Paging

Introduction

OS → usually takes one of two approaches when solving any space-management problem

chop things up into variable-sized pieces

Segmentation

problems:

→ fragmented
↓
allocation challenging over time

chop-up page into fixed-size pieces

Paging

the Atlas

Instead of splitting up a process's address space into some number of variable-sized logical segments (code, heap, stack)

⊗

divide it into fixed-size units
↓
page

Physical memory becomes an array of fixed-size slots called page-frames

VA
Process
↓
Page

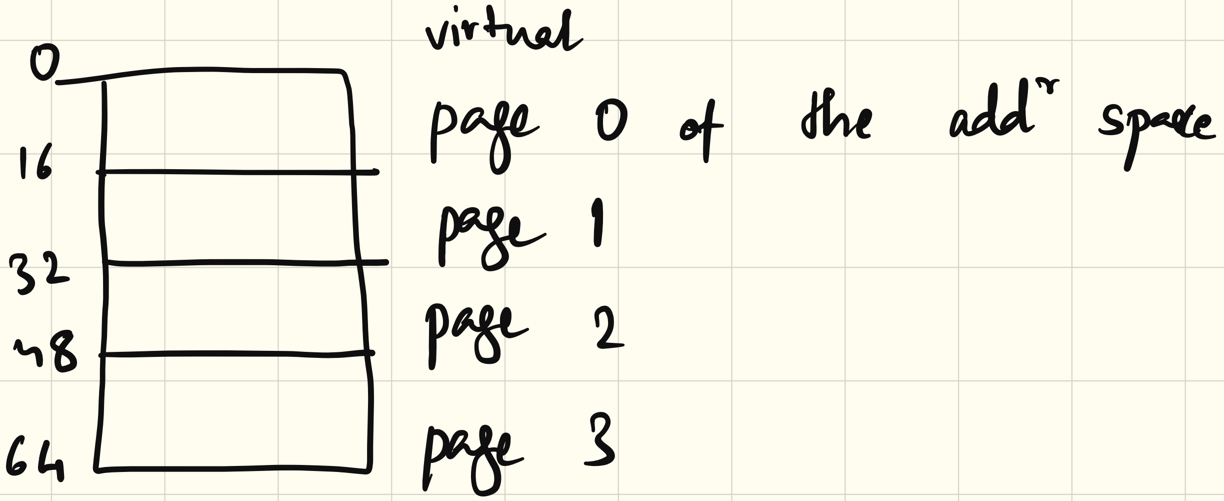
PA
Physical Memory
↓
Page frames.

Challenge

How to virtualize memory with pages:

- avoid segmentation problems
- minimal space and time overheads

Simple example and overview.



Simple 64-byte add^r space

- 4 16-byte pages

Real address spaces → 32 bit address

2^{32} bytes

4 GB

tennis court

→ 64 bit

2^{64} bytes

Europe

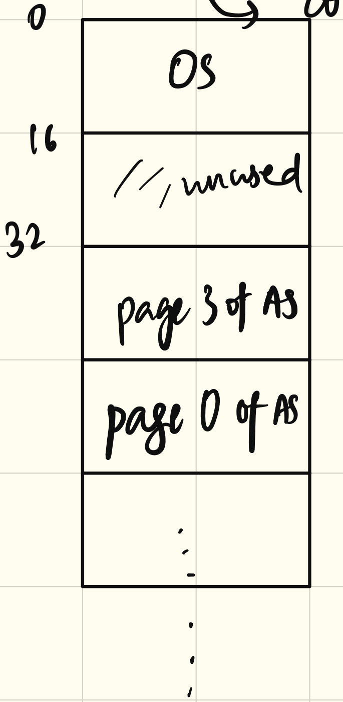
large $\left\{ \begin{array}{l} 2^{4} \times 2^{30} \times 2^{30} \text{ bytes} \\ G \times G \text{ bytes} \end{array} \right.$

Physical memory

→ consists of 8 page frames

↓

128 byte memory



Most important advantage of paging

↓

① flexibility

→ support the abstraction effectively

→ we won't make assumptions about direction of stack and heap.

② Simplicity

→ OS keeps (maybe) a list of free pages to allocate memory.

→ To record where each virtual page of the address Space is placed in memory, the OS usually keeps a per-process DS known as page table page table

→ stores address translations for each of the virtual pages of the address space.

per process DS

exception:
inverted page table

OS will manage many such tables.

movl <virtual add^{r} > , %eax.

→ ignore instruction fetch

to translate VA

split it into
① Virtual Page No.
② offset

e.g. VA Size = 64 bytes

⇒ 6 bits for VA

$$\overbrace{\boxed{b_5}\boxed{b_4}}^{\text{VPN}}\overbrace{\boxed{b_3}\boxed{b_2}\boxed{b_1}\boxed{b_0}}^{\text{offset}}$$

page size = 16 bytes = 4 bits = offset

2 bits = select one of 4 pages

movl 21 , %eax

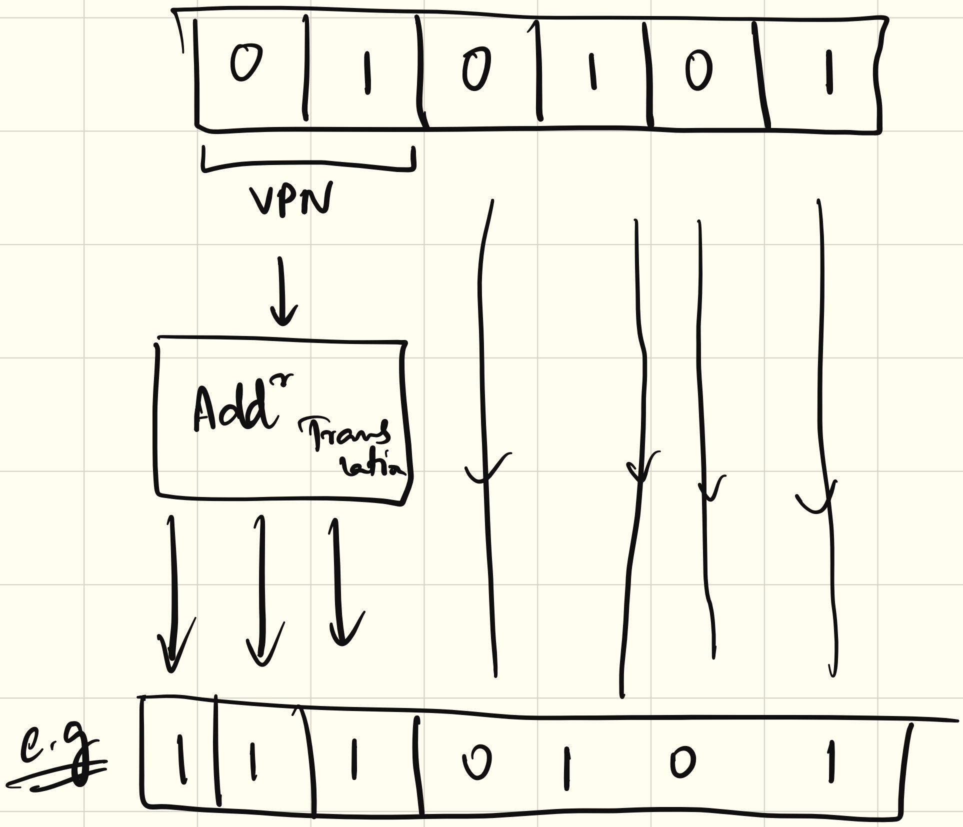
↓

$\underline{01}0101$

↓

page 1 — use page table → find which physical frame v. page no. 1 resides in.

$\boxed{\text{PFN (Physical Frame \#)}}$



Questions

1. Where are the page tables stored
2. What are the typical contents of the page table, and how big are the tables?
3. Can it be slow?

Where are the page tables stored

Page tables can get terribly large.

e.g. 32 - bit address space

4 KB page size

4 KB = 2^{12} bytes

offset = 12 bits

VPN = 20 bits

2^{20} translations ~ million

OS would have to manage per process

assuming we need 4 bytes per PTE to hold the physical translation + some other stuff

4MB memory per process

100 processes ⟹ 400 MB just for translations

Not good

For 64-bit ↪ even larger gruesome

Page tables are so big → ∴ not stored in any special hardware

↓

store in memory

Page tabe data structure ⤳ older systems (hardware determined)

↓

(modern systems) flexibly managed by the OS.

What's actually in the Page Table?

