

Virtual Memory (II)

Address Translation

Outline

- Overview
- Segmentation
- Paging: Introduction
- Paging: Faster Translations (TLBs)
- Paging: Smaller Tables

Overview

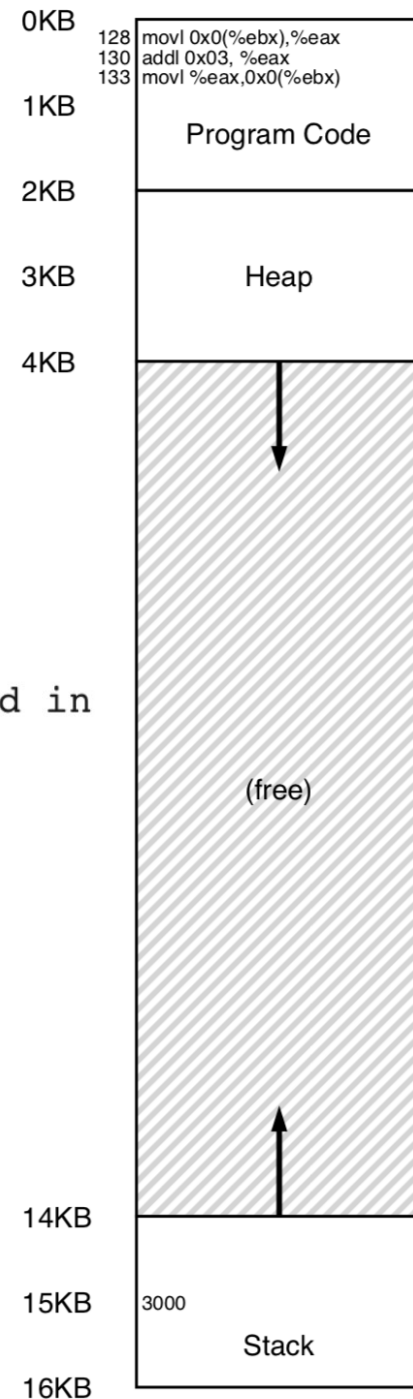
- Address Translation
 - Process 看到、使用的地址都是 virtual address
 - 真正访问硬件前，需要转化为 physical address

Overview

- An Example

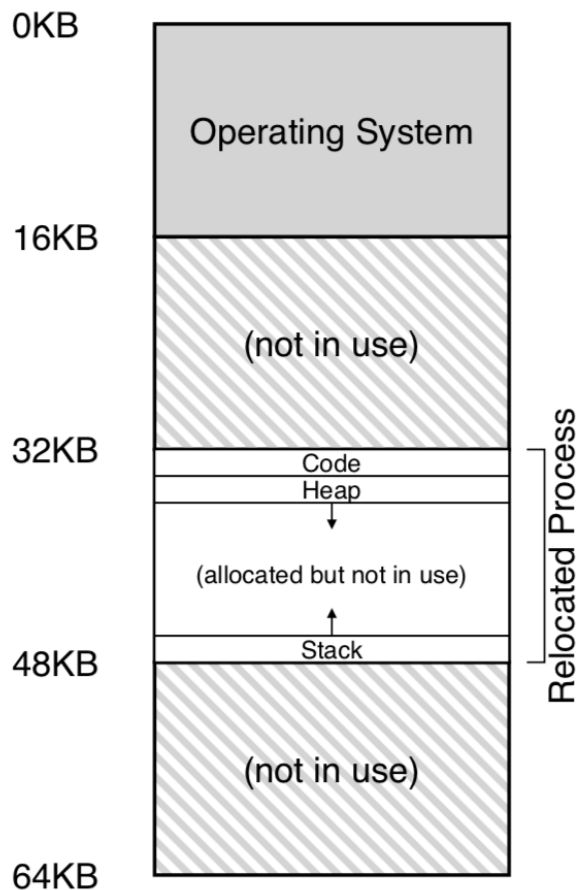
```
void func()  
    int x;  
    ...  
    x = x + 3; // this is the line of code we are interested in
```

```
128: movl 0x0(%ebx), %eax    ;load 0+ebx into eax  
132: addl $0x03, %eax      ;add 3 to eax register  
135: movl %eax, 0x0(%ebx)  ;store eax back to mem
```



Overview

- 进程看到的地址不是真实的



**Physical
Address**

**Virtual
Address**



16KB

Overview

- Address Translation 背后的操作是什么？
 - Transparent
 - 程序操作的地址，当作 va 或 pa 都能运行
 - Efficient
 - 速度要很快，因为访存是个很常见的操作
 - 所以，要借用快速的硬件
 - MMU：Memory Management Unit
 - Load/store 指令时，CPU 利用 MMU 去做 address translation
 - 但 OS 软件也有参与，因为 MMU 做地址翻译的元数据要放在内存里，这块内存空间是 OS 管理和提供的（MMU 是 CPU 的一部分）
 - 进程没有权限访问 PA，但也不是借用 system call 陷入内核态获取权限（因为太慢），而是 CPU 直接用硬件解决

Overview

- 具体的硬件解决方案
 - 1950's : base and bounds/dynamic relocation
 - 需要两个硬件寄存器：
 - Base
 - Bounds
 - 因此，能够找到对应的物理内存，而且限制用户只能访问自己的内存空间（bound 之内）

Overview

- An example

```
128: movl 0x0(%ebx), %eax
```

- PC=128
- 取指令: $PA=PC+base = 128+32K = 32896$
- 硬件按照 PA 去取指令, 放到 CPU 里的指令寄存器
- CPU 执行该指令
- 取数字的地址 $=0+ \%ebx+base$
- ...

Overview

- 对于其中的 bounds
 - 硬件在访问内存前，首先检查是否越界
 - 如果不越界，正常访问
 - 如果越界，则 CPU 会抛出 exception，然后进入 exception handler 的代码
 - 一般该进程会被终止

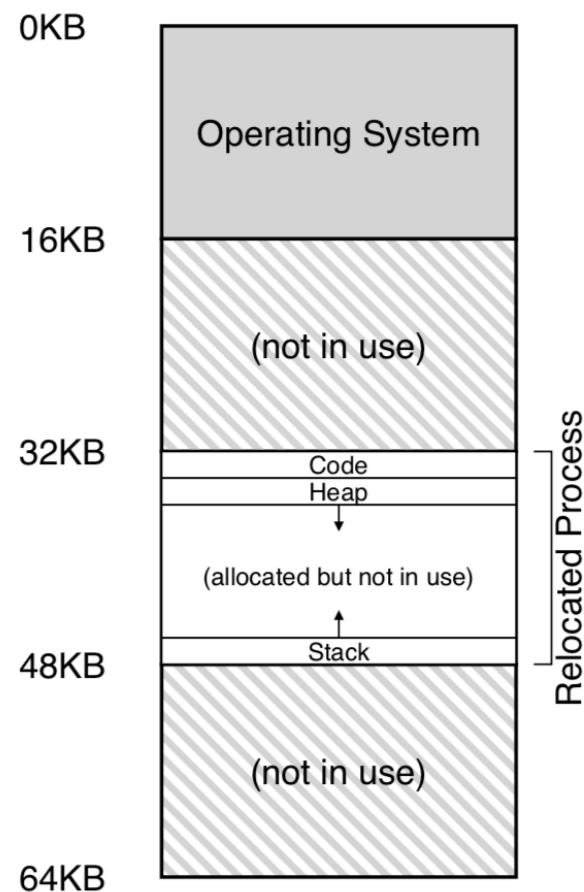
Overview

- Example Translations
 - process 的地址空间是 4KB
 - 位于物理地址 16KB
 - Virtual Address 0 → Physical Address 16 KB
 - VA 1 KB → PA 17 KB
 - VA 3000 → PA 19384
 - VA 4400 → Fault (out of bounds)

Segmentation

Segmentation

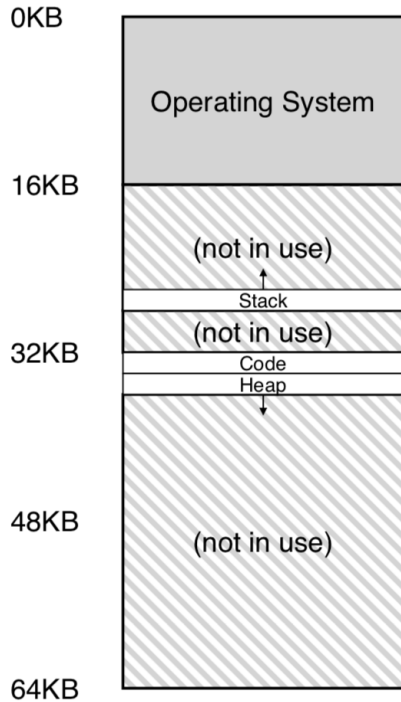
- 前面所述 base-bounds 方法的问题
 - heap 和 stack 区域之间的未使用空间也要分配物理内存
 - 严重浪费



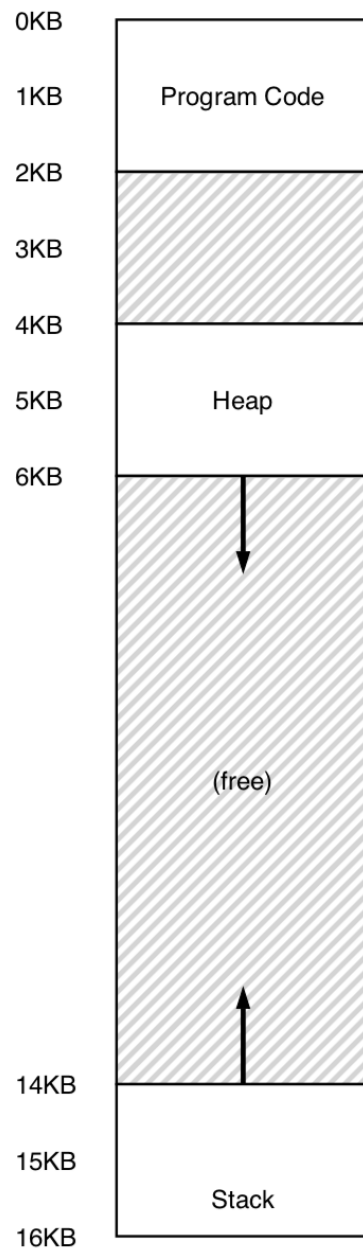
Segmentation

- 解决思路
 - 不再把一个 process 的内存空间映射为一段物理内存
 - 而是每个区域独立进行映射
 - Code
 - Heap
 - Stack

Physical
Address



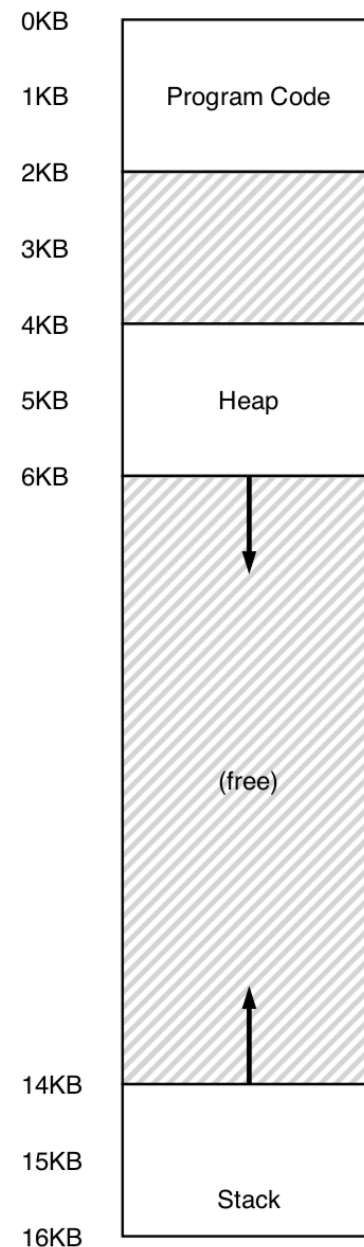
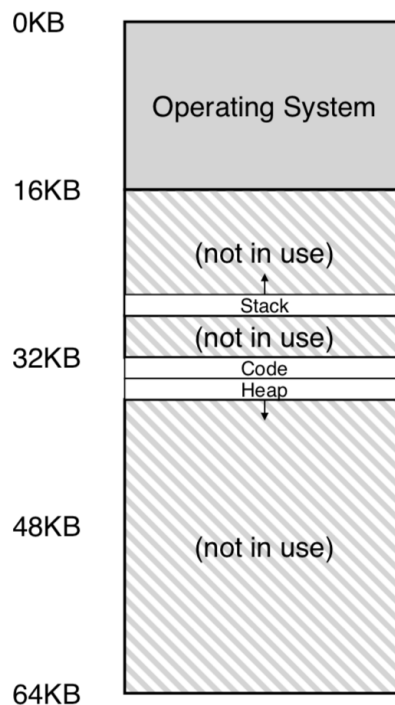
Virtual
Address
SS



Segmentation

- 地址翻译: $VA=100$ (code)
 - Code 区域
 - $PA=32K+100=32868$
- 如果访问 $VA=7K$
 - 不在任何一个 segment 中
 - Segmentation fault

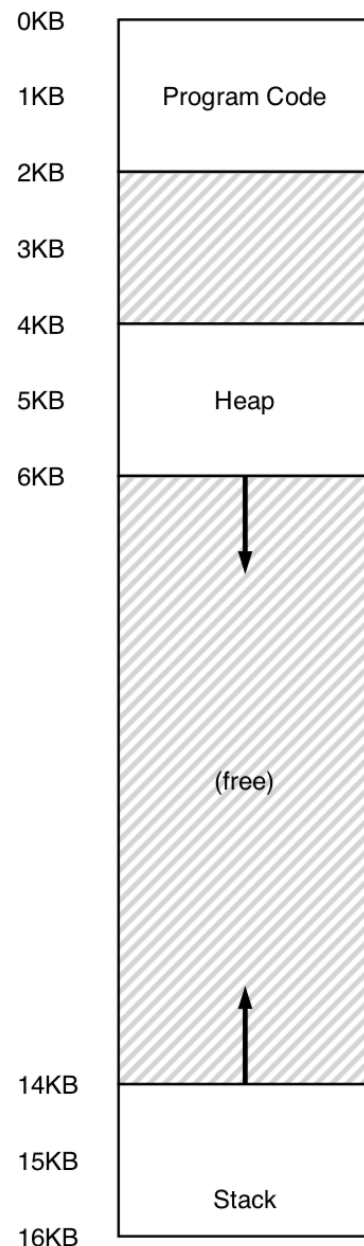
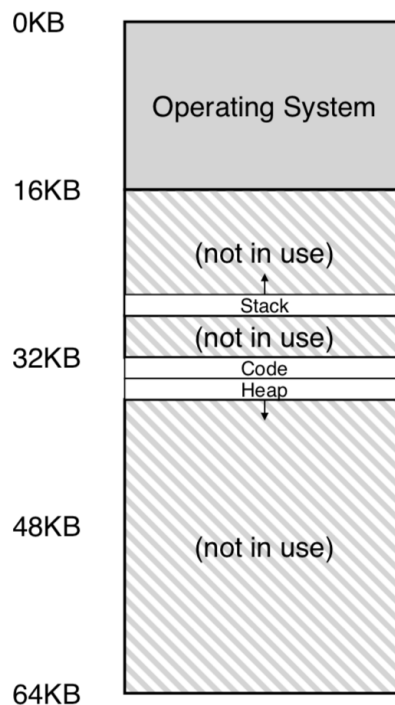
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Segmentation

- $VA=4200$
 - Heap 区域
 - $VA=4K+104=heap$ 区域偏移 104
 - $<2K$ (size)
 - $PA=34K+104=34920$

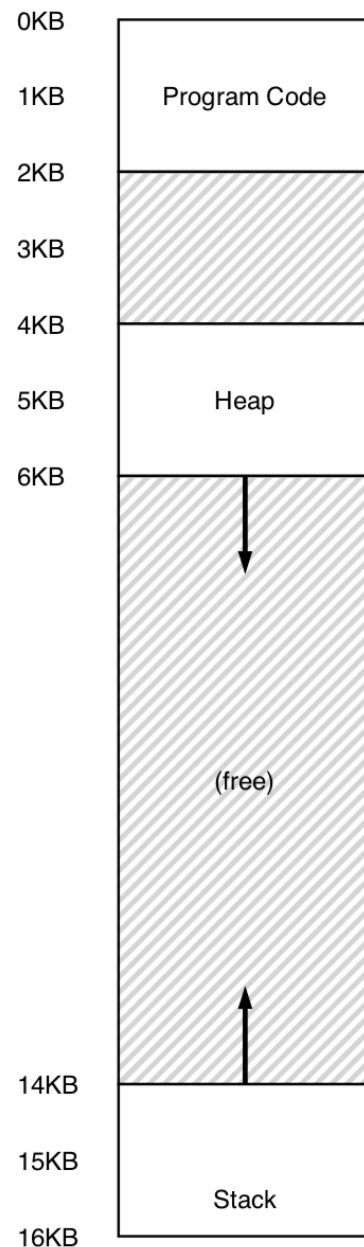
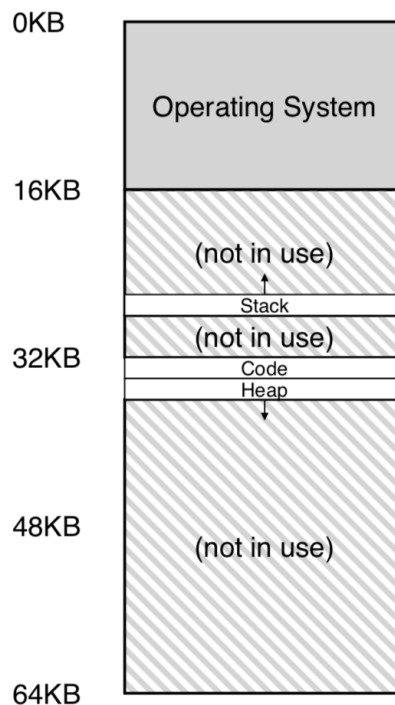
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Segmentation

- 以上为人工计算
- 那么 CPU 拿到一个 VA，怎么知道它是属于那个 segment 呢？

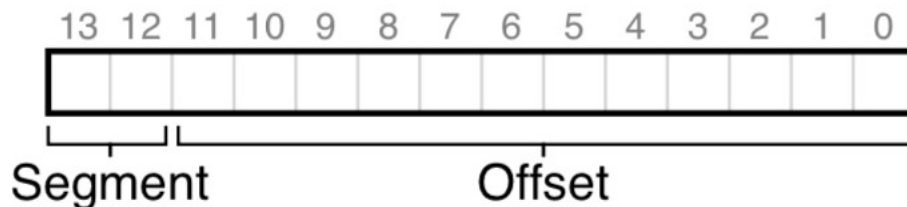
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Segmentation

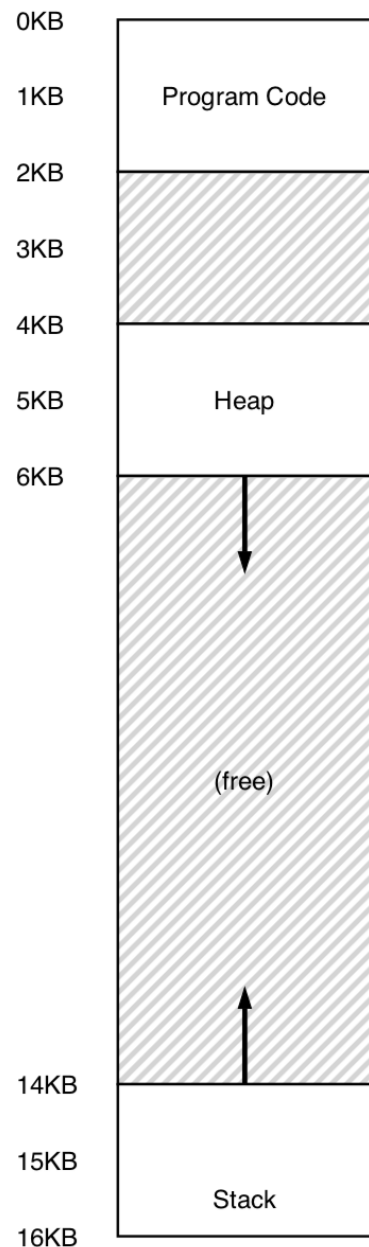
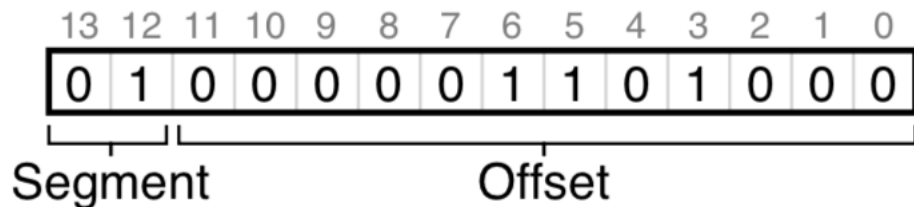
- #1. explicit approach
 - 在 VA 中预留几个 bits 标记属于哪个 segment
 - 例如 VAX/VMS 系统
 - 如果有 3 个 segment ， 那么需要 2 个 bits
 - 最高两位 00-code
 - 01-heap
 - 1x-stack

