:  8    4:  4
        14: 16
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvvvvvE iiiiiiiiii
Data  c9XhFAp970 CqFuArsJE
Live      +   +       ++ +
```

start

write "t" to page 4

new

```
FTL      0: 15    2: 18    3:  8    4: 19
        14: 16
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvvvvvv iiiiiiiiii
Data  c9XhFAp970 CqFuArsJEt
Live          +       ++ ++
```

Page 19 has some data now "t"

If a write(0,'q') is now performed by the OS on the SSD state from the last question, what underlying SSD operations must be performed in order to accomplish this write?

```
FTL       0: 15   2: 18   3:  8   4: 19
         14: 16
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvvvvvv iiiiiiiiii
Data  c9XhFAp970 CqFuArsJEt
Live            +         ++ ++
```

Erase on block 2

Store q in page 20

```
FTL       0: 15    2: 18    3:  8    4:  4
         14: 16
Block 0          1          2
Page    0000000000 1111111111 2222222222
        0123456789 0123456789 0123456789
State   vvvvvvvvvv vvvvvvvvvE iiiiiiiiii
Data    c9XhFAp970 CqFuArsJE
Live          +    +       ++ +
```

initial state

Garbage collector

```
FTL       0: 15    2: 18    3: 21    4: 20
         14: 16
Block 0          1          2
Page    0000000000 1111111111 2222222222
        0123456789 0123456789 0123456789
State   EEEEEEEEEE vvvvvvvvvE vvEEEEEEEE
Data               CqFuArsJE  F7
Live                       ++ + ++
```

erased

live data

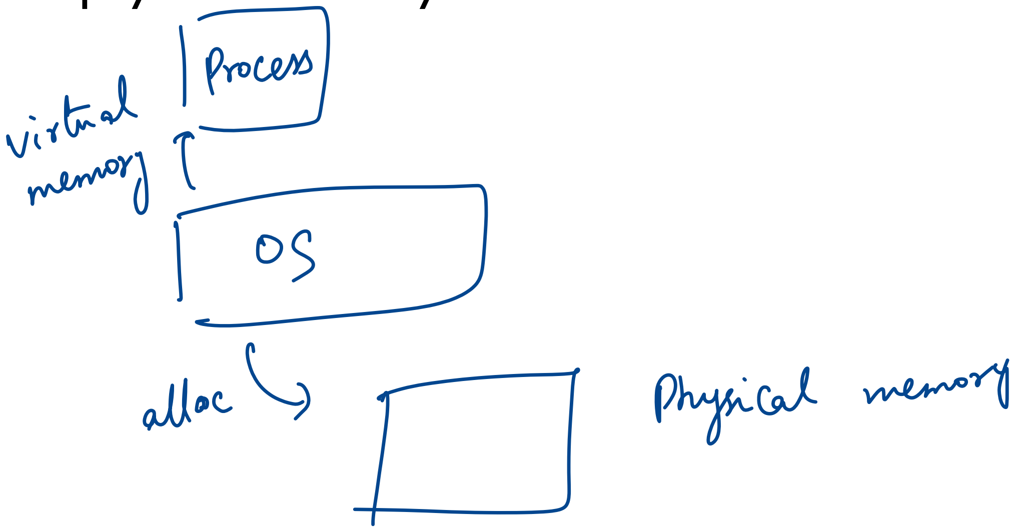
VIRTUALIZING MEMORY

Challenge: Who manages physical memory allocation?

