CSE2431 – Lecture Topic 5
Virtual Memory (Part 3)
Paging

The Ohio State University
College of Engineering

Virtual Memory (pt 3) Paging

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: Chapter 18-19 in Required Textbook

Previous lecture

- Dynamic Relocation
  - How could we perform address translation
  - How to determine the correct segment for translation
  - Splitting and Coalescing Mechanism, Implementation and Optimization
- Virtual memory procedures (part 3)
  - Dynamic relocation
  - Paging
  - Swaping

Outline: Virtual Memory (Part 3)

- Memory overview (part 1)
- Virtual memory procedures (part 2)
  - Dynamic relocation
- Virtual memory procedure (part 3)
  - Paging
  - Swaping

Revision

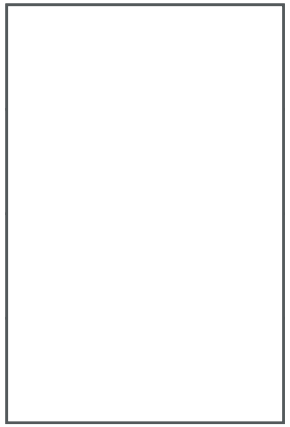
- Physical memory: a fixed number of bytes
- Users' data: variable-sized objects (e.g. `segments`, `malloc`, …)

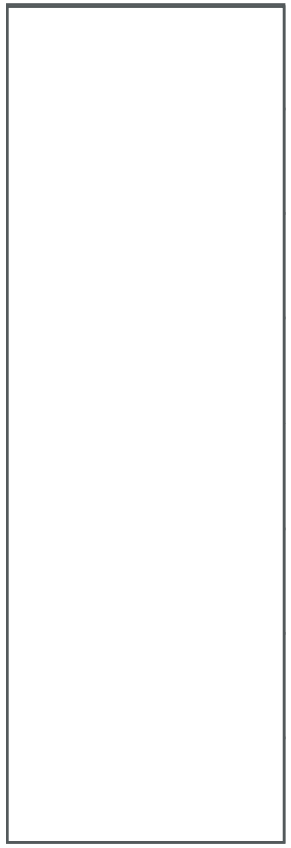
- Dynamic relocation: chop memory into variable-sized pieces
  - Problem: internal fragmentation and external fragmentation

- We will learn another solution: chop memory into fixed-sized pieces (pages)

Overview with an example

A process needs 64B memory

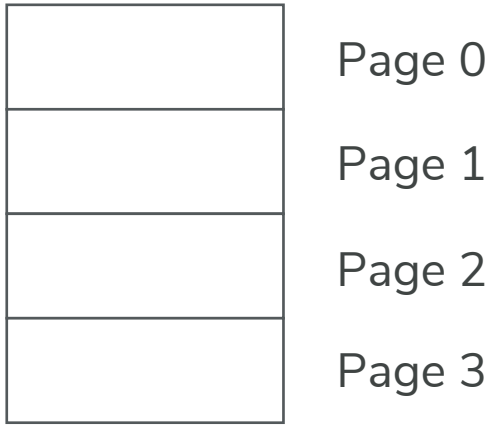




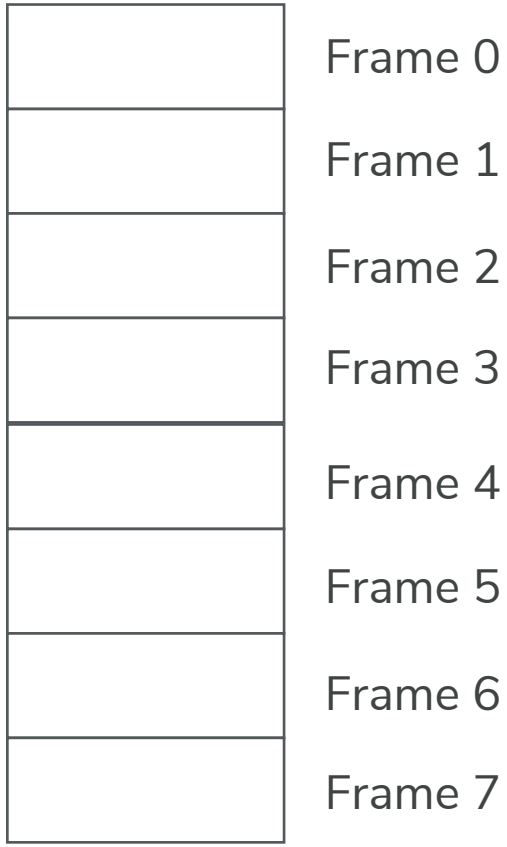
Physical Memory: 128B

Overview with an example

A process needs 64B memory



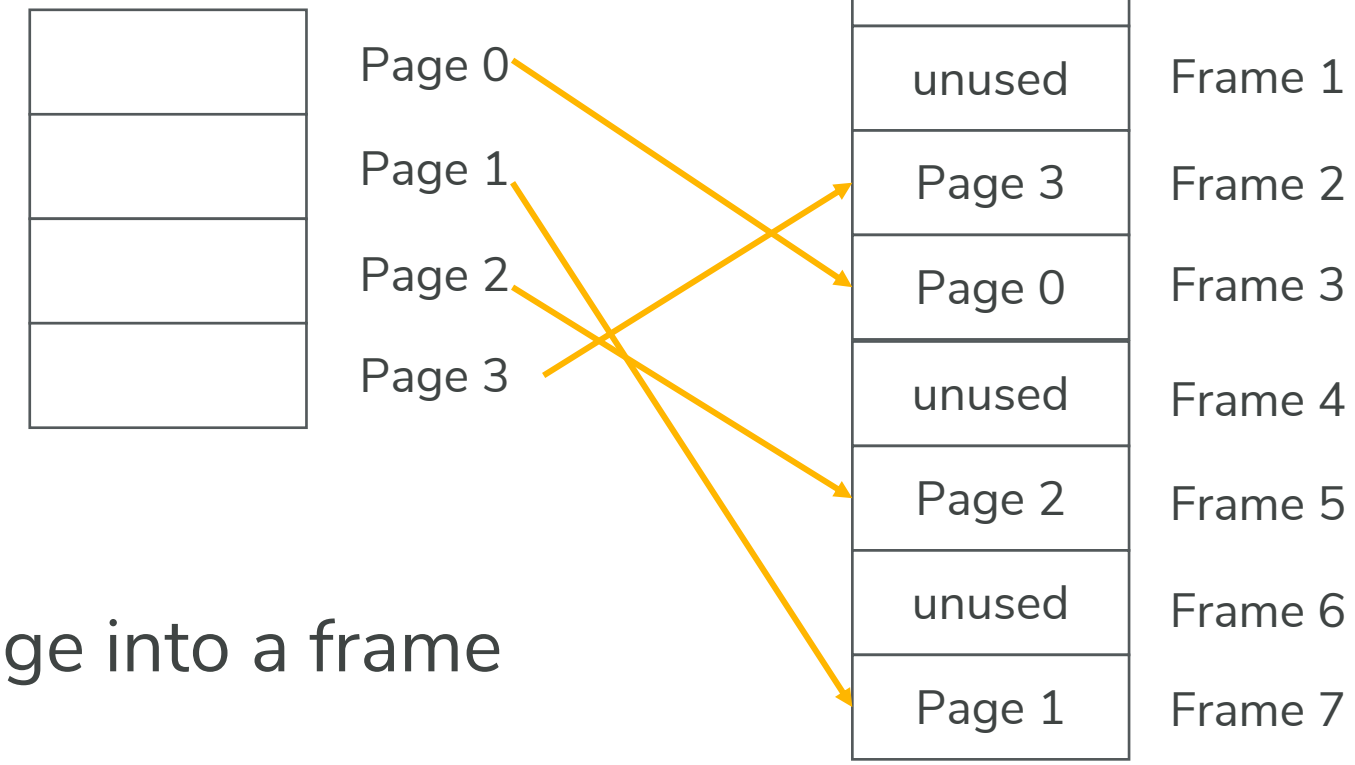
Physical Memory: 128B



Divide memory space into 16B pieces (page for virtual memory and frame for physical memory)

Overview with an example

A process needs 64B memory



Place each page into a frame

Paging: Advantages and Challenges

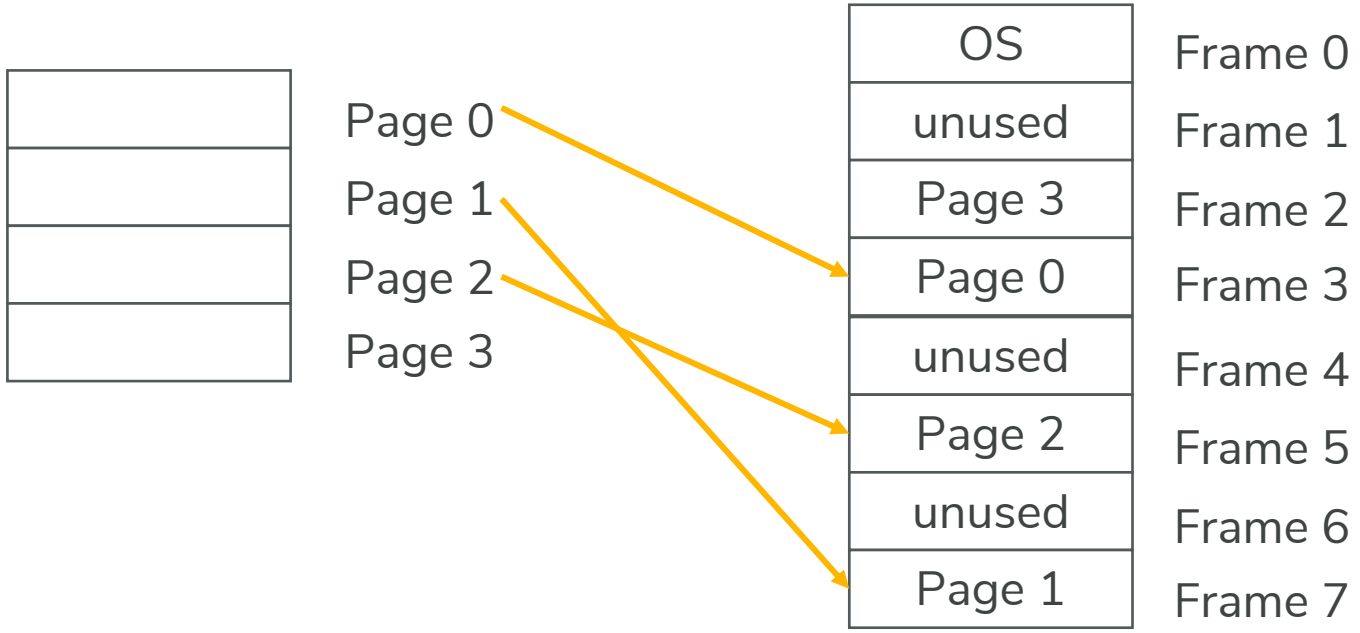
- Advantages:
  - **Flexibility**: no need to allocate the max possible memory. A process can request a new page when its memory is not enough.
  - **Free-space management** is simple: a list of free frames
  - **No external fragmentation**

- Challenges: how to translate a virtual address into a physical address?
  - Revision question: how to do the mapping with the dynamic relocation/segment approach?

Translating virtual address to physical add.

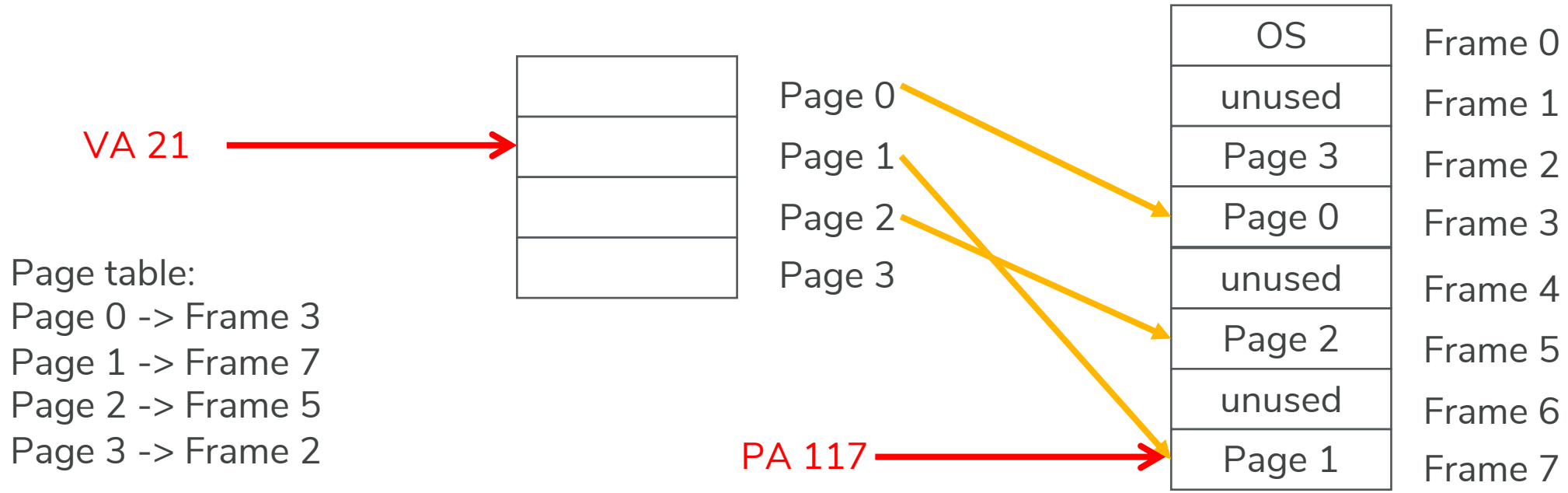
- Obviously OS needs to know the mapping from page to frame
  - This data structure is called a **page table**.
  - OS maintains **a page table for each process** (page 0 of process 1 is different from page 0 of process 2).