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Segmentation

- 14 位 VA 中
 - 前两位为 00 ，即 VA 最开始的 code
 - 前两位为 01 ，即 $VA=2^{12}=4K$ 开始
- An example
 - $VA = 4200 = 4K + 104$
 - 前两位是 01 ，是 heap
 - $PA = 34K+104$



Segmentation

- Code
 - SEG MASK would be set to 0x3000
 - SEG SHIFT to 12
 - OFFSET MASK to 0xFFF.

```
1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

Segmentation

- 由于 2 个 bit ， 3 个 segment
 - 浪费了一个状态
 - 因此，有些系统将 code 和 heap 放在一个 segment ， 因此只用 1 个 bit

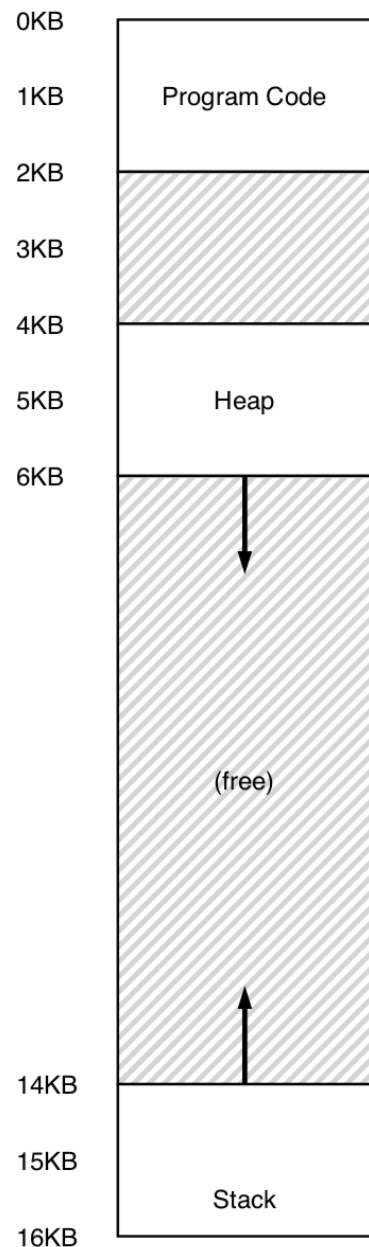
Segmentation

- #2. **implicit** approach
 - 硬件通过看地址如何形成的，来判断属于那个 segment
 - 例如，通过 PC 形成的地址，那就是在 code 段内
 - 如果通过 stack or base pointer ，那就是在 stack 段
 - 其他的地址都在 heap 段

Segmentation

- Stack 段怎么办?
 - 它的空间倒着增长
 - 最小的地址不确定
- Solution
 - 首先需要硬件多一点支持，记录 segment 的地址是否负增长
 - 如果是负增长，那么 Pa 计算方法有些区别

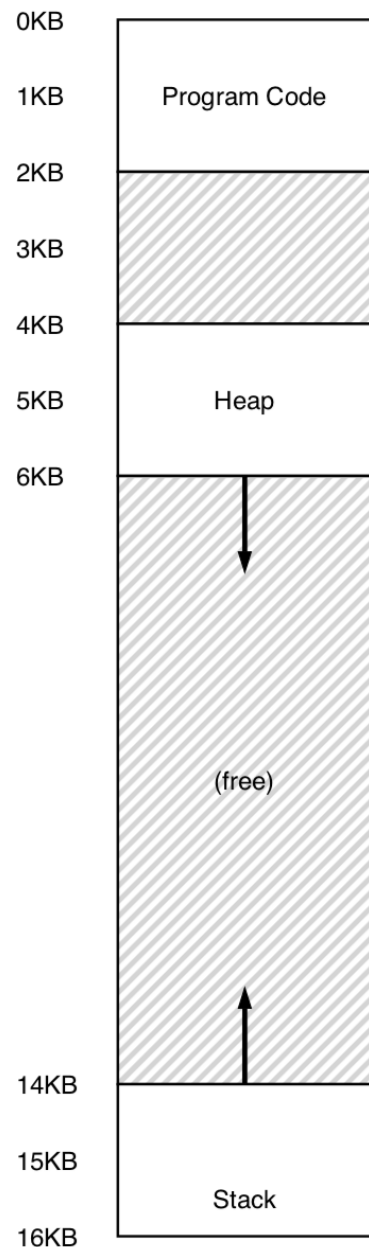
Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0



Segmentation

- 负增长 segment 地址翻译
 - VA=15K
 - 11 1100 0000 0000 (0x3C00)
 - 最高的两位 11 确定是 stack segment
 - Offset = 1100 0000 0000 = 3K
 - Segment 最大可能大小 (4K)-3K=1K，作为实际偏移量，而且是从 stack 首地址开始的负偏移量
 - PA=28K-1K=27K

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0



Segmentation

- Support for sharing
 - 进程之间共享一部分内存
 - 例如代码库（共享代码）
 - Sharing 需要硬件提供一点额外的支持
 - 标记 segment 的权限（R/W/E）
 - 例如 code segment 可读可执行，但不可写；heap 和 stack 不可执行

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Segmentation

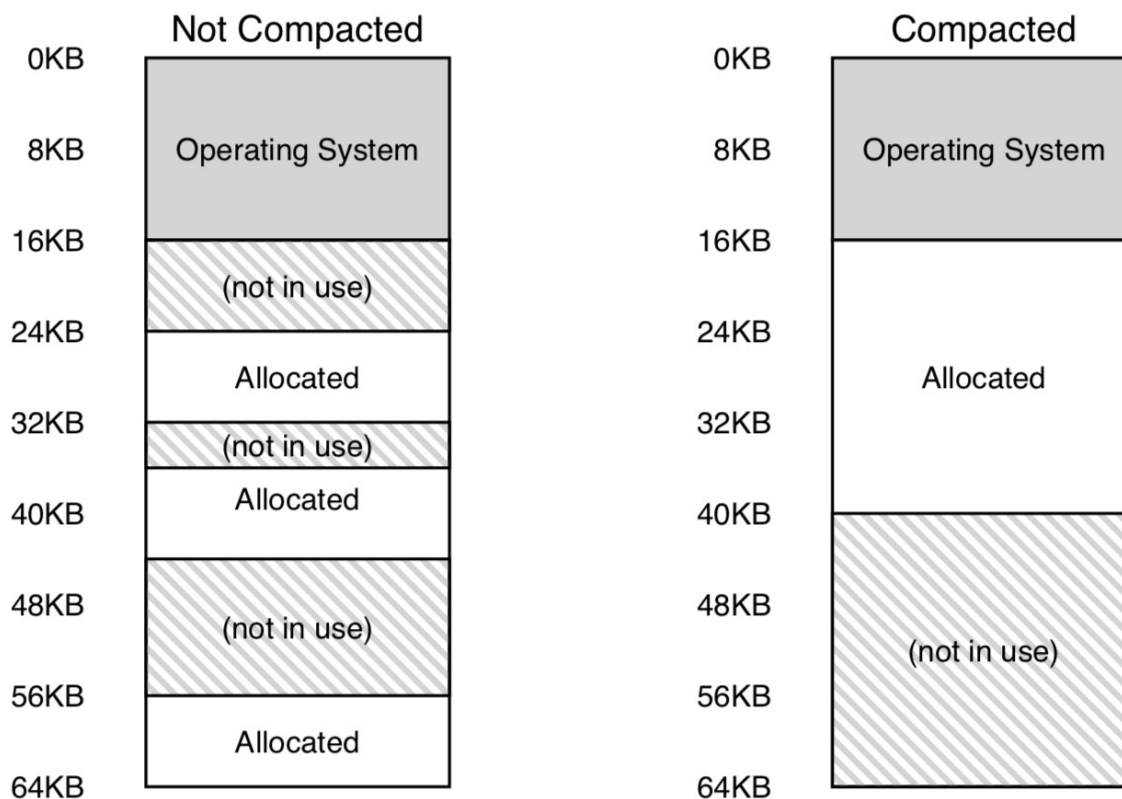
- Fine-grained vs. Coarse-grained Segmentation
 - Coarse-grained
 - Code, heap, stack
 - Fine-grained
 - 例如 Multics
 - 需要更大空间记录 segment 状态
 - Segment table , 放在 memory 中
 - 如早期计算机 Burroughs B5000 支持上千个 segments , compiler 决策把 code 和 data 放到这些 segments 中

Segmentation

- OS Support
 - #1. context switch
 - 每个 segment 的 base, bounds 等信息需要在切换时保存和恢复
 - 大量进程, 每个进程都有自己的一套
 - Kernel space, process 结构体中
 - #2. free space management
 - 随着进程创建和回收, fragmentation 问题
 - Compaction 操作

Segmentation

- OS Support
 - #2. free space management
 - Compaction 操作: expensive



Segmentation

- OS Support
 - #2. free space management
 - Free list (malloc 也是类似问题)
 - **best-fit**
 - **worst-fit**
 - **first-fit**
 - **buddy algorithm**
 - … 上百个策略

“Dynamic Storage Allocation: A Survey and Critical Review” Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles In International Workshop on Memory Management Scotland, United Kingdom, September 1995
A great survey paper on memory allocators.

Segmentation

- Segmentation 的局限性：
 - 空间回收开销；
 - segmentation still isn't flexible enough to support our **fully generalized, sparse** address space.
 - For example, if we have **a large but sparsely-used heap** all in one logical segment, the entire heap must still reside in memory in order to be accessed.

课堂练习

- 一个计算机虚拟内存空间大小为 1MB，物理内存空间大小为 8MB。虚存空间依次分为 4 个 segment（code, data, heap, stack），采用 segmentation 内存管理，用 Va 最高两位标记 segment，有寄存器记录下列信息

segment	base	size	positive
code	1MB	4KB	1
data	7MB	2KB	1
heap	2MB	8KB	1
stack	3MB	16KB	0

- 1) Va 地址多少位?
- 2) VA=2KB, 520KB, 1010KB, 求对应 PA (写出计算过程, 每个转换中的 segment 编号, offset)

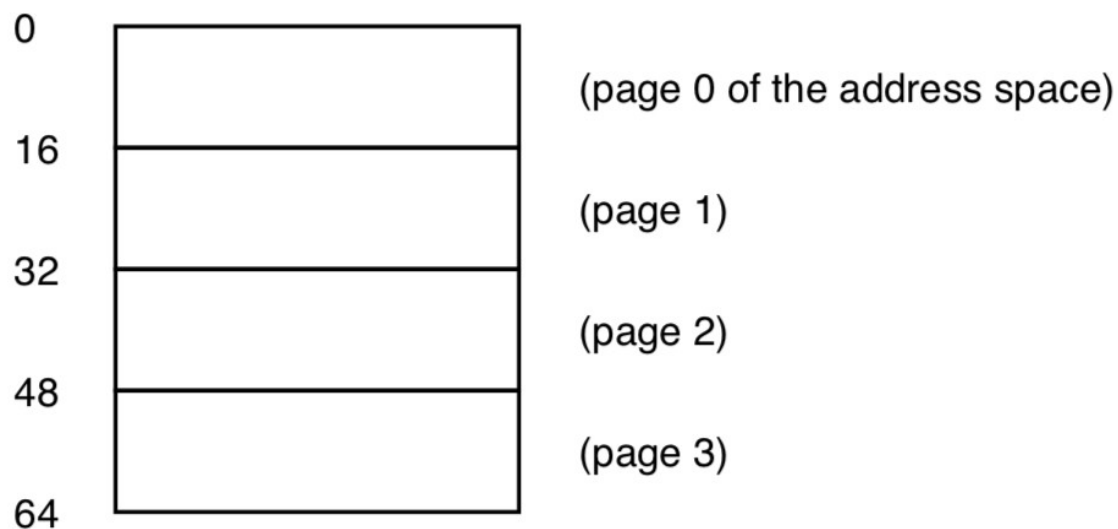
练习答案

- 1) VA : 20-bit
- 2) 地址计算
 - VA=2K , segment=00 , code 段, offset=2K , PA=1MB+2KB
 - VA=520K , segment=10 , heap 段, offset=8K , 内存访问越界
 - VA=1010K , segment=11 , stack 段, 负增长, segment 最大可能大小是 $1\text{MB}/4=256\text{KB}$, offset= $256\text{K}-(1010-768)\text{K}=14\text{KB}$, PA=3MB-14KB

Paging: Introduction

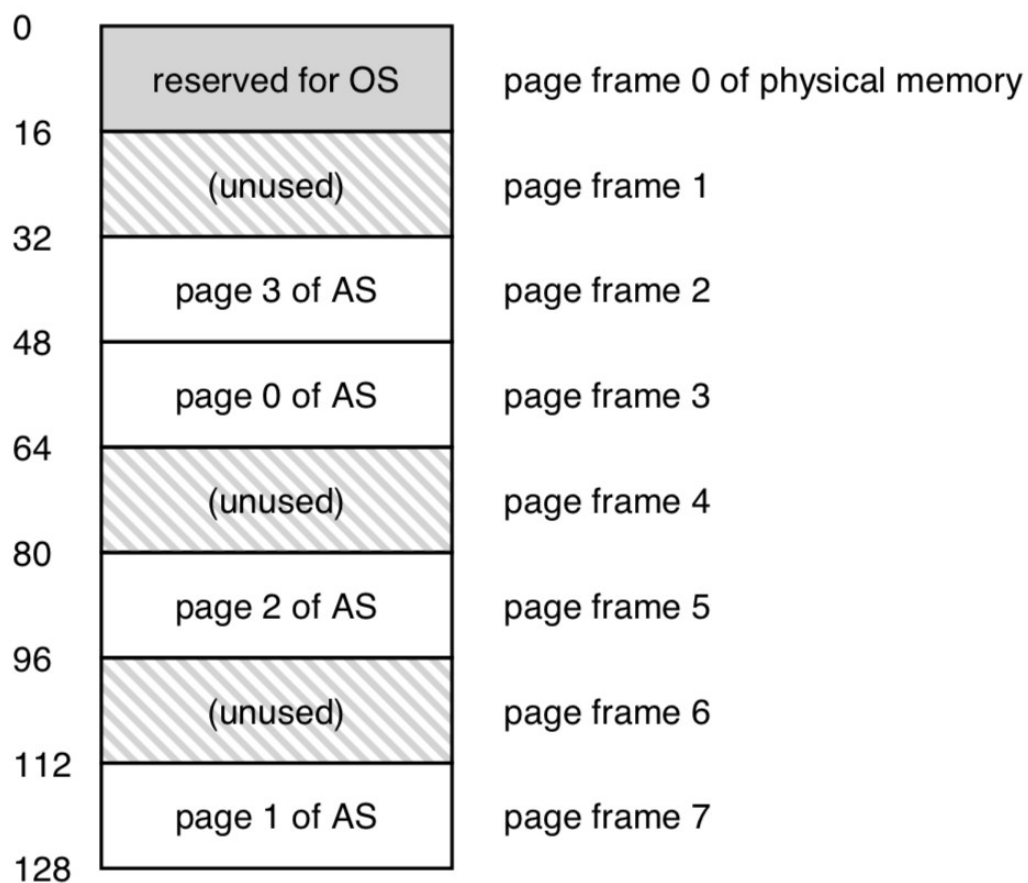
Paging

- VA->PA 的另一种方式（segmentation 之外）
- Paging: 固定大小，映射的单位
- 早期计算机 Atlas（1962）
 - e.g., 64B 的地址空间，page 大小为 16B



Paging

- Va 空间 64B ， Pa 空间 128B



Paging

- Paging 的优点：
 - #1. Flexibility
 - 能够应对大块稀疏空间的问题（解决 segmentation 的问题）
 - #2. simplicity (free-space management)
 - 对齐
 - 连续的 VA 不需要对应连续的 PA
 - 所以，只需要维护所有的 free page ，都可以分配出去，不需要凑成大块连续空间（compaction）

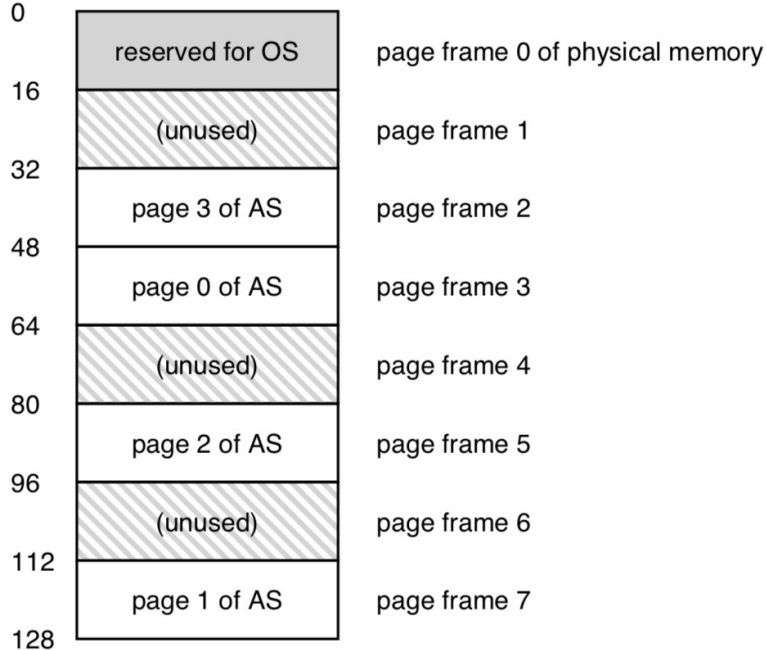
Paging

- Paging 的缺点

- VA- \rightarrow Pa 映射信息比较多（比 segmentation 多很多）

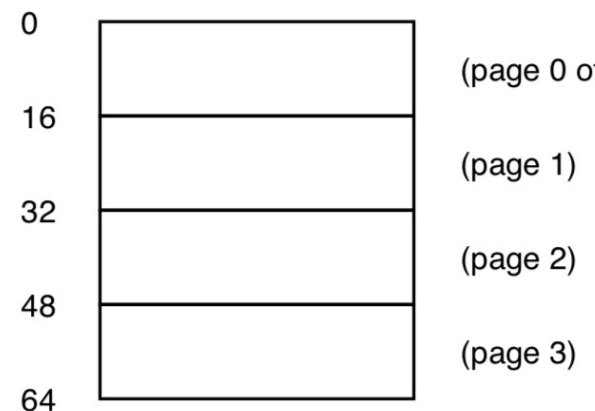
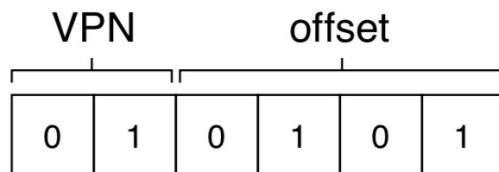
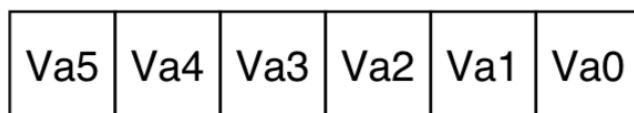
- Page Table

- (Virtual Page 0 \rightarrow 0)
 - (VP 1 \rightarrow PF 7)
 - (VP 2 \rightarrow PF 5)
 - (VP 3 \rightarrow PF 2).



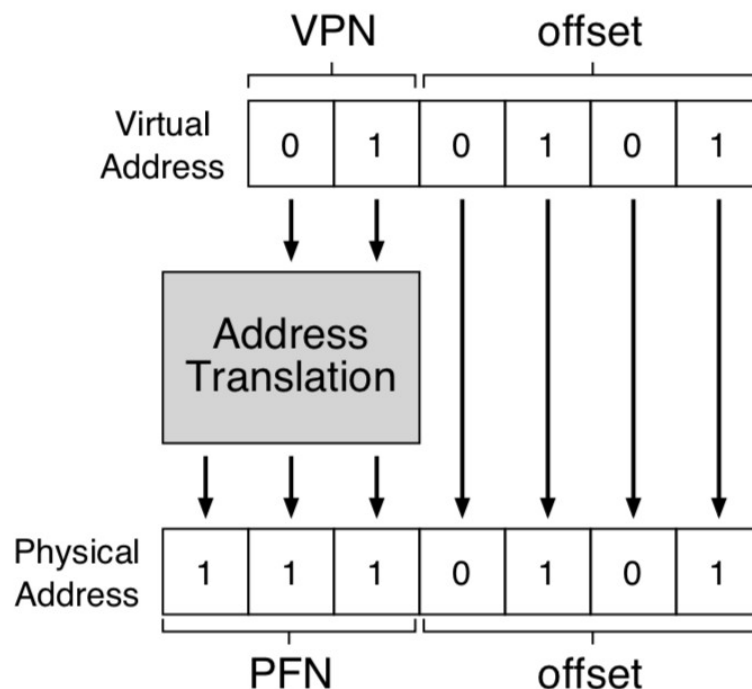
Paging

- Page Table
 - 每个进程一个
 - 一个数组，index 为 VPN，content 为 PA
(**linear page table**)
 - VPN: virtual page number
 - E.g., VA 空间 64 bytes，VA 6 位
 - Page 大小 16B，offset 4 位
 - VPN：2 位，一共 4 个



Paging

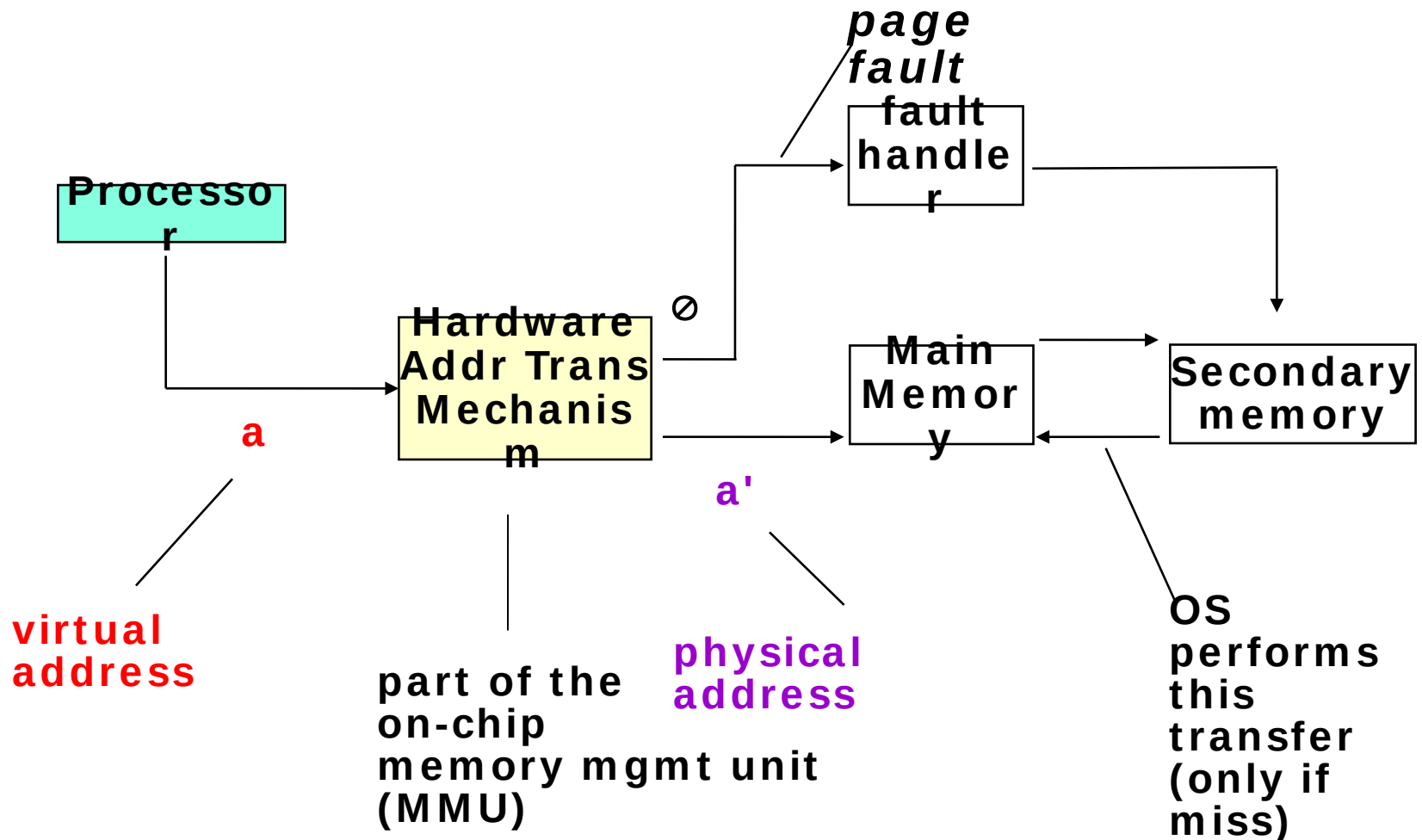
- Address translation
 - VPN -> PFN (physical frame number)
