Why should I take this course? ①

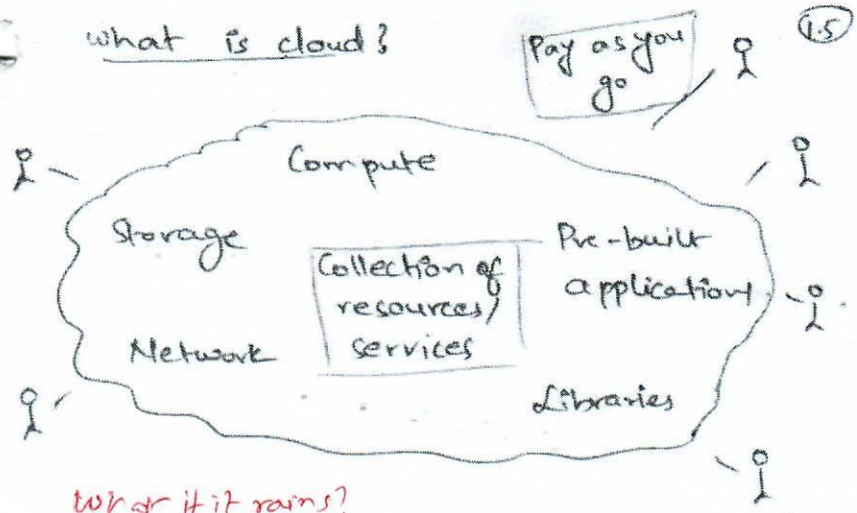
Why should I care about cloud?

virtualization?

rust?

1

what is cloud? (1.5)



what if it rains?

why cloud? ①

Don't want to manage resources

☹ Hardware faults are common

Assume your laptop crashes after 5 years

≈ 1800 days

If you've 200 of them, one is crashing 9 days

Why cloud? Elasticity 2m

- My ~~blog~~ memes app today got very popular
  100 → 100,000 users
-  Laptop crashed. my chance of becoming popular is missed.
- Solution: use 1,000 laptops. · High upfront work

Why cloud? Provides fault tolerance 3m

- Doesn't lose data even when individual disks might fail
- Moves your computation to other machine if CPU overheats and dies.

→ AWS SLA ←

- Downtime is expensive > $1M/hour ①
  - Crashed server
  - Disconnected wifi/electricity
  - Tornado/Earthquake
- Loss of trust
  - Bank lost last 100 transactions

Why cloud? **Proximity** 1m

My memes app is getting popular in Canada.

- **Ping time ≈ 300 ms**
- Every 100 ms delay drops **7% users**
- Rent a VM next door to user ≈ 20ms

→ Region-wise ping times ←

Virtualization ①

What?
- Act of "virtualizing" physical resources

Why?
relavance to cloud

Challenges?

---

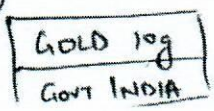
③

Investment 

20yrs sell

- Management, security overhead

Buy virtual gold: Gold bonds



back to VMs ②

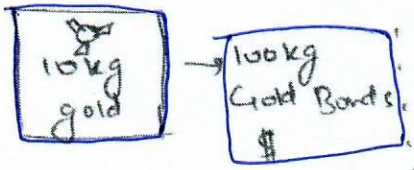
Why? (Buyer)

- No management
- Little upfront cost
- Elasticity
- Trusted

- DDos attac
- Power outage
- Earthquak
- SW update

Why? (Seller) ②

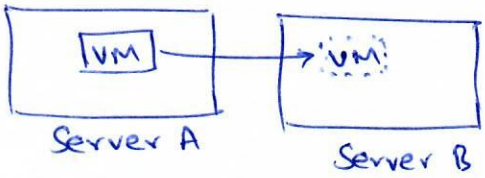
- Overprovision



- Anyways buyi physical resources for self
- Amazon, Google
- Rent unused.
- Incremental mgmt overhe

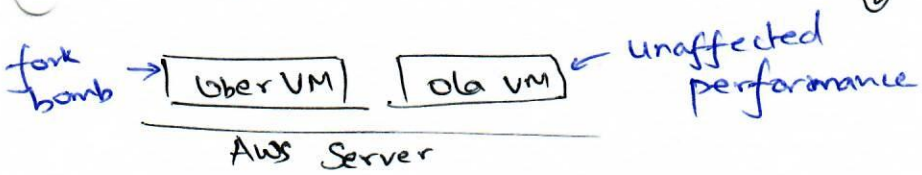
Virtualization supports cloud (1) ②

- Live migration / checkpoint-restore
  - zero downtime



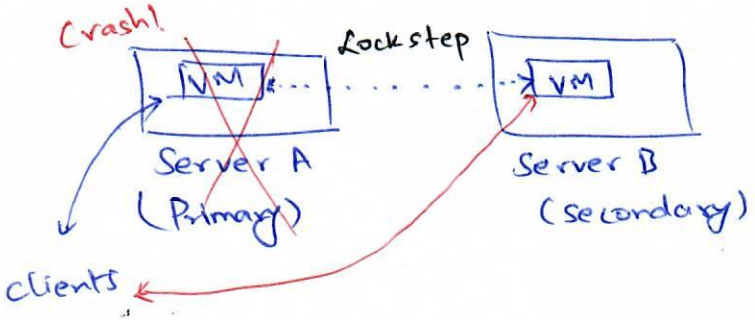
- Load balancing
- Management: restart/upgrade server A (supercomputers)
- Proximity.

* Strong Isolation (even performance)



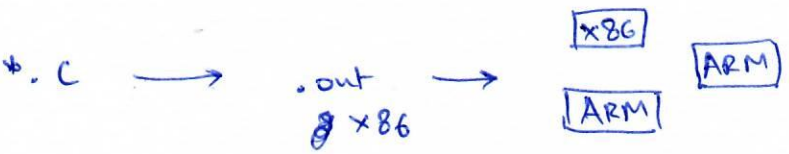
* Auditting / monitoring / Intrusion analysis
  - What all did the VM do?

State machine replication



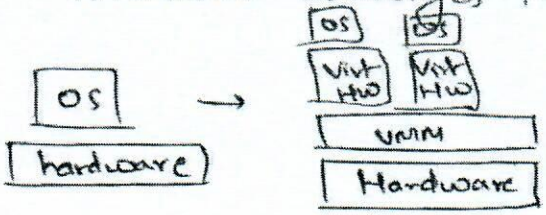
Fault tolerance!

- Binary ~~form~~ one architecture can run on another



- Diversified placement
- Support legacy software

Virtualization challenges: Feasability ②



- Zero/minimal modification to OS, apps (OS thinks it owns HW)
- Performance

What's new? ⑫

Traditional IaaS - Rent VMs

1 How many VM?

Dropping requests ← Too few

Idle ⇒ losing money → Too many

Sophisticated capacity planning, elasticity

New cloud. ①

FaaS: specify functions, triggers

: Auto scaling

: Do not pay when function is not running >AWS Pricing<

New challenges: fast boot, dynamic provisioning, sharing data, etc

Pricing comparison (3)

* FaaS. Mobile application backend

  $2.33

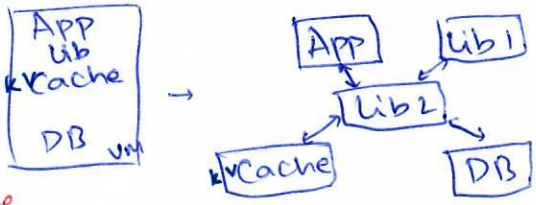
* IaaS. a1.medium → 0.0255/hr * 24 * 30

  = $18.36

* Ignoring data transfer/storage costs

What's new (2) (2)

Monolithic → Microservices



Can we improve performance? Specialize for one workload?

- Scale independently
- Different fault tolerance properties ⇒ crash Lib, can't crash DB

Why take this course? (1.5)

1. You'll work with cloud
   - Startup: using cloud infra (personal projects)
   - AWS / Google / Azure: Managing
   - ML experiments
   - Research: interesting problems
2. Hands on — virtualization: beautiful history
3. Study real systems → firecracker

Highly collaborative project (1)

- Collaboration is hard

=> GIT, CI/CD, PRs, unit tests

- Design is hard :) Interfaces/abstraction
  but fun

- Real end-to-end product

---

why rust? (2)

- Fastest growing language "most-loved" according to stack overflow
- Fast. Systems language C/C++
- Automatic memory management Java/Python/Go (without GC)
- Modern paradigms: Iterators, closures
- Safe: no null pointers, no double mutation, no pointer arithmetic

OS background: Process virtualization

(2)

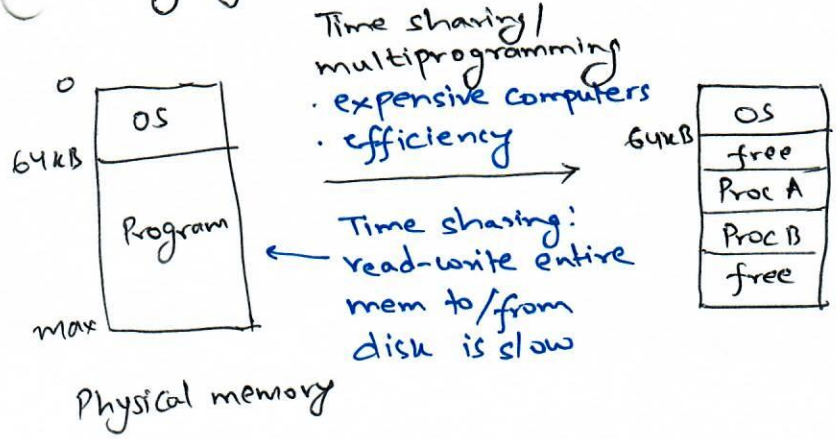
main.c → a.out

a.out	a.out
↓	
W 0x1234 5	W 0x1234 4
5? = R 0x1234	4? = R 0x1234

Memory of Processes are completely isolated

Early systems

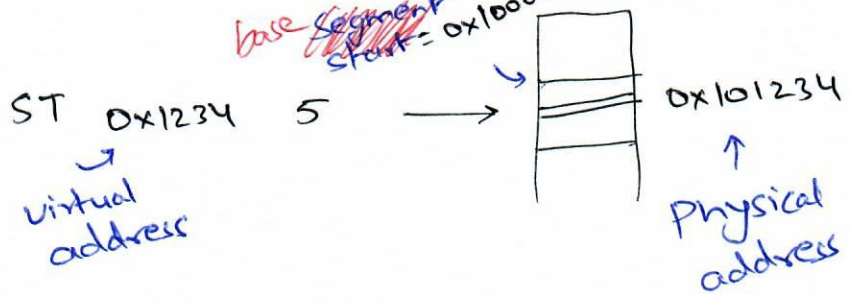
(3)



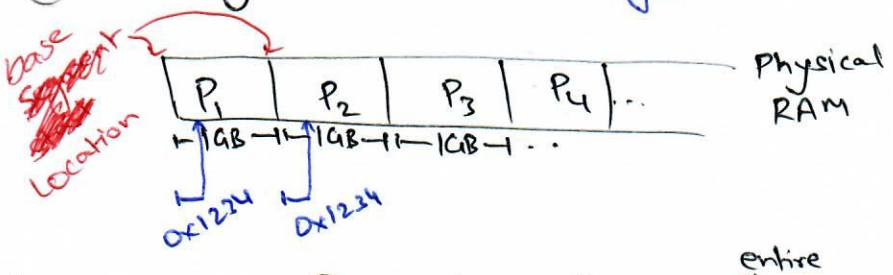
Base and bound

~~Segmentation~~: ~~Performance~~

(2)

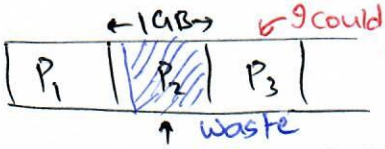


Memory virtualization; ~~segmentation~~ base & bound ③



+ Virtualization: Process think it owns entire memory
+ Performance: Just adder
+ Isolation/security - P2 cannot see P1's memory
- Utilization

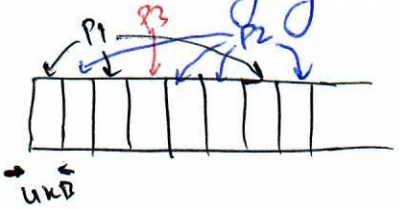
~~Segmentation~~ Problems - Low utilization ③



could use >1GB
Low flexibility
only wanted 1 MB

{}: can only start 4 applications?