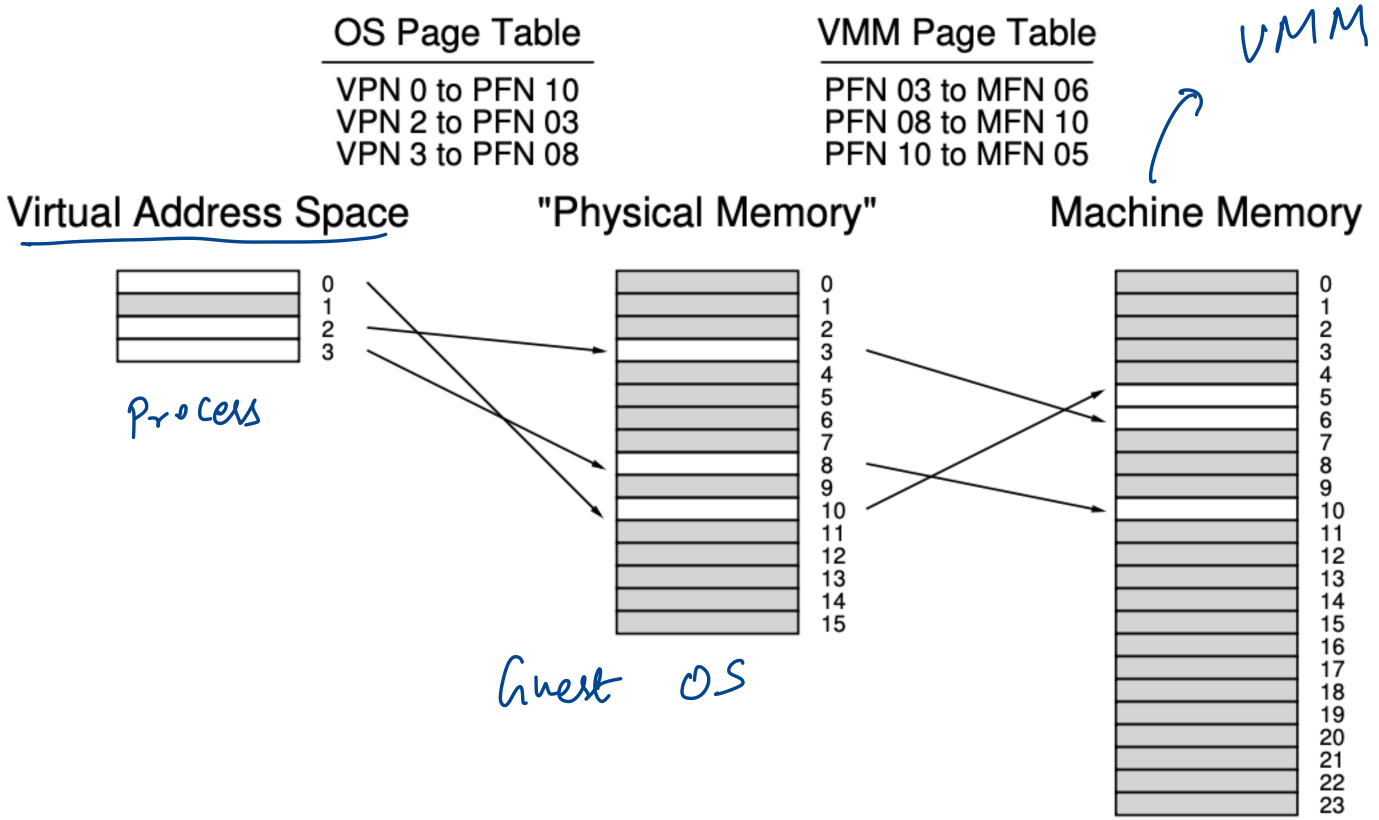
How do we share physical memory across Guest OSes?

Page tables

Virtual → Physical



Extra level of indirection!



