DCS216 Operating Systems

Lecture 19
Memory (2)
Segmentation & Paging

May 8th, 2024

Instructor: Xiaoxi Zhang

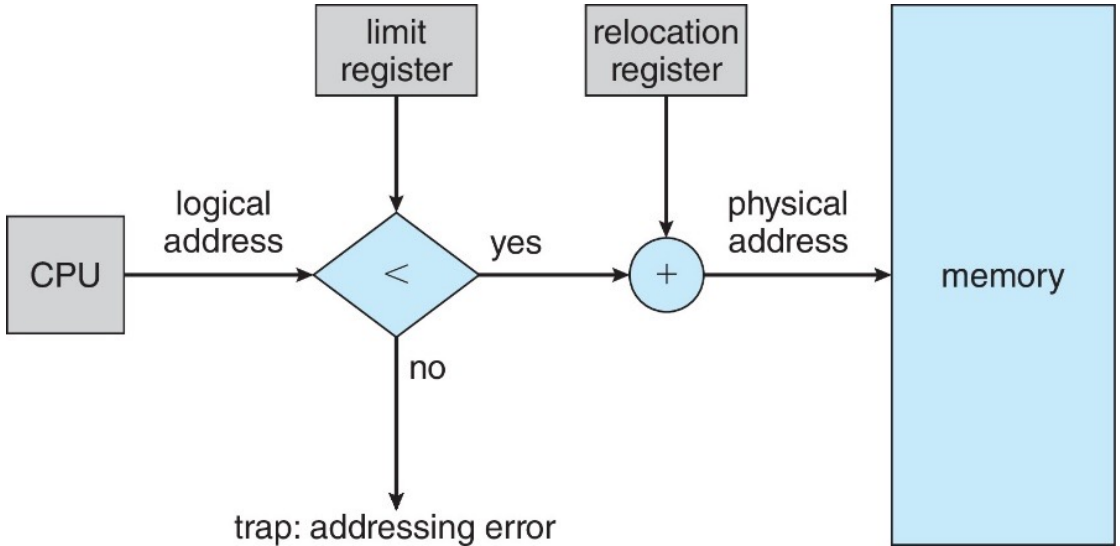
Sun Yat-sen University

Content

- Segmentation (分段)
- Paging (分页)
  - **Page Table** & Address Translation
  - Page Table Entry (**PTE**)
  - **TLB** (**T**ranslation **L**ook-aside **B**uffer)
  - Structure of Page Table
    - **Hierarchical** Page Tables (**Multi-Level** Page Tables)
    - **Hashed** Page Tables
    - **Inverted** Page Tables

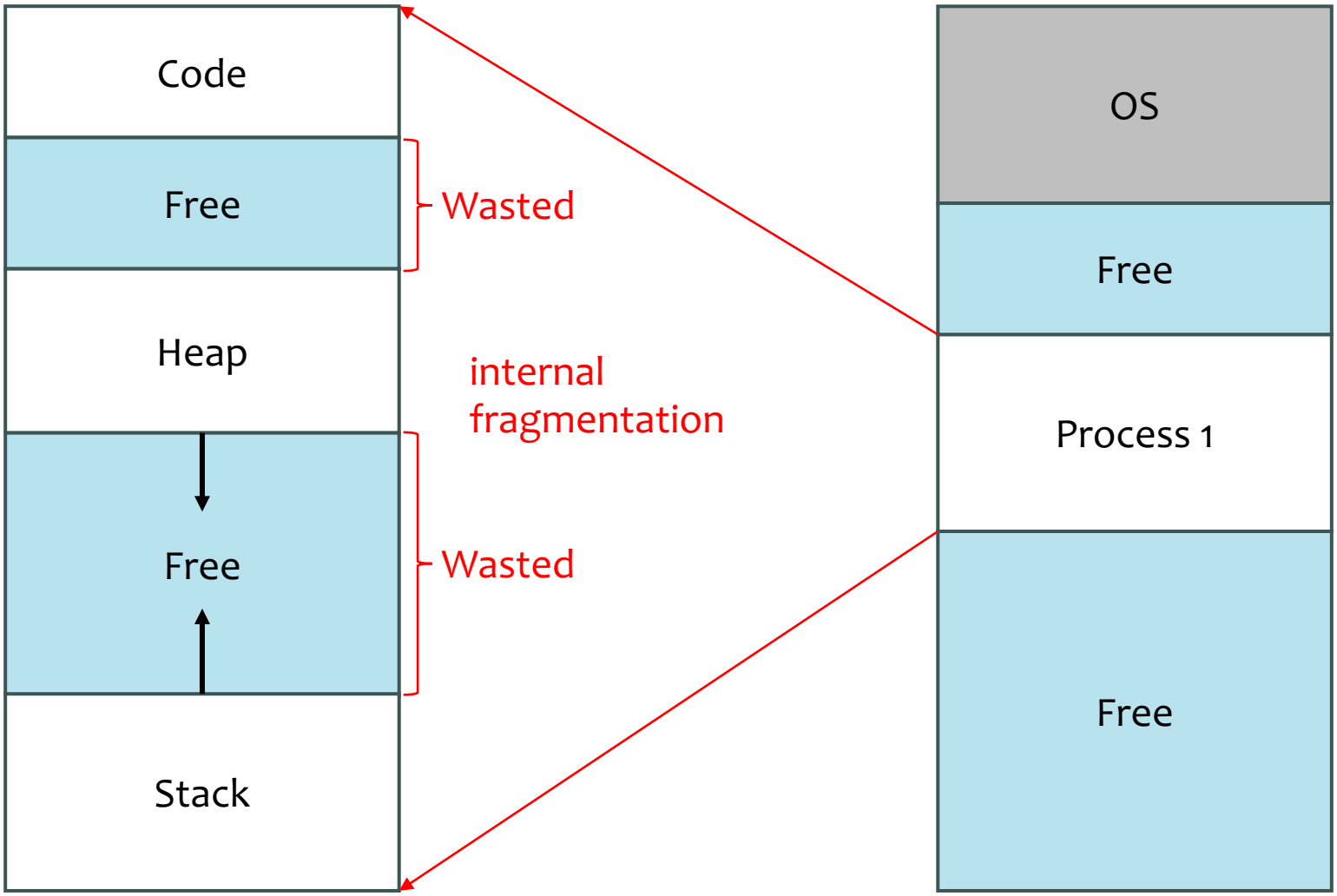
Recall: Simple B&B (Base and Bounds)

- **Issues with simple B&B Method**
  - (*External*) **Fragmentation** problem over time
    - Not every process is the same size ⇒ memory fragmented over time.
  - (*Internal*) **Fragmentation** (inherent)
    - Address space within each process might be wasted
    - Missing support for sparse address space
  - Hard to do inter-process **sharing**
    - Process address space completely isolated
    - No way to share code segments or shared memory



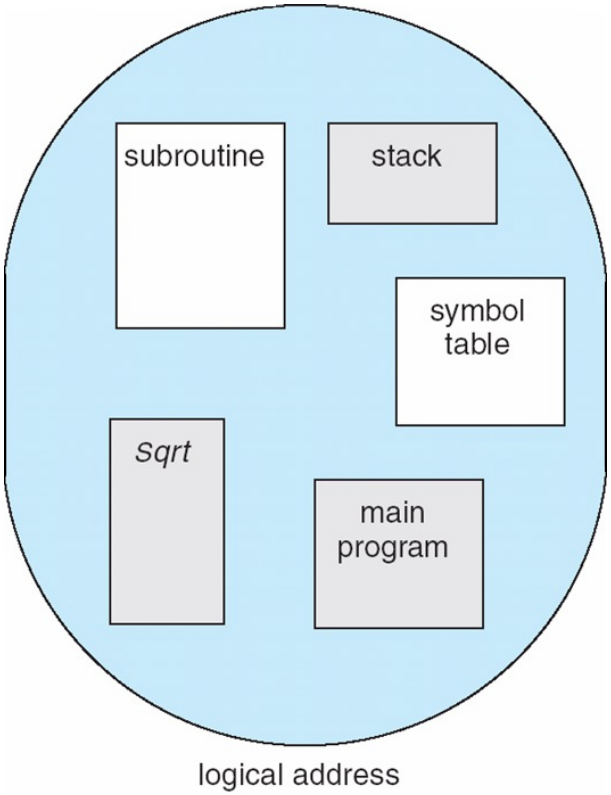
Segmentation

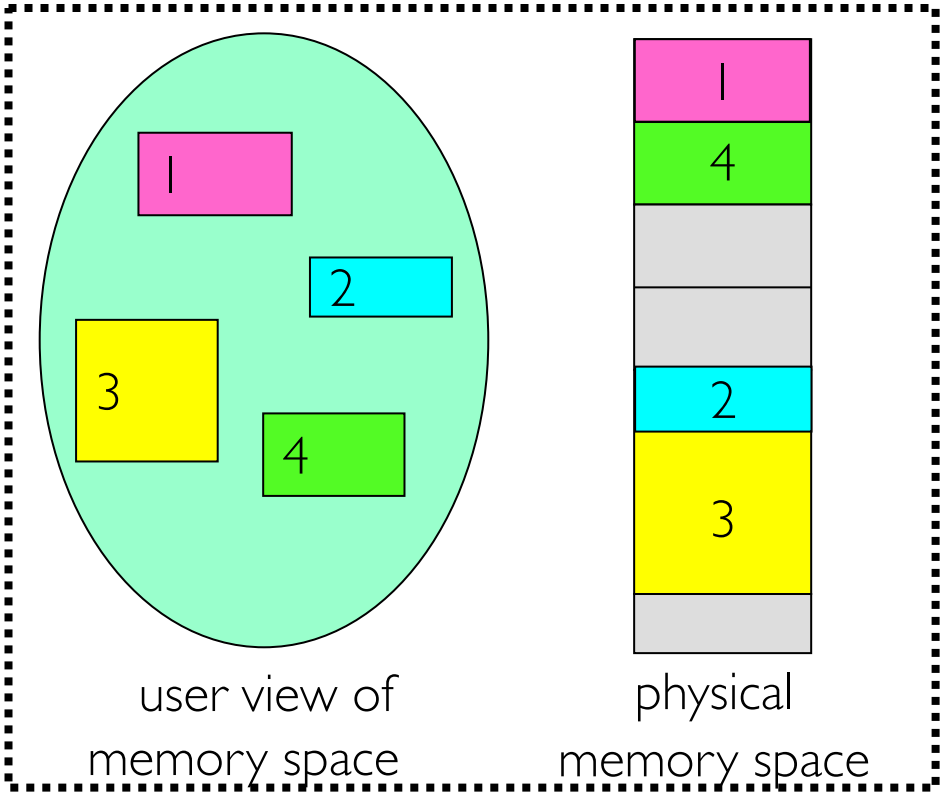
- So far, we have assumed that the **entire address space** of a process is loaded into physical memory.



Segmentation

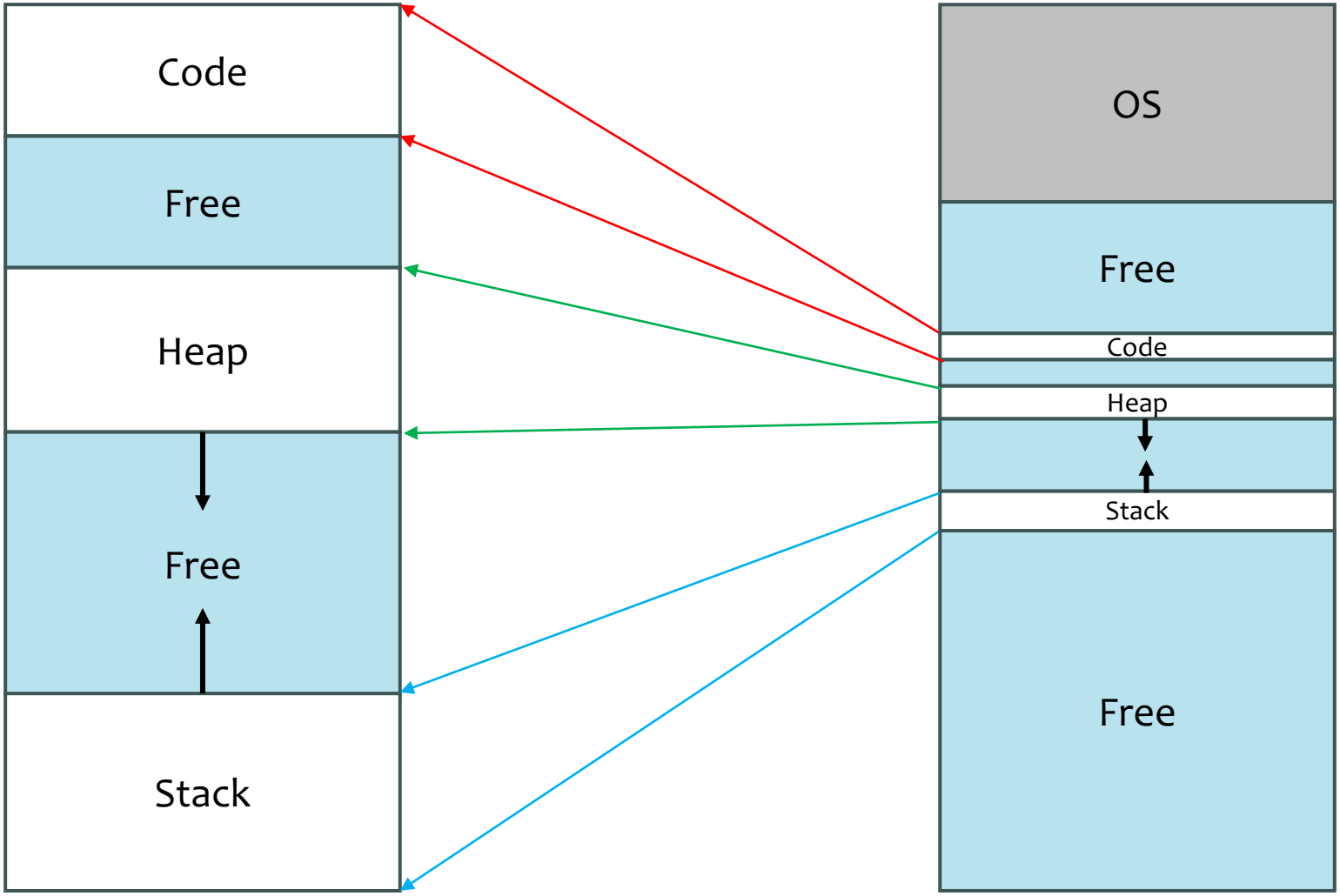
- **The Idea:** instead of having just **ONE base** and **limit** register pair in **MMU**, why not have a (**base** + **limit**) pair **per logical segment**?
  - Logical Segment ⇒ CODE, HEAP, STACK, etc...





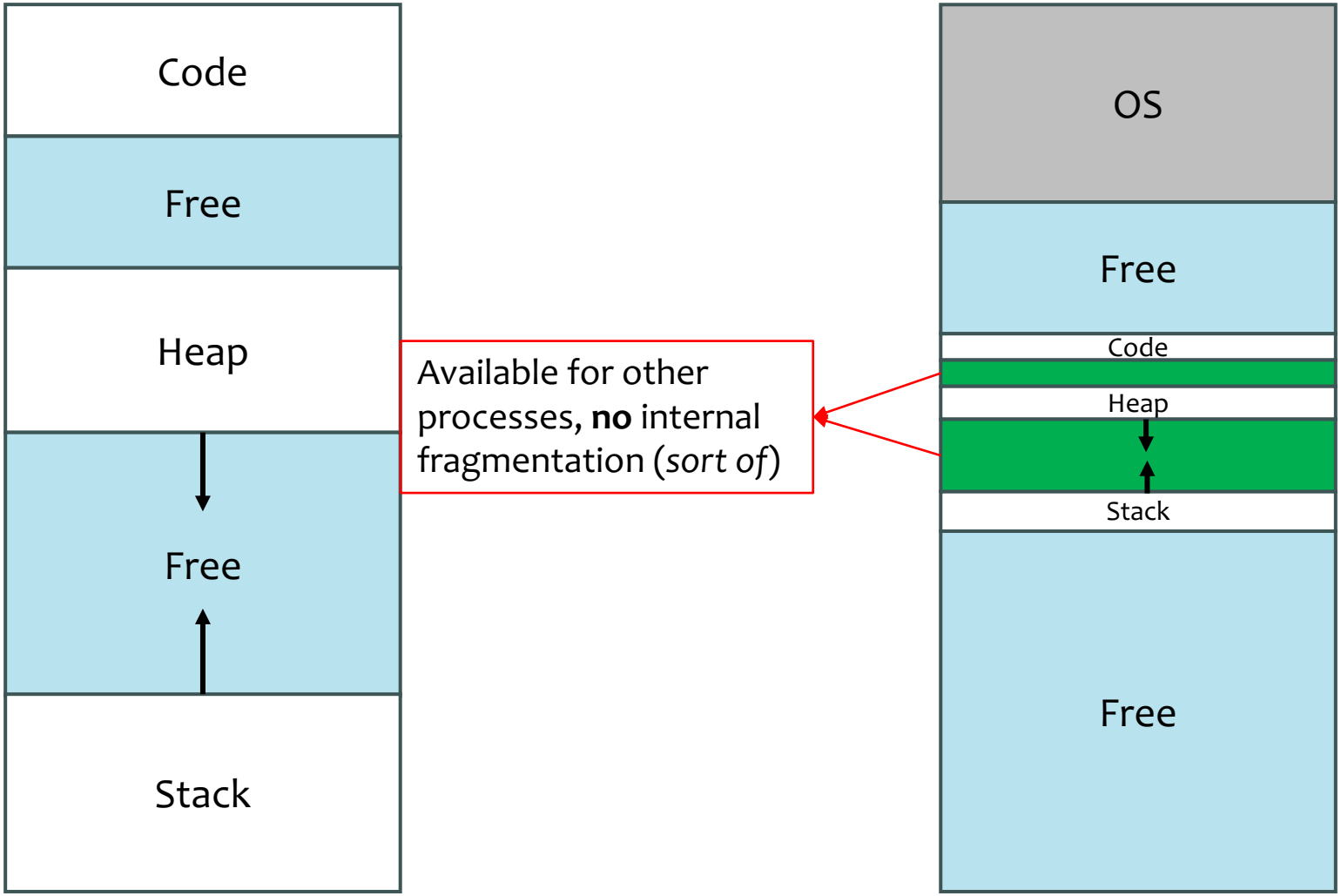
Segmentation

- Each segment is assigned a region of contiguous memory



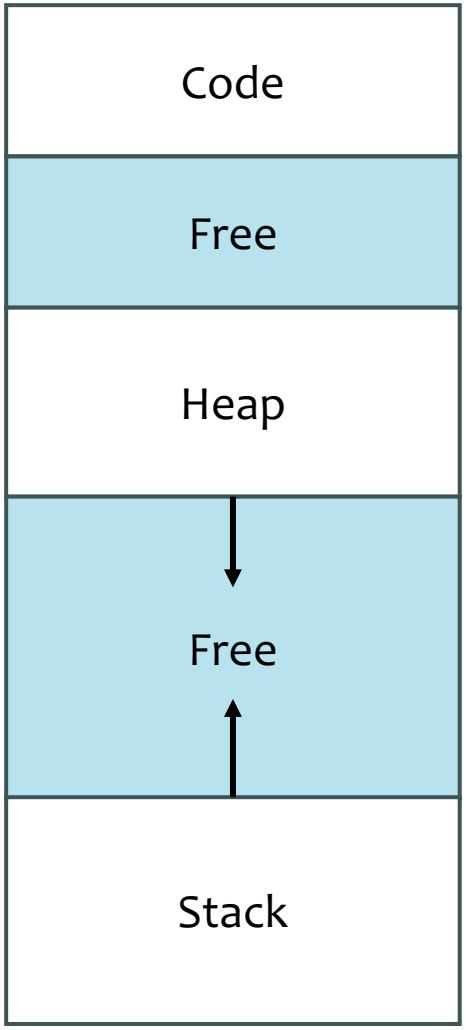
Segmentation

- Each segment is assigned a region of contiguous memory



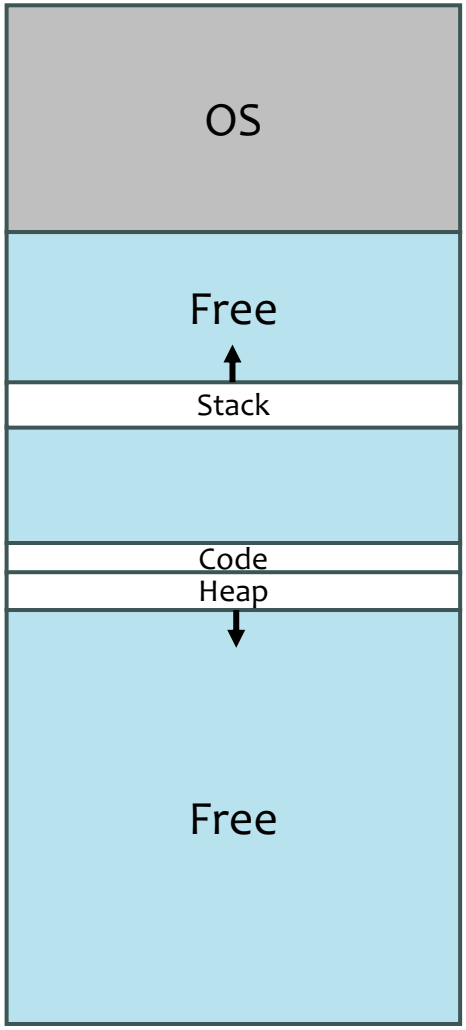
Segmentation

- Can reside anywhere in the physical memory
  - More flexible for **compaction**.