Solution: Paging ③



+ Flexibility: Dynamic mapping
  - Allocate a page only when required
  - Send a page to disk if not used (demand paging)

+ Dynamic mapping ⇒ over provisioning ①

👍: can start many more processes

+ over-provisioning ⇒ better resource utilization

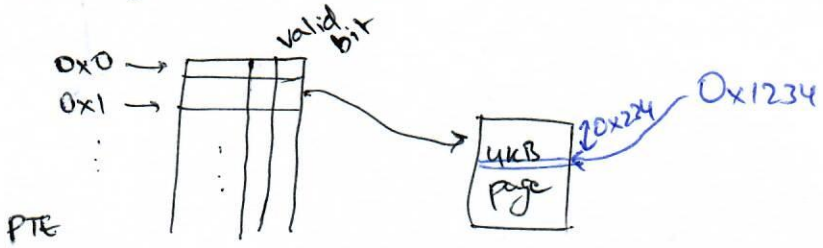
Lao free/unused memory

- flexibility ⇒ complex address translation ①

Virtual address 0x1234 → ?? Physical address

Page number 0x1 → ?? Frame number / Physical frame number

Page table: Page# → Frame# ③



Read-only bit: Don't accidentally write code section

Dirty bit: Was the page written to

Present bit: Allow sending to disk (over-provisioning)

Page table is huge ∴ (5)

32-bit arch ⇒ 2^{32} virtual addresses | 32 GB physical RAM ⇒ 2^{35} physical addresses

Page size → 2^{12} 4KB | # frames → 2^{23}

pages → 2^{20}

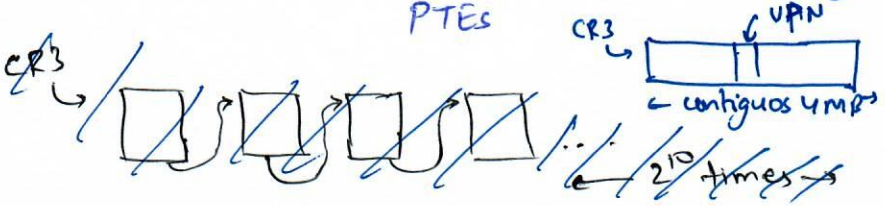
⇒ 2^{20} PTEs. Each PTE has 23 bits + (V, P, RO, ..) ≤ 32 bits ~ 4B

⇒ One 4KB page can have (2^{12}) (3)

$2^{12}/4 = 2^{10}$ PTEs

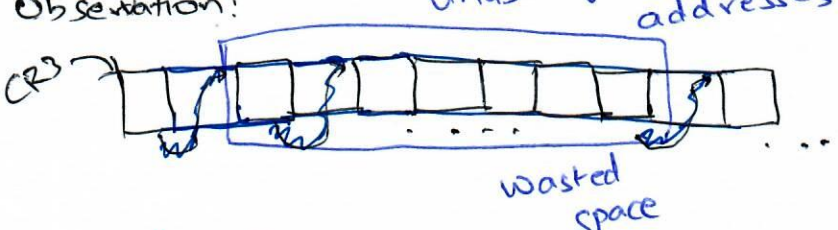
Total PTEs required → 2^{20}

Naive solution: Use 2^{10} pages containing PTEs



⇒ $2^{10} * 4KB = 4MB$ of pages for holding page tables of 1 process! (3)

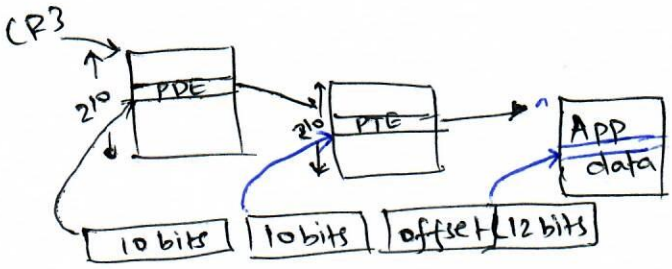
Observation:



Some observation as segmentation

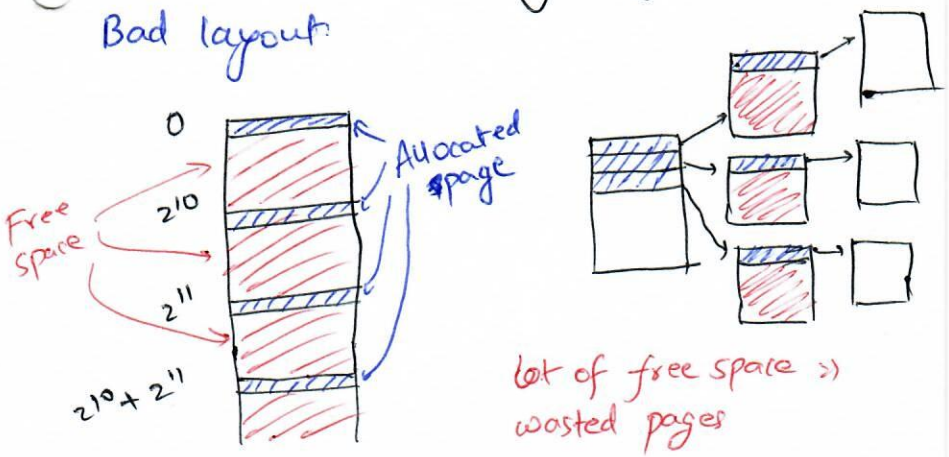
Same solution: Lazily allocate page to page table (3)

Hierarchical page table



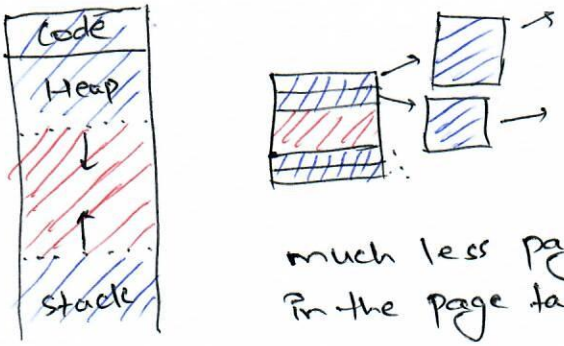
Effect on memory layout (4)

Bad layout



lot of free space => wasted pages

Good layout: segments (3)



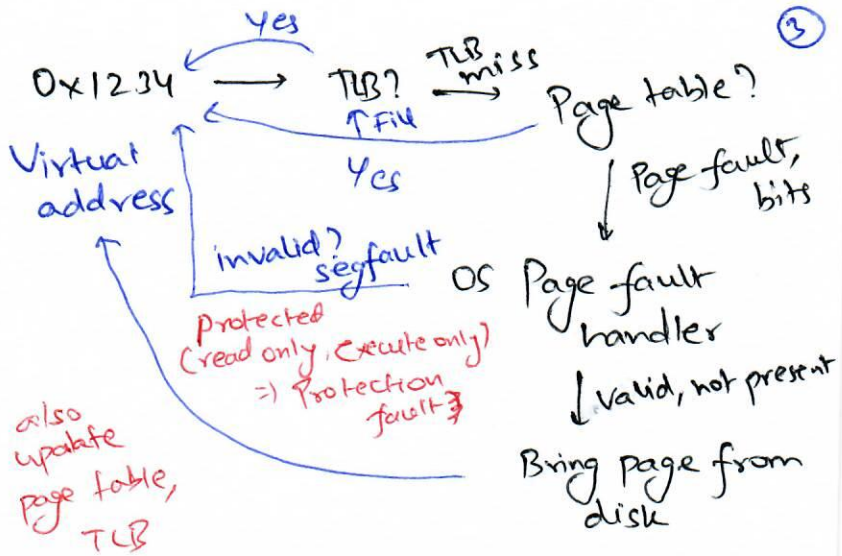
much less pages in the page table

Problem: Every load/store needs 3 memory accesses (2) 2 page table + 1 actual

For 64 bit architectures

3 → 6

Soln: TLB remember page # → frame # mapping



Memory virtualization (2) paging vs static segmentation

* Each process thinks it own entire (virtual) address space => Transparently move pages to disk

~ Security

\+ Overprovisioning => better resource utilization

\- Lot more complexity in hardware/OS

Last lecture: Memory virtualization ②

Transparency: I think I own memory

Base & bound:

addr translation → adder + check bounds

✓ Perf
✓ Hardware complexity
× Flexibility
× Utilization

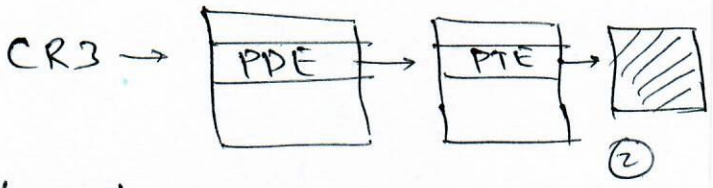
3

Paging ②

- Bits: valid, present, dirty, r/o
- Page table
  - Large ⇒ Hierarchical
- TLB – spatial locality for TLB hits
- Demand paging

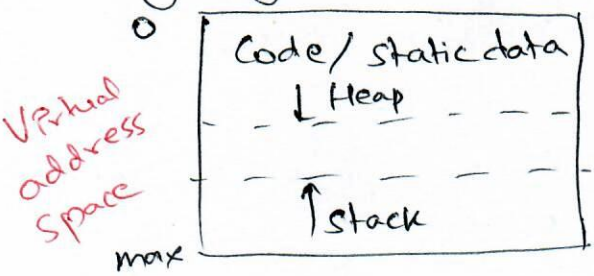
Perf ✓
HW Complexity ×
Flexibility ✓
Resource util ✓