→ DS to map VPN to PFN

→ simplest : linear page table
↓
just an array
→ indexed by VPN

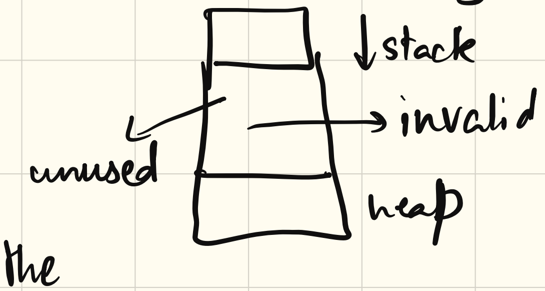
page table [VPN] ← PFN

→ more advanced DS in later chapters

Contents of each PTE (page-table entry)

① Valid bit → e.g when a program starts running

if process tries to access such address (invalid)
↓
generate trap to the OS
↳ terminates the process



Valid bit → crucial for supporting sparse address space

→ remove the need to allocate physical frames for unused memory.

② Protection bits

→ read write execute

traps if different permission than expected.

③ Present bit

↳ physical memory or disk? (swapped out ?)

Swapping — address spaces larger than physical memory.

Swapping allows the OS to free up physical memory by moving rarely used pages to disk.

④ Dirty bit

⑤ Reference / accessed bit

↳ whether the page has been accessed

useful during page replacement → to know which pages are popular.

⑥

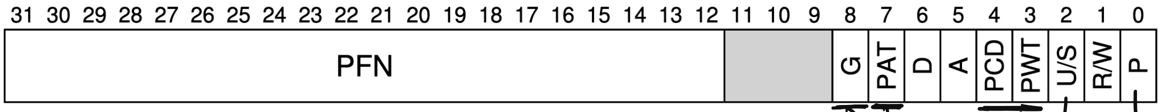


Figure 18.5: **An x86 Page Table Entry (PTE)**

Why no valid bit?

P ✓

P = 1 ⟹ both present and valid

P = 0 ⟹ may not be present but is valid

OR

not present
not valid

triggers trap to the OS

use add^n structure it keeps to determine

if the page is valid, and thus perhaps should be swapped back in.

or not

Illegal memory access

Paging: Also Too Slow

→ Page Tables might be too big

→ They can slow things down too.

movl 21 , %eax

↓ first fetch the proper PTE

→ physical address 117

Slides: important stuff

* Executing program

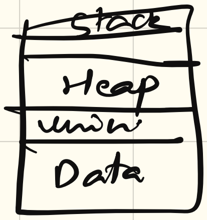
→ File to process memory view.

* All processes have same add^{r} space.

→ OS performs memory virtualisation with the help of hardware.

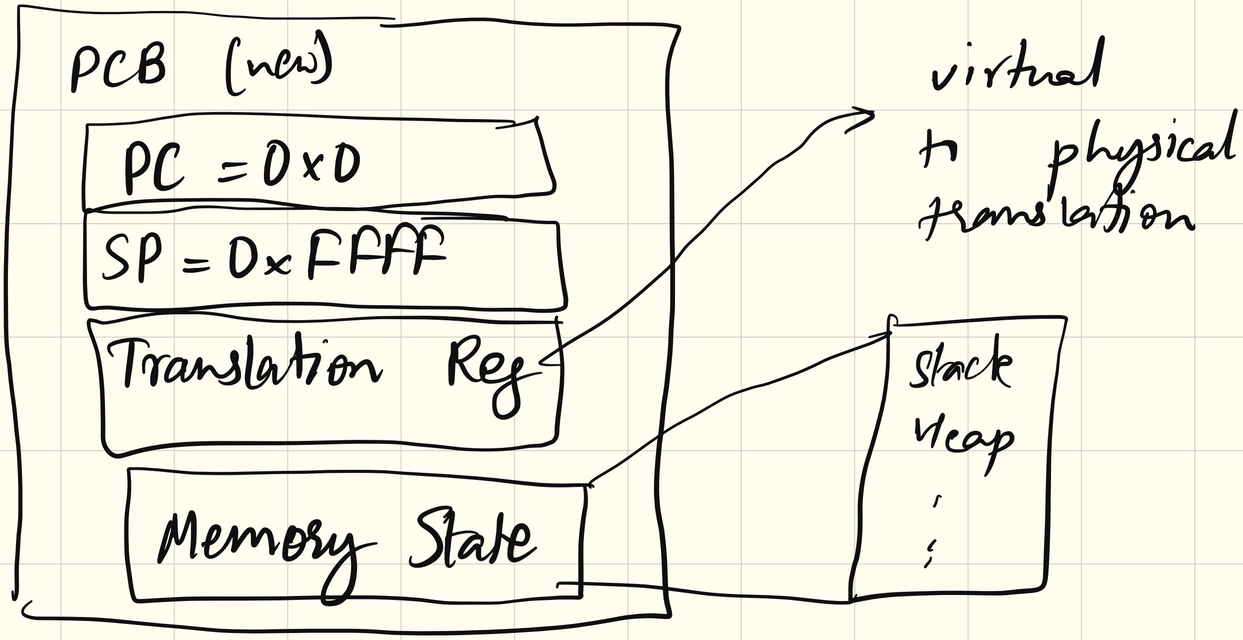
* Responsibilities of OS during program load :

→ Create add^{r} space →



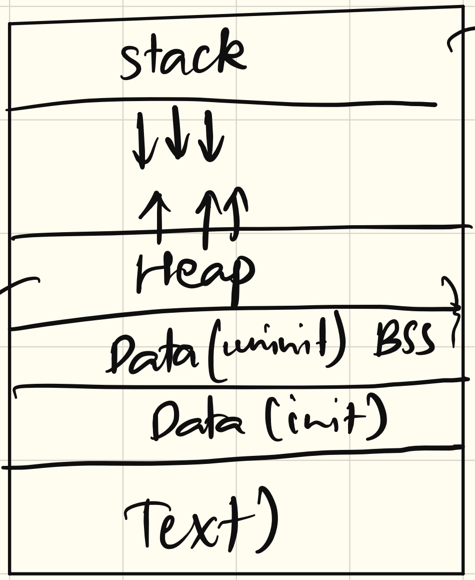
→ load binary

→ Update PCB register state



Memory APIs

→ User has no direct control on physical memory .



→ no special system calls

BSS → brk, sbrk

brk → input void* address → make end of BSS = addr

sbrk → input long size → make end = end + size

→ return old addr

sbrk(0)
↳ returns the current location of BSS

* MAP_FIXED
NULL ⇒ remap things

mmap: discontiguous allocation

e.g: allocate 4096 bytes w/ read + write

prt = mmap (NULL, 4096, PROT_READ | PROT_WRITE, MAP_ANONY | ... , -1, 0 }

NULL → start addr; 4096 → len; PROT_READ | PROT_WRITE → prot; MAP_ANONY | ... → flags; -1 → fd ??; 0 → offset

→ etext → end of Text
edata → end of data (initialijed)
end → end of BSS
} at program load time

→ printing addr of f^n and vars:

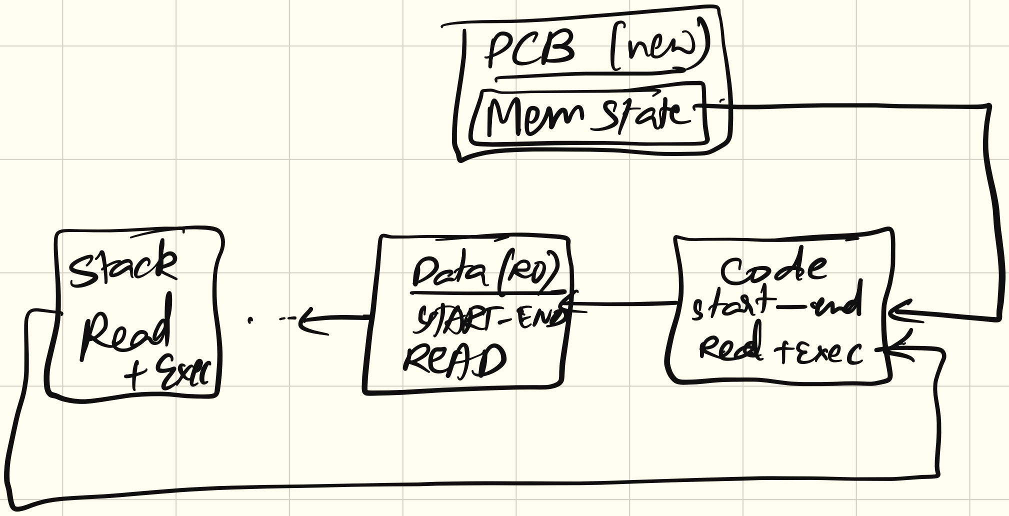
Linux : /proc/<pid>/ maps

Example: Memory State of PCB

→ Circular list

START and END never overlap b/w two segment areas.

→ Can be merged/extended if permission matched.