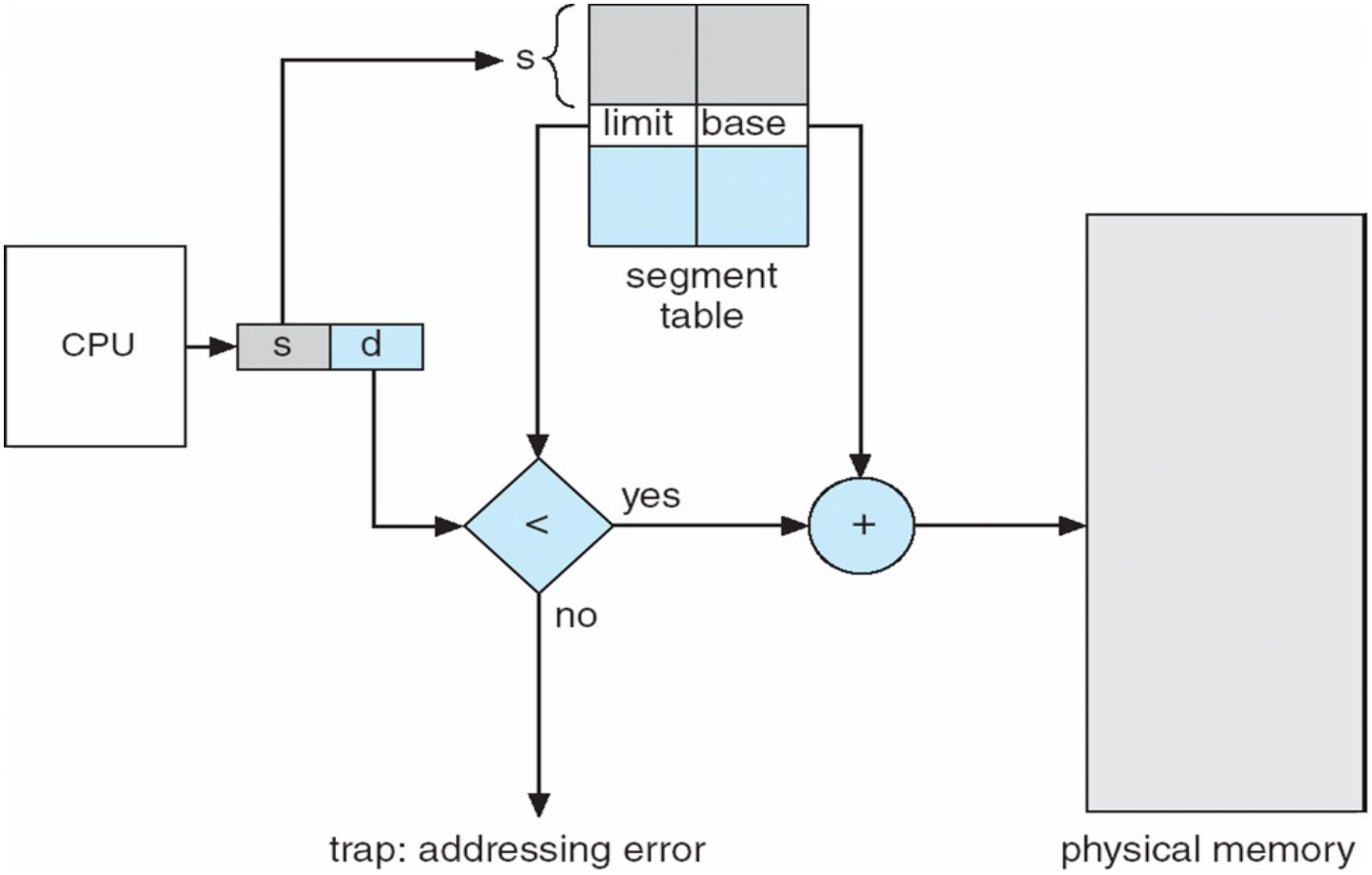
MMU:

Segment	Base	Limit
Code	32000	2000
Heap	34000	2500
Stack	24000	3000



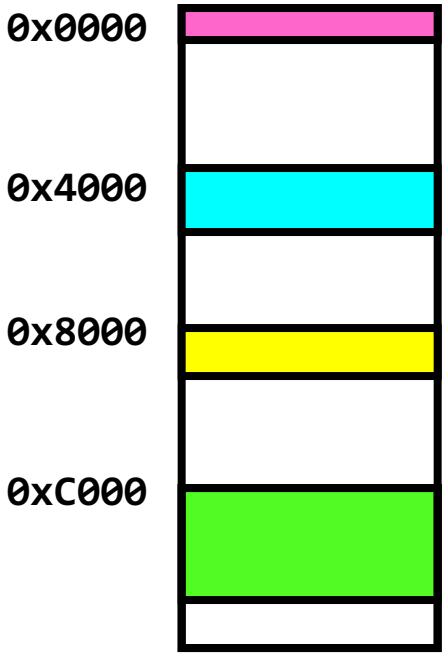
Segmentation Hardware

- Address Translation

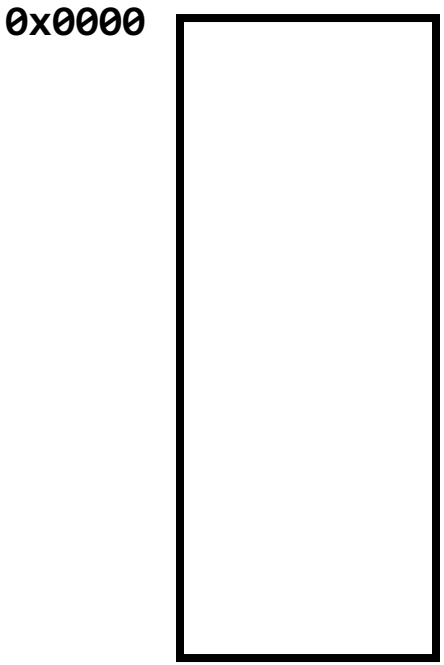


Example: Four Segments (16-bit Address)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Virtual
Address Space

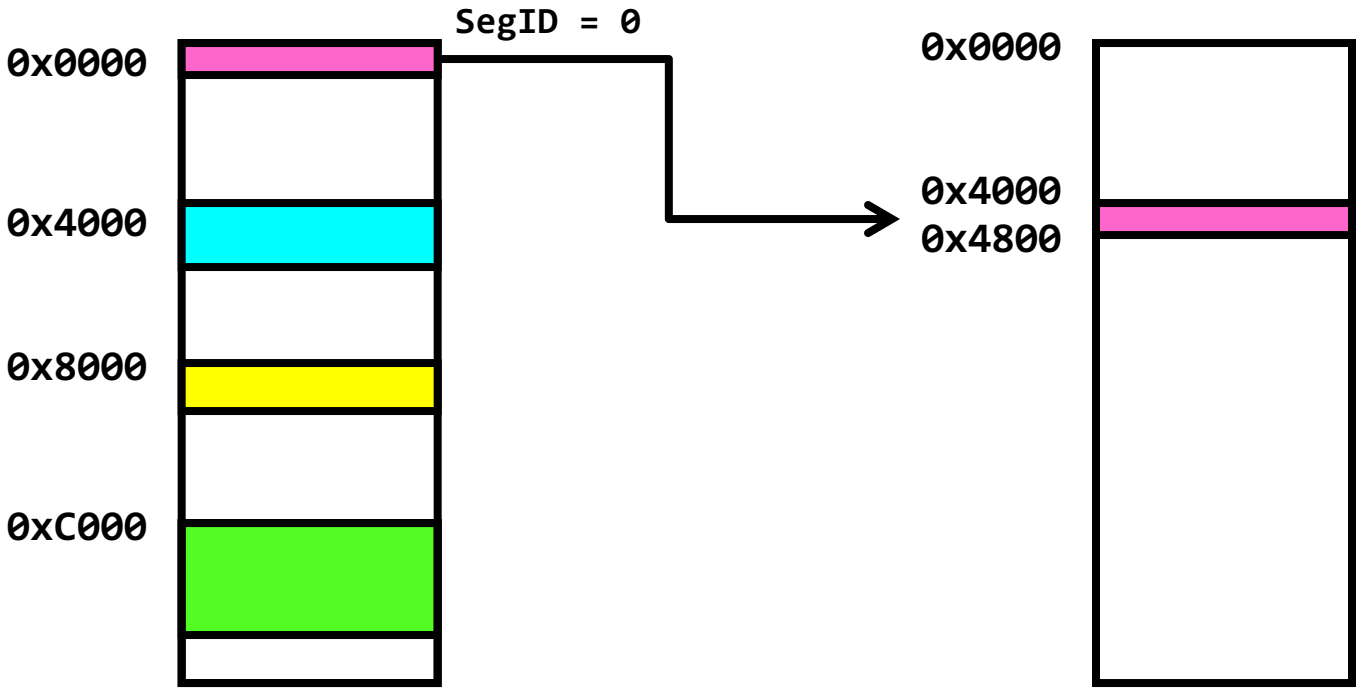


Physical
Address Space

Segmentation

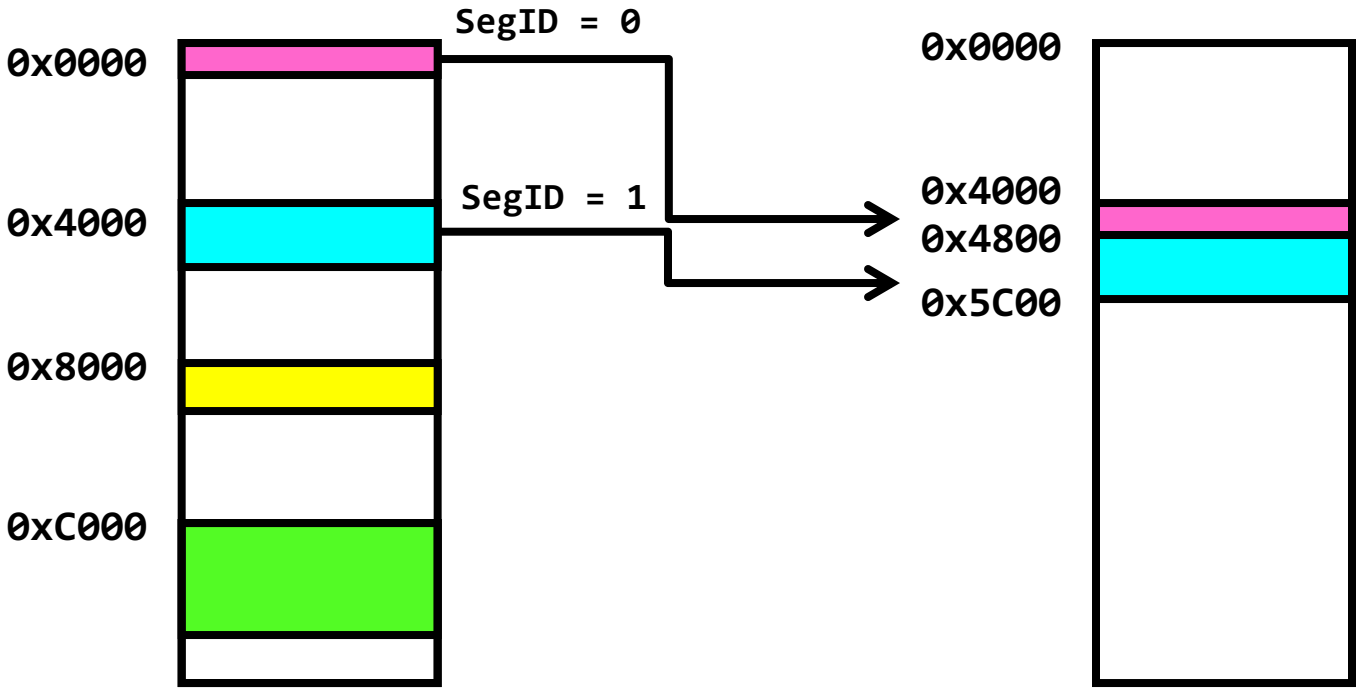
Example: Four Segments (16-bit Address)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Example: Four Segments (16-bit Address)

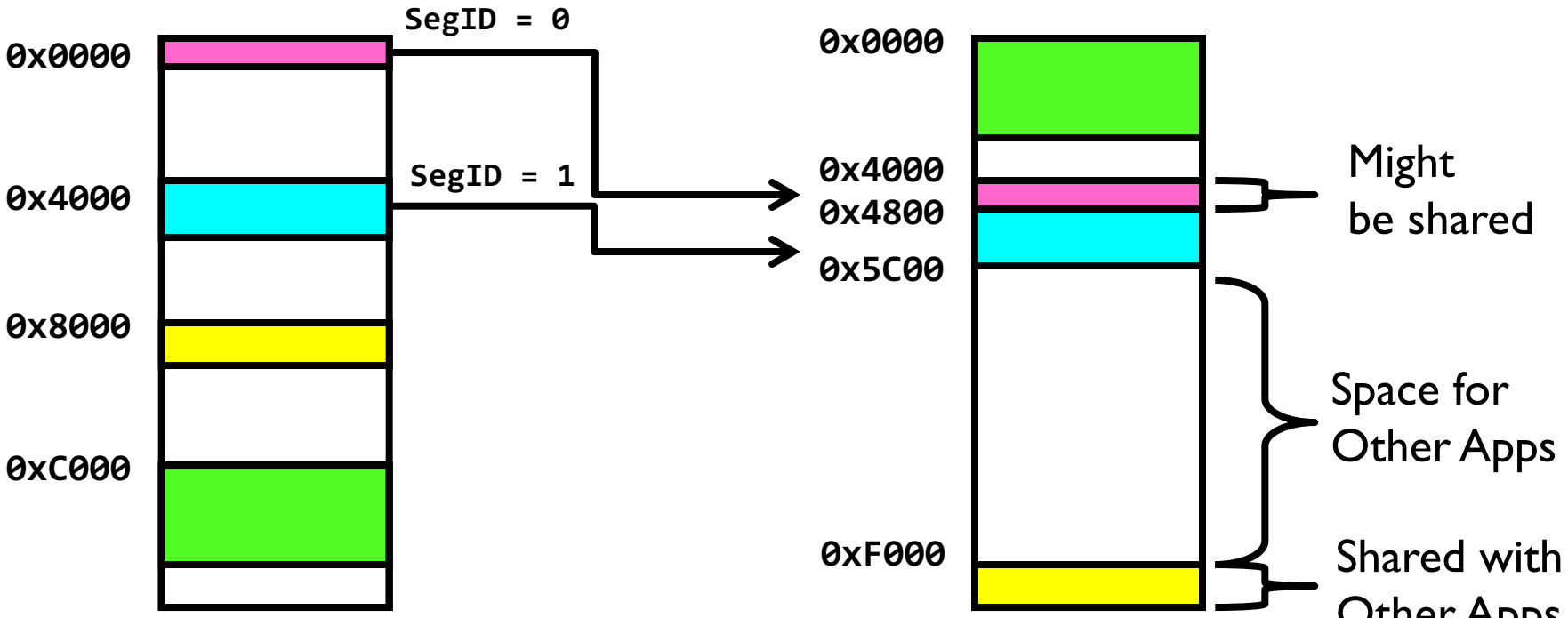
Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Segmentation

Example: Four Segments (16-bit Address)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Example of Segment Translation (16-bit Address)

```
0x0240     main:       la $a0, varx
0x0244                 jal strlen
  …                       …
0x0360     strlen:     li    $v0, 0  ;count
0x0364     loop:       lb    $t0, ($a0)
0x0368                 beq   $r0,$t0, done
  …                       …
0x4050     varx        dw    0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC

Segmentation

Example of Segment Translation (16-bit Address)

```
0x0240     main:      la $a0, varx
0x0244                jal strlen
  …                      …
0x0360     strlen:    li    $v0, 0  ;count
0x0364     loop:      lb    $t0, ($a0)
0x0368                beq   $r0,$t0, done
  …                      …
0x4050     varx       dw    0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

Segmentation

Example of Segment Translation (16-bit Address)

```
0x0240     main:      la $a0, varx
0x0244                jal strlen
  …                      …
0x0360     strlen:    li   $v0, 0   ;count
0x0364     loop:      lb   $t0, ($a0)
0x0368                beq  $r0,$t0, done
  …                      …
0x4050     varx       dw   0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x0360. Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x0000 → $v0, Move PC+4→PC

Segmentation

Example of Segment Translation (16-bit Address)

```
0x0240     main:      la $a0, varx
0x0244                jal strlen
  …                      …
0x0360     strlen:    li    $v0, 0  ;count
0x0364     loop:      lb    $t0, ($a0)
0x0368                beq   $r0,$t0, done
  …                      …
0x4050     varx       dw    0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x0360. Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x0364. Translated to Physical=0x4364. Get "lb $t0, ($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Segmentation

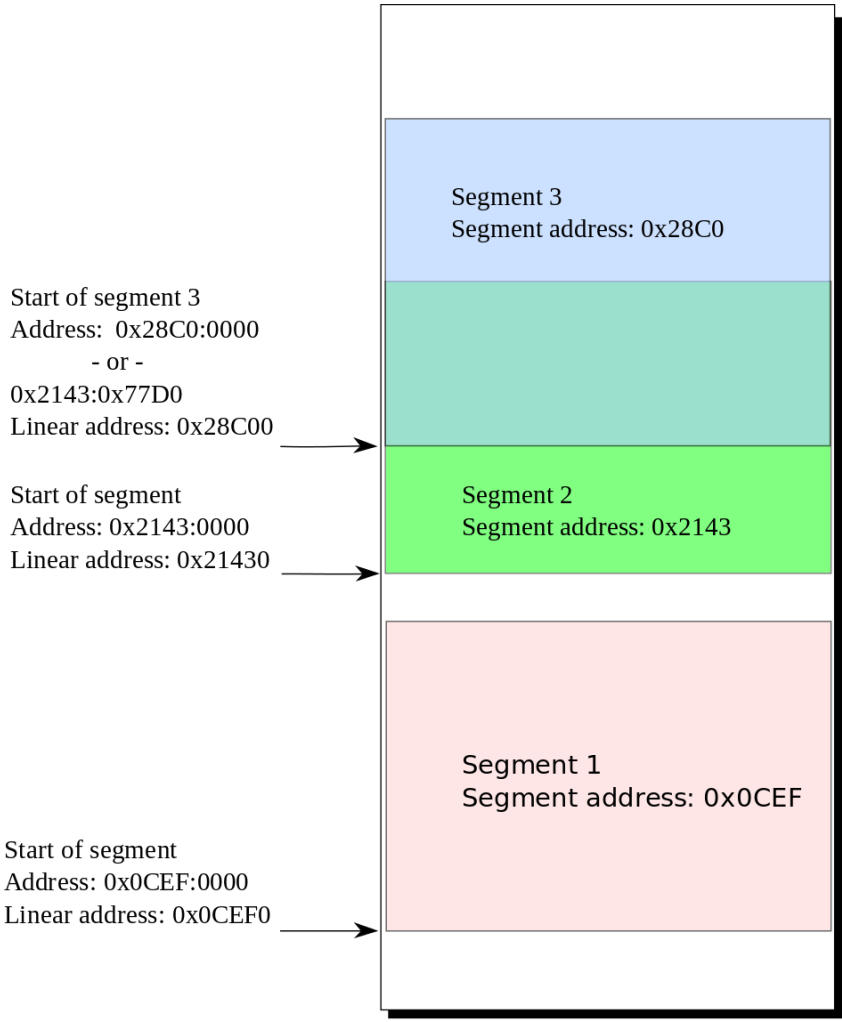
8086/8088 Real-Mode (16-bit)

```
  0000 0110 1110 1111 0000   Segment
+      0001 0010 0011 0100   Offset
-------------------------
  0000 1000 0001 0010 0100   Address
```

```
$ pintos-gdb kernel.o
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
...
Reading symbols from kernel.o...
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not
built into this configuration
of GDB.  Attempting to continue with the default
i8086 settings.

The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

CS Offset ⟹ 0xffff0



Segmentation

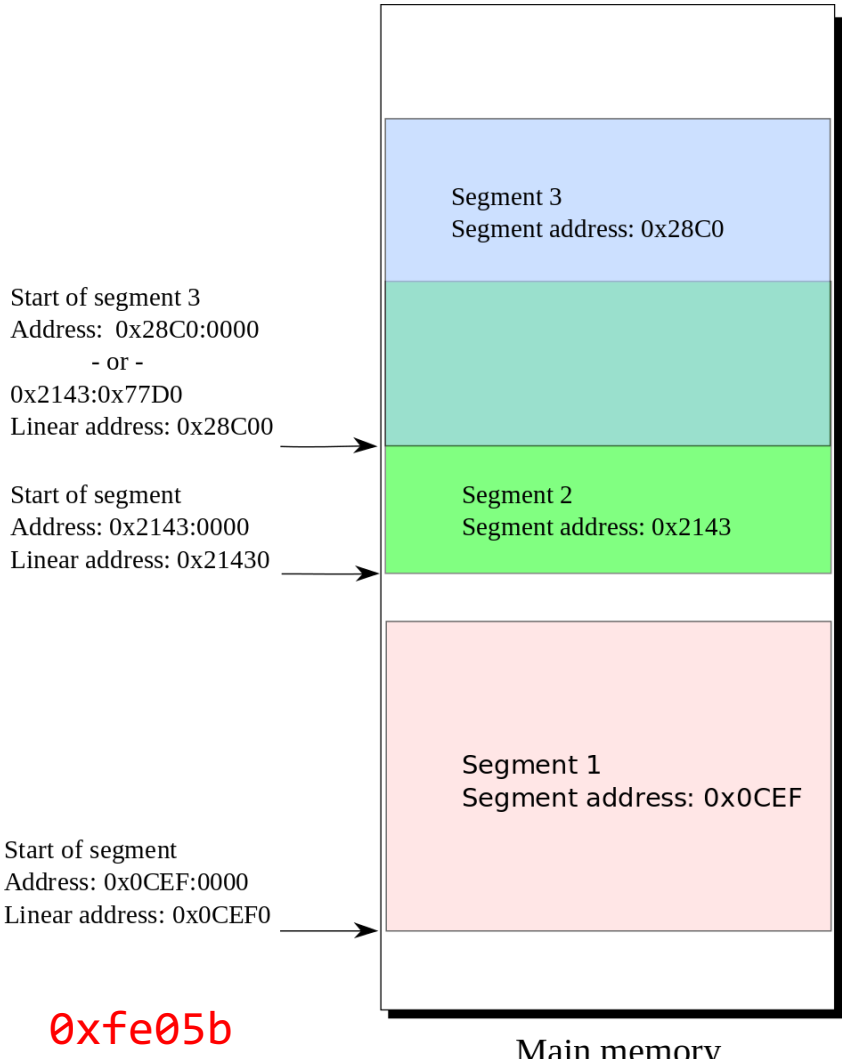
8086/8088 Real-Mode (16-bit)

```
  0000 0110 1110 1111 0000   Segment
+      0001 0010 0011 0100   Offset
-------------------------
  0000 1000 0001 0010 0100   Address
```

```
$ pintos-gdb kernel.o
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
...
Reading symbols from kernel.o...
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not
built into this configuration
of GDB.  Attempting to continue with the default
i8086 settings.

The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

(Code Segment)CS Offset ⇒ 0xfe05b



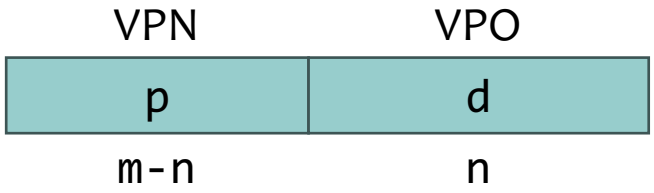
Segmentation

- **Problems with Segmentation**
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
    - Compaction
  - Limited options for swapping to disk
  - **Fragmentation**: wasted space
    - **Internal**: **Solved** (to some extent…)
    - **External**: still free gaps between allocated chunks.
  - How to avoid **External Fragmentation**?
    - Fixed-Sized chunks ⇒ **Paging**!

- **Paging**
  - **The Idea:** Allocate Physical Memory in **Fixed Size** Chunks
  - Breaking **physical memory** into fixed-sized blocks called **frames**.
  - Breaking **logical memory** into blocks of the **same** size called **pages**.