Page table:
Page 0 -> Frame 3
Page 1 -> Frame 7
Page 2 -> Frame 5
Page 3 -> Frame 2



Translating virtual address to physical add.

- Ex: What is the physical address (PA) of virtual address (VA) 21?
  - Assuming each page/frame has 16 bytes.
  - VA 21 is in Page 1. It is **5th byte** in Page 1 (i.e. **offset 5** in Page 1).
  - It maps to Frame 7 and still offset 5. So PA = 16*7 + 5 = 117.



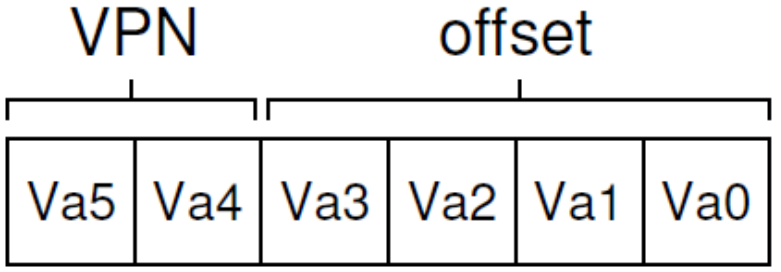
Translating virtual address to physical add.

- Translation: given a virtual address VA, how to calculate PA?
  - Calculate the corresponding page number and the offset in the page
    - Page = VA / page size; Offset = VA % page size
  - Map page to frame using the page table
    - Frame = map (page)
  - Calculate physical address PA
    - PA = Frame * frame size (same as page size) + offset

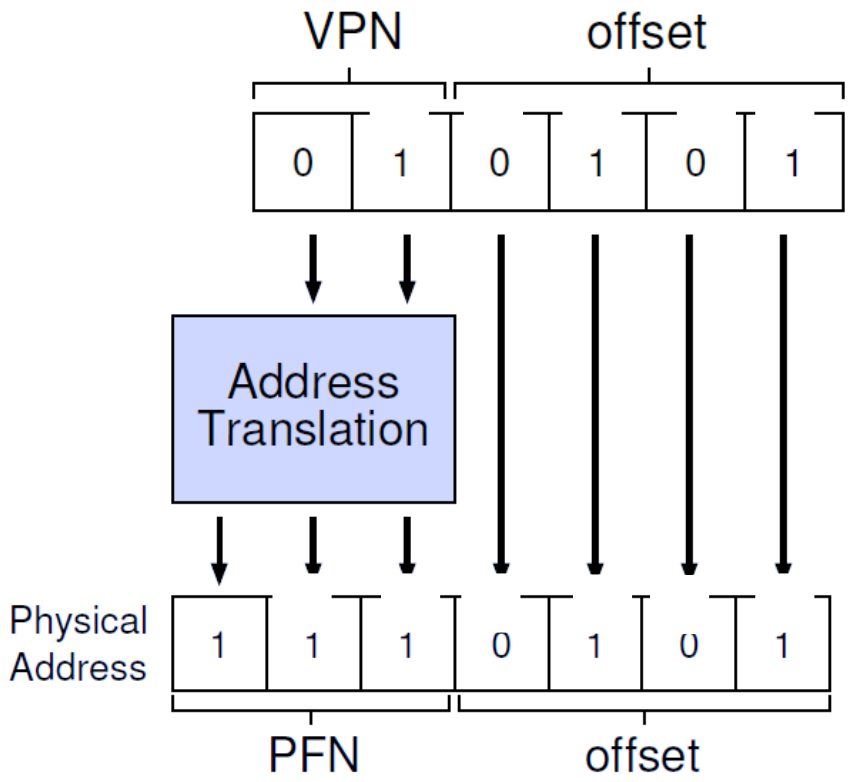
- If page/frame size is order of 2, which is a typical case, above calculation can be simplified

Translating virtual address to physical add.

- If page/frame size is 2^M, and an address has N bits
  - Offset = VA % page size = last M bits of VA. We have learned why in "Bit Operations".
  - Page number = VA / page size = first N-M bits of VA.
  - PA = frame number * frame size + offset = frame number << M + offset.
  - Since offset has M bits, PA can be further simplified as concatenating frame number and offset (e.g **111 << 4 + 0101 = 1110101 = 117_{10}**)
- Go back to the previous example: virtual address is 64B and each page is 16B.
  - What are N and M?
  - N = $\log_2(64)$=6. M = $\log_2(16)$=4.



Translating virtual address to physical add.



1. Divide VA into VPN and offset

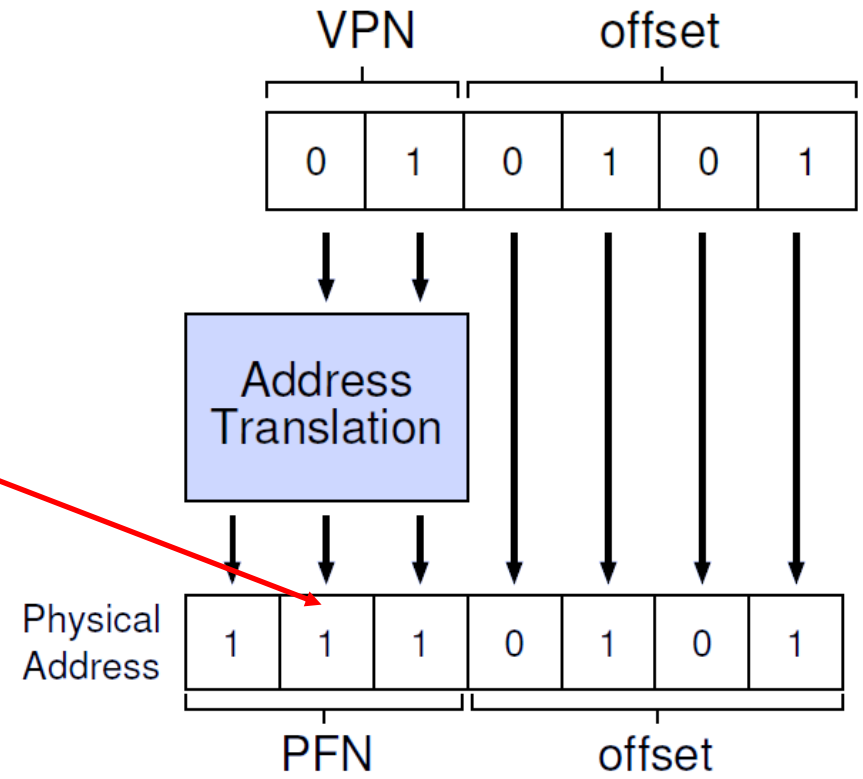
2. Map VPN to PFN
Offset doesn't change

3. Concatenate PFN and offset

Example

- What is the physical address of VA 21?
  - 21 = 010101b

Page table:
Page 0 -> Frame 3
Page 1 -> Frame 7
Page 2 -> Frame 5
Page 3 -> Frame 2



1. Divide VA into VPN and offset

2. Map VPN to PFN
Offset doesn't change

3. Concatenate PFN and offset

Implementaion

- So far we have talked about the basic idea of paging

- To implement it, there are many remaining questions:
  - What is the data structure for page table?
  - How large is a page table?
  - Where to store page table?
  - What is the overhead of address translation?

Data structure for page table

- Simplest solution: an array
  - array[x] stores the frame number of page x.
- We will learn other solutions, but let's start with this simplest solution.
- In practice, each array entry usually contains more information
  - Valid bit
  - Protection bits
  - Present bit
  - Dirty bit
  - Reference bit
  - ….
  - (Don't worry. We will learn them in details later)

Size of a page table

- Consider 32-bit address space and 4KB page/frame size
  - 32bit address space means VA has 32bits. It can support up to 4GB of virtual memory, which is actually not large today.
  - How many bits are for offset? 4KB = 2^{12}B, so 12 bits.
  - How many pages can a process have? 2^{32-12}.
  - Suppose each entry takes 4 bytes, size of a page table? 4 * 2^{20}= 4MB
  - Each process needs 4MB of space to store page table.
  - How many processes does you computer have?

- Most of today's computers use 64bit address space.

Where to store page table?

- Page table is **quite large**. It does not fit special hardware like high-speed cache.

- OS usually stores a page table at a specific location in memory.

Overhead of address translation

- Whenever an instruction accesses memory address VA
  - Calculate page number and offset of VA
  - Find frame number by referencing PageTable[page number]. **Note this is an additional memory access**
  - Calculate PA by combining frame number and offset
  - Access PA
- A memory access is turned into two memory accesses and some computation
  - At least twice overhead (memory access is usually much more expensive than computation).
  - This is not satisfactory.

Example

- Read Chapter 18.5 yourself

- Notes:
  - Without considering caching, each instruction will incur at least one memory access: CPU needs to fetch the instruction from memory.
  - Some instructions, such as mov, load, and store, will incur one or more additional memory accesses.
  - With paging, all these memory accesses will be doubled.

Summary

- Advantages of Paging:
  - Flexible
  - Simple for free space management
  - No external fragmentation

- Remaining challenges:
  - Page table can be large
  - Overhead is high

Faster translations (TLB)

- Problem:
  - Page table is quite large and does not fit into faster device.
  - Search page table requires an additional memory access

- Can you think of a solution? (Hint: it is a classic idea in CS)

- Basic idea: caching part of the page table in faster hardware
  - This cache is called a translation-lookaside buffer (TLB)
  - It is part of a CPU's memory-management unit (MMU)

Caching in general

- Faster devices are more expensive, so cannot use it to store all data
- Use faster device to store a subset of data that is frequently accessed
- Assumption: data has locality
  - **Temporal locality**: data that is accessed has a high likelihood to be re-accessed soon
  - **Spatial locality**: if data is accessed, adjacent data has a high likelihood to be accessed soon
  - They are not always true, but people find them to be true for many applications.

Basic workflow with TLB

- Calculate page number and offset from VA
- Search page number in TLB
- If success (TLB hit)
  - Calculate PA
- Else (TLB miss)
  - Search page number in page table
  - Put entry in TLB
  - Re-execute instruction
- (Privilege checking is ignored in this pseudo code. See book for detail)

TLB efficiency

- **Obviously TLB is efficient when hit ratio is high**
  - **Hit ratio depends on workload**

- **Exercise:** suppose your program accesses an 4096-element integer array sequentially, what is TLB hit ratio? Assuming
  - Each integer is 4B
  - Each page/frame is 4KB
  - The array starts at virtual address 0.
  - TLB is empty at the beginning. TLB has 16 entries.
  - Ignore memory accesses to fetch instructions

TLB efficiency

- When accessing array[0]
  - It is in page 0. Not in TLB. Miss. So put page 0 in TLB.
- When accessing array[1]
  - It is in page 0. In TLB now. Hit.
- When accessing array[2]
- ……
- When accessing array[1024]
  - It is in page 1. Not in TLB. Miss. So put page 1 in TLB.
- ……

TLB efficiency

- There is a TLB miss when accessing a new page. All following accesses hit TLB.
- Each page holds 1024 integers.
- So TLB hit ratio is 1023/1024. This is pretty high.
- Can you imagine a workload that incurs low TBL hit ratio?
  - Random access to big array has worst locality and incurs low TLB hit ratio.

Handling TLB miss

- It can be implemented either in hardware or software

- Hardware: some CPUs have built the logic in hardware
  - OS needs to tell TLB the location of a page table, which must be organized in the way specified by CPU.
  - TLB handles everything else.
- Software: some CPUs allow OS to handle TLB miss
  - TLB raises exceptions when a miss occurs.
  - TLB allows OS to manipulate its entries with special instructions.
  - OS can design its own format of page tables freely.

TLB content

- A typical TLB can have 32, 64, or 128 entries.
- An entry obviously has page number and frame number.

VPN | PFN | other bits

- An entry usually has some other bits:
  - Valid bit: whether an entry contains valid info. Set to false when initialized. Also used in context switch.
  - Protection bits: whether a page is readonly (same as in page table)
  - Address-space identifier (ASID): used in context switch
  - Dirty bit: whether a page has been modified (same as in page table)
  - .....

Context switch

- Revision: what is a context switch?

- Remember that each process has its own page table: page 0 of process 1 is different from page 0 of process 2
  - After a context switch, TLB still contains information from the old process, which is obviously not correct.