* Inheriting Address Space through fork()

→ child inherit the memory state of the parent

↳ data structure is copied into child PCB

→ Any change through mmap() or brk() is per-process

* Overriding Address Space through exec'

→ Add^{r} is reinitialized using the new executable

→ Changes to newly created AS depends on the logic of new process

Address Space Granularity

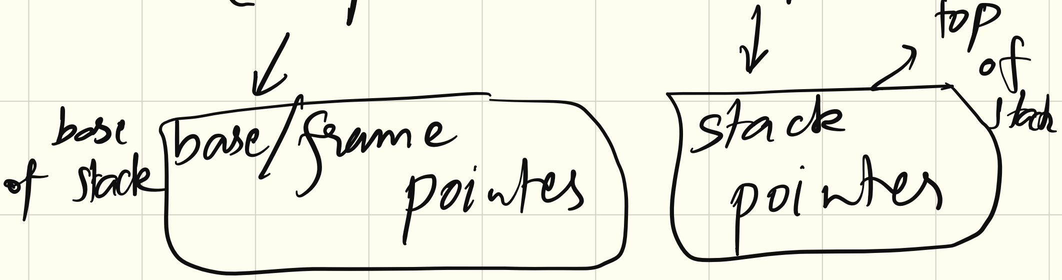
* Translation at address space granularity

→ ISA, x86

register	mov %rcx, %rax
immediate	mov $5, %rax
absolute	mov 8000001, %rax
indirect	mov (%rcx), %rax
displacement	mov -16(%rbp), %rax

→ examples

→ compiler: does not know stack address, hence blindly uses the register (rbp and rsp)



OS during binary load

```
load_new_executable ( PCB *current, File *exe){
    verify (exe)
    reinit_addr_space (current -> mmstate);
    allocate_phys_mem (current)
    load_exe_to_phys_mem (current, exe);
VAs { set_user_sp ( current -> mm_state ->
    {                              stack_start);
    { set_user_pc (current -> mm_state ->
                                   code_start)

    return to user
}
```

Process State After exec( )

→ OS configures the base register depending on the physical loc^n

PCB

user execution state

PC = 0

SP = 8KB

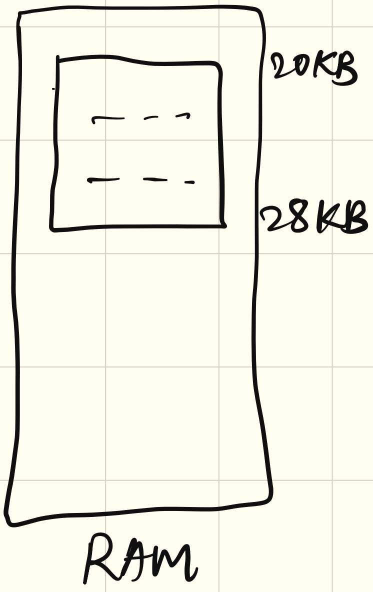
T_Base = 20KB

CPU

PC = 0

SP = 8KB

Base = 20KB



instr fetch (vaddr = 10)

→ instr fetch (padd^r = 20KB + 10)

"push %rbp"

→ Assuming RSP = 8KB

"push %rbp" → results in a memory store at addr (8KB − 8)

CPU translates the add^r to (28 KB − 8)

→ How to stop illegal accesses?

VA size = 8 KB

Access = 20 KB

↓

Phy Add^{r} = 40 KB $\boxed{\times}$

→ Limit Registers

↪ can be changed from privileged mode

→ Hardware raises a fault if the program violates the limit

OS fault handler may kill the process.

PCB:

User exe

PC = 0

SP = 8KB

T-Base = 20

T_limit = 8

CPU

PC = 0

SP = 8KB

Base = 20 KB

Limit = 8 KB

∴ How is memory isolation achieved?

→ limit register.

Context switch

→ Base and limit register values saved in the outgoing process PCB during context switch.

→ Loaded from PCB to the CPU when a process is scheduled

Disadvantages of Translation at AS Granularity

→ Physical memory must be greater than AS size.

  → against abstraction philosophy

- → Memory inefficient
  - → Physical memory size is same as addr space size irrespective of actual usage
    - → Memory wastage
  - → Degree of multiprogramming is very less.

Segmentation

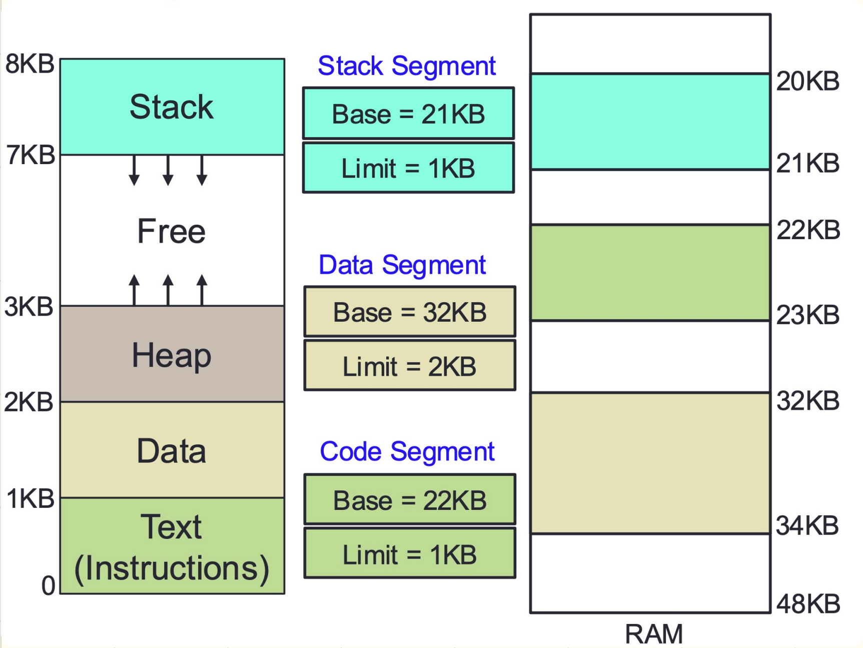
- → Extension of the basic sheme w/ more base-limit register pairs.

- → Ex: code add^r
  data add^r
  stack add^r

Segmentation : Explicit Addressing

→ Part of the code is used to explicitly specify segments.

→



VA = 8 KB ; add^{r} length = 13 bits ; 3 segments

→ Two MSBs used to specify the segment :

00 → code

01 → data

11 → stack

→ hardware selects the segment register based on the value of 2 MSB bits and the rest of the bits are used as offset.

→ Max size of each segment = 2KB

→ Physical allocation is still done on an on-demand basis

Disadvantages with explicity addressing

→ Inflexible

→ Data and Stack cannot be sized dynamically

→ Wastage of VA space,

→ in our example, 2KB VA is unusable

Segmentation: Implicit Addressing

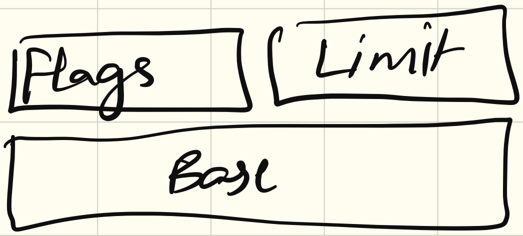
→ Hardware selects the segment register based on the operation

→ Code segment for instruction access
  → fetch $addr^r$, jump target, call $addr^r$

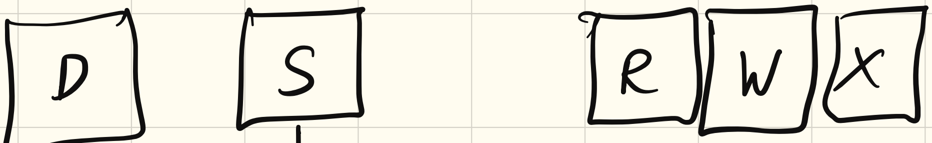
→ Stack segment for stack operations
  → push pop, indirect addressing with SP, BP

→ Data segment for other $addr^r$

Segmentation : (Protection and Direction



Flags



D ↓ direction (+ or -)