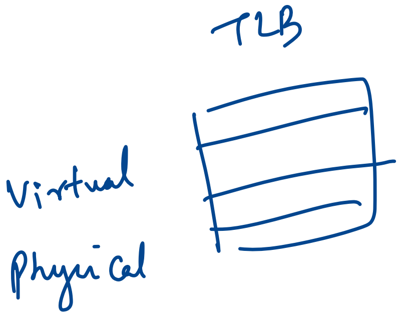
BEFORE: SOFTWARE TLB HANDLER

TLB miss in hardware

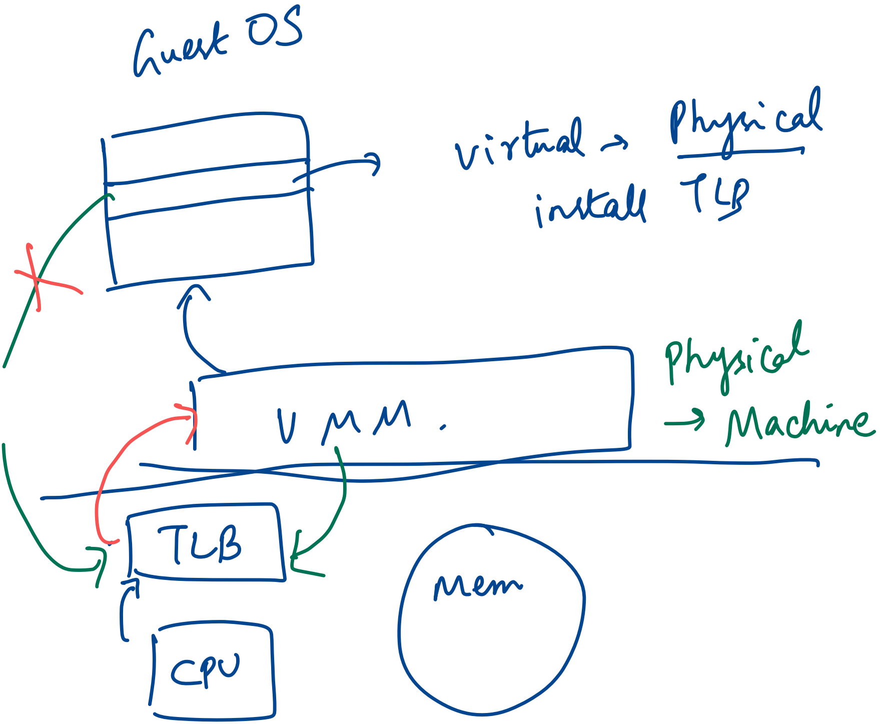
Trap into OS
OS walks pagetable
Get Virtual→Physical
Update TLB using privileged instruction

flush the tlb / update the TLB

Application



NEW: SOFTWARE TLB HANDLER



TLB miss

2 traps

Trap into VMM
Call OS Handler

OS walks pagetable
Get Virtual→Physical
Update TLB using privileged instruction

Trap handler
Physical → Machine
Update TLB

TLB MISS OVERHEADS

Extra trap into VMM for Physical → Machine mapping

Avoid using Software "TLB" in VMM to cache Virtual → Physical

Part of Page tables from Guest OS

Hardware managed TLBs

- VMM maintains Shadow page table per of Virtual → Machine
- Trap when OS tries to update PTE (e.g., lcr3)

x86

for every PT in guest OS create a shadow PT

Guest OS → update PTE ↓ trap into the VMM

SO, CAN WE VIRTUALIZE X86?

expose hardware / processor state

Table 2.2: List of sensitive, unprivileged x86 instructions

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>

PARA VIRTUALIZATION, X86 EXTENSIONS

So far: No change to the guest OS. No changes to the hardware.

Downside: Overheads can be quite high?

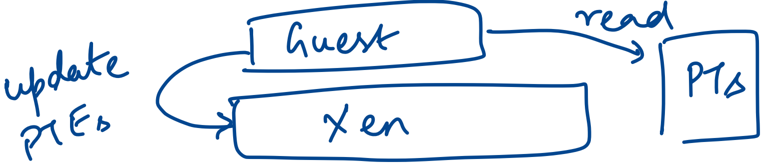
Para virtualization

Can we make (small?) modifications to the guest OS for efficiency?

Hardware

Instruction set extensions (Intel, AMD)

→ x86 virtualization friendly



XEN

early 2000s

Modify guest OS: simply undefine all of the 17 non-virtualizable instructions!