HPT

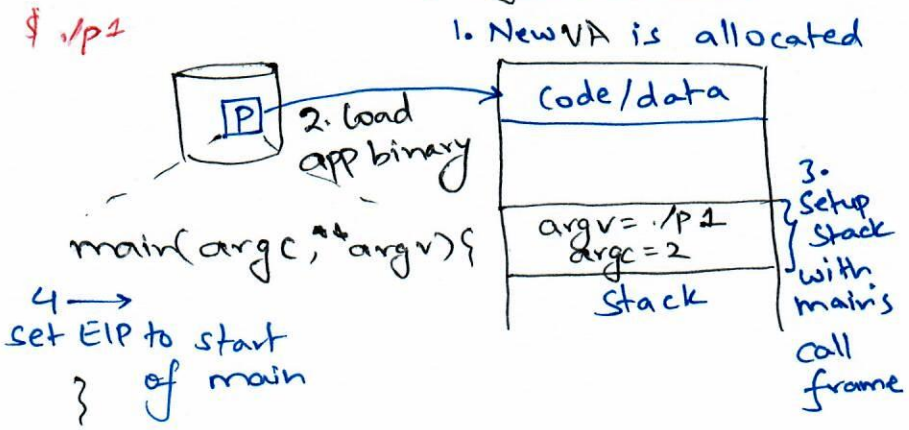


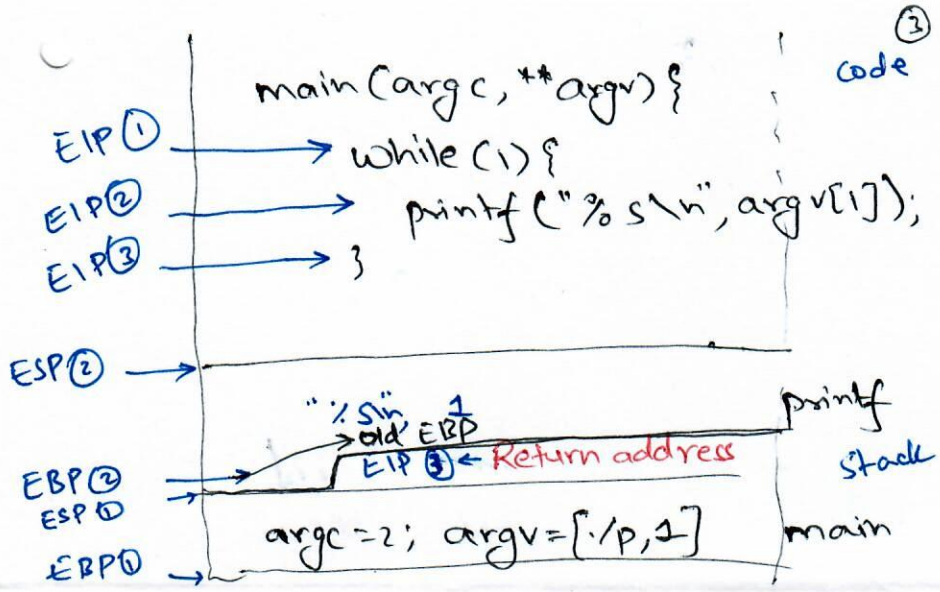
②

Memory layout



How does a program run? ③





CPU register (x86) ②

CR3 → Page table ← Hidden from programs

EIP → Instruction pointer ← Auto Incremented by CPU

ESP → Stack pointer. Top of stack

EBP → Bottom of Current Call frame — Where to allocate call frames

(managed by Compiler)

General: eax, ebx, ecx...

using actual hardware registers ②

CPU: P1 P1 P1 P1 | P2 P2 P2 ... | P3

Early batch computers → Time

wait for program to end

↓ Resource utilization

CPU: P1 P1 wasted cycles P1 P1 P1

↓ I/O ↑

I/O is slow

Operator (manual): ①

- Kill run away process
- Prioritize process
  - manage list
- Only used during office hours
- Only way to intervene is to kill a process

P: main(argc, **argv) { ①

modifies actual hardware registers

```
while(1)
    printf("%s\n", argv[1]);
}
```

$ ./p A &; ./p B &; ./p C &...

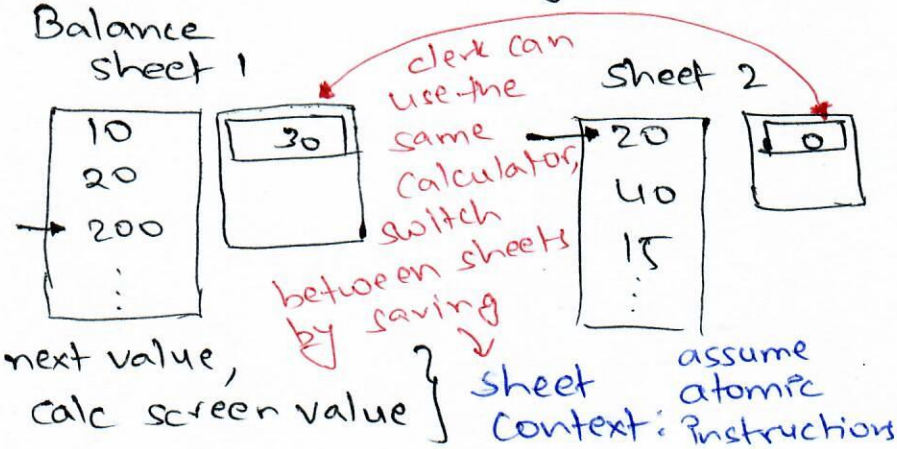
A
B
A
C
B

multiple processes are able to run even though there is single CPU

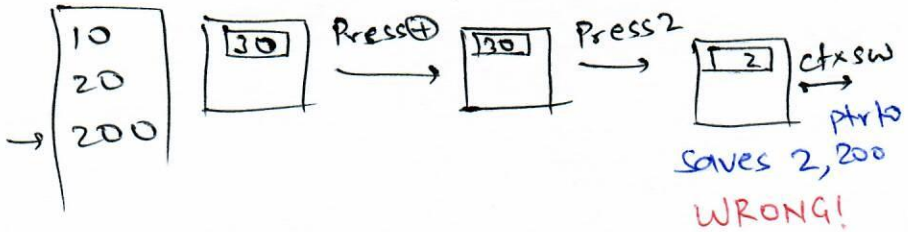
CPU virtualization! (2)

Every process THINKS that it owns CPU registers and directy changes them when it runs! Transparency

Bank clerk analogy (1)



Context sw can't happen in middle of instruction (2)



Sheet context assumes 1 instruction → adding num

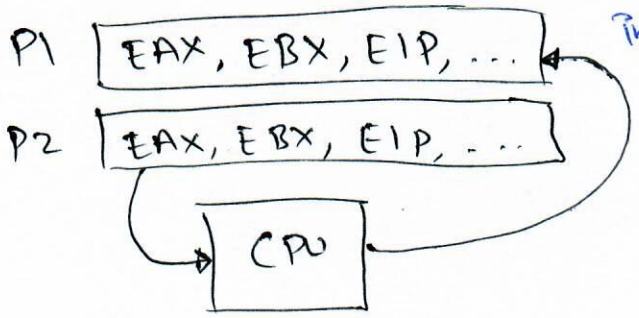
Each instruction → ATOMIC

Context switch

(1)

Process Control block

Can't happen in middle of instruction



Comparison w/ operator

(3)

Flexibility ↑

→ Run multiple processes simultaneously

→ Insert high-priority job without rebooting

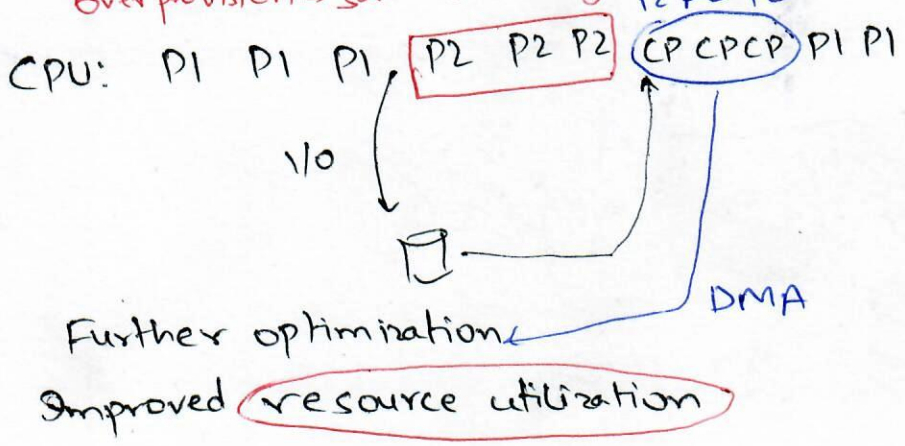
→ Dynamically Kill / spawn processes

→ No need for operator → OS!

Performance benefit

(1)

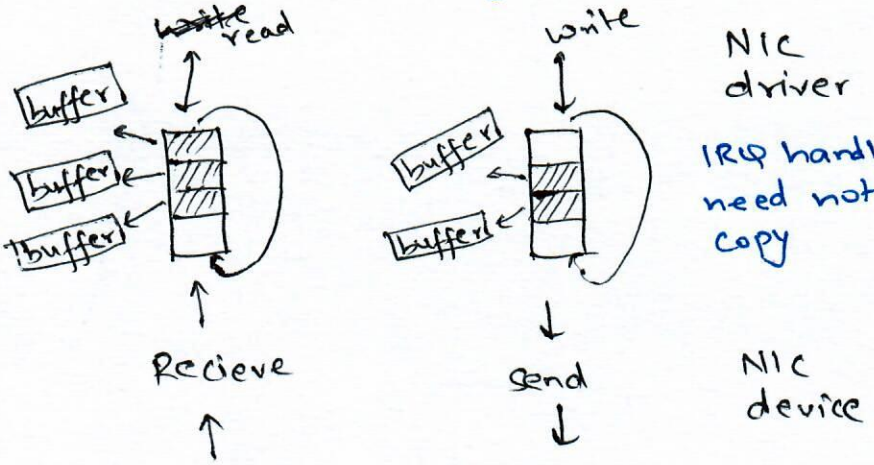
overprovision → saved wasted cycles



Further optimization

Improved resource utilization

Aside: DMA ring buffers ②



Problem: Context switch is light ②

⇒ few registers

Impact on performance is bad.