- Solution 1: empty TLB by setting all entries as invalid.
- Solution 2: use ASID

Context switch Problem Example

- Suppose Process P1 is running. It assumes the TLB might be caching translations that are valid for it. Assume, that the 10th virtual page of P1 is mapped to physical frame 100.
- Then if process 2 exists, and the OS soon decides to perform a context switch and run P2. If the 10th virtual page of P2 is mapped to physical frame 170. Then there is an issue for hardware here. → A crux

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

ASID during context switch

- Problem: page number is not unique across processes
  - Two processes can both have page 1
- Solution: instead of mapping page number to frame number, TLB can map (ASID, page number) to frame number
  - Each process has a unique ASID
  - (ASID, page number) is a unique identifier for a page.

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

A real TLB entry

- Read the textbook 19.7

Making page table smaller

- Let's go through the simple solution again
  - A page table is an array.
  - Array[x] stores the frame number of page x, plus some bits

- For **32-bit machines** with **4KB** pages
  - Max memory space of a process is 2^{32}.
  - It can have 2^{20} pages, so a page table can have 2^{20} entries.
  - Each entry takes 4 bytes. So the total space of a page table is 4MB.
  - One page table per process and a machine can run hundreds of processes in parallel.

Making page table smaller

- Opportunity: not every process uses all 4GB of memory
  - This means many entries in the page table are invalid.
  - Can we just store the valid entries?

- Problem: a process can use memory in a sparse way (e.g 0-1K, then 4K-8K, then 100K-101K, …)
  - This means valid entries scatter in the page table

- Can you think of a solution?

Making page table smaller

- Let's simplify the problem a little bit:
  - We have a big array
  - Most of its entries are invalid
  - Valid entries scatter in the array

- Can you think of a more compact way to store the array?

Solution 1: Bigger pages

- Obviously page table is smaller if page is bigger.
- But bigger page **increases internal fragmentation**.
- That's why in practice, most OSes use 4KB or 8KB pages.
  - Linux starts to provide huge page support (2MB pages).

Solution 2: Combining segmentation and paging

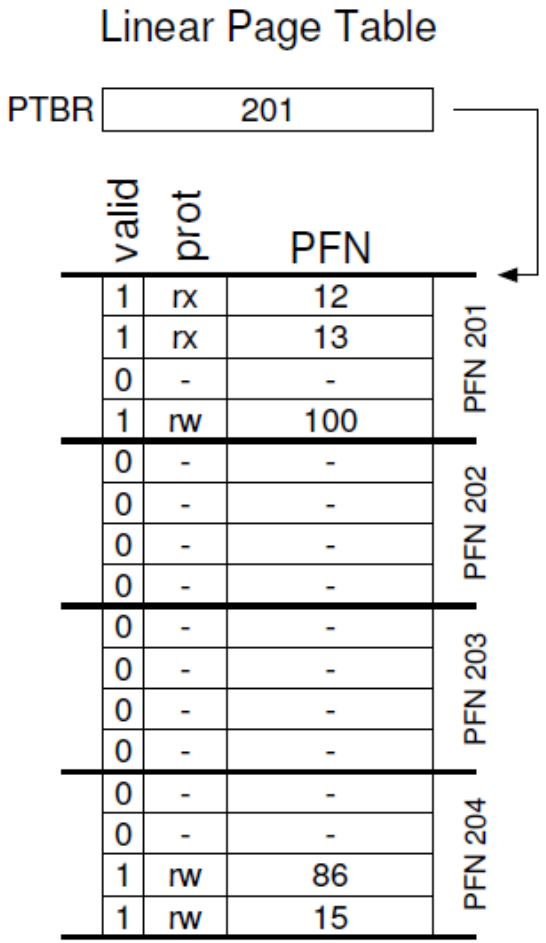
- Maintain a page table for each segment
- **Segment is contiguously allocated**, so its page table is **compact**.
- Problems:
  - Segmentation is not flexible. It restricts a process' memory layout, which is against the idea of virtualization
  - Page table becomes variable-sized, which is hard to maintain.

Solution 3: Multi-level page tables

- Multi-level indexing is an important idea. It has been used in many areas in computer science, including designing page tables.

- Basic idea: Apply the paging idea to the page table
  - **Chop** a page table into page-sized units
  - If an entire unit only contains **invalid entries**, then **don't allocate** the unit
  - Use a page directory to store information of each unit
    - If the unit is not allocated, the page directory stores a bit "invalid".
    - If the unit is allocated, the page directory stores the address of the unit.

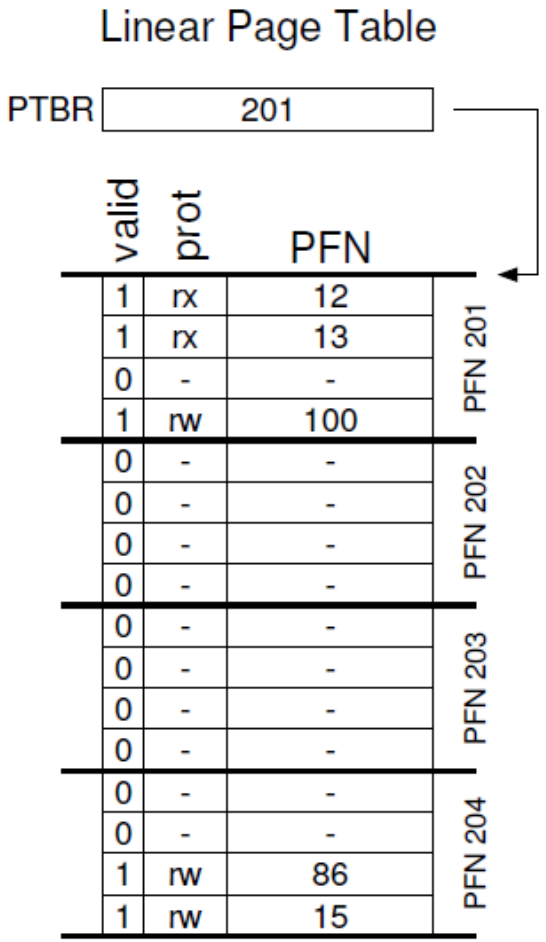
Example of multi-level page tables

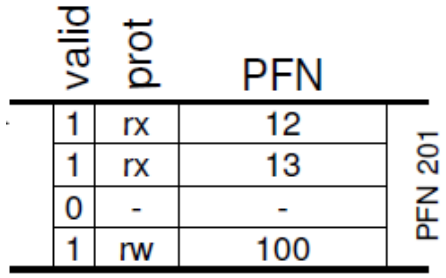


PTBR = Page Table Base Register

Chop the page table into 4 page-sized units

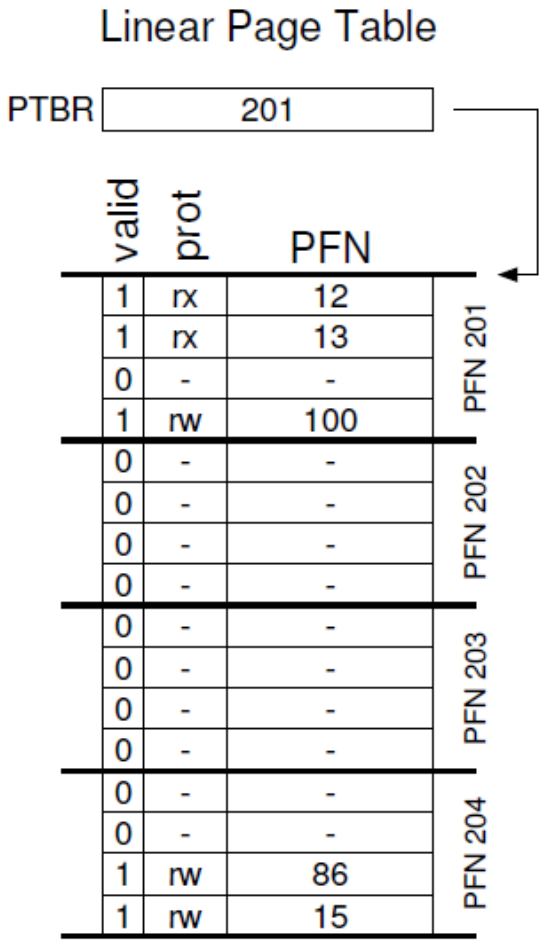
Example of multi-level page tables

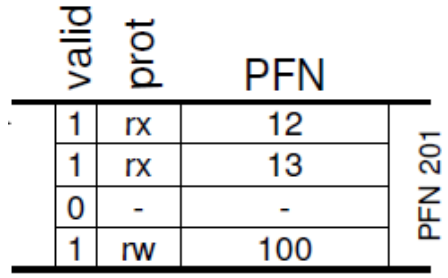




The first unit contains some valid entries, so allocate and store the entire unit

Example of multi-level page tables

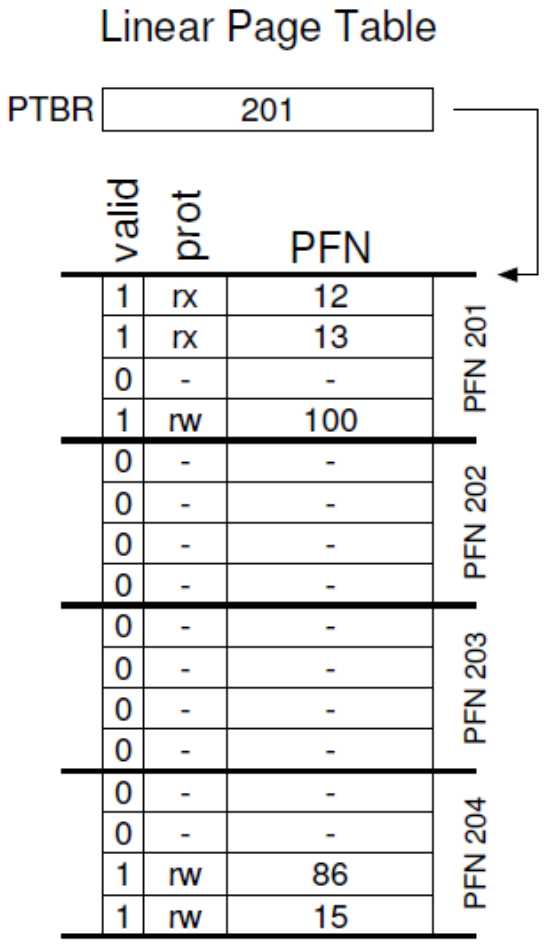


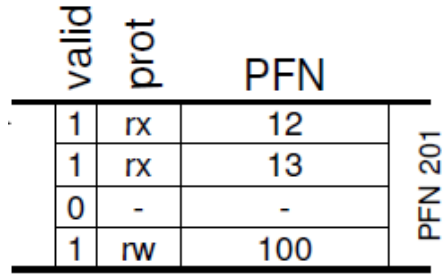


[Page 1 of PT: Not Allocated]