Alternate interrupt architecture

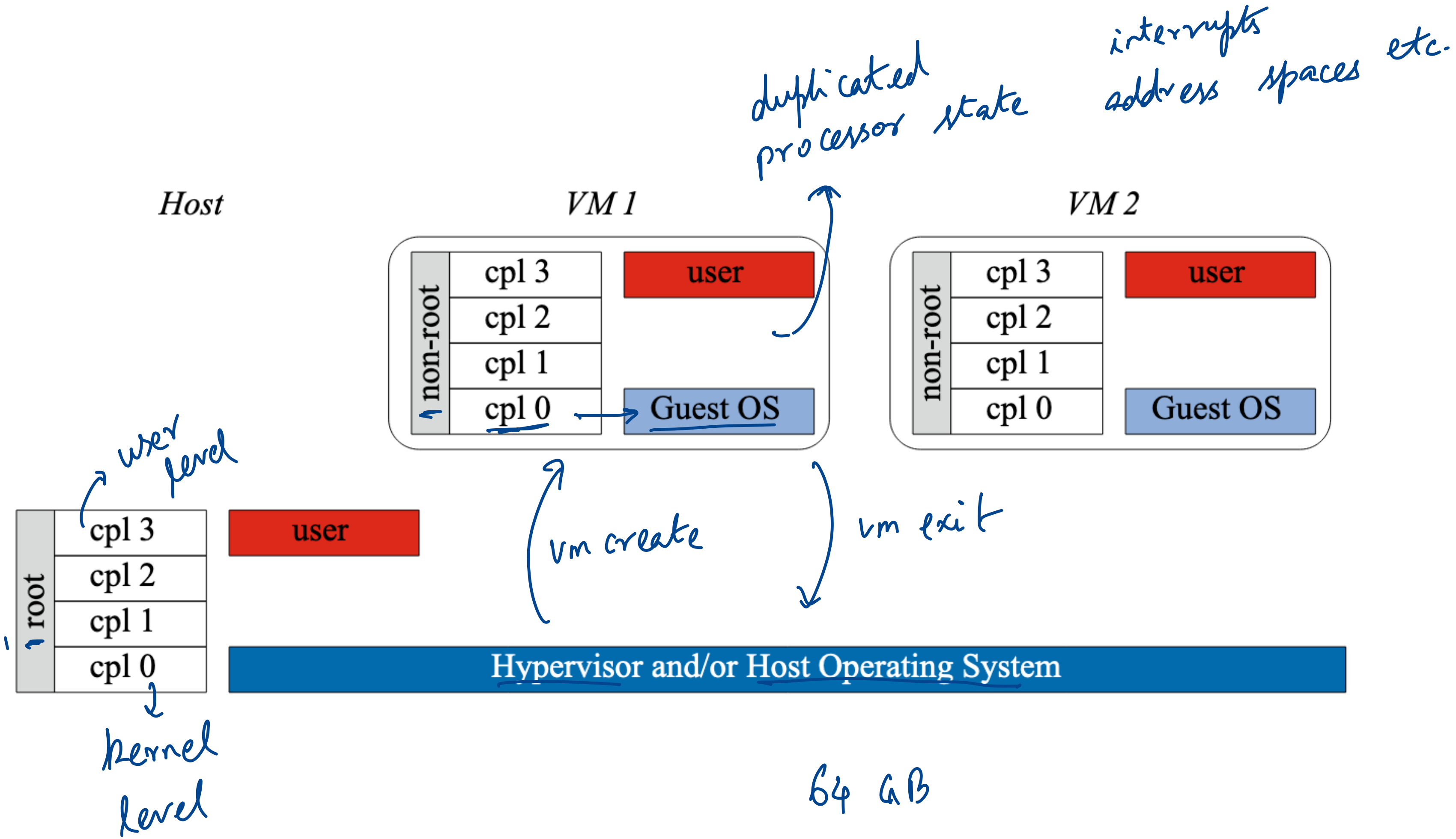
	Memory Management
Segmentation	Cannot install fully privileged segment descriptors and cannot overlap with the top end of the linear address space.
Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontinuous machine (aka host-physical) pages.
	CPU
Protection	Guest OS must run at a lower privilege level than Xen.
Exceptions	Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handler remains the same.
System calls	Guest OS may install a “fast” handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirection through Xen on every call.

INTEL VT-X EXTENSIONS

True Hardware Support meeting Popek / Goldberg Criteria
Do not change the semantics of individual instructions, instead duplicate the entire visible state and introduce a new mode of execution: the root mode.

- Hypervisor is in root mode, Guest OS in non-root mode.
- Special new instructions for detecting mode (only available in root mode, otherwise a trap is caused).
- New mode only used for virtualization
- Each mode has own address space
- Each mode has own interrupt flag

CR3 → each mode has its own PTBR


duplicated processor state
interrupts address spaces etc.
Host
VM 1
VM 2
non-root
cpl 3
cpl 2
cpl 1
cpl 0
user
Guest OS
non-root
cpl 3
cpl 2
cpl 1
cpl 0
user
Guest OS
user level
root
cpl 3
cpl 2
cpl 1
cpl 0
user
vm create
vm exit
Hypervisor and/or Host Operating System
kernel level
64 GB

Next class: Multi-CPU scheduling

Thanksgiving break!