- for stack ↓ used to calculate H/W add^r

S ↓ privilege ↓ useful in code segment

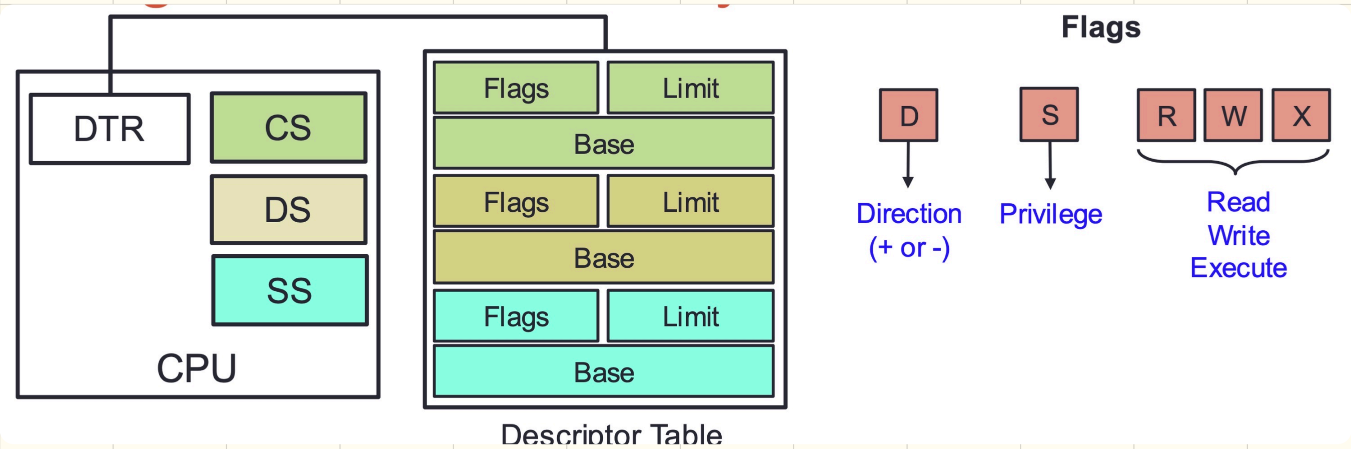
RWX: enforce isolation and sharing

Segmentation in Reality

→ DTR : descriptor table register is used to access descriptor table

→ # descriptors depend on architecture

→ Separate descriptors for user and kernel mode



Advantages of Segmentation

→ Easy and efficient addr translation

→ Save memory wastage for unused addresses

Disadvantages

- → External fragmentation
- → cannot support discontiguous sparse mapping

Paging

- → addresses external fragmentation due to variable size segments
- → allows discontiguous sparse mapping

→ Basic idea :

1. Partition the address space into fixed size blocks (call it pages)
2. Partition physical memory in a similar way (call it page frames)
3. OS : page $\xleftrightarrow[\text{mapping}]{}$ page frames

④ H/W uses the mapping to translate VA to PA.

→ Paging example (Pages)

32×1024

$2^5 \; 2^{10}$

VA Size = 32 KB,

Page size = 256 bytes ⇝ 2^8

Add^r length = 15 bits

{0x0 – 0x7FFF}

$$\# \text{ of pages} = \frac{32\ \cancel{KB}}{1/4\ \cancel{KB}}$$

$$= 128$$

Example: For VA = $0x\underbrace{05}_{\text{page number} = 5}\underbrace{10}_{\text{offset} = 16}$

7 bits	8 bits
Page #	Offset

Paging example (Page frames)

Physical Add^r size = 64 KB

Add^r length = 16 bits {0x0 – 0xFFFF}

# of PFNS = 256

e.g: For PA = $0x\,\underbrace{1F}_{\text{PFN} = 31}\,\underbrace{51}_{\text{offset} = 81}$

Page Table Mapping :

→ each entry in page table is called a Page - Table Entry (PTE)

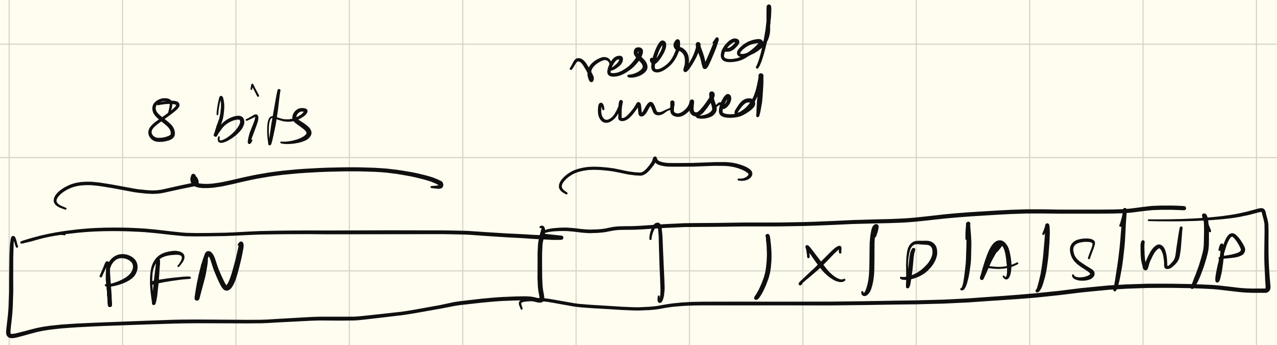
Page Table Walk

```
PTW ( vaddr V,  PTable  P)
  // returns  physical  address
{
    Entry  =  P[ V >> 8 ]
    if (Entry. present)
        return  (Entry. PFN << 8)
                     + (V & 0x FF) ;
    Raise  Page Fault;
}
```

Where is page table stored?

→ Page table is stored in the RAM.
  Page table registes (CR 3 in x86) contains the addr.

Structure of PTE



P → present bit → entry is valid

W → write bit → write allowed

S → privilege bit, 0 → only kernel mode access

A → accessed bit, → set by H/W during walk

D → dirty bit → set by H/W during walk

X → execute bit (instruction fetch)

Maximum size of physical memory

→ PFN → 8 bits

page size = 256 bytes = $\frac{1}{4}$ KB

$$\therefore \quad \text{Max RAM size} = 2^8 \times \frac{1}{4} \text{ KB}$$

$$= 64 \text{ KB}$$

Address Translation: Multilevel Paging and TLB

→ With increased addr space size, single level page table entry is not feasible:

→ ↑ page size → fragmentation ↑

→ Small pages may not be suitable to hold all mapping entries.

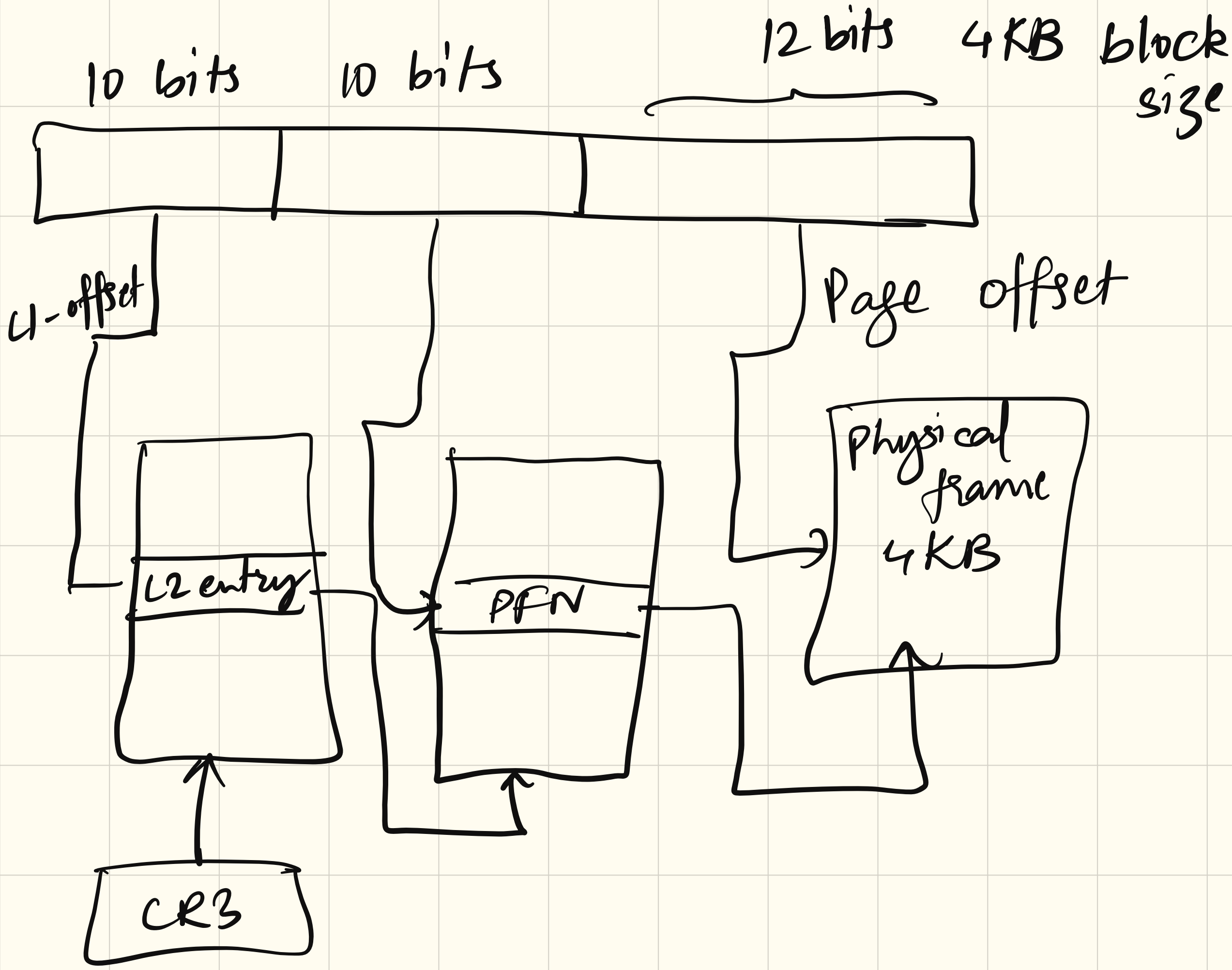
* Example: Paging with 32-bit AS

→ Sol^n: Multi-level page table.