Address Translation Scheme

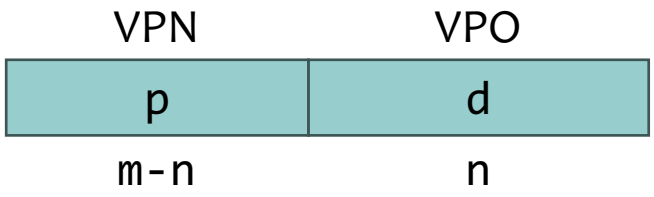
- Thus, **Logical Address** (*generated by CPU*) is divided into:
  - **Page Number** (p) or **Virtual Page Number** (**VPN**)
    - Used as an index into a page table which contains base address of each page in physical memory
  - **Page Offset** (d) or **Virtual Page Offset** (**VPO**)
    - combined with base address to define the physical memory address



- For given **logical address** space 2^m and **page size** 2^n

Address Translation Scheme

- Thus, **Logical Address** (*generated by CPU*) is divided into:
  - **Page Number** (p) or **Virtual Page Number** (**VPN**)
  - **Page Offset** (d) or **Virtual Page Offset** (**VPO**)



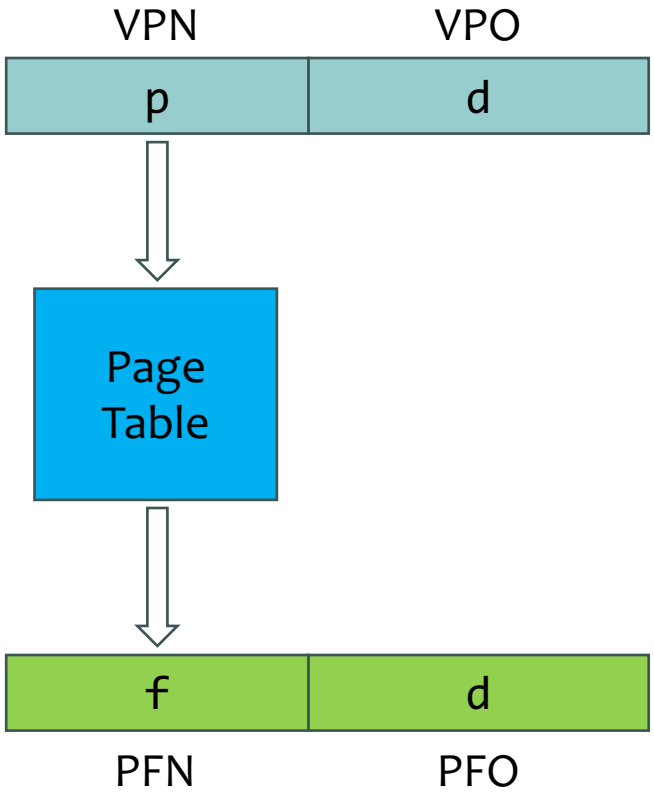
  - For given **logical address** space 2^m and **page size** 2^n
- Similarly, **Physical Address** (*generated by MMU*) is divided into:
  - **Frame Number** (f) or **Physical Frame Number** (**PFN**)
  - **Frame Offset** (d) or **Physical Frame Offset** (**PFO**)

VPO == PFO

  - For given **physical address** space $2^{m'}$ and **page size** 2^n

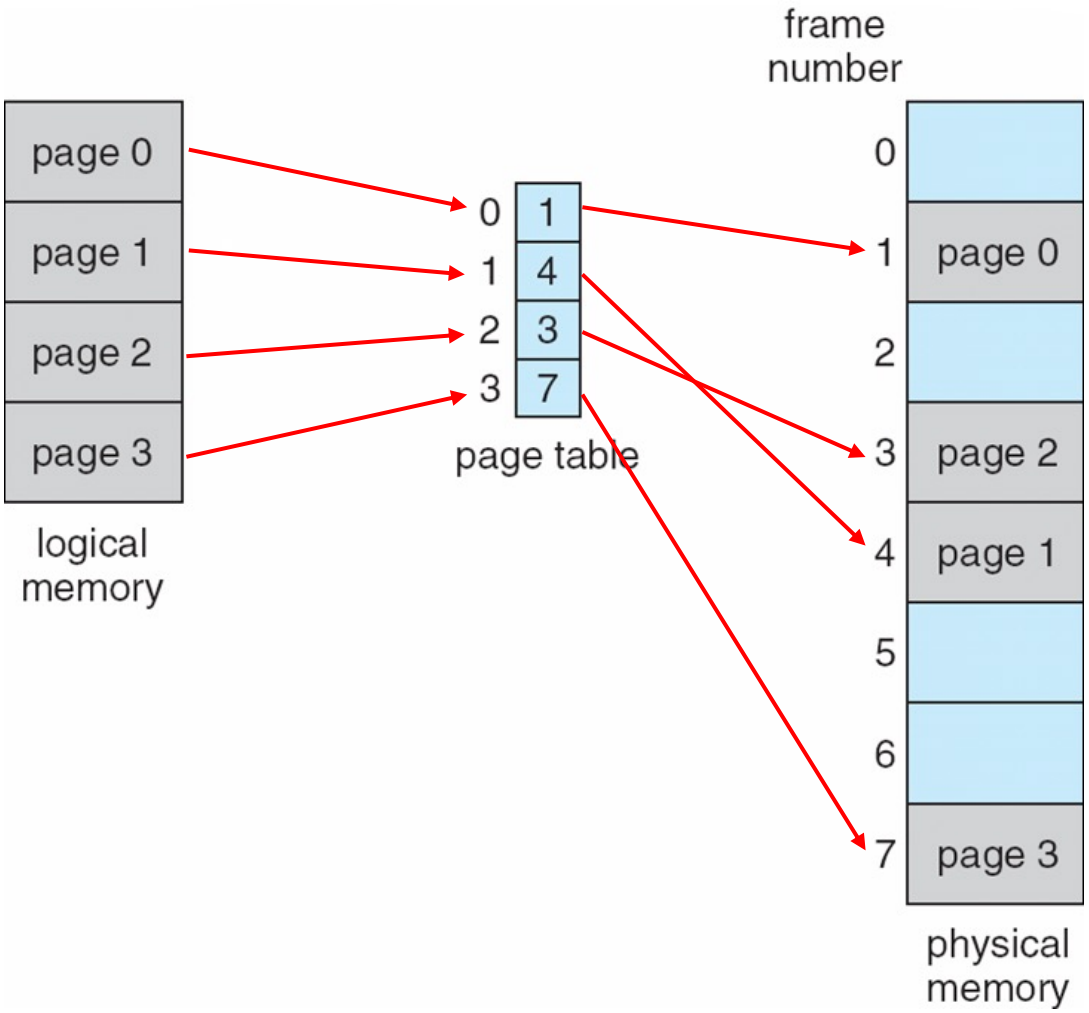
Paging

- **The Idea:** Allocate Physical Memory in **Fixed Size** Chunks
- Breaking **physical memory** into fixed-sized blocks called **frames**.
- Breaking **logical memory** into blocks of the **same** size called **pages**.
- The **Page Table** is a mapping (data structure) from **VPN** to **PFN**



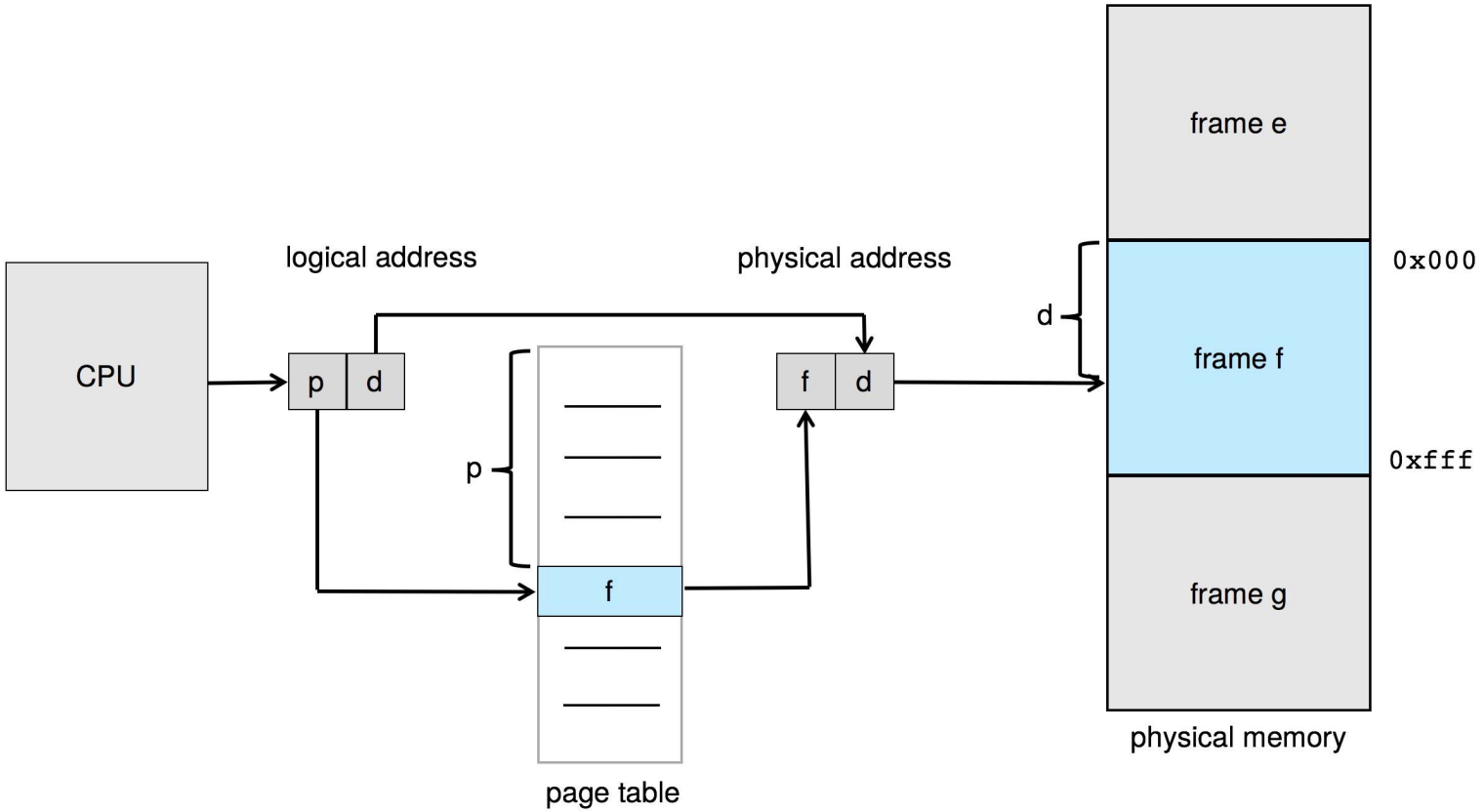
Paging

Paging Example



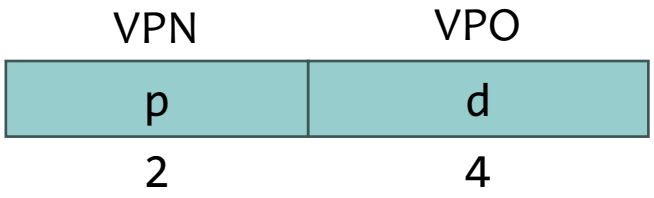
Address Translation Scheme

- Steps taken by the MMU to translate a logical addr to a physical addr:
  1. Extract the page number (p) and use it as an index into the page table.
  2. Extract the corresponding frame number (f) from the page table.
  3. Replace p in the logical address with the frame number f.



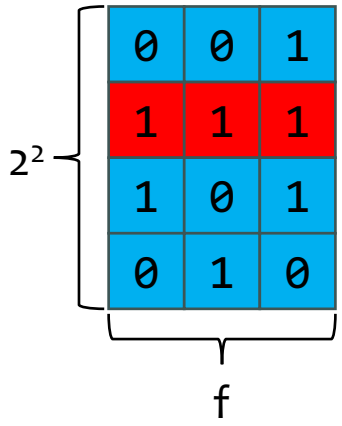
Address Translation Scheme

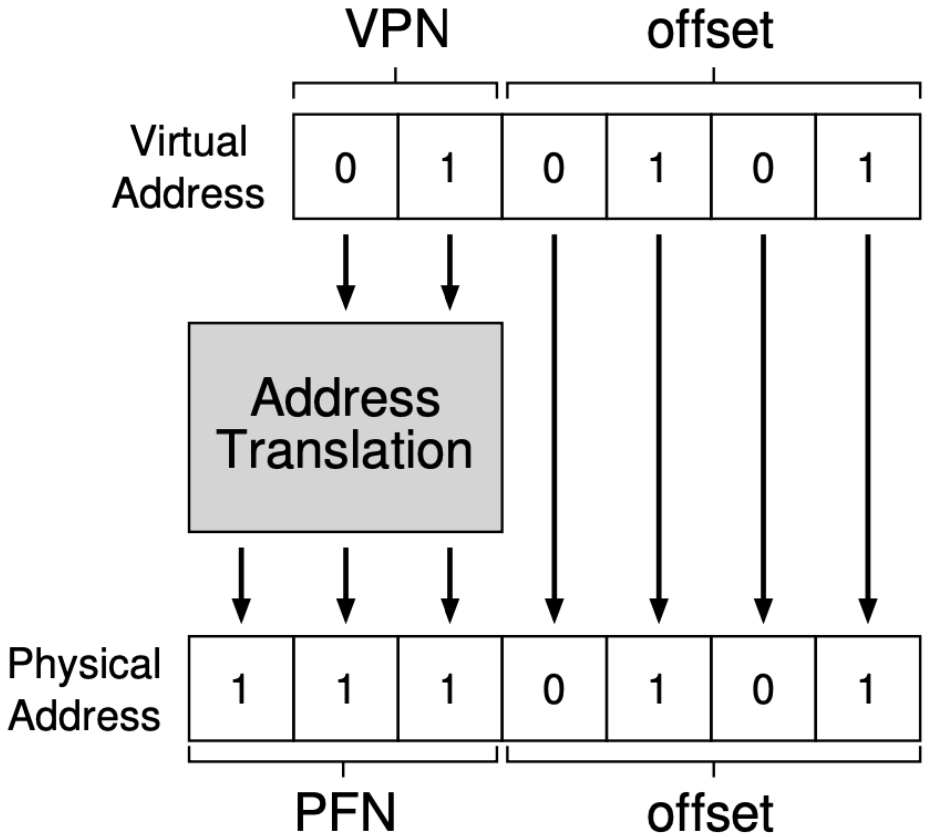
- Example: **Virtual Address** (6 bits) ⇒ **Physical Address** (7 bits)



- Number of **entries:** 2^2
- Size of each **page table entry** (**PTE**): 3

Size of the **page table**: $2^2 \times 3 = 12$ (bits)





Address Translation Scheme

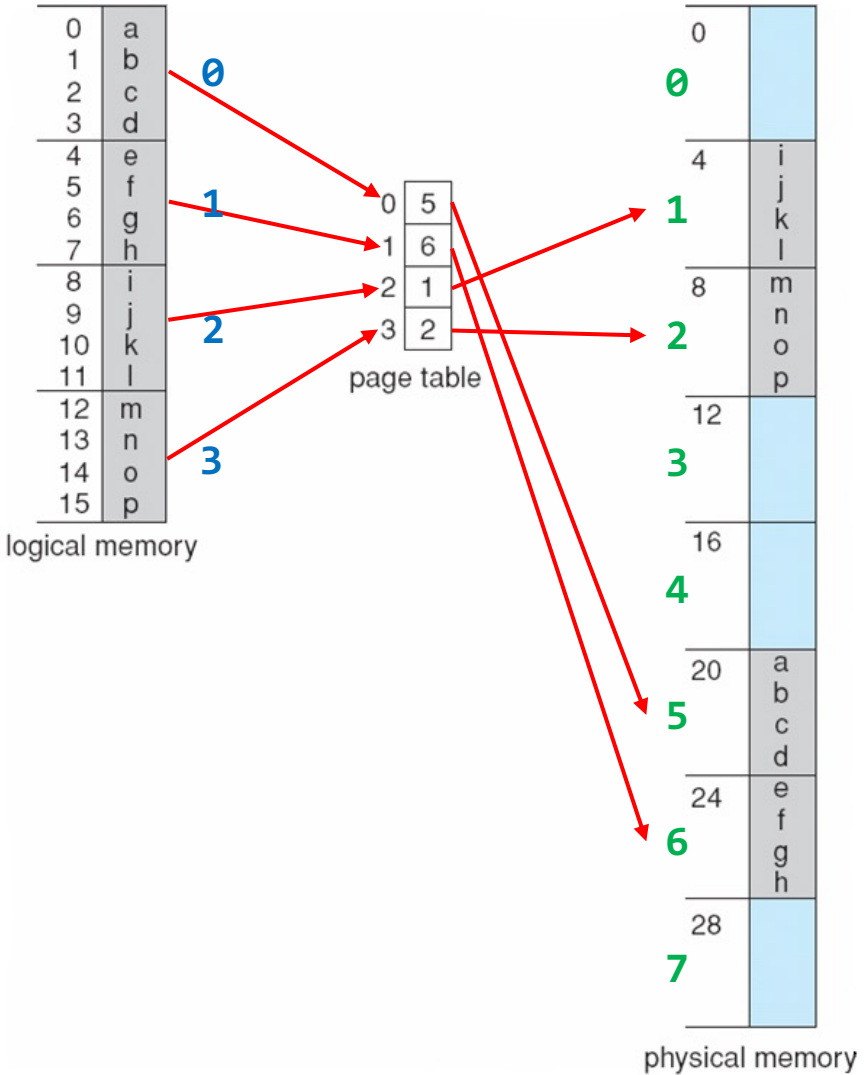
- Example: **Virtual Address** (4 bits) ⇒ **Physical Address** (5 bits)

VPN	VPO
p	d
2	2

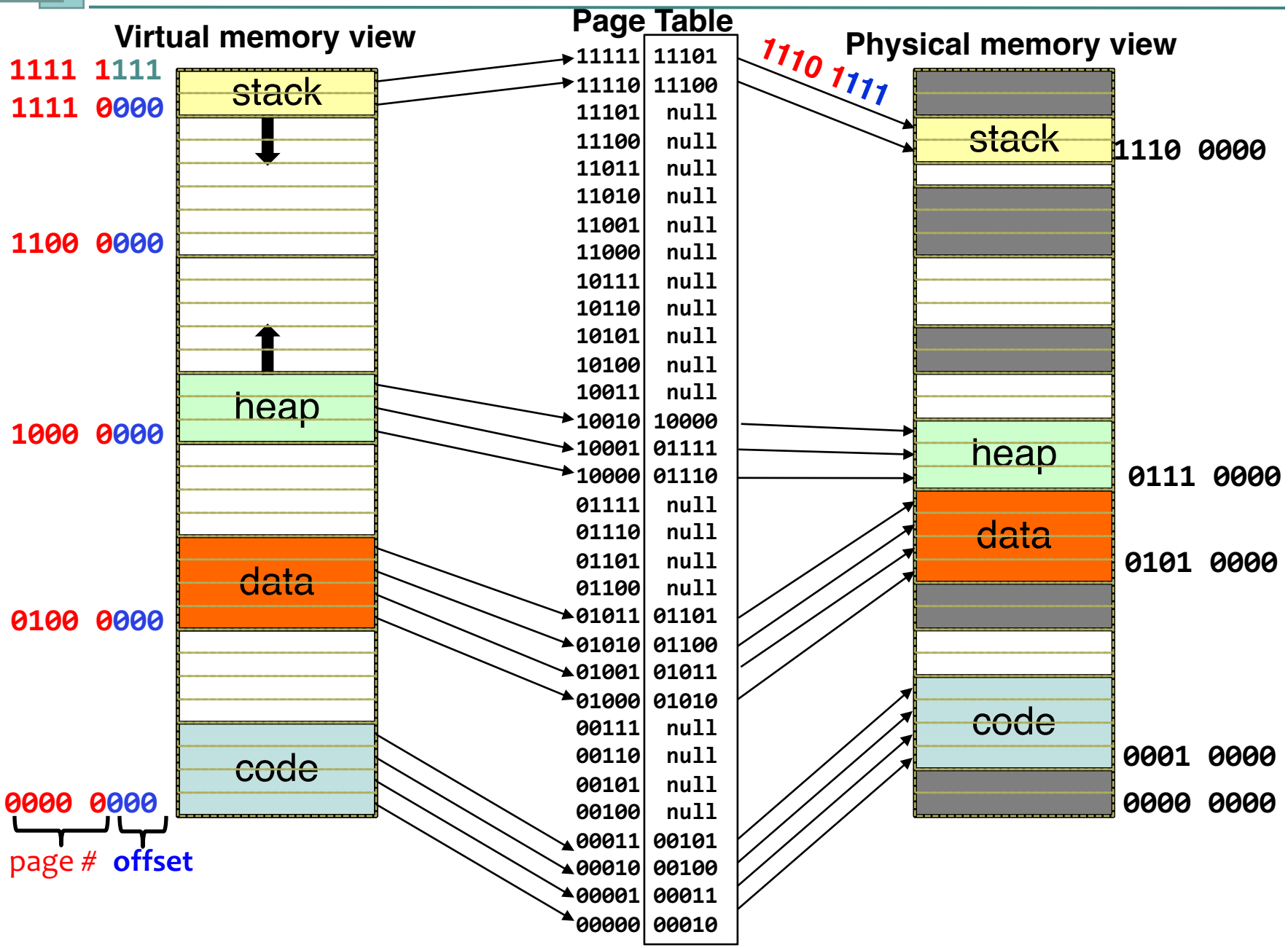
Page size = $2^n = 2^2 = 4$ Bytes
Number of pages: 2^2
Page table size: $2^2 \times 3 = 12$ (bits)