 - Offset 不變



Address Translation

- $V = \{0, 1, \dots, N-1\}$ virtual address space
- $P = \{0, 1, \dots, M-1\}$ physical address ~~space~~
- $N > M$
- Address Translation
- MAP: $V \rightarrow P \cup \{\emptyset\}$ address mapping function
- $\text{MAP}(a) = a'$ if data at virtual address a is present at physical address a' in P
 $= \emptyset$ if data at virtual address a is not

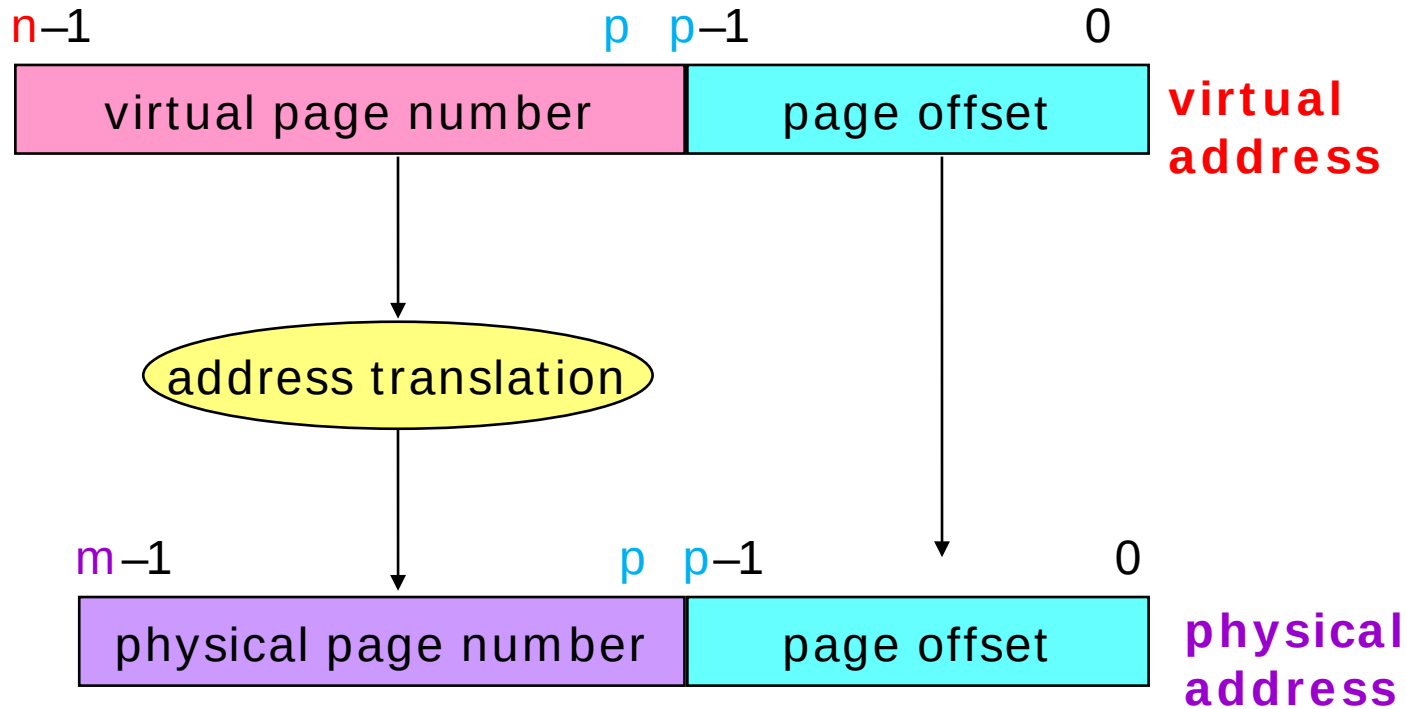
Address Translation



Address Translation

- Basic Parameters
 - $N = 2^n =$ Virtual address limit
 - $M = 2^m =$ Physical address limit
 - $P = 2^p =$ page size (bytes).

Address Translation

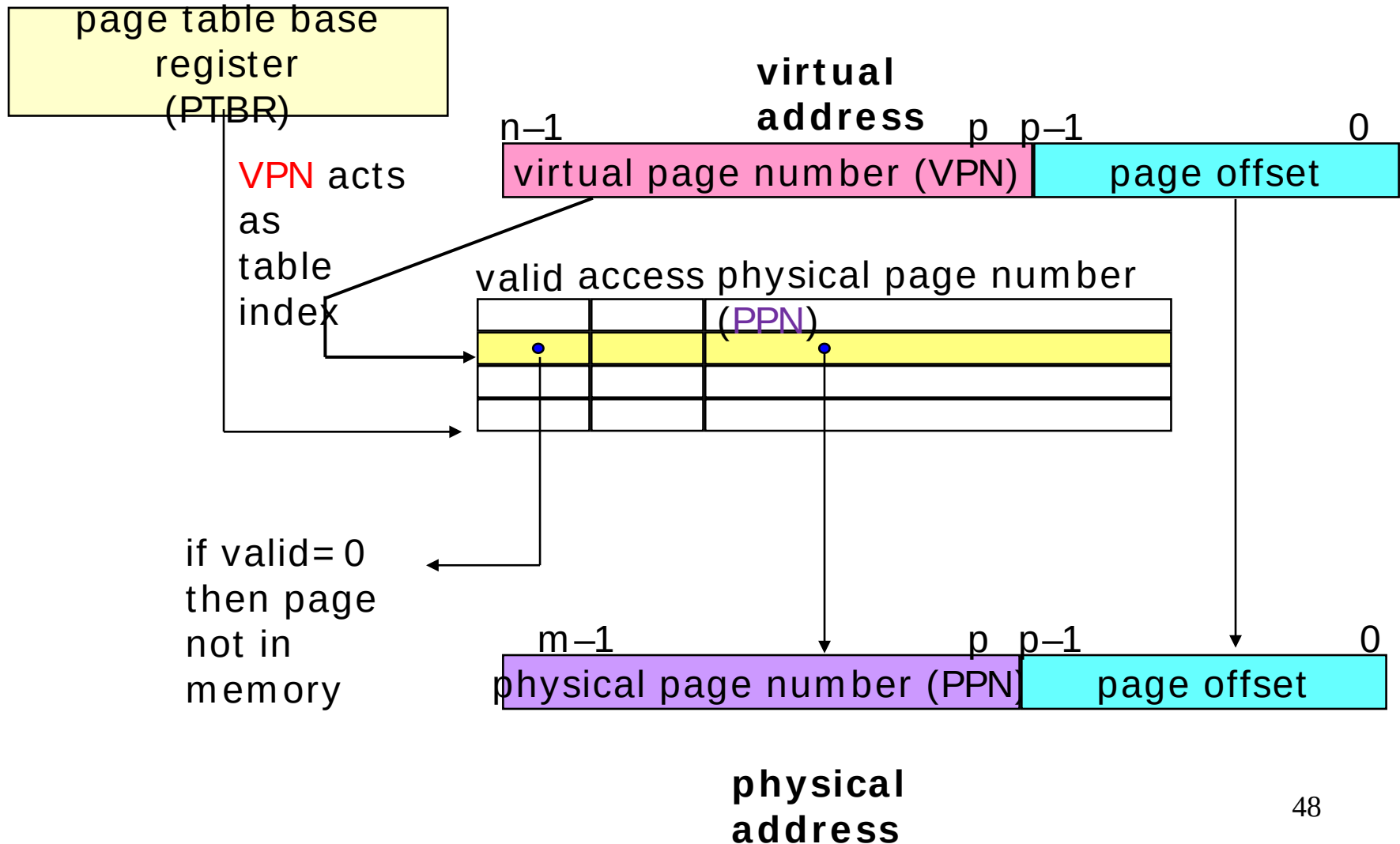


Notice that the page offset bits don't change as a result of translation

Address Translation

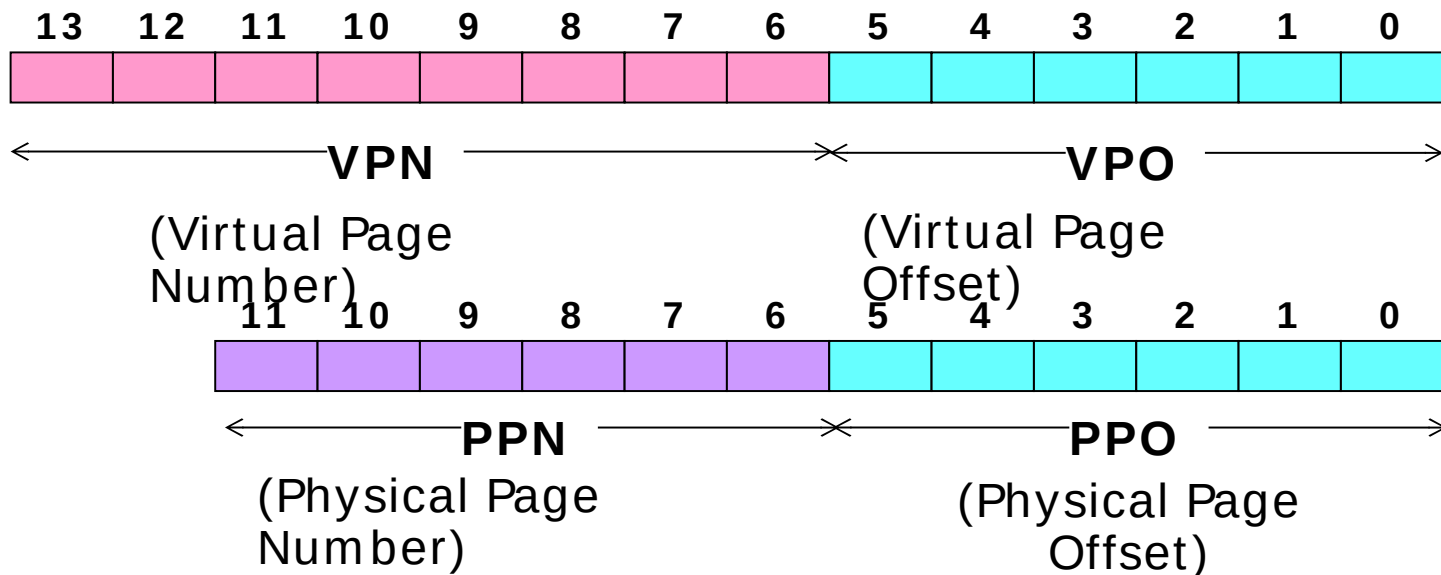
- Basic Parameters
 - **N** = 2^n = Virtual address limit
 - **M** = 2^m = Physical address limit
 - **P** = 2^p = page size (bytes).
- Components of the virtual address (VA)
 - **VPO**: Virtual page offset
 - **VPN**: Virtual page number
- Components of the physical address (PA)
 - **PPO**: Physical page offset (same as VPO)
 - **PPN**: Physical page number

Address Translation via Page Table



Simple Memory System Example

- Addressing
 - 14-bit virtual addresses
 - 12-bit physical address
 - Page size = 64 bits (6-bit)



Simple Memory System Page Table

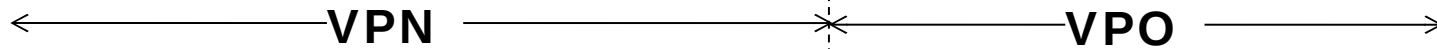
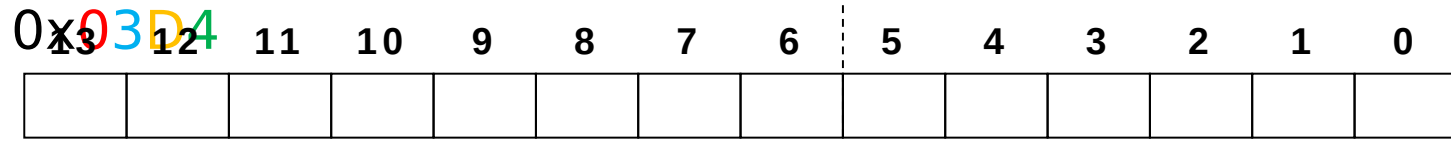
- Only show first 16 entries

	PPN	Valid
00	28	1
01	-	0
02	33	1
03	02	1
04	-	0
05	16	1
06	-	0
07	-	0

	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	-	0
0C	-	0
0D	2D	1
0E	11	1
0F	0D	1

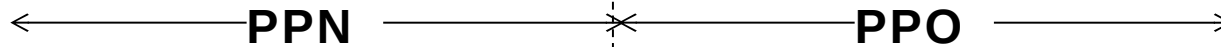
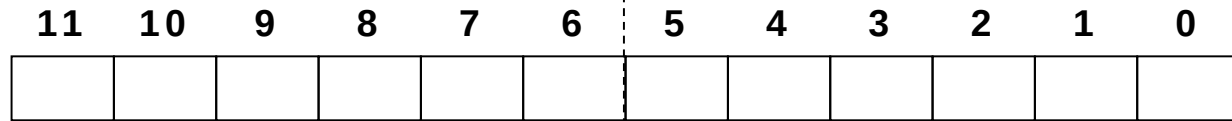
练习 4-2 : Address Translation

Virtual Address:



VPN: _____ VPO: _____

Page Fault? _____

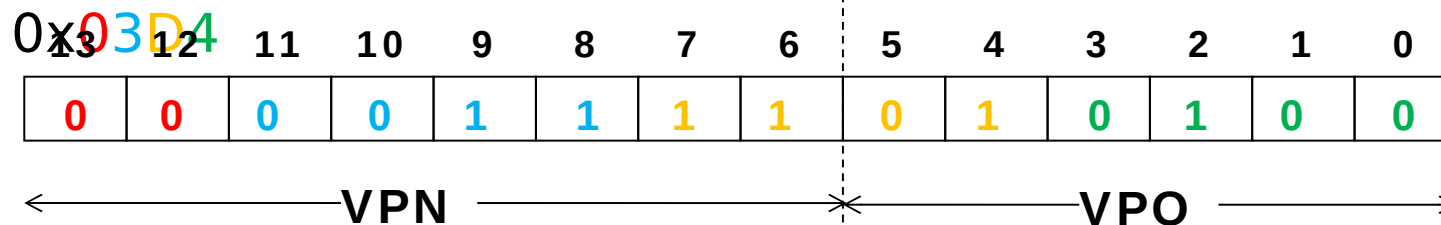


PPN: _____ PPO: _____

PA: _____

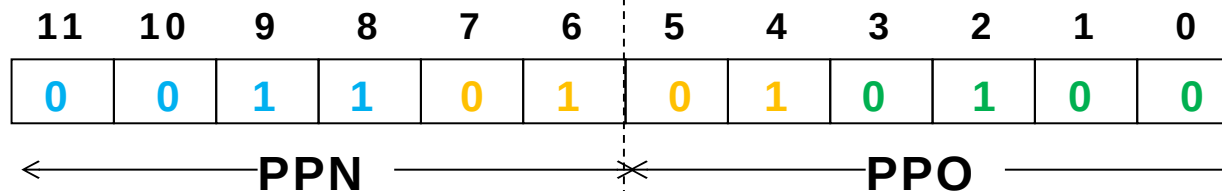
练习 4-2 答案

Virtual Address:



VPN: **0x0f** VPO: **0x14**

Page Fault? **No**



PPN: **0x0D** VPO: **0x14**

PA: **0x354**

Simple Memory System Page Table

- Only show first 16 entries

	PPN	Valid
00	28	1
01	-	0
02	33	1
03	02	1
04	-	0
05	16	1
06	-	0
07	-	0

	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	-	0
0C	-	0
0D	2D	1
0E	11	1
0F	0D	1

练习 4-3

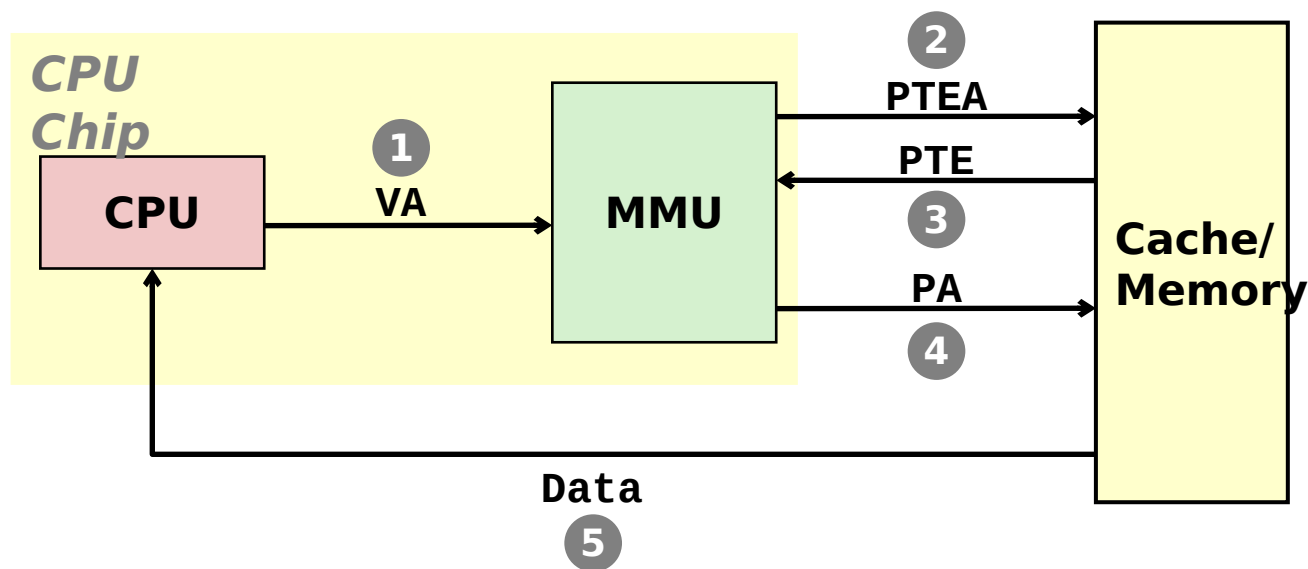
- 给定一个 32 位的虚拟地址空间和一个 24 位的物理地址，当页面大小 P 分别为 1KB，2KB，4KB，8KB 时，请确定 VPN、VPO、PPN、PPO 的位数。

练习 4-3 答案

- $P=1\text{KB}$
 - $VPN=22; VPO=10; PPN=14; PPO=10$
- $P=2\text{KB}$
 - $VPN=21; VPO=11; PPN=13; PPO=11$
- $P=4\text{KB}$
 - $VPN=20; VPO=12; PPN=12; PPO=12$
- $P=8\text{KB}$
 - $VPN=19; VPO=13; PPN=11; PPO=13$

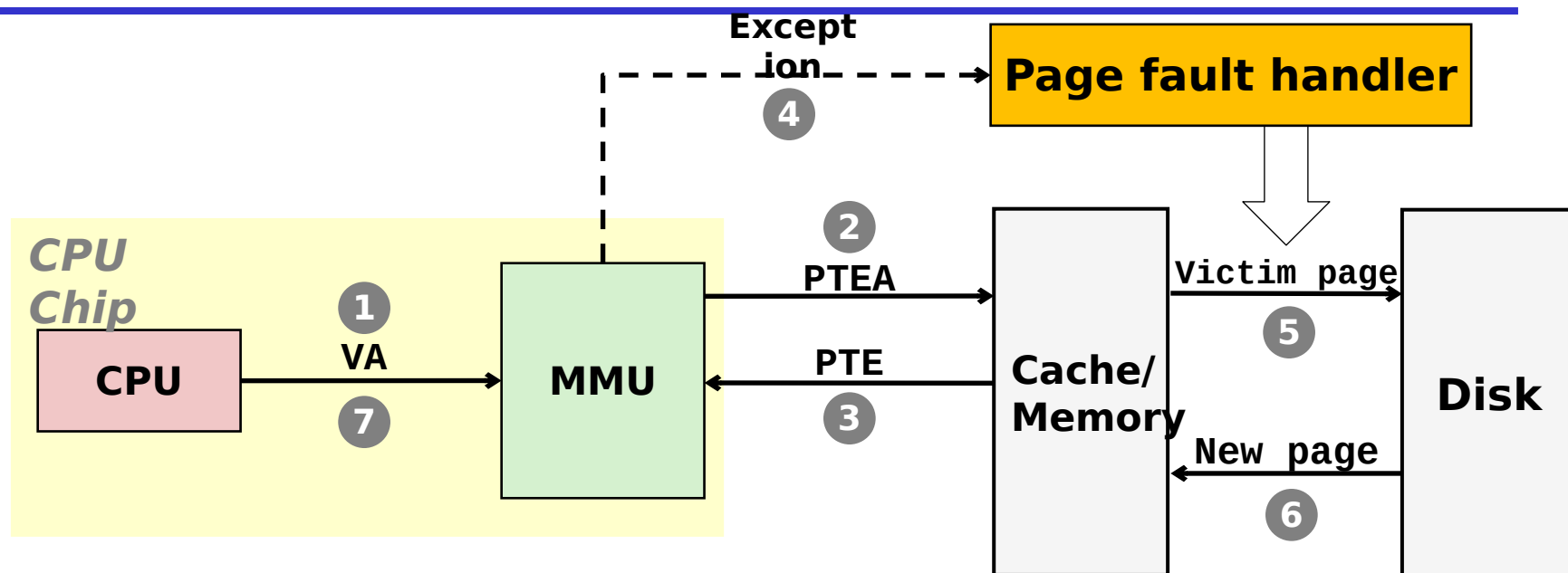
Page Hit

VA: virtual address
PTEA: page table entry address
PTE: page table entry
PA: physical address



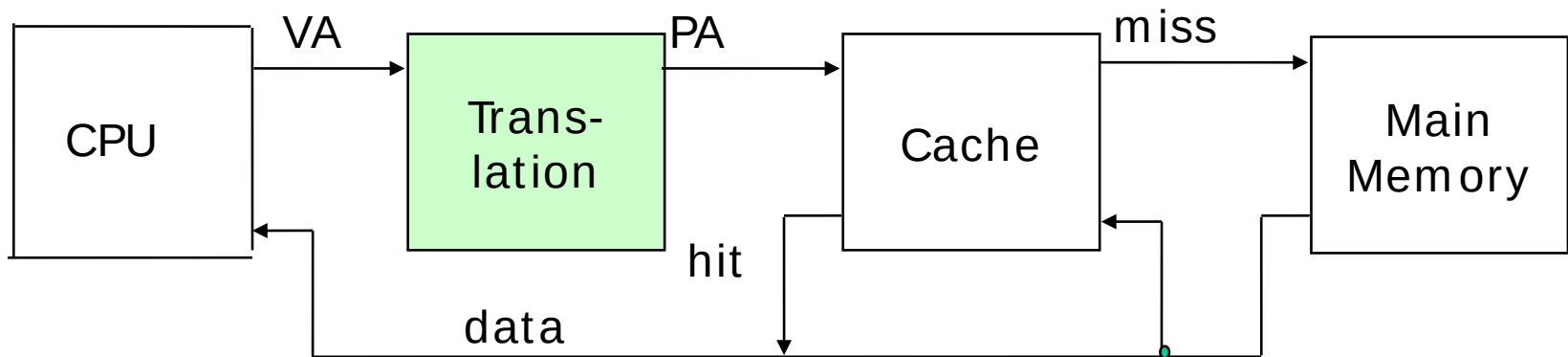
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

Page Faults



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

Integrating Caches and VM



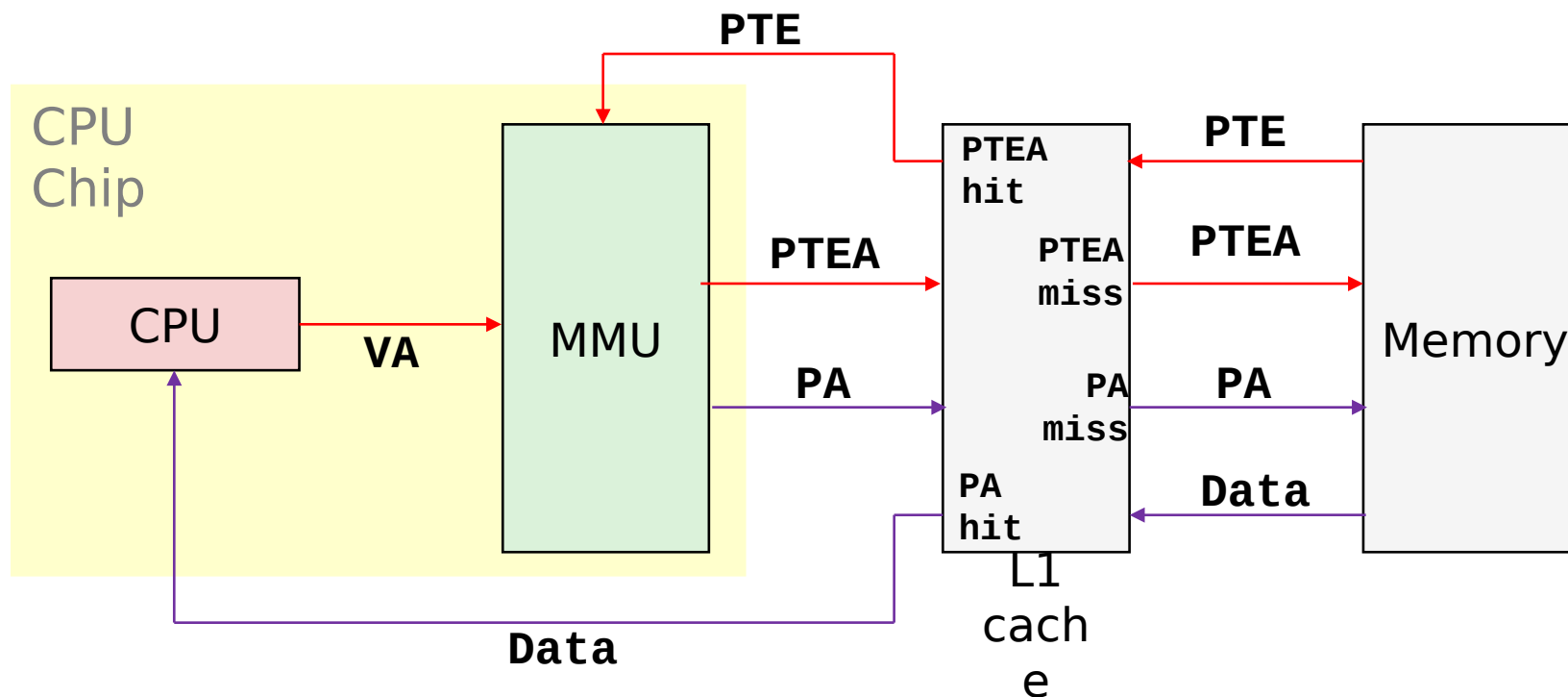
Integrating Caches and VM

- Most Caches “Physically Addressed”
 - Accessed by physical addresses
 - Allows multiple processes to have blocks in cache at same time (如果 cache 用 VA 呢?)
 - Allows **multiple** processes to **share** pages
 - Cache Coherence
 - Cache doesn’t need to be concerned with protection issues
 - **Access rights** checked as part of address translation

Integrating Caches and VM

- Perform Address Translation Before Cache Lookup
 - But this could involve a memory access itself (of the **PTE**)
 - Of course, page table entries can also become cached

Integrating Caches and VM



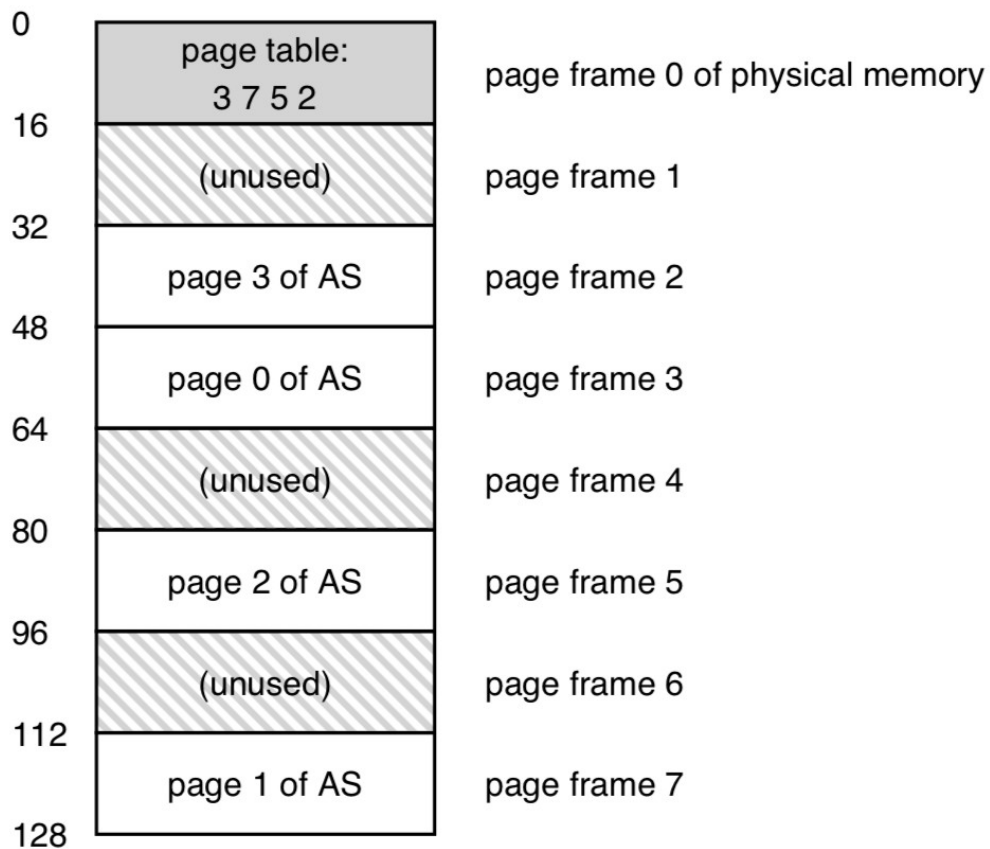
实际系统，有 TLB 专门作为 page table 的缓存（后续介绍）

Paging

- Page table 开销计算
 - 例如 32 位计算机，page 大小为 4KB
 - 所以 VPN 20 位，offset 12 位
 - 每个 page table entry (PTE) 是 4B
 - 假设有 100 个进程同时在运行
 - 则 page table 总大小为 $100 * 4B * 2^{20} = 400MB$
 - 非常庞大!

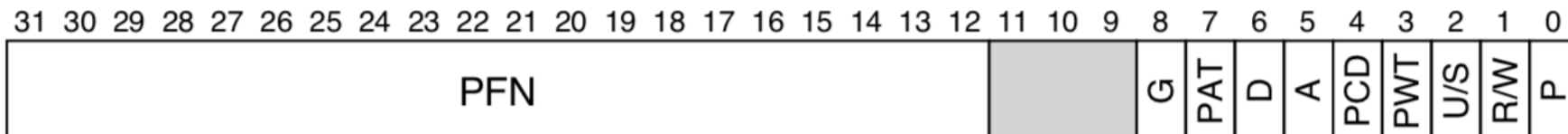
Paging

- Page Table
 - 没办法在 MMU 硬件上存储，只能放在 main memory 中



Paging

- 真实的 page table entry (PTE)
 - valid bit
 - protection bits (R/W) (r/w/e 权限)
 - present bit (P) (on disk/ in physical memory)
 - dirty bit (D) (与 disk 版本是否一致)
 - reference bit/accessed bit (A) (体现 page 的流行度, 与 replacement 有关)
 - U/S: user/supervisor, 用户态是否可以访问
 - PAT/PWT/PCD/G: 与硬件 cache 相关



Paging

- 地址翻译实例
 - `Movl 21, %eax` (直接寻址 21) 【立即数 \$21】
 - 假设有 page-table base register , 存储当前进程 page table 首地址
 - 获取 PTE 地址
 - `VPN_MASK = 0x30` (hex 30, or binary 110000)
 - `SHIFT = 4`
 - (虚存空间 64B , Page 大小 16B)