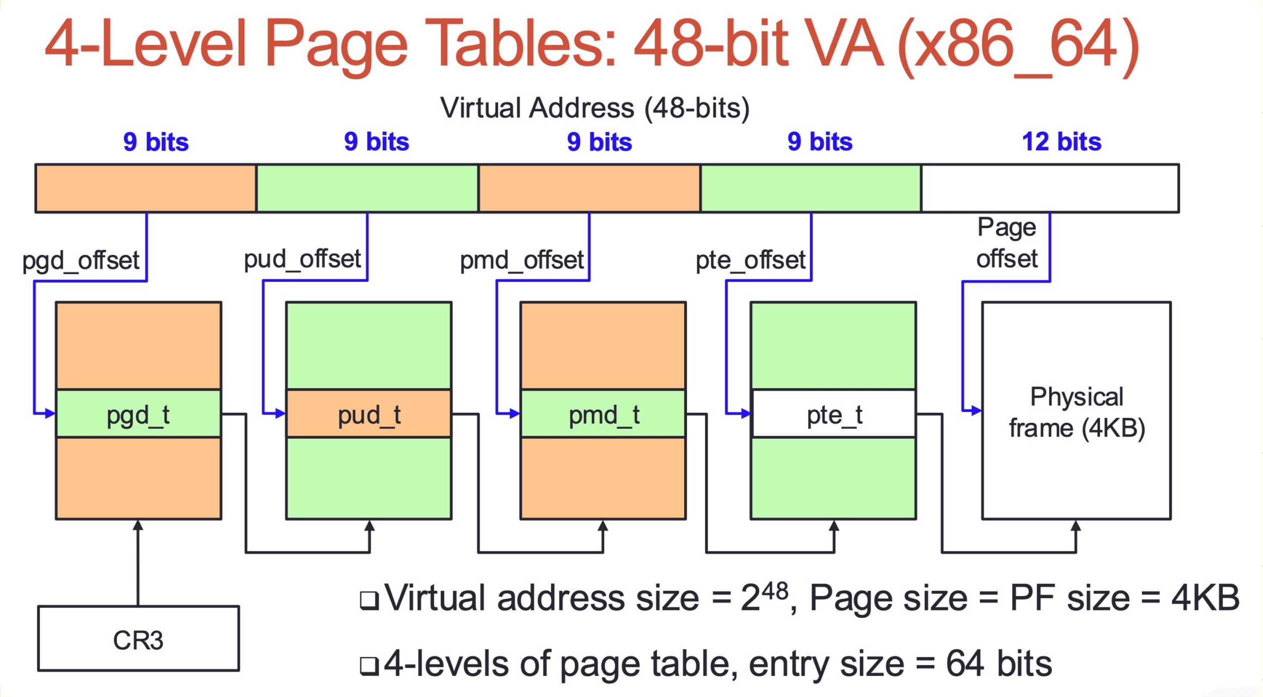
Two-level Page Tables (32-bit VA)

→ Level-1 page table contains entsies pointing to level-2 page table stsnctures

→ L2 entry contrains PFN + flags



→ 4 level page table : 48-bit VA
(x86_64

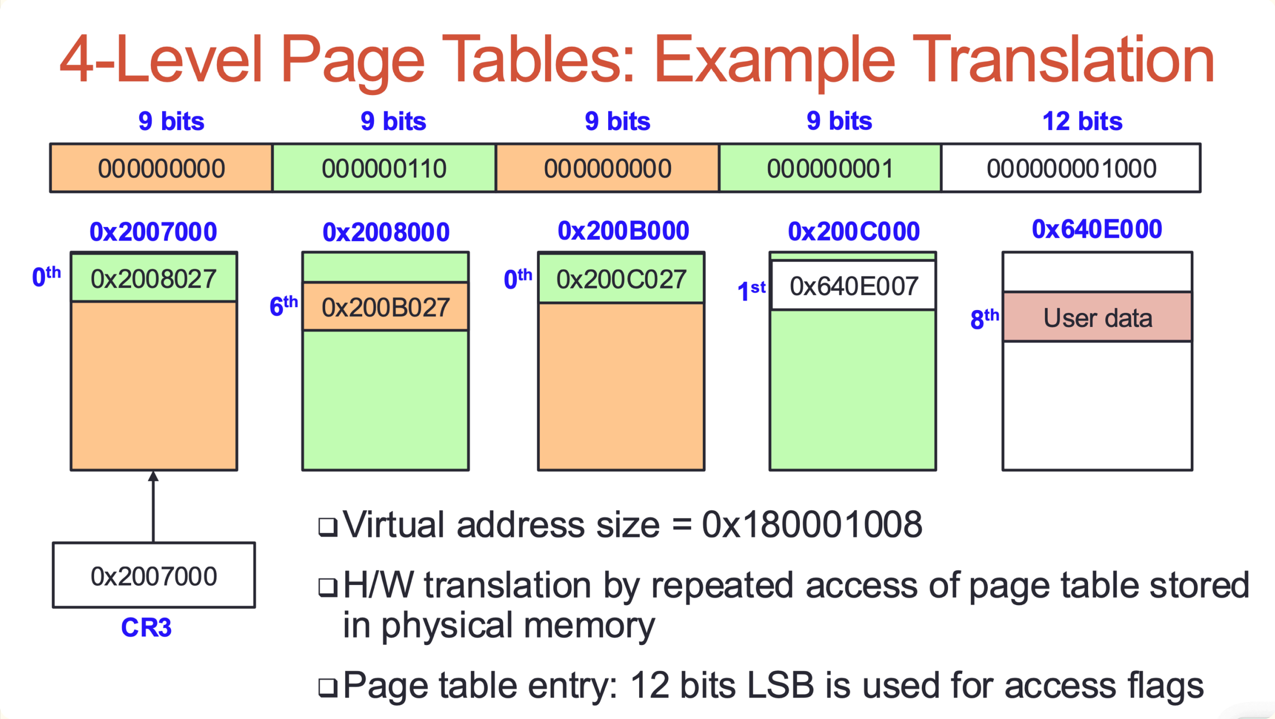


4-Level Page Tables: 48-bit VA (x86_64)

Virtual Address (48-bits)

9 bits	9 bits	9 bits	9 bits	12 bits
pgd_offset	pud_offset	pmd_offset	pte_offset	Page offset

- ❑ Virtual address size = 2^{48}, Page size = PF size = 4KB
- ❑ 4-levels of page table, entry size = 64 bits



4-Level Page Tables: Example Translation

9 bits	9 bits	9 bits	9 bits	12 bits
000000000	000000110	000000000	000000001	000000001000

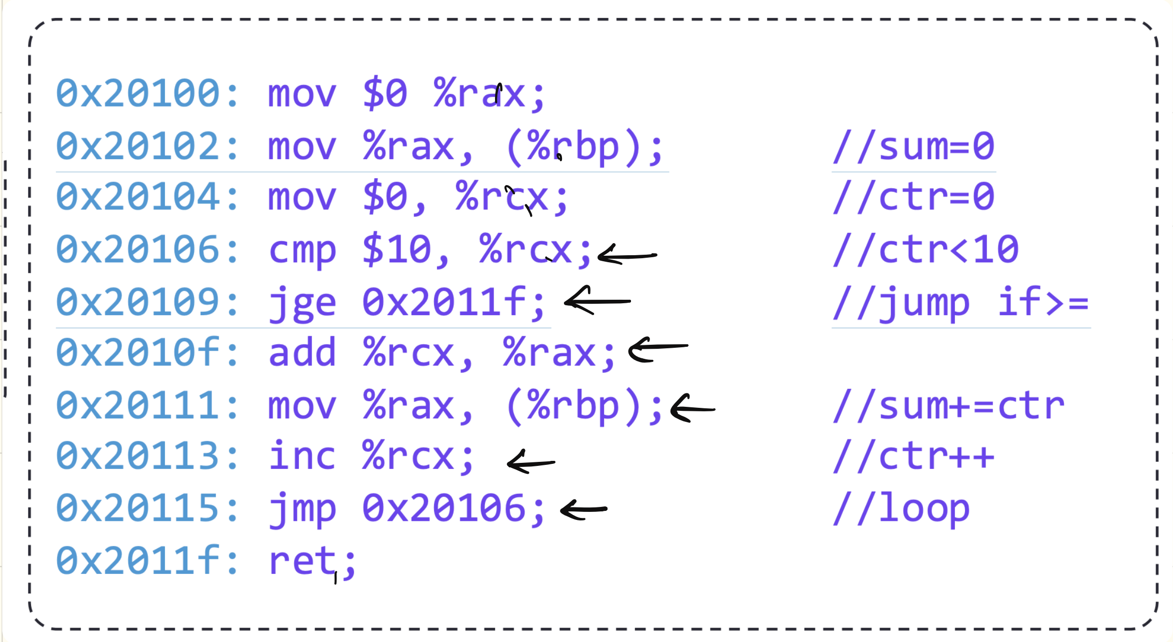
- ❑ Virtual address size = 0x180001008
- ❑ H/W translation by repeated access of page table stored in physical memory
- ❑ Page table entry: 12 bits LSB is used for access flags