PFN	PFO
f	d
3	2

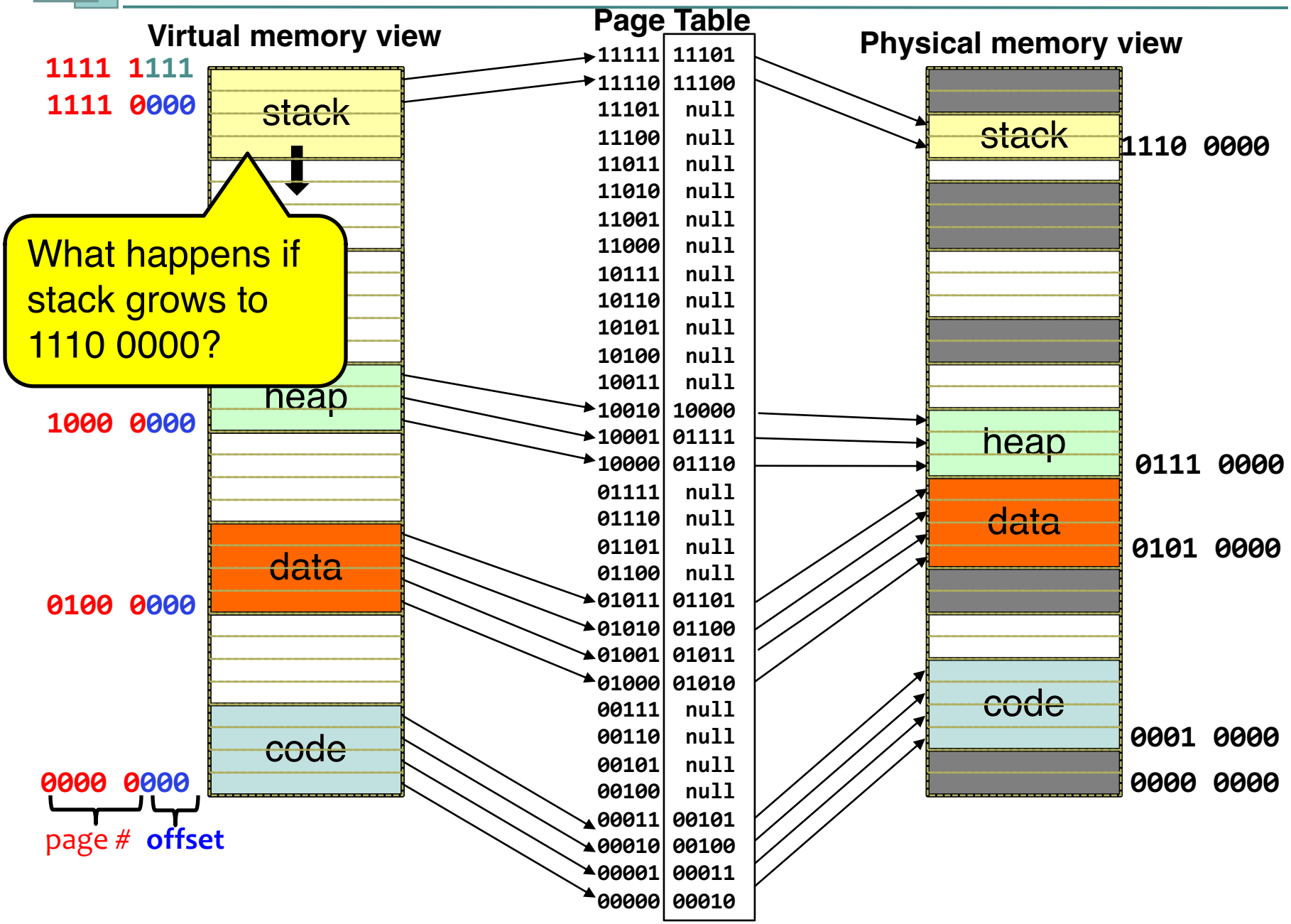
Number of frames: 2^3



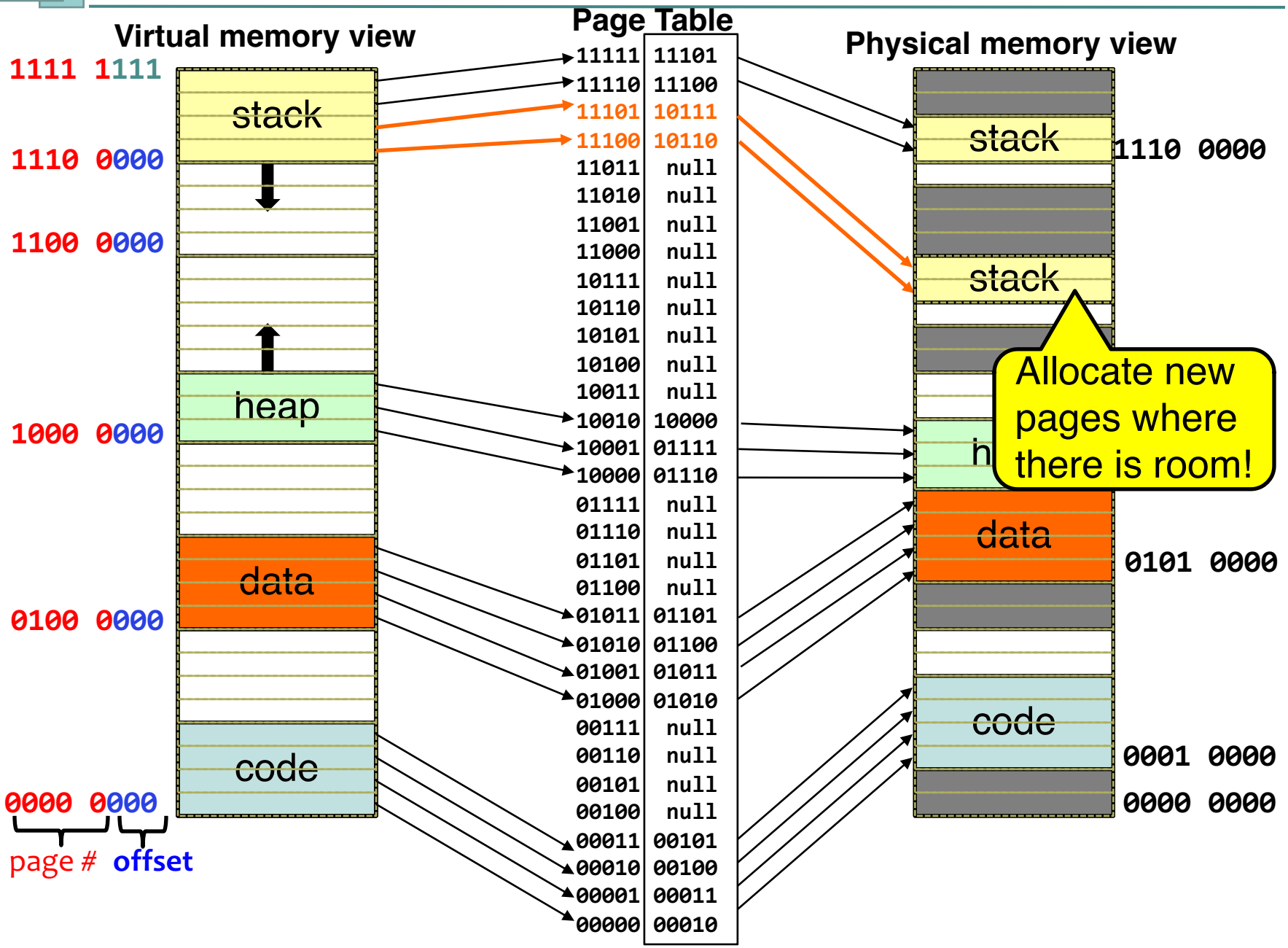
Page Table Example



Page Table Example

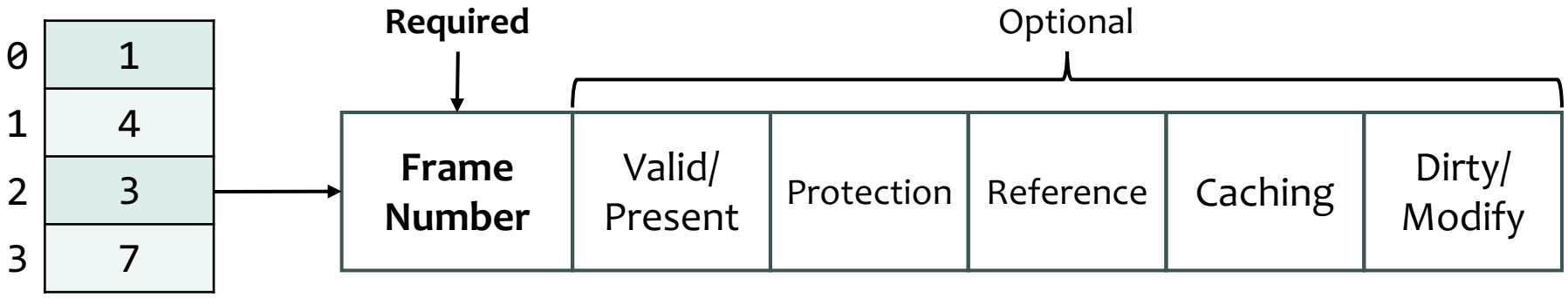


Page Table Example



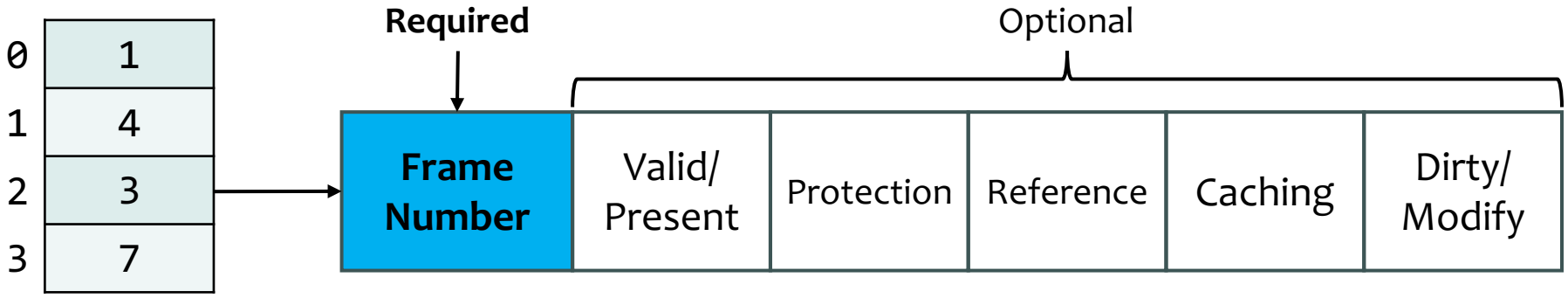
Page Table Entry (PTE)

- **PTE** (**页表项**) is an entry in the Page Table that contains information about a particular page of memory, *which may vary from OS to OS*.
- The most important info in **PTE** is the **Physical Frame Number** (**PFN**).
- **PTE** can optionally contain other bits, e.g., valid bit, protection bits.
- What does a **Page Table Entry** (**PTE**) look like?



Page Table

Page Table Entry (PTE)

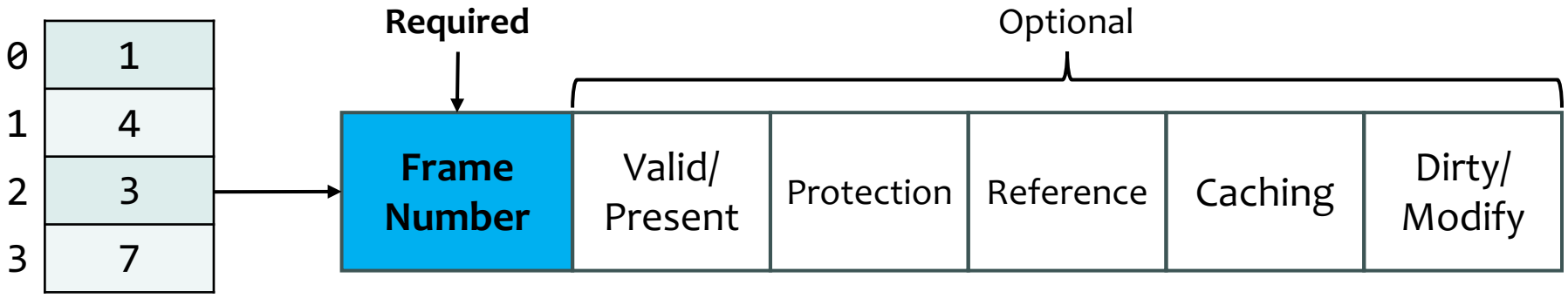


Page Table

- **Frame Number** denotes the frame where the page is present in the physical memory
- The number of **bits** required for **PFN** depends on the number of frames in the physical memory:

$$Number\ of\ bits\ for\ PFN = \log_2 \frac{Size\ of\ Physical\ Memory}{Frame\ Size}$$

Page Table Entry (PTE)



Page Table

- *Example*: Consider a machine with **64MB** of physical memory and a **32-bit** virtual address space. Assume a page size of **4KB** and the **PTE** contains only the **PFN,** what is the size of the page table?
- **Ans:**

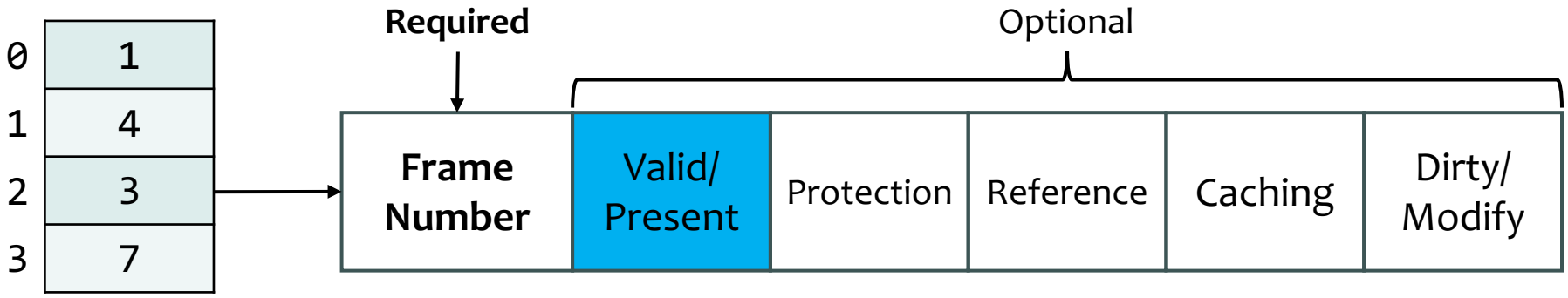
PFN Size = $\log_2(\frac{64MB}{4KB}) = \log_2(16K) =$ 14 bits ≈ 2 Bytes

# of Page Table Entries = $2^{32-\log_2(4K)} = 2^{20}$

Page Table Size = $2^{20} \times 2\ Bytes$ = 2MB

alignment

Page Table Entry (PTE)

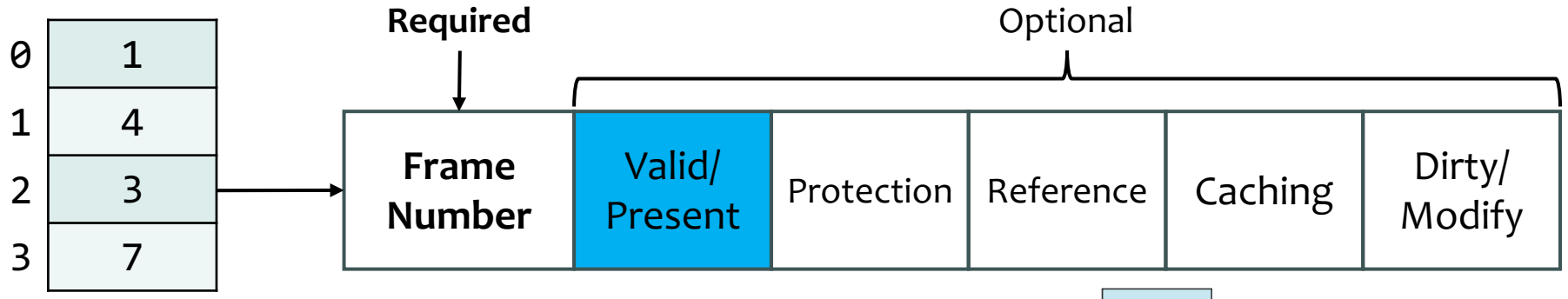


Page Table

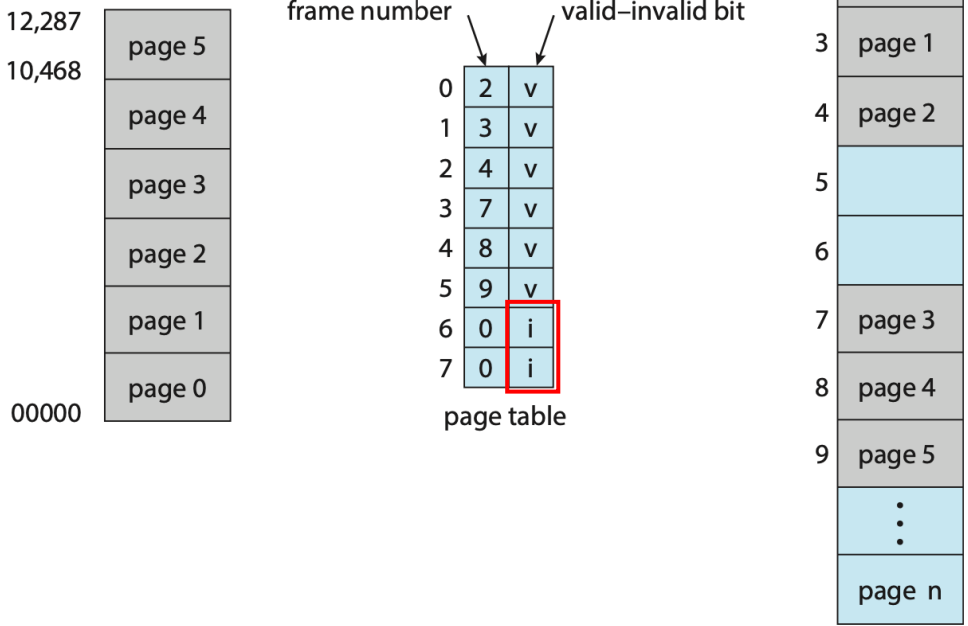
- The **Valid** bit (or **Present** bit) is commonly attached to a **PTE**.
- When this bit is set to 1, it means that the page is in physical memory, and the translation provided by the **PTE** can be used to convert a Virtual Address to a Physical Address.
- When this bit is set to 0, it means that the **PTE** is not currently valid, either the page is not present in physical memory, or the entry is not being used. Any access to this page will trigger an error (**Page Fault**).

Paging

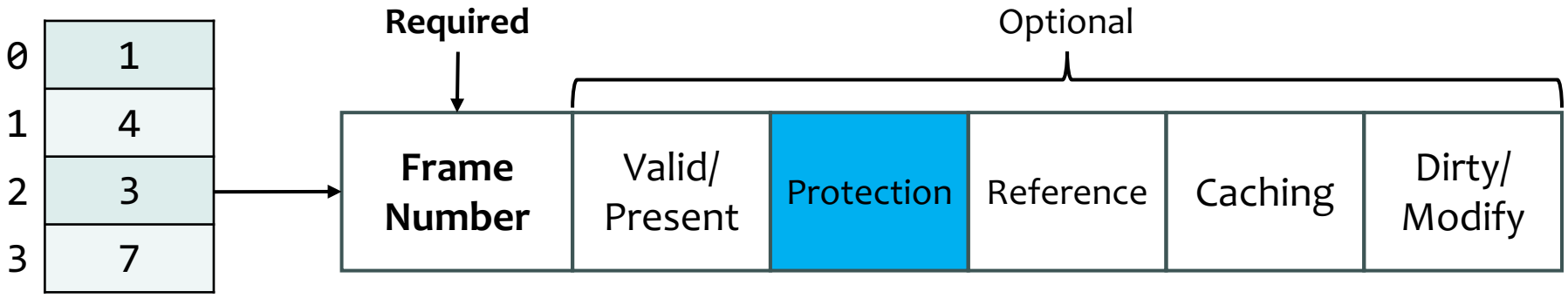
Page Table Entry (PTE)



Page Table



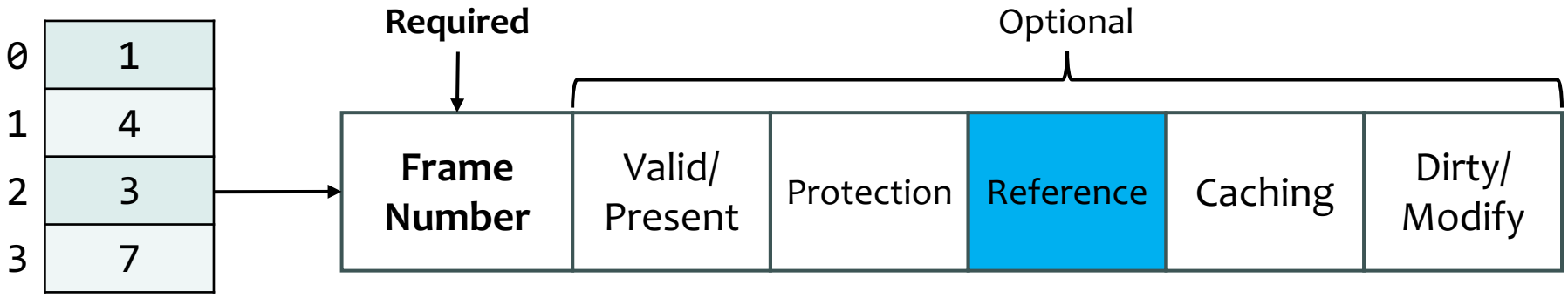
Page Table Entry (PTE)



Page Table

- The protection bit specifies the permissions for **READ**, **WRITE**, or **EXECUTE** operations on that page.
- For example, we can use 3 bits to indicate [**Read**, **Write**, **Execute**].
  - If a page belongs to the **Code** segment of a process, then that page should have protection bits of 101 (Read and Execute, but no Write).
  - A page belonging to the **Stack** segment should have protection bits of 110 (Read and Write, but no Execute).

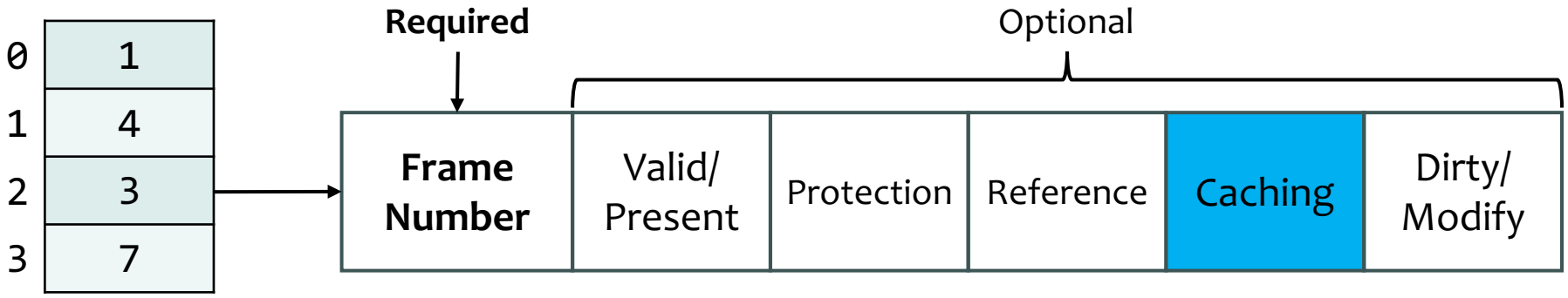
Page Table Entry (PTE)



Page Table

- The reference bit specifies whether the page has been referenced in the last clock cycle or not.
- It is set to 1 when the page is accessed.
- Useful for implementing page replacement algorithms in **TLB**.
  - We will talk more about it in later lectures.

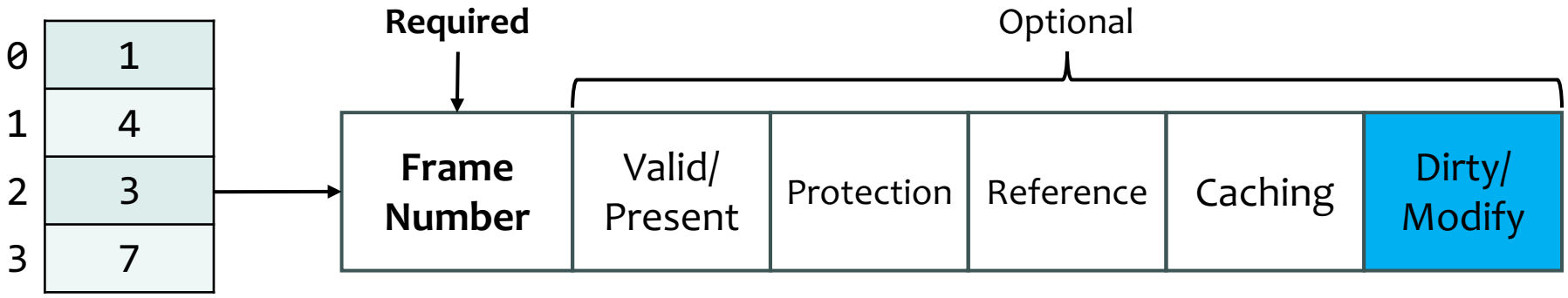
Page Table Entry (PTE)



Page Table

- The caching bit is used for enabling or disabling caching of the page.
- When we need fresh data, we have to disable caching so as to avoid fetching of old data from the cache.
- When caching has to be disabled, this bit is set to 1. Otherwise, it is set to 0.

Page Table Entry (PTE)

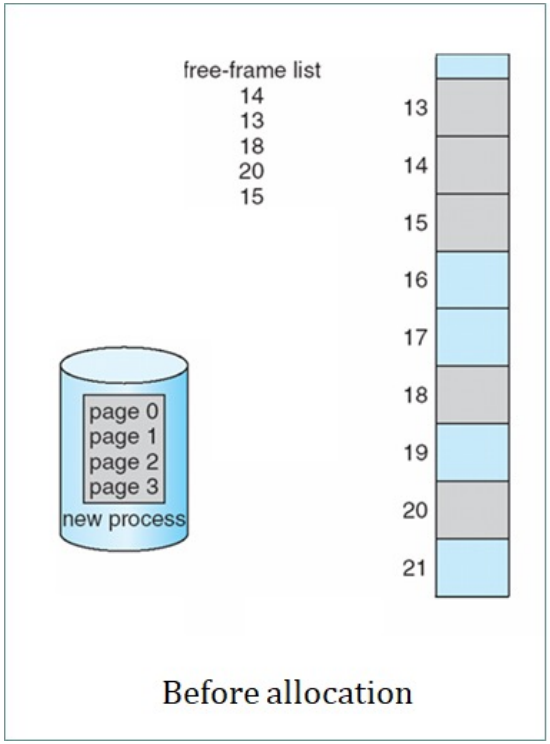


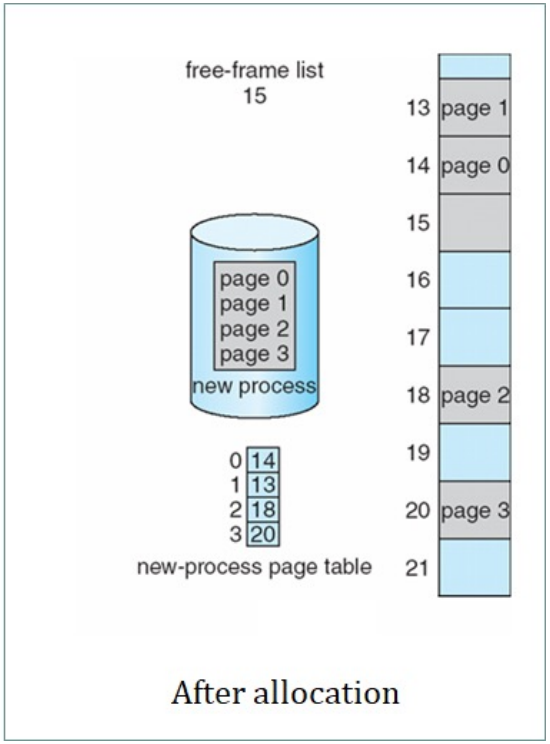
Page Table

- The **Dirty** bit is also known as the **Modify** bit.
- It specifies whether the page has been modified or not.
- If the page has been modified, then this bit is set to `1`, otherwise `0`.
- This bit helps in avoiding unnecessary writes to the secondary memory when a page is being replaced by another page.

Paging

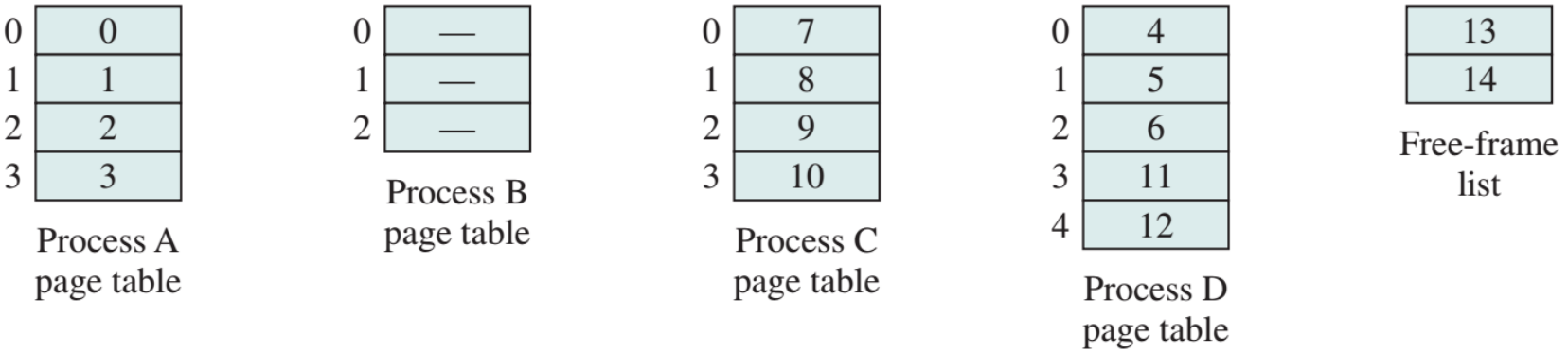
- Process pages can be assigned to any free frames in physical memory
  - A process does not need to occupy a contiguous portion
  - To run a program of size N **pages**, we need to find N free **frames** of physical memory and load the program.
    - Use free-frame-list to keep track of free frames in physical memory
    - Use page table (per-process) to translate pages to frames.