P1 <> P2 : different address spaces

D-cache, I-cache, TLB misses

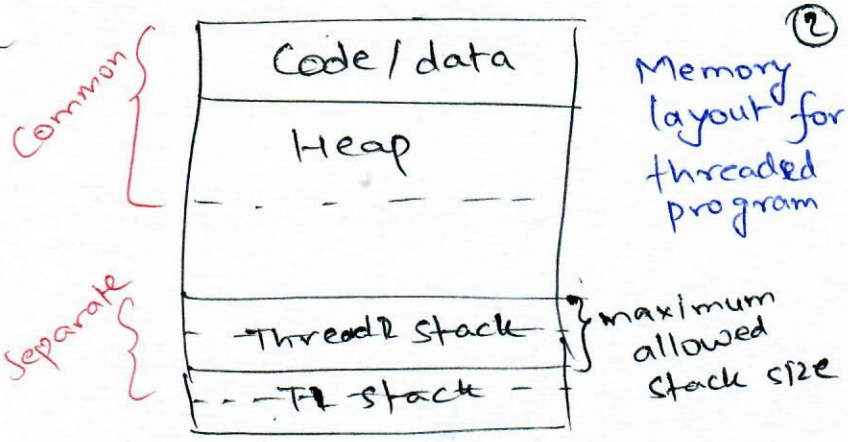
↳ flushed in some architectures

Improve perf. ①

- Only change registers
- Don't change address space

⇒ Threads

- T1 T1 T1 T2 T2 T2 T1 T1 T1
  ↳ fast
- very effective for I/O heavy processes



Memory layout for threaded program ②

Separate registers

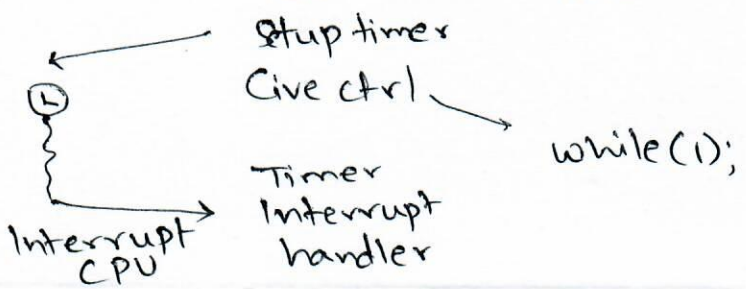
Prob

Give control, keep ability to take back control

```
while(1);   Single CPU
```

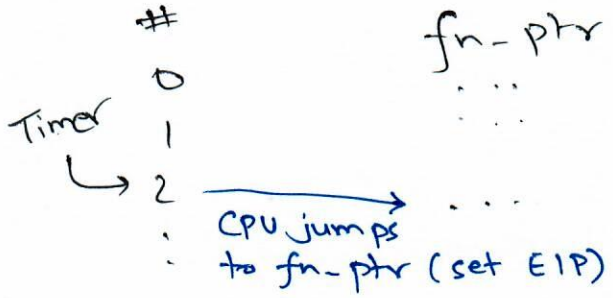
①

Soln



Which handler? ②

- Interrupt descriptor table
  Set up at boot



Timer interrupt
↳ scheduler

 : Modify IDT ②

#	fn-ptr
0	...
1	...
↳2	 → does not call sched

Privileged Instruction ⇒ Traps to OS
⇒ Kill proc

---

x86 rings ①

ring 3 — User

Other privileged inst^{n}:
- Disable interrupts
- Change CR3

ring 0 — Kernel

Interesting programs Need to use privileged instructions ②

- Sending data on network
- Reading disk contents (100x heavier than function calls)

⇒ System calls

Execute INT N Instruction
↳ IDT

CPU registers (x86)

CR3 → Page table ← Hidden from programs

EIP → Instruction pointer ← Autoinc. by CPU

ESP → Stack pointer. Top of stack

EBP → Bottom of call frame

Managed by Compiler

General: EAx, EBX, ECx, ...

---

x86 rings

ring 3 — user app 1, app2

Privileged instructions
- Disable Intrpt
- Change CR3

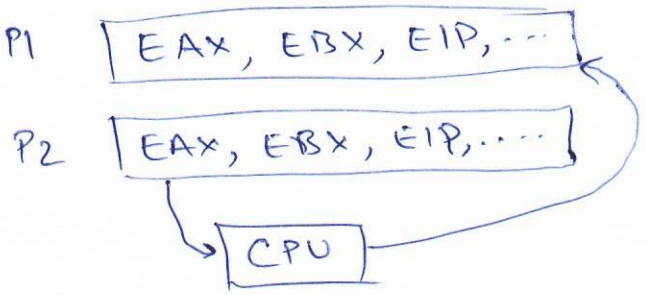
ring 0 — kernel

sys calls

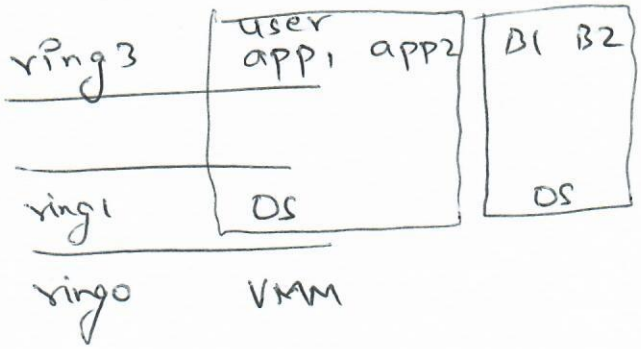
Context switch

atomic instructions

Process control block



Virtualizing OS



Virtualization in OS

Let process think it owns the hardware

- CPU register - Direct execution
- Address space

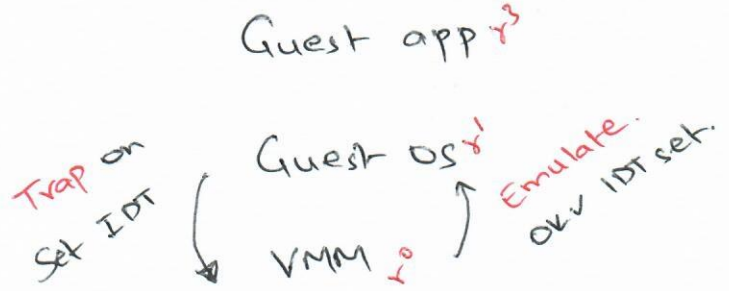
* Isolation *

Virtualization in VMM

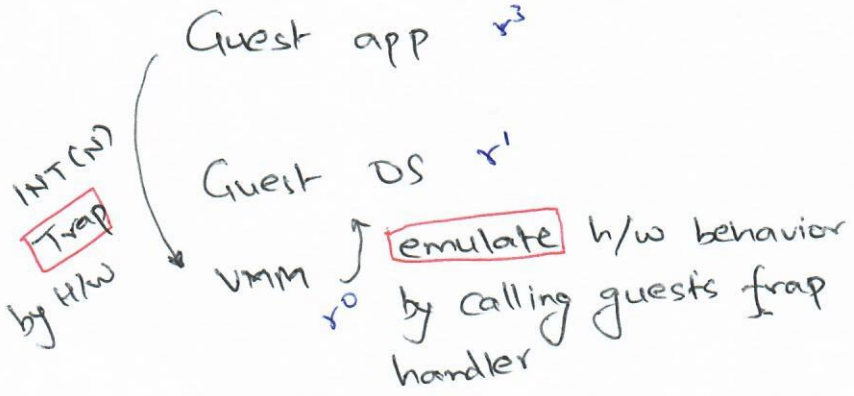
Let guest OS think it owns hardware

- CPU registers
(Guest) EAX, EBX, .. CR3, Interrupt flags
- Physical address space
- Interrupt descriptor table
- I/O devices

Trap and emulate hypervisors ②



Trap and emulate hypervisors ②



Correctness requirements ②

- All sensitive instructions must trap. Popek Goldberg theorem 1974s
- What happens if change CR3
  disable interrupt
  doesn't trap.

EFLAGS register ③

- Interrupt enable flag (IF)

not set by popf instruction
in ring 1 ↳ Does not trap

Why forget Popek Goldberg theorem? ①

Hot areas in 60s - 70s

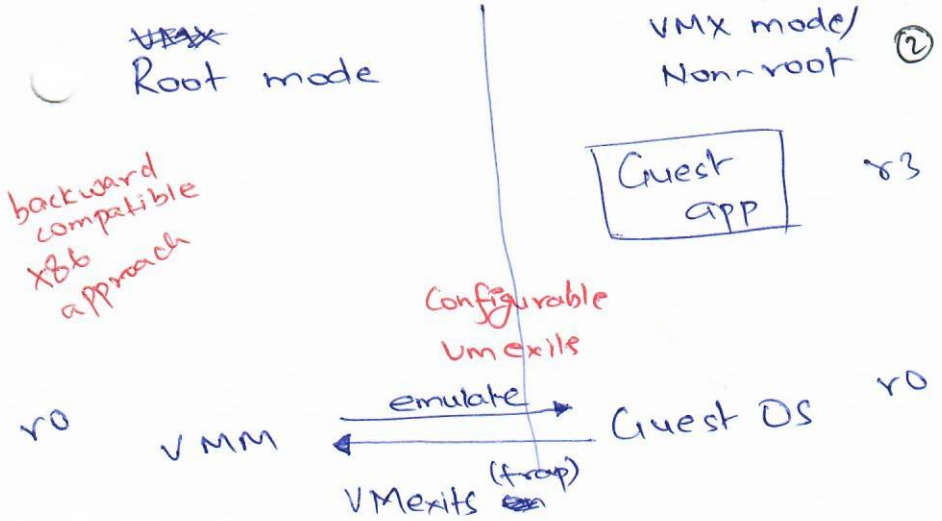
- expensive mainframe
- multiple operating system (no posix)
- let users run own OS ⇒ no rewrite of apps
- develop new OS

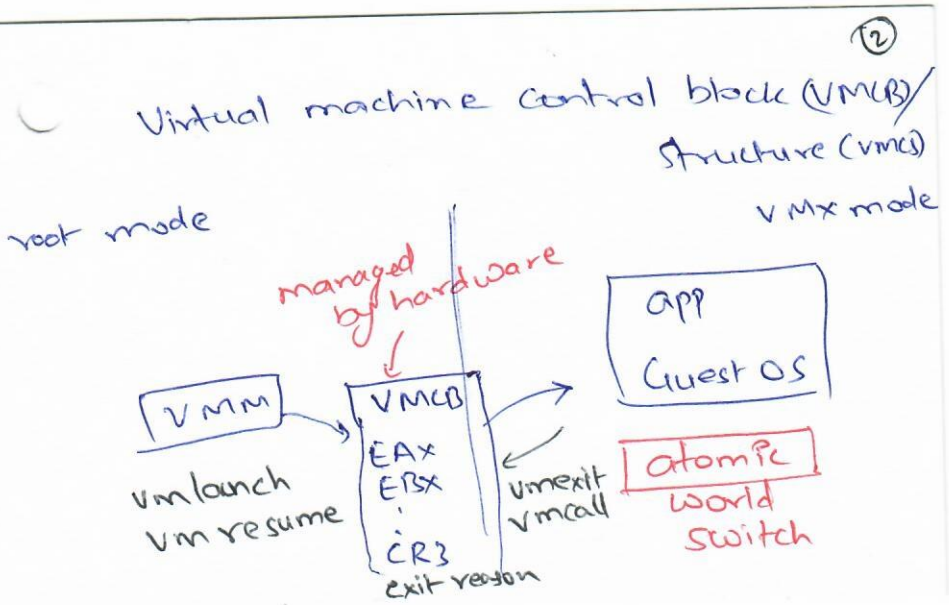
Not so hot in late 70s - late 90s ②

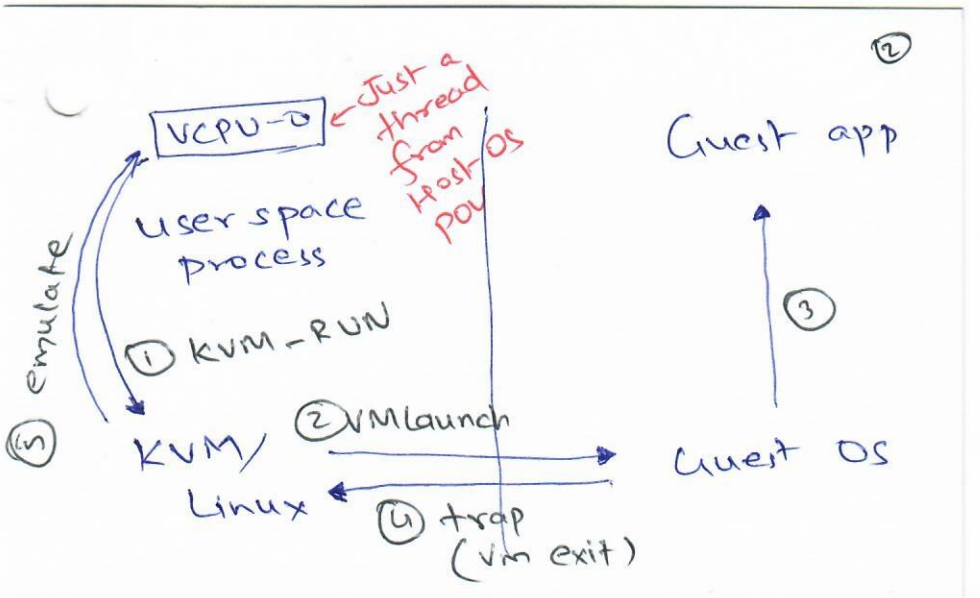
- Personal computing
- My own computer. One OS