```
VPN      = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

Paging

- 地址翻译实例

```
1 // Extract the VPN from the virtual address
2 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4 // Form the address of the page-table entry (PTE)
5 PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7 // Fetch the PTE
8 PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Paging

- A Memory Trace

```
int array[1000];  
...  
for (i = 0; i < 1000; i++)  
    array[i] = 0;
```

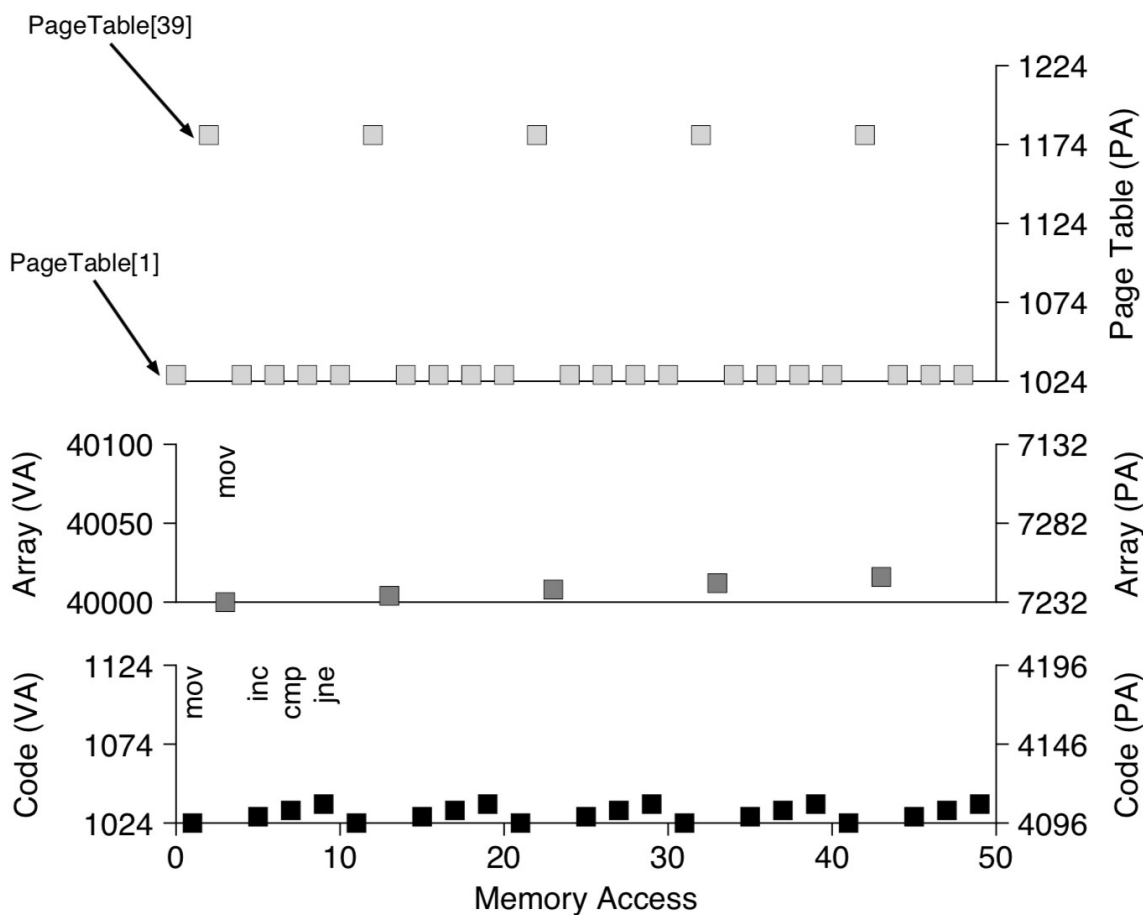
```
0x1024 movl $0x0, (%edi,%eax,4)  
0x1028 incl %eax  
0x102c cmpl $0x03e8,%eax  
0x1030 jne 0x1024
```