The second unit contains only invalid entries, so don't allocate the unit

Example of multi-level page tables



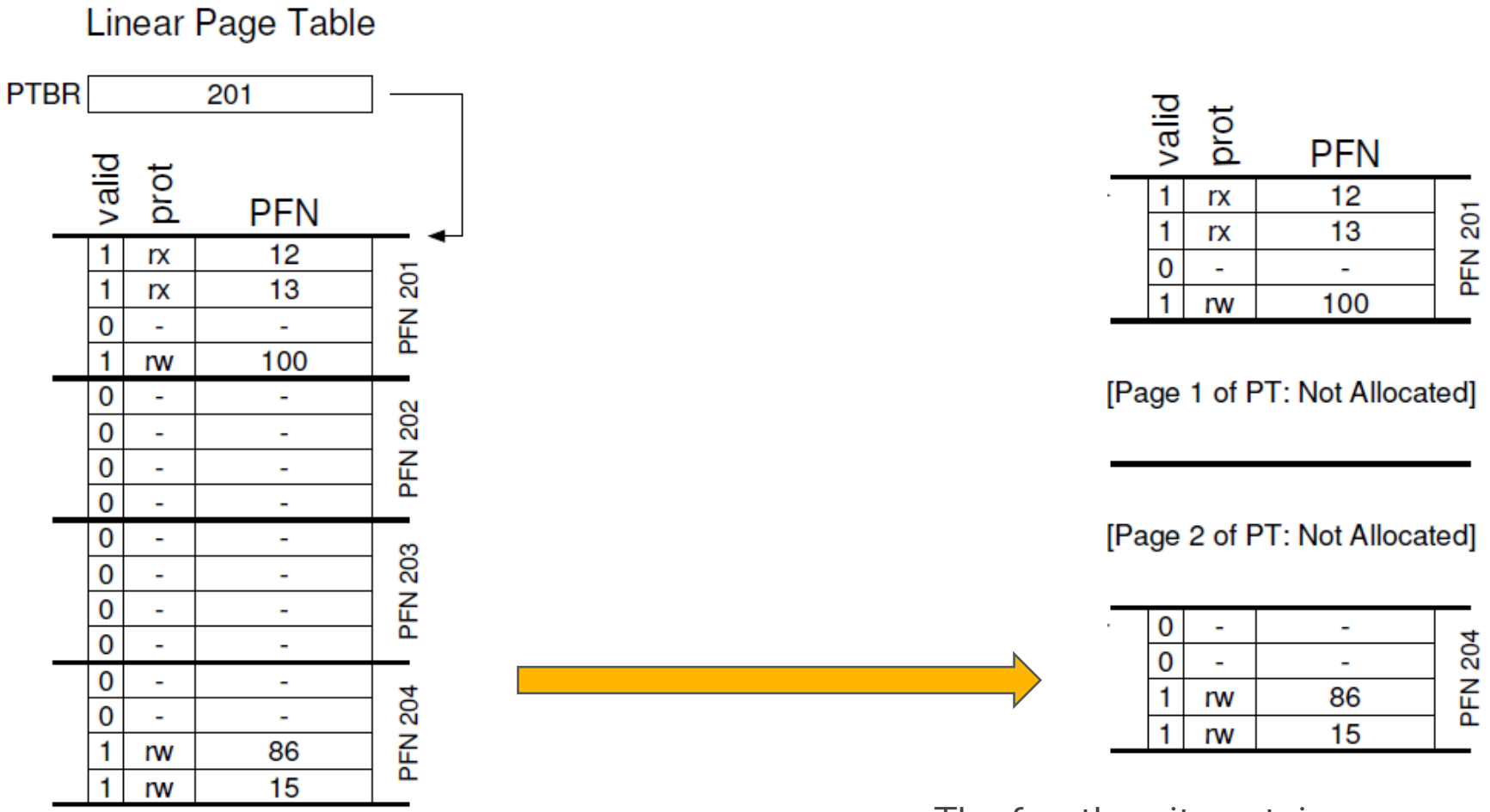


[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

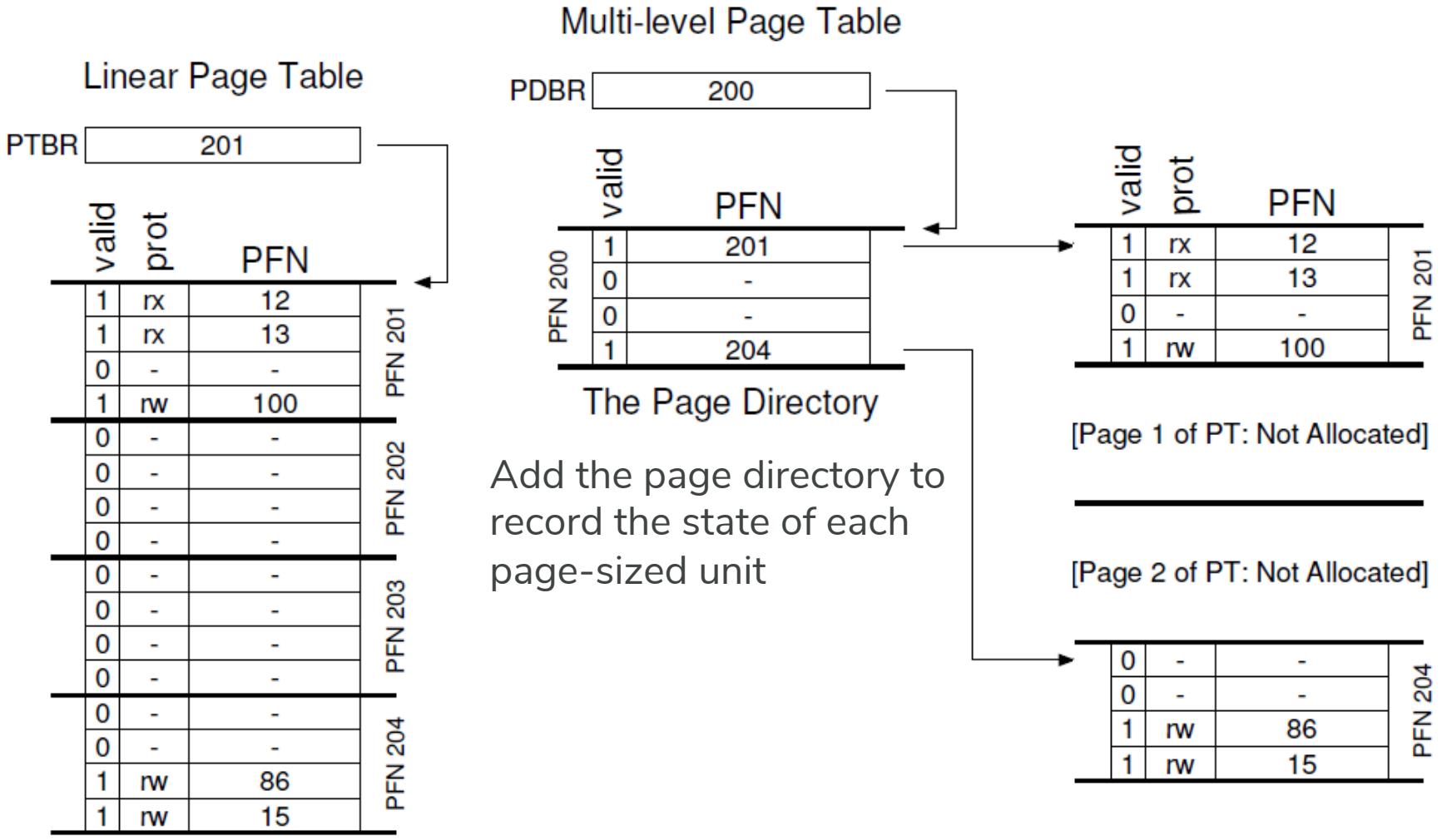
The third unit contains only invalid entries, so don't allocate the unit

Example of multi-level page tables

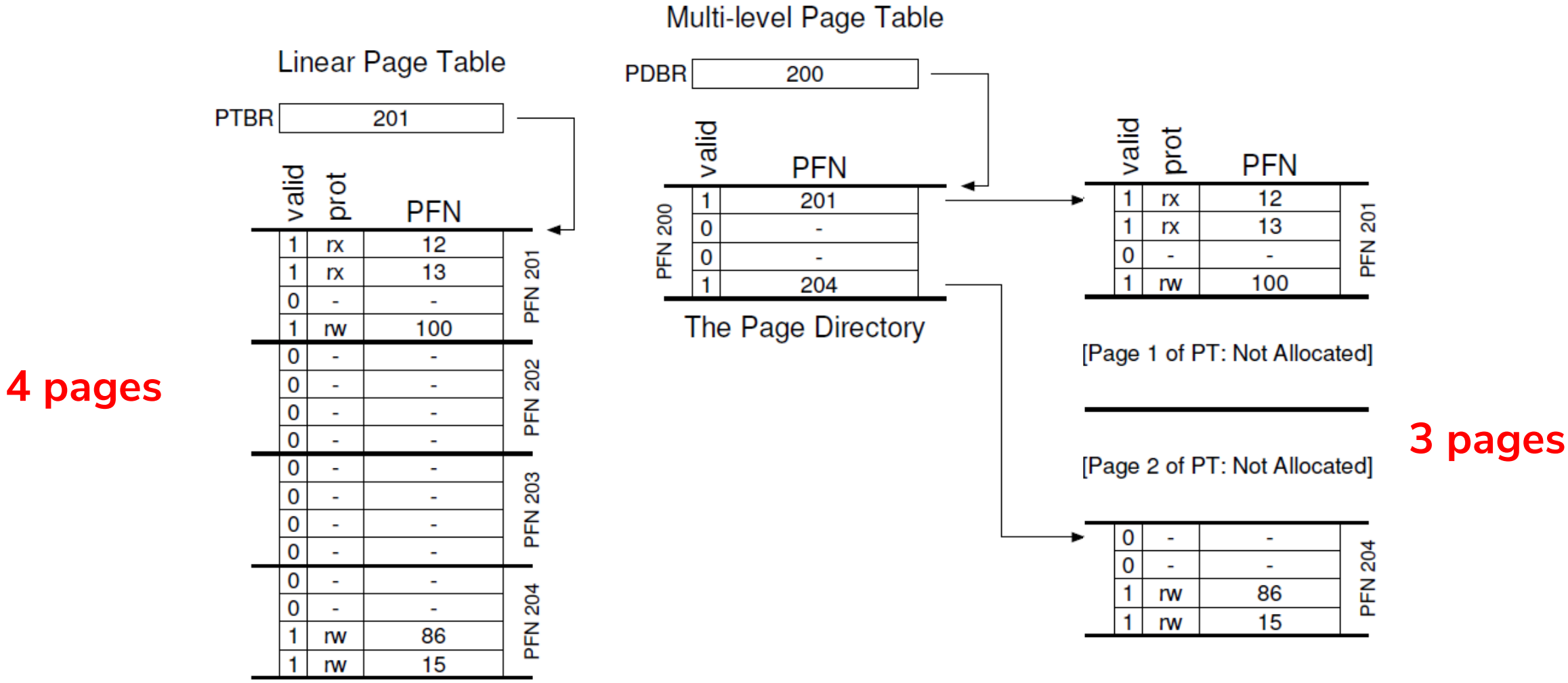


The fourth unit contains some valid entries, so allocate and store the entire unit

Example of multi-level page tables



Example of multi-level page tables



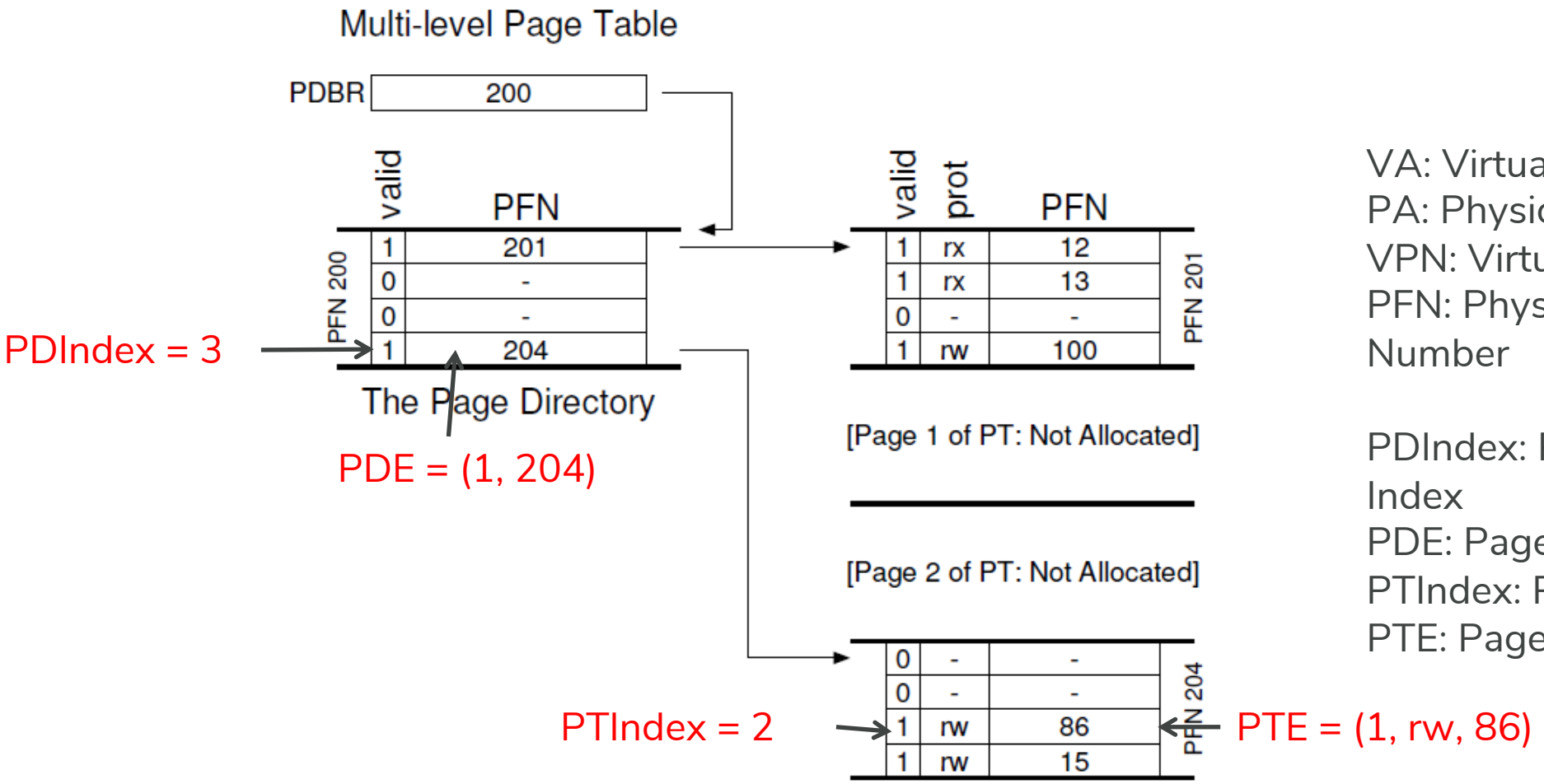
If most entries are invalid, multi-level page table can save a lot of space

Multi-level Page Table: Pros and Cons

- Pros:
  - Save space when most page table entries are invalid
  - Easier to allocate page table in memory (instead of allocating 4MB contiguous space for a page table, now we can allocate separate pages)