Not again from late 90s

- Intrusion detection
- Fault tolerance
- multiple OS


~~VMX~~
Root mode
VMX mode/
Non-root
②
backward compatible x86 approach
Guest app
r3
Configurable VM exits
r0
VMM
emulate
Guest OS
r0
VMexits (trap)


②
Virtual machine control block (VMCB)/ Structure (VMCS)
root mode
VMX mode
managed by hardware
VMM
VMCB
app
Guest OS
EAX
EBX
CR3
VM launch
VM resume
VM exit
VM call
atomic
world switch
exit reason


②
VCPU-0
Just a thread from Host OS pov
Guest app
user space process
emulate
① KVM-RUN
③
KVM/ Linux
② VMLaunch
Guest OS
④ trap (VM exit)

Direct execution

①

When CPU goes to VMX mode, none of host processes are running

(Similar to when process runs, Os does not run)

KVM?

①

Abstracts over hardware details

Intel <> ARM <> AMD

different instructions

Semantics

Kvm API

---

ioctl based : KVM

①

- Create VM
- allocate mem to VM
- read/write virt registers
- Run vcpu
- Inject interrupts

- Last class: Popek Goldberg Theorem (3)

Sensitive Instructions ⊂ Privileged Instruction

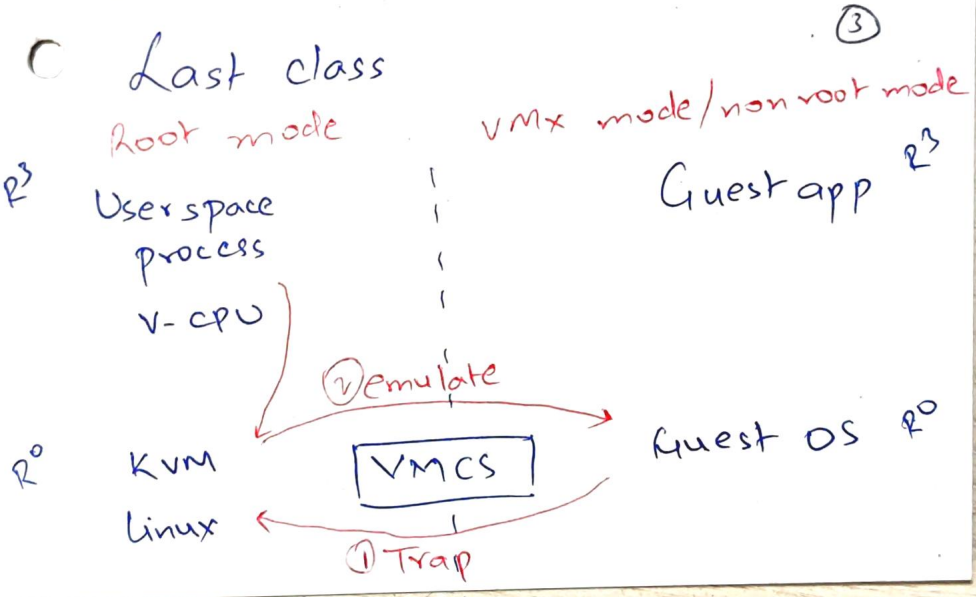
Take control of resources

↳ Trap

eg:- Change CR3
- Change IDT
- Disable Interrupts

Problem with popf

- Last class (3)



- Great performance for CPU benchmarks (2)

- 0-9% slowdown on SPEC benchmark

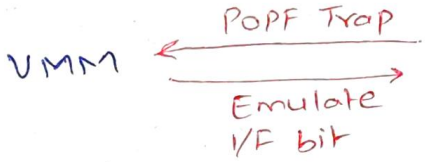
(Figure 2 ASPLOS-06. Comparison of sw hw tech for x86 virt)

- Housekeeping overhead Timer Interrupts

- Exits determine performance for trap-and-emulate hypervisors ③

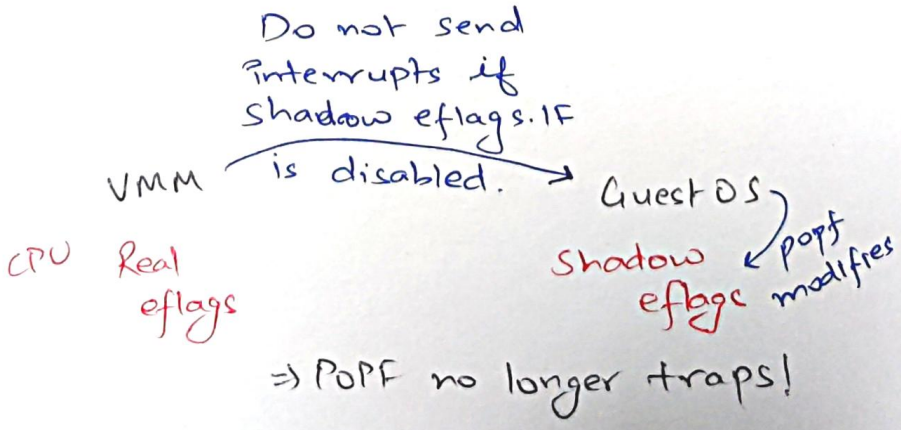
Micro Arch.	Launch date	VMM round trip cycles	Syscall round trip cycles
Nehalem	3Q09	1009	138
Sandy bridge	1Q11	784	134

- Example: guest running popf frequently ①



Solution: Avoid exits as much as possible!

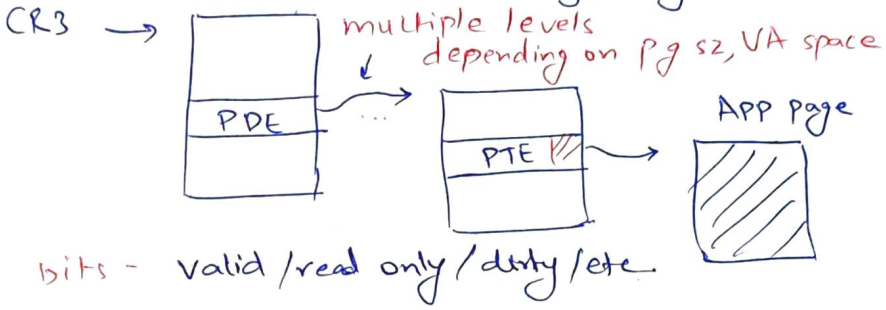
- Dealing with popf: Shadow EFLAGS register in VMCB ②



⇒ POPF no longer traps!

Virtualize memory! Paging ③

Typically 4K



bits - valid / read only / dirty / etc.

Metrics of success ②

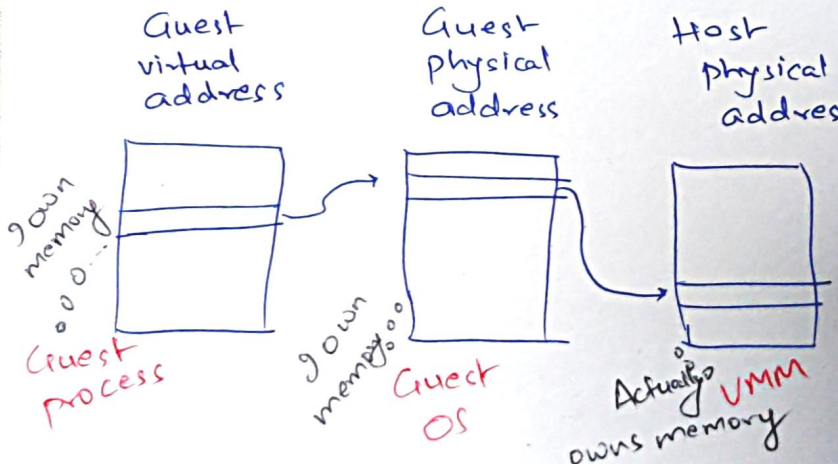
Transparency: Process (→ Guest OS, Guest process) thinks it owns memory

Isolation: One process (→ Guest OS, Guest process) can not read another's memory

Performance: Once mapped, address translation need not involve OS (Guest OS, VMM) Fast load/stores

General idea: ③

Two levels of address translation



Idea 1: Interpretive Execution (Lab) ②

30xx → mem[xx] = AC

Virtual address

array owned by Interpreter

✓ Isolation

✓ Transparency

✗ Performance

eg: IBM System 370
SIE instruction → start Interpretive Exec

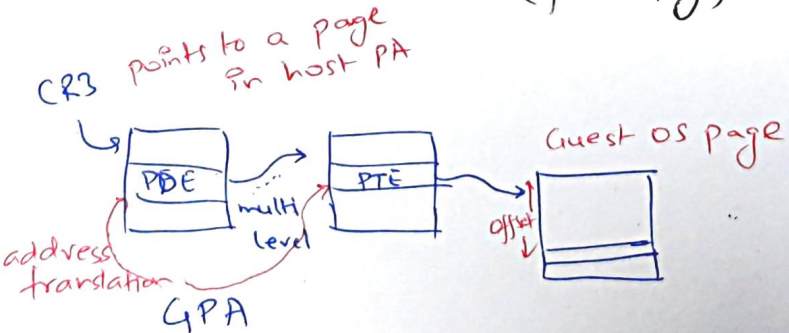
Idea 2: Reuse paging hardware ②

Shadow Page table:

map directly from ⇒ GVA → HPA

⇒ Most memory ops LD/ST run at native speed

Step1: Prepare page table for GPA (for booting) ②



- Now guest OS can boot. (3)
  - Initially, address translation is disabled on real hardware
- CR0, bit 31 can enable-disable paging.