```
prompt> gcc -o array array.c -Wall -O  
prompt> ./array
```

Paging

- DRAM footprint

- 假设虚存空间 64KB
- Page 大小 1KB
- Page table 从物理地址 1KB 开始
- code 从 0x1024 开始, VPN=1
- array 从 40000 开始到 44000, VPN=39 ~ 42
 - $40000/1024=39.1$



Paging

- Pros:
 - 避免空间的 fragmentation（按 page 大小对齐；有细粒度的 page table 支持，VA 连续空间不需要 PA 连续）
 - Flexibility, VA 空间稀疏使用时效率高
- Cons:
 - Page table 空间开销大
 - 额外内存访问，导致性能变差

Paging

- Page table 的两个没有解决的问题：
 - 运行太慢（两次访问内存）
 - 占用内存空间太大

Paging: Faster Translations (TLBs)

(Translation Lookaside Buffer)

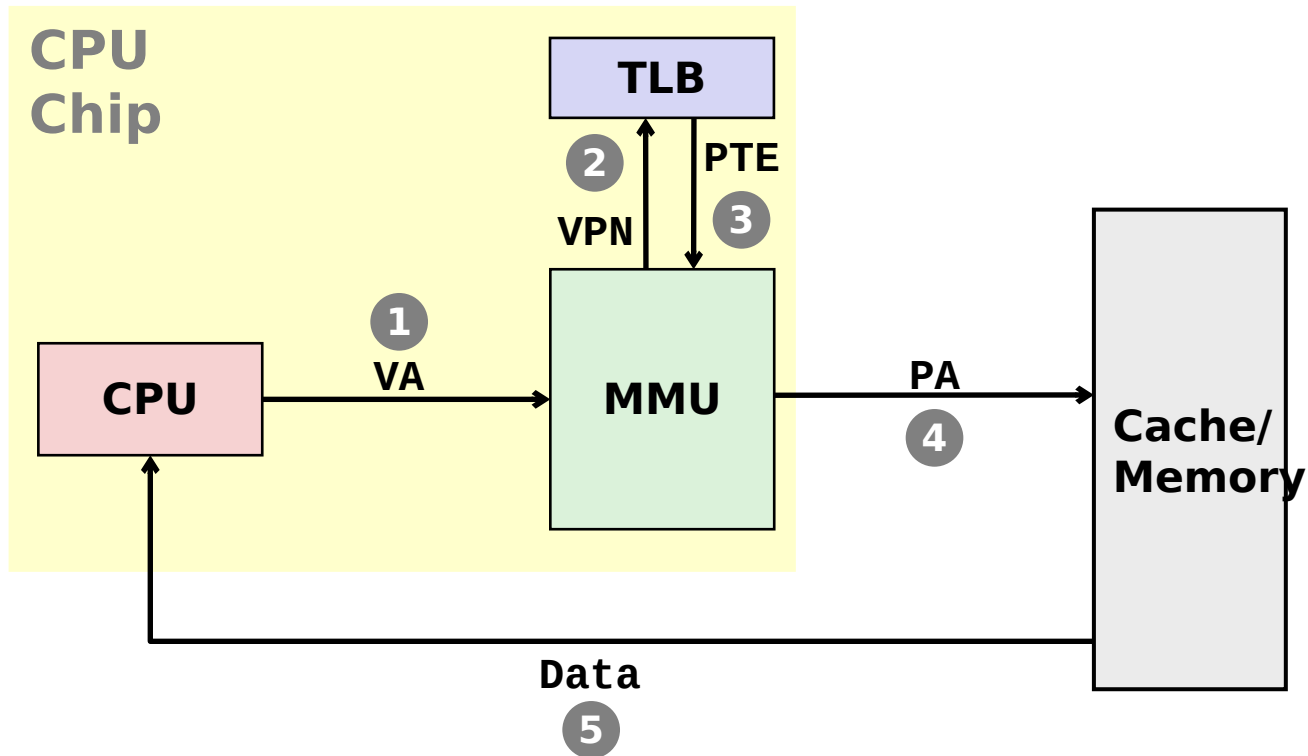
Paging: Faster Translations (TLBs)

- 目标：
 - 怎么加速 paging 方案的内存访问？
- 思考：
 - 用什么思路来加速？
 - 需要 hardware 提供什么支持？
 - 需要 OS 提供什么支持？

Paging: Faster Translations (TLBs)

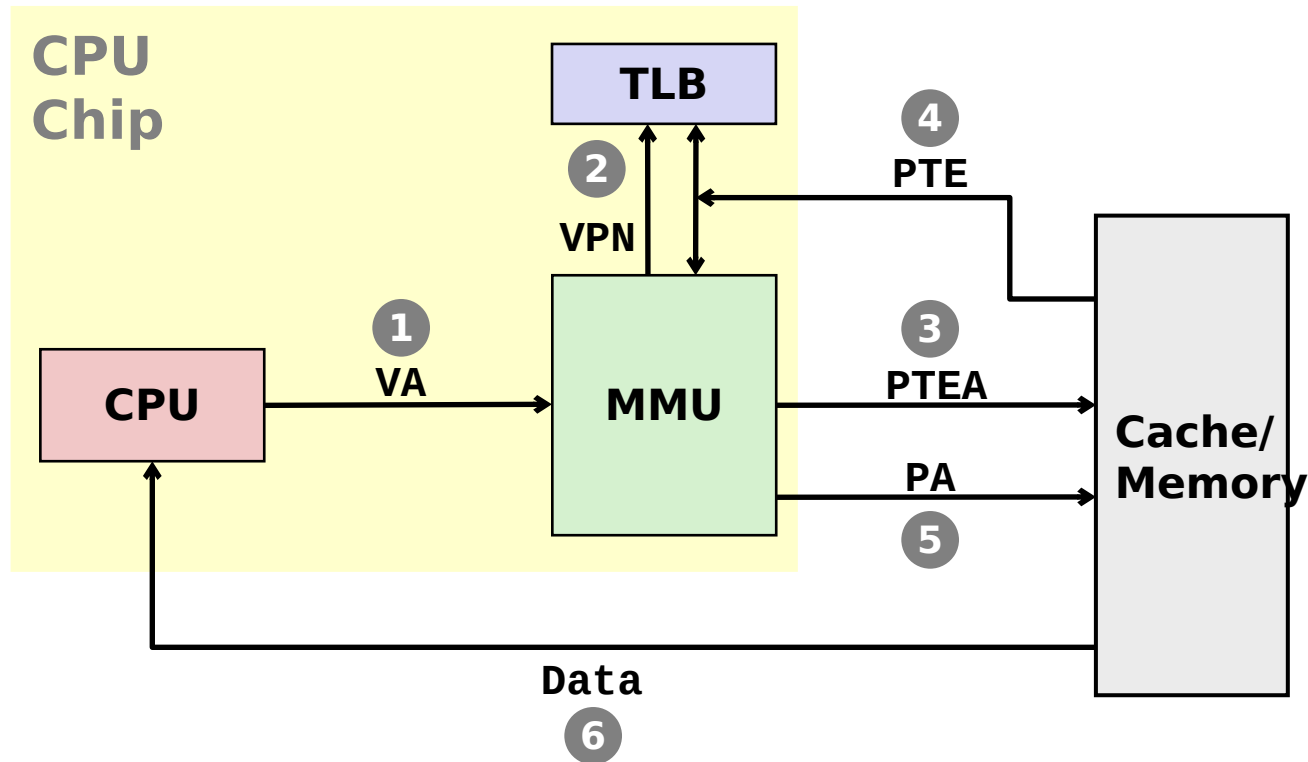
- **Translation-Lookaside Buffer (TLB)**
 - Hardware support (old friend of OS)
 - Part of **memory-management unit (MMU)**
 - hardware **cache** of popular virtual-to-physical address translations (VPN->PPN) (**address-translation cache**)
 - 没有 TLB ， paging 的 virtual address 方案就是不实用的

TLB Hit



A TLB hit eliminates a memory access

TLB Miss



A TLB miss incurs an additional memory access (PTE)

Fortunately, TLB misses are rare. Why?

Paging: Faster Translations (TLBs)

- Tiny Address Space Example

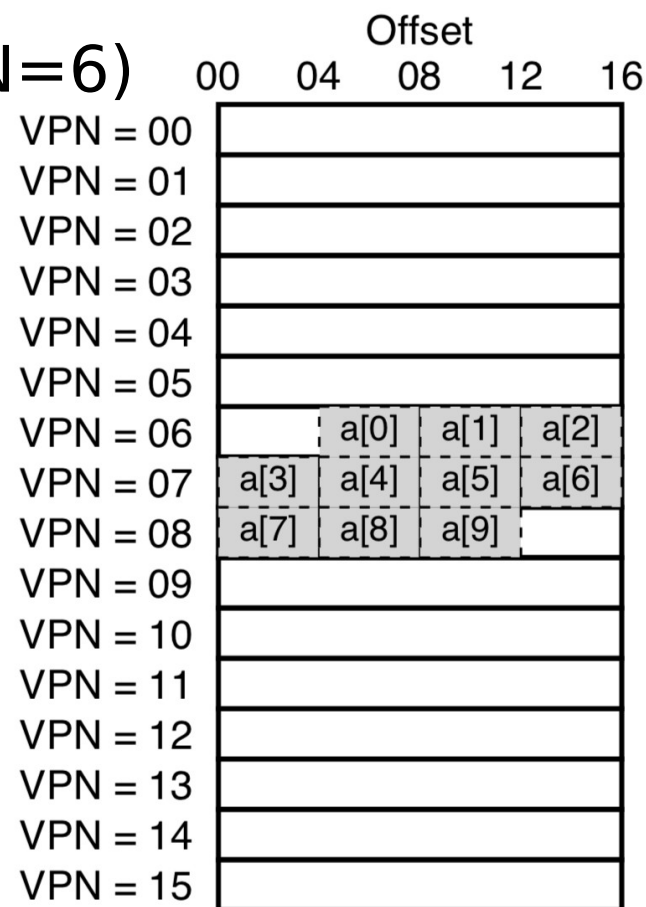
- VA space: 8-bit, one page: 16B

- Array: starting at VA=100 (VPN=6)

- VPN: 4-bit, VPO: 4-bit

- 计算下列代码的 TLB 命中率:

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```



Paging: Faster Translations (TLBs)

- Tiny Address Space Example

- VA space: 8-bit, one page: 16B

- Array: starting at VA=100 (VPN=6)

- VPN: 4-bit, VPO: 4-bit

- 计算下列代码的 TLB 命中率:

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

a[0], a[3], a[7]

miss;

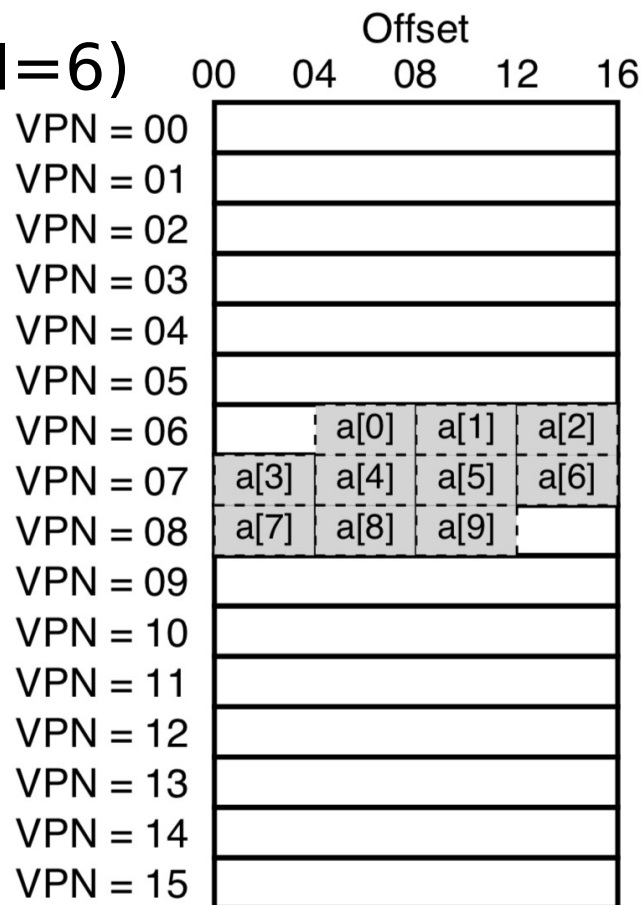
其余 hit。命中率

=70%

**(Spatial
locality)**

Page 增大，命中

率进一步提升



Paging: Faster Translations (TLBs)

- Who Handles The TLB Miss?
 - Hardware, or OS?
 - 早期，都是 hardware 处理（不信任 OS），TLB 需要知道 page table 的物理地址
 - E.g., intel x86 architecture
 - 现代，software-managed TLB
 - 当 TLB miss 时，抛出一个 exception
 - 陷入 kernel mode，在 handler 处理 TLB miss
 - 之后回到原指令流，重试指令，TLB hit
 - E.g., MIPS R10k, Sun SPARK v9 (RISC)

Paging: Faster Translations (TLBs)

- TLB Control Flow Algorithm (OS Handled)