- Cons:
  - Now a page table search can take two memory accesses
  - Searching is obviously more complex

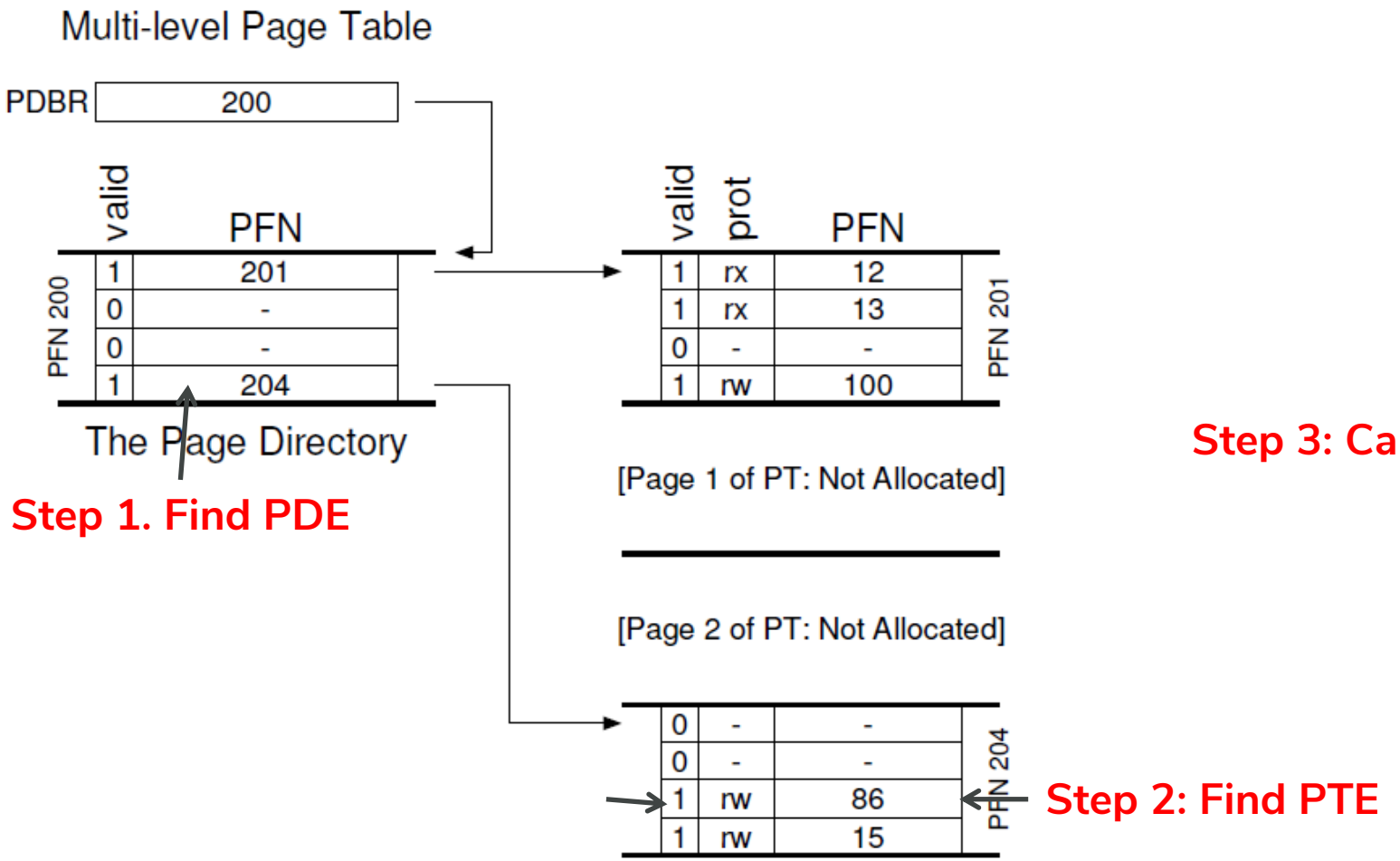
Implementing page table search



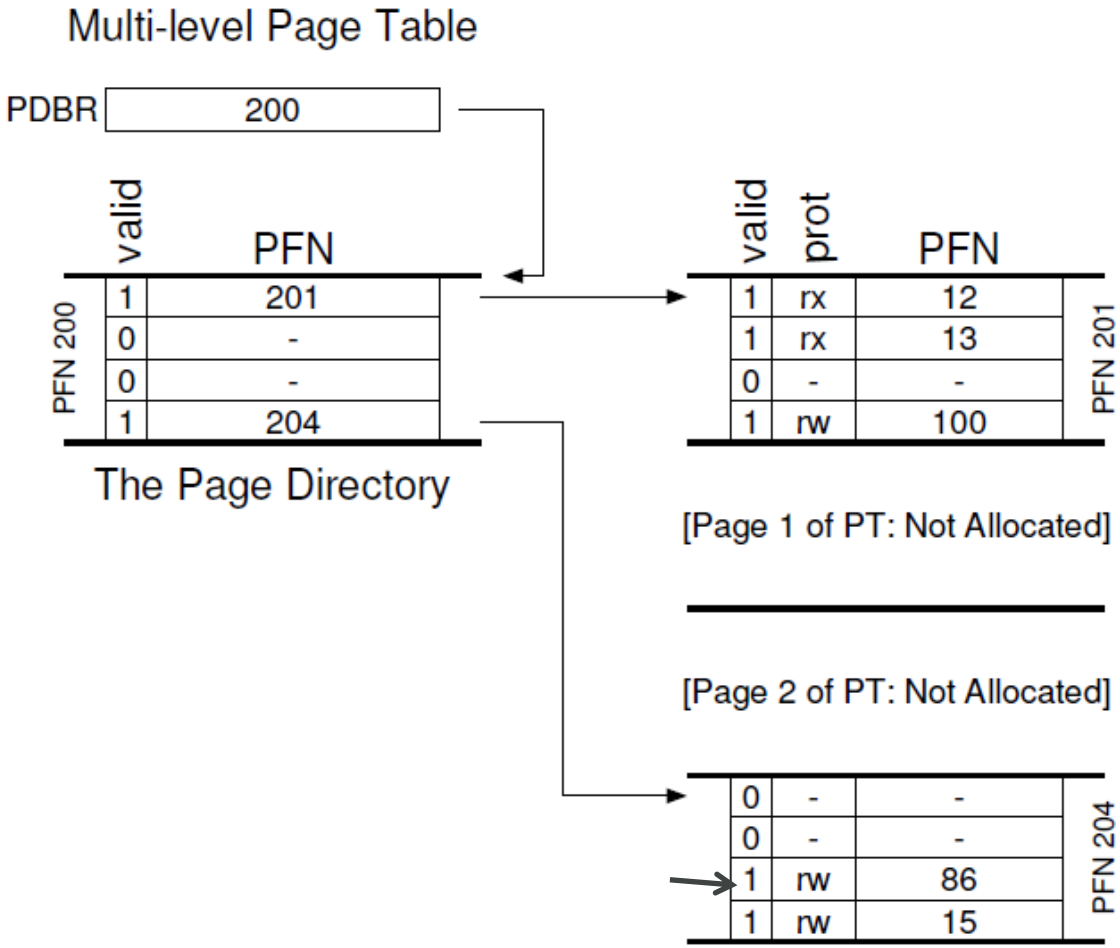
VA: Virtual address
PA: Physical address
VPN: Virtual Page Number
PFN: Physical Frame Number

PDIndex: Page Directory Index
PDE: Page Directory Entry
PTIndex: Page Table Index
PTE: Page Table Entry

Implementing page table search

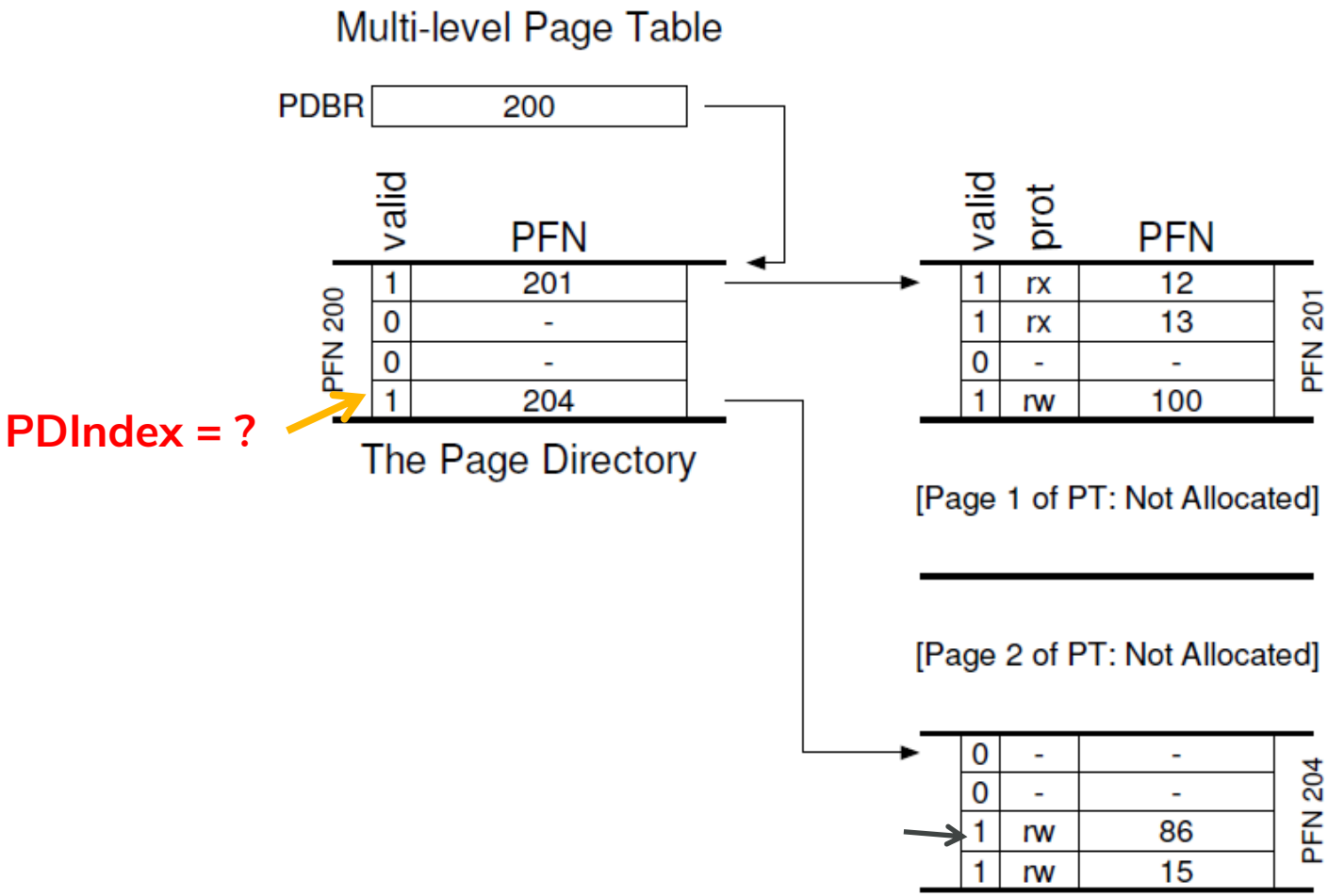


Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

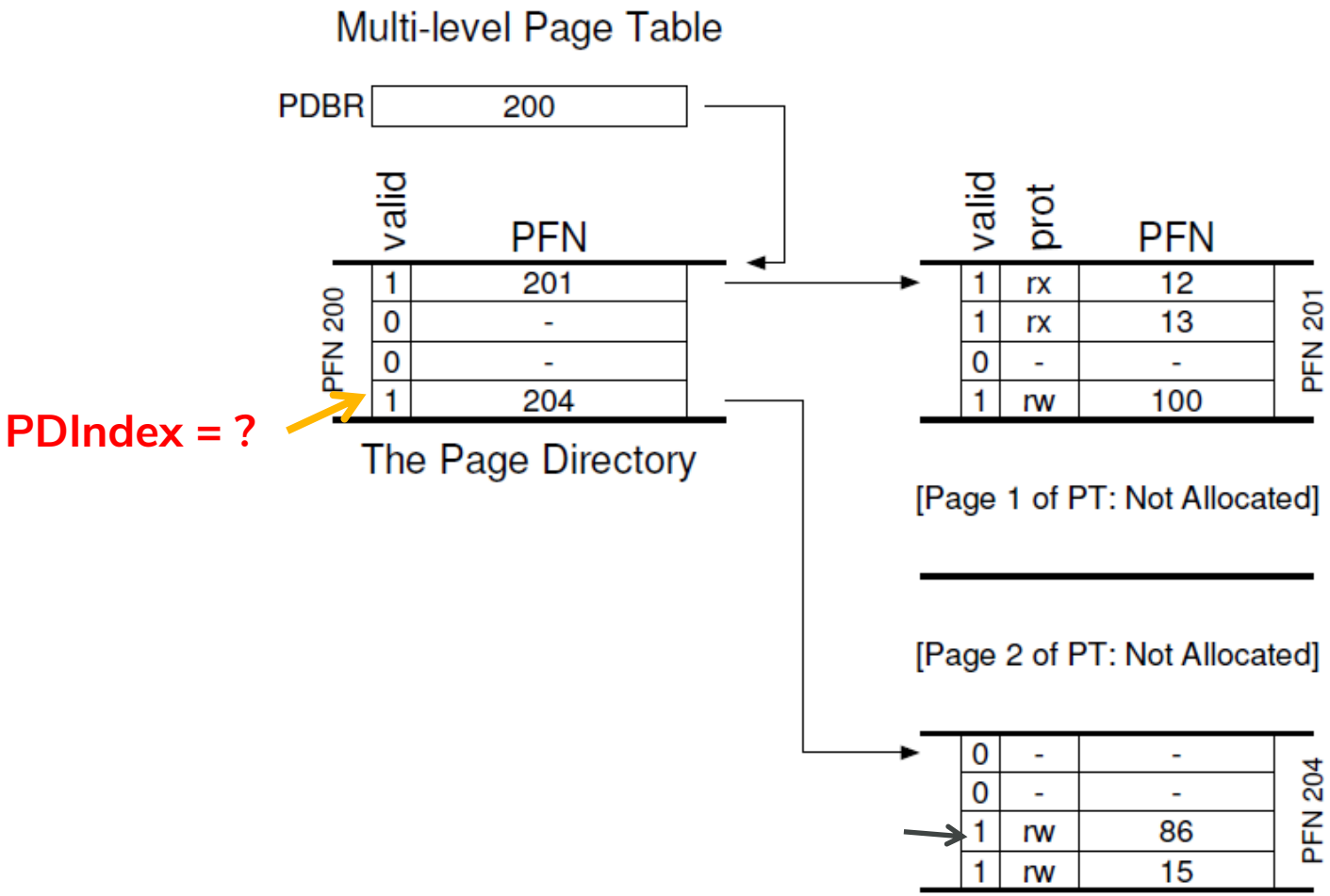
Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