Guest OS tries to disable paging → Trap → VMM changes CR3 to the prev. page table
← emulate

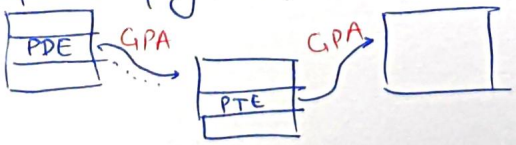
- Guest "thinks" it has disabled paging. (3)

Reading CR0.31 shall { trap or return false { shadow register

But address translation is active.
Addresses treated as GPA

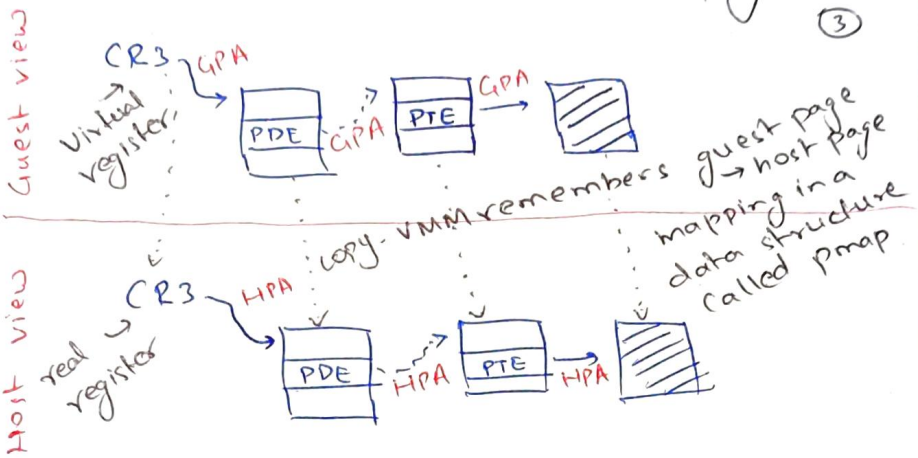
- Guest tries to run a process (2)

① Prepares page table

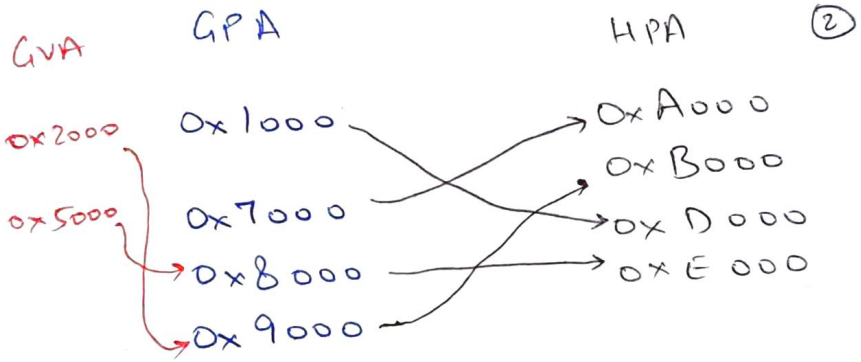


② Changes CR3 to top-level GPA → Trap

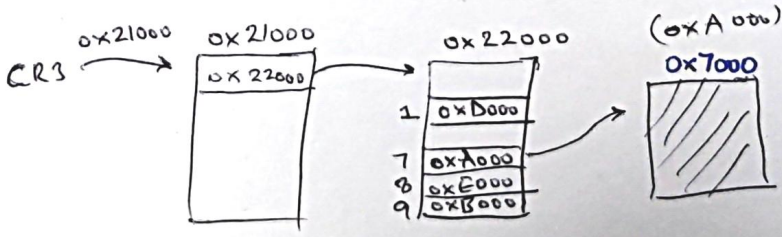
VMM constructs shadow page table (3)



Example: memory mapping setup for VM (2)

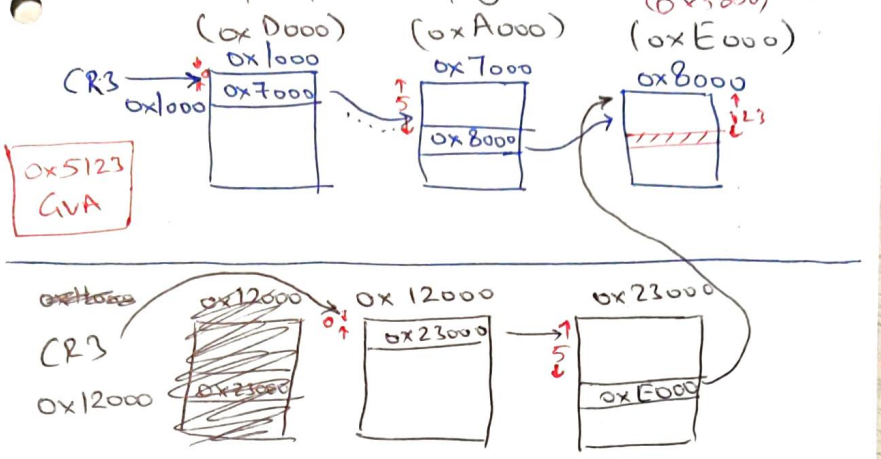


Example: memory setup for VM



When CR0.31 bit is disabled by guest, hypervisor converts GPA → HPA

Guest prepares page table (3)



(3)

- Transparency: If guest reads CR3, it is emulated or kept as Shadow CR3 — 0x1000

Performance: Directly use MMU once page table is setup

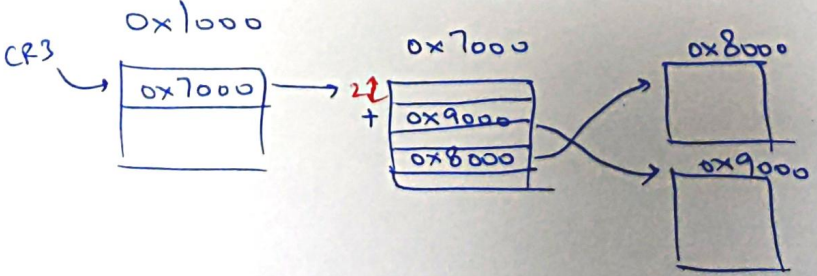
Safety / Isolation: Guest can't directly touch shadow page table. 0x12000, 0x23000 are NOT in guest's address space

(2)

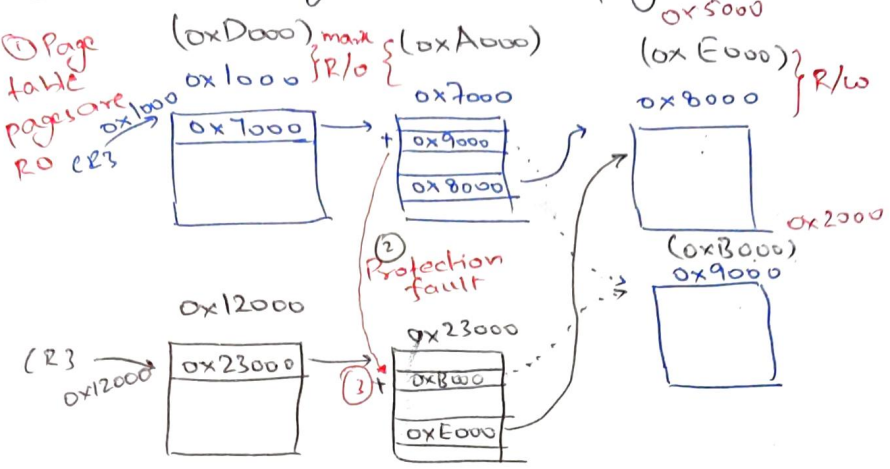
- Guest app mmaped a GVA.

Guest OS did not allocate (lazy)

LD 0x2123 → PF in VMM → emulate → PF in guest OS



Maintaining shadow page table ③



Memory tracing- ②

- Trace writes- mark R/O
- Trace read/writes- mark invalid

Another use- memory mapped I/O

→Memory / performance optimizations ②

Need 1 shadow page table per guest page table.

① VMM deletes a shadow page table

Ok- Rebuild when CR3 is updated back to 0x1000.

Hidden Page fault → Transparent to guest

② Move 0xE000 page to disk. Set "Not Present" in shadow page table. ③

App runs LD 0x5123

↳ Page fault

Bring from disk to 0xF000.

Update ALL shadow page tables pointing to 0xE000

* Maintain backward mappings in "pmap"

③ Move 0xA000 to disk. When ② guest OS tries to update page table.

→ Page fault

Bring to 0xC000

Update GPA → ~~GVA~~ HPA mapping

0x7000 → 0xC000

Summary: ②

① Trap on mv CR3

② Copy PT to shadow PT

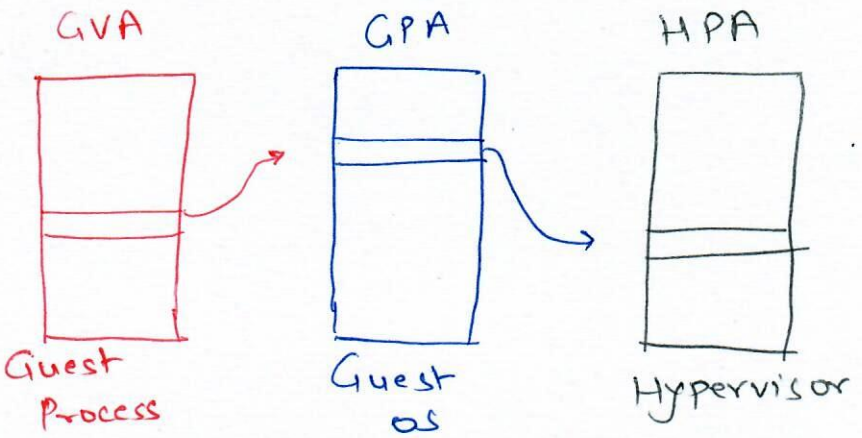
③ Give shadow PT directly to h/w for address translation

④ Memory tracing for maintaining consistency

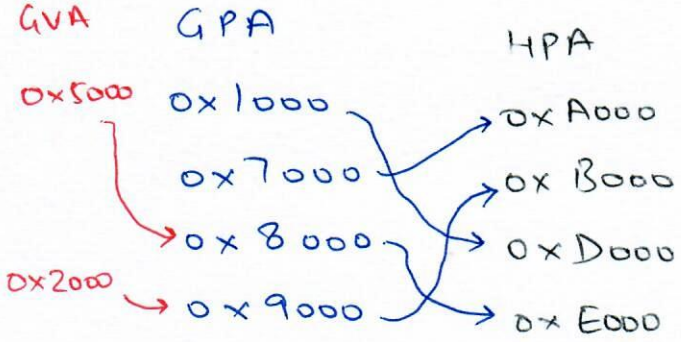
⑤ Pagefault, hidden page faults → backward mapping

- Transparency ✓ Isolation ✓ (2)
- Performance: Native LD/ST in common case ✓
  - Overhead on process creation
  - " " new PTE
  - " " page faults
  - Additional overhead - hidden page fault
- Resource utilization: maintain copy of page table for every guest virtual address space