```
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
11     // TLB Miss
12     RaiseException(TLB_MISS)
```

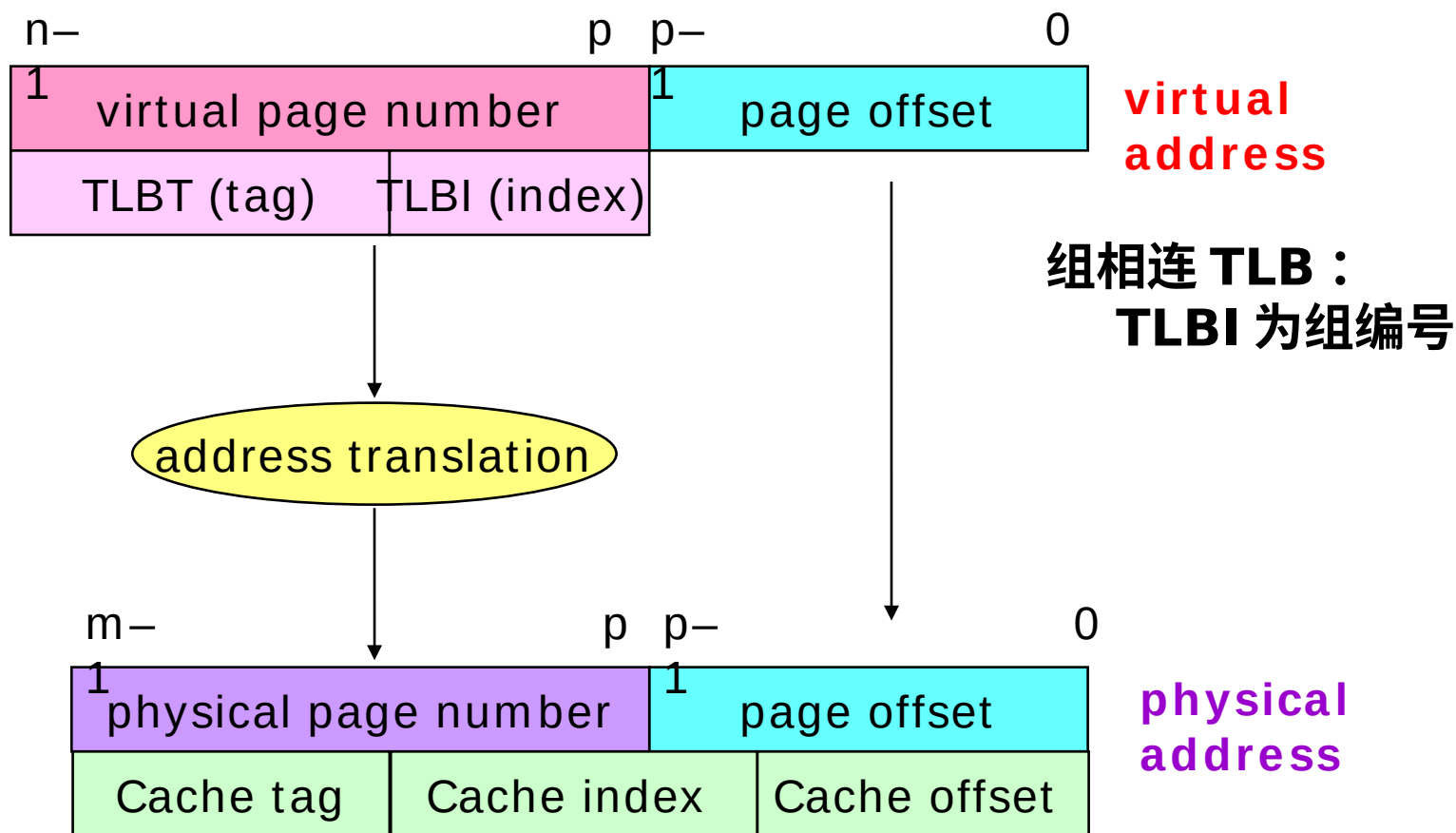
Paging: Faster Translations (TLBs)

- 软件处理 TLB miss 方式的优点：
 - **Flexibility**: OS can use any data structure it wants to implement the page table, without necessitating hardware change.
 - **Simplicity**: hardware doesn't have to do much on a miss

Paging: Faster Translations (TLBs)

- TLB Contents: What's In There?
 - 典型 TLB : 32, 64 或 128 entries , fully associative (全相连)
 - 格式:
 - VPN | PFN | other bits
 - Other bits:
 - A Valid bit: 该 entry 是否为有效的地址翻译
 - 暂未使用的 TLB entries
 - 例如进程切换时, 上一个进程的 TLB entries 被 invalidate
 - » 或者 TLB entries 中含有进程标记 (ASID)
 - Protection bits
 - A Dirty bit
 - page fault-> 新的映射 -> 应修改 page table-> 先写入

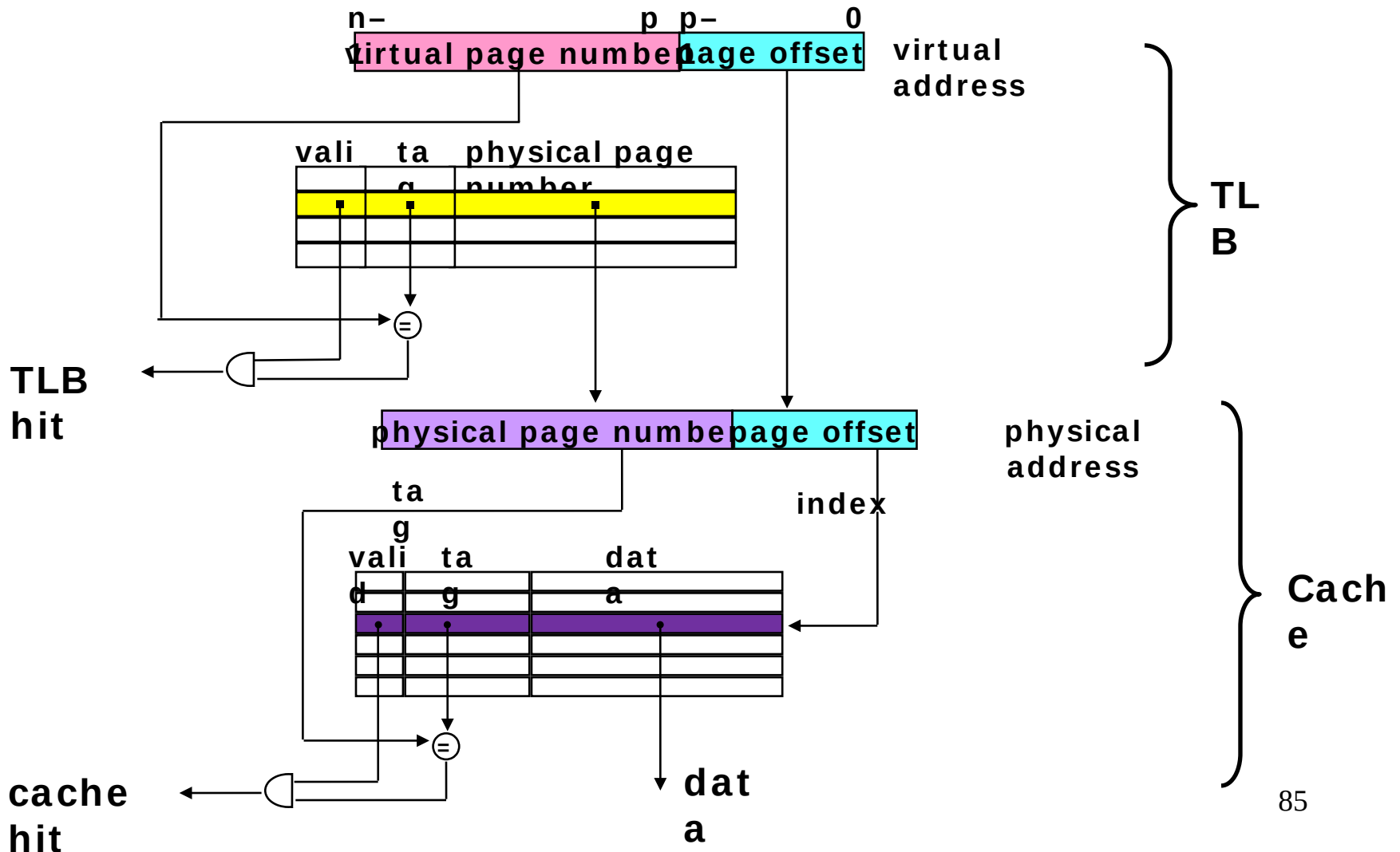
Paging: Faster Translations (TLBs)



Address Translation

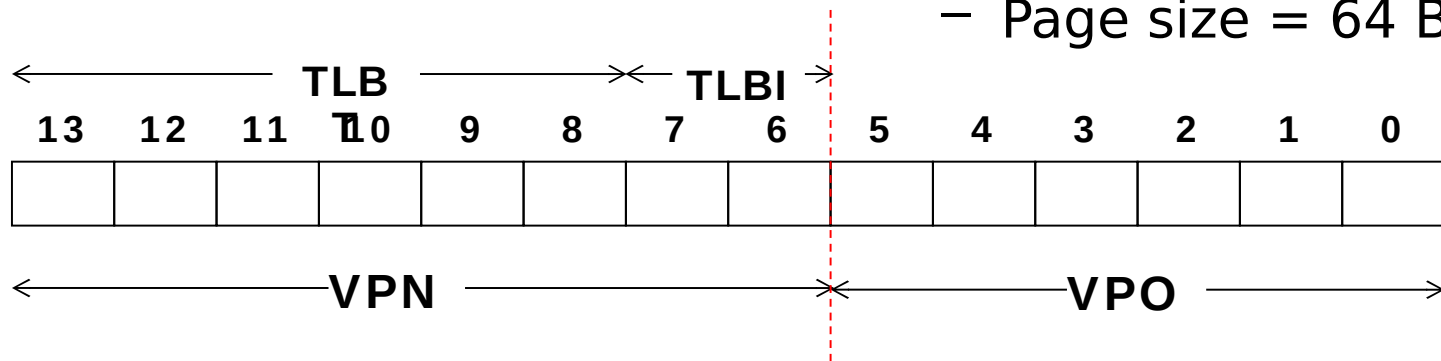
- Components of the virtual address (VA)
 - **VPO**: Virtual page offset
 - **VPN**: Virtual page number
 - **TLBI**: TLB index
 - **TLBT**: TLB tag
- Components of the physical address (PA)
 - **PPO**: Physical page offset (same as VPO)
 - **PPN**: Physical page number
 - **CO**: Byte offset within cache line
 - **CI**: Cache index
 - **CT**: Cache tag

Paging: Faster Translations (TLBs)



Simple Memory System TLB

- TLB
 - 16 entries
 - 4-way associative (2-bit)
- Addressing
 - 14-bit virtual addresses
 - 12-bit physical address
 - Page size = 64 B (6-bit)

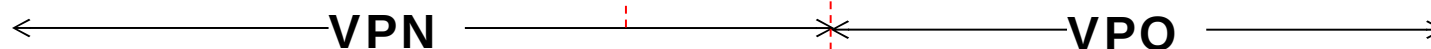
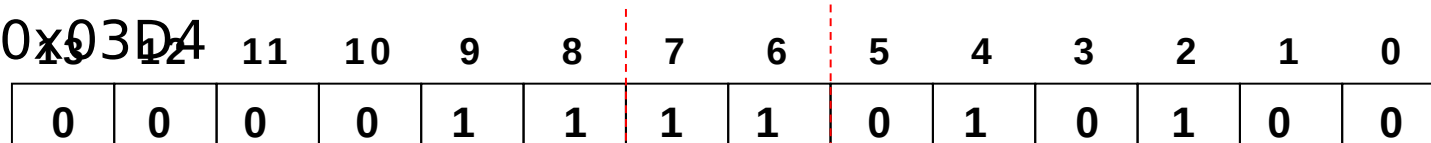


	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

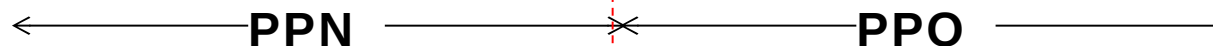
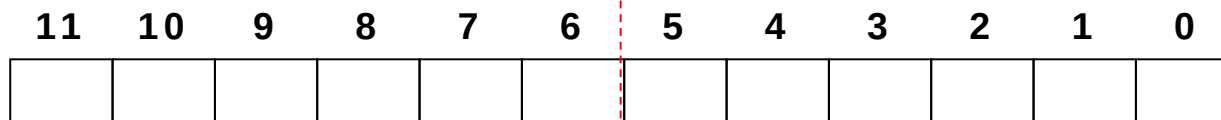
练习 2-4

Virtual Address

0x03D4



VPN: ___ TLBI: ___ TLBT: ___ TLB Hit? ___ Page Fault? ___

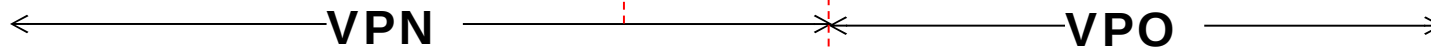
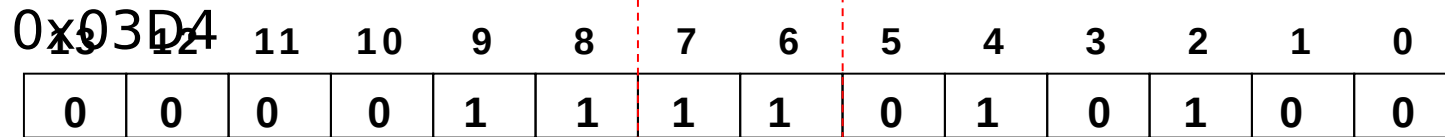


PPN: _____ PPO: _____

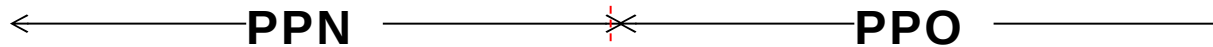
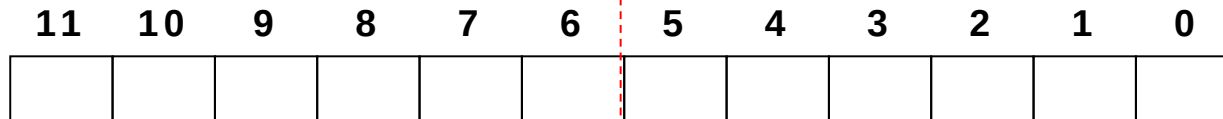
PA: _____

练习 2-4 答案

Virtual Address



VPN: **0x0f** TLBI: **0x03** TLBT: **0x03** TLB Hit? ___ Page Fault? ___

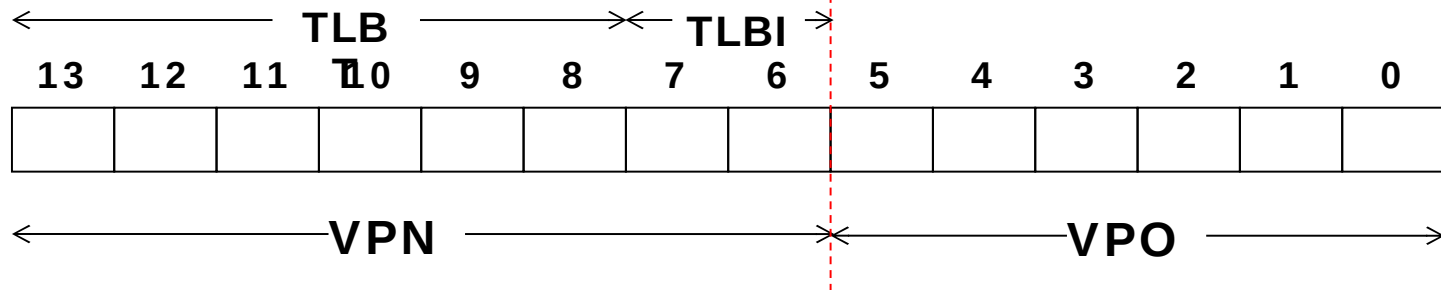


PPN: _____ PPO: _____ PA: _____

练习 2-4 答案

- TLB
 - 16 entries
 - 4-way associative (2-bit)
 - 四路指每组 (set) 四项，并发查找
 - $16/4 = 4$ (一共四组，所以 TLBI 2-bit)

- Addressing
 - 14-bit virtual addresses
 - 12-bit physical address
 - Page size = 64 B (6-bit)

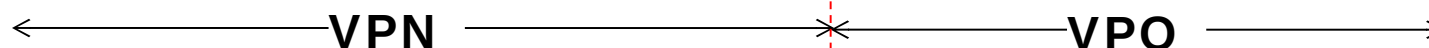
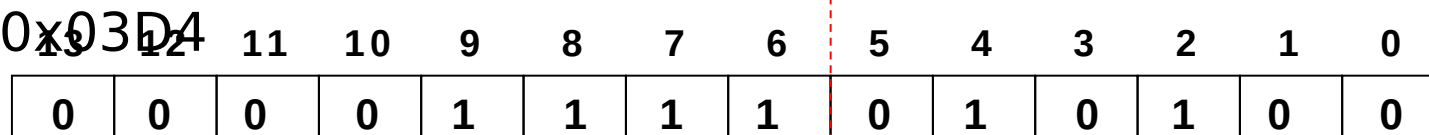


	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

练习 2-4 答案

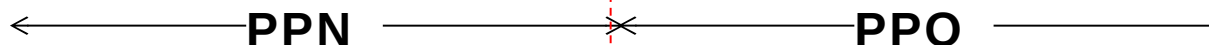
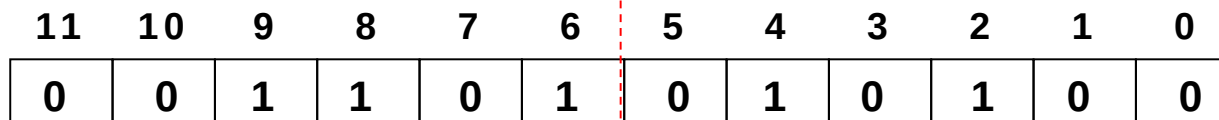
Virtual Address

0x03D4



VPN: 0x0f TLBI: 0x03 TLBT: 0x03 TLB Hit? Yes Page Fault?

No



PPN: 0x0D PPO: 0x14

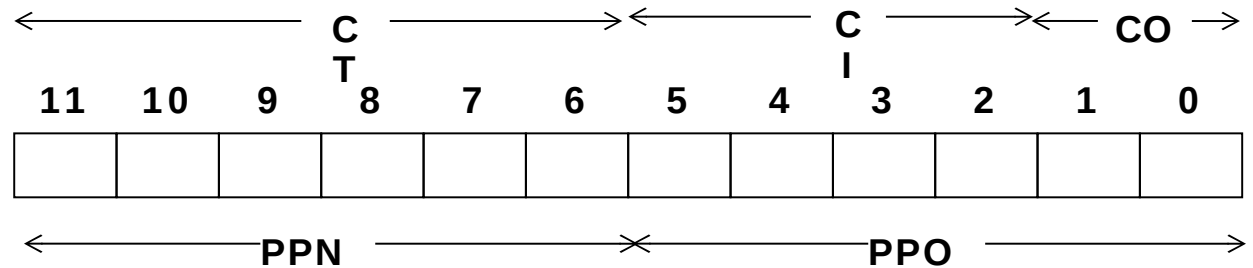
PA:

0x354

Simple Memory System Cache

- Cache

- 16 lines
- 4-byte line size
- Direct mapped

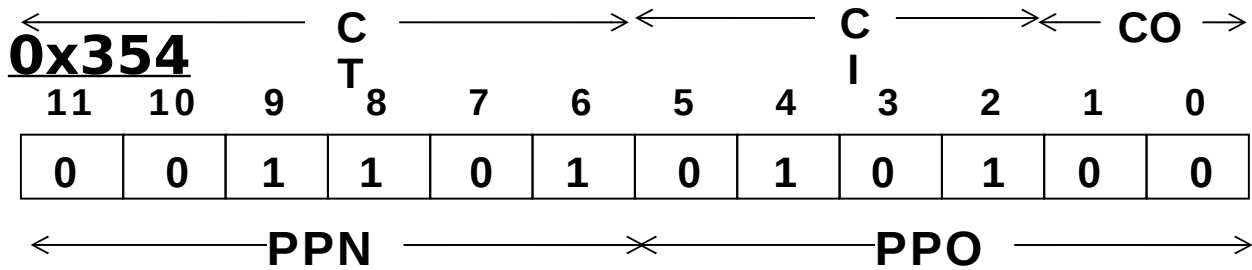


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	-	-	-	-
A	2D	1	93	15	DA	3B
B	0B	0	-	-	-	-
C	12	0	-	-	-	-
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	-	-	-	-

练习 2-5

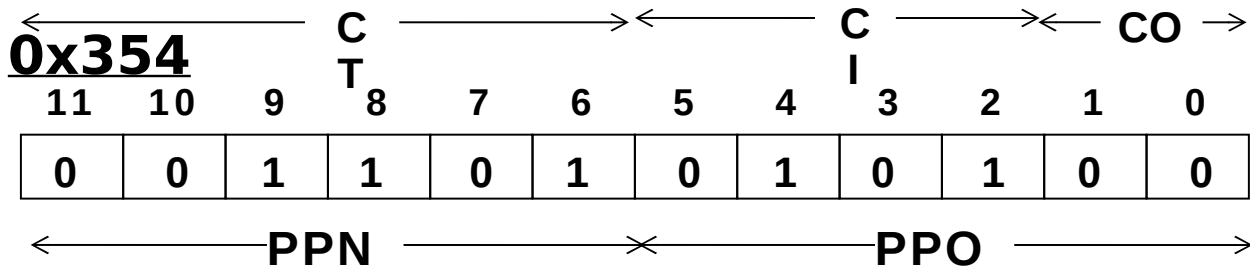
PA:



Offset: ____ CI: ____ CT: ____ Hit? ____ Byte: ____

练习 2-5 答案

PA:

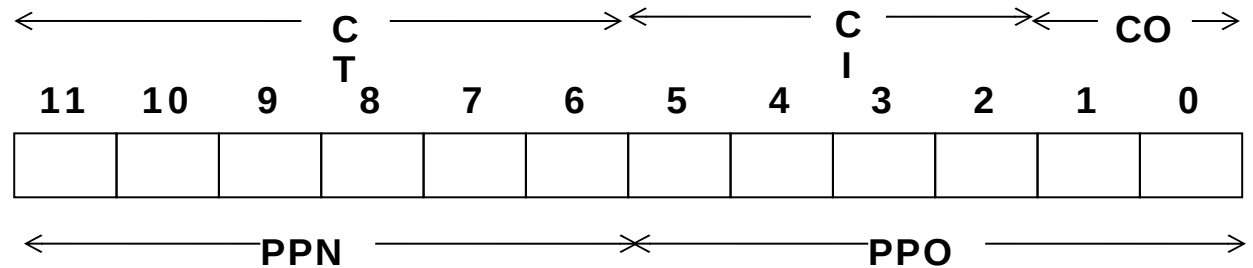


Offset: **0x0** CI: **0x05** CT: **0x0D** Hit? ___ Byte: **0x**___

Simple Memory System Cache

- Cache

- 16 lines
- 4-byte line size
- Direct mapped

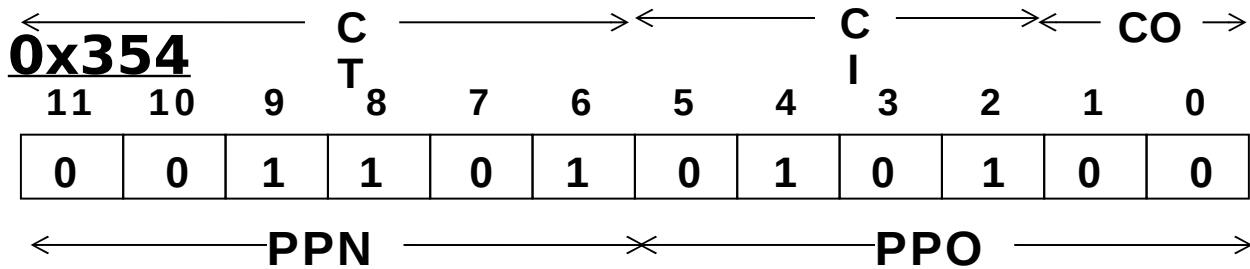


	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	-	-	-	-
A	2D	1	93	15	DA	3B
B	0B	0	-	-	-	-
C	12	0	-	-	-	-
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	-	-	-	-

Address Translation Example

PA:



Offset: **0x0** CI: **0x05** CT: **0x0D** Hit? **Yes** Byte: **0x36**

Paging: Faster Translations (TLBs)

- TLB Issue: Context Switches
 - TLB 中存储的是当前 process 的 page table
 - 当发生 context switch 的时候，旧的缓存项可能干扰
 - E.g., 进程从 P1 切换到 P2 ， VPN=10 可能有两个 entry
 - 具体应该如何在 context switch 时处理 TLB ？

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

Paging: Faster Translations (TLBs)

- TLB Issue: Context Switches
 - #1. 简单地清空 TLB ，重新开始