VPN = VA / N (Same as before)
Offset = VA % N (Same as before)
What is PDIndex?

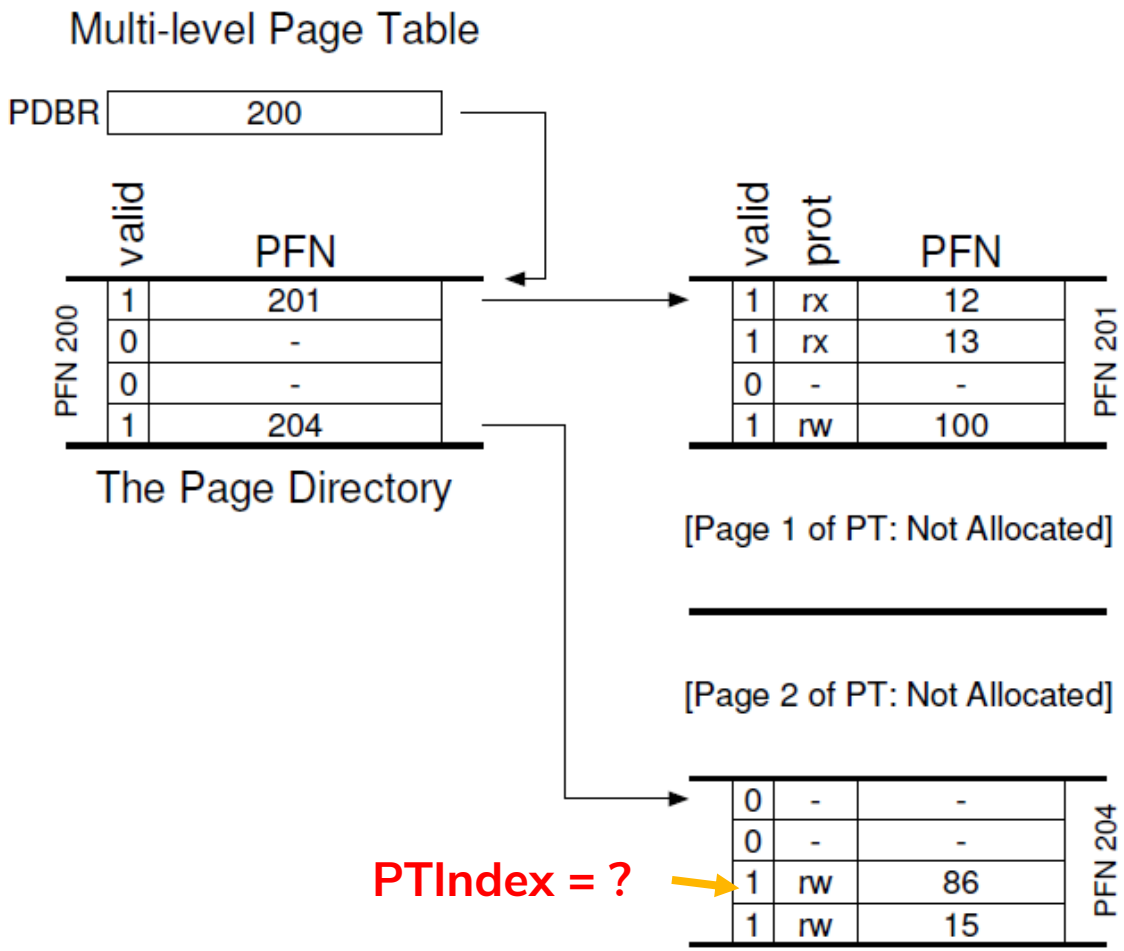
Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

VPN = VA / N (Same as before)
Offset = VA % N (Same as before)
PDIndex = VPN / M
PDE = PDBR * N+ (PDIndex * sizeof(PDE))
If invalid, raise exception
If valid, continue

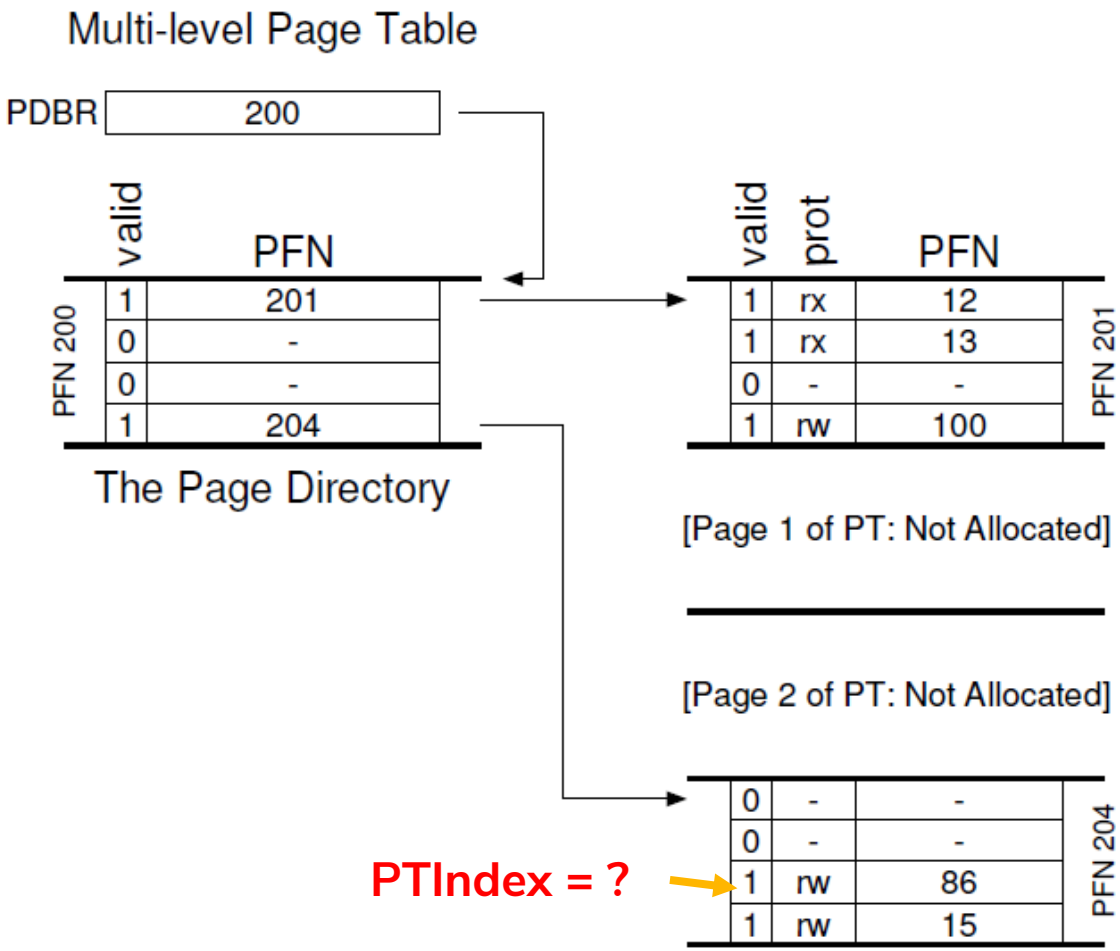
Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

VPN = VA / N (Same as before)
Offset = VA % N (Same as before)
PDIndex = VPN / M
PDE = PDBR * N+ (PDIndex * sizeof(PDE))
What is PTIndex?

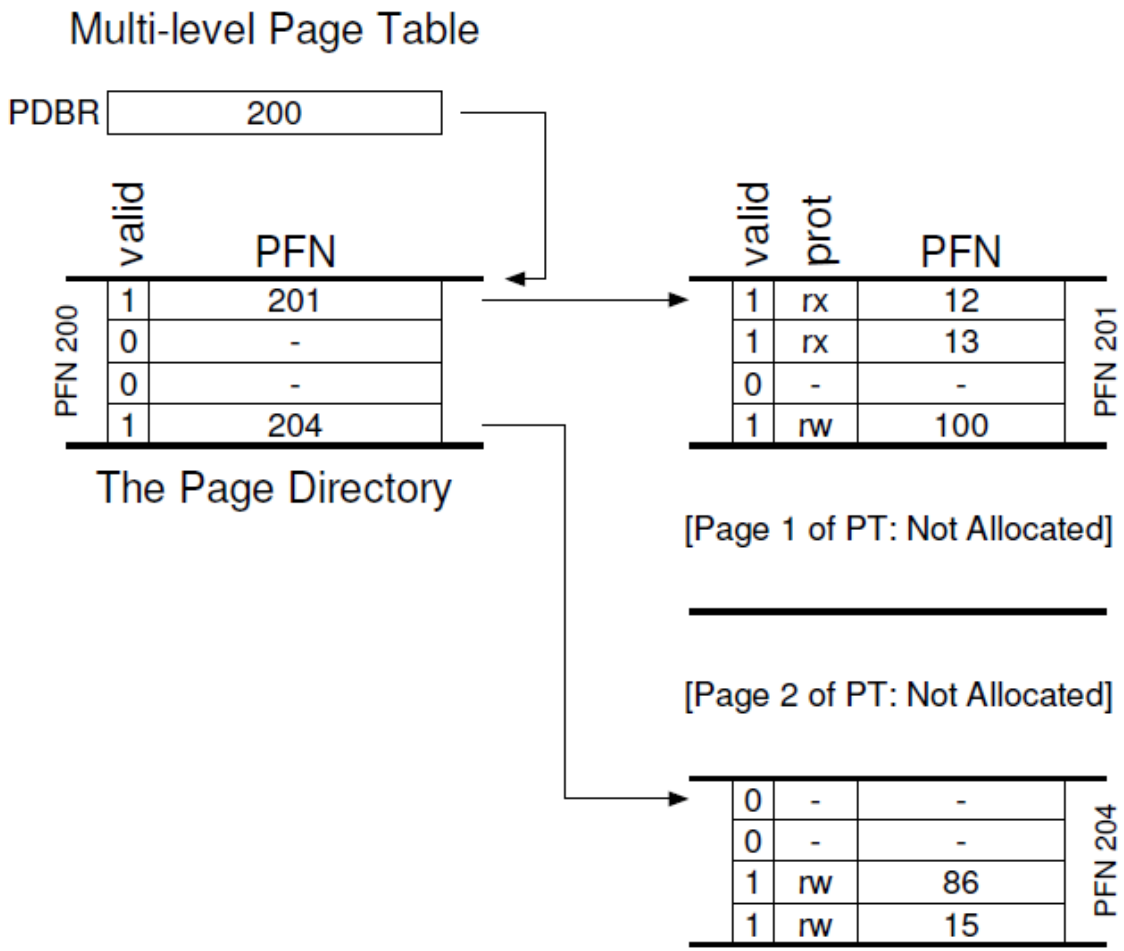
Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

VPN = VA / N (Same as before)
Offset = VA % N (Same as before)
PDIndex = VPN / M
PDE = PDBR * N+ (PDIndex * sizeof(PDE))
PTIndex = VPN % M
PTE = PDE. PFN * N + (PTIndex * sizeof(PTE))