Translation efficiency

Consider 4-level page table, how many memory accesses required for translation?

```
sum = 0;
for (ctr = 0; ctr < 10; ctr++)
    sum += ctr
```

Assume no cache



```
0x20100: mov $0 %rax;
0x20102: mov %rax, (%rbp);        //sum=0
0x20104: mov $0, %rcx;            //ctr=0
0x20106: cmp $10, %rcx; ←         //ctr<10
0x20109: jge 0x2011f; ←           //jump if>=
0x2010f: add %rcx, %rax; ←
0x20111: mov %rax, (%rbp); ←      //sum+=ctr
0x20113: inc %rcx; ←              //ctr++
0x20115: jmp 0x20106; ←           //loop
0x2011f: ret;
```

Instruction execution: loop = 10 x 6

others = 5

Memory accesses during translation = 65 x 4 = 260

Data / stack access : initialization = 1

loop = 10

Memory access during translation = 11 * 4 = 44

Distinct accesses ?

Assume : stack address range

0x 7FFF000 – 0x8000000

One code page (0x 20)

and one stack page 0x 7FFF

→ cache these : TLB

Paging w/ TLB: Translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FF	0x890

```
Translate ( vaddr v ) {
    PageAddress P = V >> 12;
    TLBEntry = Lookup (P);
    if (entry. valid)
        return entry. pte;
    entry = PageTableWalk (v);
    MakeEntry (entry);
    return entry. pte;
}
```

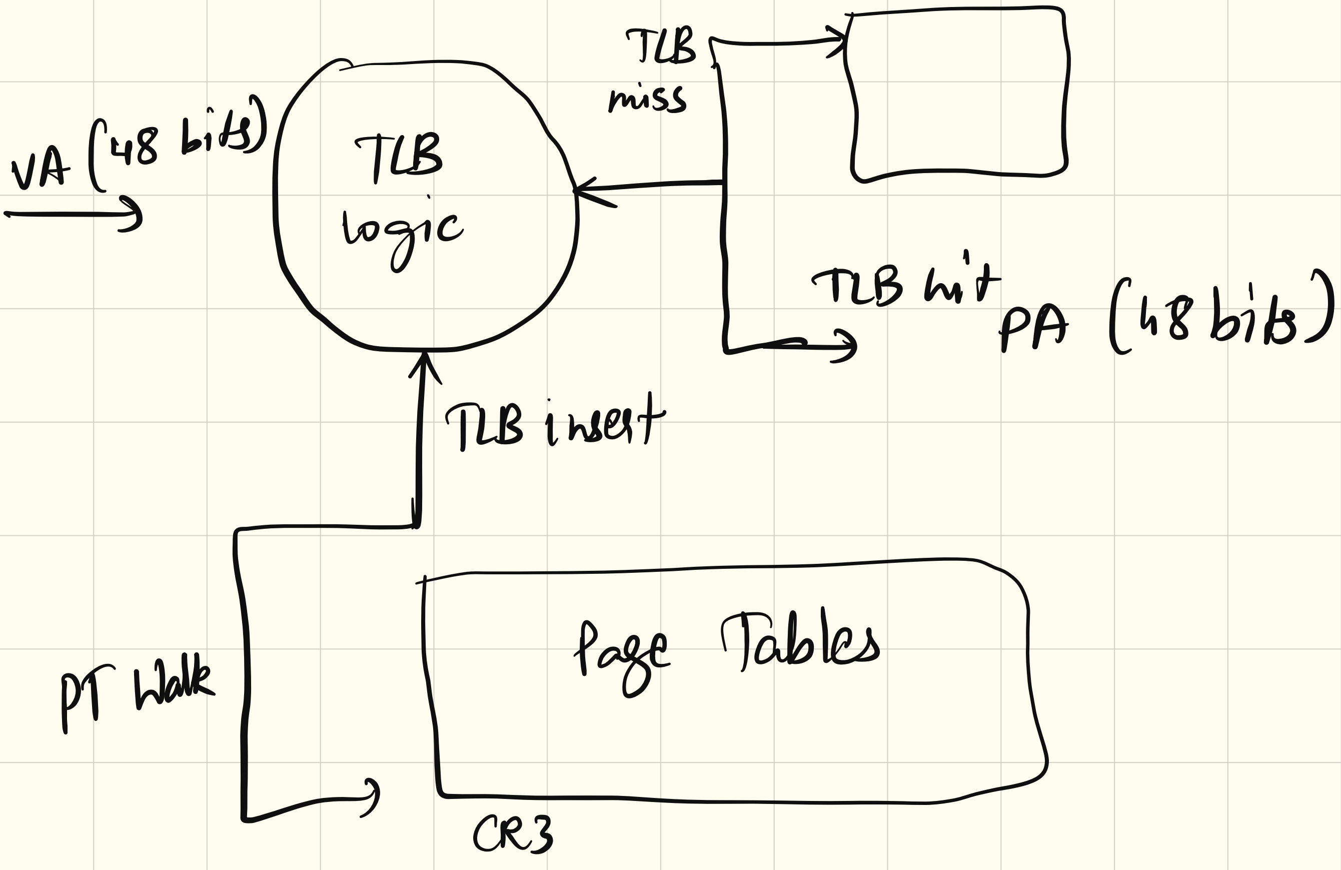
TLB → Translation Lookaside Buffer

→ hardware cache

→ stores Page to PFN mapping

→ after first miss for instr fetch, all accesses hit

Address Translation (TLB + PTW)



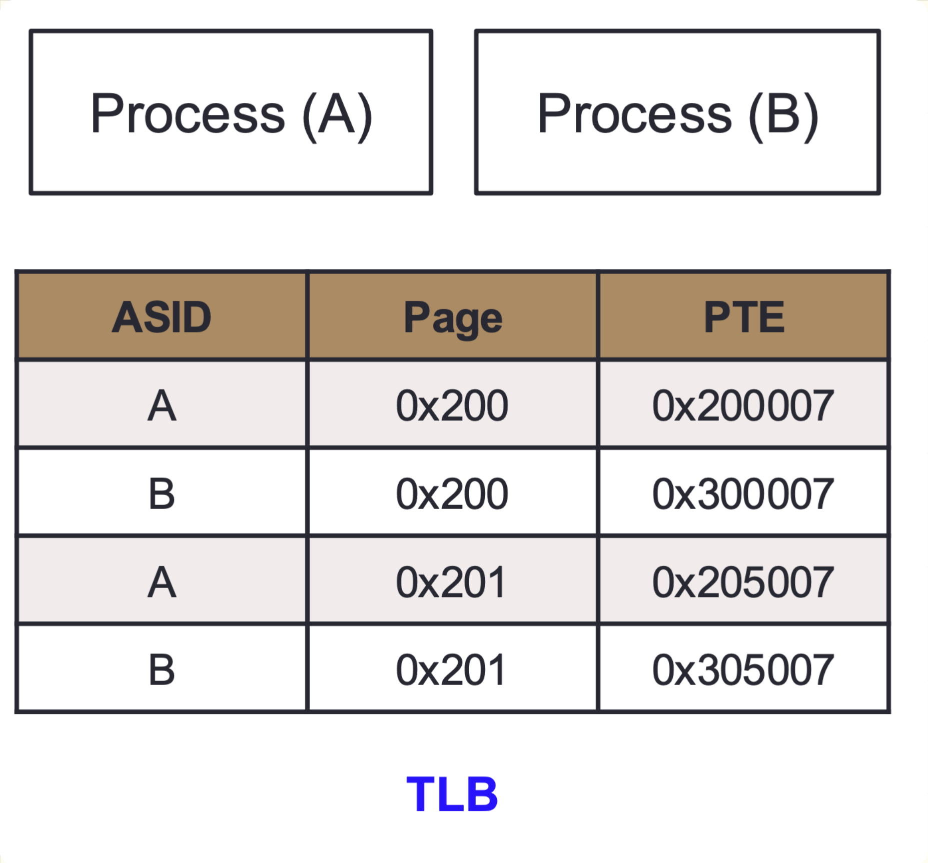
→ Seperate TLBs for instruction an data, multilevel TLBs

→ In x86, OS cannot make entries into the TLB directly, it must flush entries

* How is TLB shared across multiple processes?