 - OS：利用特权指令来控制硬件
 - 硬件：直接将 TLB 所有位清零
 - 问题：TLB 命中率回到 0 ，性能较差
 - #2. **address space identifier (ASID)**
 - 可以按照 PID 来理解
 - 可以区分不同 VA 空间，不必清空 TLB ，也不会发生错误

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

Paging: Faster Translations (TLBs)

- TLB Issue: Context Switches
 - 两个 VPN 指向同一个 PFN (PPN)
 - 合理，因为两个进程进行了内存共享
 - Fork
 - 进程间通信 -> 共享内存

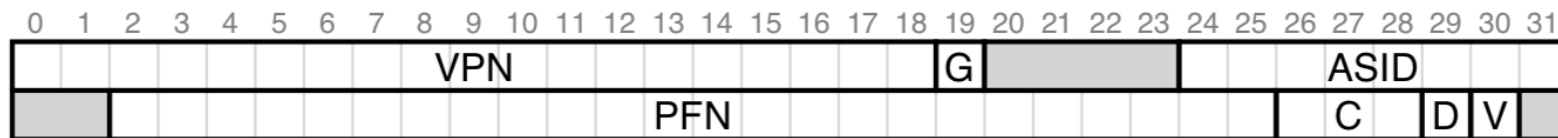
VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

Paging: Faster Translations (TLBs)

- Issue: Replacement Policy
 - **least-recently-used (LRU)**
 - **random**
 - LRU 未必命中率比 random 高
 - 例如进程间规律 switch ， LRU 导致每个进程初始命中率很低

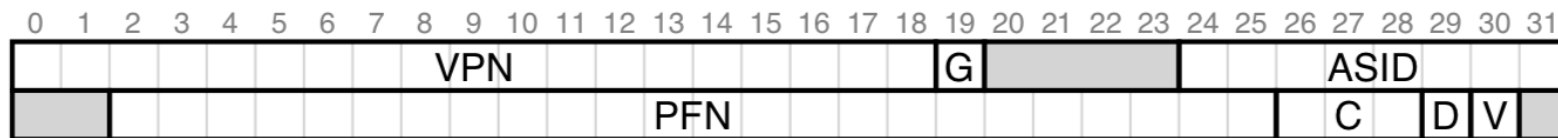
Paging: Faster Translations (TLBs)

- A Real TLB Entry
 - MIPS R4000: 32-bit address space, 4KB pages
 - Software-managed TLB
 - A TLB entry: 64 bits
 - VPN: 20-bit, offset: 12-bit
 - 实际 VPN 19 位，因为用户只能用一半，另一半留给 kernel
 - PFN: 24-bit ，物理内存支持到 64GB (2^{24} B)



Paging: Faster Translations (TLBs)

- A Real TLB Entry
 - ASID: 8-bit , 最多 256 个进程
 - A global bit (G): 全局共享给所有进程, 如果设置, ASID 被忽略
 - Coherence bit (C): 硬件 cache 相关
 - Dirty bit (D)
 - Valid bit (V)
 - 一般有 32 或 64 个 entry , 少量预留给 OS



Paging: Faster Translations (TLBs)

- A Real TLB Entry

- 特权指令，操作 TLB

- TLBP: 看 TLB 中是否存在一个特定的 translation
 - TLBR: 将一个 TLB entry 读到寄存器
 - TLBWI: 写入一个 TLB entry
 - TLBWR: 随机替换一个 TLB entry



Paging: Faster Translations (TLBs)

- RAM ISN'T ALWAYS RAM (CULLER'S LAW)
 - Random-access memory (RAM) ，理论上访问各地址的速度一样
 - 但实际上因为有 cache 、 TLB 的存在，访问不同地址的速度差距可能很大
 - 所以， **RAM isn't always RAM**

进程与地址翻译

- (Linux) 每个进程有自己的虚拟地址空间
 - 两个进程可能同一个虚拟地址
- 页表
 - 每个进程自己独立的页表（不同的内存区域）
 - MMU 要知道进程号 (PID)，才能找到对应的页表
 - 所以 MMU 是硬件加速页表的访问，但页表内存需要 OS 去管理，因为进程是 OS 下才有的概念（对于 MMU 只是 ID-> 页表这样多个页表的行为模式，不需要知道进程的概念）

进程与地址翻译

- TLB

- 两个进程可能有冲突： $Va \rightarrow Pa$, $Va \rightarrow Pb$ （不同进程的虚拟地址都是 Va ，但对应的物理地址不同）
- 没问题，但性能不佳的方案
 - 进程 B 切到进程 A ， invalidate TLB 中所有条目，从头开始
 - 再换回 B ， 没有 cache ， 实际映射关系不变
- ASID （ Address Space ID ）
 - 每个进程一个 ASID ， 真的索引信息是 ASID+VPN

Paging: Smaller Tables

Paging: Smaller Tables

- 回顾 Paging 方案的两个问题：
 - 速度慢（TLB 加速）
 - 占用内存空间大（? ）
 - 32-bit address space, 4KB pages, 4-byte page-table entry
 - 一个进程： $1\text{M} * 4\text{B} = 4\text{MB}$
 - 100 个并发进程： 400MB

Paging: Smaller Tables

- Given:
 - X86: 32-bit address space
4KB (2^{12}) page size, 4-byte PTE
 - X86-64: 48-bit address space
4KB (2^{12}) page size, 8-byte PTE
- Problem:
 - X86: Would need a **4 MB** page table!
 - $2^{20} * 4$ bytes (20bit = 32bit - 12bit)
 - X86-64: Would need a **512 GB** page table!
 - $2^{36} * 8$ bytes (36bit = 48bit - 12bit)

Paging: Smaller Tables

- Simple Solution : Bigger Pages
 - Page size: 4KB -> 16KB
 - 一个进程: $2^{32}/2^{14} = 2^{18} = 256K$ 个 entry , 1MB 空间
 - 100 个进程: 100MB
 - 问题: **internal fragmentation**
 - 用户至少分配一个 page , 但可能只用了其中的一小部分, 浪费空间
 - Page 变大, 削弱 paging 方案的 flexibility 优势

Paging: Smaller Tables

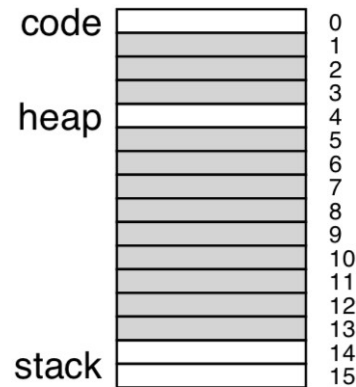
- Hybrid Approach: Paging and Segments
 - Paging: 细粒度，空间开销大，适合稠密地址空间
 - Segment：粗粒度，空间开销小，适合稀疏地址空间
 - Hybrid: 利用二者优势
 - 最早是 multics 作者之一 Jack Dennis 提出

Paging: Smaller Tables

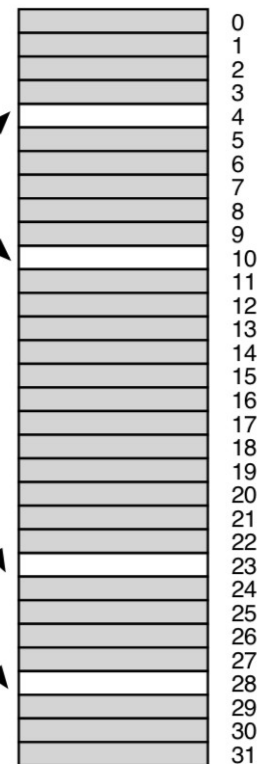
- Hybrid Approach: Paging and Segments
 - E.g., 16KB address space, 1KB pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Virtual Address Space



Physical Memory



Paging: Smaller Tables

- Hybrid Approach: Paging and Segments
 - Segment: $\text{base} + \text{bound}$ ， 放在专门的寄存器中
 - Code, heap, stack 三个 segment ， 因此有三对 $\text{base} + \text{bound}$ 寄存器
 - 存放当前活跃进程的三个 segment 信息
 - 每个 segment 建立一张页表， bound 为段内已使用的最大内存
 - 后面没有分配的空间不计入 page table ， 可以节省空间
 - 假设 code segment 只使用了前三个 page (0, 1, 2) ， 那么 code segment page table 只记录三个 entry ， $\text{bound} = 3$

Paging: Smaller Tables

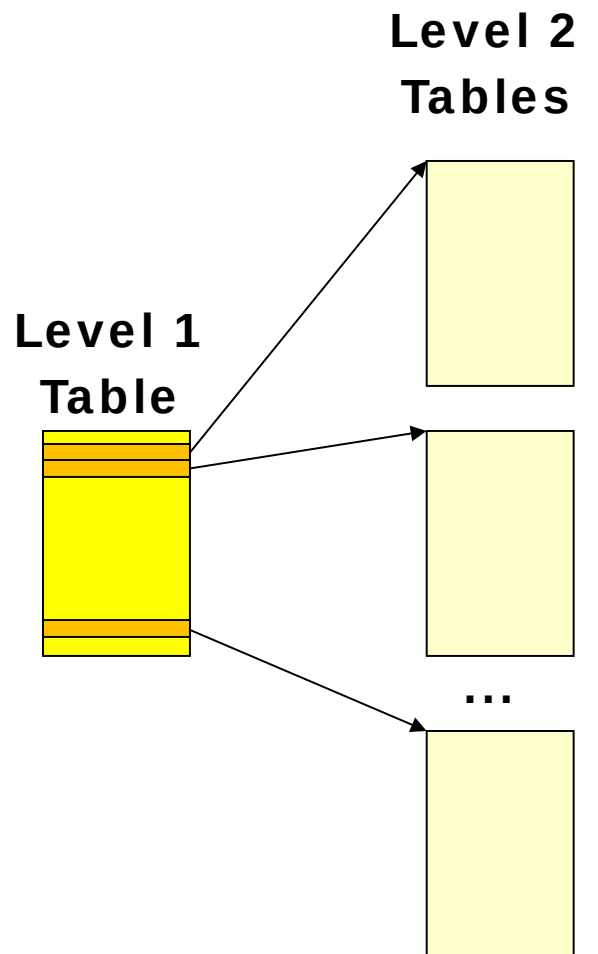
- Hybrid Approach: Paging and Segments
 - 段页式内存管理
 - 也存在问题：
 - 如果空间稀疏（内部空，而不是后面空），效率也不高
 - 两种方式，导致额外的 fragmentation

Paging: Smaller Tables

- 为什么 Page Table 占用空间会大？
 - 每个 VPN 都需要一个 entry
 - 但实际上多数进程并没有真正利用多少 VPN，占用多少 PPN
 - i.e., Page Table 往往是稀疏的，有很多 entry 是空的，却占用了大量空间
 - Solution: **多级页表**，压缩空白 entry 所占的空间