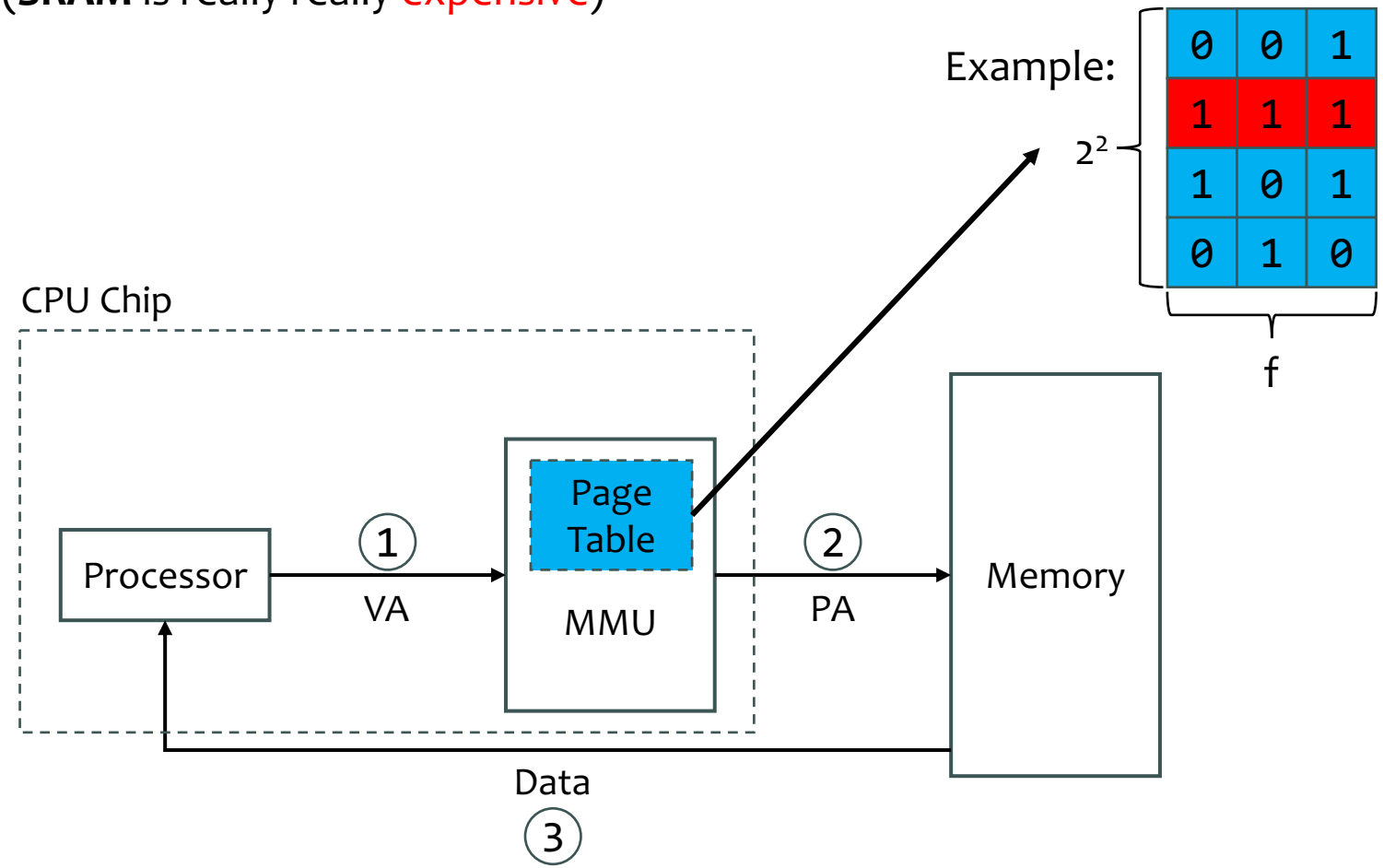
Paging

- OS now needs to maintain (in main memory):
  - A **page table** for each process
    - Each entry of a page table consists of the frame number (physical) where the corresponding page is located
    - The corresponding page table is indexed by the page number (logical) to obtain the frame number.
  - A **free frame list** of available frames
    - can be implemented as a simple table.
  - Example:



Hardware Implementation of Page Table

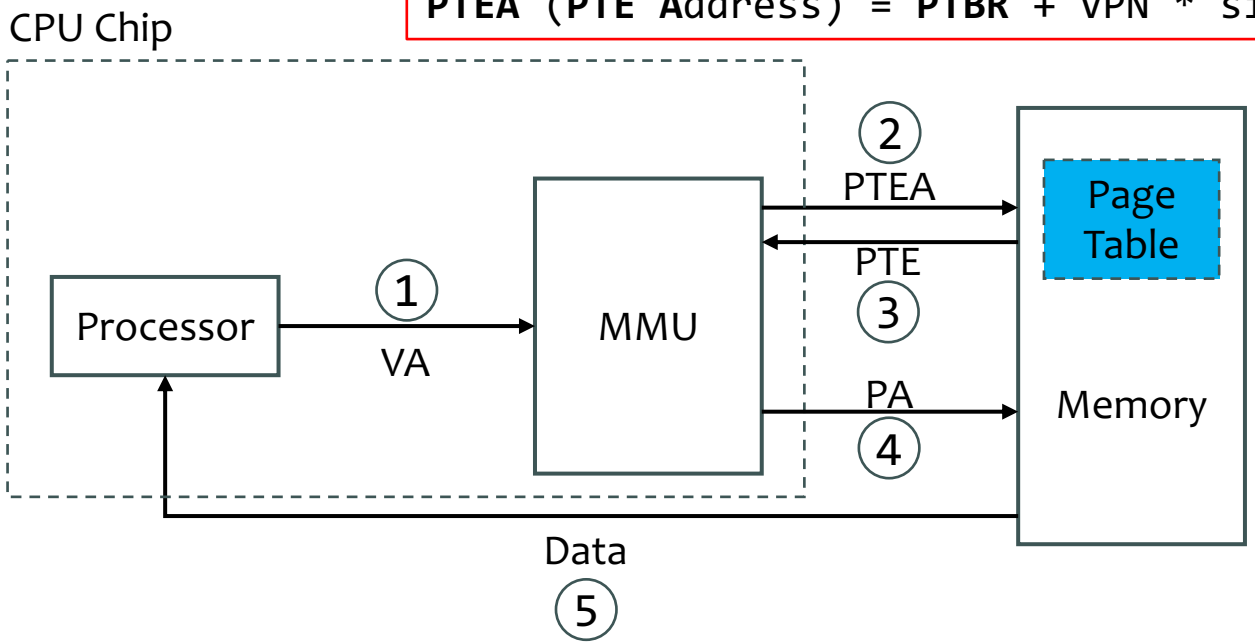
- **Case 1:** Implement the page table as a set of **dedicated registers**
  - Problem: This can only be used when page table is reasonably **small**. (**SRAM** is really really expensive)



Hardware Implementation of Page Table

- **Case 2:** Keep the page table in **main memory** and a **Page-Table Base Register** (**PTBR**) points to the page table.
  - Problem: The time required to access a user memory location is increased as there are two memory accesses needed to access a byte
    - One for the **Page-Table Entry** (**PTE**)
    - One for the actual **Physical Address** (**PA**)

```
PTEA (PTE Address) = PTBR + VPN * sizeof(PTE)
```



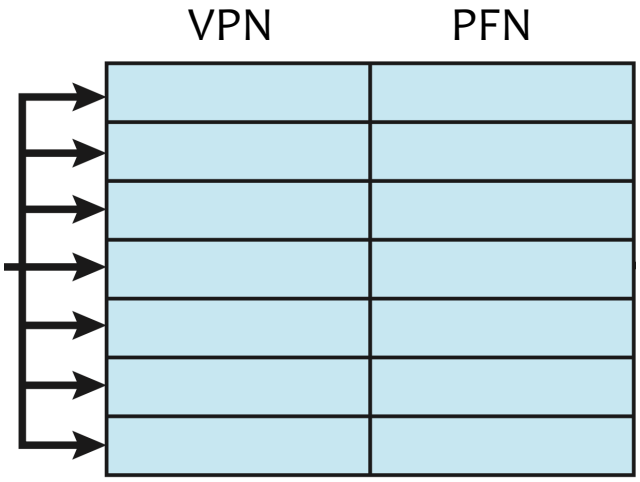
- **Hardware Implementation of Page Table**
  - **Case 1:** Implement the page table as a set of **dedicated registers**
    - Problem: This can only be used when page table is reasonably small. (**SRAM** is really really expensive)
  - **Case 2:** Keep the page table in **main memory** and a **Page-Table Base Register** (**PTBR**) points to the page table.
    - Problem: The time required to access a user memory location is increased as there are two memory accesses needed to access a byte
      - One for the **Page-Table Entry** (**PTE**)
      - One for the actual **Physical Address** (**PA**)
  - **Solution: TLB** (**T**ranslation **L**ook-Aside **B**uffer)
    - (转换后援缓冲器， or 快表)
    - Combine **Case 1** and **Case 2**:
      - thought of the **TLB** as a cache for popular **PTE**s
    - The **TLB** is associative, high-speed memory that consists of two parts:
      - a **key** (or tag),
      - a **value**.

Hardware Implementation of Page Table

- **Solution: TLB** (**T**ranslation **L**ook-Aside **B**uffer)
  - (转换后援缓冲器， or 快表)
  - Combine **Case 1** and **Case 2:**
    - thought of the **TLB** as a cache for popular **PTE**s
    - **TLB** contains a **small subset** of the entries in the actual Page Table.
  - The **TLB** is associative, high-speed memory that consists of two parts:
    - a **key** (or tag) containing the VPN
    - a **value** containing the PFN
- What does a **TLB** look like?
  - Each entry of the TLB is {**VPN** + **PTE**} plus some extra bits.
  - **Searching is fast:** a **TLB** lookup in modern hardware is part of the instruction pipeline, incurring no performance penalty.
  - However, the **TLB** must be kept **small**. It is typically between 32 and 1024 entries in size.

TLB: a Closer Look

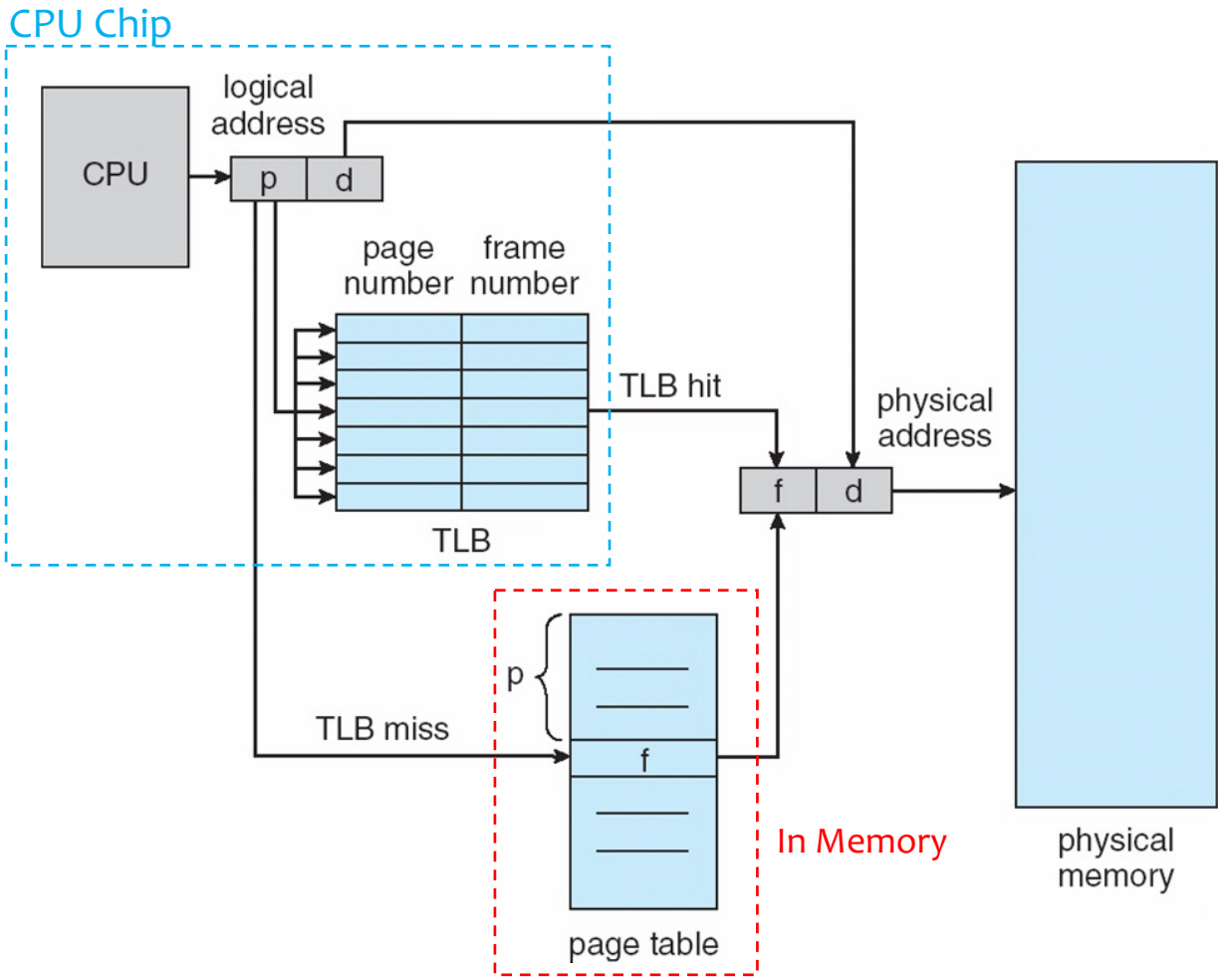
- What does associative mean?
  - Any given translation can be anywhere in the TLB, and the hardware will search the entire TLB in parallel to find the desired translation.
  - I.e., when the associative memory is presented with an item, the item is compared with all keys **simultaneously ($O(1)$)**, instead of **one by one ($O(N)$)**, *where* **N** *is the number of entries of* ***TLB***.



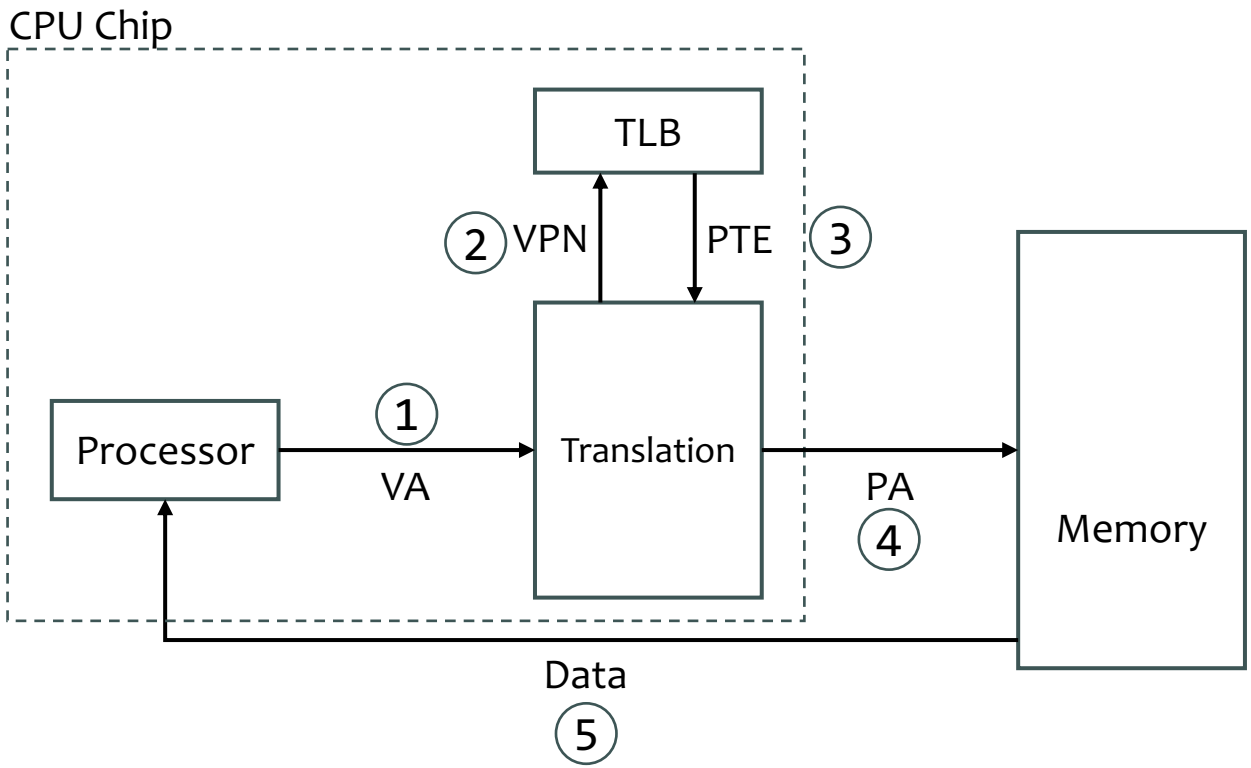
VPN: Virtual **P**age **N**umber
PFN: Physical **F**rame **N**umber

TLB

- Paging Hardware with TLB

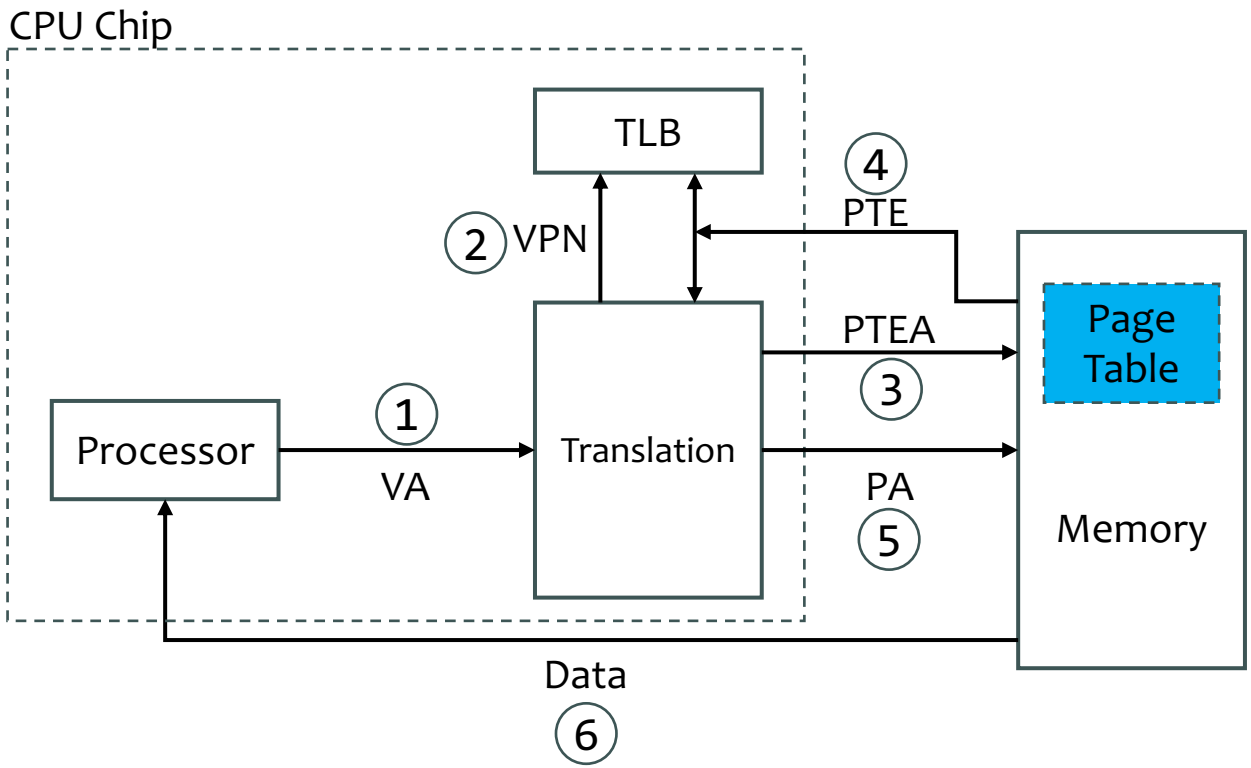


- **TLB**
  - **TLB hit**: If the page number (**VPN**) is in the **TLB**, there is no need to access the page table in memory
    - **MMU** = Translation + **TLB**



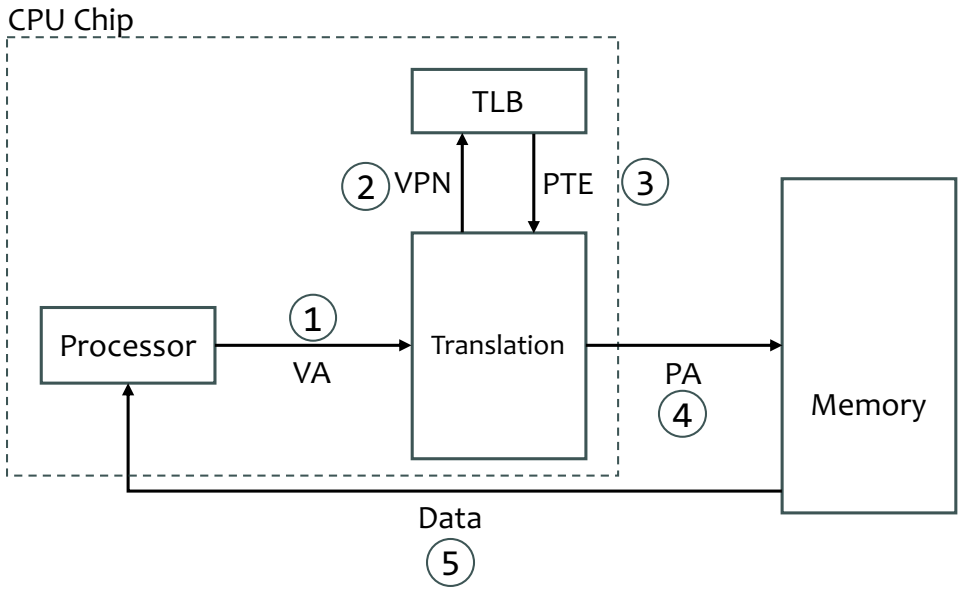
TLB

- **TLB miss**: If the page number (**VPN**) is **NOT** in the **TLB**, then the **TLB** will have to fetch the missing **PTE** from the page table in memory.
  - The newly fetched **PTE** is stored in the **TLB**, possibly overwriting an existing entry.