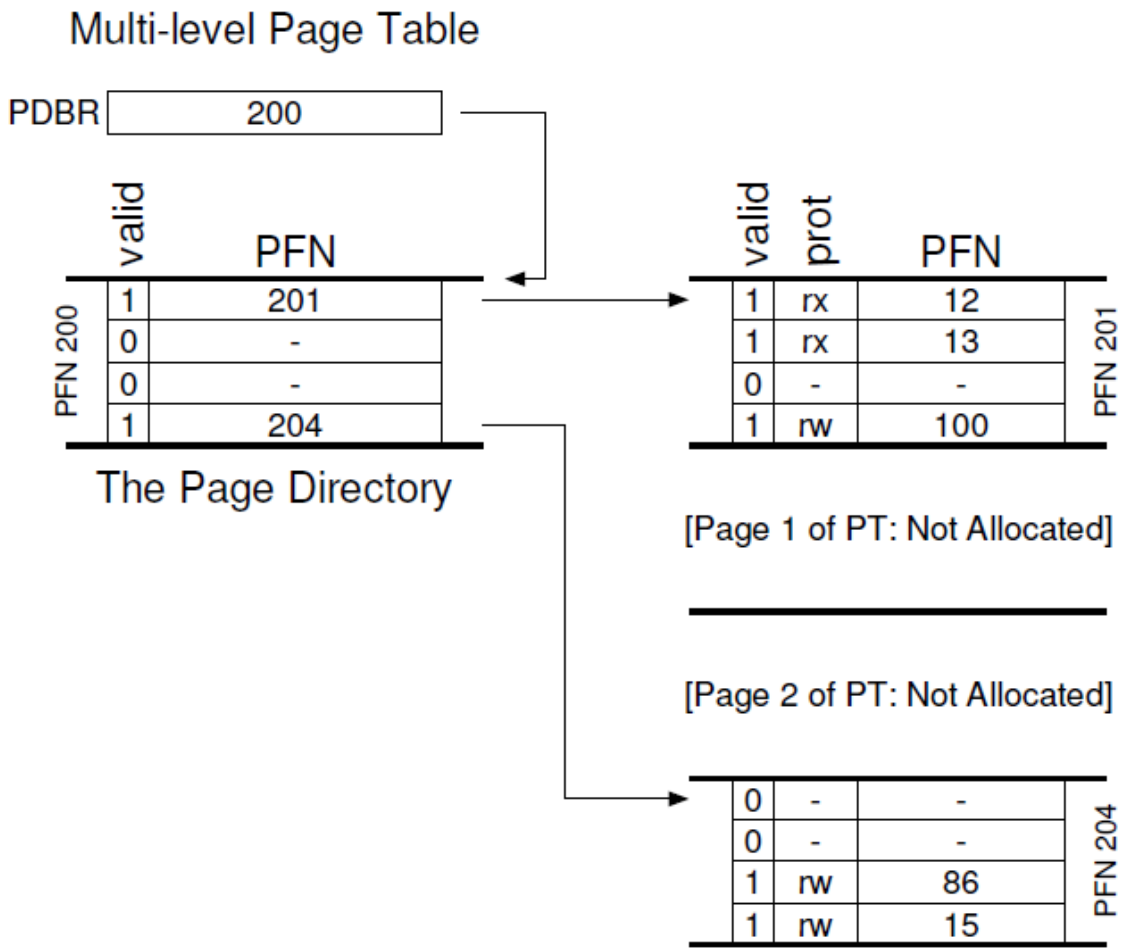
Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

VPN = VA / N (Same as before)
Offset = VA % N (Same as before)
PDIndex = VPN / M
PDE = PDBR * N+ (PDIndex * sizeof(PDE))
PTIndex = VPN % M
PTE = PDE. PFN * N + (PTIndex * sizeof(PTE))
PA = PTE.PFN * N + offset

Implementing page table search



Given VA:
Page Size = N
Each page of page table has M PTEs
Page Directory has D PDEs

VPN = VA / N (Same as before)
Offset = VA % N (Same as before)
PDIndex = VPN / M
PDE = PDBR * N + (PDIndex * sizeof(PDE))
PTIndex = VPN % M
PTE = PDE. PFN * N + (PTIndex * sizeof(PTE))
PA = PTE.PFN * N + offset
Remember we still have TLB
(Read Figure 20.6)

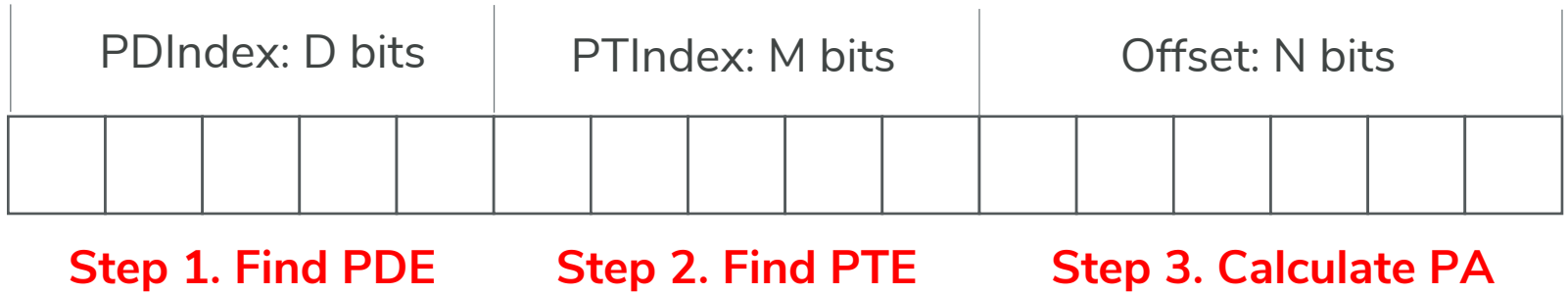
Implementing page table search

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4    if (CanAccess(TlbEntry.ProtectBits) == True)
5      Offset   = VirtualAddress & OFFSET_MASK
6      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7      Register = AccessMemory(PhysAddr)
8    else
9      RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11   // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14   PDE     = AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18     // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN<<SHIFT) + (PTIndex*sizeof(PTE))
21     PTE     = AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

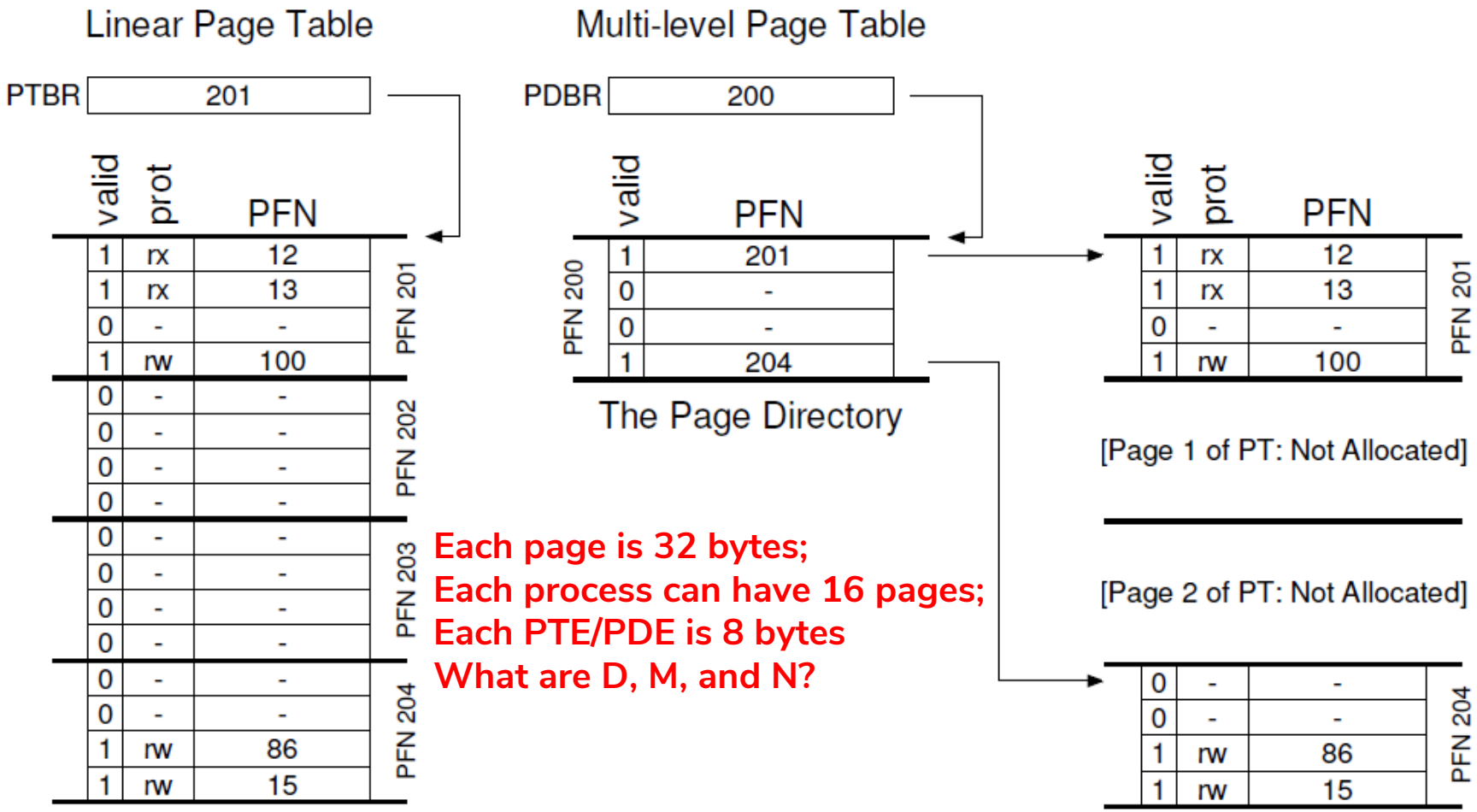
Figure 20.6: **Multi-level Page Table Control Flow**

Solution 3: multi-level page tables

- Once again, if page size, number of PTEs per page, and number of PDEs are all orders of 2 (this is a typical case), calculation can be simplified
  - Page size = 2^N, number of PTEs per page = 2^M, number of PDEs = 2^D.

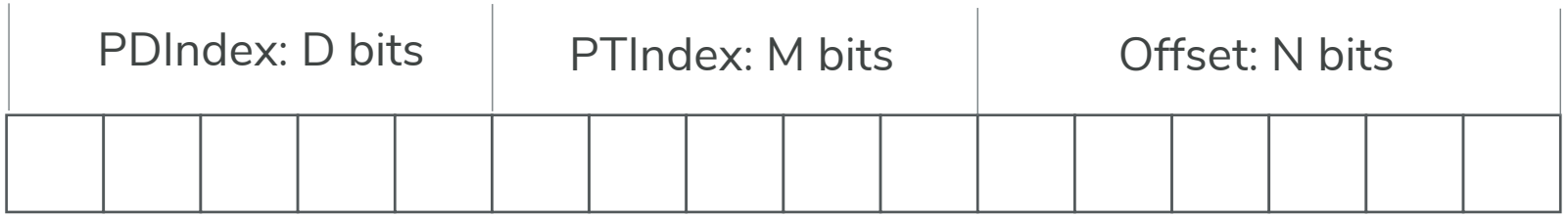


Let’s do an exercise



Let’s do an exercise

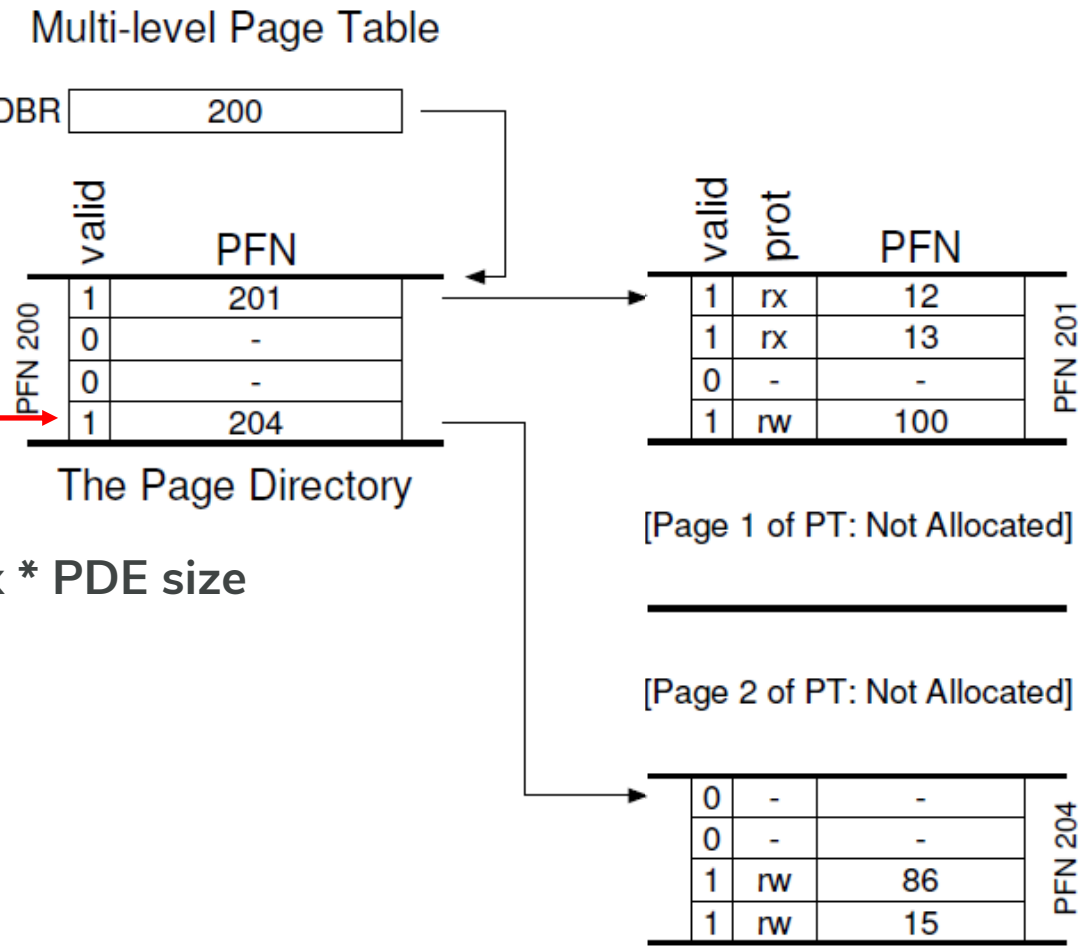
- Page size = 2^N, number of PTEs per page = 2^M, number of PDEs = 2^D.