Memory virtualization in VMM ②



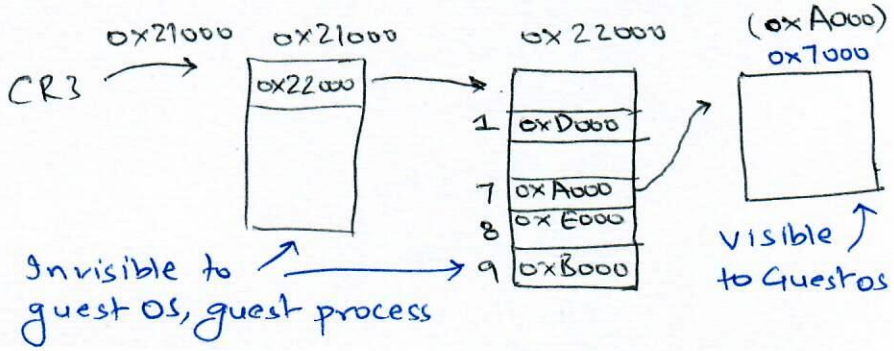
Example: memory mapping for VM ②

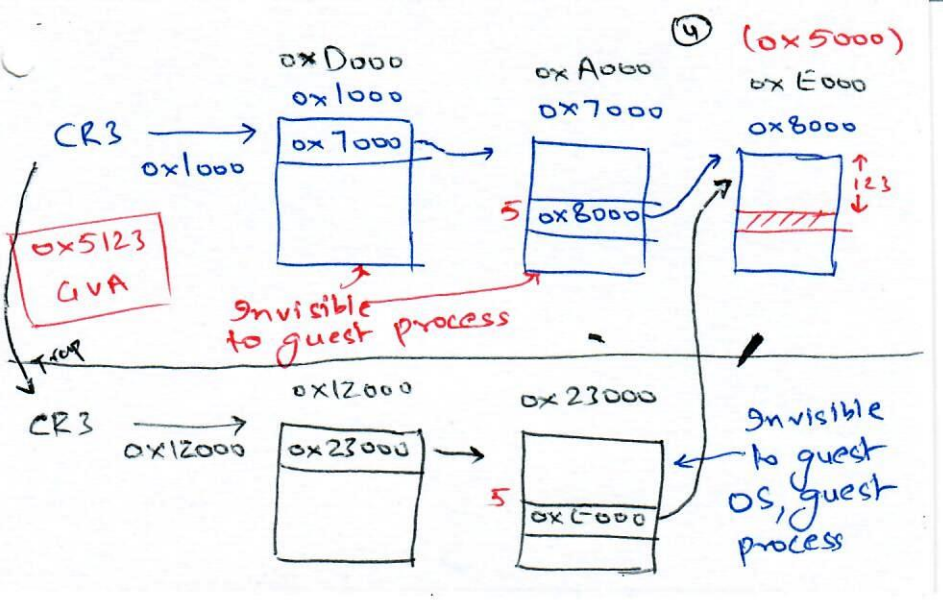


②

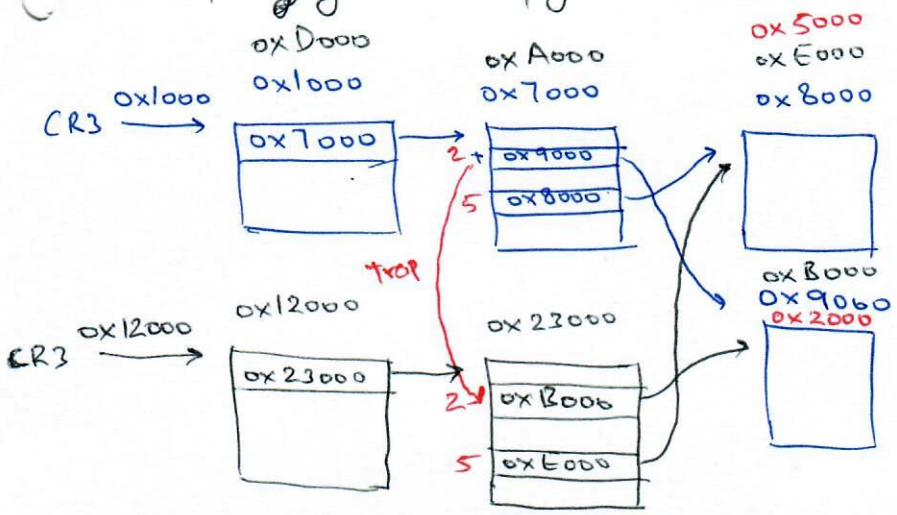
When guest boots, CR0.31 is disabled (virtual memory is disabled)

GPA → HPA





Maintaining shadow page tables ②



Process fork ③

(fork.c)

```
int main() {
    printf("Hello");
    int rc = fork();
    int x = 5;
    if (rc<0){ exit(1); /* fork failed */ }
    if (rc>0){ printf("parent"); x=1; }
    if (rc==0){ printf("child"); x = 2; }
    printf("x: %d", x);
}
```

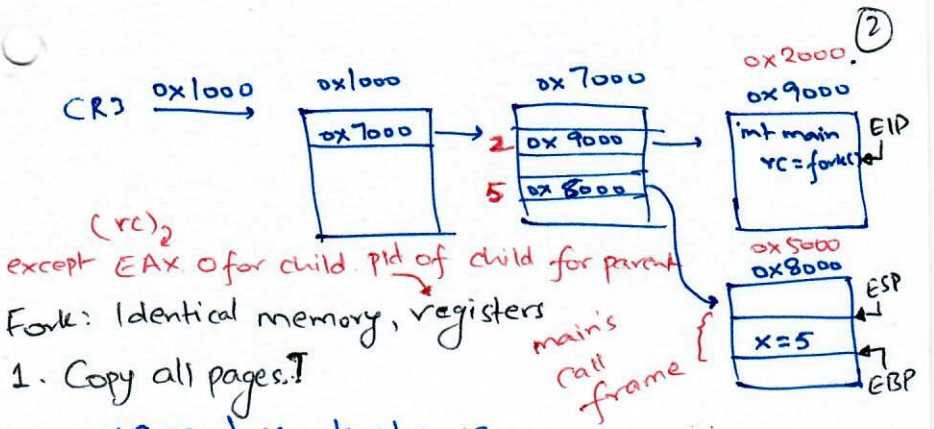
Observations

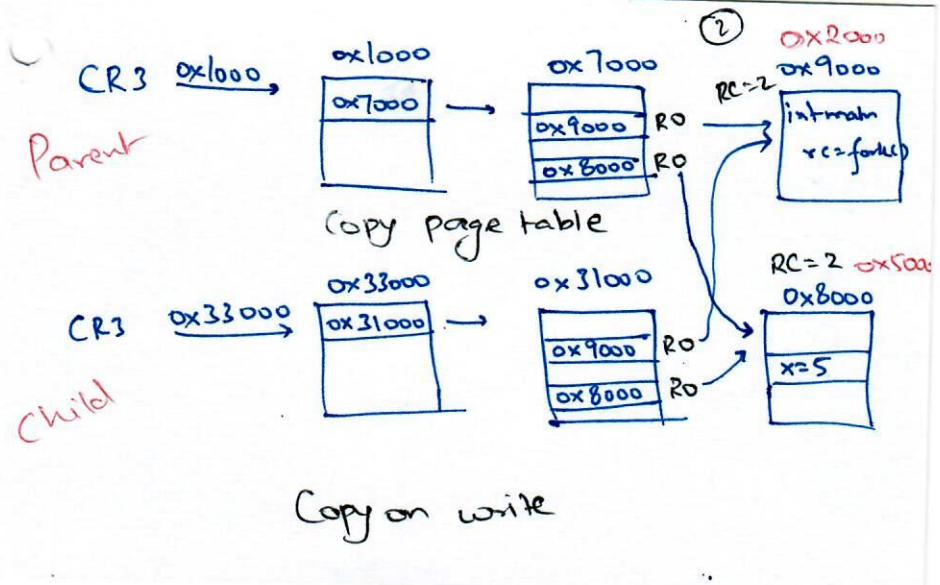
- Child does not print "Hello"
- Parent prints x → 6
  Child prints x → 10

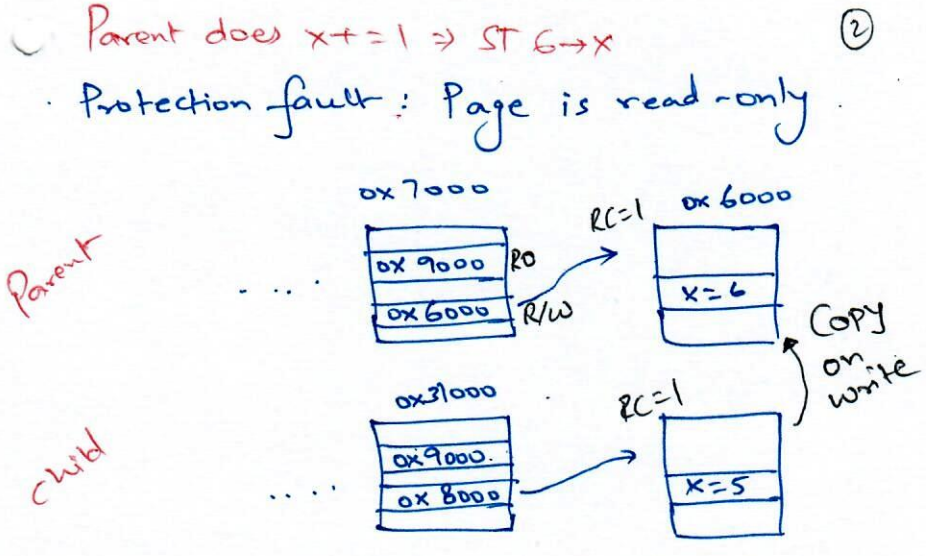
Other things are also passed along-
- open file descriptors
- namespaces, priority, etc.

Why?