Option 1: flush whole TLB → performance problem

Option 2: Address Space Identifier (ASID) along with each TLB entry to identify the process.



Process (A)

Process (B)

ASID	Page	PTE
A	0x200	0x200007
B	0x200	0x300007
A	0x201	0x205007
B	0x201	0x305007

TLB

Why is page fault necessary?

→ Page fault is required to support memory over-comittment through lazy allocation and swapping.

Pagefault Handling in x86

```
if ( !pte.valid ||
        (access == write && !pte.write)
        || (cpl != 0  && pte.priv==0))
      { CR2 = address;
        errorcode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise PageFault;
  } // simplified
```

cpl = current priviledge level

pte.priv = priv level of pte.

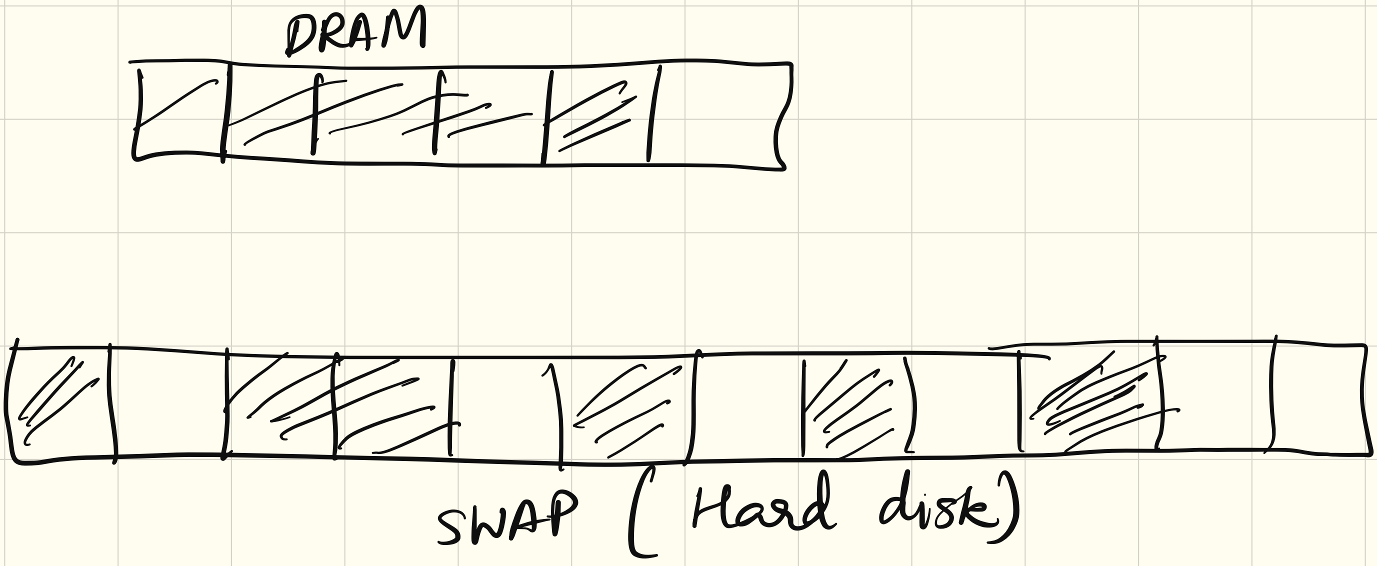
→ error code is pushed into the kernel stack by the hardware (x86)

OS Fault Handler

```
                              virtual
HandlePageFault (u64 address , u64 error_code)
{
    entry = P[V >> 8];
    if (AddressExists(current -> mm_state,
                              address) &&
            AccessPermitted(current -> mm_state,
                              error_code)){
            PFN = allocate_pfn();
            install_pte(address, PFN);
            return;
    }
    RaiseSignal(SIGSEGV);
}
```

Page Fault and Swapping

Swapping (swap-out)



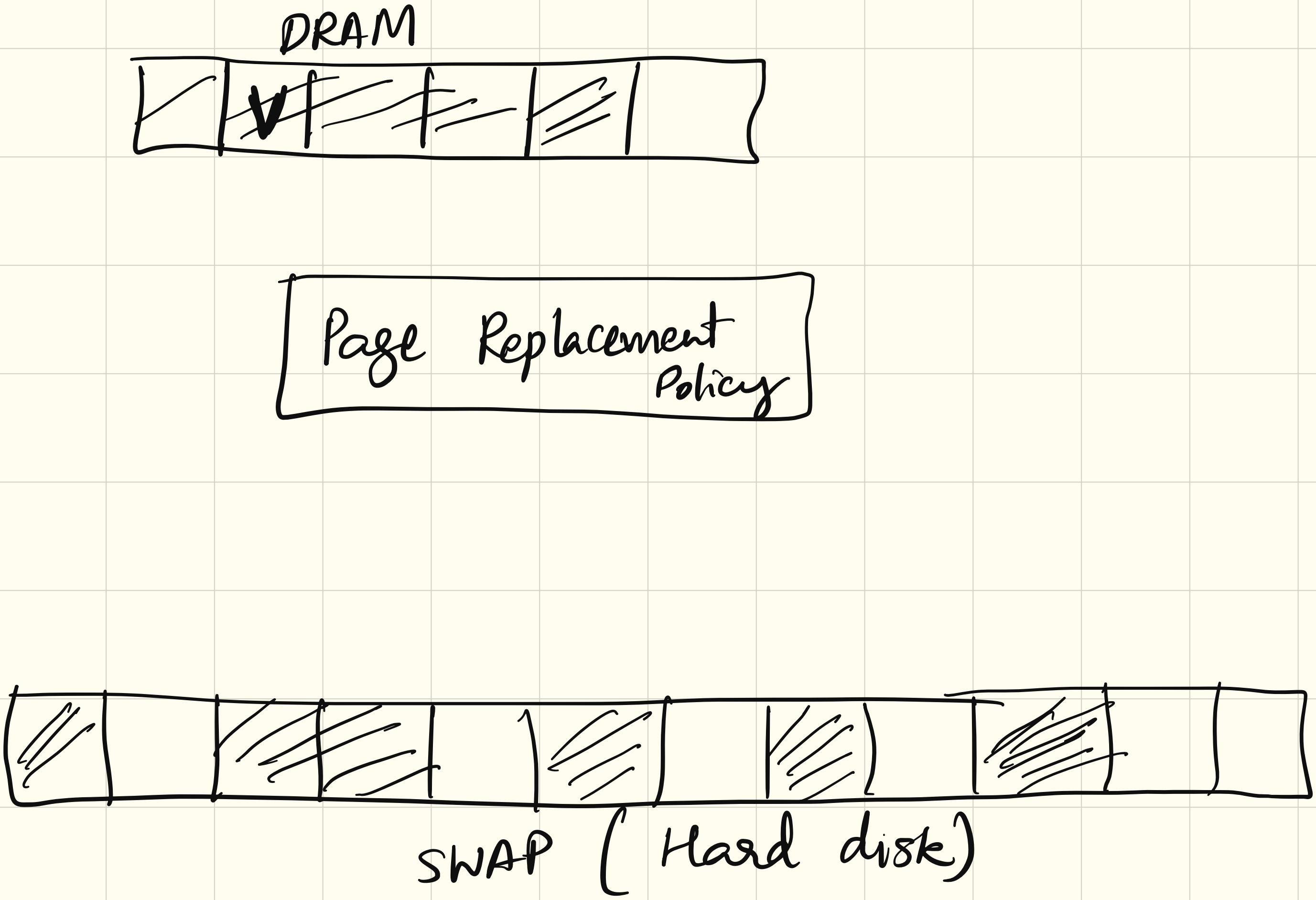
AllocatePFN():

free PFNs ↓

cannot break promise to applications,

∴ swap out, but which one?