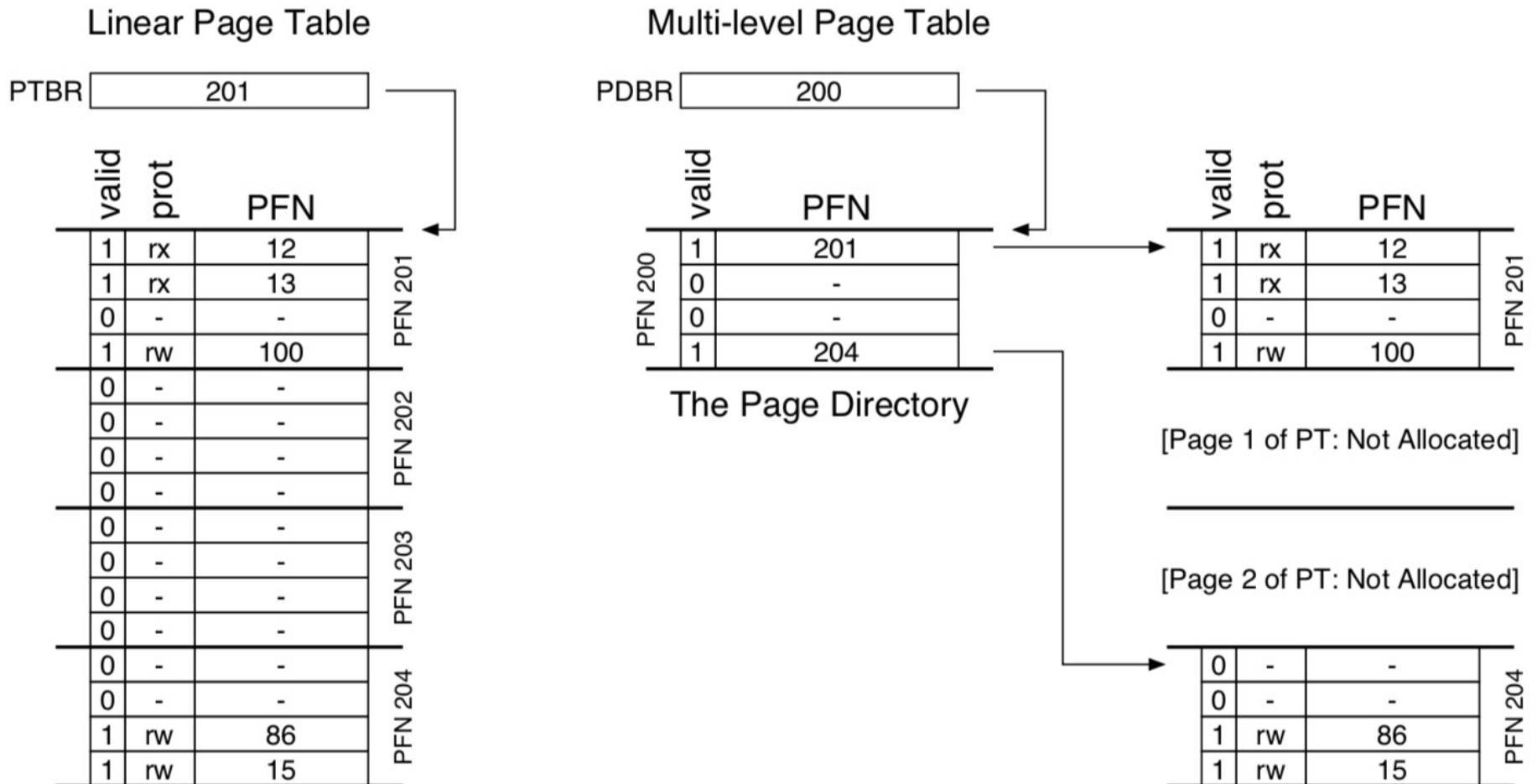
Paging: Smaller Tables

- Common solution
 - multi-level page tables
 - e.g., 2-level page table
 - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
 - Level 2 table: 1024 entries, each of which points to a page

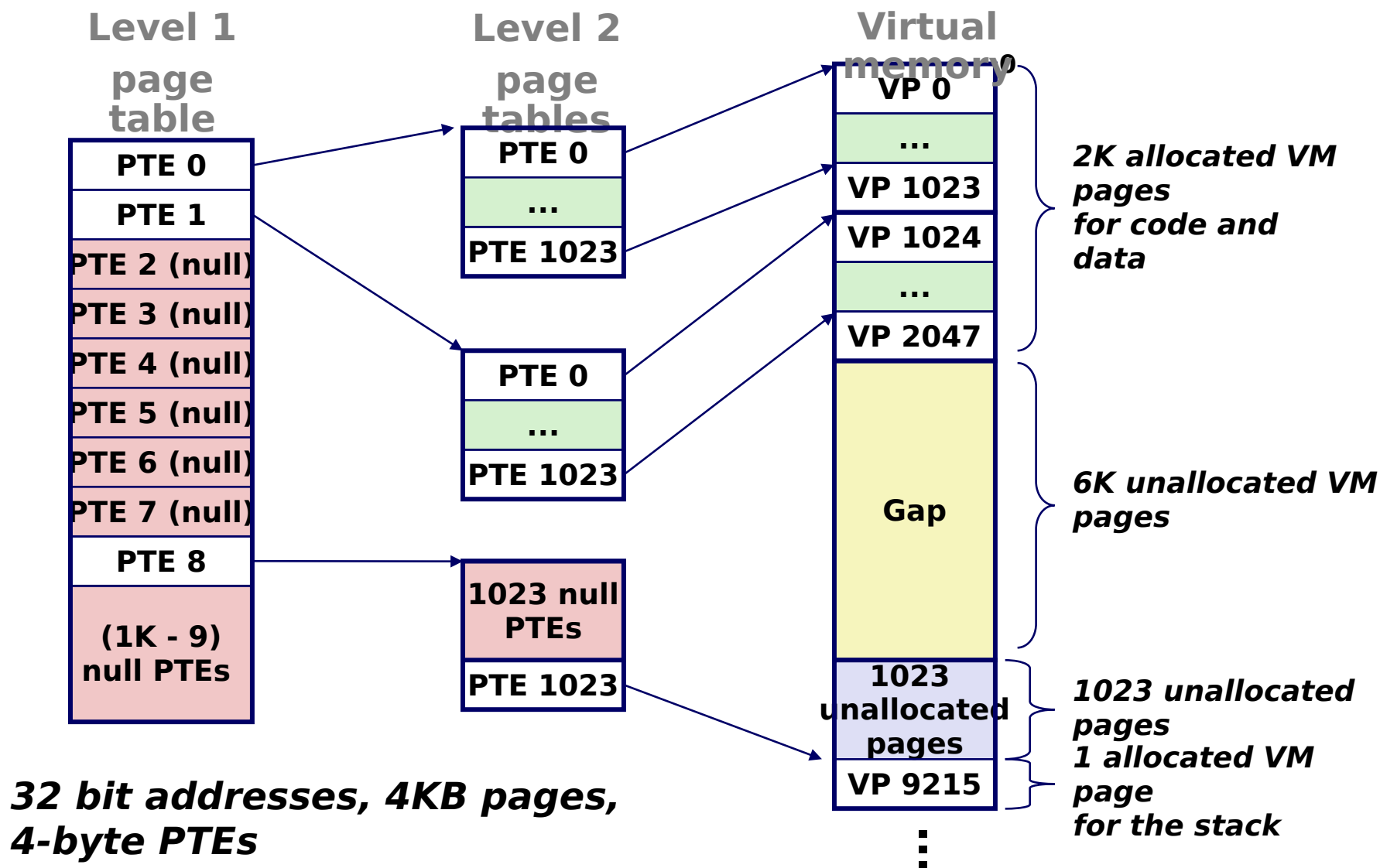


Paging: Smaller Tables

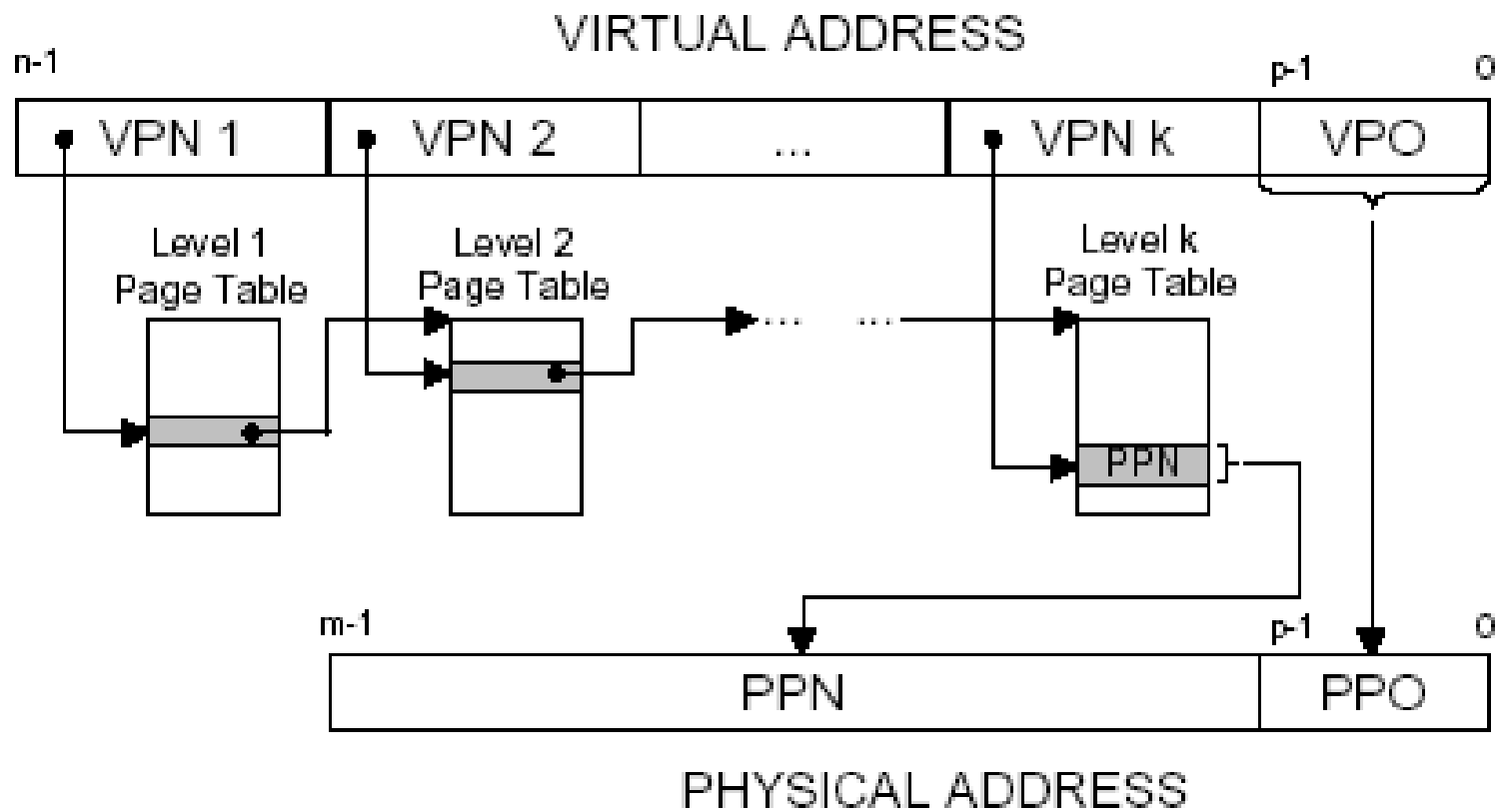
- Multi-level Page Tables



Multi-Level Page Tables: 中间可以有空洞



Paging: Smaller Tables



Paging: Smaller Tables

- **A Detailed Multi-Level Example**

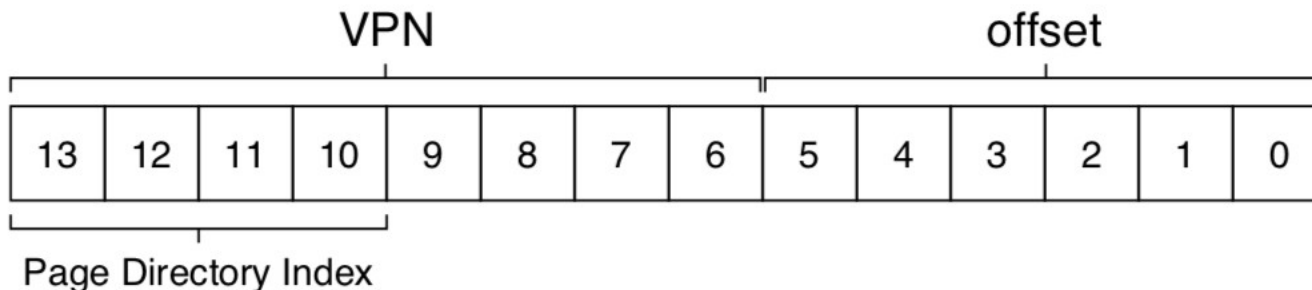
- 14-bit virtual address space
- 8-bit for VPN, 6-bit for offset
- 4-byte PTE
- A linear page table: 256 entries
- Page table: 1KB (256*4B)
- Page 0, 1 -> code
- Page 4, 5 -> heap
- Page 254, 255 -> stack

0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

Paging: Smaller Tables

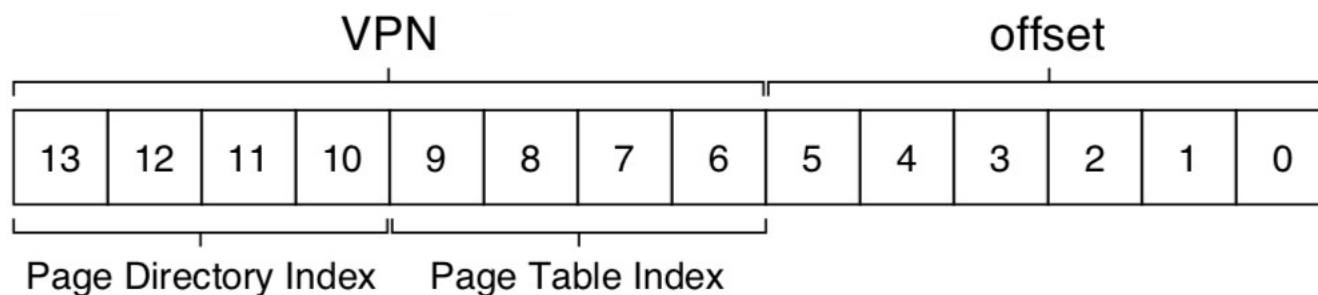
- **A Detailed Multi-Level Example**

- 一个 page 是 64B ， 能放 16 个 PTE
- 所以一级页表是 16 个 entry ， 各负责 $16(=256/16)$ 个 VPN
- Page-directory index (PDIIndex) 为 VPN 的前 4 位， 用来定位是属于一级页表中的哪个 entry



Paging: Smaller Tables

- Page table index (PTIndex)
 - VPN 的后四位，是二级页表的 index



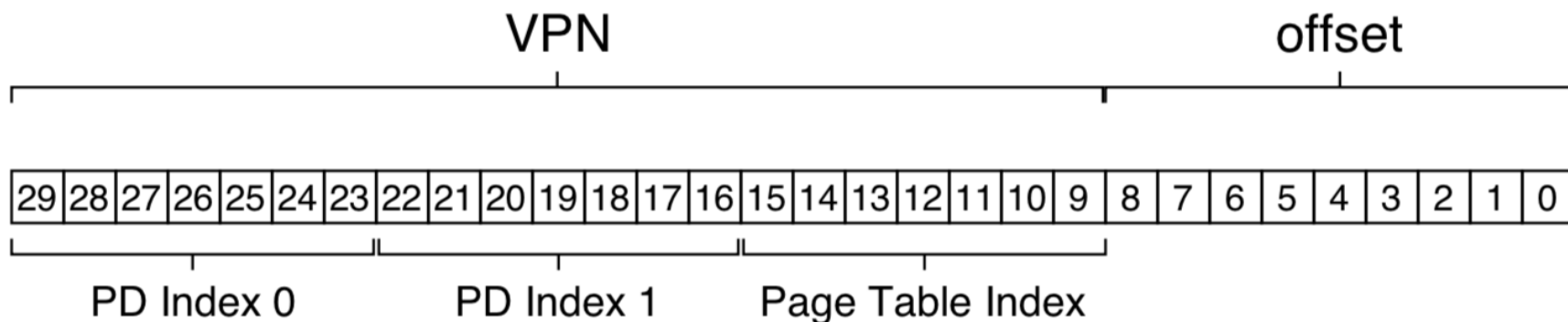
Paging: Smaller Tables

- 二级页表结果

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	-	0	—
—	0	23	1	r-x	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	80	1	rw-	-	0	—
—	0	59	1	rw-	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	-	0	—
—	0	-	0	—	55	1	rw-
101	1	-	0	—	45	1	rw-

Paging: Smaller Tables

- **More Than Two Levels**
 - 在 VPN 前面划分多个 PD Index



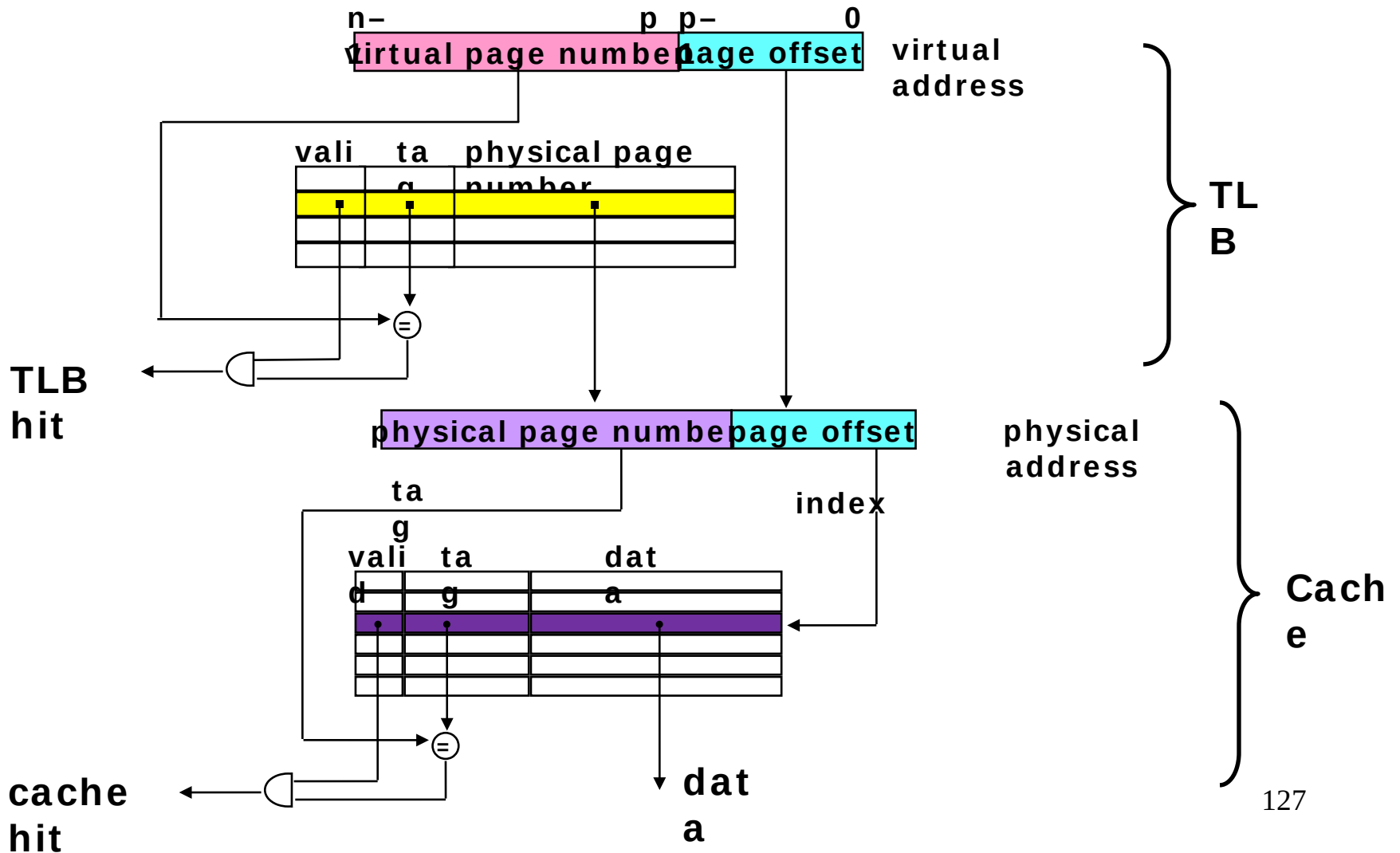
Paging: Smaller Tables

- Inverted Page Tables
 - VPN -> PPN , 比较稀疏, 多个进程 => 空间开销大
 - 但 Pa 有效, 利用率高, 比较稠密 =>
 - Inverted page tables (PPN->VPN)
 - 缺点是用户需求是 VPN->PPN 的查找
 - 线性搜索开销太大, 所以需要有额外的结构来加速搜索, 例如 hash 结构
 - e.g., IBM PowerPC
 - Page table are just data structure. You can do lots of crazy things on it.

Summary

- Programmer's view of virtual memory
 - Each process has its own **private** linear address space
 - Cannot be corrupted by other processes
- System view of virtual memory
 - Uses memory efficiently by caching virtual memory pages
 - Efficient only because of locality
 - Simplifies memory management and programming
 - Simplifies protection by providing a convenient interpositioning point to check permissions

回顾

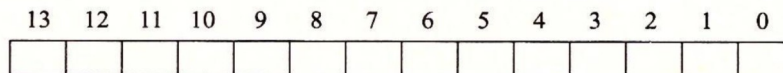


课堂练习

在下面的一系列问题中，你要展示 9.6.4 节中的示例内存系统如何将虚拟地址翻译成物理地址，以及如何访问缓存。对于给定的虚拟地址，请指出访问的 TLB 条目、物理地址，以及返回的缓存字节值。请指明是否 TLB 不命中，是否发生了缺页，是否发生了缓存不命中。如果有缓存不命中，对于“返回的缓存字节”用“-”来表示。如果有缺页，对于“PPN”用“-”来表示，而 C 部分和 D 部分就空着。

虚拟地址：0x027c

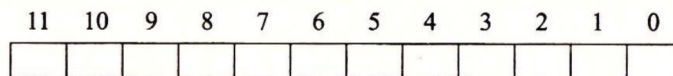
A. 虚拟地址格式



B. 地址翻译

参数	值
VPN	
TLB 索引	
TLB 标记	
TLB 命中? (是/否)	
缺页? (是/否)	
PPN	

C. 物理地址格式



D. 物理地址引用

参数	值
字节偏移	
缓存索引	
缓存标记	
缓存命中? (是/否)	
返回的缓存字节	

- Page:64B
- VA:14-bit
- PA:12-bit
- TLB: 四路组
相连, 16 items
- Cache: 直接映射,
16 个组, cacheline
4B

课堂练习

- Page:64B
- VA:14-bit (0x027c)
- PA:12-bit
- TLB: 四路组相连, 16 items
- Cache: 直接映射, 16 个组, cacheline 4B

B. 地址翻译

参数
VPN
TLB索引
TLB标记
TLB命中? (是/否)
缺页? (是/否)
PPN

D. 物理地址引用

参数
字节偏移
缓存索引
缓存标记
缓存命中? (是/否)
返回的缓存字节

位 标记位 PPN 有效位 标记位 PPN 有效位 标记位 PPN 有效位 标记位 PPN 有效位

0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

a) TLB: 四组, 16 个条目, 四路组相联

VPN PPN 有效位

00	28	1
01	-	0
02	33	1
03	02	1
04	-	0
05	16	1
06	-	0
07	-	0

VPN PPN 有效位

08	13	1
09	17	1
0A	09	1
0B	-	0
0C	-	0
0D	2D	1
0E	11	1
0F	0D	1

b) 页表: 只展示了前 16 个 PTE

索引 标记位 有效位 块0 块1 块2 块3

0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	-	-	-	-
A	2D	1	93	15	DA	3B
B	0B	0	-	-	-	-
C	12	0	-	-	-	-
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	-	-	-	-

c) 高速缓存: 16 个组, 4 字节的块, 直接映射

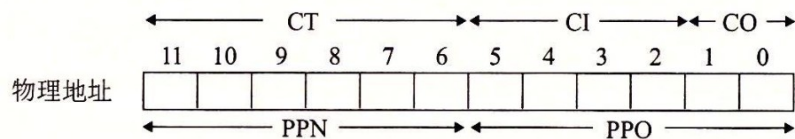
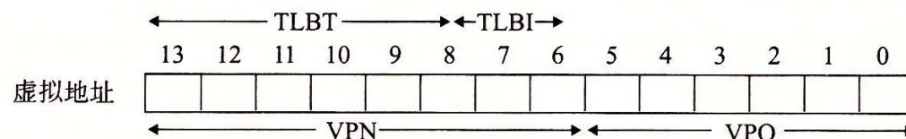
练习答案

A. 00 0010 0111 1100

B. VPN: 0x9
 TLBI: 0x1
 TLBT: 0x2
 TLB hit? N
 page fault? N
 PPN: 0x17

C. 0101 1111 1100

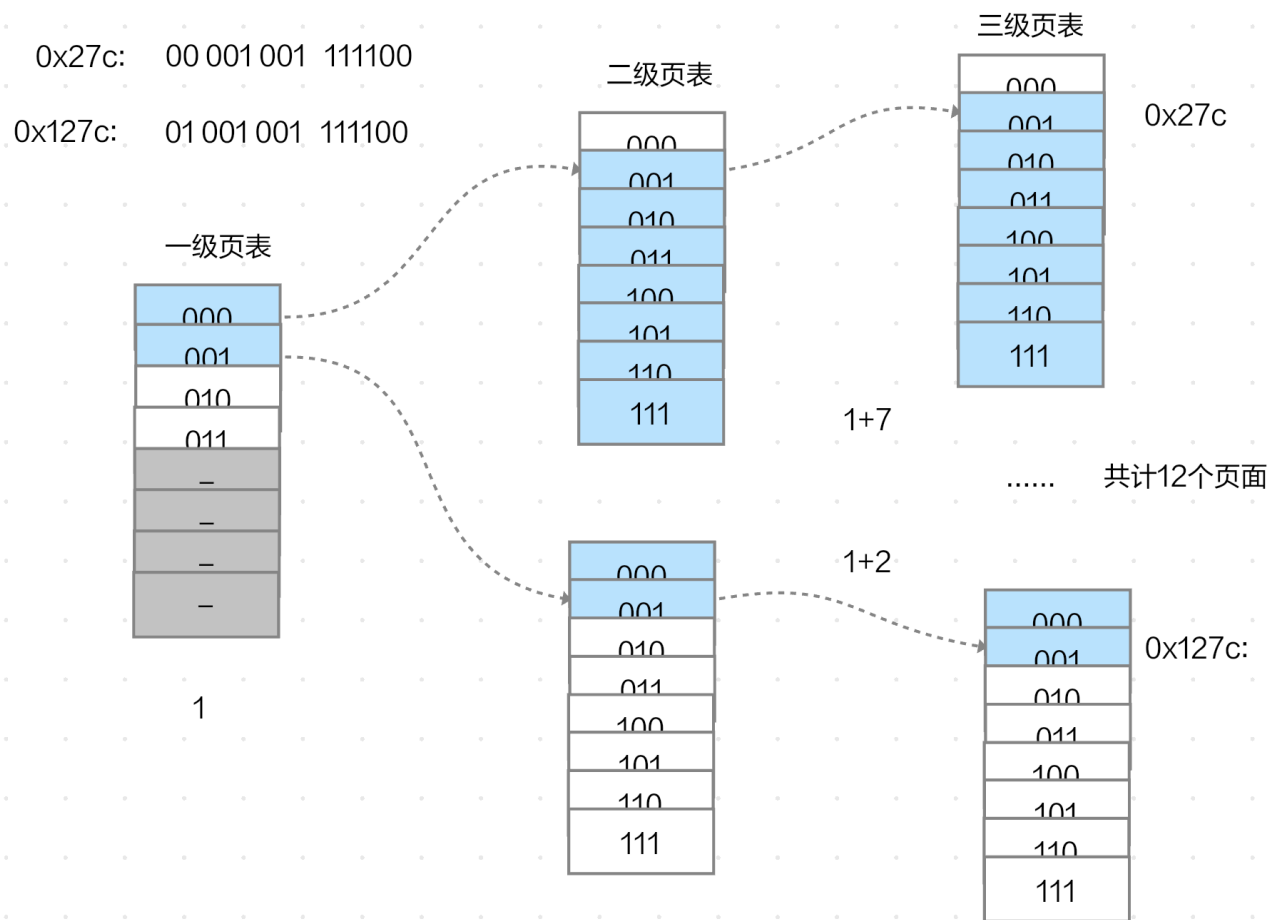
D. CO: 0x0
 CI: 0xf
 CT: 0x17
 cache hit? N
 cache byte? -



课堂练习

- Page:64B , VA:14-bit
- 假设采用多级页表节省空间, 每个页表项是 8B
- 1. 请问系统应该采用几级页表?
- 2. 地址 $0x027c \sim 0x127c$ 所用页表, 共占用多少个 Page? (如果有空余空间, 在一级页表体现)
- 3. 画出以上地址段示意图

练习答案



Homework 5

- 一周后提交