- Each page is 32 bytes; each process can have 16 pages; each PTE is 8 bytes
  - $32 = 2^5$, so N = 5
  - Each page can have 32/8 = 4 PTEs, so M = 2
  - Number of PDEs = 16 pages in total / 4PTEs per page = 4, so D = 2.
  - What is PA of VA 111001010b?

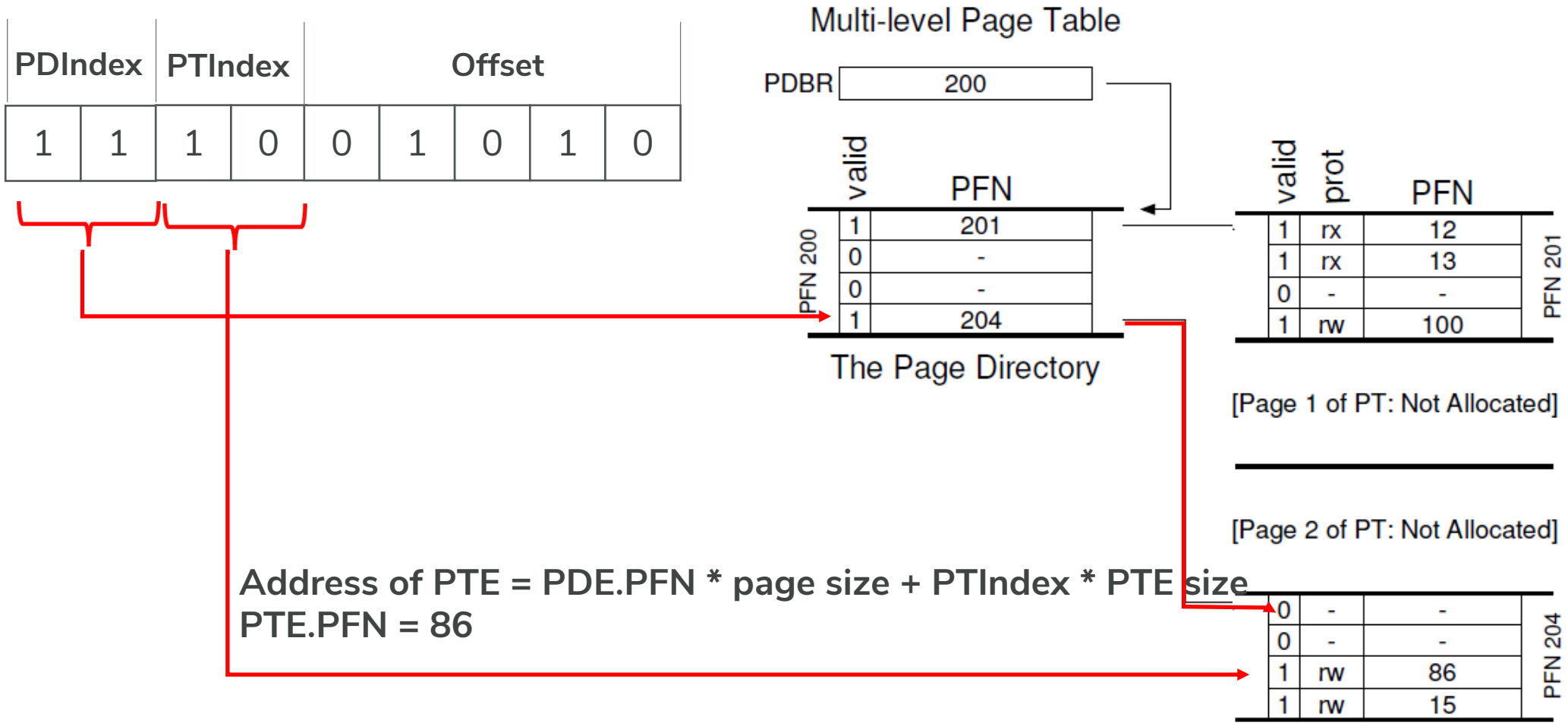
Let’s do an exercise

PDIndex		PTIndex		Offset				
1	1	1	0	0	1	0	1	0



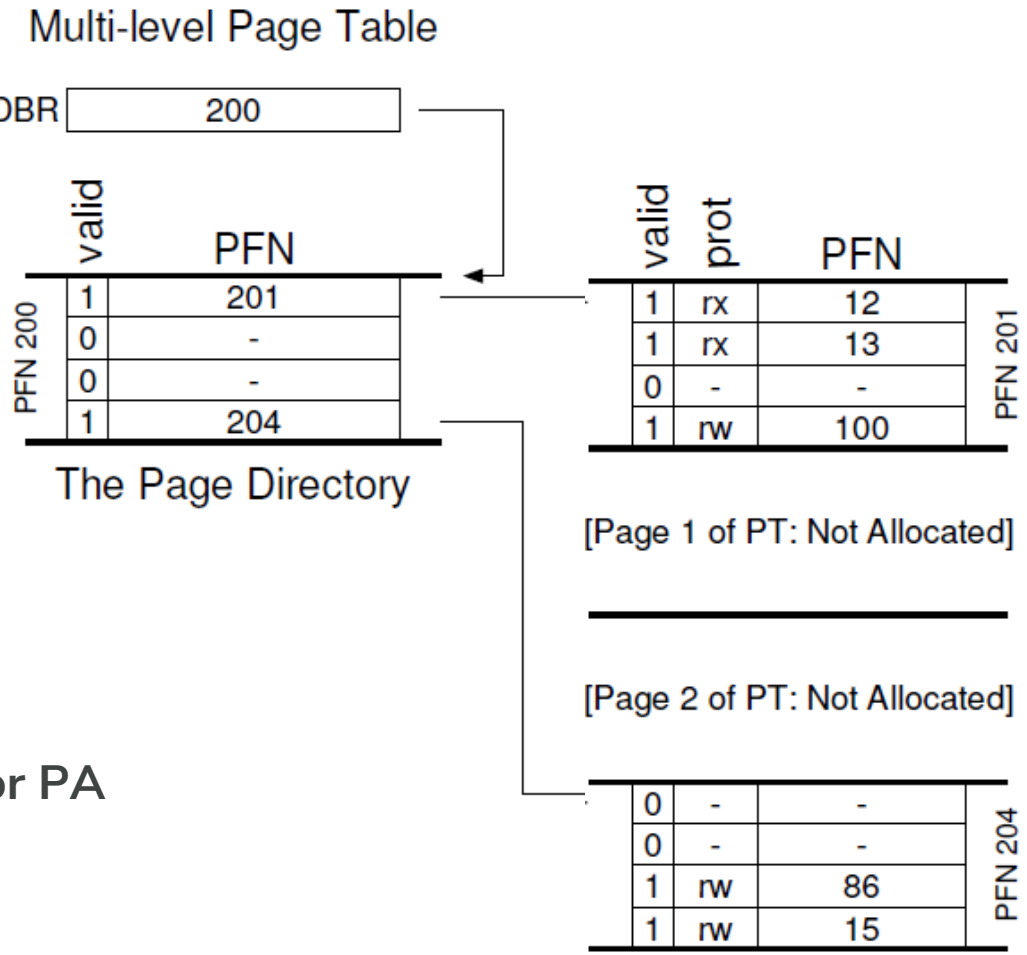
Address of PDE = PDBR * page size + PDIndex * PDE size
PDE.PFN = 204

Let’s do an exercise



Let’s do an exercise

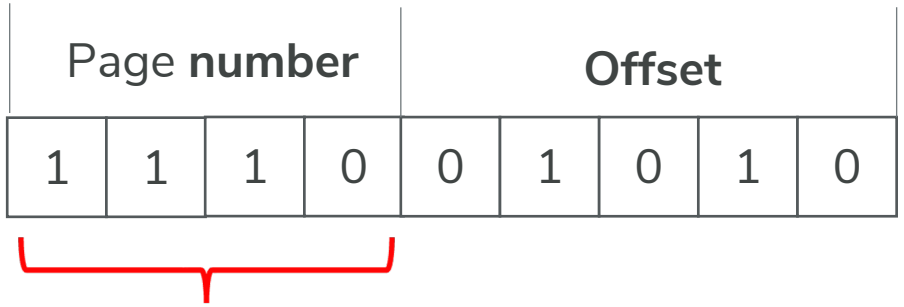
PDIndex		PTIndex		Offset				
1	1	1	0	0	1	0	1	0

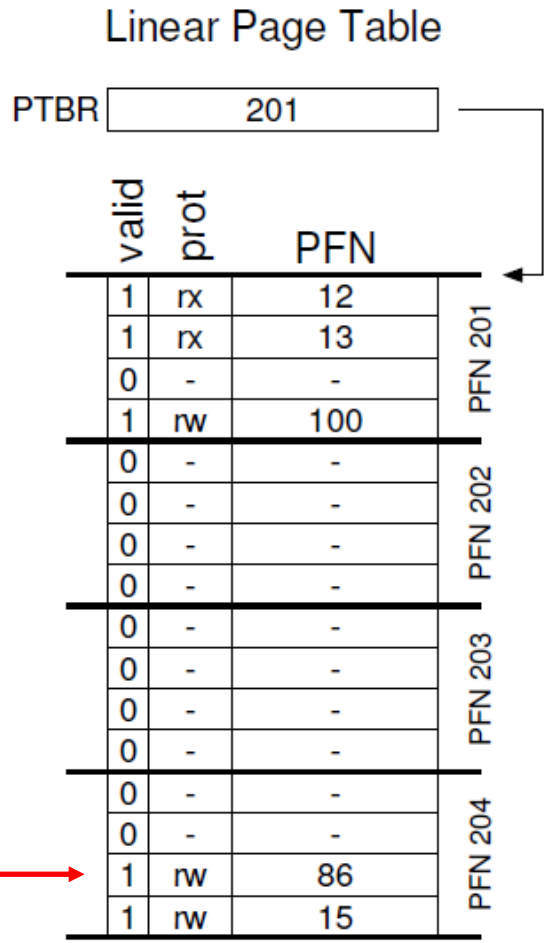


PA = PTE.PFN * page size + offset
= 86 * 32 + 01010b

The result depends on the number of bits for PA
Suppose PA has 13 bits, then
PA = 01010110 01010b

Compare to linear page table





Multi-level page table does not change result.
It's a way to store page table more compactly

PA = PTE.PFN * page size + offset
= 86 * 32 + 01010b

Size of a two-level page table

- What is the min and max size of a page table?
  - Assume 32-bit address space, each page is 4KB, each PTE and PDE is 4 bytes.

- How many pages can a process have?
- How many PTEs per page?
- How many second-level pages?
- How many PDEs?
- What is the size of page directory?

Size of a two-level page table

- What is the min and max size of a page table?
  - Assume 32-bit address space, each page is 4KB, each PTE and PDE is 4 bytes.
- Min size = 4KB (only page directory. All pages are invalid)
- Max size = 4KB + 2^{10} * 4KB = 4KB + 4MB (All pages are valid)
  - This is the page directory size + size of single-array solution

How about 64-bit address space?

- Now we are in 64-bit world.
- What is the min and max size of a page table?
  - Assume 64-bit address space, each page is 4KB, each PTE and PDE is 4 bytes.

- How many pages can a process have?
- How many PTEs per page?
- How many second-level pages?
- How many PDEs?
- What is the size of page directory?

Support larger space

- Solution 1: three-level page tables (and maybe more)
  - Pros: save space
  - Cons: even more memory accesses when a TLB miss occurs

- Solution 2: inverted page table
  - Basic idea: instead of maintaining an entry for each page, we can maintain an entry for each frame
  - Pros: size is always determined by physical memory size. We can maintain one table for the whole OS instead of one or each process.
  - Cons: How to do the mapping? Need something like a hashmap, which is usually slower than array based paged tables.

Summary: Paging

- Paging vs. Dynamic Relocation
- Page Table
- TLB
- Multi-level Page Table