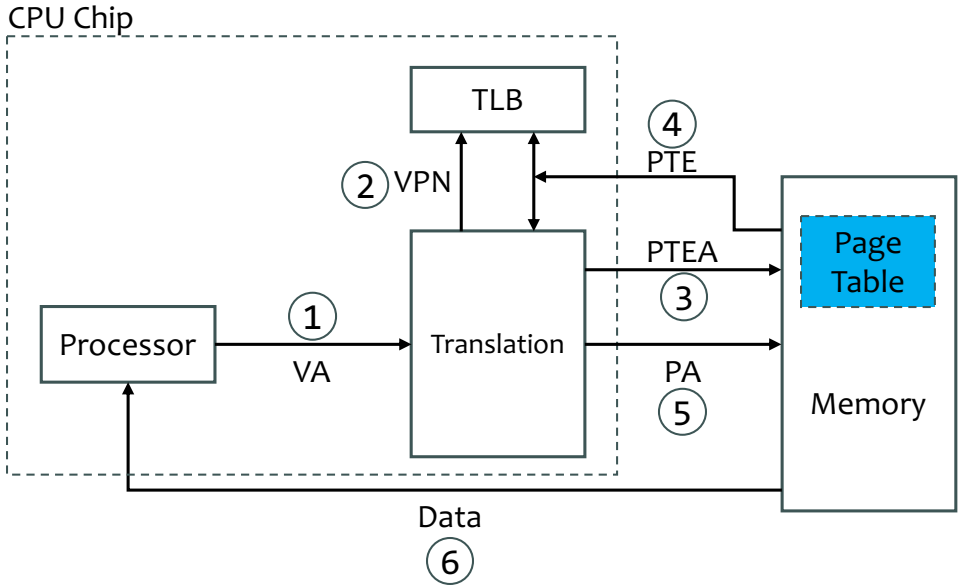


TLB

TLB hit:



TLB miss:



- **TLB**
  - TLB takes advantage of the Locality Principle (局部性原理)
    - Popular pages/frames are more likely to be accessed more often.
  - TLB hit ratio: the percentage of times that the page number of interest is found in the TLB.
    - TLB hit ratio can generally be up to 90%+.
    - A 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
    - Suppose that it takes 10 nanoseconds to access memory
      - 1) TLB **hit**: memory access time is 10 ns
      - 2) TLB **miss**: memory access time is 20 ns
    - **Effective Access Time** (**EAT**):
      - **EAT** = $80\% \times 10 + 20\% \times 20$ = 12 ns
      - implying 20% slowdown in memory access time.
    - Consider a more realistic hit ratio of 95%:
      - **EAT** = $95\% \times 10 + 5\% \times 20$ = 10.5 ns

Paging

- **How big is the Page Table?**
  - The page size in a typical OS (e.g., Linux) is 4KB.

```
$ getconf PAGESIZE
4096
```

    - For **32-bit** virtual address space, the **Virtual Address** is divided into:
      - 20-bit VPN
      - 12-bit Offset
    - Number of page table entries (**PTE**s): 2^{20} = 1M
    - Assume a physical memory size of 16GB (2^{34} Bytes):
      - 22-bit PFN
      - 12-bit Offset
    - Size of each page table entry (**PTE**):
      - 22-bit ~ 3 Bytes $\Rightarrow$ 4 Bytes (due to alignment)
    - Total size of a page table: 1M x 4 Bytes = 4MB
    - The page table is per-process, so the total size of page tables in the system is 4MB x (# of processes)!
      - Assume a system with 1000 processes running, **4GB** is allocated to address translation alone!

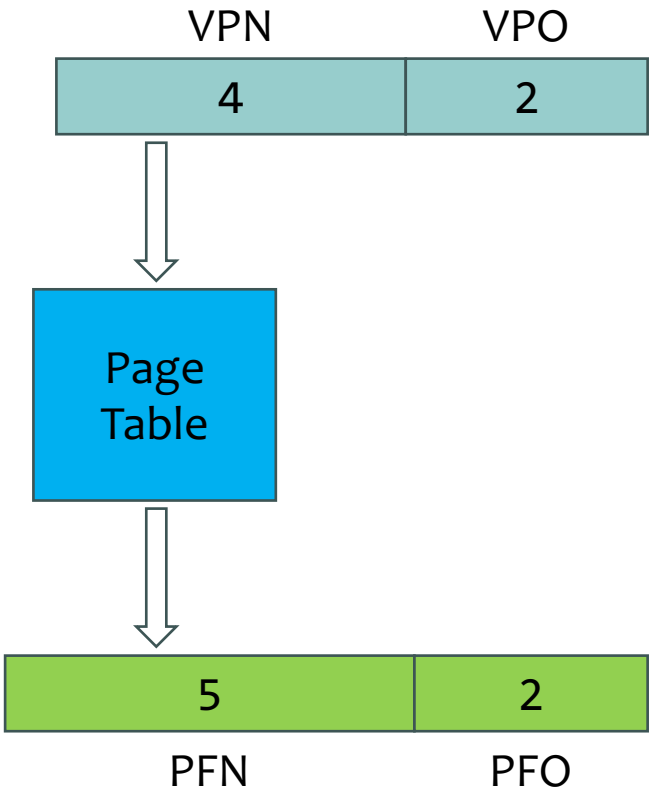
- **How big is the Page Table?**
  - The page size in a typical OS (e.g., Linux) is 4KB.
    - For **64-bit** virtual address space...
      - Don't even think about it...

```
$ getconf PAGESIZE
4096
```

  - Simple array-based page tables (usually called **linear page tables**) are too big, taking up far too much memory on the system.
  - As a result, we need to somehow reduce the size of the page tables in physical memory.
    - How can we make page tables **smaller**?
  - We explore some of the most common techniques:
    - **Hierarchical** Page Tables or **Multi-Level** Page Tables (分层页表)
    - **Hashed** Page Tables (哈希页表)
    - **Inverted** Page Tables (倒置页表)

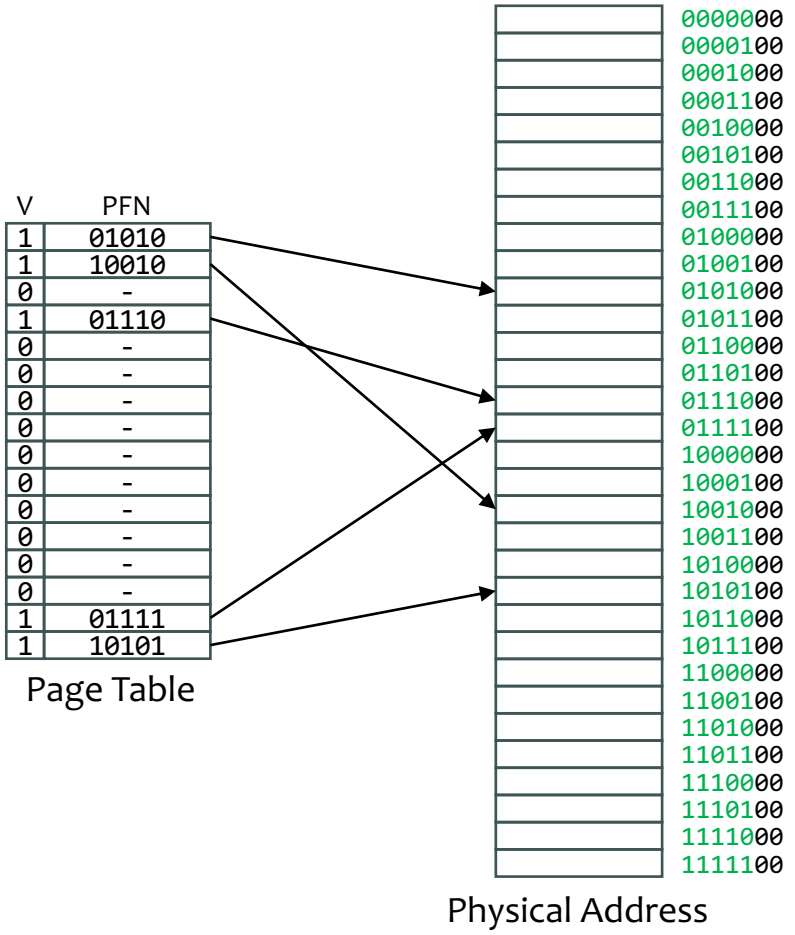
Linear Page Table

- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)



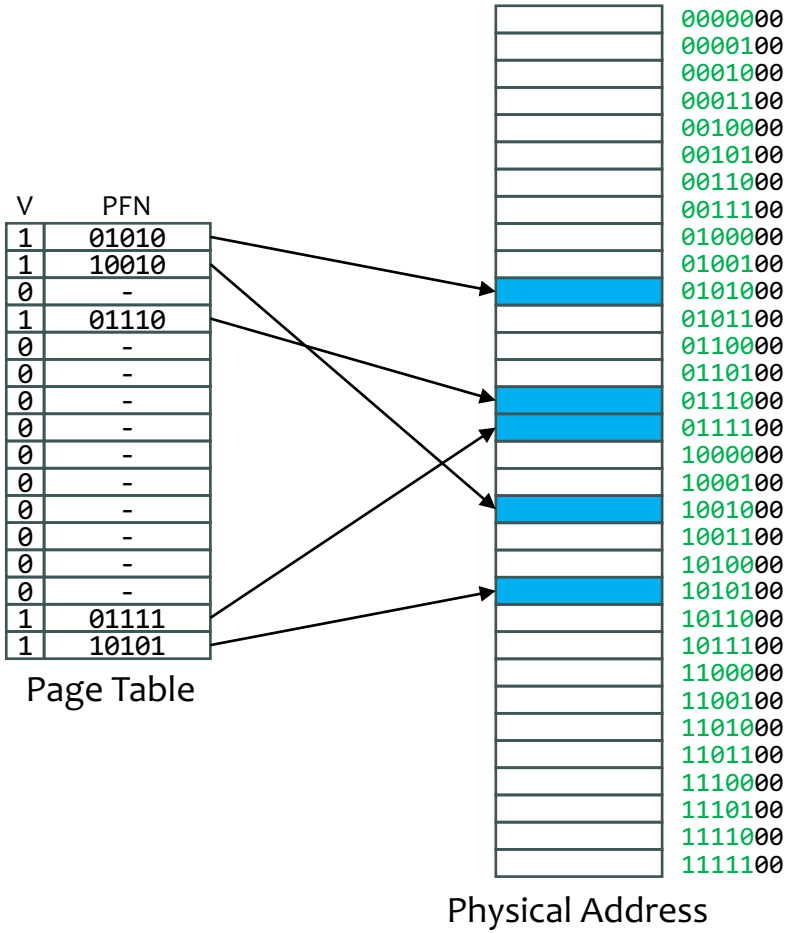
Linear Page Table

- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)



Linear Page Table

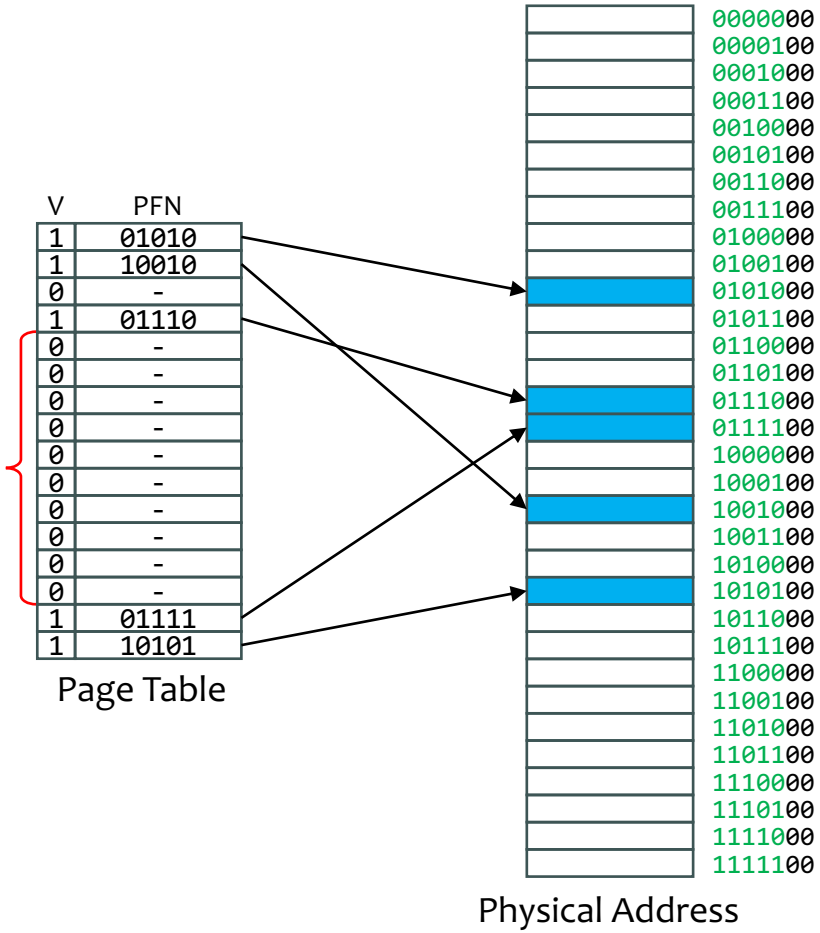
- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)



Linear Page Table

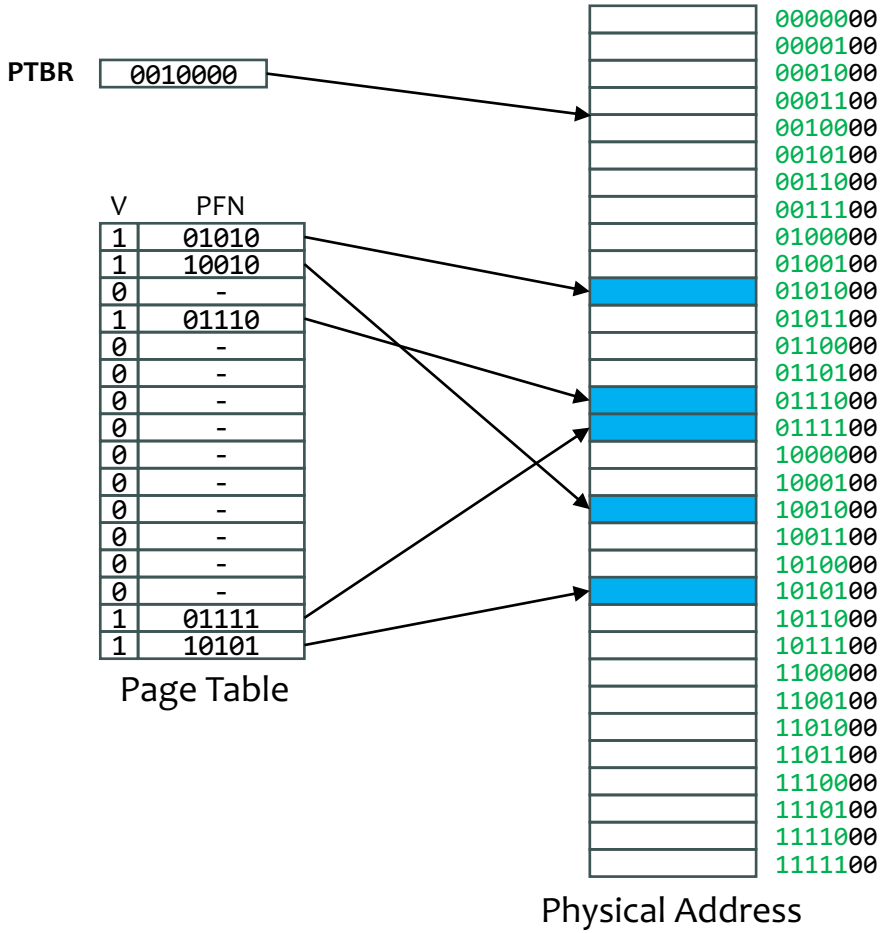
- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)

even though most of the middle regions of the virtual address space are empty, we still require page table space allocated for those regions



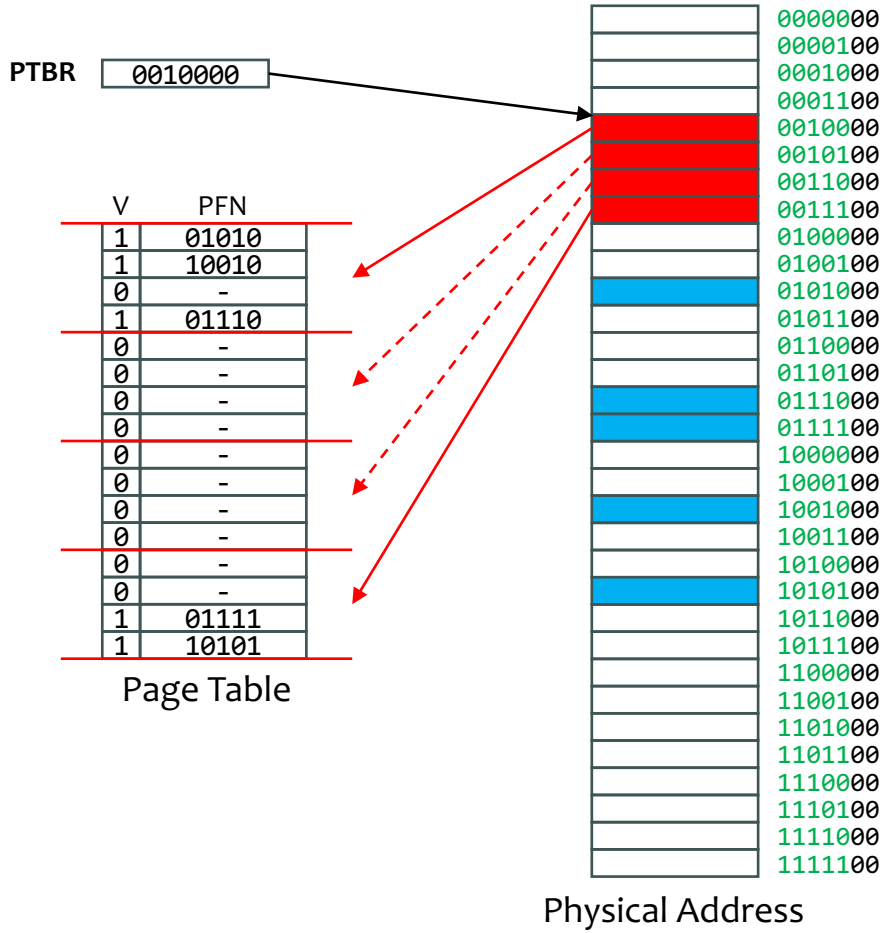
Linear Page Table

- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)
  - The Page Table itself resides in the physical memory.



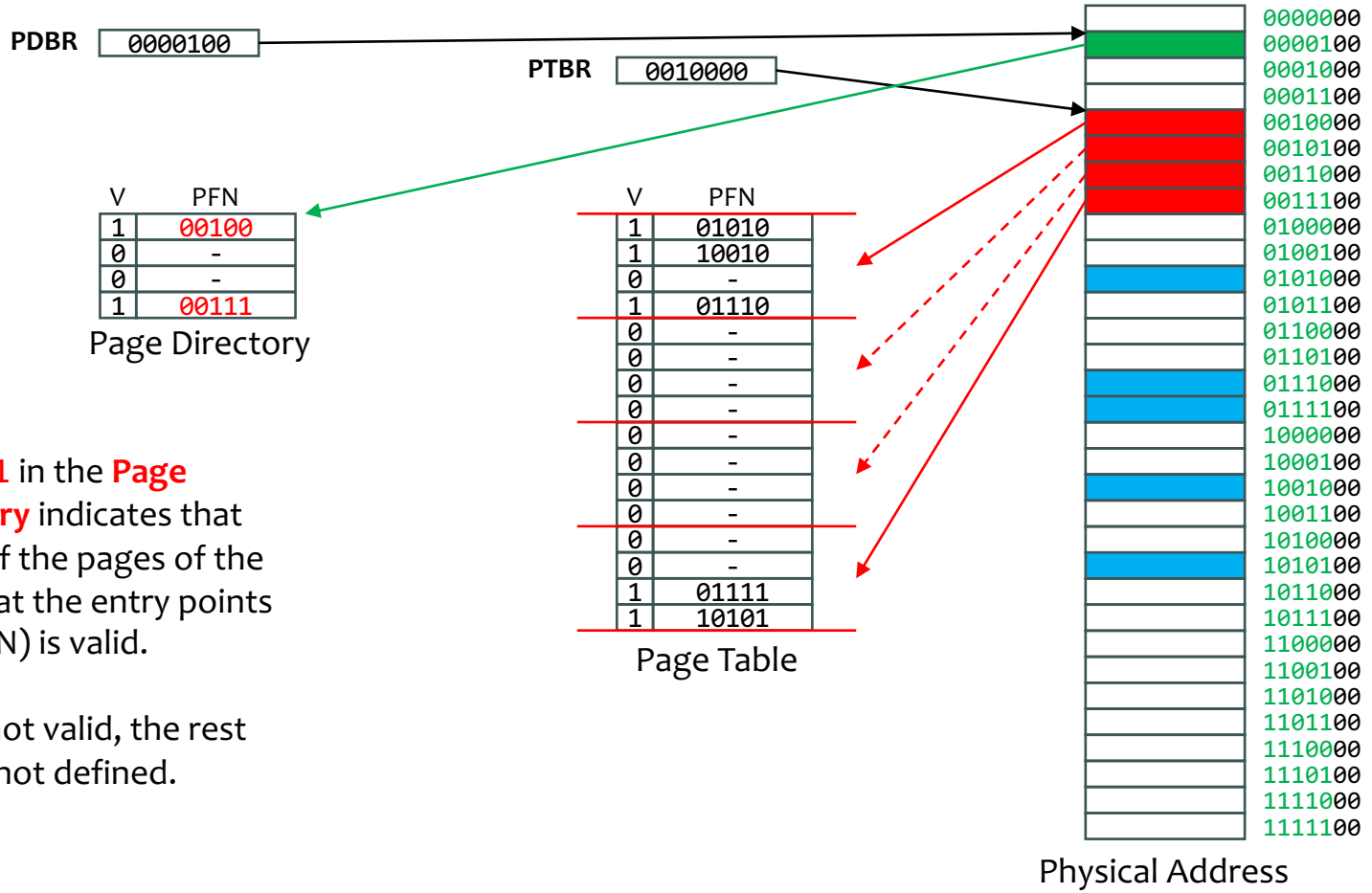
Linear Page Table

- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)
  - The Page Table itself resides in the physical memory, it is also **paged**.



Multi-Level Page Table

- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)
  - The Page Table itself resides in the physical memory, it is also **paged**.

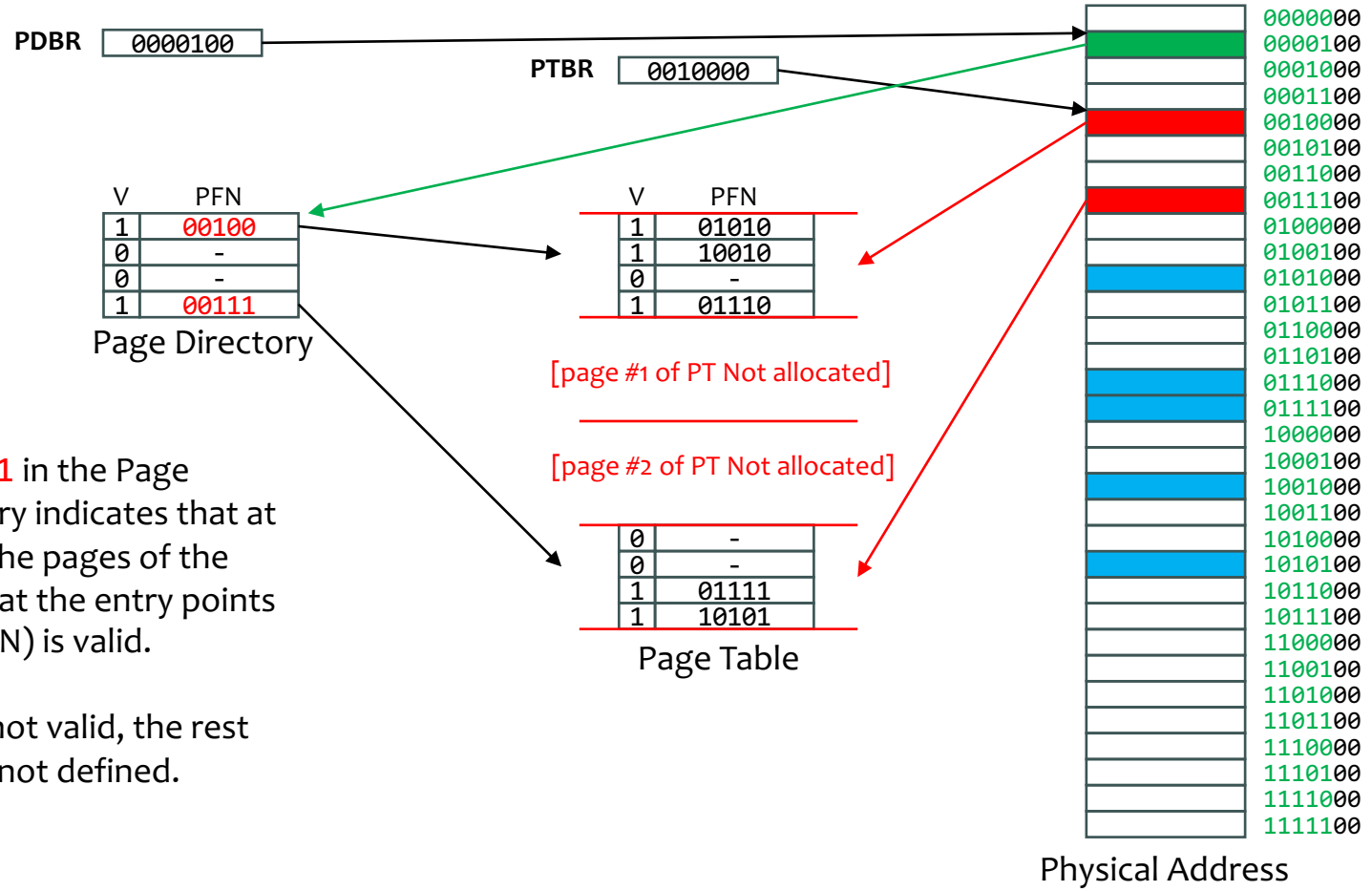


A valid bit of 1 in the **Page Directory Entry** indicates that at least one of the pages of the page table that the entry points to (via the PFN) is valid.