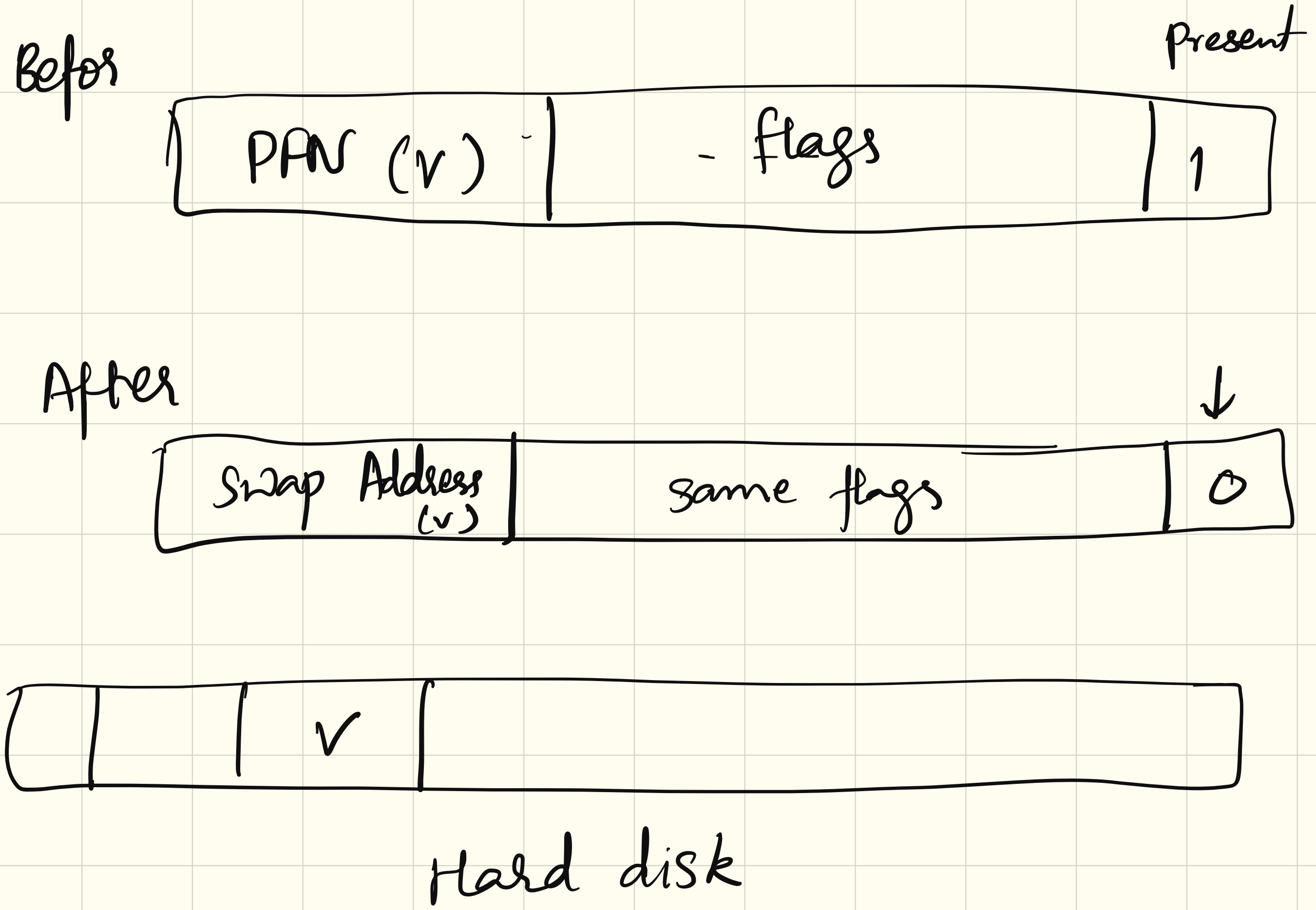
Decide using a page replacement policy



→ Update the present bit of victim entry to 0

→ page fault

Replace PFN with swap Address



Any future translation will result in page fault.

Page Fault Handler will copy it back from the swap device

Swap-in:

```
                         virtual
HandlePageFault (u64 address , u64 error_code)
{
    entry = P[V >> 8];
    if (AddressExists (current -> mm_state,
                                  address) &&

          AccessPermitted (current -> mm_state,
                                       error_code)){

          PFN = allocate_pfn();
          if (is_swapped_pte(address))
               swap in (getPTE (address), PFN);
          install_pte (address, PFN);
          return ;
    }
    RaiseSignal (SIGSEGV);
}
```

Page Replacement

Objective: Minimize # of page faults (due to swapping)

- → 3 parameters
  - → A given sequence of accesses to virtual pages
  - → # of memory pages (frames)
  - → page replacement policies

- → Metrics to measure effectiveness
  - → # of page faults
  - → Page fault rate → fraction of memory accesses that result in page fault
  - → AMAT

Belady's Optimal Algorithm (MIN)

→ Strategy: Replace the page that will be referenced after the longest time.

→ Example:

→ # of frames = 3

→ Reference sequence (in temporal order)

				1, 4, 5		1, 2, 5	
1 →	3 →	1 →	5 →	4 →	1 →	2 →	5
Yes	Yes	No	Yes	Replace 3	No	Replace 4	No ↓

3 ←	5 ←	2 ←	2
Yes	No	No	No

# of page faults = 6

Belady's MIN algorithm is proven to be optimal, but impractical as it requires knowledge of future accesses.

FIFO

Strategy: Replace the page that is in the memory for the longest time

Ex: # of frame = 3

1	1, 3	1, 3	1,3,5	4,3,5	4,1,5	4,1,2	5,1,2
1 →	3 →	1 →	5 →	4 →	1 →	2 →	5
Yes	Yes	No	Yes	Yes	Yes	Yes	Yes

↓

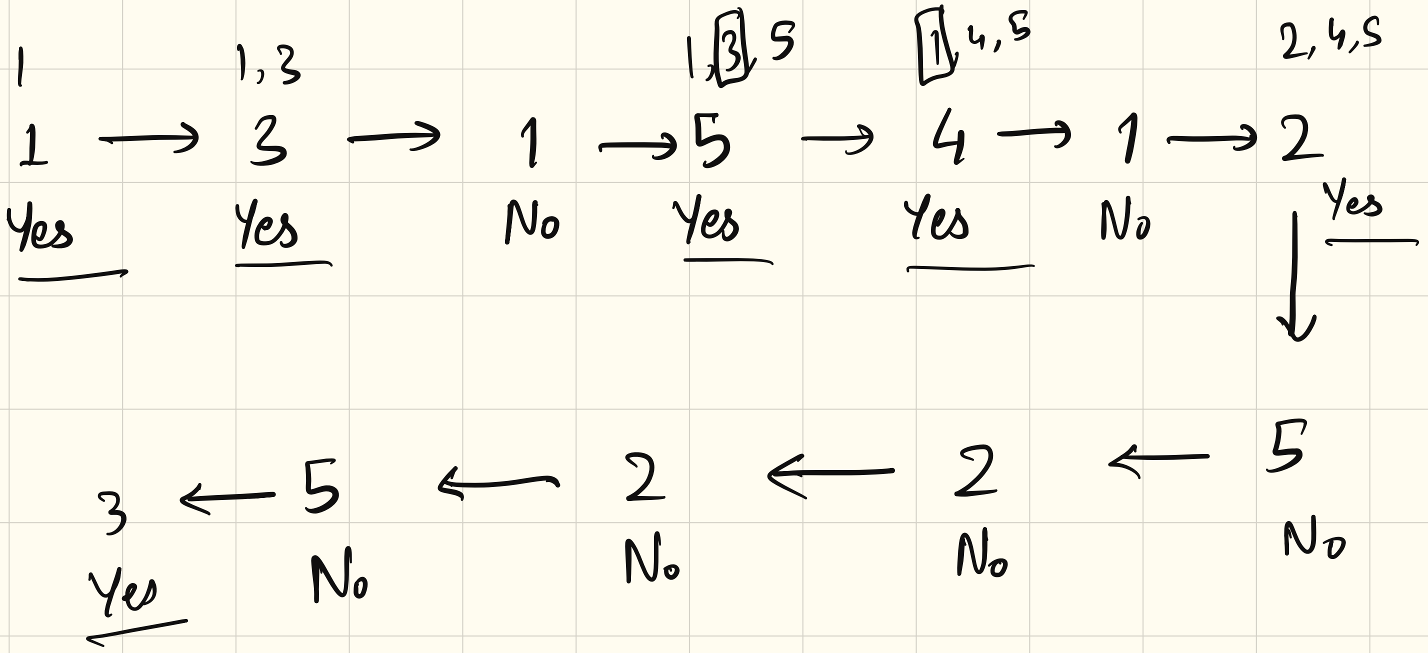
3	←	5	←	2	←	2
Yes		No		No		No

8 page faults — 3 cold start

Least Recently Used (LRU)

Replace the page that is not referenced for the longest time.

Eg: # of frame = 3



→ LRU is shown to be useful for workloads with access locality

→ approximate using CLOCK