* shell.c → gap between fork and exec for output redirection
* In-value storage (e.g. Redis) snapshot
  fork → write DB to disk
  ↳ parent process continues to serve requests


CR3 0x1000
0x1000
0x7000
0x7000
2
0x9000
5
0x8000
0x2000
0x9000
int main
rc = fork()
EIP
0x5000
0x8000
ESP
x=5
EBP
main's call frame
(rc)2
except EAX 0 for child pid of child for parent
Fork: Identical memory, registers
1. Copy all pages.
0x9000 does not change
2. Copy on write


CR3 0x1000
Parent
0x1000
0x7000
0x7000
0x9000
RO
0x8000
RO
RC=2
0x2000
0x9000
int main
rc = fork()
Copy page table
CR3 0x33000
Child
0x33000
0x31000
0x31000
0x9000
RO
0x8000
RO
RC=2
0x8000
x=5
Copy on write


Parent does x+=1 ⇒ ST 6→x
· Protection fault: Page is read-only
Parent
0x7000
0x9000
RO
0x6000
R/W
RC=1
0x6000
x=6
Copy on write
Child
0x31000
0x9000
0x8000
RC=1
x=5

Shadow page table overheads ③

→ 1 fault for every PTE R/w → R/O

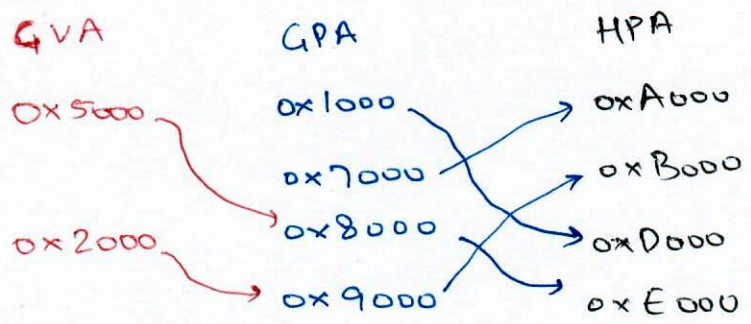
→ Changing CR3 to child
: Copy page table to shadow page table

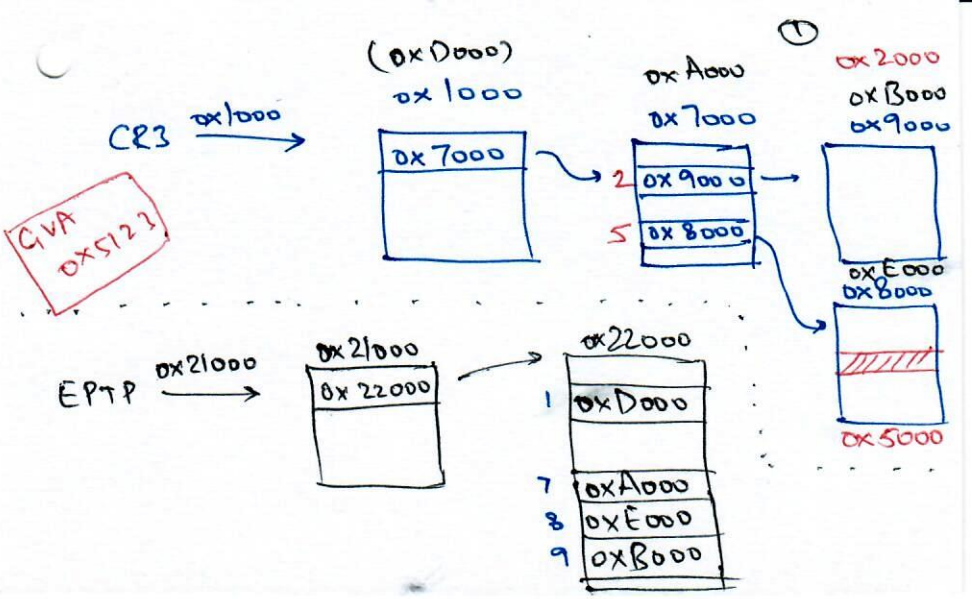
→ At a write, protection fault at VMM → forwarded to guest → guest copies page, updates PTE → again a fault → hypervisor updates shadow page table.

Overheads: Pentium 4 672 ②

- Fork a proc, wait for child to end, fork again → 4.4 x slowdown
- PTE modification
  native → single cycle store
  Shadow PT → 12733 cycles

Extended Page Tables ①





Page walk GVA 0x5123 ③

0x1000 ? → 0x21000 → 0x22000 → 0xD000

0x7000 ? → 0x21000 → 0x22000 → 0xA000

0x8000 ? → 0x21000 → 0x22000 → 0xE000

8 memory references

0x5 → 0xE cached in TLB

m level from GPA → HPA

n level for GVA → GPA

⇒ m+n+m*n references

- Hardware support for 2-D address translations ②
- Changing CR3 need not trap ⇒ fast ctx sw
- Modifying PTEs need not trap ⇒ fast fork

(4)

Table 1:- Instruction and data translations

3.9x - 4.57 x slowdown

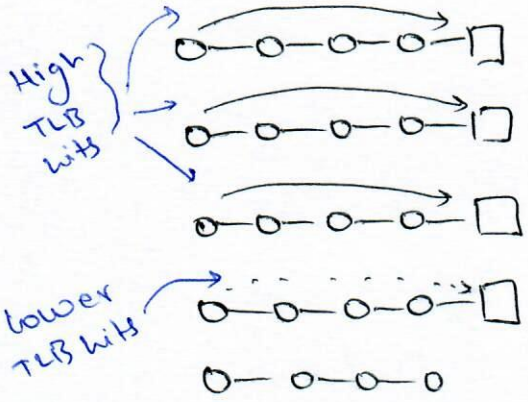
Figure 3:- Percentage of unique page entries for each 2D page walk reference

Most PTEs are common on higher levels

Figure 6:- 86-93% of native performance

Accelerating 2D page walks for virtualized systems ← Paper [ASPLOS'08]

Two main ideas: 1) Nested TLB (2)



Also cache GPA → HPA in NTLB

Figure 8

+7.3% performance improvement end-to-end

---

2) Large pages (2)

→ Increases TLB reach → Also helps in shadow page tables

→ Reduces m and n

⇒ Less number of references in 2D page walk

Reduces PFs

2MB nested pages give 21-43% fewer TLB misses than 4K nested pages

(1)

Memory management

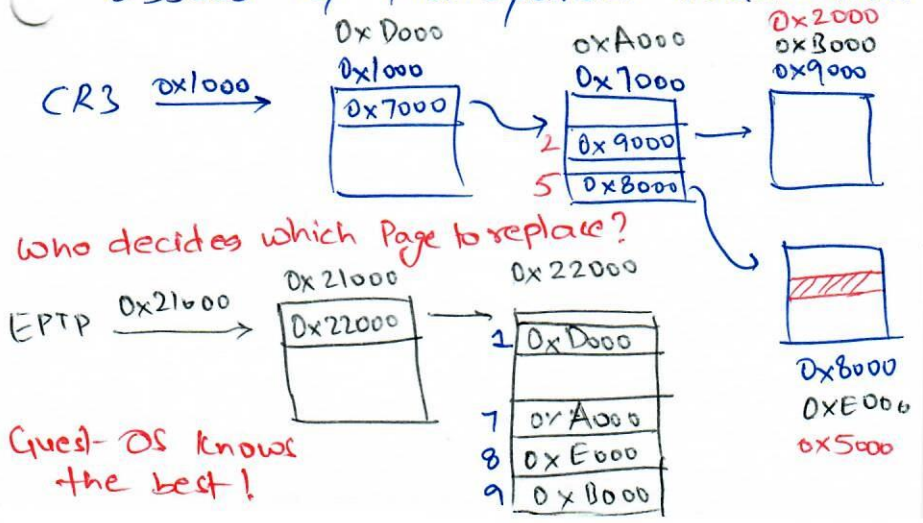
Reclamation
- Ballooning

Sharing memory
- Content based page sharing

---

(2)

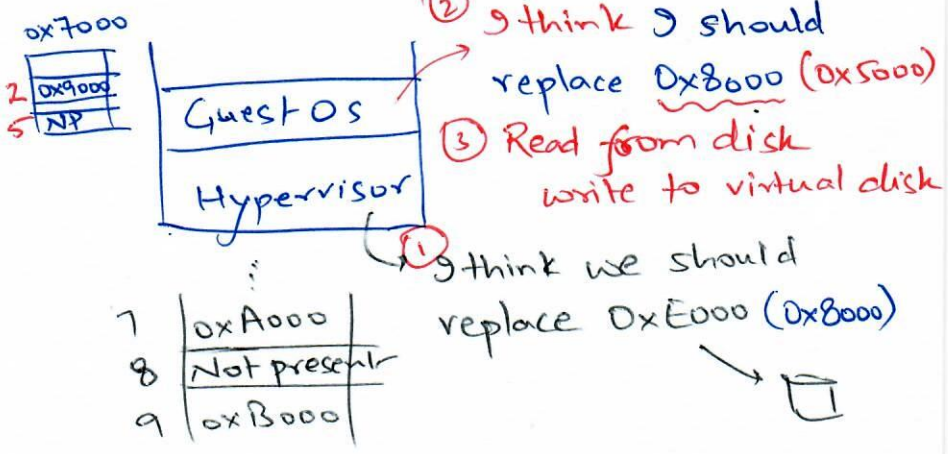
Issues w/ transparent reclamation



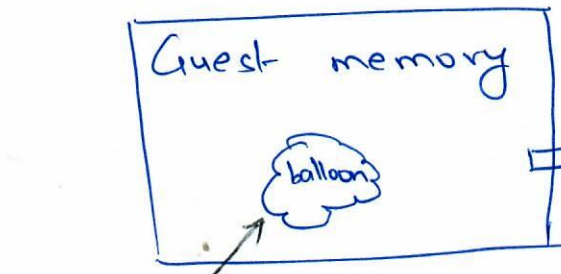
---

(3)

Double paging problem



Memory ballooning (2)



Guest OS moves pages to disk to give to balloon driver

Hypervisor inflates on memory pressure

Figure 2 OSDI'02 — similar perf to VM initialized w/ less memory

Memory ballooning (2)

- Gives control to Guest OS
- Effective in taking memory

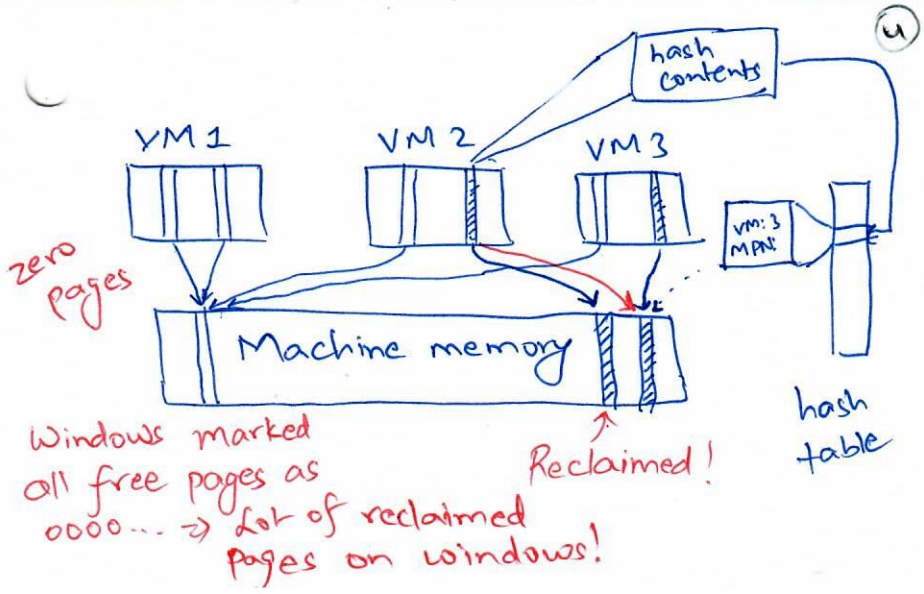
If Guest OS refuses/disables ballooning driver
⇒ hypervisor forcibly takes pages
Guest OS may suffer from double paging

Content-based page sharing (4)

- scan a random page periodically
- Hash ↳ opt: R/o page marked by OS
- Match ⇒ fully match page
- Match ⇒ Mark CoW. Reclaim

⇒ Transparent to Guests!

Figure 5 [OSDI02] 7-32% reclaimed memory



Interesting trivia

Linux KSM (kernel samepage merging) / .. shared memory

- Similar hash-based implementation [Nov '08]
- VM Ware Patent!
- Red-black tree based implementation [Apr '09]