If the **PDE** is not valid, the rest of the **PDE** is not defined.

Multi-Level Page Table

- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)
  - The Page Table itself resides in the physical memory, it is also **paged**.

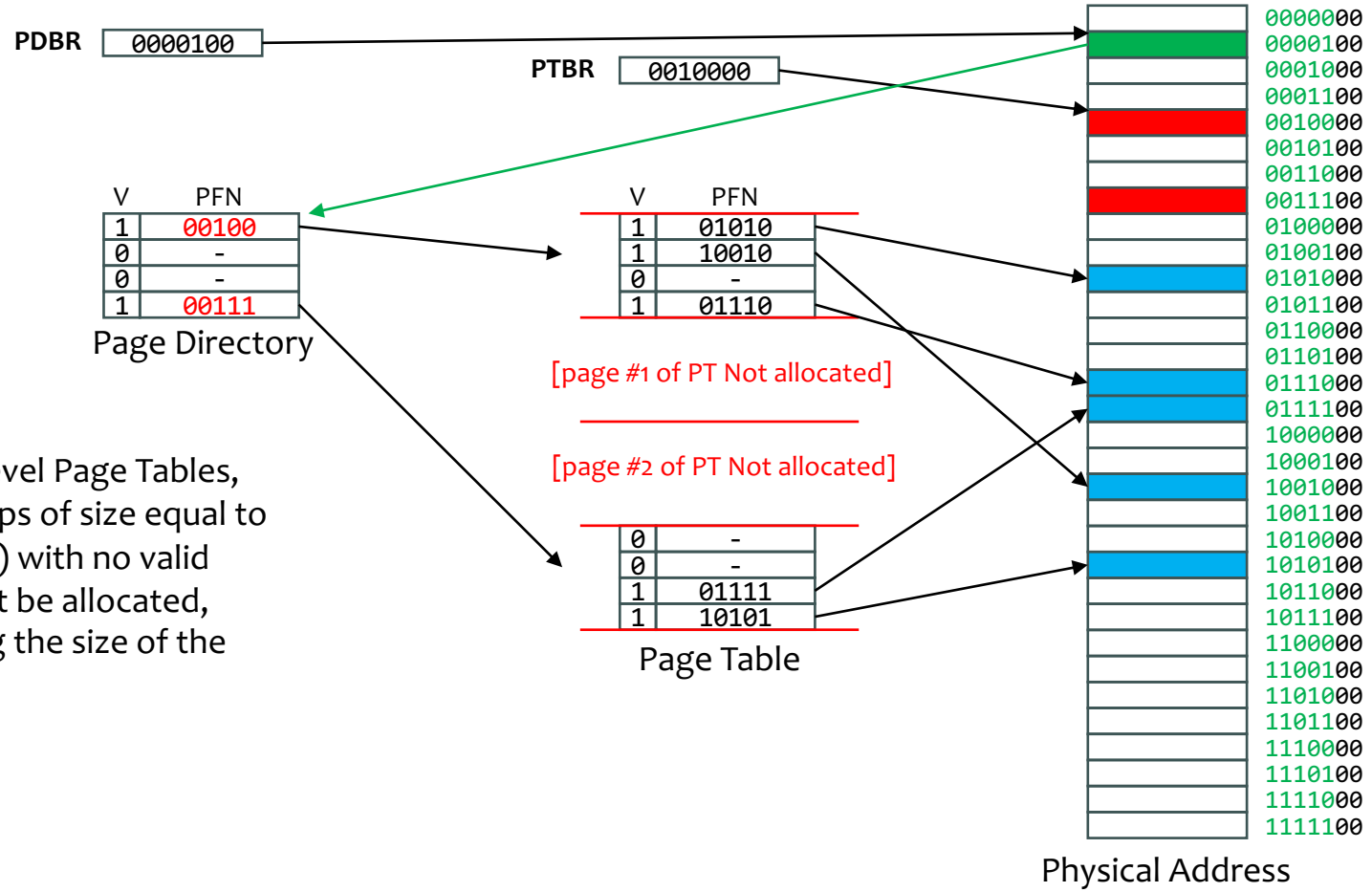


A valid bit of 1 in the Page Directory Entry indicates that at least one of the pages of the page table that the entry points to (via the PFN) is valid.

If the PDE is not valid, the rest of the PDE is not defined.

Multi-Level Page Table

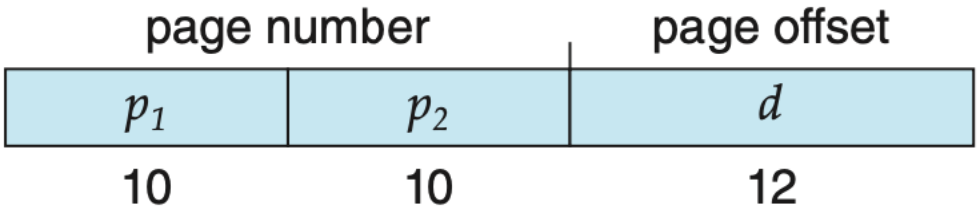
- Consider a system with **Page Size** of 4 bytes, **PA** 7 bits, **VA** 6 bits
  - ⇒ **Offset** 2 bits, **VPN** 4 bits, **PFN** 5 bits. (*Assume **PTE** size of **1 byte**.*)
  - The Page Table itself resides in the physical memory, it is also **paged**.



With Multi-Level Page Tables, PTEs (in groups of size equal to the page size) with no valid pages will not be allocated, thus reducing the size of the page table.

Multi-Level Page Table

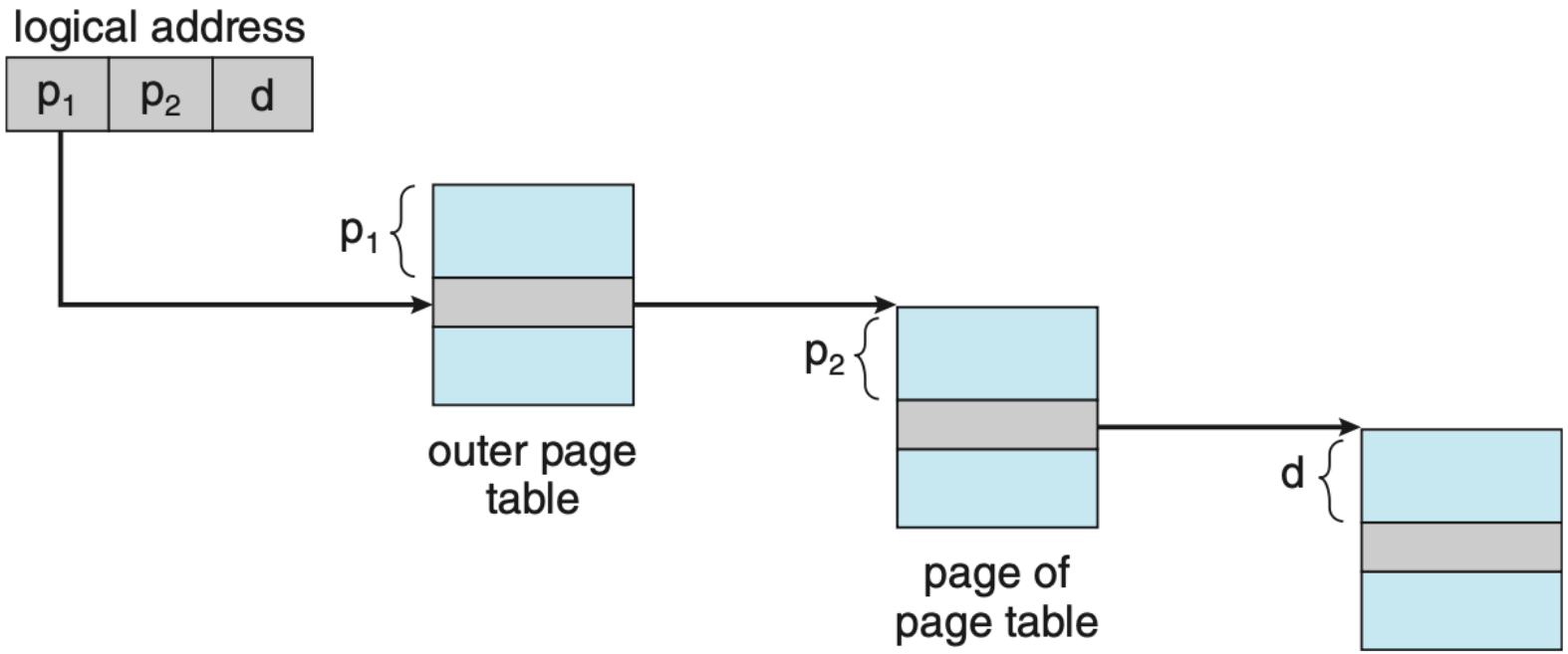
- Consider a typical 32-bit virtual address space with page size of 4KB.
- Virtual Address is divided into:
  - VPN (20 bits) + VPO(12 bits)
- Since the page table is paged, the 20-bit VPN is further divided into:
  - A 10-bit outer page number
  - A 10-bit inner page offset (an inner page is composed of 2^{10} entries with 4-byte address pointer for each entry)
- Thus, a Virtual Address is as follows:



where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table.

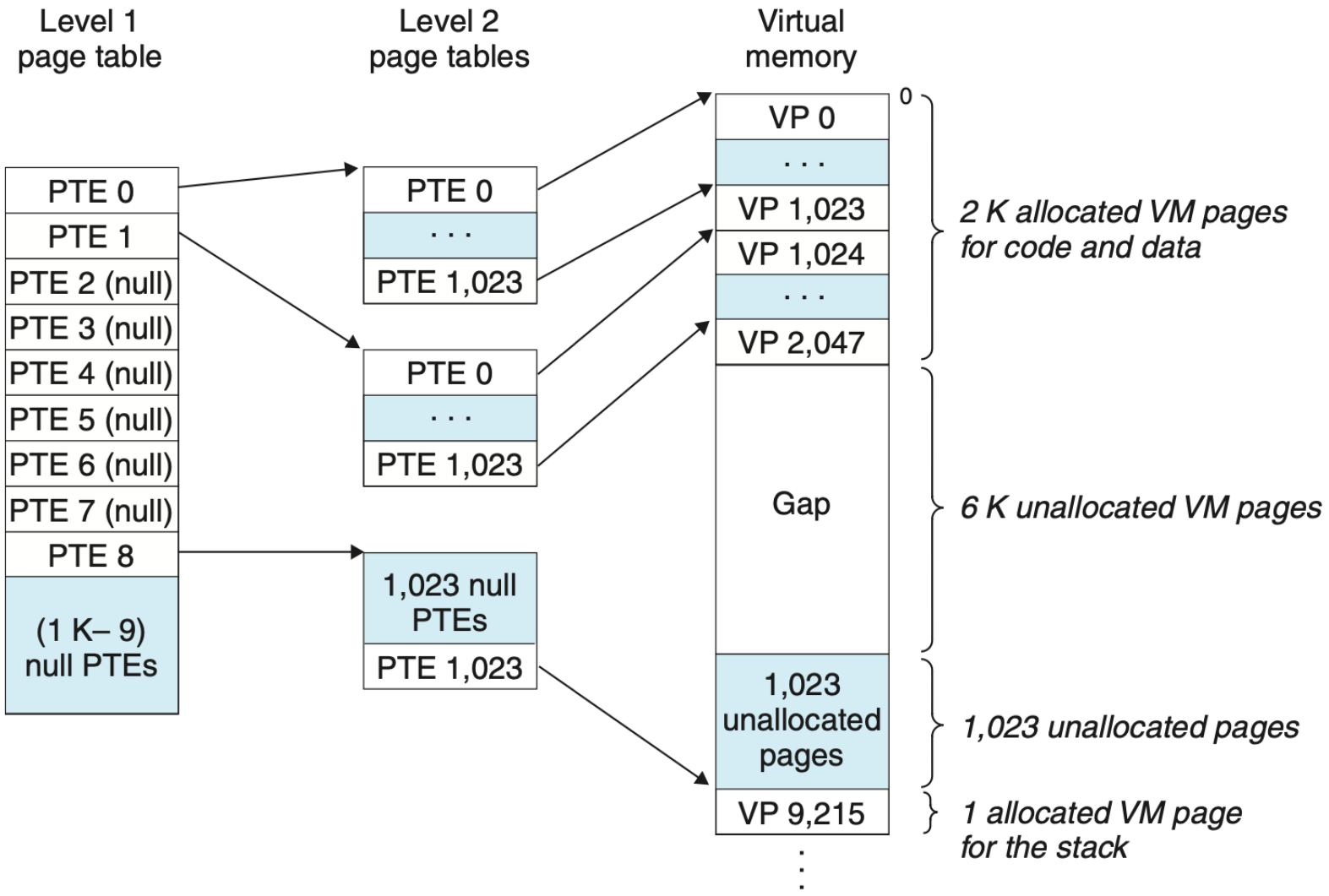
Multi-Level Page Table

- Two-Level Paging Scheme



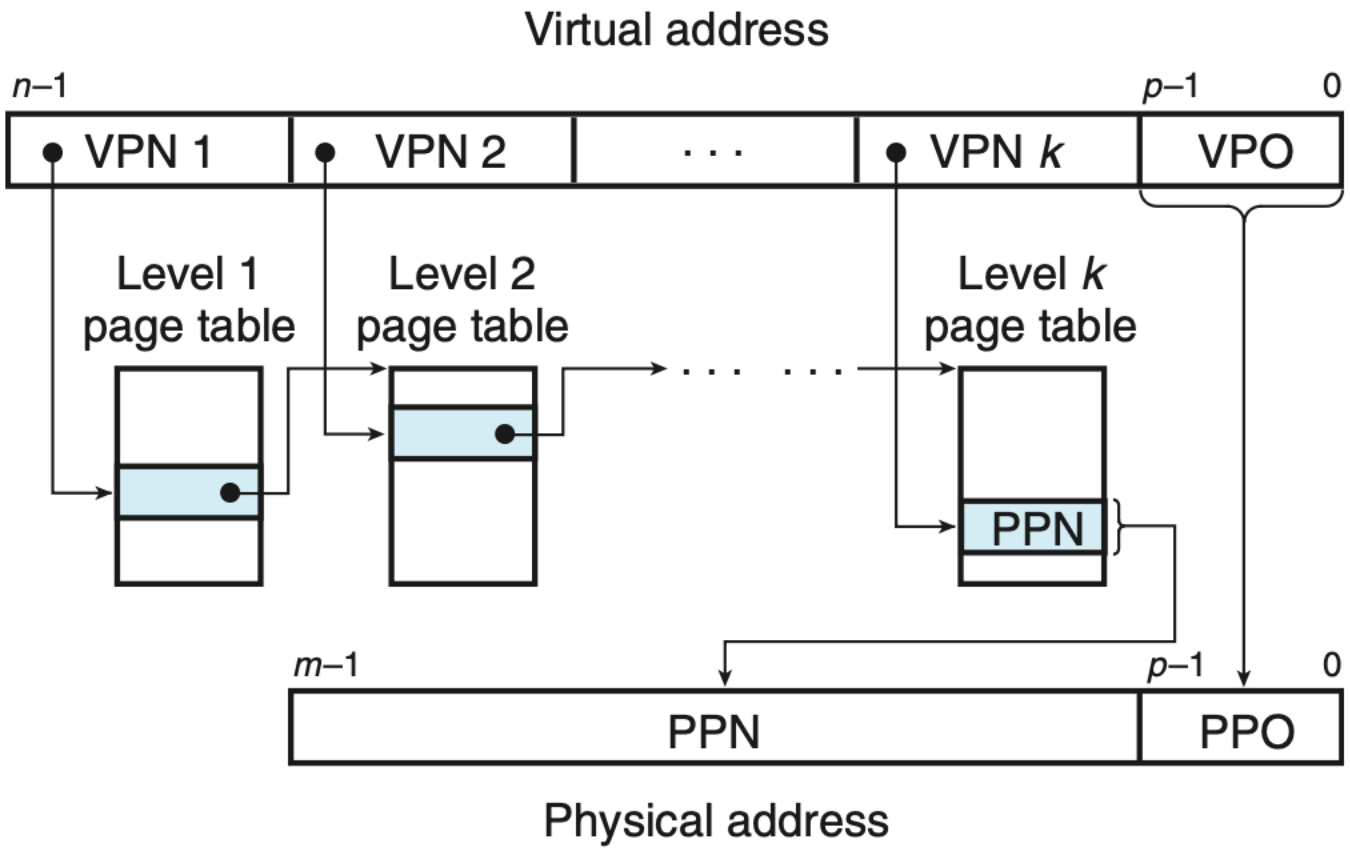
Multi-Level Page Table

- Two-Level Paging Scheme



Multi-Level Page Table

- k-level Paging Scheme



Multi-Level Page Table

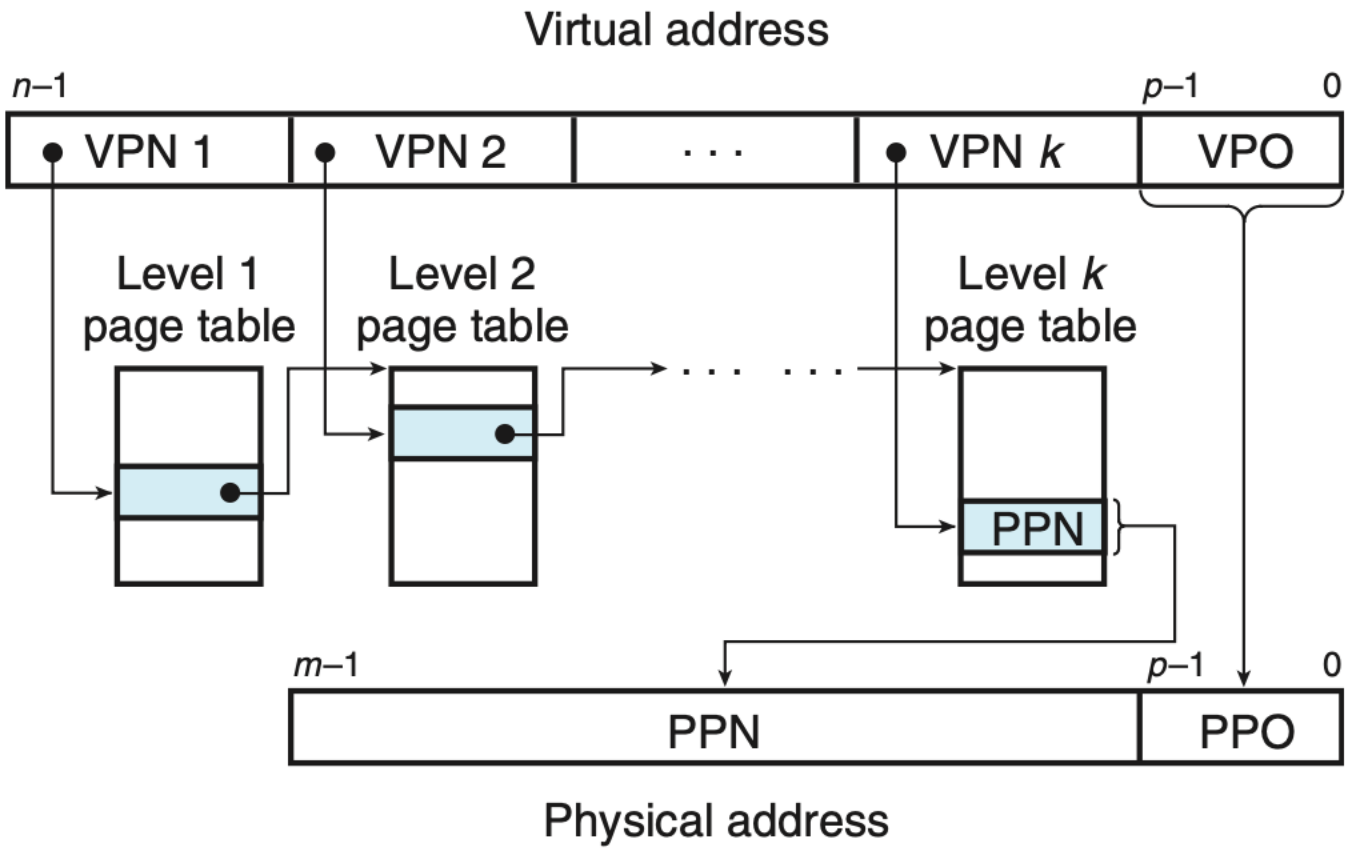
- **Question:** How many memory accesses are required for referencing a paged memory using a **One-Level** (Linear) Page Table?
  - 1 (for Page Table access for the **PA**) + 1 (for actual **data** access)

- **Question:** How many memory accesses are required for referencing a paged memory using a **Two-Level** Page Table?
  - 1 (for Page Directory access for the **Level 2 Page Table Address**)
    + 1 (for level 2 Page Table access for the **PA**)
    + 1 (for actual **data** access)

- **Question:** How many memory accesses are required for referencing a paged memory using a **Four-Level** Page Table?
  - 5

Multi-Level Page Table

- For a **K-Level Page Table,** it requires up to **K** memory accesses in order to translate a **Virtual Address** into a **Physical Address.**



Multi-Level Page Table

- For a **K-Level Page Table**, it requires up to **K** memory accesses in order to translate a **Virtual Address** into a **Physical Address**.
- **Multi-Level Page Table** is a classic example of **time-space trade-off**.
  - We wanted smaller tables (*less **space***)...
  - but not for free ⇒ one **extra** memory access **per** level (*more **time***)...
- However, thanks to our friend **TLB**, such extra memory accesses only occur in case of **TLB** miss.
  - **TLB** hit: **1** memory access (**PA** can be obtained from **TLB** without querying the **Page Table in memory**)
  - **TLB** miss: **K+1** memory accesses (**K** for {VA to PA}, **1** for {PA to data})

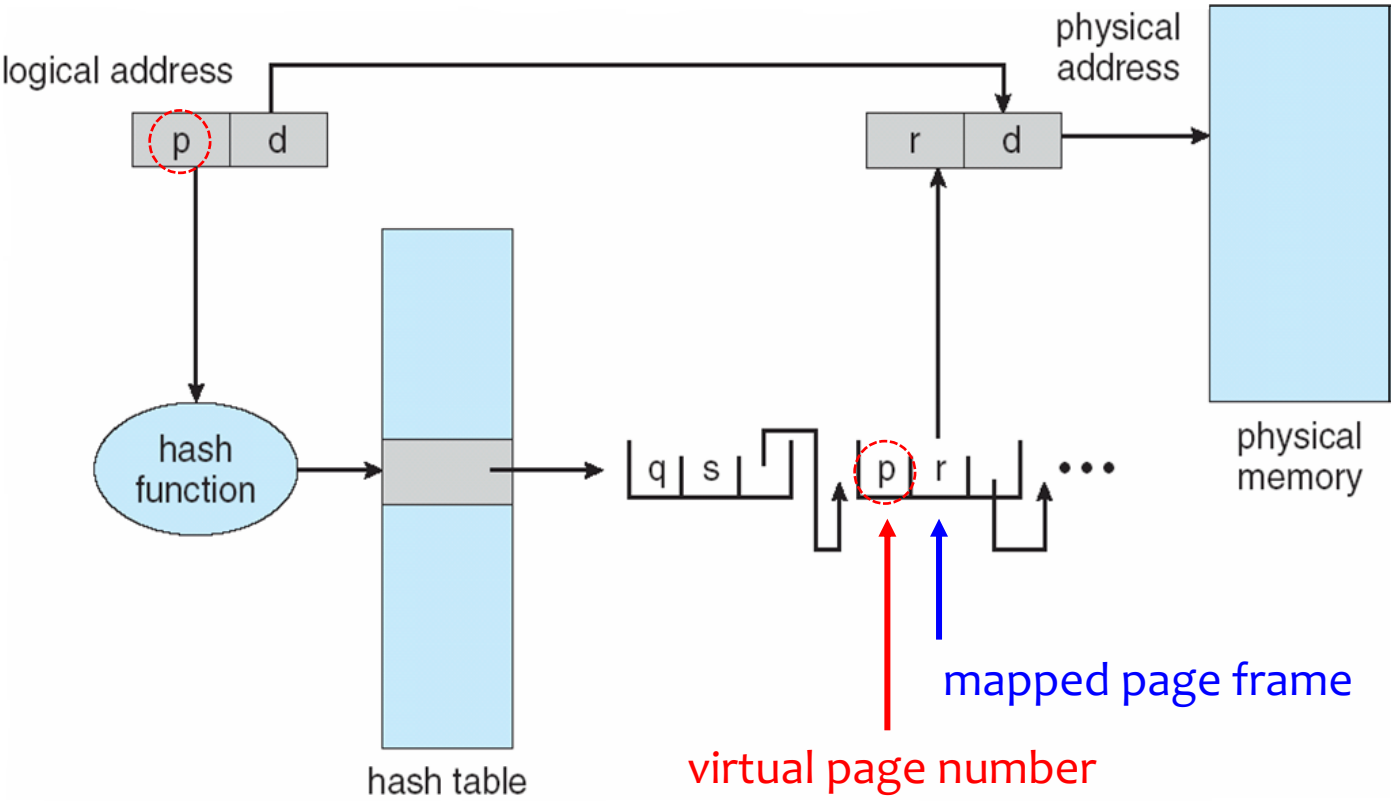
TLB with Multi-Level Page Table

- TLB hit ratio: the percentage of times that the page number of interest is found in the TLB.
  - TLB hit ratio can generally be up to 90%+.
  - A 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
  - Suppose that it takes 10 nanoseconds to access memory with a **Two-Level** Page Table scheme.
    - 1) TLB **hit**: memory access time is 10 ns
    - 2) TLB **miss**: memory access time is **30** ns
  - **Effective Access Time** (**EAT**):
    - **EAT** = 80% × 10 + 20% × 30 = 14 ns (compared to 12ns for 1-Level)
    - implying 40% slowdown in memory access time.
  - Consider a more realistic hit ratio of 95%:
    - **EAT** = 95% × 10 + 5% × 30 = 11 ns (compared with 10.5ns for 1-Level)

Hashed Page Table

- Common in address space > 32 bits.
- The Virtual Page Number is hashed into a page table.
  - This page table contains a chain of elements hashing to the same location (separate chaining solution for hash collision).
- Each element of the chain table contains:
  - 1. The Virtual Page Number (**VPN**)
  - 2. The value of the mapped Physical Frame Number (**PFN**)
  - 3. A pointer to the next element
- Virtual Page Numbers are compared in this chain for a match
  - If a match is found, the corresponding physical frame is extracted.

Hashed Page Table



Hashed Page Table

- Variation for 64-bit addresses is clustered page tables.
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1.
  - Therefore, a single page-table entry (**PTE**) can store the mappings for multiple Physical Frame Numbers.
  - Especially useful for sparse address spaces (where memory references are non-contiguous and scattered)

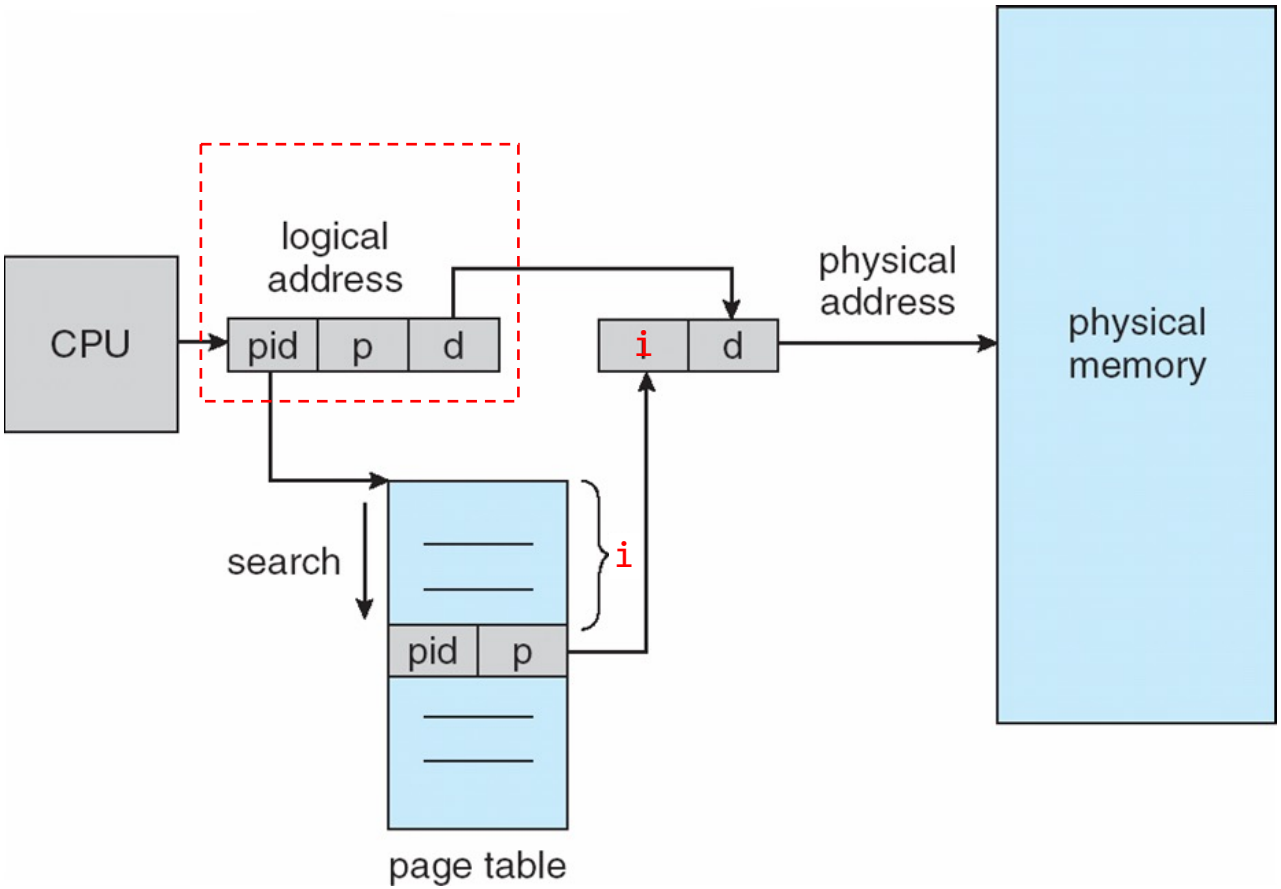
Inverted Page Table

- In most OSes, a separate page table is maintained for each process.
  - ⇒ for **N** processes, there will be **N** number of page tables.
- For large processes, there would be many pages and for maintaining info about these pages, there would be too many entries in their page tables which itself would occupy a lot of the memory.
- Hence, memory utilization is not efficient as a lot of memory is wasted in maintaining page tables itself.
- **Solution:** Use Inverted Page Tables.
  - An inverted page table has one entry for each physical frame of memory.

Inverted Page Table

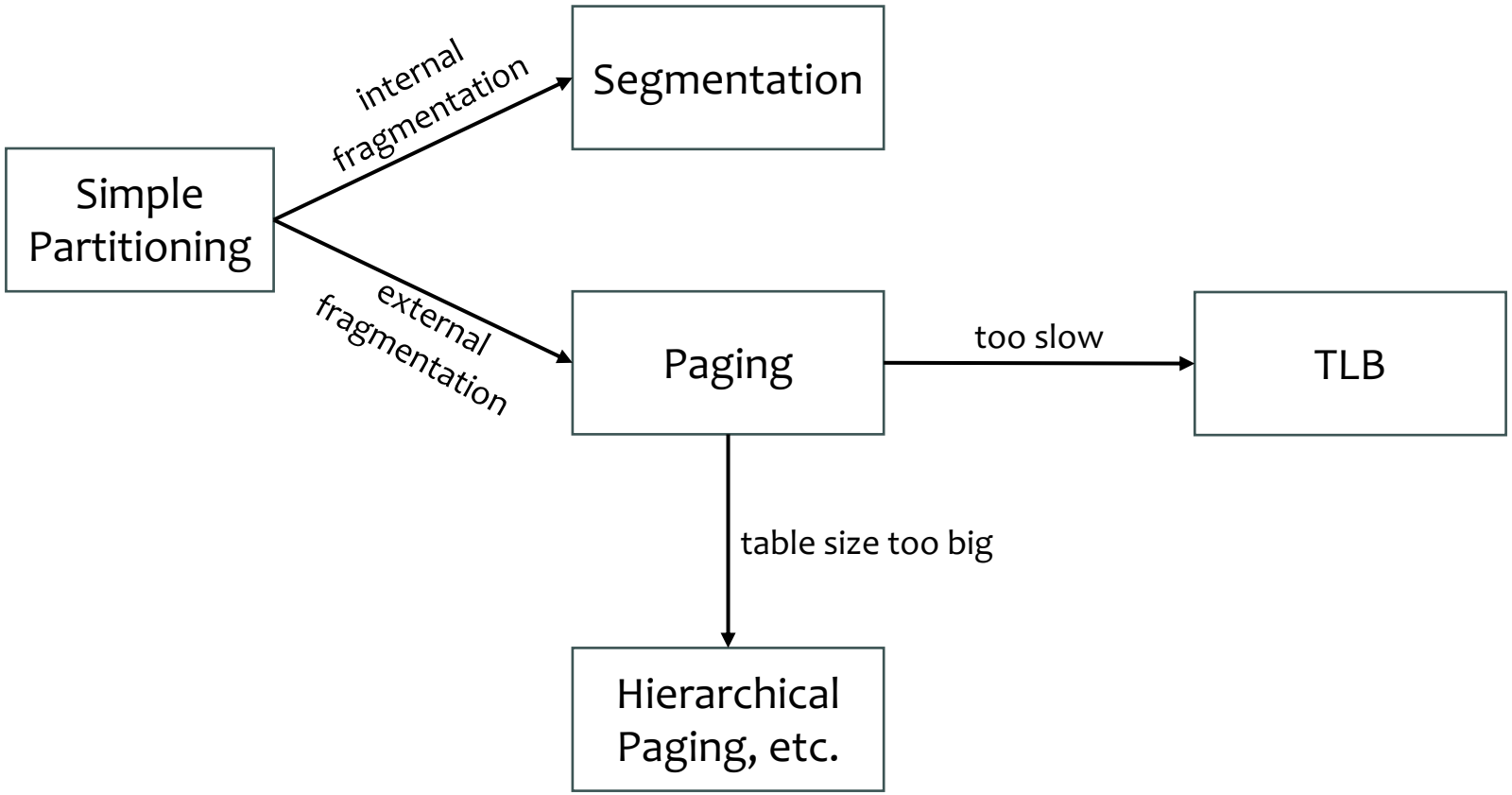
- Rather than each process having a page table and keeping track of all possible virtual pages, track all physical pages instead.
- Entries of an Inverted Page Table (**倒置页表**) consists of the virtual address of the page stored in that physical memory location, with information about the process that owns that page.
  - One entry for each physical frame of memory
    - e.g., `<pid, page_number>`
    - used by 64-bit Ultra SPARC, PowerPC, ...
- This scheme decreases memory needed to store each page table, but increases time needed to search the table when a page is referenced.
- Use hash table to limit the search to one (or at most a few) PTEs.
  - TLB can accelerate access.
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address.

Inverted Page Table



Inverted page table architecture

Summary


Simple Partitioning
internal fragmentation
Segmentation
external fragmentation
Paging
too slow
TLB
table size too big
Hierarchical Paging, etc.

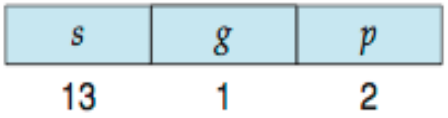
Example

- **Intel IA-32 and IA-64 Architecture**
  - Dominant industry chips
  - Pentium CPUs are 32-bit and called IA-32 architecture
  - Current Intel CPUs are 64-bit and called IA-64 architecture

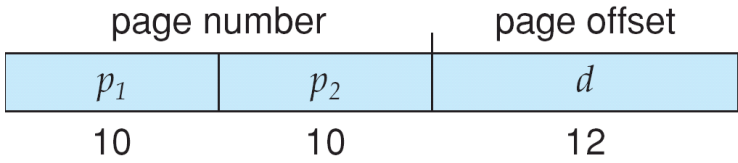
- **Intel IA-32 Architecture**
  - Supports both **segmentation** and **segmentation with paging**.
    - Each segment can be **4GB**
    - Up to **16K** segments per process
    - Divided into two partitions
      - First partition of up to **8K** segments are private to process
        - Kept in **local descriptor table** (**LDT**)
      - Second partition of up to **8K** segments shared among all processes
        - Kept in **global descriptor table** (**GDT**)

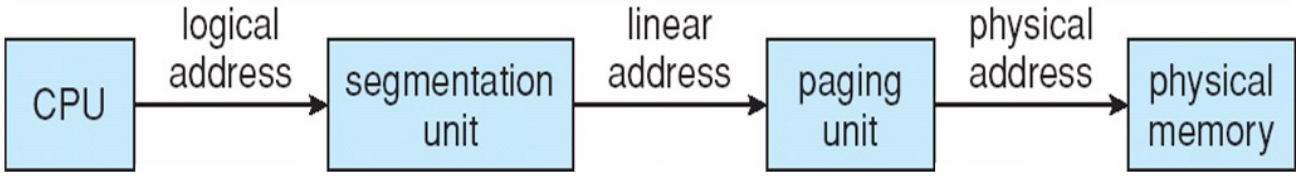
Intel IA-32 Architecture

- CPU generates logical address
  - Selector given to **segmentation unit**
    - which produces linear addresses

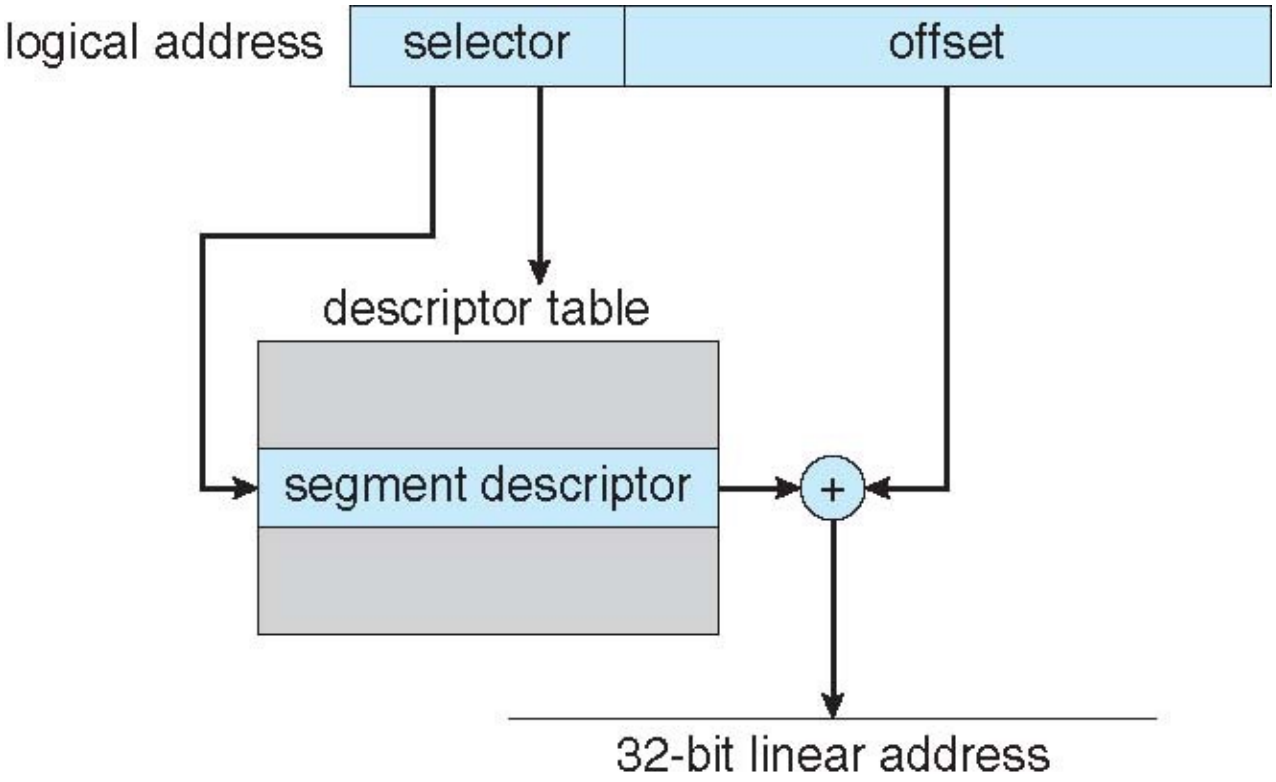


  - Linear address given to **paging unit**
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Page sizes can be **4KB** or **4MB**

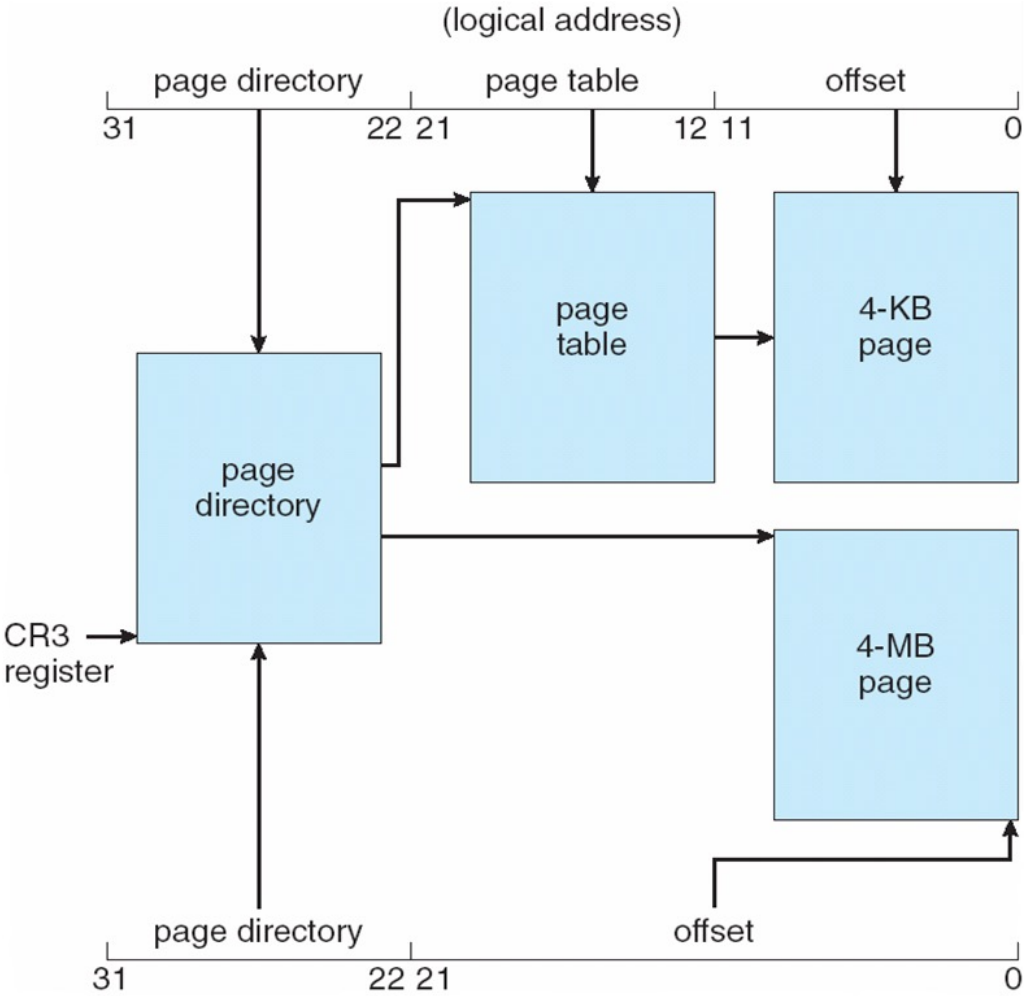




Intel IA-32 Segmentation

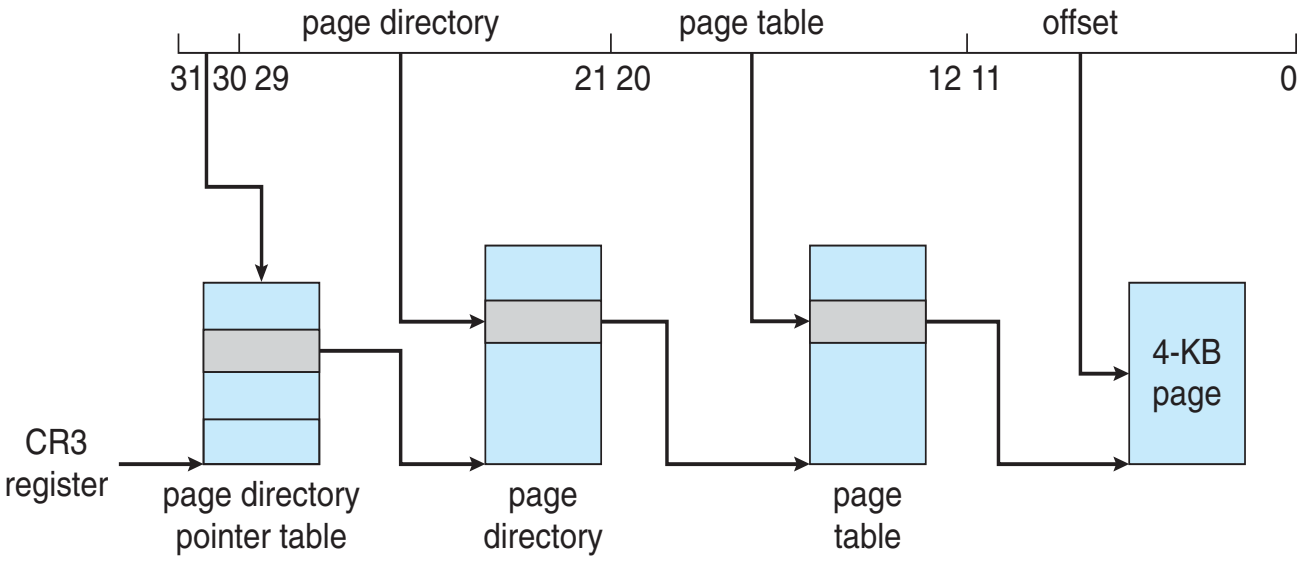


Intel IA-32 Paging



Intel IA-32 Page Address Extension

- 32-bit address limits ⇒ **4GB** limit of memory space
  - this led Intel to create **Page Address Extension** (**PAE**)
  - Paging went to a **3-Level** scheme (from the original 2-Level)
  - Top two bits refer to a **Page Directory Pointer Table**
  - **Page Directory** and **Page Table** entries moved to **64 bits** in size
  - Net effect is increasing address space to 36 bits ⇒ **64GB** of physical memory



Exercises

- It is **highly recommended** to finish all the exercises of Chapter 9

base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

9.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?

9.6 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

9.7 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

9.8 The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the BTV operating system?

9.9 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

9.10 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

Chapter 9 Exercises

9.11 Explain the difference between internal and external fragmentation.

9.12 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?

9.13 Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

9.14 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation

b. Paging

9.15 Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:

a. External fragmentation

b. Internal fragmentation

c. Ability to share code across processes

9.16 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?

9.17 Explain why mobile operating systems such as iOS and Android do not support swapping.

9.18 Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?

9.19 Explain why address-space identifiers (ASIDs) are used in TLBs.

9.20 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation

b. Paging

9.21 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?

a. 21205

b. 164250

c. 121357

d. 16479315

e. 27253187

9.22 The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the MPV operating system?

9.23 Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

9.24 Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

9.25 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

9.26 What is the purpose of paging the page tables?

9.27 Consider the IA-32 address-translation scheme shown in Figure 9.22.

a. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory translation?

Thank you!